A SURVEY OF DEEP LEARNING FOR MATHEMATICIANS

TONY FENG

These are lecture notes for Math 270 as taught at UC Berkeley in Fall 2025, an idiosyncractic course aimed at an idiosyncratic audience[1] (by Machine Learning standards). The notes survey modern developments in deep learning from and for the perspective of pure mathematicians. The emphasis is on breadth rather than on depth, with each lecture giving a superficial exposure to one major topic.

Contents

Part 1. Background 2
1. Introduction to Neural Networks 2
2. Information theory 13
3. Statistical inference 25
4. Optimization 36

Part 2. Architectures 46
5. Convolutional Neural Networks 46
6. Recurrent Neural Networks 58
7. Transformers 68

Part 3. Generative models 78
8. Large Language Models 78
9. Generative Adversarial Networks 88
10. Variational Autoencoders 90
11. Diffusion Models 96

Part 4. Reinforcement learning 106
12. Value function methods 106
13. Policy optimization 106
References 106

Date: October 19, 2025.

[1]The audience is imagined to have high mathematical sophistication, but little actual familiarity with the mathematics relevant to machine learning (e.g., statistics and optimization). Topics in higher mathematics (e.g., differential geometry) will occasionally be referenced, in a shallow but possibly enlightening way.

Part 1. Background

1. Introduction to Neural Networks

1.1. **Supervised learning.** In *supervised learning*, we will consider the following basic problem: we have a set of datapoints $\mathcal{D} = \{(x_i, y_i)\}$ which we want to model via a function $y = f(x)$.

Example 1.1.1 (Regression)**.** If $y_i \in \mathbf{R}$, then we refer to the problem of modeling $y = f(x)$ as *regression*. An example might be predicting a house price from its size, number of bedrooms, location, etc.

Example 1.1.2 (Classification)**.** If y_i is among a discrete set of classes, then we refer to the problem of modeling $y = f(x)$ as *classification*. An example might be to classify a handwritten digit into one of the classes $\{0, 1, 2, \ldots, 9\}$.

A priori, it is mysterious what shape the function f should take. To give a mathematical metaphor, we need to choose an f from the "moduli space of reasonable functions", which however is some huge unknowable thing. Our approach will therefore to build a "nice chart" to probe this moduli space. Concretely, this means parametrizing a class of functions f_θ, as θ varies in a parameter space. While in principle the parameter space could be pretty general (e.g., a manifold), in practice *we want it to be* $\mathbf{R}^N$; this is what makes it "nice".

Given this setup, we will optimize θ, by defining a "loss function" $\mathcal{L}_\mathcal{D}(\theta)$ that measures the "error" of f_θ on our given data $\mathcal{D}$, and then (attempt to) minimize $\mathcal{L}_\mathcal{D}(\theta)$ via gradient descent.

Remark 1.1.3. The mapping $\theta \mapsto f_\theta$ need not be injective, and will certainly never be surjective in practical situations.

"Deep learning" refers to a specific type of chart into the space of functions, namely, the parametrization of functions by *neural networks*. These are old constructions, inspired by the structure of biological neurons, which have recently found dramatic success when scaled to extreme sizes. This first lecture will explain the basic architecture and workflow of a neural network.

1.2. **Unsupervised learning.** The adjective "supervised" in "supervised learning" refers to the fact that for the x_i in the data $\mathcal{D}$, we know the corresponding "labels" y_i which we are trying to predict. Sometimes this data can be obtained naturally (e.g., in the housing price example) but sometimes it has to be labeled manually (as in the handwriting classification example), which places a serious limitation on data availability.

There is also "unsupervised learning", whose goal is to discover patterns and structure in data without having labels. Unsupervised learning naturally divides into further subfields, depending on the precise objective. A classical example of an unsupervised learning problem is *clustering*, which finds natural ways to group data into clusters. This can be approached for example by the K-means clustering algorithm, which is not based on neural networks. In Part 3, we will learn about the more modern topic of *generative models*. Here, the data consists of samples $\mathcal{D} = \{(x_i)\}$ from an unknown probability distribution P, and the objective is to generate synthetic samples from P.

Example 1.2.1. OpenAI's GPT models are generative models that do unsupervised learning to imitate human-generated text. Here P would be the probability distribution of (say) natural language sentences, which comprise a thin slice in the space of all possible combinations of letters.

Unsupervised deep learning is assembled from the same fundamental building blocks as in supervised deep learning, so in this section we will focus on supervised learning for concreteness.

1.3. **Linear regression.** Before we talk about neural networks, we will talk about much simpler special class of parametrized functions: *linear* functions. Let's see how our paradigm plays out in this toy case.

Suppose we have a collection of datapoints $\mathcal{D} = (x_i \in \mathbf{R}^p, y_i \in \mathbf{R})_{i=1}^n$ which we want to model by a function $y = f(x)$. If we demand that f is affine linear, then it must be of the form

$$f_{W,b}(x) = xW + b$$

where we call $W \in \mathbf{R}^{p\times 1}$ and $b \in \mathbf{R}$ the *weight* and *bias*, respectively.

Remark 1.3.1. Mathematically, the bias can be simulated by a bias-free network by adding an additional node at each layer which always outputs 1, and augmenting the weight matrix to be

$$\widetilde{W} = \begin{bmatrix} W \\ b \end{bmatrix}.$$

Indeed, we have

$$\begin{bmatrix} x & 1 \end{bmatrix} \widetilde{W} = \begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} W \\ b \end{bmatrix} = xW + b.$$

This trick means that the mathematics of affine linear functions is qualitatively unaffected by whether or not we incorporate a bias, and we will exploit this at some points to simplify notation.

For simplicity of notation, we will just restrict to the case $b = 0$. If y really is given exactly by an affine-linear function of $f(x)$, and if we also have enough independent datapoints, then we can just solve for W. In practice, we would never expect to see exact linearity, so what we really want is a "best approximation" to y by a linear function, and we will decide what this means precisely. We want some way to quantify the "error" of f_W on the dataset $\mathcal{D}$; we will then take the parameters W which minimize this error. Many choices are possible; perhaps the most natural are the "Mean Absolute Error" (MAE)

$$\mathrm{MAE}_{\mathcal{D}}(W) = \frac{1}{n}\sum_{i=1}^{n} |f_W(x_i) - y_i|$$

and the "Mean Squared Error"

$$\mathrm{MSE}_{\mathcal{D}}(W) = \frac{1}{n}\sum_{i=1}^{n} (f_W(x_i) - y_i)^2.$$

We usual prefer the latter because it is more analytically convenient, and in this linear case we can even solve for the optimal W cleanly in closed form.

1.3.1. *Least squares regression.* We want to find

$$\hat{W} := \arg\min_W \mathrm{MSE}_{\mathcal{D}}(W) = \arg\min_W \sum_{i=1}^{n} (y_i - x_i W)^2.$$

Let's introduce some notation to express this in compact linear algebraic terms. Write

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \text{and} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{1.3.1}$$

for the matrices obtained by stacking the y_i (resp. x_i) vertically. Recall that each $y_i \in \mathbf{R}$ and and each $x_i \in \mathbf{R}^p$. Now we can rewrite

$$\sum (y_i - x_i W)^2 = \frac{1}{n}||Y - XW||_2^2 = \frac{1}{n}(Y - XW)^\top (Y - XW). \tag{1.3.2}$$

To find the optimal $\hat{W}$, we will differentiate (1.3.2) with respect to W.

1.3.2. *Optimal weights.* Next, to find the optimal weights, let's differentiate (1.3.2) with respect to W:

$$\begin{aligned} \frac{\partial}{\partial W} \mathrm{MSE}_{\mathcal{D}}(W, b) &= \frac{1}{n}\left[\frac{\partial}{\partial W}\Big((Y - XW)^\top\Big)(Y - XW)\right] \\ &\qquad + \frac{1}{n}\left[(Y - XW)^\top \frac{\partial}{\partial W}\Big((Y - XW)\Big)\right] \\ &= \frac{1}{n}\Big[- X^\top (Y - XW)\Big] - \Big[(Y - XW)^\top X\Big] \\ &= -2\frac{1}{n} X^\top (Y - XW). \end{aligned}$$

Hence the optimal weight $\hat{W}$ satisfies the equation

$$X^\top (Y - XW) = 0. \tag{1.3.3}$$

If $(X^\top X)$ is invertible, we can solve this for $\hat{W}$:

$$\boxed{\hat{W} = (X^\top X)^{-1}(X^\top Y).}$$

Remark 1.3.2. What would it mean for $(X^\top X)$ to be non-invertible? This happens precisely when X fails to be injective as a linear transformation. Concretely, this means that *the features (column vectors) are linearly dependent*; in particular, this certainly happens if the number of data points n is smaller than the number of features p. When this happens, many different choices of weights W lead to the same function f_W, so there clearly cannot be a unique optimum $\hat{W}$.

1.3.3. *Geometric intuition.* In retrospect, the equation (1.3.3) is geometrically obvious. Indeed, the space of all honest linear predictions on the feature values of the data is precisely the column span of X. Among such linear predictions, the "closest" one to Y is the $X\hat{W}$ which satisfies the property that $Y - X\hat{W}$ is perpendicular to the column space of X. In other words, this is precisely the condition that

$$X^\top (Y - X\hat{W}) = 0.$$

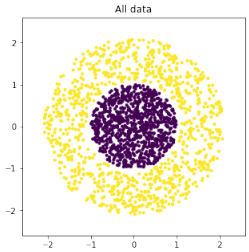


FIGURE 1.4.1. A classification problem with two classes, labeled yellow and purple. The decision boundary is meant to be a circle of radius 1 around the origin, which cannot be captured by a linear function. This figure was generated during the MLWM Week 1 Workshop Colab.

1.4. **Neural networks.** The level sets of affine linear functions are affine hyperplanes, so it is clear that some functions cannot be well approximated by affine linear functions. As a toy example, consider for example the classification problem depicted in Figure 1.4.1.

Neural networks are essentially a generalization of linear approximation where we allow f_θ to be a *piecewise linear*[2] function. It is at least intuitively clear that this class of functions is expressive enough to well approximate any reasonable f. More precisely, a neural network computes a *composition* of functions of the form

$$x \xrightarrow{g^1} z^1 \xrightarrow{h^1} a^1 \xrightarrow{g^2} z^2 \xrightarrow{h^2} a^2 \to \cdots \xrightarrow{h^{L-1}} z^{L-1} \xrightarrow{g^L} y \tag{1.4.1}$$

where each g^ℓ is an affine linear transformation and each h^ℓ is a non-linear "activation function" such as in §1.4.2.[3]

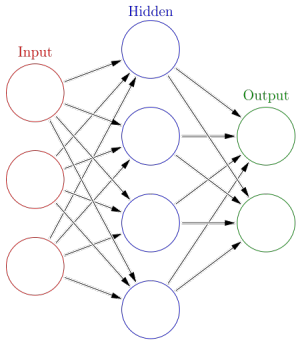
Remark 1.4.1. When there are no hidden layers, i.e., $L = 1$, then (1.4.1) collapses back down to linear approximation, which as we've seen has limited expressivity. On the other hand, there are "Universal Approximation" theorems such as [iF89], which assert that as soon as there is at least one hidden layer with any non-polynomial activation, then any continuous function of the input can be approximated to arbitrary precision on any compact domain by a function of this form.[4]

[2]This is literally true if we our activation function is ReLU, which is the common choice, but not literally true with all activation functions (e.g., sigmoid or tanh).

[3]Whether the last function involves an "activation" depends on the problem – for regression the last function g^L would typically be linear, as we wrote in (1.4.1), while for classification the last function would be a non-linearity, so (1.4.1) would be further post-composed with an h^L.

[4]As far as I know, however, such theoretical results have not been useful for predicting optimal architectures to use in practice.

1.4.1. *The anatomy of a neural network.* A neural network is divided into layers.



The ℓth layer performs the computation

$$a^{\ell-1} \xrightarrow{g^\ell} z^\ell \xrightarrow{h^\ell} a^\ell$$

Each z^ℓ and a^ℓ is a vector, whose ith component is computed by a single neuron (also known as "perceptron"). The motivation for this structure comes from biology.

During the flow of input through a neural network, neuron i receives a vector $a_i^{\ell-1}$ of input values, say of dimension $n_i^{\ell-1}$. The "vanilla" case is where the layer is *fully connected* (i.e., *dense*), in which case each neuron receives the output of every neuron in the previous layer, so $n_i^{\ell-1}$ is the number of neurons in layer $\ell-1$ (and is independent of i); we shall restrict ourselves to this case for ease of notation. The neuron itself includes the data of:

- *weights*[5] $w_i^\ell \in \mathbf{R}^{n^{\ell-1}}$, and
- a *bias* $b_i \in \mathbf{R}$.

The neuron calculates a "pre-activation value"

$$z_i^\ell = g(a^{\ell-1})_i = a^{\ell-1} w_i^\ell + b_i \in \mathbf{R}. \tag{1.4.2}$$

Finally, the neuron's "post-activation value" is

$$a_i^\ell = h_i(z_i^\ell) \tag{1.4.3}$$

where h_i is a non-linear "activation" function, of which some examples are given in §1.4.2.

Terminology 1.4.2. The *depth* of the neural network is the number of layers (denoted L in (1.4.1)) and the *width* of a layer is its number of neurons (denoted n^ℓ above).

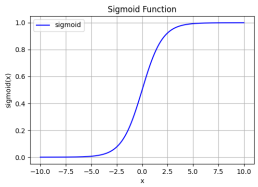
The parameters which are updated during training are the weight and bias parameters $w^\ell = (w_i^\ell)_{i=1}^{n^\ell}$ and $b^\ell = (b_i)_{i=1}^{n^\ell}$. These are called "learnable parameters" to distinguish them from *hyperparameters* such as width and depth, which are chosen in advance rather than learned during training.

1.4.2. *Activation functions.* The activation functions $h^1, h^2, \ldots, h^L$ are non-linear, and are the only source of non-linearity in the neural network. They are given entrywise by applying the same non-linear function to each entry, except possibly for h^L which reparametrizes the output to the desired form.

Example 1.4.3. The *sigmoid* function is

[5]We caution that our convention for weight matrices is the transpose of PyTorch's convention, which uses instead $z = aW^\perp$.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

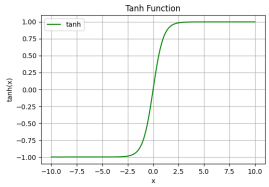


Observe that $\sigma(x)$ is monotonically increasing and valued in $(0, 1)$. Note that

$$\sigma(x) = \frac{1}{1+e^{-x}} = 1 - \sigma(-x).$$

Example 1.4.4. The hyperbolic tangent function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

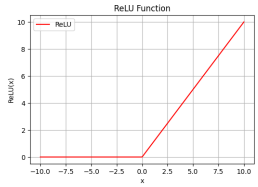


is another candidate activation function, which has similar properties to the sigmoid function.

Both σ and tanh are examples of *saturating* non-linearity, meaning that they quickly level off as $|x|$ gets large. Consequently, if $|x|$ is very large, then $|\sigma'(x)|$ is very small, which means that gradient descent can be very slow in such domains. This problem is exacerbated when composing many such functions together, leading to the "vanishing gradients" problem for deep neural networks.

Example 1.4.5. The *ReLU* (for "rectified linear unit") function is

$$\mathrm{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$



Compared to the sigmoid function, ReLU is considered non-saturating, at least in the positive direction. (For a discussion of why saturation in the negative direction is mostly innocuous, see Example 1.6.5.) There are variants of ReLU (like "leaky ReLU") which addresses the left saturation, but the simple ReLU function has come to be the dominant choice for non-linearities in deep neural networks, because it works well in practice and scales well due to its simplicity.

Terminology 1.4.6. In linear regression, we called the entries of the input vectors "features". Since every layer of a neural network looks like linear regression, we continue to refer to the values calculated by intermediate neurons as "features", even though in practice they can be hard to interpret concretely. (Other names for features in the hidden layers are "representations" or "embeddings".)

In deep networks, one can observe a hierarchical structure to features: in early layers, the features are simple combinations of the input features, while later neurons track more complex and subtle structure. Illustrative examples will be seen in the lectures on CNNs and RNNs.

Remark 1.4.7. Activation functions are trying to capture the intuition from biology that neurons should "activate" when they recognize a feature. Consider for concreteness the ReLU function $a(x) = \mathrm{ReLU}(zw + b)$. If $zw + b < 0$ then the neuron transmits no signal and we interpret it as being "off"; otherwise it transmits the signal $zw + b$.

1.5. **Learning.** Returning to the template of §1.1, suppose we are given a training data set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ for which we want to model $y_i = f_\theta(x_i)$. We take θ to consist of all the weights W and biases b of a neural network which computes f_θ.

The point of Machine Learning is to find the *optimal* $\hat{\theta}$ from the data. How do we do this? As we discussed in the toy example of Linear Regression (§1.3), we first need to settle on a precise notion of optimality, by defining a measure of "error" on the data. This is called the *loss function*. We will then try to find the θ that minimizes the loss function.

1.5.1. *Loss function.* For regression, the most common loss function is again the "mean squared error"

$$\mathcal{L}_D(\theta) = \mathrm{MSE}_{\mathcal{D}}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2.$$

For classification problems, this does not apply in general since our outputs y_i and $f_\theta(x_i)$ are class labels instead of numbers. Even when the y_i can be viewed as numbers, as in the handwritten digit classification example, we probably do not want to think of them as such because the numerical value does not necessarily reflect the notion of similarity that we are pursuing. For example, for the purposes of handwriting classification, '3' and '8' are close in terms of visual appearance, but they are relatively far apart numerically. On the other hand, '3' and '4' are close numerically but not that close visually.

Instead for classification problems we *refine* the objective to something more quantitative: a *probability distribution on the possible classes*. Then our neural network will actually compute

$$F_\theta(x, y) = (\text{predicted probability of assigning class } y \text{ to } x).$$

From F_θ, we can easily extract

$$f_\theta(x_i) = \arg\max_y F_\theta(x_i, y).$$

Now how do we measure the error on a given datapoint (x_i, y_i)? A common choice is the "cross-entropy"

$$\mathcal{L}_{(x_i,y_i)}(\theta) = -\ln F_\theta(x_i, y_i); \tag{1.5.1}$$

we will discuss the mathematical justification for this in the lectures on Information Theory and Statistical Inference. As a sanity check, note that $\mathcal{L}_{(x_i,y_i)}(\theta) = 0$ if $F_\theta(x_i, -)$ is concentrated on the correct label y_i, and increases to ∞ as $F_\theta(x_i, -)$ decreases to 0.

Remark 1.5.1. To ensure that $F_\theta(x, -)$ is a probability distribution, we need

$$\sum_y F_\theta(x, y) = 1. \tag{1.5.2}$$

While it is clear that this can be enforced mathematically by imposing some constraint on $\theta \in \mathbf{R}^N$, this is awkward to do explicitly. Instead, it is better to reparametrize so that $\theta \in \mathbf{R}^N$ remains unconstrained. For example, we can create a probability distribution out of n arbitrary real numbers $(u_1, \ldots, u_n)$ by applying the "softmax" function

$$(u_i) \mapsto \Big(\frac{e^{u_i}}{\sum_{k=1}^n e^{u_k}}\Big).$$

Thus, we enforce (1.5.2) by postcomposing with softmax in the last layer of the NN.

1.5.2. *Gradient descent.* The training objective is to find θ minimizing the loss function $\mathcal{L}_{\mathcal{D}}\colon \mathbf{R}^N \to \mathbf{R}$. We do this by sampling an initial point θ_0, and then iteratively adjusting it by *gradient descent*.

The Taylor approximation of a function $\mathcal{L}$ is

$$\mathcal{L}(\theta+\xi) \approx \mathcal{L}(\theta) + \mathrm{d}\mathcal{L}(\xi) \approx \mathcal{L}(\theta) + \langle \nabla_\theta(\mathcal{L}), \xi\rangle.$$

By Cauchy–Schwarz, the inner product is maximized among unit vectors ξ when ξ is proportional to a positive multiple $\nabla_\theta(\mathcal{L})$. This makes it clear that, to first order, $-\nabla_\theta(\mathcal{L})$ is the direction of "steepest descent" for $\mathcal{L}$. Therefore, at each training step, we calculate

$$\nabla_\theta(\mathcal{L}_{\mathcal{D}}) = \frac{1}{n}\sum_{i=1}^{n}(\nabla_\theta \mathcal{L}_{(x_i,y_i)}),$$

and then make an update of the form

$$\theta \mapsto \theta - \eta\nabla_\theta(\mathcal{L}_{\mathcal{D}}). \tag{1.5.3}$$

(In computer science notation, this translates to "$\theta \leftarrow \theta - \eta\nabla_\theta(\mathcal{L}_{\mathcal{D}})$"). The insistence on parameter space $\mathbf{R}^N$ is to ensure that the update (1.5.3) stays in the parameter space.

Remark 1.5.2. Actually, in practice we would not use the entire dataset for (1.5.3), as this would be too slow to compute, especially for large datasets. Since the loss is additive over datapoints, we can obtain an unbiased estimator for the gradient (up to scalar multiple) by picking a random datapoint (x_i, y_i) and using $\nabla_\theta(\mathcal{L}_{(x_i,y_i)})$ instead of $\nabla_\theta(\mathcal{L}_{\mathcal{D}})$. This is fast, but it has the downside of being very noisy. In practice, we would do something in between: pick a random batch $\mathcal{B} \subset \mathcal{D}$, and average the gradient over that batch.

The hyperparameter η in (1.5.3) is called the *learning rate*. Note that the learning rate is *not* a unitless quantity, which means that its numerical value is not intrinsically meaningful, and in practice it needs to be tuned consistently with the normalizations of the input and output.

Example 1.5.3. A common way to normalize the input is by the "Z-score", which is the number of standard deviations from mean (calculated empirically). However, this is really only appropriate for data which is approximately *normally distributed* data. Sometimes we instead linearly scale to a given desired range such as $[-1, 1]$, or apply one of these scaling functions after some transformation (such as the logarithm function), depending on the data. With Z-score normalization, it is common to see a learning rate around 10^{-2}.

The intuition is that a learning rate which is too large will oscillate too much and not be able to take good steps, while a learning rate which is too small will converge too slowly. In practice, sophisticated neural networks will use a learning rate scheduler that adjusts the learning rate during training, for example by a formula (linear, exponential, or power law decay), or adaptively (e.g., starting at 0.1 and decreasing multiplicatively by a factor of 0.1 when the validation loss flattens sufficiently).

1.5.3. *Train/validation/test split.* When training a neural network in practice, the data is split randomly into "training data" (around 80%), "validation data" (around 10%), and "test data" (around 10%). The training data is used for gradient descent on the model weights, and the validation data is used for things like model selection, hyperparameter tuning, and early stopping (a regularization method). Finally, the test data is used only at the end, to

test that the neural network can generalize to unseen examples, as opposed to "memorizing" the training examples.

A training *epoch* is one full cycle of passing over the training data. During each epoch, the training data is broken up into batches of B samples, and the gradient update is averaged over the batch. This averaging addresses the noisiness of individual datapoints. For a typical (small to medium) neural network, B is typically on the order of 100 to 1000. But for a large language model, a pre-training batch size would be much larger, e.g., on the order of 10^6 for GPT-3.

The number of epochs depends on the size and complexity of the network, but is usually on the order of 1-100. Large Language Models (LLMs) are an exception: they do not pass (during the pre-training phase) multiple times through the training data[6] (in fact, the training phase involves deduplication on data, so that not even the whole training set is used). This is because the training data is so large to begin with that there has (until now) been enough fresh data, and LLMs have so many parameters that they could overfit to data.

1.6. **Backpropagation.** The key to the neural network architecture is an *efficient* algorithm to implement gradient descent, called *backpropagation*. To describe it, we begin by setting up the notation.

1.6.1. *Setup.* We assume a neural network of the form (1.4.1),

$$x \xrightarrow{g^1} z^1 \xrightarrow{h^1} a^1 \xrightarrow{g^2} z^2 \xrightarrow{h^2} a^2 \to \cdots \xrightarrow{h^{L-1}} a^{L-1} \xrightarrow{g^L} z^L.$$

For each $\ell = 1, 2, \ldots, L$, let n^ℓ be the number of neurons in the ℓth layer and $h^\ell \colon \mathbf{R}^{n^\ell} \to \mathbf{R}^{n^\ell}$ be the activation function at the ℓth layer. The function g^ℓ is given by

$$z^\ell = g^\ell(a^{\ell-1}) = a^{\ell-1}W^\ell \tag{1.6.1}$$

where W^ℓ is a matrix of dimension $n^{\ell-1} \times n^\ell$.

The loss on input (x, y) is

$$\mathcal{L}(y, z^L) = \mathcal{L}(y, (h^{L-2}(\ldots h^1(z^1)W^1)W^2\ldots)W^L). \tag{1.6.2}$$

Remark 1.6.1. Here we assume that each layer is fully connected; any other case is mathematically specialized from this one by forcing some weights to be zero. We also assume that there are no biases, which again is mathematically fully general by Remark 1.3.1.

1.6.2. *Loss gradient with respect to weights.* In the update steps, we change the parameter W^L by some multiple (called the *learning rate*) of the gradient $-\frac{\partial \mathcal{L}}{\partial W^L}$. Backpropagation is a way to calculate this efficiently using the chain rule. Note that $\mathcal{L}$ only depends on W^ℓ through z^ℓ, so we have

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \frac{\partial \mathcal{L}}{\partial z^\ell} \circ \frac{\partial z^\ell}{\partial W^\ell}. \tag{1.6.3}$$

From (1.6.1), we see that $\frac{\partial z^\ell}{\partial W^\ell}$ is left multiplication by $a^{\ell-1}$. Therefore,

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}_{n^\ell \times 1} \circ \underbrace{a^{\ell-1}}_{1 \times n^{\ell-1}} \tag{1.6.4}$$

Therefore, the gradient is the transpose of (1.6.4),

$$\boxed{\nabla_{W^\ell}(\mathcal{L}) = (a^{\ell-1})^\top \nabla_{z^\ell}(\mathcal{L}).} \tag{1.6.5}$$

[6]As we will see in §8, this is not literally true: for GPT-3, the training data is stratified by quality, and higher quality data does get seen more than once during training

Note that this is an $n^{\ell-1} \times n^\ell$ matrix. The vector $a^{\ell-1}$ is stored in memory on the forward pass through the network, so that it can be used here to compute $\nabla_{W^\ell}(\mathcal{L})$.

1.6.3. *Digression on the gradient.* In multivariable calculus, one usually sees the gradient of a function $f\colon \mathbf{R}^N \to \mathbf{R}$ defined as the "transpose of the derivative". Because of our convention that z^ℓ and a^ℓ are row vectors, the gradient would then be a $N \times 1$ "column vector". However, the convention is that the gradient of a scalar-valued function $\mathcal{L}$ with respect to a *tensor* W has the same shape as W. This is desirable because $\nabla_W(\mathcal{L})$ then lies in the same ambient space as W, and it is then easy to interpret the $\nabla_W(\mathcal{L})$ as a direction of "steepest descent" for W.

In particular, if W is an $m \times n$ matrix of variables, then by $\nabla_W(\mathcal{L})$ we mean the $m \times n$ matrix such that

$$\nabla_W(\mathcal{L})_{ij} = \frac{\partial \mathcal{L}}{\partial W_{ij}}. \tag{1.6.6}$$

However, because we are mathematicians we prefer to formulate the notion of gradient in a coordinate-free way. The correct general context is that of a Riemannian manifold $\mathcal{M}$. This means that $\mathcal{M}$ is equipped with a Riemannian metric, which is a "smoothly varying" choice of inner product on each each tangent space; we won't use anything other than these individual inner products, so we will not be more precise.

Example 1.6.2. We will describe a canonical Riemannian metric on $\mathcal{M} = \mathrm{Hom}(\mathbf{R}^n, \mathbf{R}^m)$. Since $\mathcal{M}$ is a vector space, its tangent spaces at all points $p \in \mathcal{M}$ are canonically identified with $\mathrm{Hom}(\mathbf{R}^n, \mathbf{R}^m)$, which can then be identified back with $\mathcal{M}$ itself. On this space, we have an inner product

$$\langle A, B \rangle := \mathrm{Tr}(A^\top B); \tag{1.6.7}$$

this is positive-definite (i.e., $\langle A, A \rangle \geq 0$ with equality if and only if $A = 0$), since $A^\top A$ itself defines a positive-definite form on $\mathbf{R}^n$.

Remark 1.6.3. Recall some elementary linear algebra: suppose V is any finite-dimensional $\mathbf{R}$-vector space equipped with a non-degenerate bilinear pairing. Then any linear functional $f\colon V \to \mathbf{R}$ is uniquely represented as pairing with an element of V, namely the pre-image of f under the isomorphism $V \xrightarrow{\sim} V^*$ coming from the self-duality.

Let $\mathcal{M}$ be a Riemannian manifold and $f\colon \mathcal{M} \to \mathbf{R}$ a smooth function. Then the derivative of f at $p \in \mathcal{M}$ is a linear transformation

$$\mathrm{d}f_p\colon \mathbf{T}_p\mathcal{M} \to \mathbf{T}_{f(p)}\mathbf{R} = \mathbf{R}.$$

The map $\mathrm{d}f_p$ is a synonym for $\frac{\partial f}{\partial p}$.

Definition 1.6.4. The *gradient* $\nabla_p f$ of $f\colon \mathcal{M} \to \mathbf{R}$ at $p \in \mathcal{M}$ is the element of $\mathbf{T}_p\mathcal{M}$ that represents the linear functional $\mathrm{d}f_p$ in the sense of Remark 1.6.3. In other words, it is characterized by the property that

$$\langle \nabla f_p, \xi \rangle = \mathrm{d}f_p(\xi) \text{ for all } \xi \in \mathbf{T}_p(\mathcal{M}).$$

This definition clarifies that the derivative $\mathrm{d}f_p$ is a linear functional on the tangent space, while the gradient is an actual element of the tangent space.

1.6.4. *Loss gradient with respect to features.* In (1.6.5), we saw how to calculate $\nabla_{W^\ell}(\mathcal{L})$ in terms of $\nabla_{z^\ell}(\mathcal{L})$. Since $\mathcal{L}$ only depends on z^ℓ through $z^{\ell+1}$, we can calculate this recursively using the chain rule.

$$\frac{\partial \mathcal{L}}{\partial z^\ell} = \frac{\partial \mathcal{L}}{\partial z^{\ell+1}} \frac{\partial z^{\ell+1}}{\partial a^\ell} \frac{\partial a^\ell}{\partial z^\ell}. \tag{1.6.8}$$

Note that $\partial a^\ell / \partial z^\ell$ has the form $\operatorname{diag}(\mathrm{d}h^\ell(z^\ell))$, a diagonal (in particular, symmetric) $n_i \times n_i$ matrix. Since $z^{\ell+1} = a^\ell W^{\ell+1}$, $\partial z^{\ell+1} / \partial a^\ell$ is right multiplication by $W^{\ell+1}$. However, these expressions cannot be inserted mindlessly into (1.6.8) as the matrix multiplication

$$\underbrace{\frac{\partial \mathcal{L}}{\partial z^{\ell+1}}}_{n^{\ell+1}\times 1} \underbrace{W^{\ell+1}}_{n^\ell \times n^{\ell+1}} \underbrace{\frac{\partial a^\ell}{\partial z^\ell}}_{n^\ell \times n^\ell}$$

does not make sense. When confused about derivatives of tensor-valued functions of tensors, one can lean back on the formalism of §1.6.3. First, for a tangent vector $\mathrm{d}z^\ell$ to z^ℓ,

$$\frac{\partial a^\ell}{\partial z^\ell}(\mathrm{d}z^\ell) = \mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)$$

where $\odot$ is the *Hadamard product* (meaning that multiplication is calculated entrywise) and for ReLU, we have[7]

$$\mathrm{d}h^\ell(z^\ell)_i = \begin{cases} 1 & z_i^\ell \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Next, we apply

$$\frac{\partial z^{\ell+1}}{\partial a^\ell}\Big(\mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)\Big) = \Big(\mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)\Big) W^{\ell+1}$$

Using cyclicity of the trace, we find that

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z^\ell} &= \frac{\partial \mathcal{L}}{\partial z^{\ell+1}}\Big(\Big(\mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)\Big) W^{\ell+1}\Big) \\
&= \Big\langle \nabla_{z^{\ell+1}}(\mathcal{L}), \Big(\mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)\Big) W^{\ell+1} \Big\rangle \\
&= \operatorname{Tr}\Big(\nabla_{z^{\ell+1}}(\mathcal{L})^\top \Big(\mathrm{d}z^\ell \odot \mathrm{d}h^\ell(z^\ell)\Big) W^{\ell+1}\Big) \\
&= \operatorname{Tr}\Big(\Big[\mathrm{d}h^\ell(\mathrm{d}z^\ell) \odot \Big(W^{\ell+1}\nabla_{z^{\ell+1}}(\mathcal{L})^\top\Big)\Big]\mathrm{d}z^\ell\Big) \\
&= \Big\langle \Big[\nabla_{z^{\ell+1}}(\mathcal{L})(W^{\ell+1})\Big]^\top \odot \mathrm{d}h^\ell(\mathrm{d}z^\ell), \mathrm{d}z^\ell \Big\rangle
\end{aligned}$$

which implies that

$$\boxed{\nabla_{z^\ell}(\mathcal{L}) = \Big[\nabla_{z^{\ell+1}}(\mathcal{L})(W^{\ell+1})^\top\Big] \odot \mathrm{d}h^\ell(\mathrm{d}z^\ell).} \tag{1.6.9}$$

Note that this recursive formula can be computed from left to right, in which case it only ever involves multiplying a matrix times a *vector*, which saves a dimension compared to matrix multiplication.

Example 1.6.5. Consider a NN with ReLU activations. Since $\mathrm{ReLU}'(x) = 0$ if $x < 0$, a neuron which is off has a local gradient $\mathrm{d}h^\ell(z^\ell)_i = 0$, so its weights will not update during backpropagation. Why is this reasonable?

Let's tell a story for concreteness: imagine we have a Neural Network that tries to classify mathematical proofs as Correct or Incorrect. Our particular neuron might be trying to detect

[7] ignoring the case $z_i^\ell = 0$ at which h^ℓ is non-differentiable

if the proof is using the method of "handwaving". Thus the inputs z might be tracking words in the proof, and then zw picks out (say) the phrases "obvious", "clear", "well-known", "easy to see", or "left to the reader".

Suppose these phrases don't appear in a given input, so the neuron in question does not activate during inference. If the end prediction is wrong, then it shouldn't cause us to update the weights of the neuron itself, since the prediction didn't use that feature anyway.

Example 1.6.6. For completeness, we analyze the base case $\nabla_{z^L}(\mathcal{L})$. This depends on the exact form of the loss function.

For regression, it is typical to take the MSE loss

$$\mathcal{L}(y, z^L) = ||z^L - y||_2^2,$$

so that $\delta^L = \nabla_{z^L}(\mathcal{L}) = z^L - y$.

For classification, recall that from z^L we obtain an estimated probability distribution $\hat{p} = \text{SoftMax}(z^L)$, and then take the cross-entropy loss. Therefore,

$$\mathcal{L}(y, z^L) = -\sum_i y_i \ln\Big(\frac{e^{z_i^L}}{\sum_k e^{z_k^L}}\Big) = -\sum_i y_i z_i^L + \ln \sum_k e^{z_k^L}.$$

Some algebra reveals that the gradient of $\ln \sum_k e^{z_k^L}$ is simply $\hat{p}$, so that

$$\nabla_{z^L}(\mathcal{L}) = \hat{p} - y.$$

1.6.5. *Backpropagation algorithm.* Putting everything together, we arrive at the **Backpropagation Algorithm.**

(1) Recursively compute $\nabla_{z^L}(\mathcal{L}), \nabla_{z^{L-1}}(\mathcal{L}), \ldots, \nabla_{z^1}(\mathcal{L})$ according (1.6.9) and the following discussion.
(2) Compute $\nabla_{W^\ell}(\mathcal{L}) = (a^{\ell-1})^\top \nabla_{z^\ell}(\mathcal{L})$ for $\ell = L, L-1, \ldots, 1$.
(3) Update $W^\ell \leftarrow W^\ell - \eta \nabla_{W^\ell}(\mathcal{L})$, where η is the learning rate.

1.7. **Regularization.** A perennial theme in Machine Learning is the problem of *overfitting*, which refers to when the model adapts too specifically to the training data, and is then unable to generalize to new examples.

Generally speaking, overfitting occurs when the training data is too small relative to size of the model (measured by its number of parameters). Ideally, one would like to increase training data, but this may not be feasible. For fixed training data, there are measures called *regularization* in order to combat overfitting. Some examples include:

- Introduce an additional term into the loss objective to "penalize" complexity, such as adding $+0.01||W||$ for some norm on the vector of weights W (usually the L^2 norm or the L^1 norm).
- "Early stopping": stop training when the validation loss levels off, even if the training loss is improving.
- "Dropout": randomly drop a proportion of the neurons during training.

We will discuss these further in §4.1 below. In many ways, the description of neural network training given in this first lecture is oversimplified, and will be corrected in §4.

2. Information theory

We will summarize some ideas from information theory, which began with Claude Shannon as a "mathematical theory of communication" [Sha48]. Historically, information theory has had important applications to coding and compression, which we will touch on. It is

not clear a priori why this has anything to do with machine learning, but ultimately we will see a connection between compression and generative sequence modeling, which helps to explain why the concepts of information theory play an important role in (for example) generative AI.

Convention: we use ln for the natural logarithm and $\log = \log_2$ for the base 2 logarithm.

2.1. **Information.** Suppose we want to capture the concept of "information" mathematically. For concreteness, let's begin by thinking about the amount of information conveyed in a written message. Naively, you might first try to measure information by the length of the message, but it is clear that this fails to capture important features of information.

Example 2.1.1. The English language has a lot of redundancy. Consider the following MathOverflow comment[8]:

> Paper I considers surface fibered in geom. integral curves over proj. line, generic fiber smooth and at worst nodal fibers; subsumed by Grothendieck's relative Picard schemes (for any proj. flat family of geom. int. schemes), really its relative id. component (makes sense since Pic smooth for relative curves); see FGA Explained. II is special case of description of ℓ-adic repns for ab. variety over Frac(dvr) when Neron model has sst reduction, III grinds out n-torsion Galois repn for ''generic'' ell. curve over $k(j)$; II & III subsumed by Neron models (didn't exist!).

This is a heavily compressed message, yet it can (to a reader educated in algebraic geometry) be understood and completely uncompressed.

Slogan 2.1.2. Shannon's key idea is that the information in a message can be measured by how *surprising* the message is.

In turn, we can think of "surprise" as a synonym for "unlikely". To quantify this precisely, we work in the context of a probability ensemble (Ω, P). In this notation, Ω is our *outcome space* (sometimes called *sample space*), and P is the probability function on it. For simplicity, we focus our discussion on the case of *discrete* probability distributions; the reader should be able to easily extrapolate to the continuous case.

Definition 2.1.3. Let $A \subset \Omega$ be an event. The *Shannon information* of A is

$$I(A) := \log(1/P(A)) = -\log P(A). \tag{2.1.1}$$

Note that $I(A) \geq 0$, as any event carries non-negative information. The more improbable the event, the higher the information it carries.

Example 2.1.4. The probability that we have the same birthday (let's ignore the issue of leap years) is $1/365$, which is much smaller than the probability $364/365$ that we do not have the same birthday. Indeed, if I tell you "We do not have the same birthday", then you learn little information about my birthday. But if I tell you "We have the same birthday", then you learn a lot about my birthday.

Example 2.1.5. Suppose you are reading a message that starts `Harry Potte`. Then you know the last character with extremely high probability, so when the `r` comes, it conveys very little additional information.

[8]Brian Conrad, MathOverflow comment.

Remark 2.1.6. The natural unit of information is *bits*. Indeed, information is a measure of *how many bits* are required to convey a message. Consider for example a uniform distribution on n elements. Since n elements can be enumerated using $\log(n)$ bits, to specify a sample from this distribution requires $\log(n)$ bits.

2.2. **Entropy.** Shannon goes on in [Sha48] to define the notion of *entropy*. This is closely related to the concept of "Boltzmann–Gibbs energy" in statistical physics, and it appears that Shannon was aware of this notion, and inspired by it.

Slogan 2.2.1. The (Shannon) entropy of a random system is the *expected amount of information* in it.

For example, the entropy of a discrete probability ensemble (Ω, P) is

$$H(\Omega) = \mathbb{E}[\log(1/P)] = \sum_{\omega\in\Omega} P(\omega)\log(1/P(\omega)) = -\sum_{\omega\in\Omega} P(\omega)\log P(\omega). \tag{2.2.1}$$

More generally, we define the entropy of a random variable.

Definition 2.2.2. Let $X\colon \Omega \to \mathbf{R}$ be a random variable. Then the *entropy* of X is

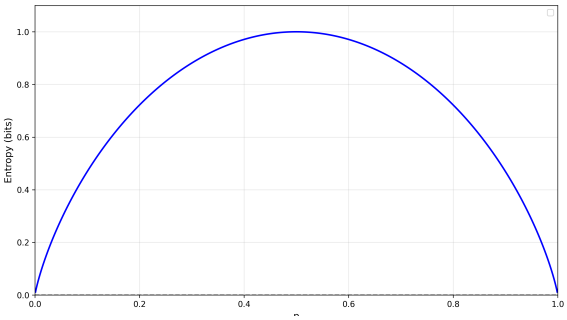
$$H(X) := \mathbb{E}[-\log P(X)] = -\sum_x P(X = x)\log P(X = x).$$

Example 2.2.3. The entropy of an English letter is estimated to be roughly 4.11 bits, because of the non-uniform distribution of letters. On the other hand, the entropy of the uniform distribution on 26 elements is $\log(26) \approx 4.7$.

Example 2.2.4. Let X be a Bernoulli random variable with parameter p. Then

$$H(X) = -p\log p - (1-p)\log(1-p),$$

which is graphed below as a function of p.



We can think of X as the outcome of a biased coin flip which comes up heads with probability p. For $p = 1/2$, we see that $H(X) = 1$, which makes sense because then X carries the information of exactly one bit. As $p \to 0$ or $p \to 1$ the entropy decreases to 0, because X approaches a deterministic value, hence carries less and less information.

Exercise 2.2.5. Show that if X and Y are independent random variables, then the entropy of the joint distribution (X, Y) satisfies

$$H(X, Y) = H(X) + H(Y).$$

Remark 2.2.6 (Perplexity)**.** A mathematically equivalent notion to entropy is *perplexity* (after which Perplexity AI is named). The *perplexity* of a random variable X is simply $2^{H(X)}$. While entropy is measured by bits, perplexity captures the number of possibilities, e.g., the perplexity of a uniform random variable on N elements is N.

2.3. **Why this formula for entropy?** We will see that entropy is a key quantity which shows up in many applications. However, the formula (2.2.1) is perhaps unintuitive at first. How could we have discovered it?

First let's think about the definition of information, (2.1.1). Why does the logarithm come up there? The key observation is that the information conveyed by the message X_1, X_2 should be the same as the information conveyed by sending X_1, followed by sending X_2. Thus, once we accept that information $I(\cdot)$ is a function of probability, it should satisfy

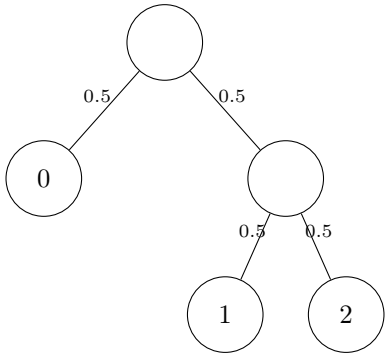
$$I(P(X_1, X_2)) = I(P(X_1)) + I(P(X_2|X_1)).$$

Given that $P(X_1, X_2) = P(X_2|X_1)P(X_1)$, it is natural to take I to be the logarithm function.

Example 2.3.1. Suppose a discrete random variable X takes values $0, 1, 2$ with $P(X = 0) = 1/2$ and $P(X = 1) = P(X = 2) = 0.25$. Then the entropy of X is

$$H(X) = 0.5\log(2) + 0.25\log(4) + 0.25\log(4) = 1.5.$$

Now let's imagine that X arises via a different "origin story", and verify that we get the same answer. Say we flip a fair coin, and if it comes up H, then $X = 0$. Otherwise, we flip again, and if the second flip is H then we set $X = 1$, otherwise $X = 2$.



Then X is a coin flip plus a 0.5 chance of another coin flip, so $H(X) = 1 + 0.5 \times 1 = 1.5$.

This example illustrates the "decomposability" of the entropy: if we "decompose" a random variable into sequences of random variables, then the entropy adds.

More generally, suppose that we have a discrete probability distribution $\{p_1, \ldots, p_n\}$. Then the entropy of the distribution should have the form $H(p_1, \ldots, p_n)$, and the generalization of the "decomposibility" property is expressed as

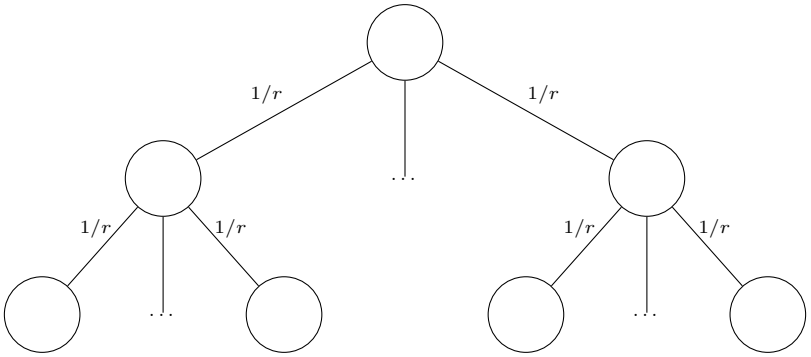
$$H(p_1, \ldots, p_n) = H(p_1, 1 - p_1) + (1 - p_1)H(\frac{p_2}{1 - p_1}, \ldots, \frac{p_n}{1 - p_1}) \tag{2.3.1}$$

Proposition 2.3.2 (Khinchin)**.** *There is a unique up to scalar function $H(\underline{p})$ over finite probability distributions $\underline{p} = \{p_1, \ldots, p_n\}$, satisfying the following properties.*

1. *(Continuity) For any fixed n, H is continuous in the p_i.*
2. *(Increasing in the size of the uniform distribution) $H(\frac{1}{n}, \ldots, \frac{1}{n})$ is increasing as a function of n.*
3. *(Decomposability) H satisfies the identity* (2.3.1)*.*

The function $H(\underline{p})$ is given by (a scalar multiple of) the Shannon entropy $-\sum p_i \log(p_i)$.

Proof. Let us first focus on the uniform distribution, and abbreviate $H_r := H(\frac{1}{r}, \dots, \frac{1}{r})$ for any $r \geq 1$. Then by decomposability of the entropy, we have $H_{r^n} = nH_r$.



We will show that $H_r \propto \log r$. For any $n \geq 1$, we can find $m \geq 1$ such that $2^m < r^n < 2^{m+1}$, so that

$$m < n \log(r) < m + 1$$

or

$$\frac{m}{n} < \log(r) < \frac{m}{n} + \frac{1}{n}.$$

By monotonicity, we have

$$mH_2 < nH_r < (m+1)H_2$$

hence

$$\frac{m}{n} H_2 < H_r < \frac{m+1}{n} H_2 \implies \frac{m}{n} \leq \frac{H_r}{H_2} < \frac{m+1}{n}.$$

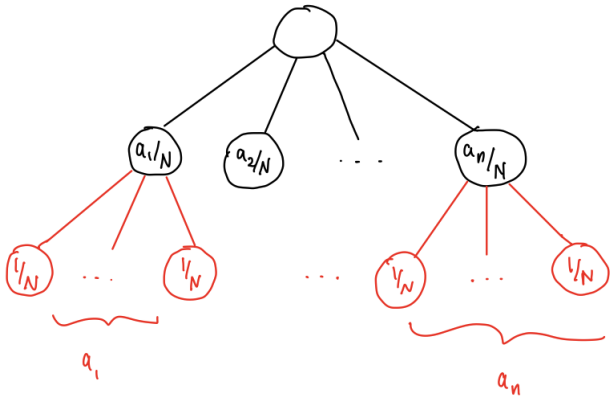
As $n \to \infty$, we have $m/n \to \log(r)$, so we conclude that $H_r = H_2 \log(r)$.

Since the proof asks for uniqueness up to scalars, we may normalize so that $H_2 = 1$; our goal is then to show that

$$H(p_1, \dots, p_n) = -\sum_i p_i \log(p_i).$$

By continuity, it will suffice to do this when the $p_i \in \mathbf{Q}$ are all rational. We may then assume that each $p_i = a_i/N$ for some common denominator $N \in \mathbf{Z}$, where each a_i is a positive integer. We may then further refine this distribution into the uniform one on N elements, by breaking the ith outcome (which has probability a_i/N) into a_i sub-outcomes

with the uniform distribution.



Then by decomposability of entropy, we obtain

$$H\Big(\frac{a_1}{N},\dots,\frac{a_n}{N}\Big)+\frac{a_1}{N}H\Big(\frac{1}{a_1},\dots,\frac{1}{a_1}\Big)+\dots+\frac{a_n}{N}H\Big(\frac{1}{a_n},\dots,\frac{1}{a_n}\Big)=H\Big(\frac{1}{N},\dots,\frac{1}{N}\Big).$$

As we have already analyzed the uniform case, we know that $H(\frac{1}{a_i},\dots,\frac{1}{a_i})=H_{a_i}=\log(a_i)$. Hence we may solve

$$H\Big(\frac{a_1}{N},\dots,\frac{a_n}{N}\Big)=\log(N)-\sum_i\frac{a_i}{N}\log(a_i)=-\sum_i\frac{a_i}{N}\log\Big(\frac{a_i}{N}\Big),$$

as desired.

□

2.4. **Conditional entropy.** Let (Ω,P) be a probability ensemble and let X and Y be discrete random variables Ω. Then the *conditional entropy* $H(X|Y)$ of X given Y is defined as

$$H(X|Y)=-\sum_y P(Y=y)\sum_x P(X=x|Y=y)\log P(X=x|Y=y)$$

Note that the inner sum is the entropy of the random variable $(X|Y=y)$. Intuitively, the conditional entropy $H(X|Y)$ measures the expected "new" information in X if Y is given.

Example 2.4.1. If X and Y are independent, then upon substituting $P(X=x,Y=y)=P(X=x)P(Y=y)$ we find that

$$\begin{aligned}H(X|Y)&=-\sum_y P(Y=y)\sum_x P(X=x)\log P(X=x)\\&=-\sum_x P(X=x)\log P(X=x)=H(X),\end{aligned}$$

as expected.

2.4.1. *Mutual information.* Let X,Y be random variables on a common outcome space. The *mutual information between X and Y* is

$$I(X;Y):=H(X)-H(X|Y).$$

Intuitively, $I(X;Y)$ is the (expected) amount of information about X already present in Y, and the identity $H(X) = I(X;Y) + H(X|Y)$ decomposes the entropy of X into the part which is already present in Y, plus the part which is not in Y.

This intuition suggests that $I(X;Y) = I(Y;X)$, which is true. To see this, we invoke the "chain rule" for entropy,

$$H(X,Y) = H(Y) + H(X|Y). \tag{2.4.1}$$

Substituting this into the definition of mutual information, we see that

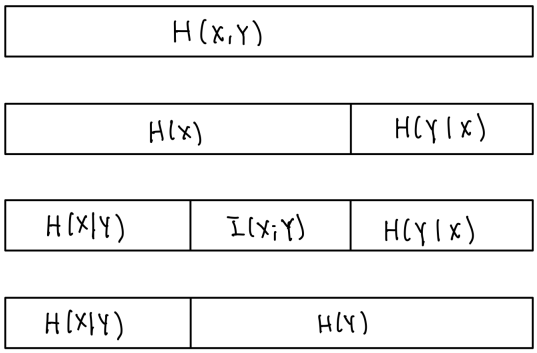
$$I(X;Y) = H(X) + H(Y) - H(X,Y),$$

which is manifestly symmetric in X and Y.

Proposition 2.4.2. *For any two random variables X, Y, we have $I(X;Y) \geq 0$.*

Proof. The proof is rather non-trivial, and follows by combining Theorem 2.4.4 and Lemma 2.4.6 below. □

The following cartoon depicts the decomposition of $H(X,Y)$ into conditional entropy and mutual information.



2.4.2. *KL divergence.* For two probability distributions P and Q on Ω, the *Kullback–Leibler (KL) divergence* is

$$\mathrm{KL}(P \mid\mid Q) = \sum_{\omega} P(\omega) \log \frac{P(\omega)}{Q(\omega)}.$$

Remark 2.4.3. There are some edge cases for when the probabilities appearing in this formula vanish. Continuity dictates what to do in these cases; the answers are as follows.

- If $P(\omega) = 0$, then the summand corresponding to ω is dropped from the formula.
- If $Q(\omega) = 0$ but $P(\omega) \neq 0$, then $\mathrm{KL}(P \mid\mid Q) = \infty$.

Theorem 2.4.4 (Gibbs' inequality)**.** *Let P, Q be any two probability distributions on a discrete sample space Ω. Then we have $\mathrm{KL}(P||Q) \geq 0$, with equality if and only if $P = Q$.*

Remark 2.4.5. Theorem 2.4.4 is taken as license to regard $\mathrm{KL}(P||Q)$ as a notion of "distance" between probability distributions P and Q. Note however that $\mathrm{KL}(P||Q) \neq \mathrm{KL}(Q||P)$, in contrast to usual notions of distance.

Proof. Let's write

$$\mathrm{KL}(P \mid\mid Q) = \sum_{\omega} P(\omega) \log \frac{P(\omega)}{Q(\omega)} = \sum_{\omega} P(\omega) \cdot \Big(- \log \frac{Q(\omega)}{P(\omega)} \Big). \tag{2.4.2}$$

We recall *Jensen's inequality* in the following form: if $f\colon [a,b] \to \mathbf{R}$ is a convex function and X is any random variable valued in $[a,b]$, then

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X]).$$

Note that function $f(x) = -\log(x)$ is convex, as $f''(x) = 1/x^2 \geq 0$. For the random variable $X(\omega) = Q(\omega)/P(\omega)$, we may rewrite (2.4.2) as $\mathbb{E}_P[f(X)]$, the expected value of $f(X)$ with respect to the probability distribution P.[9] Then Jensen's inequality says that

$$\mathbb{E}_P[f(X)] \geq f(\mathbb{E}_P[X]) = f\Big(\sum_\omega P(\omega)\frac{Q(\omega)}{P(\omega)}\Big) = f\Big(\sum_\omega Q(\omega)\Big) = -\log(1) = 0,$$

giving the desired inequality. Moreover, equality holds if and only if $Q(\omega)/P(\omega)$ is constant. □

The next Lemma tells us that we can think of mutual information as the mutual information between random variables X, Y as the KL divergence between their joint distribution and the independent distribution. Combined with Theorem 2.4.4, it completes the proof of Proposition 2.4.2.

Lemma 2.4.6. *Let X, Y be random variables on Ω. Let $P_{X,Y}$ be the joint law of (X,Y) on $\Omega_X \times \Omega_Y$, and let $P_X \otimes P_Y$ be the product law, so that*

$$P_{X,Y}(x,y) = P(X=x, Y=y) \quad \text{and} \quad P_X \otimes P_Y(x,y) = P(X=x)P(Y=y).$$

Then

$$I(X;Y) = \mathrm{KL}\left(P_{X,Y} \,\|\, P_X \otimes P_Y\right).$$

Proof. By definition, we have

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= -\sum_x P(X=x)\log P(X=x) + \sum_{x,y} P(Y=y)P(X=x|Y=y)\log P(X=x|Y=y). \end{aligned} \tag{2.4.3}$$

We can rewrite the first summand as

$$-\sum_x P(X=x)\log P(X=x) = -\sum_{x,y} P(X=x, Y=y)\log P(X=x).$$

We can rewrite the second summand as

$$\sum_{x,y} P(Y=y)P(X=x|Y=y)\log P(X=x|Y=y) = \sum_{x,y} P(X=x,Y=y)\log\frac{P(X=x,Y=y)}{P(Y=y)}.$$

Substituting these into the expression (2.4.3) for $I(X;Y)$ and combining terms, we obtain

$$\begin{aligned} I(X;Y) &= \sum_{x,y} P(X=x,Y=y)\log\Big(\frac{P(X=x,Y=y)}{P(X=x)P(Y=y)}\Big) \\ &= \mathrm{KL}\left(P_{X,Y} \,\|\, P_X \otimes P_Y\right), \end{aligned}$$

as desired. □

[9] We will write the proof under the assumption that $P(\omega) \neq 0$. The case where some $P(\omega) = 0$ then follows by continuity.

2.5. **Compression.** Shannon's original applications of information were to coding and compression. This will give an intuitive interpretation to the entropy of a random variable, as the natural limit to how much its outcomes can be "compressed" in expectation.

Example 2.5.1. Suppose we repeatedly flip a coin, which comes up heads with probability p, and record the results of N flips. Let $X = X_1X_2 \dots X_n$ be the outcomes, so the X_i are i.i.d. Bernoulli random variables with parameter p. By Example 2.2.4 and Exercise 2.2.5, we have

$$H(X) = nH(X_i) = n(-p\log(p) - (1-p)\log(1-p)). \tag{2.5.1}$$

As a sanity check, let's think about (2.5.1) in a couple of special cases. If $p = 1/2$, then $H(X_i) = 1$. This makes sense, as we need 1 bit to convey the outcome of a uniformly random bit. Correspondingly, we need $H(X) = n$ to convey the outcomes of n independent uniformly random bits.

As p tends to 0, the entropy $H(X)$ decreases. Indeed, we expect the outcomes of the coin flips will be mostly tails. In other words, we can expect X to be a "sparse vector", and we can imagine leveraging this to compress the results. Later, we will see precise schemes for doing this efficiently.

2.5.1. *Formal setup.* Let's try to make these ideas precise. Suppose we are communicating using a (finite) "alphabet" $\mathcal{A} = \{a_1, a_2, \dots\}$ of symbols. Let $\mathcal{A}^+ = \bigcup_{n\geq 1} \mathcal{A}^n$ be the set of non-empty finite sequences of symbols from $\mathcal{A}$, so for example $\{0,1\}^+$ is the set of finite bit strings.

By a *compression function*, we mean a function

$$c\colon \mathcal{A}^+ \to \{0,1\}^+,$$

which converts possible messages in our alphabet into bitstrings.

Now we will try to quantify how efficient a compression scheme is. Let $X = (X_1, X_2, \dots, X_n)$ be a random variable valued in $\mathcal{A}^n$. For example, X_i could be the outcome of a biased coin flip, as in Example 2.5.1. However, we imagine the X_i as *not* necessarily being independent.

Example 2.5.2. We could take $\mathcal{A}$ to be the English alphabet $\{\texttt{a}, \texttt{b}, \dots, \texttt{z}\}$. Then $P(X_i = \texttt{u}) \approx 2.8\%$, while $P(X_i = \texttt{u} | X_{i-1} = \texttt{q}) \approx 100\%$.

Definition 2.5.3. For $x_1 \dots x_n \in \mathcal{A}^+$, we introduce the abbreviation

$$x_{1:n} := (x_1, x_2, \dots, x_n).$$

Define $\ell_c(x_{1:n})$ to be the length (i.e., number of bits) of $c(x_{1:n}) \in \{0,1\}^+$.

For $X = (X_1, X_2, \dots, X_n)$ a sequence of random variables valued in $\mathcal{A}$, define the *expected length* of c to be $\mathbb{E}[\ell_c(X)]$.

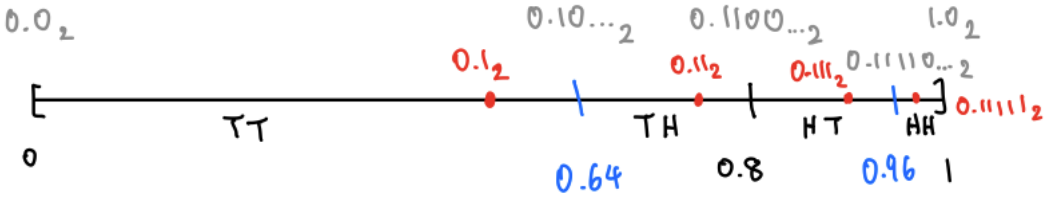
Informally, $\mathbb{E}[\ell_c(X)]$ is the expected number of bits to encode a length n message in the alphabet $\mathcal{A}$, and the goal of compression is to find a compression function c minimizing this. In this context, the entropy of X finds an interpretation as the natural limit of the expected length of any compression function c for X. Shannon's *Source Coding Theorem* essentially says that it is impossible to compress with expected length below the entropy, and that there do exist codes (e.g., Huffman codes) which achieve

$$H(X) \leq \mathbb{E}[\ell_c(X)] < H(X) + 1$$

Conversely, we will give an explicit construction to show that the entropy can essentially be reached by a compression function.

2.5.2. *Arithmetic coding.* One theoretically efficient compression scheme (which is even used in practice!) is *arithmetic coding*. The main idea of arithmetic coding is to subdivide the unit interval $[0, 1]$ into subintervals, such that each outcome $x = (x_1, \dots, x_n)$ gets assigned to an interval I_x with $P(X = x) = |I_x|$ (the size of I_x). The compression function then returns the shortest binary expansion of a number lying in the interval I_x.

Example 2.5.4. Consider two independent flips of a biased coin with T probability 0.8. There are four possible outcomes. We will assign them intervals in $[0, 1]$ by the following rule. We will build the intervals iteratively (with initial interval being all of $[0, 1]$), subdividing so that the left 80% is labeled T and the right 20% is labeled H.



In the first iteration, we have two intervals,

(1) T $\leftrightarrow [0.0, 0.8)$
(2) H $\leftrightarrow [0.8, 1.0)$.

In the next iteration, each of these is further subdivided, giving the intervals

(1) TT $\leftrightarrow [0.0, 0.64)$.
(2) TH $\leftrightarrow [0.64, 0.8)$.
(3) HT $\leftrightarrow [0.8, 0.96)$.
(4) HH $\leftrightarrow [0.96, 1.0]$.

The binary expansions of the endpoints are then

(1) $0.0 = 0.0_2$
(2) $0.64 = 0.10\ldots_2$
(3) $0.8 = 0.110\ldots_2$
(4) $0.96 = 0.11110\ldots_2$
(5) $1.0 = 1.0_2$.

Finally, we select the shortest binary expansion lying (strictly, let's say, although we could optimize this further) in each interval. This is found by looking at the first position where the left and right endpoints differ for each interval. We end up with the encoding in the following table:

Outcome	Encoding	Bits
TT	$.1_2$	1
TH	$.11_2$	2
HT	$.111_2$	3
HH	$.11111_2$	5

Compared to the naive encoding where we just output 0 for T and 1 for H, we see that we save one bit in the outcome TT, but have to send 3 extra bits for the outcome HH. Since TT is much more likely than HH, this is a good tradeoff! Indeed, the expected length of our encoding is

$$0.64 \times 1 + 0.16 \times 2 + 0.16 \times 3 + 0.04 \times 5 = 1.64$$

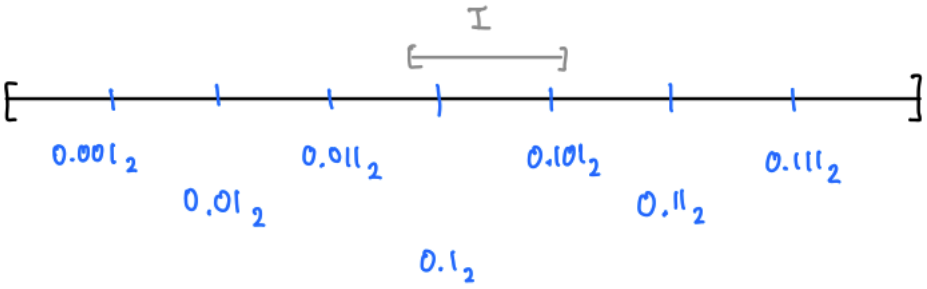
which already saves over the naive encoding that always requires 2 bits.

Why does this work? The point is that to convey a number in an interval I, we need about $\log(1/|I|)$ bits. Thus, the more probable outcomes are assigned larger intervals, and will require fewer bits to encode. The efficiency of arithmetic encoding is guaranteed by the following Lemma.

Lemma 2.5.5. *The expected length of an arithmetic code is at most* $H(X) + 2$.

Proof. We claim that *any interval* I *strictly contains a number whose binary expansion has at most* $1 + \lceil \log(1/|I|) \rceil$ *bits.*

Indeed, let $n = \lceil \log(1/|I|) \rceil$. In other words, n is the smallest integer such that $1/|I| < 2^n$, or equivalently $|I| > 2^{-n}$. We can subdivide the unit interval into 2^n equally spaced subintervals of length 2^{-n}. Then the endpoints lies at the numbers whose binary expansions have length exactly n, and so only require n bits to convey.



Since we arranged $|I| > 2^{-n}$, there will be an endpoint lying in I. By increasing n by one if necessary, we can always avoid the situation where the endpoint lies on the boundary of I, proving the claim.

Thanks to this claim, we can bound the expected length of $c(X)$ as

$$\begin{aligned}\sum_x P(X=x)\ell(c(X)) &\le \sum_x P(X=x)\Big(1 + \lceil \log(1/|I_x|)\rceil\Big)\\ &\le \sum_x P(X=x)\Big(2 + \log(1/|I_x|)\Big)\\ &= \sum_x P(X=x)\Big(2 + \log(1/P(X=x))\Big) = 2 + H(X).\end{aligned}$$

□

2.5.3. *Relation to generative sequence models.* Arithmetic coding requires the encoder and decoder to both have access to the function that converts an outcome $x_{1:n} = x_1 x_2 \dots x_n$ into an interval. In practice, this is implemented using a *generative sequence model*, which is a model for the conditional probability functions

$$P_\theta(x_n | x_{1:n-1}) := P_\theta(x_n | x_1, \dots, x_{n-1}), \quad \text{for all } n. \tag{2.5.2}$$

Given the generativee sequence model, we can construct the intervals by iteratively subdividing as in Example 2.5.4.

Remark 2.5.6. ChatGPT, is a neural network trained to predict a conditional probability function of the form (2.5.2). This suggests a mathematical equivalence between compression and generative modeling.

Arithmetic codes give one direction of this equivalence: passing from generative sequence models to compression functions. To complete the equivalence, let's see how to go the other way. Assume that we are given a compression function $c\colon \mathcal{A}^+ \to \{0,1\}^+$, so that the length $\ell_c(x)$ is defined for $x \in \mathcal{A}^+$.

Define

$$\Delta\ell_c(x_n|x_{1:n-1}) := \ell_c(x_{1:n}) - \ell_c(x_{1:n-1})$$

to be the number of *additional* bits that the compression function uses to encode x_n after having already encoded $x_{1:n-1}$. Then we define

$$P_\theta(x_n|x_{1:n-1}) = 2^{-\Delta\ell_c(x_n|x_{1:n-1})}.$$

In other words, given a compressor, we should define $-\log P(x_n|x_{1:n-1})$ to be the *number of additional bits required to encode x_n given context $x_{1:n-1}$*. Looking back at Definition 2.1.3, we see that this is precisely the "information" of x_n given $x_{1:n-1}$. It is easy to check that this reconstructs the sequence model from an arithmetic code.

Remark 2.5.7. Observe that in the loss function for classification (1.5.1), $-\log P_\theta(x_n|x_{1:n-1})$ would be exactly the "cross-entropy" loss for the model θ on the token x_n appearing after the sequence $x_{1:n}$. Therefore, for a sequence model, minimizing cross-entropy loss can be thought of equivalently as minimizing the expected length of the corresponding compressor.

Remark 2.5.8. The paper [DRD+24] investigates this equivalence empirically, by using arithmetic codes built from LLMs to compress data, and comparing the efficiency to existing standard compression functions. Their results show that the arithmetic codes compare well; however, this does not account for the size of the LLMs, whose parameters require many bits to express.

2.5.4. *Infinitude of the primes.* As a fun epilogue on the themes of this section, we will give another argument for the infinitude of the prime numbers, from an information-theoretic perspective.[10]

Theorem 2.5.9. *There are infinitely many prime numbers.*

Proof. Imagine that we are trying to compress a uniformly random integer in $[1,\dots,N]$. In expectation, we need at least $\lceil \log(N) \rceil$ bits, since the number of bit strings of length $\lfloor \log(N) \rfloor$ is bounded above by $2^{\log(N)} = N$. Furthermore, it is clear that there is an encoding that achieves this bound (namely, the binary representation).

In the rest of the proof, we will ignore these rounding functions; the reader can easily restore them as necessary.

Suppose for the sake of contradiction that there are only finitely many prime numbers $p_1,\dots,p_d$. Any positive integer $n \geq 1$ admits a unique prime factorization as a product

$$n = \prod_i p_i^{e_i}$$

for unique $e_1,\dots,e_d \geq 0$. Furthermore, since each $p_i \geq 2$, each $e_i \leq \log(n)$. Hence e_i can be encoded with $\approx \log(\log(n))$ bits, which means that each n can be encoded by specifying the $e_1,\dots,e_d$, which takes $\approx d\log(\log(n))$ bits. But this gives an encoding of the first N natural numbers with $O(\log\log N)$ bits, contradicting the first paragraph of the proof! $\square$

[10] I learned this argument from a talk by Tadashi Tokieda, although he says that he did not originally come up with it.

3. Statistical inference

3.1. **Central Limit Theorem.** Let $X_1, X_2, \ldots$ be i.i.d. random variables with (finite) mean μ.

Slogan 3.1.1. The *Law of Large Numbers* says that the random variables

$$\overline{X}_n := \frac{X_1 + \ldots + X_n}{n}$$

converge to the mean μ as $n \to \infty$.

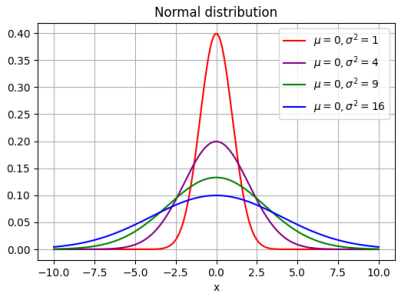
One precise formulation of Slogan 3.1.1 is that for any fixed $\epsilon > 0$,

$$\lim_{n\to\infty} P(|\overline{X}_n - \mu| > \epsilon) = 0.$$

(This is called the "Weak Law of Large numbers"; it gives a type of convergence called *convergence in probability*.)

Given the Law of Large Numbers, it is natural to ask about the random fluctuation of $\overline{X}_n$ around its mean μ. Suppose X_i has variance σ^2. Recall that the *Gaussian* (or *normal*) distribution with mean μ and variance σ^2, denoted $\mathcal{N}(\mu, \sigma^2)$, has density function

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Slogan 3.1.2. Suppose the X_i have finite mean σ^2. The *Central Limit Theorem* says that

$$\frac{1}{\sqrt{n}} \sum_{i=1}^{n} (X_n - \mu).$$

converges in distribution to a Gaussian distribution with mean 0 and variance σ^2.

The Central Limit Theorem explains the "universality" of the Gaussian distribution: it is the limiting behavior of the average of independent (identically distributed) random variables. Note that $\mathrm{Var}(\sum_{i=1}^{n}(X_n - \mu)) = \sum_{i=1}^{n} \mathrm{Var}(X_i - \mu) = n\sigma^2$, hence why $\sqrt{n}$ is the correct normalization factor to see an interesting result. This is responsible for the famous "square root cancellation" principle, asserting that an accumulation of n random fluctuations in $[-1, 1]$ can be expected to reach the order of magnitude of $\sqrt{n}$.

We will give a partial derivation of the Central Limit Theorem (one that is valid under additional technical assumptions). Recall that the *moment generating function* of a random variable X is the formal series

$$M_X(t) = \mathbb{E}[e^{tX}] = 1 + t\mathbb{E}[X] + \frac{t^2}{2}\mathbb{E}[X^2] + \ldots + \frac{t^k}{k!}\mathbb{E}[X^k] + \ldots$$

Example 3.1.3. The moment generating function of $X \sim \mathcal{N}(0, \sigma^2)$ is

$$\begin{aligned} M_X(t) &= \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} e^{tx} e^{-\frac{1}{2\sigma^2}x^2}\, dx \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} e^{-\frac{1}{2\sigma^2}(x^2 - 2t\sigma^2 x)}\, dx \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} e^{-\frac{1}{2\sigma^2}((x-t\sigma^2)^2 - t^2\sigma^4)}\, dx \\ &= e^{\frac{t^2\sigma^2}{2}}. \end{aligned}$$

More generally, this implies by a translation argument that the moment generating function of $X \sim \mathcal{N}(\mu, \sigma^2)$ is

$$M_X(t) = e^{\mu t + \frac{\sigma^2 t^2}{2}}.$$

This is essentially a *characterization* of the Gaussian distribution: the logarithm of its moment generating function is a quadratic polynomial in t, of the form $\mu t + \frac{\sigma^2 t^2}{2}$.

Now let's derive the Central Limit Theorem under the assumption that $X_1, X_2, \dots$ are i.i.d. random variables with well-defined moment generating function[11] $M_{X_i}(t)$. Without loss of generality, assume that $\mu = 0$, so that

$$M_{X_i}(t) = 1 + \frac{\sigma^2 t^2}{2} + O(t^3)$$

We have

$$M_{\frac{1}{\sqrt{n}}\sum X_i}(t) = \mathbb{E}\left[e^{t\frac{1}{\sqrt{n}}\sum_{i=1}^n X_i}\right] = \mathbb{E}[e^{tX_i/\sqrt{n}}]^n = M_{X_i}(t/\sqrt{n})^n.$$

Taking logarithms, we then see that

$$\ln M_{\overline{X}_n}(t) = n \ln M_{X_i}(t/\sqrt{n}) = n \ln\left(1 + \frac{\sigma^2 (t/\sqrt{n})^2}{2} + O((t/\sqrt{n})^3)\right)$$

Using the Taylor series of $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$, we find that all but the lowest order term vanish as $n \to \infty$, so that

$$\lim_{n\to\infty} \ln M_{\overline{X}_n}(t) = \frac{\sigma^2 t^2}{2},$$

indicating a Gaussian distribution. □

3.2. **Estimators.** Let P be a probability distribution.

Definition 3.2.1. A *random sample* from the distribution $f(x)$ is a sequence of random variables $X_1, \dots, X_n$ which are i.i.d. with distribution P.

Definition 3.2.2. A *statistic* of a random sample $(X_1, \dots, X_n)$ is a random variable of the form $W = T(X_1, \dots, X_n)$ for some function T.

Example 3.2.3 (Sample mean)**.** The *sample mean* of $X_1, \dots, X_n$ is the statistic

$$\overline{X} := \frac{1}{n}(X_1 + \dots X_n).$$

Note the difference between the *mean* of the X_i, which is a number, and the sample mean, which is itself a random variable.

[11] This does not always exist. In general, we should work instead with the *characteristic function* $\mathbb{E}[e^{itX}]$, which always exists.

In statistical inference, we will find ourselves in the following kind of setup. Suppose we have family of probability distributions P_θ, depending on a parameter θ. An *estimator* of a random sample $(X_1, \ldots, X_n)$ is a function $W(X_1, \ldots, X_n)$, which is intended to infer the parameter θ. Mathematically, an estimator is identical to a statistic; the difference between the two terms lies in their psychological connotations: we think of an estimator as "trying" to estimate something, whereas a statistic does not necessarily have any attached "purpose".

Definition 3.2.4. Below, we denote by $\mathbb{E}_\theta[W]$ the expected value of W according to the distribution P_θ. The *bias* of an estimator W of a parameter θ is

$$\mathrm{Bias}_\theta(W) := \mathbb{E}_\theta[W] - \theta.$$

This can be viewed as a function of θ. An estimator whose bias is identically 0 (as a function of θ) is called *unbiased*.

Example 3.2.5 (Estimating the mean)**.** By linearity of expectation,

$$\mathbb{E}[\overline{X}] = \frac{1}{n}(\mathbb{E}[X_1] + \ldots + \mathbb{E}[X_n]) = \mathbb{E}[X_i]$$

since the X_i are i.i.d. In other words, the sample mean is an unbiased estimator of the mean.

Example 3.2.6. Let $X_1, \ldots, X_n$ be a random sample. For each $i = 1, \ldots, n$, we can take X_i as an estimator of the mean of the X_i. This is obviously an unbiased estimator.

Example 3.2.7 (Estimating the variance)**.** Recall that the *variance* of X_i is

$$\mathbb{E}[(X_i - \mathbb{E}_\theta[X_i])^2] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2.$$

Let's define the (ad hoc terminology) "naive sample variance" to be

$$S^2_{\text{naive}}(X_1, \ldots, X_n) := \frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2.$$

Is this an unbiased estimator of the variance? Let's calculate $\mathbb{E}[S^2_{\text{naive}}(X_1, \ldots, X_n)]$. First, some algebra gives

$$\begin{aligned}\sum_{i=1}^n (X_i - \overline{X})^2 &= \sum_i \Big((X_i - \mu) - (\overline{X} - \mu)\Big)^2 \\ &= \sum_i (X_i - \mu)^2 - 2\sum_i (X_i - \mu)(\overline{X} - \mu) + \sum (\overline{X} - \mu)^2.\end{aligned}$$

Using that $\sum_i (X_i - \mu) = n(\overline{X} - \mu)$, this expression simplifies as

$$\sum_i (X_i - \mu)^2 - n(\overline{X} - \mu)^2.$$

Then taking expectation, we find that

$$\begin{aligned}\mathbb{E}\Big[\sum_{i=1}^n (X_i - \overline{X})^2\Big] &= \mathbb{E}\Big[\sum_i (X_i - \mu)^2 - n(\overline{X} - \mu)^2\Big] \\ &= n\,\mathrm{Var}[X_i] - n\frac{\mathrm{Var}[X_i]}{n} = (n-1)\,\mathrm{Var}[X_i].\end{aligned}$$

Dividing by n, we obtain

$$\mathbb{E}[S^2_{\text{naive}}(X_1, \ldots, X_n)] = \frac{n-1}{n}\,\mathrm{Var}[X_i].$$

In particular, the naive sample variance is *not* unbiased (unless $\mathrm{Var}[X_i]$ happens to be 0, in which case there is no randomness anyway). Furthermore, we see what is needed to correct it: define the *sample variance*

$$S^2(X_1, \dots, X_n) := \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \overline{X})^2 = \frac{n}{n-1} S^2_{\text{naive}}.$$

Then the same calculation shows that $S^2(X_1, \dots, X_n)$ is an unbiased estimator of the variance.

What happened here? Qualitatively, we saw that the naive sample variance is an *underestimate* of the variance. For example, looking at the case $n = 1$, we see that the naive sample variance would always be 0, which is clearly an underestimate, while the sample variance is undefined (although this may appear mathematically disturbing at first, it is actually reasonable: a single sample is not enough to conclude anything about variance).

3.3. **Maximum Likelihood Estimation.** Suppose we have family of probability distribution, parametrized by θ. We denote the PMF/PDF by $p(x|\theta)$.

- The probability of an outcome x given parameter θ is $p(x|\theta)$.
- The *likelihood* of a parameter θ given an outcome x is $L(\theta|x) := p(x|\theta)$.

Note that the right side "$p(x|\theta)$" of these formulas looks the same; the difference lies in what is being held fixed and what is viewed as variable.

Slogan 3.3.1. In practical terms, we use probability to predict outcomes from a given model, while we use likelihood to infer a model from a given outcome.

Definition 3.3.2. Suppose we are given an outcome x and want to infer θ. The *maximum likelihood estimate* of θ is

$$\hat{\theta} := \arg\max_{\theta} L(\theta|x) = \arg\max_{\theta} p(x|\theta).$$

Example 3.3.3. Let $X_1, X_2, \dots, X_n$ be a random sample from a Bernoulli distribution Bernoulli(θ). Then for $\underline{x} = (x_1, \dots, x_n) \in \{0,1\}^n$ we have

$$L(\theta|\underline{x}) = \prod_{i=1}^{n} \theta^{x_i}(1-\theta)^{1-x_i} = \theta^k(1-\theta)^{n-k} \quad \text{where } k = \#\{i \colon x_i = 1\}.$$

We could differentiate this directly, but we will instead leverage the key observation that because of monotonicity of log, *maximizing the likelihood is equivalent to maximizing its logarithm*. Now, the "log-likelihood" $\ln L(\theta|\underline{x})$ has a more convenient expression,

$$\ln L(\theta|\underline{x}) = k \ln(\theta) + (n-k)\ln(1-\theta). \tag{3.3.1}$$

Now we can easily differentiate to solve for the unique critical point $\hat{\theta} = k/n$. A little more work shows that this is the global maximum of (3.3.1), so we conclude that the maximum likelihood estimate is (the obvious guess) $\hat{\theta} = k/n$.

Example 3.3.4. Let $X_1, \dots, X_n$ be a random sample from a normal distribution $\mathcal{N}(\mu, \sigma^2)$. Then for $\underline{x} = (x_1, \dots, x_n)$, we have

$$L(\mu, \sigma^2|\underline{x}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_i-\mu)^2} = \frac{1}{(2\pi\sigma^2)^{n/2}} e^{-\frac{1}{2\sigma^2}\sum_i (x_i-\mu)^2}.$$

To maximize $L(\mu, \sigma^2|\underline{x})$, we will again use the trick of looking instead at the log-likelihood,

$$\log L(\mu, \sigma^2|\underline{x}) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2.$$

Let's examine the critical points. Differentiating with respect to the parameters, we find that

$$\frac{d}{d\mu}\ln L(\mu, \sigma^2|\underline{x}) = \frac{1}{\sigma^2}\left(\sum_i (x_i - \mu)\right), \tag{3.3.2}$$

and

$$\frac{d}{d\sigma^2}\ln L(\mu, \sigma^2|\underline{x}) = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{n}(x_i - \mu)^2. \tag{3.3.3}$$

Setting (3.3.2) equal to 0 gives $\boxed{\hat{\mu} = \frac{1}{n}\sum_i x_i,}$ and then setting (3.3.3) equal to 0 gives $\boxed{\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2.}$ Note that these are what we called the *sample mean* and *naive sample variance*, respectively. With a bit more work (to verify that these $\hat{\mu}$ and $\hat{\sigma}^2$ give global maxima), we find that this provides the maximum likelihood estimates. In particular, the MLE can be biased, as we see in this case by Example 3.2.7.

Remark 3.3.5. One *weakness* of MLE is that it can be highly unstable as a function of the data, which in practice is noisy. The following example (due to Olkin et al.) is [CB90, p.297, Example 7.2.2]. Suppose we sample from a Binomial(k, p) distribution with unknown k and p. Consider the two datasets

- $[16, 18, 22, 25, 27]$
- $[16, 18, 22, 25, 28]$

For the first, the MLE of k is $\hat{k} = 99$, while for the second the MLE of k is $\hat{k} = 190$.

3.3.1. *Relation to cross-entropy.* Let p and q be two probability distributions on the same sample space (which we conflate with their PMF/PDF). Then the *cross-entropy* of p, q is

$$H(p, q) := \mathbb{E}_p[\log(1/q)] = \mathbb{E}_p[-\log q].$$

Example 3.3.6. Supposing that p and q are discrete for simplicity, this unpacks to

$$H(p, q) = -\sum_i p_i \log(q_i).$$

As with the KL divergence, there are some edge cases in interpreting this formula (Remark 2.4.3): if $p_i = 0$, then the summand is omitted, while if $p_i \neq 0$ but $q_i = 0$, then $H(p, q) = \infty$.

In the case of discrete random variables, it is clear that $H(p, q) \geq 0$, since each $-p_i \log(q_i) \geq 0$. This tells us that we can reasonably think of $H(p, q)$ as being some measure of "distance" between p and q. (Note however that $H(p, q) \neq H(q, p)$. We'll come back to this later.) But what is it trying to capture?

Let's place ourselves in the mindset of logistic regression. Think of p as being the empirical distribution given by data (corresponding to x), and q as being the model parameter (corresponding to θ). Then the Maximum Likelihood Estimate of the parameter maximizes the likelihood of the observed data under the model q.

Imagine for concreteness that the empirical distribution p arises from N total outcomes, so outcome ω_i comes up Np_i times (if it helps psychologically, you can imagine that N is divisible enough so that Np_i is an integer). Under our model q, the probability of this happening is $q_i^{Np_i}$, so the maximum likelihood estimate is

$$\hat{q} = \arg\max_q \Big(\prod_i q_i^{Np_i} \Big).$$

Again, it is equivalent to maximize the log-likelihood,

$$\hat{q} = \arg\max_q \Big(\prod_i q_i^{Np_i} \Big) = \arg\max_q \log \Big(\prod_i q_i^{Np_i} \Big) = \arg\max_q \Big(\sum_i Np_i \log q_i \Big).$$

Now we see that the N is irrelevant, so we can pull it out. Also, to put this in our standard "loss minimization" framework, we can trade the $\arg\max$ for $\arg\min$ if we introduce a negative sign. This means that the maximum likelihood estimate can be written equivalently

$$\hat{q} = \arg\min_q H(p, q)$$

and this suggests the cross-entropy $H(p, q)$ as our loss function. In other words, *minimizing cross-entropy loss is solving for the maximum likelihood estimate!*

Remark 3.3.7. From this analysis, we see that $H(p, q)$ reflects "the probability of empirical data p under the model q". This description is not symmetric under exchanging $p \leftrightarrow q$. Inspecting the formula $-\sum p_i \log q_i$, we see that summands where q_i is small but p_i is non-negligible contribute a large value. This guides us to the heuristic principle:

Slogan 3.3.8. Cross-entropy loss punishes predictions that are "confident but wrong" that an outcome will not occur.

Example 3.3.9. In classification problems, it is common to train a neural net that computes a probability distribution q over possible classes, and then outputs the most probable class. Suppose the data is of the form $\{(x_i, y_i)\}$ where y_i is the ground truth label on the input x_i. The loss function for a datapoint (x_i, y_i) would then by cross-entropy $H(p, q)$ where p is the delta function on the true class y_i, and q is the model's probability distribution. This is simply

$$H(p, q) = -\log q(y_i).$$

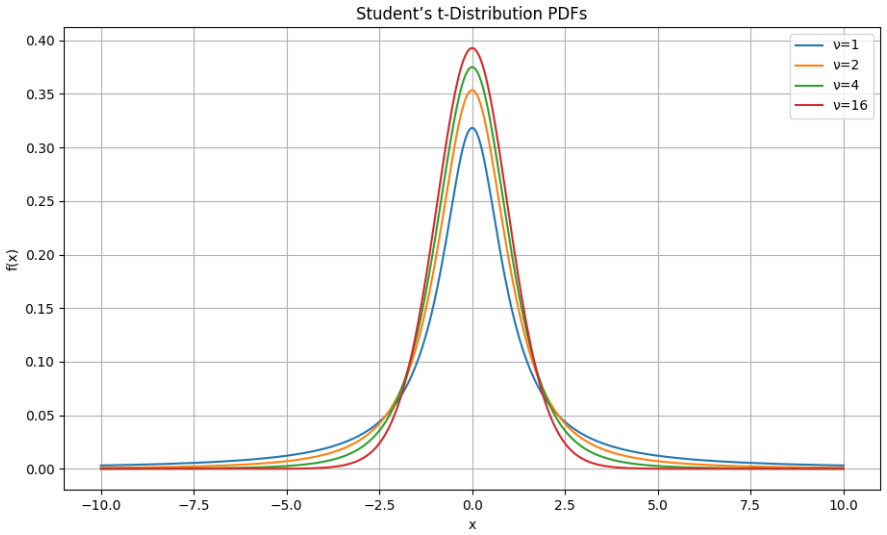
By contrast, $H(q, p)$ would be infinite as long as q is supported on some outcome other than y_i, which would typically be the case. Therefore, $H(q, p)$ would not be an appropriate loss function.

Example 3.3.10. Student's[12] "t_ν-distribution" depends on a parameter ν, and has PDF

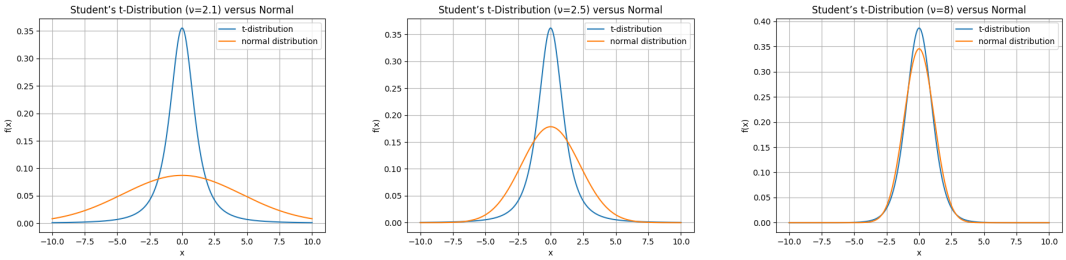
$$f_\nu(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma(\nu/2)} \left(1 + \frac{x^2}{\nu}\right)^{-(\nu+1)/2}.$$

[12]Student is a name, which a pseudonym used by William Sealy Gosset.

The graph of f_ν is "bell-shaped", roughly similar in appearance to a normal distribution.



For $\nu > 2$, f_ν has mean 0 and finite variance $\frac{\nu}{\nu-2}$. Let's compare to the normal distributions with these means and variances.



What do the cross-entropies look like in each of these situations? The following table is the result of numerical (truncated) integration over $x \in [-500, 500]$.

$H(t_{2.1}, \mathcal{N}(0,21))$	2.5738
$H(t_{2.5}, \mathcal{N}(0,5))$	2.0782
$H(t_8, \mathcal{N}(0,4/3))$	1.5625

$H(\mathcal{N}(0,21), t_{2.1})$	3.726
$H(\mathcal{N}(0,5), t_{2.5})$	2.4343
$H(\mathcal{N}(0,4/3), t_8)$	1.5722

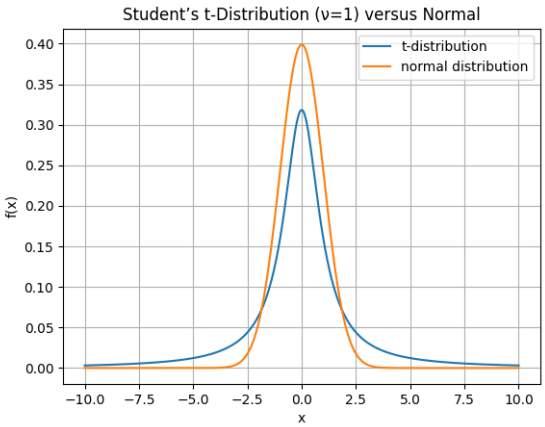
For $\nu = 2.1, 2.5$ we see that $H(t_\nu, \mathcal{N}(0,\sigma^2))$ is noticeably smaller than $H(\mathcal{N}(0,\sigma^2), t_\nu)$, while they are pretty close for $\nu = 8$. Let's try to rationalize how we could have guessed this from looking at the graphs. Intuitively, taking the t_ν-distribution as the model and the normal distribution as the data, we might imagine that it gets penalized heavily in the low-probability tails where the normal distribution lies above it. With the roles switched, the region where the t_ν-distribution lies significantly above the normal is also the highest probability region for the normal distribution.

That being said, human intuition can misleading – especially at extreme scale or high-dimensional settings – and there is a delicate balance between casting things in terms of intuition and just trusting the mathematics. Indeed, from looking at the PDFs it is clear

that Student's t-distribution is actually *heavier* in the tails asymptotically than the normal distribution, since it decays as a power of x, whereas the density function of the normal distribution decays exponentially. More precisely, $f_\nu(x)$ decays like $x^{-(\nu+1)}$ as $|x| \to \infty$, whereas

$$-\log f_{\mathcal{N}(\mu,\sigma^2)}(x) \asymp x^2.$$

In particular, we can see that for $\nu \leq 2$, the cross-entropy $H(t_\nu, \mathcal{N}(\mu, \sigma^2))$ will *diverge*. Let's examine $\nu = 1$ more closely; in this case, the variance of t_ν diverges because the PDF decays like x^{-2}, but we can still compare it to the standard normal distribution.



It's not obvious from the picture, but the t-distribution has heavier tails than the normal distribution, leading to $H(t_1, \mathcal{N}(0,1)) = \infty$, while $H(\mathcal{N}(0,1), t_1) \approx 1.6782$. Intuitively, this is because the cross-entropy penalizes the tail events of the t-distribution, which the normal distribution "thinks" should almost never occur.

3.3.2. *Relation to KL divergence.* Let p, q be two probability distributions on the same outcome space. Recall that the KL divergence of p, q is

$$\mathrm{KL}(p,q) := \mathbb{E}_p[\log(p/q)] = H(p,q) - H(p).$$

One sees both the KL divergence and cross-entropy used in machine learning, often interchangeably. What is the difference between them? Recall that in logistic regression, p is the empirical distribution determined from data, and q is the model. Hence $\mathrm{KL}(p,q)$ differs from $H(p,q)$ by the constant $H(p) = -\sum p_i \log p_i$, which is independent of the model. Since the addition of this constant doesn't affect the gradient of the loss function, these two loss functions are completely equivalent for the purpose of backpropagation.

3.4. **Bayesian estimation.** Thus far we have discussed what are called "point" estimates: a numerical estimate for a parameter θ. According to the Bayesian perspective of statistics, one should really estimate a "posterior *distribution*" on θ by using the data $\mathcal{D}$ to calculate an update to the "prior distribution". Given a "prior distribution" $P(\theta)$, this can be done using Bayes' rule:

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D} \mid \theta) P(\theta)}{P(\mathcal{D})}.$$

Example 3.4.1. Suppose the Warriors play the Lakers 3 times in the NBA conference finals, and win 2 of them. If they play again, what is the probability that the Warriors will win?

This question does not have a single "correct" answer; we just have to propose a model and see how it performs. Let's model the situation by assuming that the outcome of each game is i.i.d. and that the Warriors have win probability $\theta \in [0,1]$; in other words, the outcomes of games independent samples from a Bernoulli distribution. In Example 3.3.3 we saw that the MLE of θ is then $\hat{\theta} = 2/3$.

Note that if the Warriors had won all 3 past games, the MLE would be $\hat{\theta} = 1$, or 100% chancing of winning Game 4. This may make you uncomfortable, indicating that you have *prior beliefs* about θ which are not reflected in this model.

Let's consider instead a Bayesian perspective. We have to start by choosing a prior distribution on θ, and a the most natural one to take seems to be the uniform distribution. Let $\mathcal{D}$ be the data that the Warriors won 2 of the first 3 games, and A the event that they win game 4. Then the prior distribution of $P(A) = \theta$ is uniform, so that its density function $p(\theta)$ is the indicator function of $[0,1]$. By Bayes' rule, the *posterior distribution* $P(A|\mathcal{D})$ has density function

$$p(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)p(\theta)}{P(\mathcal{D})}.$$

We have $P(\mathcal{D}|\theta) = \binom{3}{2}\theta^2(1-\theta)$, and then

$$P(\mathcal{D}) = \int_0^1 P(\mathcal{D}|\theta)d\theta = \binom{3}{2}\int_0^1 \theta^2(1-\theta)\,d\theta. \tag{3.4.1}$$

At this point, it is useful to remember that the *Beta distribution* $B(\alpha,\beta)$ is the probability distribution with density function proportional to $x^{\alpha-1}(1-x)^{\beta-1}$, and normalization constant determined by the formula[13]

$$\int_0^1 x^{\alpha-1}(1-x)^{\beta-1} = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

From this, we see that (3.4.1) comes out to $\binom{3}{2}\frac{2!1!}{4!} = \frac{1}{4}$. Thus

$$p(\theta|\mathcal{D}) = \begin{cases} 12\theta^2(1-\theta) & \theta \in [0,1], \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we can use this to calculate the probability that the Warriors win the next game under our model: it is

$$P(A|\mathcal{D}) = \int_0^1 P(A|\theta)p(\theta|\mathcal{D})\,d\theta = \int_0^1 \theta f(\theta|\mathcal{D})\,d\theta = 12\int_0^1 \theta^3(1-\theta) = 12\frac{3!}{5!} = \frac{3}{5}.$$

Remark 3.4.2. The θ that maximizing $P(\theta|\mathcal{D})$ is called the *maximum a posteriori* (MAP) estimate. In this example, the MAP estimate (with uniform prior) happens to agree with the MLE estimate 2/3, but it could have differed if the prior distribution was non-uniform.

[13]If α and β are not integers, then the interpretation of this formula is "$(\alpha-1)!$" $= \Gamma(\alpha)$.

3.4.1. *Relation to loss functions.* Suppose we have data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ which we are trying to model with a function $y = f_\theta(x)$. Consider the L^2-regularized loss function

$$L(\theta) = \sum_{i=1}^n (y_i - f_\theta(x_i))^2 + ||\theta||_2^2.$$

This can be interpreted in terms of MAP estimation with Gaussian prior and noise. Indeed, Bayes' rule says that the posterior density $p(\theta|\mathcal{D})$ is given by

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}.$$

Since $p(\mathcal{D})$ is a constant independent of θ, we can ignore it for the purposes of optimizing θ. Hence it is equivalent to find the $\hat{\theta}$ which minimizes

$$-\ln p(\theta|\mathcal{D}) = -\ln p(\mathcal{D}|\theta) + \ln p(\theta).$$

If we choose the prior distribution on $p(\theta)$ to be the product of independent mean-zero Gaussian distributions, then we have

$$-\ln p(\theta) = \sum_i \left(\frac{\theta_i^2}{2\sigma^2}\right)$$

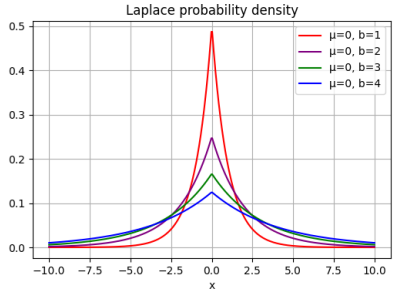
which reproduces the regularization term $||\theta||_2^2$. If we assume that the data $\{(x_i, y_i)\}_{i=1}^n$ is of the form $y_i = f_\theta(x_i) + \epsilon_i$ where the error terms ϵ_i are drawn i.i.d. from a Gaussian of mean 0, then the other term $-\ln p(\mathcal{D}|\theta)$ agrees (up to constants) with the MSE loss

$$\sum_i (y_i - f_\theta(x_i))^2.$$

Slogan 3.4.3. The L^2-regularized loss function solves for the MAP estimate for data which is drawn from a distribution with Gaussian errors, and with Gaussian prior on the parameters. (These are natural distributions to assume, by the universality of the Gaussian.)

Example 3.4.4. The *Laplace distribution* with mean μ and variance $2b^2$ has probability density function

$$f(x|\mu, b) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right)$$



By the same discussion, we see that L^1-regularization can be interpreted in terms of MAP estimation with a Laplacian prior distribution on weights.

3.5. **Evaluating estimators.** Let W be an estimator for a parameter θ.

- The *mean absolute error* (MAE) of W is $\mathbb{E}_\theta[|W - \theta|]$, viewed as a function of θ.
- The *mean squared error* (MSE) of W is $\mathbb{E}_\theta[(W-\theta)^2]$, viewed as a function of θ.

The MSE is somewhat preferred, as it is analytically convenient ($(W-\theta)^2$ is better to differentiate as a function of θ). It also has the benefit of being "interpretable" according to the decomposition

$$\begin{aligned}\mathbb{E}_\theta[(W-\theta)^2] &= \mathbb{E}_\theta[(W-\mathbb{E}_\theta[W]+\mathbb{E}_\theta[W]-\theta)]^2 = \mathbb{E}_\theta[(W-\mathbb{E}_\theta[W])^2] + (\mathbb{E}_\theta[W]-\theta)^2 \\ &= \mathrm{Var}_\theta(W) + \mathrm{Bias}_\theta(W)^2\end{aligned}$$

where we recall the notion of bias from Definition 3.2.4. The variance can be viewed as a measure of the precision versus noisiness of the estimator, while the bias can be viewed as a measure of accuracy.

Example 3.5.1. Let $X_1, \ldots, X_n$ be a random sample from a distribution with mean μ and variance σ^2. We saw in Example 3.2.5 that the sample mean $\overline{X} := \frac{1}{n}\sum_{i=1}^n X_i$ is an unbiased estimator of the mean μ of the X_i. Therefore, the MSE of this estimator is $\boxed{\mathrm{Var}[\overline{X}] = \frac{\sigma^2}{n}.}$

Example 3.5.2. Let $X_1, X_2, \ldots, X_n$ be a random sample from a distribution with mean μ and variance σ^2. We can simply take X_1 as an estimator for μ. Tautologically, this is unbiased, and tautologically its MSE is $\mathbb{E}[(X_1-\mu)^2] = \mathrm{Var}[X_1] = \sigma^2$. Compared to the sample mean, the variance is much higher, making it a "worse" estimator than the sample mean. However, it has a benefit: it is faster to compute, as we just need to draw a single sample. Therefore, we will use this estimator when training certain computationally intensive models, such as Diffusion Models, at scale.

Example 3.5.3. We move on to consider estimators of the variance. We saw in Example 3.2.5 that the sample variance

$$S^2 := \frac{1}{n-1}\sum_{i=1}^n (X_i - \overline{X})^2$$

is an unbiased estimator of the variance. Therefore, $\mathrm{MSE}(S^2) = \mathrm{Var}_{\sigma^2}(S^2)$. Through some tedious algebra, one can calculate this to be

$$\mathrm{MSE}(S^2) = \mathbb{E}[(S^2-\sigma^2)^2] = \frac{2\sigma^4}{n-1}.$$

Now suppose that the X_i are Gaussian. Then in Example 3.3.4 we saw that the MLE estimate is what we called in Example 3.2.5 the *naive sample variance*

$$S^2_{\text{naive}} = \frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2.$$

In particular, the MLE estimate is biased, with

$$\mathrm{Bias}(S^2_{\text{naive}})^2 = (\mathbb{E}[S^2_{\text{naive}}] - \sigma^2)^2 = \frac{\sigma^4}{n^2}.$$

As for the variance, we have

$$\mathrm{Var}(S^2_{\text{naive}}) = \left(\frac{n-1}{n}\right)^2 \mathrm{Var}(S^2) = \frac{2(n-1)\sigma^4}{n^2}.$$

Hence the MSE of S^2_{naive} is

$$\mathrm{MSE}(S^2_{\text{naive}}) = \frac{\sigma^4}{n^2} + \frac{2(n-1)\sigma^4}{n^2} = \frac{(2n-1)\sigma^4}{n^2}.$$

To summarize, we have seen that

$$\mathrm{MSE}(S^2_{\mathrm{naive}}) = \frac{(2n-1)\sigma^4}{n^2} < \frac{2\sigma^4}{n-1} = \mathrm{MSE}(S^2).$$

Thus the *biased* estimator given by MLE actually has a *lower* Mean Squared Error than the natural unbiased estimator. This illustrates the possibility of decreasing the MSE by trading off bias and variance.

4. Optimization

We gave an introduction to neural networks in §1, but the picture presented there was oversimplified compared to how neural networks are actually implemented in practice nowadays. For example:

- Various forms of regularization should be implemented in order to combat overfitting.
- Instead of gradient descent, weight updates are usually done with a momentum-based optimizer.
- The initial weights of the neural network should be normalized appropriately, and then during training and inference further normalization should be applied between layers.

We will go over some of these finer points in this lecture, with the goal of learning enough terms to understand research papers.

4.1. **Regularization.**

Recall the preliminary discussion of regularization from §1.7. In this section, we will look deeper at some common regularization methods.

4.1.1. *Weight decay.*

One interpretation of overfitting is that it is a result of excessive complexity of the model, possibly in response to inherent noise in data (Figure 4.1.1).

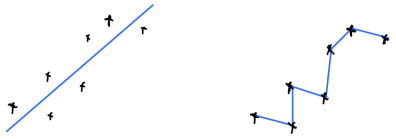


Figure 4.1.1. A cartoon of overfitting. Left: the data essentially follows a linear model, with some noise. Right: the data can be interpolated completely by a sufficiently complicated piecewise-linear model, which however will fail to generalize from training to test.

Therefore, it is natural to add a "penalty" term that measures the complexity of the weights. Mathematically, this means adjusting the unregularized loss function $\mathcal{L}^{\mathrm{unreg}}(W)$ to

$$\mathcal{L}(W) := \mathcal{L}^{\mathrm{unreg}}(W) + \lambda||W||$$

where λ is the *weight decay* coefficient (often around 10^{-4}) and $||W||$ is the norm of W (to be chosen). The most common choice is perhaps L^2 penalty, where we take the norm

$$||W|| = |W|_2^2 = \sum_{i,\ell} (W_i^\ell)^2.$$

Another common choice is L^1 penalty, where we take

$$||W|| = |W|_1 = \sum_{i,\ell} |W_i^\ell|.$$

Remark 4.1.1. With L^2-penalty, the loss gradient becomes

$$\nabla_W(\mathcal{L}) = \nabla_W(\mathcal{L}^{\text{unreg}} - \lambda||W||_2^2) = \nabla_W(\mathcal{L}^{\text{unreg}}) - 2\lambda W.$$

Therefore, the gradient update step is

$$W \leftarrow W - \eta\nabla_W(\mathcal{L}^{\text{unreg}}) = (1 - 2\eta\lambda)W - \eta\nabla_W(\mathcal{L}^{\text{unreg}}),$$

which can be interpreted as first decaying the weight by a scalar factor, and then implementing a usual gradient update step. This interpretation is behind the name "weight decay".

Example 4.1.2. Let's revisit the toy example of Linear Regression with L^2-regularization, following §1.3. The regularized loss function is

$$\mathcal{L}(W) = \underbrace{\frac{1}{n}(Y - XW)^\top(Y - XW)}_{\mathcal{L}^{\text{unreg}}(W)} + \lambda W^\top W.$$

Therefore, its gradient is

$$\nabla_W(\mathcal{L}) = \frac{-2}{n}X^\top(Y - XW) + 2\lambda W.$$

Setting this to 0, we find that the optimal weight $\hat{W}$ satisfies

$$(n\lambda\,\mathrm{Id} + X^\top X)\hat{W} = X^\top Y.$$

When $\lambda = 0$, we saw that this failed to have a unique solution when $X^\top X$ fails to be invertible, i.e., features are linearly dependent (Remark 1.3.2). For $\lambda > 0$, the matrix $(n\lambda\,\mathrm{Id} + X^\top X)$ is strictly positive definite hence always invertible, so we have

$$\hat{W} = (n\lambda\,\mathrm{Id} + X^\top X)^{-1}X^\top Y.$$

The stability of this solution is inversely proportional to the *smallest* eigenvalue of $n\lambda\,\mathrm{Id} + X^\top X$, since that controls the scale of its inverse. Therefore, we see that increasing λ makes $\hat{W}$ more stable with respect to fluctuations in X.

There is a geometric interpretation of weight decay as constrained optimization. The claim here is that the unconstrained minimization problem

$$\min_W \Big[\mathcal{L}^{\text{unreg}}(W) + \lambda||W||\Big] \tag{4.1.1}$$

can be related[14] to the *constrained* minimization problem

$$\min_W \mathcal{L}^{\text{unreg}}(W) \text{ such that } ||W|| \leq \tau \tag{4.1.2}$$

for some value of τ. This is because the Lagrangian form of (4.1.2) is

$$\min_W \Big[\mathcal{L}^{\text{unreg}}(W) - \lambda(||W|| - \tau)\Big], \tag{4.1.3}$$

which is essentially equivalent to (4.1.1).

[14] If $\mathcal{L}^{\text{unreg}}$ is convex (rarely true in practice), then there is a precise equivalence of optimization problems. For non-convex functions, the relation between optimization problems is heuristic.

Example 4.1.3. Consider the special case of linear regression with $p = 2$ features, so the unregularized Lagrangian is

$$\mathcal{L}^{\text{unreg}}(w_1, w_2) = \sum_i (y^{(i)} - (w_1 x_1^{(i)} + w_2 x_2^{(i)}))^2.$$

For L^2 weight decay, we are trying to minimize a loss function of the form

$$\mathcal{L}(w_1, w_2) = \mathcal{L}^{\text{unreg}}(w_1, w_2) + \lambda(w_1^2 + w_2^2).$$

In the constrained version of this problem (4.1.2), the constraint is a circle in the w_1, w_2 plane and the level sets of $\mathcal{L}^{\text{unreg}}(w_1, w_2)$ are ellipses. The optimal solution is the point at which the circular constraint is tangent to an elliptical level set of $\mathcal{L}^{\text{unreg}}(w_1, w_2)$; see Figure 4.1.2.

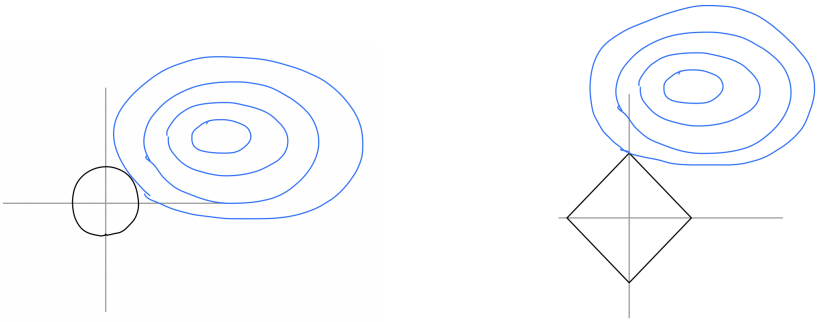


FIGURE 4.1.2. Weight decay interpreted as constrained optimization, with constraint shown in black and level sets of $\mathcal{L}^{\text{unreg}}$ shown in blue. Left: L^2 regularization, right: L^1 regularization.

For L^1 weight decay, the constraint instead defines a square, with extreme points having all but one coordinate equal to 0. Therefore, a non-extreme point cannot be a point of tangency with an ellipse unless that ellipse is in special position, so generically the optimum will occur at one of these extreme points.

The discussion of Example 4.1.3 applies as well in higher dimensions. In particular, the extreme points of the L^1 constraint are the vertices of a hypercube with all but one coordinate equal to 0, and "generically" the optimum occurs at one of these extreme points. This leads to the following intuition.

Slogan 4.1.4. Weight decay with L^1 penalty encourages *sparsity*.

4.1.2. *Dropout.* Another intuition above overfitting is that it involves "memorization" of training data (mathematically interpreted as overly complicated casework), involving complex co-adaptation of neurons. To mitigate this effect, Hinton et al [HSK+12] introduced a regularization scheme called *dropout regularization*, whose idea is to randomly eliminate certain neurons during training. The intuition is that this prunes overly complex interdependencies. Another perspective on dropout is that it is an efficient method of "model averaging", i.e., drawing an averaged prediction from many different models.

Dropout is implemented by introducing a "masking vector" m^ℓ for each layer. If the pre-activation value is $z^\ell = W^\ell a^{\ell-1} + b^\ell$, then the post-activation value *with dropout* is

$$a^\ell = m^\ell \odot h(z^\ell).$$

Here we again used the Hadamard product $\odot$, meaning two vectors of the same dimension are multiplied entrywise. During backpropagation, the errors are also propagated through the mask, so that the update steps do not affect a masked neuron.

The masking vector m^ℓ is typically constructed by making each entry a Bernoulli random variable with drop probability $1-p$. (The value used in the original paper [HSK+12] is $p = 0.5$.) At test and inference time, no dropout is applied. To compensate for the resulting statistical shift, the activation vectors a^ℓ should then be scaled, so that at inference time $a^\ell = ph(z^\ell)$. Alternatively, this can be baked into training, by defining $a^\ell = \frac{1}{p} m^\ell \odot h(z^\ell)$, so that $\mathbb{E}[a^\ell] = \frac{1}{p} p\mathbb{E}[h(z^\ell)] = \mathbb{E}[h(z^\ell)]$. Then no adjustments are needed at inference time, where $p = 1$. This is called "inverted dropout" and is implemented under the hood in TensorFlow/Pytorch.

4.2. **Optimizers.** In §1.5.2, we explained a naive version of the gradient descent algorithm

$$W \leftarrow W - \eta \nabla_W(\mathcal{L})$$

where η is the learning rate. In practice, one uses an *optimizer* that implements a subtler versions of this algorithm. The dominant optimizer at the time of this writing is Adam (which stands for *adaptive momentum*), which we will build up to in two steps.

4.2.1. *Momentum.* First, we modify gradient descent by including a *momentum* term.

Slogan 4.2.1. The idea behind momentum is maintain memory of past gradients, and update according to a "time-discounted" aggregate of all past gradients.

Mathematically, we track the time step t of the update, and let $W^{(t)}$ be the weights at time t. The training update then has the form

$$W^{(t)} = W^{(t-1)} - \eta v^{(t)},$$

where $v^{(t)}$ is a recursive modification of the gradient:

$$v^{(t)} = \nabla_{W^{(t-1)}}(\mathcal{L}) + \mu v^{(t-1)}. \tag{4.2.1}$$

Here μ is a fixed *momentum hyperparameter*, which is often something like 0.9 in practice. The v is meant to stand for "velocity"; the notation evokes some physics intuition, but really various physics notions (velocity, momentum, mass) are mixed up here. To get a sense of what this is doing, we can unroll the recursion (4.2.1) to

$$v^{(t)} = \nabla_{W^{(t-1)}}(\mathcal{L}) + \mu \nabla_{W^{(t-2)}}(\mathcal{L}) + \mu^2 \nabla_{W^{(t-3)}}(\mathcal{L}) + \dots.$$

In other words, the momentum update is a "time-discounted" aggregate of all past gradients. (This should be normalized, which we will do in Adam.)

Intuitively, momentum addresses oscillatory convergence to the solution in convergence to a local loss optimum (Figure 4.2.1).

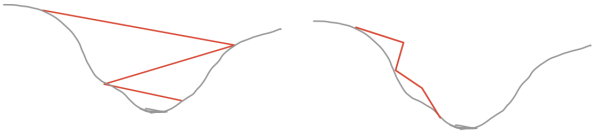


FIGURE 4.2.1. A cartoon illustrating the possible benefit of momentum. Left: gradient descent can lead to oscillatory behavior, hence poor training convergence. Right: momentum can counteract oscillation.

4.2.2. *Quadratic toy model.* Is it common to take a quadratic potential $\mathcal{L}(W) = \frac{1}{2}W^\top AW$ as a toy model (since it is easy to learn linear transformations, we allow ourselves to eliminate linear terms). Then the gradient is $\frac{1}{2}(A + A^\top)W$; let's assume that A is symmetric so that we can simplify this as AW. Then the gradient descent step is

$$W \leftarrow W - \eta AW = (\mathrm{Id} - \eta A)W.$$

To think about the dynamics of convergence, let u_i be the eigenvectors of A, with associated eigenvalues λ_i. In these terms, a gradient descent update has the form

$$\sum c_i u_i \mapsto \sum c_i(1 - \eta\lambda_i)u_i.$$

This converges if $|1 - \eta\lambda_i| < 1$, which happens when

$$0 < \lambda_i < \frac{2}{\eta}.$$

This gives a toy model of the dynamics of gradient descent. Imagine A is positive definite, so the (global in this case, but local in general) minimum is at 0. If the learning rate η is too large, then the updates will actually push away from the local minimum. Thus one wants η to be small compared to the eigenvalues of the Hessian matrix, but if η is too small then the convergence along the eigenvector with smallest eigenvalue will be extremely slow. The *condition number* $\kappa := \lambda_{\max}/\lambda_{\min}$ gives some measure of how unstable the convergence is, with a small value (i.e., close to 1) indicating the possibility of relatively robust convergence.

Remark 4.2.2. We can see in this case that the smaller the condition number, the more robust the convergence. In general, we say that a problem is "ill-conditioned" if small changes in input data cause large changes in solution, and "well-conditioned" if the solution is relatively stable under perturbations to the inputs. As this toy example shows, the conditioning is governed by the condition number of the Hessian matrix. Being well-conditioned is important for stability of training.

Now let's add momentum into the picture. Then the dynamics are governed by the equations

$$v^{(t+1)} = \mu v^{(t)} + \nabla_W(\mathcal{L}) = \mu v^{(t)} + AW^{(t)}.$$

and

$$W^{(t+1)} = W^{(t)} - \eta v^{(t+1)} = (\mathrm{Id} - \eta A)W^{(t)} - \eta\mu v^{(t)}$$

Hence, for each eigenvector u_i, the corresponding components (v_i, W_i) of $v^{(t)}$ and $W^{(t)}$ evolve according to the matrix

$$\begin{pmatrix} \mu & -\eta\mu \\ \lambda_i & 1 - \eta\lambda_i \end{pmatrix} \tag{4.2.2}$$

These are the equations that govern a damped harmonic oscillator, with a_i being the "velocity" and b_i being the "position". In this comparison, μ is the damping coefficient and λ_i is the energy from an external field.

Going through the math, one finds that this effectively doubles the range of learning rates that lead to convergence, an admittedly small effect. This is not believed to be the main significance of momentum in practice. Rather, it comes from the improvement in convergence *rate*. Indeed, when iterating the matrix (4.2.2), the convergence is governed by the eigenvalues. The best convergence rate occurs when the eigenvalues are equal. The trace of (4.2.2) is $\mu + 1 - \eta\lambda_i$ and the determinant is μ, so this critical convergence rate occurs when

$$(\mu + 1 - \eta\lambda_i)^2 = 4\mu.$$

Taking the square root of this equation, we can rearrange it as $(\sqrt{\mu}-1)^2 = \eta\lambda_i$, which leads to $\mu = (1-\sqrt{\eta\lambda_i})^2$, with convergence rate $1-\sqrt{\eta\lambda_i}$. The idea is that if $\eta\lambda_i$ is quite small, then $1-\sqrt{\eta\lambda_i} \approx \sqrt{1-\eta\lambda_i}$ improves the convergence rate by a square root.

This mathematics gives a guideline for choosing the μ and η hyperparameters in practice. If the problem is well-conditioned (i.e., the condition number κ is close to 1), then the convergence is robust. If the problem is ill-conditioned, then choose μ close to 1, and then find the largest η that converges. Usually, people start by trying something like $\mu = 0.99$ or 0.999, and $\eta = 0.01$.

For more in-depth discussion along these lines, see `https://distill.pub/2017/momentum/`.

4.2.3. *RMSProp.* The second modification comes from *Root Mean Square Propagation* (RMSProp).

Slogan 4.2.3. The idea behind RMSProp is to adapt the learning rate should adapt to the different directions of the loss gradient.

Mathematically, RMSProp implements this idea by updating less in directions that have already been updated a lot. It does so by keeping track of a "time-discounted" root mean square of past gradients in each direction. Fix a layer ℓ for concreteness, and abbreviate the ith coordinate of its weight matrix at time t as $W_i^{(t)}$. Then we keep track of a "discounted root mean square in direction i" $r_i^{(t)}$ at update step t, determined recursively by the formula

$$r_i^{(t)} = \beta r_i^{(t-1)} + (1-\beta)\left(\nabla_{W_i^{(t-1)}}(\mathcal{L})\right)^2. \tag{4.2.3}$$

The root mean square is then used to normalize the gradient update, so that the update step has the form

$$W_i^{(t)} = W_i^{(t-1)} - \frac{\eta}{\sqrt{r_i^{(t)}+\epsilon}}\nabla_{W_i^{(t-1)}}(\mathcal{L}).$$

Here one includes a $+\epsilon$ in the denominator for numerical stability reasons (to avoid division by 0). The parameter β is a "momentum parameter for the RMS", and a typical value might be 0.99. This effectively means that the learning rate adapts to the different dimensions of the gradient.

Remark 4.2.4. The equation (4.2.3) is trying to weight $r_i^{(t)}$ so that for each time step, the values $r_i^{(t)}$ for varying t are about equal in expectation, at least assuming that the $\left(\nabla_{W_i^{(t)}}(\mathcal{L})\right)^2$ are approximately equal in expectation. However, there is a question of initalization of $r_i^{(t=0)}$, which is usually taken to be 0. This means that in fact, $r_i^{(t)}$ is "underweighted" by a factor of $1-\beta^t$, and so one should really divide it by $1-\beta^t$.

4.2.4. *Adam.* Adam (short for "adaptive momentum") is essentially momentum plus RMSProp. The algorithm to find the weights $W_i^{(t)}$ at time t are given below.[15]

(1) We compute the gradient $\nabla_{W_i^{(t-1)}}(\mathcal{L})$ with respect to the weights at time $t-1$.
(2) We recursively calculate the momentum term

$$v_i^{(t)} = \mu v_i^{(t-1)} + (1-\mu)\nabla_{W_i^{(t-1)}}(\mathcal{L})$$

where $v_i^{(0)} = 0$.

[15]What we call μ and β are often called β_1 and β_2, respectively, in the literature.

(3) We recursively calculate the RMS term

$$r_i^{(t)} = \beta r_i^{(t-1)} + (1-\beta)\left(\nabla_{W_i^{(t-1)}}(\mathcal{L})\right)^2$$

where $r_i^{(0)} = 0$.

(4) Following Remark 4.2.4, we renormalize it as

$$\widetilde{v}_i^{(t)} = \frac{1}{1-\mu^t} v_i^{(t)} \quad \text{and} \quad \widetilde{r}_i^{(t)} = \frac{1}{1-\beta^t} r_i^{(t)}.$$

(5) Update

$$W_i^{(t)} = W_i^{(t-1)} - \eta \frac{\widetilde{v}_i^{(t)}}{\sqrt{\widetilde{r}_i^{(t)}} + \epsilon}. \tag{4.2.4}$$

Remark 4.2.5. The algorithm just described corresponds to Adam without weight decay. In practice, one would also incorporate weight decay. The original Adam algorithm did this by incorporating the L^2 regularization term into the loss function, which means the weight decay term is scaled by the adaptive learning rate $\eta/(\sqrt{\widetilde{r}_i^{(t)}} + \epsilon)$, effectively coupling the decay rate to the historical gradient magnitudes.

Nowadays, one uses a variant called AdamW which decouples the weight decay from the adaptive learning rate. In view of Remark 4.1.1, this is implemented by simply changing the update (4.2.4) to

$$W_i^{(t)} = (1-\eta\lambda) W_i^{(t-1)} - \frac{\eta}{\sqrt{\widetilde{r}_i^{(t)}} + \epsilon} \widetilde{v}_i^{(t)}. \tag{4.2.5}$$

4.3. **Normalization.** Backpropagation through a deep network requires a deep application of the chain rule. This can cause the formation of gradients to be unstable. For example, multiplying a long chain of large derivatives leads to the "exploding gradients problem", while multiplying a long chain of small derivatives leads to the "vanishing gradients problem". Hence for stability, normalization of data is important. This includes:

- Preprocessing input data. (We have already discussed this in Example 1.5.3.)
- Initialization of neural network weights.
- Normalization of inputs to hidden layers.

4.3.1. *Initialization of weights.* At the start of training, weights are initialized randomly.[16] Even for this, there is some methodology on how to initialize correctly. We will explain one commonly used method, named "Kaiming initialization" or "He initialization" after [HZRS15b]. It is easy enough to explain: the entries of the weights to a node should be drawn from a Gaussian distribution $\mathcal{N}(0, 2/n)$ where n is the number of inputs to that node.

This assumption is forced by stability of the first and second moments, if one assumes that:

- The inputs to each node in the first layer are i.i.d. mean 0 and variance σ_x^2.
- The weights are all i.i.d. with mean 0 and variance σ_W^2.
- Activation functions are ReLU.

[16]For reproducibility purposes, it is common to occasionally record and fix the random seeds used in initialization. Even so, GPU scheduling and floating point precision limitations introduce certain randomness to the training process, which can magnify into substantial issues at huge scales.

The idea is then that the mean and variance should be tuned so that distributions remain stable. For the first layer $a = \mathrm{ReLU}(xW)$, the coordinates are given by

$$a_i = \mathrm{ReLU}\left(\sum_{j=1}^{n} x_j W_{ij}\right).$$

By additivity of variance over sums of independent random variables, and multiplicativity over products of independent random variables, we have

$$\mathrm{Var}\Big[\sum_{j=1}^{n} x_j W_{ij}\Big] = \sum_{j=1}^{n} \mathrm{Var}[x_j]\,\mathrm{Var}[W_{ij}] = n\sigma_x^2\sigma_W^2. \tag{4.3.1}$$

As x_j, W_{ij}, and $\sum_{j=1}^{n} x_j W_{ij}$ all have mean 0, (4.3.1) says equivalently that

$$\mathbb{E}\Big[\Big(\sum_{j=1}^{n} x_j W_{ij}\Big)^2\Big] = n\sigma_x^2\sigma_W^2. \tag{4.3.2}$$

Since the density of $z_i := \sum_{j=1}^{n} x_j W_{ij}$ is symmetric about 0, $\mathbb{E}[\mathrm{ReLU}(z_i)^2] = \frac{1}{2}\mathbb{E}[z_i^2]$. Thus, we have

$$\mathbb{E}[a_i^2] = \frac{n}{2}\sigma_x^2\sigma_W^2.$$

If we want this to be stable through to the next layer, then σ_x and σ_W must satisfy

$$\sigma_x^2 = \frac{n}{2}\sigma_x^2\sigma_W^2$$

which forces $\boxed{\sigma_W = \sqrt{2/n}}$.

Although this derivation was universal, in the original paper [HZRS15b] the weights were taken to be *normally distributed*, which would then mean that they are drawn from distribution $\mathcal{N}(0, 2/n)$ where again n is the number of input nodes.

Exercise 4.3.1. This derivation was based on the stability of the second moment. Arguably, it would be more natural to seek stability of the variance. For normally distributed x and W, show that the corresponding normalization factor is $n\left(\frac{1}{2} - \frac{1}{2\pi}\right)$. As far as I know, this is not used in practice, possibly out of preference for the simpler answer coming from Kaiming normalization.

Remark 4.3.2. Recall from §3.4.1 that we saw the L^2 regularization arising naturally from a prior distribution on weights of the neural network, as being Gaussian with mean 0 and the *same* variance. Kaiming normalization (§4.3.1) suggests that we should actually refine the regularization term so that the weights of different neurons are multiplied by a factor proportional to $n/2$, where n is the number of inputs to the neuron. As far as I know, this kind of adaptation is not used in practice.

4.3.2. *Batch normalization.* Ioffe and Szegedy [IS15] introduced a normalization method called *batch normalization*. We have already discussed the need to normalize input data; the idea of batch normalization is to normalize the pre-activation inputs to a neuron over a batch of training data.

Let the inputs to a given neuron be $z_1, \ldots, z_B$. Define

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i, \quad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2.$$

Then we normalize

$$\widetilde{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where ϵ is a small constant added for numerical stability of division.

So far, these operations force the batch to have mean 0 and variance approximately 1, regardless of the neuron. If we stopped here, then we would be imposing artificial constraints on the neural network. Supposing for example that the activation function is ReLU; then normalizing $\widetilde{z}_i$ to have mean 0 means that one is artifically forcing about half of the neurons to activate upon applying the linearity to $\widetilde{z}_i$. (In actuality, large neural networks such as LLMs tend to see sparse activation, with substantially less than half of neurons firing on any given input.)

Therefore, we further scale and shift $\widetilde{z}_i \leftarrow \gamma\widetilde{z}_i + \beta$ where γ and β are learned parameters (which do *not* depend on i). Finally, the values $\widetilde{z}_i$ are fed into the neuron's activation function. This perhaps unintuitive step makes Batch Normalization less obvious than it might at first seem. Indeed, at first glance, it seems that this transformation will simply undo the normalization procedure. However, this is not quite the case:

- The parameters γ and β are fixed for the model itself, not for each specific batch.
- By encoding these parameters explicitly as learnable parameters, the neural network acquires more direct access to them. This can be thought of as a kind of feature engineering, where we make important features explicit, so that the model can train more efficiently.

When [IS15] was introduced, it pitched Batch Normalization as a method to reduce "internal covariate shift". In statistics, covariate shift refers to a shift in input distribution, especially between testing and training inputs. Here, however, "internal covariate shift" refers to the fact that *during training*, the distribution of inputs to a layer are changed by updates to previous layers. This is especially a problem for very deep networks, where small changes to one layer can compound through the network. In practice, Batch Normalization allows training to use higher learning rates (and thus complete faster) and be less sensitive to initialization.

At test and inference time, one encounters the problem that batches are no longer available, and normalization does not make sense for individual inputs. To deal with this, one computes while training some running average of the mean and variance over all batches, and uses those figures at inference time.

4.3.3. *Layer normalization.* Batch normalization is primarily used in fully connected layers or convolutional layers. There are certain situations where one cannot batch data, so Batch Normalization does not apply. For example, *autoregressive* models make predictions depending on their own previous outputs, and *online* models must serve responses immediately; batch normalization cannot be used in either case. *Layer normalization* was invented by Ba et al [BKH16] to work in such situations (specifically for recurrent neural networks, but it is now also used in Transformers); it should be thought of as an orthogonal normalization scheme. Figure 4.3.1 depicts batch versus layer normalization.

The idea is instead to average activations over all neurons in a layer, for a single input. To apply layer normalization, one calculates the mean and variance of the outputs from a given layer ℓ,

$$\mu^\ell = \frac{1}{N^\ell}\sum_{i=1}^{N^\ell} a_i^\ell, \quad \text{and} \quad (\sigma^\ell)^2 = \frac{1}{N^\ell}\sum_{i=1}^{N^\ell}(a_i^\ell - \mu^\ell)^2.$$

Then we renormalize the outputs as

$$\widetilde{a}_i^\ell = \frac{a_i^\ell - \mu^\ell}{\sqrt{(\sigma^\ell)^2 + \epsilon}}$$

and finally introduce a scale and shift

$$\widetilde{a}_i^\ell \leftarrow \gamma^\ell \widetilde{a}_i^\ell + \beta^\ell$$

by learnable parameters γ^ℓ and β^ℓ, as in batch normalization.

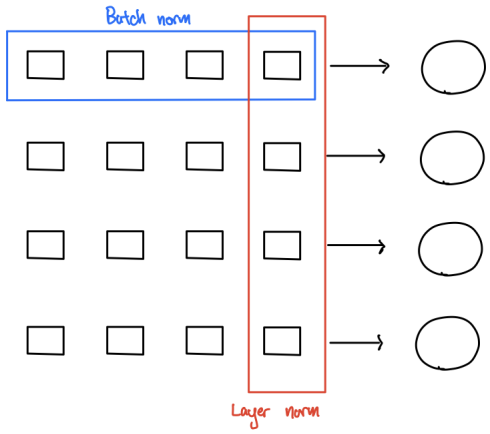
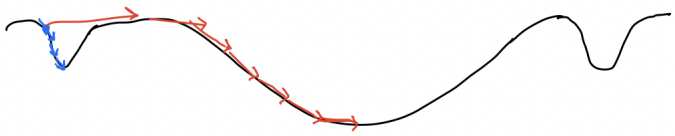


FIGURE 4.3.1. Batch normalization averages, for each neuron, over a batch of inputs to that neuron. Layer normalization averages, for each layer, over other inputs to that layer.

4.4. **Batch size.** What is the optimal batch size for training? The purpose of batching is to dampen the noise in data, and for that purpose increasing batch size is desirable. But researchers have historically believed (see [LBOM12, §4]), based on empirical observation, that training on large batches is *worse* for generalization. For example, [KMN+16, Table 2] documents experiments in which the training loss of large-batch models is comparable to that of small-batch models, while the test loss is noticeably worse.

An explanation proposed in LeCun et al [LBOM12] is that noise is actually useful for exploring the loss landscape: without noise, the parameters would explore only nearby local minima, but noise allows the parameters to bounce around to better minima.



A more precise hypothesis is made by Keskar et al [KMN+16], which tries to quantify the "sharpness" of local minima. To be clear, the notion of "sharp minima" had been studied before, and quantified in terms of the eigenvalues of the Hessian matrix $\nabla^2(\mathcal{L})$. Large eigenvalues of the Hessian imply that there are directions along which $\mathcal{L}$ increases very quickly. The intuition of *loc. cit.* is that noisy gradient updates can escape sharp minima,

hence training with large batches can get trapped in sharp minima while training with smaller batches tends to land in "flat" minima.

In practice, computing the Hessian efficiently is computationally costly, especially in very high-dimensional spaces. To test their thesis, Keskar et al defines a more empirical measure of sharpness, given by the rate of increase of $\mathcal{L}$ along small perturbations along random low-dimensional subspaces. They then gives examples where large-batch training lands in sharper minima, according to this metric. Specifically, "large-batch" means 10% of training data (in their experiments, ranging from 5000 to ≈ 70000), while "small-batch" means 256. Training was continued until validation loss plateaued.

However, it is worth noting that the thesis of [KMN+16] has been challenged by some subsequent work. Indeed, the paper [KMN+16] did not control for the fact that with large batches, the number of *updates* is smaller during the same number of epochs. According to [HHS17], the generalization gap for large-batch training can be reduced to a level comparable to small-batch training by appropriately adjusting the learning rate and the number of weight updates.

4.5. **Boosting and bagging.** There are two more "macroscopic" methods to improve the performance of models. The more naive is *bagging*, which is essentially aggregating (by averaging in the case of regression, or voting in the case of classification) over the predictions of N models trained in parallel. Mathematically, this replaces a single prediction with the sample mean of N different predictions, which could in theory reduce the MSE *if* the models are sufficiently uncorrelated. Compared to making N predictions from the same model, the hope is that different models have less correlated error; however, this comes at the cost of increasing training N-fold.

A second method is *boosting*, which was introduced by Freund–Schapire in 1997 [FS97] in the form of the AdaBoost ("adaptive boosting") algorithm. This involves training a *sequence of models*, with the failure cases of each being weighted more heavily in the training set for the next model. Boosting thus creates a "committee" of models with different specializations, which votes by weighted majority at inference time. The error of each model during training is used to weight that model's vote. Both theoretical and empirical results show that boosting can combine weak learners into a strong model.

Part 2. Architectures

5. Convolutional Neural Networks

In this lecture, we will survey *convolutional neural networks* (CNNs), an effective tool for problems related to *computer vision*. Notable applications include Image Classification (e.g., parsing handwriting) and Object Detection (e.g., detecting pedestrians in an image taken by an autonomous vehicle).

The main pedagogical reason for covering CNNs is to introduce three important concepts that arose out of their development:

(1) The importance of *depth* in neural networks. The intuition is that the compositional nature of depth (as opposed to width) expands expressive capacity. Historically, the success of deep CNNs on computer vision problems, starting with AlexNet [KSH12] in 2012, was a major impetus in revitalizing the study of neural networks. Very recently, transformer-based or hybrid transformer-CNN models have matched and even surpassed CNNs in image classification. However, CNNs are still competitive in data or compute constrained scenarios.

(2) The importance of *residual learning* (He et al) when training deep neural networks.
(3) The idea of incorporating *inductive bias* into neural network architectures. Inductive bias refers to assumptions about the nature of the relevant problem, including structural features of the data. Inductive bias can be implemented mathematically by restricting to a function class with specific properties, such as symmetry or sparsity, which one expects to hold in the problem. For an example that we have already met, one could view regularization as reflecting a form of inductive bias towards simple solutions (Occam's Razor).

5.1. **Image data.** To motivate the setup of CNNs, let's think about what image data looks like. A typical picture is encoded as a matrix of pixels, with each pixel having a numerical color representation. The length and width of the pixels are usually on the order of 10^2 to 10^3, and the colors are encoded by color scheme is encoded by three 8-bit numbers, expressing a Red/Green/Blue intensity from 0 to 255. For concreteness: a single 224×224 RGB picture is encoded by a vector of $\approx 150,000$ numbers. Thus, *image data is extremely high-dimensional*. A single dense hidden layer, with the same number of parameters, connected to the input data would by itself carry a massive number of parameters, on the order of 10^{10}.

At the same time, image data is *highly structured*, and CNNs are designed to exploit this. For example, nearby pixels tend to be strongly correlated to other, while distant pixels are essentially unrelated. We will highlight four properties of images in particular, that we will build into CNNs.

- *Hierarchical structure.* The meaning of images tends to be compiled in a hierarchical fashion, with high-level features being assembled from low-level ones. For example, a human face is assembled from eyes plus nose and mouth, and an eye is assembled from an iris and a pupil, etc.
- *Locality.* Features are formed out of clusters of *nearby* pixels in the image.
- *Translational equivariance.* The semantic meaning of a feature tends to be preserved by translation. For example, if a cluster of pixels represents a dog, then that same cluster of pixels would represent a dog if translated elsewhere in the image.
- *Local invariance.* The semantic meaning of a feature tends to be preserved under local perturbation.

The philosophy behind CNNs is to design an architecture that builds the high-level inductive bias into the model "by hand", while letting the parameters be learned by data. (While this seems like an obvious philosophy, we mention again that the recent trend with LLMs is to use a more flexible architecture that builds in less inductive bias, and learns almost everything from the data.)

5.2. **CNN architecture.** A CNN is a neural network with "convolutional layers" that have a specific form, in which each neuron is connected to only a small number of neurons in the preceding layer, and furthermore *weights are shared* across neurons in the same layer.

5.2.1. *Input/output format.* The topology of the connections is prescribed in terms of the format of the input. In a CNN, the input to each layer is a tensor of shape $h \times w \times c$, which we think of as a 2d grid of dimension $h \times w$ ("height" by "width"), and c consists of the "channels".

Example 5.2.1. For $\ell = 1$, the input is the original image. In a typical picture setup, $h \times w$ is the number of pixels in the image, and c is the number of colors.

The dimensions h and w decrease through the CNN, while the dimension c increases. For example, for standard RGB pixel pictures, there are initially $c = 3$ channels, while h and w are fairly large (e.g., 10^2 to 10^3). Towards the end of the CNN, h and w might be on the order of 10 while c might be on the order of 10^3.

5.2.2. *Convolutional layers.* A convolutional neural network fits into the general format described in §1, so each layer computes a function of the form

$$a^{\ell-1} \mapsto \mathrm{ReLU}(a^{\ell-1}W^\ell + b).$$

The distinctive feature of a CNN is that it includes many *convolutional layers*, where the specific form of the weight matrix W^ℓ is very specific. Far from being fully connected, in these layers neurons connect to a small $n \times n$ *receptive field* in the input array, where n is typically a small odd number such as $n = 3$ or $n = 5$.[17] See Figure 5.2.1 for an example.

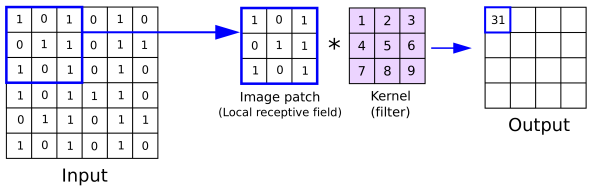


FIGURE 5.2.1. Example of a feature map in a CNN, from `https://anhreynolds.com/blogs/cnn.html`.

In each layer, the learnable parameters are the entries of an $n \times n$ weight matrix K, called the *filter* or *kernel*. Crucially, *the filter K is shared across neurons in the same layer*. The pre-activation value of the feature map is given by multiplying each of the receptive field by the corresponding entry of the filter, and then summing:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(\text{receptive field})_{ij}K_{ij}.$$

Remark 5.2.2. If we call $A^{\ell-1}$ (a square matrix) the input to layer ℓ for the chosen channels, then the corresponding pre-activation matrix Z^ℓ has

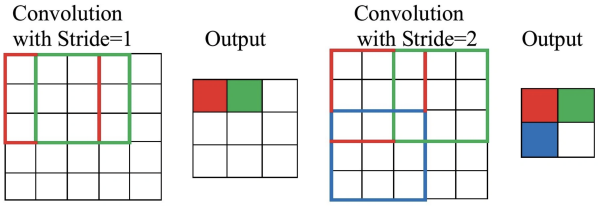
$$Z^\ell_{x,y} = \sum_{i=-\frac{n-1}{2}}^{\frac{n-1}{2}}\sum_{j=-\frac{n-1}{2}}^{\frac{n-1}{2}} A^{\ell-1}_{x+i,y+j}K_{ij}.$$

If we reparametrize the indices, then this looks like the formula for convolution, which is where the name "convolutional neural network" comes from. Strictly speaking, however, it is *not* actually convolution in the mathematical sense.

To calculate all the outputs of the layer, the receptive field is shifted by an amount called the *stride*, and the analogous calculation is applied to each. Typically, CNNs use a stride of

[17]The preference for odd sidelengths is so that there the filter is symmetric around a central pixel. Another reason is to ensure that *same padding* (§5.2.6) is possible.

1 in the feature maps.



The hidden convolutional layer is also called a *feature map*, intuitively because the neurons represent features.

Example 5.2.3. The neuron activates if the receptive field is "close enough" to the filter. For example, the filter

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

can be interpreted as a feature that detects vertical edges. See Figure 5.2.2 for an example.

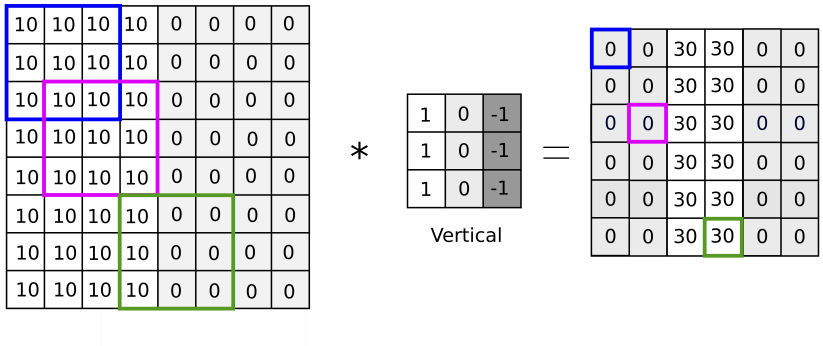


FIGURE 5.2.2. Example of a filter detecting vertical edges, from `https://anhreynolds.com/blogs/cnn.html`.

Let's examine how the convolutional layers solve some of the problems discused in §5.1. Firstly, the small receptive fields express the inductive bias of locality. They also reduce the dimension of the model.

A priori, the filters could have depended on the neuron, but as we mentioned, the filters are *shared* across all neurons in a given layer. This again reduces the dimension of the weights. Furthermore, it captures the inductive bias of translational equivariance by making the feature map equivariant with respect to translations.

5.2.3. *Multiple channels.* So far, we were focusing for simplicity on the case where there is a single numerical value attached to each practice. In practice, there would typically be multiple values, for example 3 RGB bytes in an image. To handle these, we break up each the values into *channels*, and then treat each separately with independent filters. For example, when processing an RGB image we can use three filters to separately process the bytes for each color intensity.

Each channel has one filter, and represents one feature (thus is analogous to a single neuron in a standard neural network), such as the presence of vertical lines (Example 5.2.3). In a real CNN, we would have many channels. Thus, a convolutional layer has dimension $h \times w \times c_{\text{IN}} \times c_{\text{OUT}}$ where c_{IN} is the number of *input* channels, and c_{OUT} is the number of output channels. Typically, the spatial dimensions h and w decrease through a CNN network, while the feature parameters $c_?$ increase.

Example 5.2.4. It is meaningful to have a convolutional layer with filter size 1×1. The purpose of such a layer is to apply a function to the channels. In particular, it allows to alter (typically, to reduce) the number of channels.

5.2.4. *Pooling layers.* A CNN weaves in a *pooling layer* after every few convolutional layers. Pooling layers implement a kind of aggregation over local pixels that captures the inductive bias of local invariance under perturbations.

Structurally, pooling layers look like convolutional layers, but with the filters are fixed beforehand (so there are no learnable parameters in the pooling layers). The other differences are:

- The feature maps are not necessarily given by convolution with a kernel matrix.
- The stride is generally taken to be equal to the dimension l of the filter, so that the receptive fields are disjoint. In particular, pooling scales the length and width down by a factor of l. Thus, pooling implements a form of *downsampling* (i.e., dimension reduction).

Example 5.2.5. Perhaps the most common form of pooling is *max pooling*, which takes the maximum of the input values in the receptive field. Another example is *average pooling*, which takes the average of the input values in the receptive field; see Figure 5.2.3.

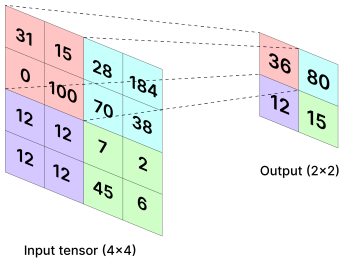


FIGURE 5.2.3. Example of an average pooling filter, from `iq.opengenus.org`.

Typically, a pooling layer is followed by a convolutional layer that *upsamples* by increasing the dimension by a factor of l. The net effect of these two layers is to multiply the dimension by $\frac{1}{l^2} l = \frac{1}{l}$.

5.2.5. *Fully connected ending.* CNNs end with a few fully connected layers, once the channel dimension is small enough for this to be feasible.

To summarize, the following is a typical CNN structure.

(1) 2-3 convolutional layers followed by a pooling layer.
(2) Repeat (1) several times.
(3) Conclude with 2-3 fully connected layers. These "flatten" the features from a tensor to a vector.

Let us also take stock of how the inductive biases in §5.1 are accounted for in this process.

- *Locality* is imposed by the small receptive fields.
- *Translational equivariance* is imposed by filter sharing between neurons in the same layer.
- *Local invariance* is imposed by pooling layers.
- *Hierarchical structure* is imposed by the deep structure of the neural network.

5.2.6. *Padding.* A mathematically trivial, but in practice important, aspect of filters is "padding". If the filter is of size l, then an $h \times w$ image goes to $(h-l+1) \times (w-l+1)$. This causes the size to decrease considerably when composing many feature maps in sequence. *Padding* refers to the introduction of auxiliary pixels so that the feature map is larger.

The most common kind of padding is *same padding*, which arranges for the feature map to have the *same size* as its input. In order to treat the boundary symmetrically, one pads each edge of the boundary with padding of the same size p. Hence for same padding, one wants $h + 2p - l + 1 = h$, or $p = \frac{l-1}{2}$, which requires l to be odd.

The input entries added during padding are artificial, and in practice are often just taken to be 0. In order for this not to alter the statistical properties of the data, this is often coupled with preprocessing the data so that the average pixel value is 0 [SZ14].

5.2.7. *Interpretability.* At least the first layer of a CNN tends to be "interpretable" in the sense of having visually apparent meaning. The filters can be visualized by viewing their values as grayscale intensities (after suitable scaling and translation), and then displaying them as an image. See Figure 5.2.4 for the result of doing this to the first convolutional layer of AlexNet [KSH12].

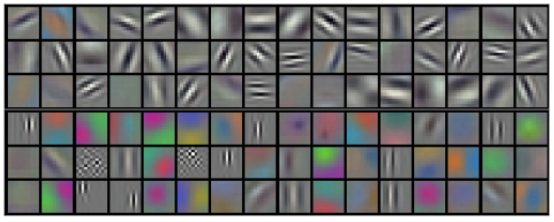


FIGURE 5.2.4. Filters learned from the first convolutional layer of AlexNet, from [KSH12, Figure 3].

5.3. **Case study: the ImageNet Challenge.** The *ImageNet project* was conceived and created by Fei-Fei Li [RDS+15]. It consists of 15 million images labeled into 22,000 categories. Each year from 2010 to 2017, an annual ImageNet Challenge showcased the top ML algorithms for image classification.

Each ImageNet Challenge was a miniaturized version of the ImageNet database, with 1.28 million training images, 50,000 validation images, and 100,000 test images into 1,000 categories. See Figure 5.3.1 for some examples.

There were two scoring metrics: whether the true classification was in the "top-1" and "top-5" classifications of the algorithm. The ImageNet Challenge spurred the development of CNNs, and we will give a selection of landmark breakthroughs.

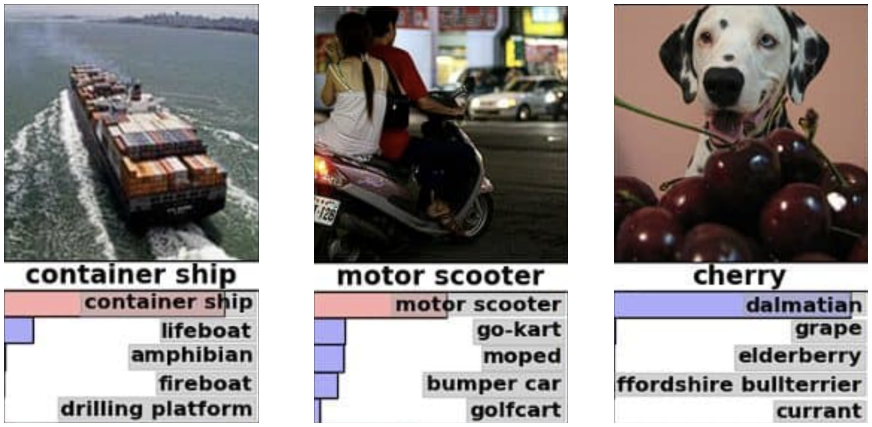


FIGURE 5.3.1. Sample of pictures from the ImageNet challenge, with ground-truth answer and AlexNet's top-5 classification [KSH12, Figure 4].

5.3.1. *AlexNet.* In 2012, the ImageNet Challenge was won by a CNN called AlexNet, developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton [KSH12]. It achieved a top-5 error rate of 15.3%, a huge improvement over the previous best 26.2%. The main factor behind this improvement probably came from the size of their model, which was enabled by a highly optimized implementation of convolution on 2 GPUs that trained for 5-6 days. (For comparison, at the time of this writing modern LLMs are trained on 10^5 GPUs for months.)

AlexNet had 5 convolutional layers and 3 fully connected layers, making it an especially deep CNN for that time. Each convolutional layer contained less than 1% of the learnable parameters, but its importance was affirmed by *ablation* (removing them and measuring the effect on performance).

In addition, AlexNet used a ReLU non-linearity instead of tanh or σ, whose non-saturating nature was important given the deep nature of the network. AlexNet also featured some other architectural innovations like "local response normalization" and "overlapping pooling", which however have not stood the test of time.

In terms of training, in addition to parallelizing on 2 GPUs, AlexNet popularized some important regularization tricks.

(1) They performed *data augmentation* by (1) translating and reflecting the image, and (2) perturbing RGB intensities. The latter was done by adding multiples of the PCA vectors to each pixel, the intuition being that this is reflects "realistic" correlations of RGB values occurring in natural images.
(2) They used dropout regularization.

5.3.2. *VGGNet.* In 2014, the VGGNet of Simonyan–Zisserman [SZ14] took runner-up in the ImageNet Challenge with a top-5 error rate of 7.32%.

VGGNet was influential in systematically demonstrating the value of increasing depth, using a simple uniform architecture.

- VGGNet had 16-19 weight layers, with very small convolution filters (3×3 with stride 1) and pooling filters (2×2 with stride 2). For comparison, AlexNet had 8 layers and used a mix of 11×11 (with stride 4), 5×5 and 3×3 filters.

- VGGNet had a simple design rule for pooling layers: use 2×2 pooling filters (with stride 2), and double the number of channels. Thus, the width and height of the feature map size are each halved, while the number of channels doubles.[18] See Figure 5.3.2 for an illustration of the VGGNet architecture.

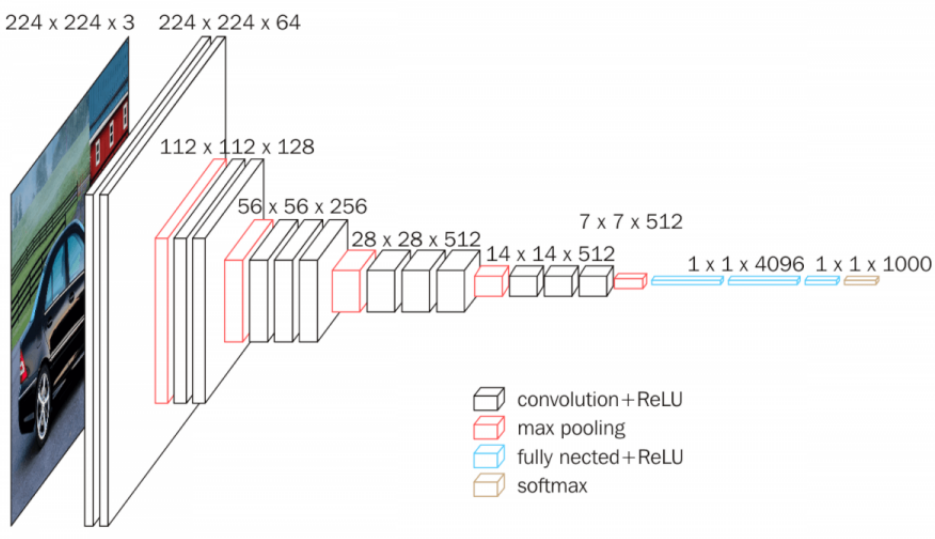


FIGURE 5.3.2. Depiction of the VGGNet architecture, from Varshney's Kaggle notebook. Note that when the size of the feature map is halved, the number of channels doubles.

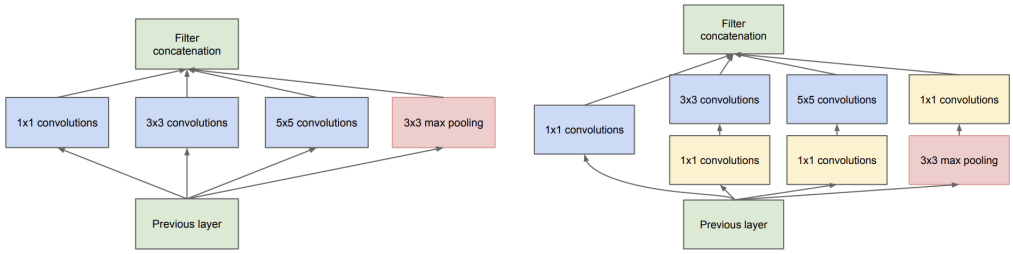
Example 5.3.1. As a toy example to contemplate the depth vs. filter size tradeoff, compare a 5×5 filter versus stacking two 3×3 filters. The latter creates a composite 5×5 receptive field, but has $18 < 25$ parameters and more non-linearities.

5.3.3. *Inception.* The actual winner of the 2014 ImageNet Challenge was a deep neural network developed by Szegedy et al at Google [SLJ+14], and codenamed "Inception" (as a reference to the meme, "We need to go deeper", which spawned from the movie Inception). The official name was GoogLeNet, which achieved a 6.67% top-5 error rate.

Despite achieving a better score, the stated goal of Inception was actually to improve efficiency. For example, they report a 12x reduction in parameters over AlexNet, while increasing depth to 22.

The idea was, interestingly, inspired by an academic paper [ABGM13], which advocates for dense networks clustering neurons with highly correlated outputs (reminiscent of the "Hebbian principle: neurons that fire together, wire together", from neuroscience). This leads Szegedy et al to do feature concatenation on filters of different sizes: $1 \times 1, 3 \times 3, 5 \times 5$, and 3×3 max-pooling. In order to keep the number of outputs from exploding, each of the larger filters is placed after a 1×1 downsampling convolutional layer. In total, this is the "Inception" module, depicted in Figure 5.3.3.

[18] Why this rule? In [HZRS15a], it is said to be "so as to preserve the time complexity per layer", where halving M and N is offset by doubling C_{IN} and C_{OUT}. Note that in the transition layers where C_{OUT} doubles but not C_{IN}, the complexity is approximately halved instead of preserved.



(a) Inception module, naïve version (b) Inception module with dimension reductions

FIGURE 5.3.3. Depiction of the Inception module, a stack of layers in GoogLeNet [SLJ+14, Figure 2].

5.3.4. *ResNet.* The 2015 ImageNet challenge was won by ResNet [HZRS15a], which introduced a fundamental new perspective on neural networks, achieving 3.57% error rate on the 2015 ImageNet challenge (surpassing human performance, which is cited to be around 5% [RDS+15]). The winning network was substantially deeper than previous submissions (152 layers).

The original issue with depth was the "vanishing gradients" problem, preventing convergence during training. This was addressed by gradient clipping and intermediate normalization layers.[19] The next issue was that, even when training of deep networks converged, at some point adding more layers led to a *degradation* in performance – see Figure 5.3.4. This is not believed to be a consequence of overfitting (as the performance degradation is witnessed in both training and test error).

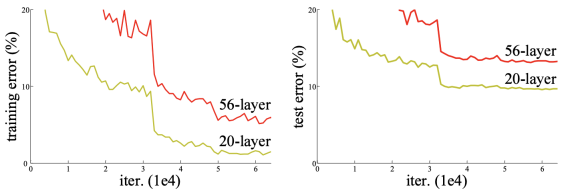


FIGURE 5.3.4. Naïvely training deeper networks leads to worse training and test performance [HZRS15a, Figure 1].

At first glance, this degradation is puzzling since deeper networks are clearly at least as expressive as shallow networks, for example by having later layers simply learn the identity function. This appears to have been a motivation for the key idea of [HZRS15a], which was to introduce "skip connections", whereby layer outputs are fed to later layers directly as in Figure 5.3.5. Note that adding skip connections (essentially) does not affect the parameter count or computational complexity.

The basic idea is that if the original layer was trying to learn a function $F(x)$, then adding the skip connection makes it learn the "residual" function $R(x) := F(x) - x$ instead. In particular, if $F(x)$ is the identity function then $R(x) = 0$, which is easy to learn.

[19] One checks that this works by measuring the gradient norms.

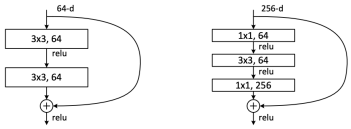


FIGURE 5.3.5. A "skip connection" in a residual neural network [HZRS15a, Figure 5].

If $F(x)$ and x do not have the same dimension, as is often the case, then one applies a linear function W to x which makes the dimensions agree. In a CNN, the dimensions increase through the network, and three possibilities are considered: (A) pad with 0, (B) identity matrix if dimensions do not increase, and learn a linear map $x \mapsto Wx$ if they do increase, (C) learn a linear map $x \mapsto Wx$ in all cases. There is a small improvement from (A) to (B), and a very small improvement from (B) to (C) which is negligible enough that the authors do not use (C).

Remark 5.3.2. The Taylor expansion of a function $F(X)$ decomposes it into its linear and higher order components,

$$F(x) = b + Wx + R(X),$$

where we can think of $R(X)$ as the "residual function" of the linear component. From this perspective, the idea of residual learning seems quite natural, but the fact that one can almost just as well take W to be the identity matrix in practice is surprising (at least to me).

Apart from the skip connections and the increased depth, the architecture of ResNet was very similar to VGGNet. In particular, it followed the same simple, uniform design principles for filter size (3×3 only) and dimension (double the channel dimension when the feature map dimensions halve). Skip connections are applied 2 or 3 layers; see Figure 5.3.6.

The training procedure of ResNet was also simple. Normalization layers were used but no dropout regularization.

Remark 5.3.3 (Do we still need to go deeper?)**.** [HZRS15a] also tried training a 1202-layer ResNet. Compared the 110-layer ResNet, this had comparable training error but substantially worse test error, indicating overfitting.

5.4. **Adversarial attacks.** Image recognition models are highly susceptible to *adversarial attacks*, meaning the construction of examples that the neural network will misclassify. Moreover, this can often be done by perturbing a correctly classified image in a way that is *imperceptible to humans*; see Figure 5.4.1.

A simple approach to constructing adversarial examples is to perform gradient *ascent* on the examples. That is, given the model parameters θ, we view a particular loss function $\mathcal{L}_x(\theta)$ as a function of the input x instead of the model parameter θ. Then we do gradient ascent to perturb some initial x_0 in a direction that *increases* the loss, $x_0' = x_0 + \mathrm{d}x$. In order to control the size of the modification, we constrain $\mathrm{d}x$ to be small.

In gradient descent, we effectively constrain $\mathrm{d}x$ using the L^2 norm. It turns out, however, that for the constraint of making the perturbation imperceptible, it is very effective to constrain $\mathrm{d}x$ in the L^∞ norm. The intuition here is that images are represented by discrete values, e.g., in steps of $1/255$ for RGB intensities, hence changes smaller than $1/255$ should be literally imperceptible. Thus, if we choose $\mathrm{d}x$ such that $||\mathrm{d}x||_\infty = \varepsilon$ for some small ε on

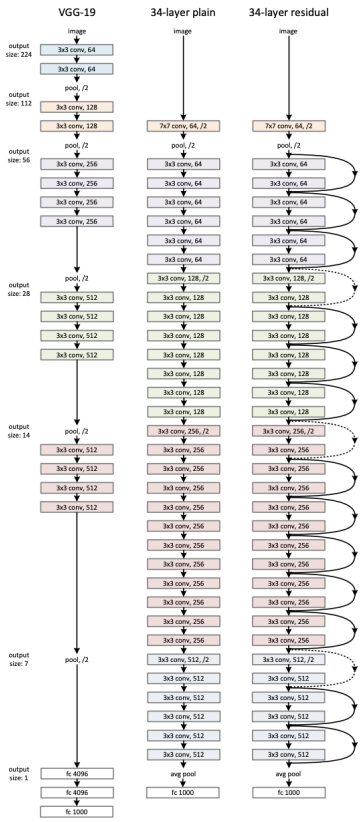


FIGURE 5.3.6. The architecture of the 34-layer ResNet [HZRS15a, Figure 3].

the order of $1/255$, then $x_0' = x_0 + \mathrm{d}x$ is almost indistinguishable from x_0'. On the other hand, the pre-activation value

$$x_0' W = x_0 W + (\mathrm{d}x)W.$$

changes by $(\mathrm{d}x)W$, which could have magnitude around $\varepsilon ||W||_1$. This can be astronomical if the dimension of W is large, as is often the case.

This is the basis of the *Fast Gradient Sign Method* (FGSM) [GSS14] of Goodfellow et al. The updates then have the shape

$$x' = x + \varepsilon \operatorname{sign}(\nabla_x \mathcal{L}_x(\theta))$$

where in [GSS14] the ε might be around $2/255 \approx 0.0078$.

To see why this is the correct answer, we can reformulate the problem as follows: for fixed w, find

$$\arg\max_{||v||_\infty \leq 1} vW.$$

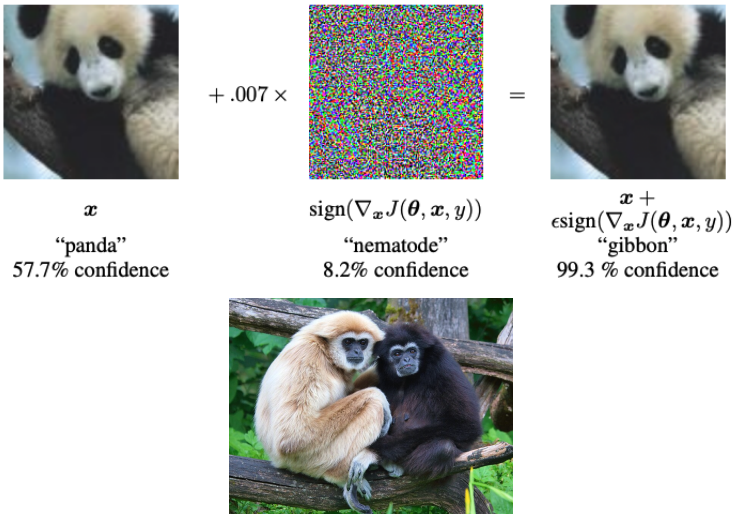
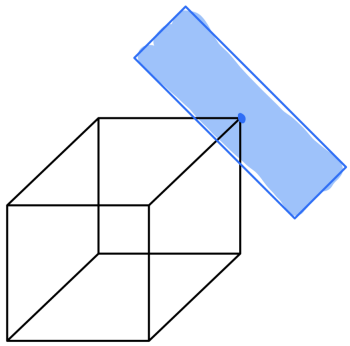


FIGURE 5.4.1. Top: an adversarial example generated by the FGSM method. Pixel intensity in $[0, 256]$ is renormalized to the interval $[-1, 1]$, the smallest change in pixels being about 0.007. Adding a small error vector causes GoogLeNet (winner of the 2014 ImageNet Challenge) to misclassify a panda as a gibbon [GSS14, Figure 1]. Bottom: a picture of a gibbon, from Wikipedia.

The answer is the coordinate-wise sign vector $\text{sign}(W)$. Indeed, since this is a linear program on a hypercube, it is maximized on one of the vertices, which all have coordinates $\{\pm 1\}^N$.



It is easy to see this more generally for bounded convex domains. If $v = \lambda v_0 + (1-\lambda)v_1$, then $\langle u, v\rangle = \lambda\langle u, v_0\rangle + (1-\lambda)\langle u, v_1\rangle$ is maximized by replacing v with whichever v_i maximizes $\langle u, v_i\rangle$. Thus, the function is maximized by "pushing" the input to the extreme points.

Remark 5.4.1. It is possible to embed this simple argument into a complicated but more general picture. There is a general theory of *norm duality*: a norm $||\bullet||$ on a vector space

V induces a *dual norm* $||\bullet||_*$ on V^*, by

$$||\xi||_* := \sup_{v \in V} \langle \xi, v \rangle.$$

Consider $V = \mathbf{R}^N$ (thus identifying $V^* \cong \mathbf{R}^N$) with the norm $||\bullet|| = L^p$ for $p \geq 1$. Then Hölder's inequality says that for q such that $\frac{1}{p} + \frac{1}{q} = 1$, we have

$$|\langle \xi, v \rangle| \leq ||\xi||_p ||v||_q$$

with equality if and only if $|\xi_i^p| \propto |v_i^q|$ as a vector in i. This shows that the dual norm to $L^q(\mathbf{R}^N)$ is $L^p(\mathbf{R}^N)$. Our example pertains to the special case $p = 1$ and $q = \infty$, where Hölder's inequality says that

$$\sup_{||v||_\infty = 1} \langle v, \xi \rangle$$

is maximized when $v_i = \operatorname{sgn}(\xi_i) \xi_i^{1/\infty} = \operatorname{sgn}(\xi_i)$. This is a special case of a general duality: constraining with one norm often leads to update steps related to the dual norm.

The FGSM method allows [GSS14] to investigate the landscape of adversarial examples. By varying ε and $\mathrm{d}x$, they conclude that adversarial examples occupy large, continuous regions. Also, adversarial examples tend to work across different models: FGSM examples constructed using a specific model tend to also be misclassified by other models. This means in particular that *ensembles* of models are not robust against adversarial attacks. The takeaway is that neural networks are quite fragile in practice.

6. Recurrent Neural Networks

In this lecture, we will begin to discuss architectures for natural language processing. This would include problems such as the following:

- *Classification*: e.g., sentiment analysis, spam filtering.
- *Machine translation* between different languages.
- *Generation*: e.g., text summaries, image captioning.

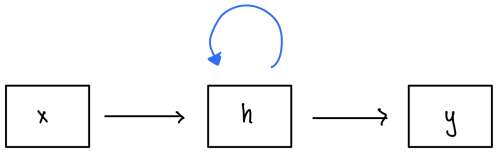
The major difference of the NLP setting is that the inputs are *variable length* sequences, and outputs could also be variable length sequences. Historically, *recurrent neural networks* (RNNs) were introduced to handle such tasks. They are based on the intuitively appealing idea of adding "memory" to a neural network, in the form of a hidden state that is modified as the sequence is processed.

More recently, RNNs have been superseded by Transformers, which we will discuss in the next section. Thus, from a modern perspective RNNs have become mostly obsolete. However, we want to take away some important ideas from the development of RNNs, such as the *encoder-decoder architecture* and the idea of *representation learning*.

6.1. **RNN architecture.**

6.1.1. *Hidden states.* The basic idea of a RNN is to add a *self-connection to each layer*, which records a "hidden state" h. In this way, when the network processes a sequence input,

each term of the sequence is able to "remember" information about previous terms.



Example 6.1.1. Let's spell out how this looks in a 1-layer RNN. The input is a sequence $x_1, x_2, \ldots, x_T$ with each $x_i \in \mathbf{R}^{d_x}$.

We have a hidden state $h \in \mathbf{R}^{d_h}$ (perhaps initialized to 0). The hidden layer has weight matrices $W_{hh} \in \mathbf{R}^{d_h \times d_h}$ and $W_{xh} \in \mathbf{R}^{d_x \times d_h}$. Each forward pass takes the form

$$h \leftarrow f(hW_{hh} + xW_{xh} + b). \tag{6.1.1}$$

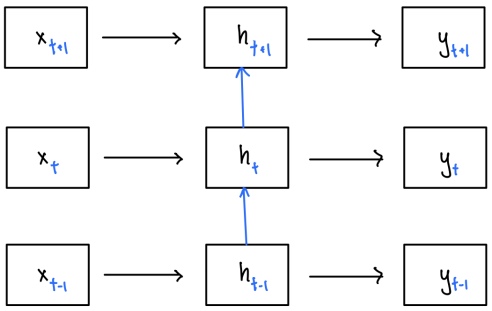
where f is the non-linearity. Typically RNNs used tanh instead of ReLU, in order to keep the entries bounded; otherwise compounding in the time direction could lead to exploding gradients.

Finally, the output y is computed from the hidden state h by a final fully connected layer, which could for example be of the form

$$y = \text{SoftMax}(hW_{hy}).$$

The weights of the $W_{??}$ matrices, as well as the bias b, are all learned in training.

6.1.2. *Unrolling.* Because of the recursive nature of the hidden state, an RNN can be viewed as a discrete-time dynamical system. To visualize how an RNN works, it is useful to "unroll" it, thinking of the sequence index as a "time" variable t.



(6.1.2)

Example 6.1.2. The unrolled form of (6.1.1) is a recurrence relation

$$h_t = f(h_{t-1}W_{hh} + x_tW_{xh})$$

The hidden state h_t at time t contains aggregated information about $x_t, x_{t-1}, \ldots, x_1$. Finally, the output y_t at time t is computed from h_t by another NN layer, say of the form

$$y_t = \text{SoftMax}(h_tW_{hy}).$$

Thus we see that when an RNN operates on input x_t, it effectively has the topology of a neural network of depth t. Let's call these "time layers" to distinguish them from "space layers", of which there is only one depicted in (6.1.2). In particular, the gradient descent and backpropagation updates are computed just as in a depth-t neural network. However, note that the unrolled network (6.1.2) shares weights across layers, since the neurons are

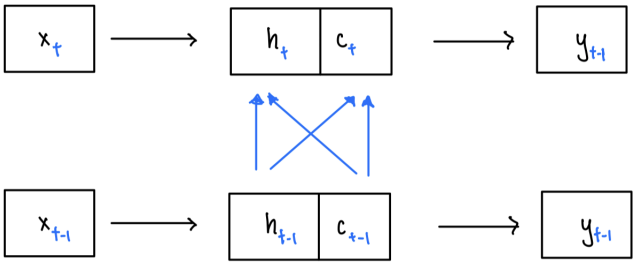
the same (at different snapshots in time). Hence if $\mathcal{L}_t(W)$ is the loss on (x_t, y_t), then we have

$$\frac{\partial \mathcal{L}_t}{\partial W} = \frac{\partial \mathcal{L}_t}{\partial h_t} \sum_{s=1}^{t} \frac{\partial h_t}{\partial h_s} \frac{\partial h_s}{\partial W}.$$

Note that the derivative $\frac{\partial h_t}{\partial h_s}$ involves $t-s$ multiplications by the matrix W_{hh}. On a training batch $x_1, \dots, x_T$, we would actually update with respect to the gradient of the aggregated loss $\sum_{t=1}^{T} \mathcal{L}_t$.

6.1.3. *Adding long-term memory.* We have advocated thinking of recurrent neural networks as being like deep neural networks, with variable depth. This comes with all the benefits and pitfalls of deep neural networks, e.g., the vanishing gradients problem. For CNNs, we saw that skip connections can be used to solve this problem, but it does not make sense to add skip connection forward in time, so we cannot implement residual learning in the "time" direction. This means that RNNs, as described so far, will degrade rapidly with sequence length.

A metaphor is that the hidden state is "short-term memory", which becomes attenuated through time. Hochreiter and Schmidhuber [HS97] suggested to address this with *Long Short-Term Memory* (LSTM). Their idea was to add an additional recurrent connection, effectively tracking a "long-term memory cell" c_t. Mathematically, this means that that there are *two* hidden states, one of which is designated as a long-term memory cell, as reflected by the nature of its dynamics.



Namely, the dynamics of c_t are governed by a recurrence of the form

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t \tag{6.1.3}$$

whose form reflects an inductive bias about the way that long-term memory should work.

- The function f_t (for "forget") has the form

  $$f_t = \sigma(x_t W_f + h_{t-1} U_f + b_f)$$

  where σ is the *elementwise* sigmoid. In particular, $f_t \odot c_{t-1}$ is multiplying each entry of c_{t-1} by a number in $(0, 1)$; intuitively, this term selects what to forget from the previous long-term memory.
- The function i_t (for "input") has the form

  $$i_t = \sigma(x_t W_i + h_{t-1} U_i + b_i),$$

  and $\widetilde{c}_t$ ("potential memory to add") has the form

  $$\widetilde{c}_t = \tanh(x_t W_c + h_{t-1} U_c + b_c).$$

Intuitively, the term $i_t \odot \widetilde{c}_t$ is trying to identify what memory to add from the new input at time t.

The key point which makes c_t "long-term" is that (6.1.3) is roughly *additive* in c, rather than multiplicative. This means that $\frac{\partial c_t}{\partial c_s} \approx f_t \odot f_{t-1} \odot \ldots \odot f_{s+1}$ with each f_i having entries in $(0, 1)$, leading to a more controlled gradient flow than $\frac{\partial h_t}{\partial h_s}$, which involves many multiplications of a large matrix W_{hh}.

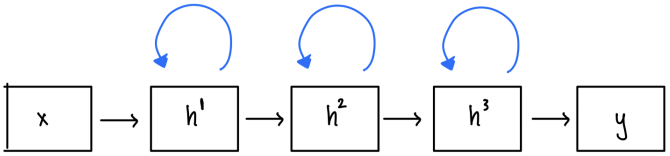
Finally, the hidden state is updated as

$$h_t = o_t \odot \tanh(c_t)$$

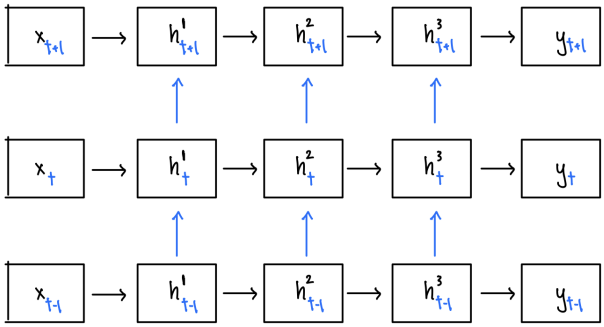
where $o_t = \sigma(x_t W_o + h_{t-1} U_o + b_o)$.

All the weights and biases (the W, U, and b terms) are learnable parameters.

6.1.4. *Deep RNNs.* So far we have only discussed RNNs with a single layer, even though we saw that they are effectively deep in the "time" direction. However, we can also stack RNN layers to gain deep in the "space" direction as well.



To think about the dynamics, it is again useful to "unroll" the RNN as a network with both a "space" depth and a dynamic "time" depth.



6.2. **Interpretability.** Andrej Karpathy's blog post on RNNs contains several instructive examples. We will look at some of his examples on *interpreting* the role of selected neurons.

In Figures 6.2.1 and 6.2.2, the first line displays the input sequence and the five rows below indicate the model's 5 most probable guesses for the next character, with the intensity of red proportional to the modeled probability (dark red meaning likely, white meaning not likely). In the first line, the pre-activation value in $[-1, 1]$ of a cherrypicked neuron is indicated by the color scale, with blue being close to -1 (not excited) and green being close to 1 (excited).

We see that the neuron in Figure 6.2.1 seems to be a detector for website URLs, while the neuron in Figure 6.2.2 seems to be a detector for being inside the `[[...]]` environment.

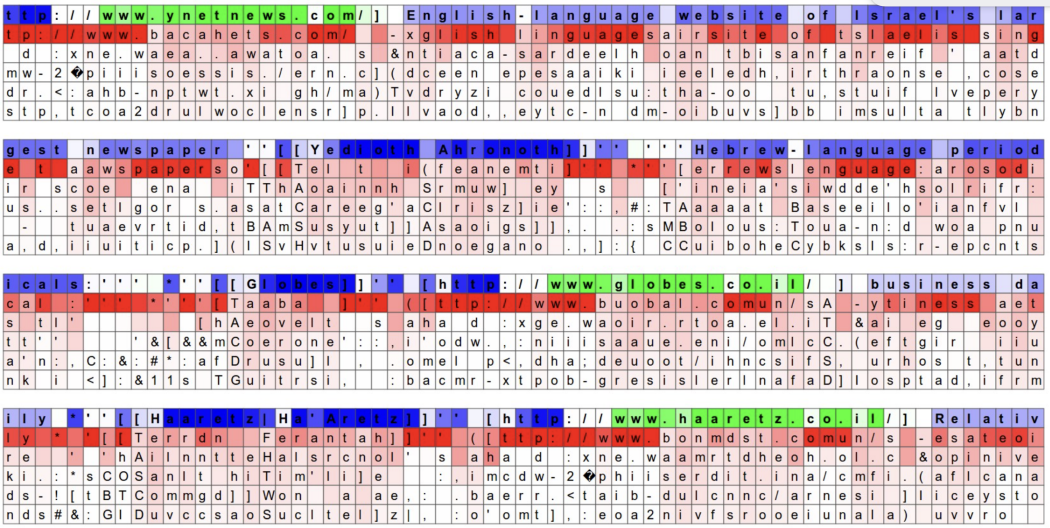


FIGURE 6.2.1. From the section *Visualizing the predictions and the "neuron" firings in the RNN* in `https://karpathy.github.io/2015/05/21/rnn-effectiveness`.

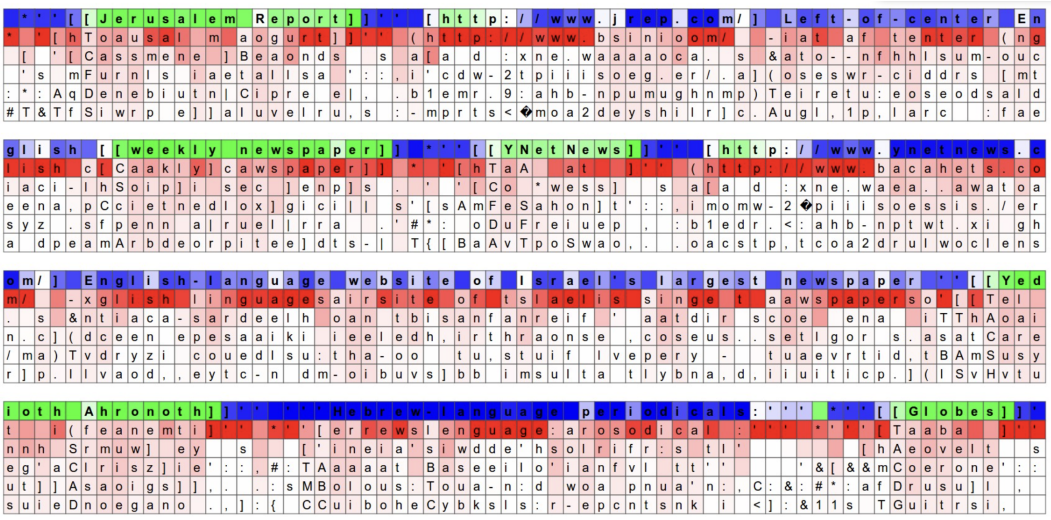


FIGURE 6.2.2. From the section *Visualizing the predictions and the "neuron" firings in the RNN* in `https://karpathy.github.io/2015/05/21/rnn-effectiveness`.

6.3. **Variable length output.** So far we have discussed using a hidden state in order to handle variable length inputs, but our illustrations made it appear as if the output length were a function of the input length. Going back to the example problems that we stated at the beginning of the lecture, we see that this is not realistic. It can be remedied by having

the model start output generation only with a special start token (such as <SOS>) and stop generation only with a special stop token (such as <EOS>). But a specific design, called the *encoder-decoder architecture*, gained particular popularity for handling situations where the output sequence length could differ substantially; we will discuss this next.

6.4. **Encoder-decoder architecture.** The encoder-decoder architecture seems to have been introduced almost simultaneously by two independent groups: Cho et al [CvMG+14] in June 2014, and then Sutskever et al [SVL14] in September 2014. The idea is to build a neural network with two halves: an "encoder" RNN, which turns a variable-length sequence into a fixed-length vector (its *representation*), and then a "decoder" RNN with the inverted structure, that turns a fixed-length representation back into a variable-length sequence. The template for the architecture is depicted in Figure 6.4.1.

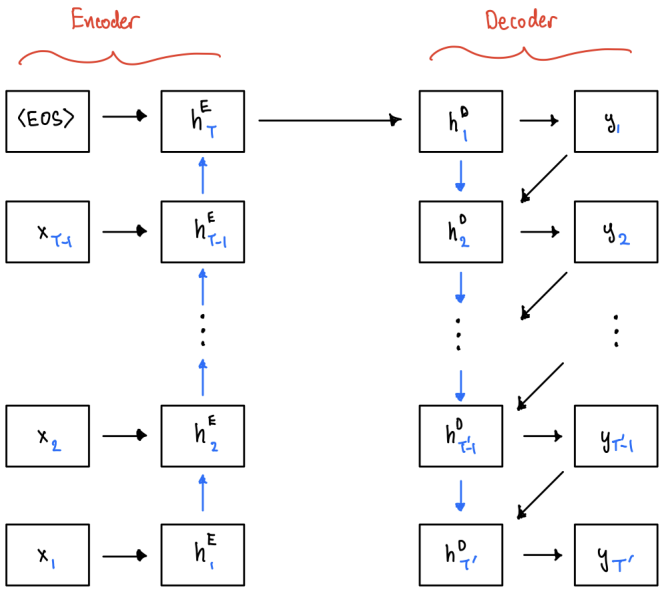


FIGURE 6.4.1. Unrolled illustration of the encoder-decoder RNN architecture. The left half constitutes the encoder, which processes a variable-length input sequence into a fixed-length representation. The right half constitutes the decoder, which processes the fixed-length representation into a variable-length output sequence.

This architecture is especially popular when:

(1) The output length T' might be significantly different from the input length T, or
(2) The input and output are of different natures, e.g., when translating one language to another, or captioning an image.

Mathematically, the RNN is predicting a conditional probability distribution

$$p(y_{1:T'}|x_{1:T}).$$

The encoder that reads in the input sequence $x_{1:T}$, and passes its hidden state h^E_T to the decoder. This hidden state h^E_T is a fixed-length "representation" (aka "embedding") of $x_{1:T}$,

with dynamics governed by an equation of the form

$$h_t^E = f(h_{t-1}^E, x_t).$$

The decoder RNN generates y_t, and has hidden state of the form

$$h_t^D = g(h_{t-1}^D, y_{t-1})$$

with initialization $h_0^D = h_T^E$. Note that the decoder's output at time $t-1$ serves as its input at time t.

These two halves are jointly trained to minimize the conditional log-likelihood of an output sequence $\underline{y}$ given an input sequence $\underline{x}$,

$$\min_\theta \left(-\log p_\theta(\underline{y}|\underline{x})\right). \tag{6.4.1}$$

Actually, in practice one would take the average of such loss over a batch, but we will ignore this for notational convenience. To express (6.4.1) in terms of the neural network, write $\underline{y} := y_{1:T'}$ and $\underline{x} := x_{1:T}$. The RNN itself calculates a prediction of the form

$$p_\theta(y_t|x_{1:T}, y_1, \dots, y_{t-1}) = p_\theta(y_t|h_T^E, y_1, \dots, y_{t-1})$$

where $h_T^E = E(x_{1:T})$ is the embedding of $x_{1:T}$. This then gives an expression for the conditional probability $p_\theta(\underline{y}|\underline{x})$ as

$$p_\theta(\underline{y}|\underline{x}) = p_\theta(y_{1:T'}|h_T^E) = \prod_{t=1}^{T'} p_\theta(y_t|h_T^E, y_1, \dots, y_{t-1})$$

so that (6.4.1) can be rewritten as

$$-\log p_\theta(\underline{y}|\underline{x}) = -\sum_{t=1}^{T'} \log p_\theta(y_t|h_T^E, y_1, \dots, y_{t-1}). \tag{6.4.2}$$

Note that each summand $-\log p_\theta(y_t|h_T^E, y_1, \dots, y_{t-1})$ can be interpreted as the cross-entropy loss on token y_t.

6.5. **Representation learning.** As the encoder-decoder network trains, the encoder learns a mapping E from variable-length sequences $\underline{x}$ to fixed-length vectors h_T^E. This is an example of "representation learning" (in the sense of Terminology 1.4.6): learning a hidden *latent representation* of a input data. Here the adjective "latent" connotes features which are not explicitly observed in the data.

Latent space is also often called "embedding space", and the latent representation is often also called an "embedding".

The latent representation is abstract, and could be hard to interpret concretely. One can look at examples to get an intuitive feel for latent space. For example, one would expect that sequence vectors which are literally quite different, but similar in semantic meaning, like "`Sup dog?`" and "`How are you?`", would end up close to each other in latent space. Another intuition is that vector arithmetic is meaningful in latent space, as seen in the famous example

$$E(\texttt{king}) - E(\texttt{man}) + E(\texttt{woman}) \approx E(\texttt{queen}).$$

A 2D projection of the learned representation in [CvMG+14] is displayed in Figure 6.5.1; more such figures can be found there.

The standard intuition about real-world data is that it is messy and unstructured. By contrast, a good latent representation compresses into "nice" coordinates. As a mathematical metaphor, one can imagine the "meaningful" datapoints (e.g., natural language sentences

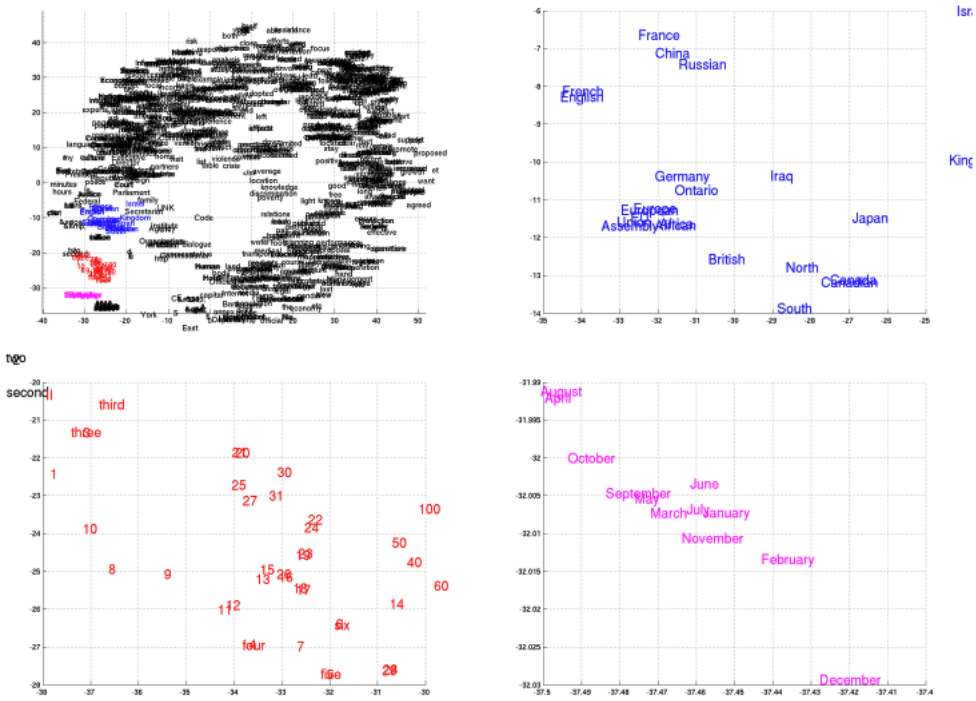
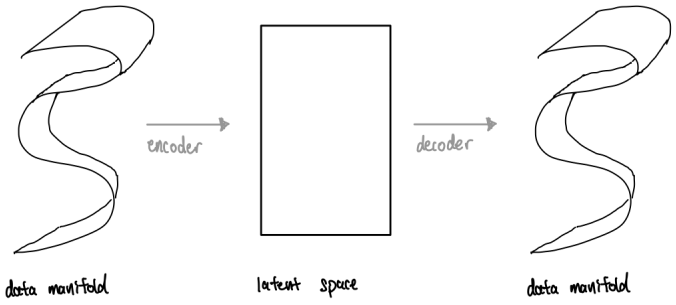


FIGURE 6.5.1. Top left: 2D projection of the representation learned by the encoder in [CvMG+14]. The other panels are zoomed-in views of colored regions, displaying a semantic pattern to that region of latent space. From [CvMG+14, Figure 6].

among all possible combinations of English words) as forming a very thin manifold in an extremely high-dimensional space of possibilities. The encoder aims to learn a mapping from this manifold to a Euclidean space, which admits an inverse, to be learned by the decoder. This is analogous to learning local coordinate charts for a low-dimensional manifold embedded in high-dimensional space. The cartoon below depicts a visualization of this framework.



In these terms, the data is viewed as an *implicit* presentation of a manifold, as a subset of some very high-dimensional space, and the encoder-decoder network is trying to present it

in terms of a Euclidean atlas. The point of embedding in latent space is that this is a better setting to do calculations: a principle familiar from analysis on manifolds.

The "Manifold Hypothesis" refers to the principle that real-world data is typically a rather low-dimensional manifold embedded in high-dimensional space, and thus the dimension of latent space can be taken to be quite small. For example, we saw in the discussion of §5.3 on the ImageNet challenge that a standard 224x224 image has dimension $224 \times 224 \times 3 \approx 150,000$, and gets downsampled to an embedding of dimension 512 by a 34-layer ResNet.

6.6. **Sampling.** When sampling from the RNN at inference time, one ideally wants to generate the output sequence $\underline{y}$ maximizing the conditional probability $p(\underline{y}|\underline{x})$ given an input sequence $\underline{x}$. However, the number of possible sequences blows up exponentially with length, so it is not feasible to consider them all. Note that *greedy selection*, where the single most probable token is selected at each step, does not necessarily lead to the most probably *sequence*.

6.6.1. *Temperature sampling.* One can consider instead just sampling from the distribution computed by the neural network. However, this tends to give poor results. The reason is that, as discussed around Slogan 3.3.8, cross-entropy penalizes the model for being confident but wrong, so the model is incentivized to allocate a tiny possibility to all possible tokens. Thus, there tends to be a long tail of many improbable tokens each of which carries very little weight, but which aggregate to a non-trivial proportion of the total probability.

Temperature sampling is a variant of this which introduces a "temperature" parameter τ that allows one to suppress the long tail. The parameter τ enters in renormalizing the softmax function:

$$\mathrm{SoftMax}_\tau(a_i) = \frac{\exp(a_i/\tau)}{\sum_j \exp(a_j/\tau)}.$$

The limit $\tau = 0$ concentrates on the single most probable state, i.e., greedy selection, while $\tau = 1$ is the standard softmax. Thus a temperature in $(0, 1)$ strikes an intermediate between the two. LLMs use temperature sampling, and a typical temperature value might be $\tau \in [0.5, 0.7]$.

We will also discuss some other sampling methods that come up.

6.6.2. *Beam search.* The idea of Beam search is to maintain a set of k running "hypotheses" $y^{(1)}_{1:t}, y^{(2)}_{1:t}, \ldots, y^{(k)}_{1:t}$ at each time step, and select the best one at the end.

To generate token $t+1$, we use the neural network to calculate the top-k next tokens for each of the k hypotheses $y^{(1)}_{1:t}, y^{(2)}_{1:t}, \ldots, y^{(k)}_{1:t}$, and then prune this down to the next list of top-k hypotheses,$y^{(1)}_{1:t+1}, y^{(2)}_{1:t+1}, \ldots, y^{(k)}_{1:t+1}$.

Actually, there is a difference between the calculation of log probability at inference time versus during training. Notice that (6.4.2) tends to penalize longer sequences, since each token contributes a negative log likelihood. Therefore, at inference time one should normalize the log likelihood by the length of the sequence, i.e., consider the *average* log likelihood per token. But during the actual beam search, when processing a token one does not know how long the final sequence will be. Hence in order to implement this with beam search, a separate list of "finished" sequences need to be maintained, with the beam search only running over unfinished sequences. Once sequences are finished, their log likelihood can be normalized and compared.

6.6.3. *Nucleus sampling. Nucleus sampling* strikes an intermediate between beam search and sampling directly from the modeled distribution. The idea is to truncate the long tail by considering only the most probable tokens up to a certain proportion (a fixed hyperparameter, e.g., 0.9) of the total mass, and then sampling from them according to the model's conditional probabilities (renormalized).

6.7. **Case studies.** We will discuss some early examples of encoder-decoder RNNs, which were trained for machine translation. For this, a common evaluation metric is "BLEU score", which compares the machine translation to a reference translation, giving points for having n-grams in common. BLEU score is technically valued in $[0, 1]$, but typically reported as a percentage in $[0, 100]$. A higher score is better, with < 10 considered very poor, 20-30 considered understandable but not fluent, 40-50 considered good, and 60+ considered close to human performance.

6.7.1. *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation.* The paper [CvMG$^+$14] by Cho et al introduced the encoder-decoder architecture and analyzed its performance on English-to-French translation of phrases (typically 5-7 words instead of a full sentence). They did not use the RNN as a standalone translation system, but rather as a "helper" to rescore the output of a traditional Statistical Machine Translation system. In this way, they were able to achieve a BLEU score of 34.64 on the WMT'14 dataset, which consists of 12M English/French sentence pairs, with 304M English words and 348M French words.

Parameters. Cho et al used a single-layer RNN with "1000 hidden units", meaning the hidden state has dimension 1000. To decrease the dimension of the matrix W_{xh}, they used a low-rank approximation by forcing $W_{xh} = W_{xz}W_{zh}$ where z has dimension 100. A back-of-the-envelope calculation thus indicates that the model had $\approx 10^8 - 10^9$ learnable parameters.

The model was trained on a table of phrase pairs drawn from the WMT'14 dataset. Training was done with batch size of 64.

Innovations. Compared to our discussion of RNNs, Cho et al made a few modifications that were important for performance. Firstly, they used a different formula for the dynamics of their hidden states than the conventional LSTM formulas from §6.1.3, which they call a "Gated Recurrent Unit" (GRU). Secondly, they added a compartment to the hidden unit of the decoder which retains the representation learned by the encoder *without any modification.* In terms of the unrolled picture, this means that the encoder's representation is fed directly to the hidden unit of the decoder at each time step: see Figure 6.7.1, and compare to Figure 6.4.1. This prevents the decoder from "forgetting" the encoder's representation as the output sequence length grows.

6.7.2. *seq2seq.* The encoder-decoder architecture seems to have been independently discovered by Sutskever et al. in the paper [SVL14] appearing a few months after the work of Cho et al. On the same WTM'14 English to French translation dataset, their system, called "seq2seq" achieved a BLEU score of 36.5 when used in a similar way as Cho et al's (i.e., in combination with a phrase-based Statistical Machine Translation system). Notably, however, seq2seq also attained impressive performance translating full sentences end-to-end, achieving a BLEU score of 34.81 by itself. In particular, seq2seq exhibited decently robust generalization to longer sequences.

Innovations. Compared to [CvMG$^+$14], Sutskever et al opted for a deeper RNN, with 4 layers. They also discovered that the simple trick of *reversing* the input sequence order

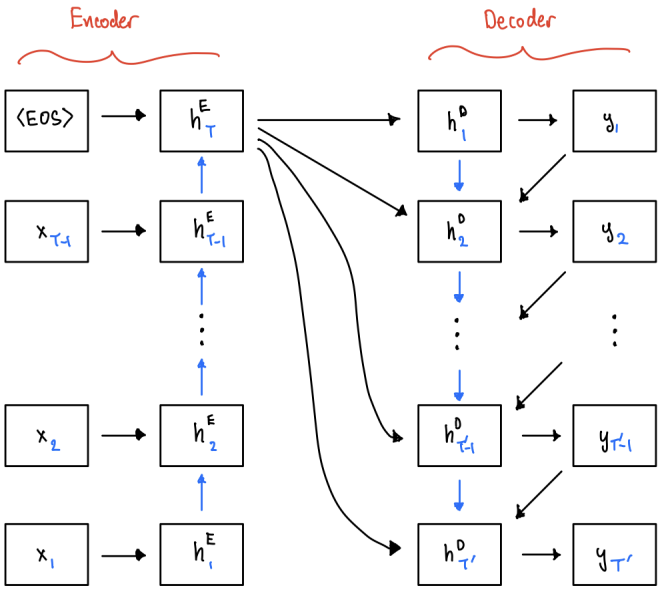


FIGURE 6.7.1. Unrolled illustration of the encoder-decoder RNN architecture from Cho et al [CvMG+14]. The encoder's representation is directly fed to the decoder at each time step.

helped a lot with performance: instead of translating $x_1x_2x_3 \mapsto y_1y_2y_3$, they translate $x_3x_2x_1 \mapsto y_1y_2y_3$. The intuition seems to be that in the reversed setup, x_1 is closer to y_1. We note however that Sutskever et al do not seem to use the trick of connecting the learned representation directly to the decoder at each time step (Figure 6.7.1), which also mitigates the problem of memory attenuation.

Parameters. The model of Sutskever et al had 4 hidden layers in each LSTM (i.e., encoder and decoder). Word embedding and hidden state dimension were both 1000; the total learnable parameter count was 348M.

Training ran for 7.5 epochs, with batch size 128. The initial learning rate was 0.7, and was halved every half-epoch starting with epoch 5. The training was parallelized on 8 GPUs, and took about 10 days.

The BLEU score of 34.81 was achieved using beam search with size $k = 12$, although $k = 2$ already achieved 34.50 and even $k = 1$ achieved a respectable 33.00.

7. TRANSFORMERS

This lecture covers *Transformers*, the architecture underlying modern generative models such as LLMs. The key aspect of Transformers is the Attention mechanism, a different way of handling variable length inputs. Attention was originally introduced by Bahdanau et al in 2014 [BCB14] as an augmentation to RNNs. Transformers were invented by Vaswani et al in 2017 [VSP+17]; as their title *Attention is All You Need* suggests, the key insight was to jettison RNNs entirely and make Attention the focal point.

7.1. Attention.

The basic intuition of Attention is that when processing a sentence, the words pick up meaning depending on their relation to other words in the sentence. For

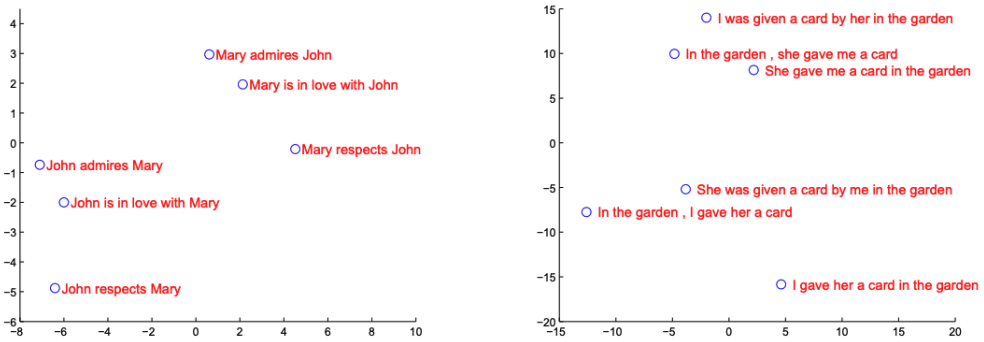


FIGURE 6.7.2. Examples of phrases in two regions of latent space from the representation learned in [SVL14]. From [SVL14, Figure 2].

example, consider the following two sentences:

```
I crossed the river to get to the bank.
```
(7.1.1)

```
I crossed the street to get to the bank.
```
(7.1.2)

Although the sentences are almost identical, the word "bank" has two different very meanings. In the first case, it is referring to a *river* bank, as is hinted by the presence of the word "river" earlier in the sentence. In the second case, it is referring to a *financial* bank, as is hinted by its location across the "street".

A RNN would handle this by maintaining "hidden states" and "memory cells" which record some understanding of the context. By contrast, Attention will *transform the inputs themselves*, so that each word becomes a superposition of itself along with other words that give contextual guidance.

7.1.1. *Toy model.* Here is a toy mathematical model for what Attention is trying to accomplish. We are given an input sequence (x_i), we want to turn it into a new sequence (y_i) where each y_i is a "superposition" of the x_i, in the sense that

$$y_i = \sum a_{ij} x_j \tag{7.1.3}$$

where $a_{ij} \in [0,1]$ and $\sum_j a_{ij} = 1$. In our examples above, one might imagine that in (7.1.1) the word $x_i =$ `'bank'` might become $y_i = 0.9 \cdot$ `'bank'` $+ 0.1 \cdot$ `'river'`, while in (7.1.2) the word $x_i =$ `'bank'` might become $y_i =$ (say) $0.9 \cdot$ `'bank'` $+ 0.1 \cdot$ `'street'`.

Remark 7.1.1. Depending on the NLP task, it might be natural to demand some "upper-triangularity" to the matrix (a_{ij}), corresponding to the fact that at the time of seeing x_i, only the previous x_j with $j < i$ have been seen.

7.1.2. *Metaphor to information retrieval.* In the toy model, how might we come up with the coefficients a_{ij}? Basically we are asking: given a word x_i, which of the other words x_j is relevant for x_i? This is similar the problem of *information retrieval*; an example use case is building a database where the user can enter a search query and get a list of relevant results. A basic idea for doing this (to first approximation) is to attach a *Key* vector K to each item in the database, and also a *Query* vector Q to each query. The Key and Query vectors should be of the same dimension, so we can take their dot product to obtain a measure of similarity.

Example 7.1.2. For a concrete example, suppose our database consists of movies. For each movie, the Key vector K might encode what we think are important attributes: genre, director, starring actors, reviews, etc. The Query vector Q would attempt to extract these attributes from the user's search query, and the inner product $\langle K, Q\rangle$ would reflect the strength of the match.

Finally, in information retrieval we will return a *Value* vector V which might differ substantially from the key K. In our movie database example, the Value might for example just be the title of the movie. In NLP tasks, it will itself be a function of inputs which will be learned.

7.1.3. *Attention.* Now imagine an NLP setup where our input is a sequence of vectors $x_1, \ldots, x_N$. Write

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

where each x_i has dimension d.

Our plan is to compute similarity coefficients

$$\text{Similarity}(K(x_j), Q(x_i)), \quad \text{for each pair } i, j.$$

Here, the Key and Query vectors $K(x_i)$ and $Q(x_i)$ have dimension $d_K = d_Q$. To first approximation, we can take the dot product

$$\langle K(x_j), Q(x_i)\rangle = K(x_j)Q(x_i)^\top$$

to be a measure of similarity. To turn this into a "superposition", satisfying $\sum_j a_{ij} = 1$, we would like to apply the softmax function. However, we should do some renormalization first, to maintain the scale of the features. To find the normalization constant, we approximate the individual entries of $K(x_j)$ and $Q(x_i)$ as i.i.d. random variables with a fixed variance, say $\sigma^2 = 1$. Then the variance of the dot product is d_K, so scaling by $1/\sqrt{d_K}$ recovers a variance of 1.

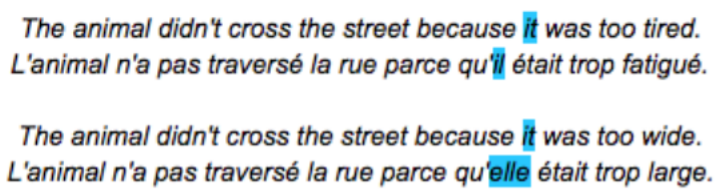
To summarize, we will have

$$a_{ij} = \text{SoftMax}\left(\frac{1}{\sqrt{d_K}} K(x_j)Q(x_i)^\top\right)_j.$$

Finally, after incorporating the additional flexibility of a Value encoding, the output of the Attention layer on x_i becomes

$$y_i = \sum_j \text{SoftMax}\left(\frac{1}{\sqrt{d_K}} K(x_j)Q(x_i)^\top\right)_j V(x_j). \tag{7.1.4}$$

Example 7.1.3. This example is taken from [Goo17]. Consider the English-to-French translations



Although the sentences have essentially identical structure, in the first sentence the word "it" refers to the animal, while in the second sentence "it" refers to street. Figure 7.1.1 displays the Attention scores in one of the later layers of [VSP+17], which clearly reflect this difference.

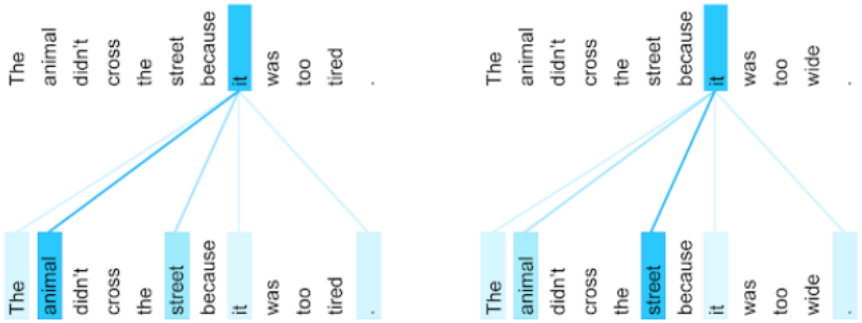


FIGURE 7.1.1. Visualization of attention coefficients (as intensity of the blue color) for the word "it" in two different English sentences. The attention scores indicate that "it" refers to different subjects in the sentences. From [Goo17].

7.1.4. *What are the Query, Key, and Value vectors?* We have not yet explained how to calculate $K(x_i), Q(x_i)$, and $V(x_i)$. These will be given by linear functions of the input, of the form

$$\begin{aligned} Q(x_i) &= x_i W_Q \\ K(x_i) &= x_i W_K \\ V(x_i) &= x_i W_V \end{aligned}$$

The weights of the matrices W_Q, W_K and W_V will be learned parameters. The dimension of $W_?$ will be $d \times d_?$ where d is the dimension of x_i (the "model dimension"). We need $d_K = d_Q$, and we typically take $d_K = d_V$ as well, although this is not necessary.

Remark 7.1.4. We never use W_Q and W_K separately; they always appear together as $W_K W_Q^\top$. In principle, we could just learn a $d \times d$ matrix W_{KQ} instead. We are effectively taking W_{KQ} to be of the form $W_K W_Q^\top$, which can be seen as a way of forcing W_{KQ} to have low rank (at most $d_K = d_Q$, which is typically much smaller than d – see Example 7.1.6).

7.1.5. *Multihead attention.* In practice, it will be useful to keep track of independent "attention heads", which attend to different aspects of the input. This is analogous to the different channels in a CNN. To do this, one stacks W_K^i, W_Q^i, W_V^i for $i = 1, \dots, h$. Typically, the dimensions are arranged so that $hd_V = d$, so that the total dimension of the inputs and outputs are the same after accounting for the number of attention heads.

Remark 7.1.5. Instead of tracking multiple attention heads, why not simply take use longer $K/Q/V$ vectors, of dimension d? The hope is that different heads will "specialize" to track different information. If we put them all under the same softmax function, then some would be swamped by others.

Example 7.1.6. For the original transformer, $d = 1024$, $h = 16$, $d_K = d_V = 64$. For the largest GPT-3 model (175B parameters), $d = 12288$, $h = 96$, and $d_K = d_V = 128$.

7.2. **Positional embedding.** As described so far, Attention depends only on the multiset of input $x_1, \dots, x_T$ and not to their order. However, we know that order matters in NLP. The idea to address this is to also incorporate a "positional encoding" into the sequence, which tracks the position of the input. The most naive way to do this would be to append an index, e.g., $\widetilde{x}_n = (x_n, n)$. However, it is not really the numerical value of the index that matters, but rather its position relative to the other inputs, so this imposes an unwanted artificial inductive bias.

One might instead try to *learn* an embedding of indices, $n \mapsto r(n)$. This is a valid approach, but can struggle to generalize to indices not seen in training.

Perhaps surprisingly, the method of [VSP+17] is to take $\widetilde{x}_n = x_n + r(n)$ where $r(n)$ is a handcrafted "sinusoidal" vector. The intuition behind taking a vector sum instead of putting the position in a separate coordinate seems to be that two "random" vectors in high-dimensional space are likely almost orthogonal, so it should be possible to essentially separate out the position vectors.

As for the specific form of $r(n)$, the basic intuition seems to be to record the angle $(\cos n, \sin n)$ instead of n itself. However, if n gets large then different positions will map to almost the same angle, so [VSP+17] instead takes $r(n)$ to be a vector with coordinates $(\cos \frac{n}{\omega}, \sin \frac{n}{\omega})$ for increasing "frequencies" ω; the later coordinates decreases the repetition horizon. More precisely, their position vector r_n has the form

$$(r(n))_{2i} = \sin\left(\frac{n}{10000^{2i/d}}\right), \qquad (r(n))_{2i+1} = \cos\left(\frac{n}{10000^{2i/d}}\right).$$

7.3. **Transformers.** Attention was originally conceived by Bahdanau et al [BCB14] as an added component of Recurrent Neural Networks. However, the paper [VSP+17] proposed the *Transformer* architecture, in which Attention completely replaces the recurrent layers. We will describe the architecture of *loc. cit.* (displayed in Figure 7.3.1) as a "vanilla" model for transformers; in §8.4 we will meet some subsequent modifications of this architecture.

7.3.1. *Encoder-decoder structure.* The Transformer of [VSP+17] has two halves. As with an RNN, the encoder half embeds a variable-length input sequence as a latent representation, and the decoder half is an *autoregressive*[20] model that uses the encoder's representation to produce an output sequence. But unlike an RNN, the latent representation now itself has *variable length* (except that it is limited by some context window): it consists of the Attention-weighted Values (7.1.4). Following the discussion of §7.1, at least for the first layer, this representation can be imagined intuitively as a superposition of Values attached to the original input tokens.

The encoder is similar to an RNN's encoder, except with recurrent layers replaced by Attention layers. Thus, an encoder layer consists of an attention sublayer followed by a standard fully connected sublayer. A 1-layer Transformer is depicted below, although in practice there would be several layers (the paper [VSP+17] uses 6 layers in both encoder

[20]meaning that its own previous output is used as input

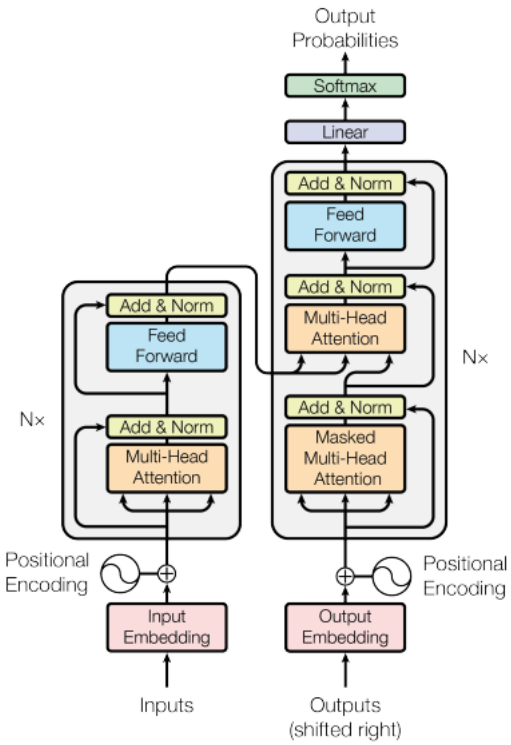
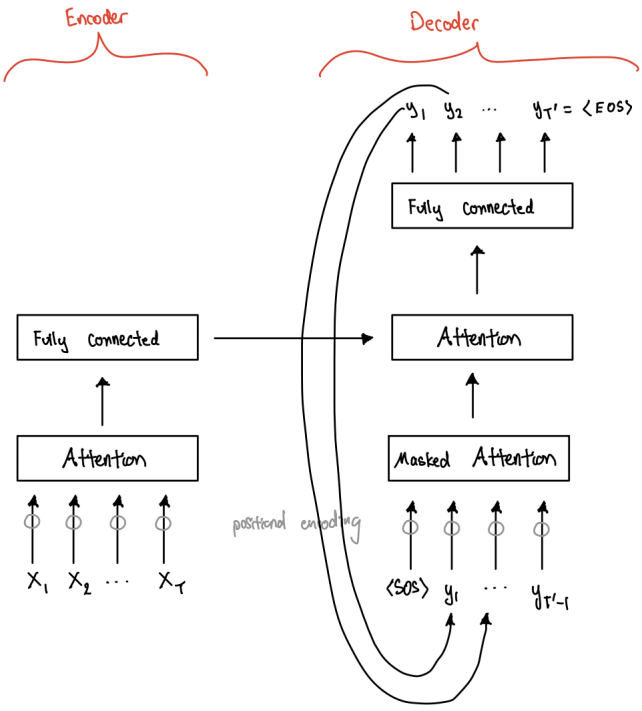


FIGURE 7.3.1. The original Transformer architecture from [VSP+17, Figure 1]

and decoder).



The decoder half is similar, except that (like in an RNN) it is *autogressive*: its own output tokens $y_1, \ldots, y_{t-1}$ are fed back into itself as input in order to predict token y_t. Consequently, the decoder contains an additional Attention layer that processes its own output. This layer is *masked* so that tokens cannot attend to "future" tokens; mathematically, this means that the Attention scores $K(x_j)Q(x_i)^\top$ for $j > i$ are set to $-\infty$ (so that they become 0 after applying softmax).

Positional embeddings are applied from the input to the first layer of the encoder, as well as from the output to the first layer of the decoder.

Finally, both encoder and decoder enjoy the bells and whistles of a deep neural network that we have already learned about: residual skip connections (around each of the sublayers), and layer normalization (since batch normalization is not possible for autoregressive models).

7.3.2. *Hyperparameters.* As mentioned above, in the original Transformer of Vaswani et al there were 6 layer stacks in both the encoder and decoder. For the largest model of [VSP+17], the embedding dimension of input tokens was $d = 1024$, while the dimension of the feedforward networks was 4096. There were $h = 16$ attention heads, and key and value dimensions were both $d_K = d_V = 64 = d/16$.

7.3.3. *Training.* The paper [VSP+17] was trained on 8 (NVIDIA P100) GPUs over ≈ 3.5 days; the estimated number of total FLOPs was 2.3×10^{19}. On the WMT'14 dataset, the largest Transformer model achieved a state-of-the-art BLEU score of 41.8.

7.4. **Comparison to RNNs.** Transformers are now the dominant architecture for NLP (and a host of other problems not necessarily of sequential nature, such as computer vision).

One major advantage of Attention is in facilitating the signal between inputs at large distances. In RNNs, the input x_i attenuates through n memory computations to influence x_{i+n}, while in Attention it is always a constant number of operations. The paper [KMH+20] suggests that LSTM RNNs degrade when the context exceeds 100 tokens.

Another important advantage of Transformers over RNNs is that Attention is highly *parallelizable*, which makes it amenable to scaling (e.g., by distributing computation over many GPUs). In particular, the inner products

$$K(x_j)Q(x_i)^\top = x_j W_K W_Q^\top x_i^\top$$

can all be computed in parallel, whereas an RNN must process inputs sequentially through hidden states. Because of this, Attention-based neural networks can be trained at much larger scales than RNNs.

In summary, the following table illustrates the comparison of *sequential operations* (largest number of computations that must be performed in sequence, rather than being parallelizable) and *path length* (largest number of computations intervening between two members of the sequence) for Attention versus Recurrent layers on a sequence of length n and representation dimension d.

Layer	Computational complexity	Sequential operations	Path length
Attention	$O(n^2 d)$	$O(1)$	$O(1)$
Recurrent	$O(nd^2)$	$O(n)$	$O(n)$

Left for presentation instead of lecture

7.5. **Case study: Vision Transformers.** Although transformers were originally introduced for NLP, they have proven to be flexible enough for multimodal capabilities. In the domain of Image Classification, the Vision Transform (ViT) of Dosovitskiy et al [DBK+20]

was notable for achieving state of the art performance using Transformers instead of, or in addition to, CNNs.

7.5.1. *Tokenization.* The main adaptation required to use Transformers for computer vision is in tokenization. The idea of [DBK+20] is to cut the initial image up into smaller square patches, and effectively treat each patch as a token. Learned embeddings are applied to these patches before the attention layers.

Example 7.5.1. Suppose the original image has dimension $H \times W \times C$, e.g. $H = W = 224$ and $C = 3$ for ImageNet. If the patch width is P, then the number of patches is $N = HW/P^2$, and the dimension of each patch is P^2C. Dosovitskiy et al chose patch dimensions such as $P = 16$, which with $C = 3$ leads to a model dimension of $P^2C = 768$ – the same as for the base model GPT-1 and BERT.

Note that unlike in NLP, there is no natural sequence order of the tokens produced from an image. Since Attention is not naturally sequential anyway, this is no problem, but it does mean that a different positional embedding should be applied. Several schemes for learning position embeddings were considered, but the simplest scheme of choosing a standard transversal order (raster order) worked essentially as well as any other scheme.

7.5.2. *Architecture.* The architecture of ViT was deliberately similar to that of original Transformer [VSP+17] and BERT, in order to demonstrate the flexibility of the architecture in transferring from NLP to computer vision. Three different model sizes were considered in [DBK+20]

Model	**Layers**	**dimension** (d)	**Heads** (h)	$d_K = d_V$	**Params**
Base	12	768	12	64	86M
Large	12	1024	16	64	307M
Huge	32	1280	16	80	632M

TABLE 7.5.1. Model sizes of Vision Transformers trained in [DBK+20].

7.5.3. *Training.* ViT was *supervised pre-training* on a labeled dataset and then supervised fine-tuning on downstream tasks. Notable, this constrasts with the training procedure for Transformers used for LLMs, in which the pre-training is done on *unlabeled* data.

The pre-training was done successively on datasets ImageNet (1.3M images), ImageNet-21k (14M images), and JFT (303M images). The best performances were obtained by a Huge ViT with patch width $P = 14$, and a Large ViT with $P = 16$.

Performance on ImageNet was measured over the course of pre-training and compared to CNN ResNets, indicating that CNN ResNets perform better at small data scales, but plateau earlier and are eventually surpassed by the best Vision Transformer – see Figure 7.5.1. This fits with the intuition that the CNN's architecture accommodates inductive biases relevant to image recognition, hence requires less training to perform well. However, it also suggest that at sufficient scales, more generic models eventually learn these inductive biases just as well, and then even benefit from the flexibility of not being architecturally constrained.

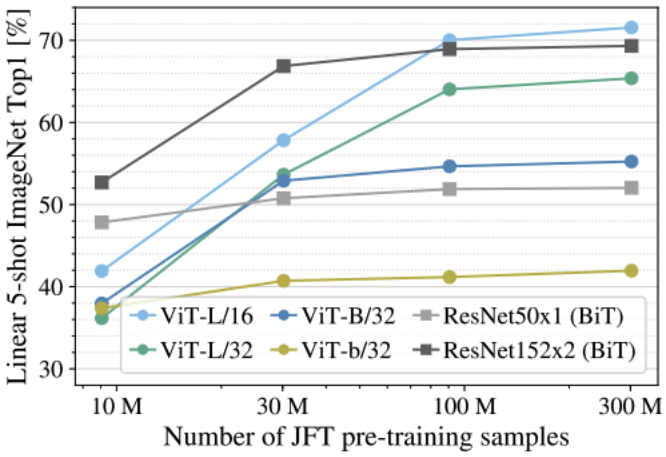


FIGURE 7.5.1. Performance on ImageNet of various models at different scales of pre-training. ViT-L/P (resp. ViT-B/P) is the large (resp. base) ViT with patch width P. Deep ResNets dominate at 10M and 30M pre-training samples, but are eventually surpassed by ViT-L/16 after 100M pre-training samples. From [DBK+20, Figure 4].

7.5.4. *Interpretation.* One can construct *Attention maps* to visualize the patches which contribute to the final classification (e.g., using Attention Rollout [AZ20]); see Figure 7.5.2. In these Attention maps, pixels in the original image are highlighted according to their weight in the "superposition" (in terms of the discussion of §7.1.1).

To compare with CNNs, it is interesting to measure the "mean Attention distance" through the network, which is analogous to the size of the receptive field in CNNs. This is is displayed in Figure 7.5.3. In the early layers of the network, a wide range of distances is witnessed by different Attention heads; as the network depth increases, the mean attention distances coalesce around the upper limit of the initial spread.

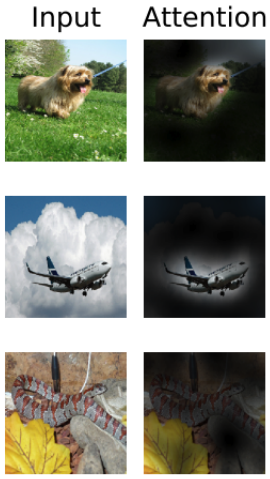


FIGURE 7.5.2. Some samples of Attention maps, from [DBK+20, Figure 7].

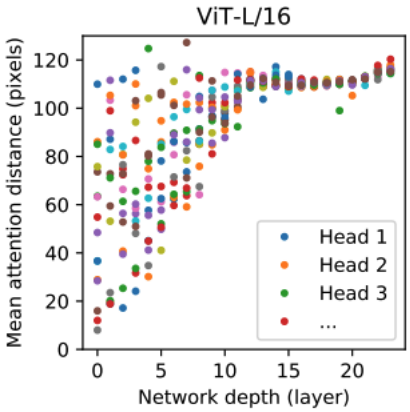


FIGURE 7.5.3. Plot of mean attention distance against network depth, from [DBK+20, Figure 7].

Left for presentation instead of lecture

7.6. **Scaling transformers: Transformer-XL.** The original Transformers had small context windows, capped at hundreds of tokens (e.g., 512 for BERT). For comparison: at the time of writing these notes, modern LLMs have context windows extending into the hundreds of thousands or even millions. A basic bottleneck of Attention is that it scales quadratically with the length of the context window.

As a case study, we will examine an early effort towards scaling effective context length: Transformer-XL by Dai et al [DYY+19]. This paper introduced two main ideas. Firstly, in order to combat the *context fragmentation* problem in which contextual meaning is abruptly lost beyond the context window, they introduced a *recurrent mechanism* by injecting a hidden state into the Attention computation. This served to cache prior context in memory, to some extent. While this trick did work well in the experiments of [DYY+19], which extended the effective context window from hundreds to thousands, it is not really used in modern LLMs due to the scaling limitations of recurrent mechanisms. The basic idea of caching previous computations remains an important part of optimizing and scaling. However, this requires some modification to the computation of Attention.

Consider calculating Attention at inference time: the context window shifts by 1 for each output token, which means that one is almost redoing many Attention computations. However, this is not literally true because of the positional embeddings. The second, lasting, innovation of [DYY+19] is to replace the positional embeddings by "relative positional embeddings" in order to solve this problem.

Indeed, recall that in a Transformer, each token x_i is embedded as E_i and then added to a positional embedding U_i. We can think of E_i as capturing the "content" of x_i, and U_i as capturing the "position". The Attention between x_i and x_j is then

$$(E_i + U_i)W_K W_Q^\top (E_j + U_j)^\top.$$

We can expand this expression as

$$\underbrace{E_i W_K W_Q^\top E_j^\top}_{(a)} + \underbrace{E_i W_K W_Q^\top U_j^\top}_{(b)} + \underbrace{U_i W_K W_Q^\top E_j^\top}_{(c)} + \underbrace{U_i W_K W_Q^\top U_j^\top}_{(d)}. \tag{7.6.1}$$

We can think of (a) $E_i W_K W_Q^\top E_j^\top$ as a "relative content" term, which does not depend on the positions. On the other hand, the (d) term $U_i W_K W_Q^\top U_j^\top$ depends on the absolute position indices (i, j), which change as the context window shifts, whereas intuitively only the difference $j - i$ should matter. Therefore, we would like to alter this term to only depend on the relative position. The final form of relative attention from [DYY+19] is

$$\underbrace{E_i W_K W_Q^\top E_j^\top}_{(a')} + \underbrace{E_i W_K u^\top}_{(b')} + \underbrace{R_{j-i} W_R W_Q^\top E_j^\top}_{(c')} + \underbrace{R_{j-i} W_R v^\top}_{(d')}. \tag{7.6.2}$$

Here R_{j-i} is learned embedding of $j - i$, and u, v are learned vectors ("global" in the sense that they are independent of i, j). Let's talk through the new interpretation of these terms. The "relative content" term (a) is the same as (a'). The term (d') is a "relative position" term, with learnable parameters R_{j-i} and v. Since v is independent of i and j, it captures "global" bias about relative position; for example, we can expect it to learn that "the previous word is important". Similarly, (b') captures "global" bias about content, such as which words have higher baseline importance (e.g., the tokens `'the'` and `'an'` are perhaps not so important). Finally, (c') captures content-based positional bias, such as which positions tend to be important for a given token. For a concrete example: in Spanish, adjectives tend to follow the nouns they describe, so if E_j is the encoding of an adjective, then the preceding word is especially important.

Part 3. Generative models

8. Large Language Models

In this lecture, we will survey the development of Large Language Models (LLMs).

8.1. **Pre-trained transformers.** A distinctive feature of modern LLMs is their massive Pre-training phase. To understand the backdrop for pre-training, recall that Vaswani et al introduced the Transformer as a model for machine translation. However, one would like a model capable of handling a variety of NLP tasks, including translation, text summarization, question answering, etc. By the late 2010s, benchmarks covering many diverse tasks were collected, such as GLUE (General Language Understanding Evaluation) in 2018, which featured 9 different types of natural language understanding tasks. This reflected a shifting emphasis from individual benchmarks to general capability. What might be a uniform system for handling all such tasks?

The idea of OpenAI's 2018 GPT-1 (short for Generative Pre-trained Transformer) [RN18] and Google's 2019 BERT [DCLT19] was to use a common *pre-training* procedure for representation learning of natural language, and then do separate *supervised fine-tuning* for each downstream task: see Figure 8.1.1. As a metaphor, pretraining could be thought of as analogous to the common "primary schooling" for children, while supervised fine-tuning is analogous to specialized advanced education that students undertake after primary school.

Crucially, the pre-training is done on *unlabeled data*, which makes it feasible at massive scales (e.g., by scraping the whole Internet for data) in comparison to supervised learning on labeled data, which is costly to curate. In this sense, pre-training is a form of unsupervised learning, although a more suggestive name might be *self-supervised learning*: unlabeled data

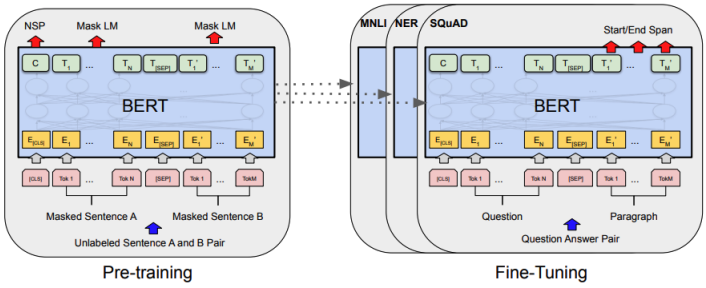


FIGURE 8.1.1. The training strategy for BERT. Left: massive-scale unsupervised pre-training results in robust representation learning. Right: for individual tasks, the model is initialized with pre-trained parameters and then undergoes small-scale supervised learning. From [DCLT19, Figure 1].

is automatically transformed into labeled data by the nature of the pre-training objective, and then subjected to a training process that looks identical to supervised learning.

Later, OpenAI demonstrated with GPT-2 and GPT-3 that sufficiently pre-trained models could already perform decently at downstream tasks *without* supervised fine-tuning. Of course, modern frontier models do undergo significant pre-training *and* post-training (including supervised fine-tuning).

Remark 8.1.1. Let us comment on the historical development of pre-training. The idea of pre-training has been around for a long time, but its signifance has changed over the years. It was emphasized already in early work of LeCun et al [LBOM12] that good initialization of weights is important for training. The 2006 *Science* paper of Hinton and Salakhutdinov [HS06] introduced and validated the important idea of initialization via *pre-training via unsupervised learning on raw data*, although its methodology for doing so was very different from that of modern LLMs. In [HS06], individual layers were pre-trained separately using Restricted Boltzmann Machines, and their weights were then loaded to initialize a neural network, while modern LLMs are pre-trained end-to-end.

With the development of better initialization schemes and optimizers, the importance of pre-training as a mechanism for weight initialization diminished. Instead, pretraining is now viewed as a form of self-supervised *representation learning* in the sense of §6.5, in which the model learns useful representations by identifying patterns in the raw data.

8.2. **Token embeddings.** In order to do machine learning on natural language, we need to convert words (or more precisely, *tokens*) into quantitative information, i.e., vectors of numbers. This is a form of representation learning commonly called *token embeddings*, that was already necessary for RNNs and vanilla Transformers, but we previously skirted it. Now it is timely to return to this point; this will be our first example of *unsupervised learning*.

The fundamental idea is that unlabeled natural language text can be automatically transformed into certain kinds of labeled data. We will illustrate some examples from *word2vec*, an technique for representation learning that was an ideological predecessor for the pre-training procedure of LLMs.

- "Bag of words": predict a target word from surrounding context words. For example, suppose we find the sentence

```
The cat sat on the mat.
```

We can choose a random word to mask, say `sat`, and create a pair

$$(\texttt{the cat \_\_\_ on the mat}, \texttt{sat})$$

The model will be shown the first member of the pair, and asked to predict the second member.

- "Skip-gram": predict context words surrounding a target word. For example, from the sentence "`the cat sat on the mat`" we might extract the target word `cat` and context words {`the, sat, on, mat`}. The training objective is then to maximize the log likelihood of predicting the context words given the target word,

$$\sum_{w_j \in \text{context words}} \log p_E(w_j | \text{target word}).$$

Note that, crucially, *the data for each of these approaches is easy to obtain*: it can be extracted from unlabeled natural language text.

Now we will, this one time only, walk through some low-level implementation details.

8.2.1. *Tokenization.* As a first remark, LLMs do not actually operate on words but on more mathematically engineered units of meaning called *tokens*. The conversion from words to tokens, called *tokenization*, can be thought of as a data pre-processing step. Some illustrative examples of tokenization might be:

$$\begin{aligned} \texttt{cooked} &\mapsto [\texttt{cook}, \texttt{ed}] \\ \texttt{cooking} &\mapsto [\texttt{cook}, \texttt{ing}] \\ \texttt{uncooked} &\mapsto [\text{un}, \texttt{cook}, \texttt{ed}] \end{aligned}$$

Tokenization is carried out algorithmically; a common tokenization scheme (used for GPT-1) is *Byte Pair Encoding*. The basic idea is to start with a vocabulary of characters and a body of reference text, and then repeatedly merge the two most frequent tokens from the vocabulary which appear in the reference text. This is basically a greedy algorithm for minimizing the number of tokens needed to express the reference text.

8.2.2. *One-hot encoding.* Next, suppose we have a vocabulary $\mathcal{V}$ of all possible tokens $x_1, x_2, \ldots, x_N$. We cannot directly feed these into neural networks, because those operate on *numbers* rather than words or tokens. A cheap way to convert the x_i into vectors of numbers is the so-called "one-hot embedding" $\mathcal{V} \to \mathbf{R}^N$, sending x_i to the vector whose ith coordinate is 1 and all other coordinates are 0.

$$\begin{aligned} x_1 &\mapsto [1, 0, 0, 0, \ldots] \\ x_2 &\mapsto [0, 1, 0, 0, \ldots] \\ x_3 &\mapsto [0, 0, 1, 0, \ldots] \\ &\vdots \qquad\qquad \vdots \end{aligned}$$

More generally, we can use one-hot encoding anytime we have *categorical* rather than quantitative features.

8.2.3. *word2vec.* Finally, to do the actual training we pick a latent dimension D and learn a linear projection $v \mapsto vE$ for $E \in \mathbf{R}^{N \times D}$. To learn E, we view it as the weights for a linear model of the form

$$\text{one-hot}(x)E$$

which is then trained on data $\mathcal{D} = \{(x_i, y_i)\}$ obtained from "bag-of-words" and "skip-gram" applied to unlabeled natural language text.

left for presentation instead of lecture

8.3. **Case study: GPT-1.** The most famous Large Language Models are OpenAI's GPT (Generative Pre-trained Transformer) series. Here we highlight some aspects of GPT-1 [RN18].

8.3.1. *Pre-training.* As in word2vec, the pre-training phase of GPT-1 was based on tasks extracted from unlabeled natural language text. Specifically, its objective was "next-token prediction": given a sequence

$$x_{1:T-1} = x_1 x_2 \dots x_{T-1},$$

the neural network learns to predict a conditional probability function on the next token, denoted

$$p_\theta(x_T | x_{1:T-1}).$$

Recall that this came up already in the discussion of RNNs (§6.4). The training is carried out using cross-entropy loss. Namely, given a training pair $(x_{1:T-1}, x_T)$, the empirical distribution is the delta distribution δ_{x_T} concentrated at x_T, so the cross-entropy against the model is

$$H(\delta\{x_T\}, p_\theta) = -\log p_\theta(x_T | x_{1:T-1}).$$

Note that, as with word2vec, such training data is easy to acquire: any unlabeled corpus of natural text can be converted into training pairs by extracting sequences of T tokens and then treating the last token as the label.

8.3.2. *Decoder-only architecture.* The GPT-models are *decoder-only* Transformers: in terms of the discussion of §7.3, they consist only of the decoder half. Indeed, their main task is *autoregressive generation* – generating sequences of tokens based on their own previous output – which does not really have a separate "input processing" stage like machine translation does. Note that the decoder half does have its own embedding layer, which partially plays the role of representation learning.

For concrete details, GPT-1 had 12 layers and $h = 12$ attention heads with $d_K = d_V = 64$, so model dimension $12 \times 64 = 768$. It was trained over 100 epochs, with minibatches of size 64.

8.3.3. *Supervised fine-tuning.* To transfer from pre-training to downstream tasks, GPT-1 underwent *supervised fine-tuning* on each specific task on the pre-trained model. This was carried out by extracting the portion of the pre-trained transformer up through the final attention layer, adding a new fully-connected layer on top of it, and then doing standard supervised machine learning on a labeled dataset (see Figure 8.3.1).

The supervised fine-tuning data for GPT-1 came from publicly available datasets, which had been curated for benchmarking on various NLP tasks such as translation, question answering, summarization, and recognizing semantic similarity.

left for presentation instead of lecture

8.4. **Case study: BERT.** In contrast to GPT-1, Google's BERT (short for Bidirection Encoder Representations from Transformers) [DCLT19] was effectively an *encoder-only* transformer (the encoder half of a transformer as described in §7.3).

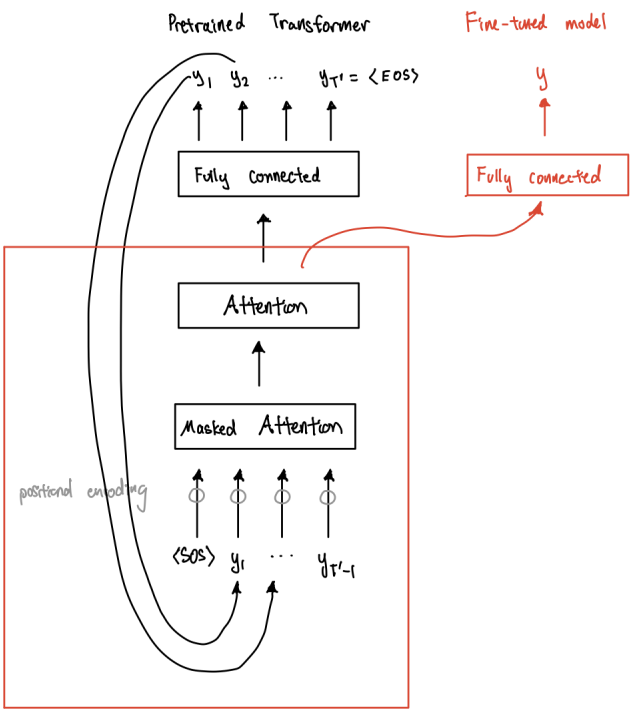


FIGURE 8.3.1. GPT-1 was pre-trained on a massive natural language corpus. Later, for each downstream task, the final attention head layer was connected to new fully connected layers, whose parameters were then trained on a smaller labeled dataset in the manner of ordinary supervised learning.

8.4.1. *Pre-training tasks.* The main novelty of BERT, compared to GPT-1, was that its pre-training objective consisted of "bidirectional" tasks where the model learns contextual relations flowing both forwards and backwards in the sequence (this is the meaning of "bidirectional"). The two tasks were called "Masked Language Model" and "Next Sentence Prediction".

(1) *Masked Language Model.* BERT was given passages of natural language text with some of the tokens randomly masked (e.g., replaced by a special `[MASK]` token), and its training objective was to guess the masked token from context. The earlier GPT-1 had pre-trained on "next token prediction", which uses only the preceding context, while Masked Language Modeling involves prediction based on *both* the left context and the right context.

Actually, the masking procedure was a bit subtler. If done in the naive way described above, there would be a mismatch between text seen in the training distribution versus downstream tasks where the `[MASK]` token does not appear. Therefore, what actually happened was that 15% of the tokens were selected at random for masking. Of these, only 80% actually got replaced by the `[MASK]` token, 10% got replaced by a random token, and 10% were left unchanged. In a single training example, the model is asked to predict all masked tokens and the loss is the average cross-entropy loss for each prediction.

Example 8.4.1 ([DCLT19, Appendix A.1])**.** Suppose the input sentence is

`my dog is hairy`

and that the word `hairy` is randomly selected for masking.

- 80% of the time, it is replaced with `[MASK]` token, so the model sees

`my dog is [MASK]`

- 10% of the time, it is replaced with a random word, say `apple`, so the model sees

`my dog is apple`

- 10% of the time, it is left unchanged, so that the model sees

`my dog is hairy`

(2) *Next Sentence Prediction.* In order to build up higher level understanding, BERT was also given sentence-level tasks. Specifically, the model was given two sentences A and B drawn from a natural language corpus, where B was the next sentence after A 50% of the time, and a random sentence from the corpus the other half of the time. The model's objective was simply to classify whether B `isNext` or is `notNext`.

Example 8.4.2 ([DCLT19, Appendix A.1])**.** For the input

$A =$ `the man went to [MASK] store`
$B =$ `he bought a gallon [MASK] milk`

the ground-truth label is `isNext`.

For the input

$A =$ `the man [MASK] to the store`
$B =$ `penguin [MASK] are flightless birds`

the ground-truth label is `notNext`.

8.4.2. *Hyperparameters.* We will discuss two BERT models: a base model and a large model.

The base model was chosen to be comparable to OpenAI's GPT-1. It had 12 layers, $h = 12$ attention heads, dimension $d = 768$ (so $d_K = d_Q = d_V = 64 = d/h$), for a total of about 110M parameters.

The large model had 24 layers, $h = 16$ attention heads, and $d = 1024$ (so again $d_K = d_Q = d_V = 64$), for a total of about 340M parameters.

8.4.3. *Training.* The pre-training corpus for BERT consisted of BooksCorpus (a dataset of about 7000 self-published books with 800M words) and English Wikipedia (about 2.5B words). These were divided into input sequences of ≤ 512 tokens. The batch size was 256, so each batch consisted of up to $128,000$ tokens. Training lasted for 40 epochs over a 3.3 billion word corpus.

Pre-training was done with the Adam optimizer, with hyperparameters $\mu = 0.9$, $\beta = 0.999$. For regularization, BERT used L^2 weight decay with $\lambda = 0.01$ and dropout with probability $p = 0.1$.

8.4.4. *Ablation studies.* Ablation studies are a way of analyzing the contribution and importance of different components of the model and training pipeline. The idea is to remove ("ablate") or change individual parts, and then measure how this affects performance. Here are some ablations that [DCLT19] considered.

- *Pre-training tasks.* Remove the NSP objective, or change the Masked Language Modeling task from bidirectional to unidirectional (left-to-right only). These were all found to hurt performance significantly [DCLT19, Table 5].
- *Model size.* Increasing the number of layers and attention heads (fixing $d_K = d_V = d_Q$) led to smooth improvements in performance.
- *Fine-tuning approach.* Instead of fine-tuning the pre-trained model directly, there is a *featured-based approach* to using a pre-trained model. This involves plugging the pre-trained model in as features to another task-specific architecture. This also achieved good performance with BERT, but not quite as good as the fine-tuning approach.

8.5. **GPT-2, GPT-3, and beyond.** Both GPT-1 and BERT train separate models for different downstream tasks, after a common pre-training phase. A key paradigm shift occurred with GPT-2 [RWC$^+$19], which demonstrated that a single, sufficiently pre-trained model could handle diverse downstream tasks *without* separate specialization through fine-tuning.

Recall that in the GPT-1/BERT paradigm, for each task τ we obtain a separate model through supervised fine-tuning, that calculates

$$p_\theta^\tau(x_T|x_{1:T-1}).$$

Alternatively, we can think of this as a single model that conditions on both input *and* task, $p_\theta^\tau(x_T|x_{1:T-1}, \tau)$. The insight of GPT-2 was that it can be effective to condition on the task *simply by adding the task description to the input context.*

Example 8.5.1. Suppose $x_{1:T}$ is

```
The cat sat on the
```

The pre-trained transformer will naturally try to continue this sentence, perhaps with the word `mat`. But suppose that we instead want a translation of this phrase from English to French. In the GPT-1/BERT paradigm, we would take part of the pre-trained transformer, insert it into an architecture for the translation task, and then do supervised learning. But in the GPT-2 paradigm, we instead augment the context with something like the following.

```
Your task:  Translate the following English text to French.
English text:  The cat sat on the
French text:
```

The model would then understand to start translating into French, rather than completing the English sentence.

This idea was taken further in GPT-3 [BMR$^+$20], which studied the effect of including sample demonstrations of the desired task into the prompt, a technique called *few-shot prompting.* In fact, an intriguing result from GPT-3 was that for sufficiently large models, if enough (e.g., 10) samples are included in the prompt then even the original task description becomes unnecessary. However, this effect was only observed in sufficiently large models: see Figure 8.5.1, which shows that a small (1.3B parameter) model does not benefit from few-shot examples, while a large 175B parameter model benefits much more than a medium 13B parameter model. This gave rise to the concept of *emergent behavior*: qualitative

capabilities "emerge" only in sufficiently large models like a "phase transition", not predicted by extrapolation from scaling.

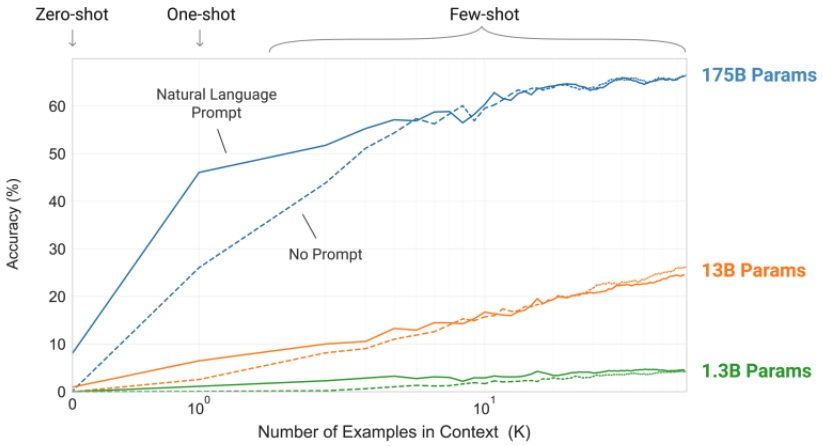


FIGURE 8.5.1. Graph of performance against number of few-shot samples included in context, over several different model sizes, from [BMR+20, Figure 1.2]. The small model (1.3B parameters) does not seem to be able to learn from context. By constrast, larger models learn well from context, to the point where the task-specific prompt is no longer necessary.

The papers [RWC+19] (GPT-2) and [BMR+20] (GPT-3) showed that massive models with extensive pre-training could perform well on a diverse collection of downstream tasks *without any supervised fine-tuning at all*. The driving force behind performance was simply scale: the growth of various parameters from GPT-1 to the largest GPT-2 and GPT-3 models is indicated in Table 8.5.1.

	GPT-1	**GPT-2 (1.5B)**	**GPT-3 (175B)**
Publication Year	2018	2019	2020
Parameters	117M	1.5B	175B
Layers	12	48	96
Hidden size (d_{model})	768	1600	12288
Attention heads	12	25	96
Context length	512	1024	2048
Batch size	64	512	?
Tokens per batch	32,768	500k	3.2M
Training data size	BooksCorpus (4.5 GB)	WebText (40 GB)	CommonCrawl, etc. (570GB, 300B tokens)

TABLE 8.5.1. Largest configurations of GPT-1, GPT-2, and GPT-3.

Of course, modern LLMs today do involve a significant amount of supervised fine-tuning on labeled data. This is part of the "post-training" phase (along with Reinforcement Learning, which we haven't discussed yet). However, instead of fine-tuning separate models as in

GPT-1 and BERT, all supervised fine-tuning is conducted on the same pre-trained model, using context to condition on the task. Supervised fine-tuning requires labeled data, which as we have mentioned is costly to curate. Data labeling itself has become a huge industry, enough so that pure labeling companies like Scale AI have arisen to meet the demand.

8.6. **Data.** Modern LLMs are pre-trained on trillions of tokens. This requires a huge corpus of natural language text, which is scraped from the internet. For example, even as of 2019 the Common Crawl dataset contained hundreds of billions of words.

However, the Common Crawl contains a large proportion of low-quality data (i.e., unintelligible contents), which is counterproductive for training. Significant effort is required for filtering, cleaning, and deduplication of pre-training data. Thus, data curation is a significant part of the LLM development pipeline by itself. In fact, modern AI labs tend to have separate divisions for Data curation, Pre-training, and Post-training.

8.6.1. *Filtering.* OpenAI sketches in [BMR+20, Appendix A] the classifier used to filter Common Crawl data for GPT-3 pre-training. It is a form of negative sampling, where "verified" high-quality data such as WebText, Wikipedia, and online books are used as positive examples, and random instances from Common Crawl are used as negative examples.

8.6.2. *Deduplication.* Since some internet data is repeated over many different webpages, retraining on the same data risks overfitting. Another consideration is that many benchmarks can be found on the internet, so pre-training on them would contaminate performance. Thus, another important step is in filtering training data is *deduplication*, i.e., removing (almost) duplicate data. A classic method for doing this is hashing, but one can bet that the major AI labs have developed more sophisticated deduplication procedures.

8.6.3. *Data mixtures.* When we discussed the vanilla template for LLMs in §1, we implicitly assumed that datapoints were drawn i.i.d. However, the reality is that not all data on the internet is created equal. During pre-training, data is weighted according to its quality. For example, articles from Wikipedia and ArXiv would be considered "higher-quality" compared to average internet data.

The data mixture for training GPT-3 is depicted in Figure 8.6.1. Here, WebText2 is

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

FIGURE 8.6.1. The data mixture used to train GPT-3. From [BMR+20, Table 2.2].

the successor to a new dataset that OpenAI created when building GPT-2 [RWC+19]. The original idea was to scrape outbound links from Reddit posts, using karma as a proxy for human-labeled quality.

On average, pre-training runs for less than one epoch, meaning that not even all the training data is used. However, the highest quality data (e.g., Wikipedia data) is reused several times.

It is also beneficial to schedule pre-training so that higher quality data, which is typically more advanced, is used later. This can be viewed as a form of "curriculum learning", in which the model is given harder tasks when it is more ready for them.

8.7. **Scaling laws.** As we have already discussed, the gains from GPT-2 and GPT-3 were obtained by scaling the size and training of models. Especially for GPT-3, this was predicted by experimental investigation of "scaling laws", which continue to motivate the scaling of Large Language Models.

The first systematic investigation for scaling laws in neural-network based LLMs was carried out by Kaplan et al [KMH+20]. They studied the behavior of performance (measured by test loss) as a function of the compute budget C (measured in total FLOPs – floating point operations), data size D, and model size N (number of parameters). These are related approximately by the formula $C \approx 6ND$. The important qualitative conclusions are as follows.

- **Scale, not shape**. Performance depends strongly on the size of the parameters, and weakly on architectural hyperparameters such as depth versus width. According to the "Chinchilla Scaling Laws" of Hoffman et al [HBM+22], for a fixed C it is optimal to scale D and N equally.[21]
- **Predictable performance gains.** Validation loss exhibits a power law relationship with each of the three scale factors (when bottlenecked by the other two).
- **Predictable performance transfer**. When the model is evaluated on out-of-training-distribution text, its performance is highly correlated with validation loss. In fact, the out-of-distribution test loss is essentially a constant shift of validation loss.
- **Optimal batch size is predictable.** The optimal batch size is roughly a power of the (cross-entropy) loss, and is determinable by the noise scale of the gradient [Kap, §4.5].

We will sketch how Hoffman et al [HBM+22] investigated scaling laws. They tried three different approaches, each leading to the conclusion that optimal scaling should obey a power law of the form $\hat{N} \propto C^{1/2}$, $\hat{D} \propto C^{1/2}$.

(1) Fix a sequence of models of different sizes, and plot their training loss up as FLOPs vary. Each model size produces a graph of training loss versus FLOPs: see Figure 8.7.1.[22] For each FLOP count, one can take the model with lowest training loss as the optimal size. This interpolates to a power law estimate of optimal model size versus FLOPS; and one can similarly interpolate the optimal training tokens versus FLOPS.
(2) Fix a sequence of FLOP budgets, and plot the training loss as model size varies: see Figure 8.7.2. Then one can directly see the optimal model size for a given FLOP budget. Plotting this, there is a clear power law relationship between FLOPs and optimal model size.
(3) Interpolate a prediction for loss as a function of the form $L(N, D) = E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}$. Here E captures an irreducible error from the nature of the task; the other

[21]This amends [KMH+20], which suggested that for every 10x C, D should 1.8x while N should 5.5x. See [PS24, PWJ+24] for analyses of this discrepancy, which suggest that it was due to incorrect accounting in [KMH+20].

[22]In fact, each model size was trained with 4 different learning rates, so there would 4 graphs per model size.

terms express inverse power law contributions with exponents to be empirically determined.

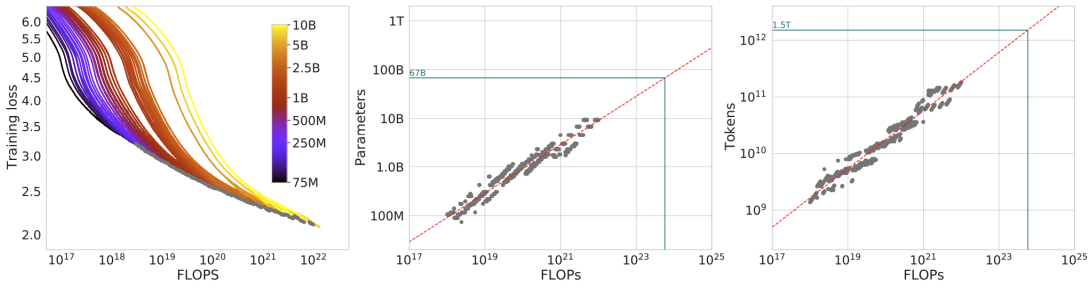


FIGURE 8.7.1. Approach 1 of Hoffman et al [HBM+22, Figure 2] for finding the optimal scaling laws. Left: graph of training loss for various model sizes as a function of FLOPs. Middle: for each FLOP count, find the model size with lowest training loss and plot it. There is a clear log-linear relationship, with slope 0.5, between FLOPs and parameters. Right: for each FLOP count, calculate the training tokens for the model with lowest training loss. Again, there is a clear log-linear relationship, with slope 0.5.

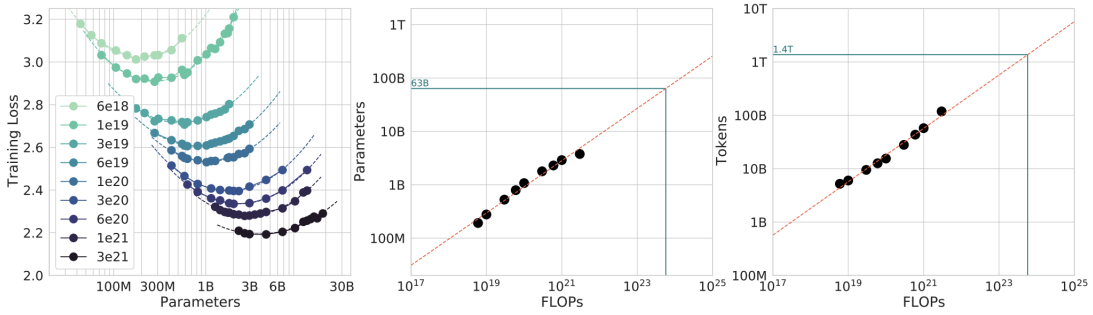


FIGURE 8.7.2. Approach 2 of Hoffman et al [HBM+22, Figure 2] for finding the optimal scaling laws. Left: graph of training loss versus parameter size for various compute budgets. Middle: for each FLOP count, find the model size with lowest training loss and plot it. There is a clear log-linear relationship, with slope 0.5, between the FLOPs and parameters. Right: for each FLOP count, calculate the training tokens for the model with lowest training loss. Again, there is a clear log-linear relationship, with slope 0.5.

9. Generative Adversarial Networks

We will now move on to self-supervised generative models designed for *image generation*. The next two lectures will cover Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and Diffusion models. We begin with GANs because they are simple and intuitive (although historically they actually came after VAEs), and quite flexible.

To motivate GANs, we go back to the idea of representation learning we first encountered in the discussion of RNNs (§6.5). We can view realistic image data as forming a thin manifold M in some enormously high-dimensional space $\mathcal{X}$ of all possible images (which would be $\approx 10^9$-dimensional just for 224×224 pixel RGB images, as estimated in §5.1). The idea is to construct a "chart" from latent space $g\colon \mathcal{Z} \to M$, and then to sample from M by sampling $z \in \mathcal{Z}$ (from some standard distribution, e.g., Gaussian) and then forming $x = g(z)$. We will find g by choosing a parametric family $g_\theta(z)$, given by a neural network, and optimizing over θ. But how do we formulate a "loss" for g_θ if we only have access to *unlabeled* image data?

9.1. **Generator-discriminator approach.** The ingenious idea behind GANs, introduced by Goodfellow et al [GPAM$^+$14], is to insert the function g_θ into an adversarial two-player game, where the optimal behavior of g_θ will be to produce outputs resembling real images. In order to do this, we introduce an auxiliary function $d = d_\phi$, itself parametrized by a neural network, which predicts the probability of an image being real. Then d_ϕ serves as the "adversary" of g_θ.

We henceforth call g_θ the "generator", because its role is to generate synthetic images, and d_ϕ the "discriminator", because its role is to discriminate between synthetic images and real ones. Imagine this as a game where:

(1) The generator creates an image.
(2) The discriminator receives an image which has 50% chance of coming from the generator, and 50% chance of being a real image.
(3) The discriminator wins a point if it correctly classifies the image as real or synthetic, and the generator wins a point if the discriminator is incorrect.

Then the Nash equilibrium strategy is for the generator to create images that are indistinguishable from real ones, in which case any discriminator has only 50% chance of success.

Concretely, say that on an input image x, the discriminator computes $d_\phi(x) = 1$ if it thinks x is real, and $d_\phi(x) = 0$ if it thinks that x is synthetic. Then the discriminator's cross-entropy loss function is

$$\mathcal{L}(\phi, \theta) = -\mathbb{E}_{x_{\text{real}} \sim P_{\text{real}}}[\log d_\phi(x_{\text{real}})] - \mathbb{E}_{z \sim P_{\mathcal{Z}}}[\log(1 - d_\phi(g_\theta(z)))] \tag{9.1.1}$$

where P_{real} is the probability distribution of real image data, and $P_{\mathcal{Z}}$ is a pre-chosen distribution on latent space, usually a Gaussian $\mathcal{N}(0, \mathrm{Id})$. In practice, we can compute this by sampling a batch x_{real} from the real data $\mathcal{D}$, and another batch of equal size $x_{\text{synth}} \sim g_\theta(z)$ from the generator, and averaging the cross entropy loss over their mixture.

The discriminator wants to minimize its loss $\mathcal{L}(\phi, \theta)$, while the generator wants to maximize it. Thus, the discriminator performs gradient descent on its parameters ϕ as usual, while the generator performs gradient *ascent* on its parameters θ (because it is trying to construct adversarial examples, cf. §5.4).

Remark 9.1.1. Although we initially rooted our discussion in image generation, which was historically the first domain of application for GANs, note that this framework of adversarial training is actually very general, and makes sense in any domain.

Indeed, we have been agnostic about the actual architecture for the generator and discriminator. The GAN is really a "meta-architecture" in that it specifies a way to jointly train two sub-neural-networks, but not the specific architectures of either. That would be domain-dependent, and for image data the natural choice would be CNNs.

9.2. **Pitfalls.** GANs have several problems in practice. First of all, their training dynamics can be unstable, with generator and discriminator circling around the equilibrium but not converging (quickly) to it.

Also, if the generator and discriminator are imbalanced in capability, then training can be very slow.

Finally, training can reach bad equilibria called "mode collapse", where the generator learns to create a distribution that is concentrated on a few examples, instead of the true distribution P_{true}. As a toy example, imagine training a model to generate handwritten digits. The generator could learn to generate a handwritten '7' well, without learning any other digits.

Left for presentation instead of lecture

For all these reasons, it proved to be difficult to get GANs to work well in practice. The work of Radford et al [RMC15] was one of the first to successfully scale up GAN training to the level of producing reasonably realistic images. Their main adaptations seem to be as follows.

- Instead of using fixed pooling functions, Radford et al replaced pooling with learned strided convolutions in the discriminator, and learned upsampling (i.e., "fractionally-strided") convolutions in the generator.
- They eliminated fully connected layers typically placed on top of convolutional layers, instead connecting the final convolutional layers directly to the input of the generator, and output of the discriminator.
- They applied Batch Normalization, but somewhat selectively: they did *not* apply it to the output layer of the generator, nor to the input layer of the discriminator.

In Figures 9.2.1 and 9.2.2 we display some examples from the generator learned in [RMC15], which give some interesting insight about latent space. Firstly, Figure 9.2.1 demonstrates that when z moves around continuously in latent space, $g_\theta(z)$ more or less stays within the data manifold of realistic images. Secondly, Figure 9.2.2 demonstrates an interesting property that vector addition in latent space is meaningful.

10. Variational Autoencoders

Variational Autoencoders (VAEs) were introduced by Kingma et al in [KW13] and initially applied to image generation. The abstract setup is the same as for GANs: we want to be able to generate artificial samples from some true distribution $P_{\text{real}}(x)$, which we think of as being supported on a thin submanifold of a high-dimensional ambient space $\mathcal{X}$. We are given unlabeled data $\mathcal{D}$ which we assume to be i.i.d. samples from $P_{\text{real}}(x)$.

Also similarly to GANs, the idea of VAEs is to learn $P_{\text{real}}(x)$ in terms of latent space. That is, we will learn how to transform a fixed distribution on latent space $\mathcal{Z}$, say a standard Gaussian $\mathcal{N}(0, \text{Id})$, into $P_{\text{real}}(x)$. But unlike GANs, instead of directly learning a mapping $g\colon \mathcal{Z} \to \mathcal{X}$, we will try to learn the conditional density function $p_{\text{real}}(x|z)$ for $P_{\text{real}}(x)$.

10.1. **Conditional density function.** We will use a neural network to parametrize a family of functions $p_\theta(x|z)$ intended to approximate $p_{\text{real}}(x|z)$. But how should we formulate a loss function for training?

The data $\mathcal{D}$ consists of i.i.d. samples from P_{real}, which gives us access to the absolute probability density $p_{\text{real}}(x)$. In principle, the model $p_\theta(x|z)$ for conditional density gives a model for the absolute density:

$$p_\theta(x) = \int p_\theta(x, z)\, dz = \int p_\theta(x|z)p(z)\, dz. \tag{10.1.1}$$

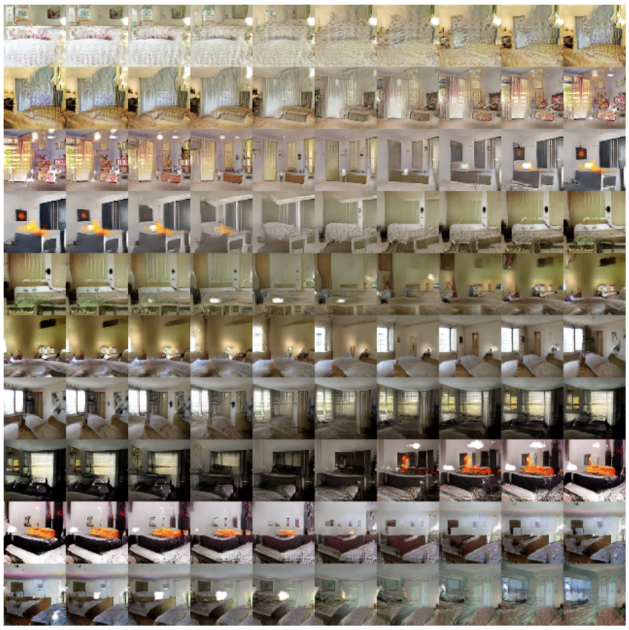


FIGURE 9.2.1. These images are created by choosing 9 different points $z_1, \ldots, z_9$ in latent space, and then applying the generator to a mesh of intermediate points z in latent space interpolating between the z_i. Each image is a plausible candidate for a picture of a bedroom, demonstrating $g_\theta(z)$ moves around smoothly in image space when z moves around smoothly in latent space. From [RMC15, Figure 4].

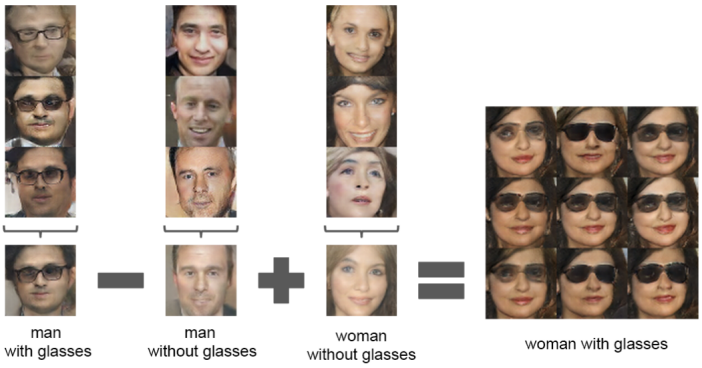


FIGURE 9.2.2. For each column, the latent representations are averaged and vector operations performed on the result to produce $z \in \mathcal{Z}$. The center sample on the right side is the generator's output on z, and the others are produced by applying the generator to $z + \varepsilon$ where ε is a small uniform noise vector. From [RMC15, Figure 7].

Idealistically, we would like to train $p_\theta(x)$ to resemble $p_{\text{real}}(x)$. As discussed in §3.3, we can try to approximate the Maximum Likelihood Estimate

$$\hat{\theta} = \arg\max_\theta \sum_{x_i \in \mathcal{D}} p_\theta(x_i),$$

by minimizing the log likelihood loss

$$-\sum_{x_i \in \mathcal{D}} \log p_\theta(x_i).$$

However, the density (10.1.1) is difficult to work with. Analytically, the integral cannot be evaluated in closed form. What if we tried to approximate it numerically, e.g., by drawing random i.i.d. samples from $p(z)$ and averaging $p_\theta(x|z)$ over them (i.e., Monte Carlo sampling)? Technically, this is an unbiased estimator, but it will be very noisy, and therefore impractical to use.

Let's try to understand why. Intuitively, the prior distribution $p(z)$ is diffuse (typically $p(z) \sim \mathcal{N}(0, I)$), while for a given x, $p_\theta(x|z)$ is highly concentrated. Look back to Figure 6.5.1: for a given word x (e.g., Germany), a randomly sampled $z \in \mathcal{Z}$ has very low likelihood $p_{\text{real}}(x|z)$; only a particular region of latent space is relevant for country names. Hence the Monte Carlo approximation would be dominated by samples landing in the particular region of latent space where the likelihood $p_\theta(x|z)$ is non-negligible. This would give a very noisy (high variance) estimator.

Remark 10.1.1. It would be better to sample from the *posterior* distribution $p(z|x)$ on latent space, as this will be concentrated on z which "explain" x. But Bayes' rule says that

$$p_\theta(z|x) = \frac{p_\theta(x|z)p(z)}{p_\theta(x)},$$

so estimating the posterior is equivalent to estimate $p_\theta(x)$, which is exactly what we were trying to estimate in the first place.

10.2. **The Evidence Lower Bound.** Let q be any probability distribution on latent space $\mathcal{Z}$. Then we may write

$$\log p_\theta(x) = \log \int p_\theta(x,z)\, dz = \log \int q(z) \frac{p_\theta(x,z)}{q(z)}\, dz = \log \mathbb{E}_q\Big[\frac{p_\theta(x,z)}{q(z)}\Big].$$

Since $\log x$ is concave, Jensen's inequality applies and says that

$$\log p_\theta(x) = \log \mathbb{E}_q\Big[\frac{p_\theta(x,z)}{q(z)}\Big] \geq \mathbb{E}_q[\log p_\theta(x,z) - \log q(z)]. \tag{10.2.1}$$

Definition 10.2.1. The *Evidence Lower Bound* (ELBO) of $\log p_\theta(x)$ with respect to $q(z)$ is

$$\text{ELBO}(x|p_\theta, q) := \mathbb{E}_q[\log p_\theta(x,z) - \log q(z)].$$

We just showed in (10.2.2) that $\text{ELBO}(x|p_\theta, q)$ is a lower bound for $\log p_\theta(x)$, i.e.,

$$\log p_\theta(x) \geq \text{ELBO}(x|p_\theta, q) \quad \text{for any probability distribution } q \text{ on } \mathcal{Z}. \tag{10.2.2}$$

Exercise 10.2.2. Show that

$$\log p_\theta(x) = \text{ELBO}(x|p_\theta, q) + \text{KL}(q(z) \;||\; p_\theta(z|x)) \tag{10.2.3}$$

This gives a more precise version of (10.2.2). Moreover, it shows that for fixed x and θ, maximizing ELBO with respect to q is equivalent to minimizing the KL divergence between q and the modeled posterior $p_\theta(z|x)$.

Lemma 10.2.3. *We have* $\mathrm{ELBO}(x|p_\theta, q) = \mathbb{E}_q[\log p_\theta(x|z)] - \mathrm{KL}(q(z) \mid\mid p(z))$.

Proof. Substituting $\log p_\theta(x, z) = \log p_\theta(x|z) + \log p(z)$ into the expression for $\mathrm{ELBO}(x|p_\theta, q)$, we have

$$\begin{aligned}\mathrm{ELBO}(x|p_\theta, q) &= \mathbb{E}_q[\log p_\theta(x|z) + \log p(z) - \log q(z)] \\ &= \mathbb{E}_q[\log p_\theta(x|z)] - \mathrm{KL}(q(z) \mid\mid p(z)).\end{aligned}$$

□

10.2.1. *Encoder-decoder structure.* Note that the lower bound (10.2.1) holds for *every* probability distribution q, so we can even allow q to depend on x. The idea behind VAE is to use $\mathrm{ELBO}(x|p_\theta, q)$ as a proxy for the true log likelihood $\log p_\theta(x)$. The function $q(z) = q_\phi(z|x)$ will be learned by a separate neural network.

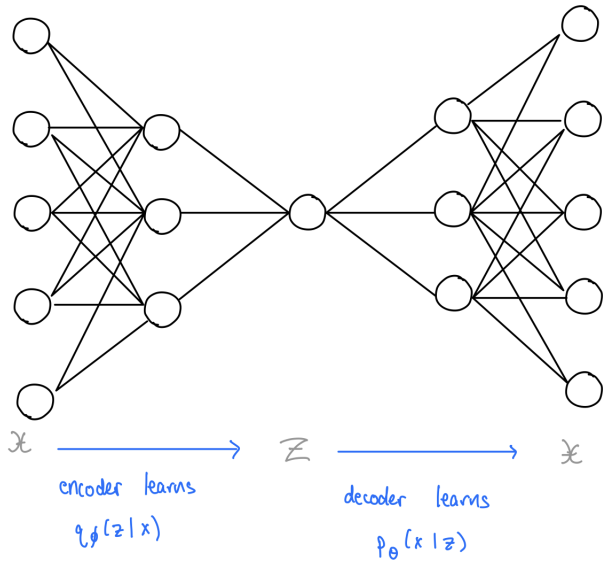


FIGURE 10.2.1. Cartoon of the VAE architecture. The encoder half learns a conditional distribution $q_\phi(z|x)$ on latent space $\mathcal{Z}$, while the decoder half learns a conditional distribution $p_\theta(x|z)$.

Hence the VAE has a structure reminiscent of the encoder-decoder architecture (see Figure 10.2.1): the encoder half learns $q_\phi(z|x)$, the decoder half learns $p_\theta(x|z)$, and the two are jointly trained to maximize

$$\mathrm{ELBO}(x|p_\theta, q_\phi) = \mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - \mathrm{KL}(q_\phi(z|x) \mid\mid p(z)). \tag{10.2.4}$$

Remark 10.2.4 (Autoencoders)**.** Figure 10.2.1 is reminiscent of an *autoencoder*, which is a neural network of the form

$$\mathcal{X} \xrightarrow{E} \mathcal{Z} \xrightarrow{D} \mathcal{X}.$$

that learns to approximate the *identity* function on $\mathcal{X}$. The key point which makes this non-trivial is that $\mathcal{Z}$ has much *lower* dimension than $\mathcal{X}$, so in this process the network learns a compression function $E\colon \mathcal{X} \to \mathcal{Z}$.

In statistics, "variational inference" is a technique for approximating a probability distribution by treating it as an optimization problem. The adjective "variational" in Variational Encoder is a reference to this concept.

10.2.2. *Interpretation.* Let's analyze each of the terms appearing in (10.2.4).

- We can think of the term $\mathbb{E}_{q_\phi}[\log p_\theta(x|z)]$ as a "reconstruction" term: it is the expected value according to our distribution $q_\phi(z|x)$ of the likelihood of z for the data x. Thus, maximizing this term pushes $q_\phi(z|x)$ to be concentrated around the $z \in \mathcal{Z}$ for which $p_\theta(x|z)$ is high.
- We can think of $-\operatorname{KL}(q_\phi(z|x) \,||\, p(z))$ as a "regularization" term: it penalizes $q_\phi(z|x)$ for diverging from the prior distribution $p(z)$. For example, with $p(z) \sim \mathcal{N}(0, \mathrm{Id})$, this prevents $q_\phi(z|x)$ from becoming something like a bump function at the z which maximizes the likelihood of x under p_θ.

Remark 10.2.5. The "β-VAE" of [HMP+16] considers a more general objective function of the form

$$\mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - \beta \operatorname{KL}(q_\phi(z|x) \,||\, p(z))$$

where $\beta > 1$ is a hyperparameter, tuned during hyparameter sweep, that adjusts the balance between the "reconstruction" term and the "regularization" term. Empirically, choosing $\beta > 1$ leads to better performance on certain tasks.

10.2.3. *Functional form of q_ϕ.* We take $q_\phi(z|x)$ to be a product of Gaussian distributions,

$$q_\phi(z|x) = \prod_{j=1}^{d} \mathcal{N}(z_j|\mu_{\phi,j}(x), \sigma^2_{\phi,j}(x)). \tag{10.2.5}$$

The encoder half of the network learns the functions $\mu_\phi(x), \sigma^2_\phi(x)$. Here d is the dimension of the latent embedding space $\mathcal{Z}$.

One reason that we choose q_ϕ of the form (10.2.5) is so that the "regularization" term $\operatorname{KL}(q_\phi(z|x) \,||\, p(z))$ has a nice closed form.

Example 10.2.6. We will calculate the KL divergence between independent Gaussians $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$. Note that

$$\begin{aligned}\operatorname{KL}(\mathcal{N}(\mu_1, \sigma_1^2) \,||\, \mathcal{N}(\mu_2, \sigma_2^2)) &= \operatorname{KL}(\mathcal{N}(\mu_1 - \mu_2, \sigma_1^2) \,||\, \mathcal{N}(0, \sigma_2^2)) \\ &= \operatorname{KL}\Big(\mathcal{N}(\frac{\mu_1 - \mu_2}{\sigma_2}, \frac{\sigma_1^2}{\sigma_2^2}) \,||\, \mathcal{N}(0,1)\Big). \end{aligned} \tag{10.2.6}$$

Finally, we have

$$\begin{aligned}\operatorname{KL}(\mathcal{N}(\mu, \sigma^2) \,||\, \mathcal{N}(0,1)) &= \mathbb{E}\Big[\frac{X^2}{2} - \frac{(X-\mu)^2}{2\sigma^2} - \log \sigma^2\Big] \\ &= \frac{1}{2}(\sigma^2 + \mu^2 - \log \sigma^2 - 1).\end{aligned}$$

Putting this back into (10.2.6), we see that

$$\operatorname{KL}(\mathcal{N}(\mu_1, \sigma_1^2) \,||\, \mathcal{N}(\mu_2, \sigma_2^2)) = \frac{1}{2}\left(\frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{\sigma_2^2} - \log \frac{\sigma_1^2}{\sigma_2^2} - 1\right). \tag{10.2.7}$$

Applying Example 10.2.6, we see that for q_ϕ of the form (10.2.5) and $p(z) = \mathcal{N}(0, \mathrm{Id})$, we have

$$\operatorname{KL}(q_\phi(z|x) \,||\, p(z)) = \frac{1}{2}\sum_{j=1}^{d}(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1). \tag{10.2.8}$$

This gives a simple analytic expression for the regularization term in (10.2.4).

10.2.4. *Functional form of p_θ.* For the reconstruction term, $p_\theta(x|z)$ would also be taken to be of a particular form, depending on the problem.

- For continuous $\mathbf{R}$-valued data, we can take a Gaussian function
$$p_\theta(x|z) \sim \prod_i \mathcal{N}(x_i; \mu_{\theta,i}(z), \sigma^2 I) \tag{10.2.9}$$
where the x_i are the individual pixels of x. Note that this imposes an *independence* assumption on the pixels, which is unrealistic in practice; it leads to a well-known weakness of VAEs that they tend to produce blurry images.

  With $p_\theta(x|z)$ of the form (10.2.9), we have
$$\log p_\theta(x|z) = -\frac{1}{2\sigma^2}||x - \mu_\theta(z)||^2 + \text{ constant},$$
which looks like a standard MSE loss. Indeed, this is the same manifestation of the relationship between MSE loss and Gaussian noise that we saw in §3.4.1.
- For binary data valued in $\{0, 1\}$, we take a Bernoulli distribution
$$p_\theta(x|z) \sim \prod_i \text{Bernoulli}(\theta_i) \tag{10.2.10}$$
and then
$$\log p_\theta(x|z) = \sum_i x_i \log \theta_i + (1 - x_i) \log(1 - \theta_i)$$
looks like the standard binary cross-entropy loss.

One reason we choose $p_\theta(x|z)$ of the above form is so that $\log p_\theta(x|z)$ will have a clean expression in closed form. However, for the ELBO objective we need to evaluate the reconstruction term $\mathbb{E}_{q_\phi}[\log p_\theta(x|z)]$. Now we seem to encounter the same problem as came up in §10.1. Since p_θ and q_ϕ are computed by a neural network, this expression will not be analytically tractable (i.e., expressible in closed form). We could try numerical approximation via Monte Carlo sampling, but didn't we already say in §10.1 that this would be too noisy of an estimator?

Actually, there is a key difference here which makes Monte Carlo sampling viable. Recall how we discussed in Remark 10.1.1 that Monte Carlo sampling would be less noisy if sampled from the *posterior* distribution $p_\theta(z|x)$ instead of the prior distribution $p_\theta(z)$. The intuition is that $q_\phi(z|x)$ will look approximately like $p_\theta(z|x)$. Indeed, from (10.2.3) we can see that for fixed x, the ELBO will be maximized when $\text{KL}(q_\phi(z) \mid\mid p_\theta(z|x))$ is minimized, which occurs exactly when $q_\phi(z|x) = p_\theta(z|x)$. Also, we saw in §10.2.2 that the "reconstruction" term in the ELBO pushes $q_\phi(z|x)$ to be concentrated around the $z \in \mathcal{Z}$ for which $p_\theta(x|z)$ is high.

10.3. **The reparametrization trick.** We have just described how to calculate the objective function
$$\text{ELBO}(x|p_\theta, q_\phi) = \mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - \text{KL}(q_\phi(z|x) \mid\mid p(z)).$$
However, in training we not only need to compute this, but to differentiate it with respect to the learnable parameters θ and ϕ. When computing $\mathbb{E}_{q_\phi}[\log p_\theta(x|z)]$ by Monte Carlo sampling, the dependence on ϕ is rather indirect: we use ϕ to calculate $q_\phi(z)$, and then sample from that distribution to approximate an integral. In particular, it is not clear how to differentiate $z = \text{sample}(q_\phi(z|x))$ with respect to ϕ.

Kingma et al [KW13] address this issue with the so-called "reparametrization trick". Instead of viewing the encoder as outputting a probability distribution, we view it as outputting the mean $\mu_\phi(x)$ and variance $\sigma_\phi(x)$ (which is what it does anyway), and we create from these a random variable

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$

Then z is distributed as $q_\phi(z|x)$. But now z is also a differentiable function of μ_ϕ and σ_ϕ^2, so we can now backpropagate through the encoder network:

$$\frac{\partial z}{\partial \phi} = \frac{\partial z}{\partial \mu_\phi}\frac{\partial \mu_\phi}{\partial \phi} + \frac{\partial z}{\partial \sigma_\phi^2}\frac{\partial \sigma_\phi^2}{\partial \phi}.$$

10.4. **Comparison to GANs.** The Variational Autoencoder consists of two halves: an encoder half that produces distribution $q_\phi(z|x)$ over latent space depending on learnable parameters ϕ, and a decoder half that produces a function $p_\theta(x|z)$. These halves "work together" to jointly maximize the ELBO. GANs also have two halves, but those have an adversarial relationship instead.

Like the GAN, the VAE is a "meta-architecture" in that it specifies a way to jointly train two sub-neural-networks, but not the specific architectures of either; those would be domain-dependent. For example, subnetworks of the β-VAE in [HMP$^+$16] had the same architecture as the GAN in [RMC15], which appeared in §9.2.

The VAE also has its own version of "mode collapse" suffered by GAN (§9.2), called *posterior collapse*, where the regularization term drives the $q_\phi(z|x)$ towards the prior $p(z)$ independently of x, and the decoder learns to ignore z entirely, thus producing the same output regardless of the input.

In practice, VAE tends to produce blurry images. One cause is that the network design forces independence of pixels, and another is that it is ultimately performing some maximum likelihood estimate on Gaussian distributions, whose solution is the sample mean (Example 3.3.4). This natural tendency to "average' results in blurriness: see Figure 10.4.1.

10.5. **Conditional generation.** Suppose we want to do something a little fancier than vanilla image generation, say, generate images from text captions. This is called *conditional* generation. More generally, we can formulate the problem as generation from a conditional probability distribution $P_{\text{real}}(x|c)$.

For this, we need to have *labeled* data $\mathcal{D}$ consisting of pairs (x, c). For example, if we want to generate images from captions, we need to have examples of images with captions, which we could perhaps scrape from the internet. To adapt the VAE, we can keep the same latent space $\mathcal{Z}$, but now our decoder learns a conditional density function $p_\theta(x|z, c)$ while our encoder learns a conditional density function $q_\phi(z|x, c)$. In principle we also have a conditional prior $p(z|c)$, but this is usually taken independent of c, say $\mathcal{N}(0, \text{Id})$. The caption c is fed into both the encoder and decoder.

11. Diffusion Models

GANs and VAEs were the basis of early approaches to image generation, but in §9.2 and §10.4, we mentioned some of their weaknesses. Diffusion Models appear to be the current state-of-the-art approach to image generation; for example, they were used in DALL-E 2.

Diffusion Models were introduced by Sohl-Dickstein et al [SDWMG15]. They viewed Diffusion Models as a mechanism for converting from *analytically tractable* known distributions (e.g., a Gaussian) and a complicated target distribution (e.g., data). Their idea was inspired

FIGURE 10.4.1. Top: images generated by GAN. Bot: images generated by VAE. The VAE's images tend to be blurry, while the GAN's images look sharper but also less smooth. From [HMP+16, Figure 1].

by statistical physics (non-equilibrium thermodynamics), and involved slowly adding noise to the target distribution to convert it to the known one, and then learning the inverse procedure with a neural network. We have already met this idea when studying VAEs, which transform a Gaussian distribution on latent space to a target data distribution.

Diffusion Models are very different from GANs and VAEs. However, since they are considerably more complicated than the architectures we have met already, we will use the comparison to VAEs as mathematical scaffolding for our discussion. Recall that the decoder half of a VAE learns a conditional probability distribution $p_\theta(x|z)$ as a function of latent space z, such that sampling $z \sim \mathcal{N}(0, \mathrm{Id})$ and then $x \sim p_\theta(x|z)$ induces a marginal distribution

$$p_\theta(x) = \int p_\theta(x|z)p(z)\,dz \approx p_{\text{real}}(x).$$

The idea of Diffusion Models is to learn $p(x|z)$ as a sequence of many small, local steps instead of one direct step. More precisely, a Diffusion Model learns a sequence of transformations

$$z = x_T \to x_{T-1} \to \ldots \to x_t \to x_{t-1} \to \ldots x_0 = x \tag{11.0.1}$$

where each $x_t \to x_{t-1}$ is a relatively small change, by learning conditional density functions $p_\theta(x_{t-1}|x_t)$. The intuition is that this decomposition into a sequence of small steps makes the learning problem more stable and easier to optimize. This can be viewed as improving the *conditioning* of the problem (Remark 4.2.2) by breaking a single, complex mapping into a sequence of simple ones.

In practice, a Diffusion Model would have $T \approx 10^3$. Formally, the VAE resembles the special case $T = 1$ of this description. However, one should *not* actually think of the VAE as a special case of a Diffusion Model. Most notably, in Diffusion the z variable is not latent; it is has the same dimension as x, and is obtained by adding noise to x.

Another intuition of Diffusion Models is that the individual steps $x_t \to x_{t-1}$ can operate at different scales as t changes. For large t, x_t is mostly noise, and the network learns to generate the coarse, global structure of the data. For small t, x_t is very close to a real data sample, and the network learns to refine fine-scale details. This facilitates the development of hierarchical structure, analogous to the way deep neural networks build feature hierarchies. This facilitates the development of hierarchical structure, analogous to the way deep neural networks build feature hierarchies.

11.1. **Design.** To learn the process (11.0.1), we need to learn a bunch of conditional probability density functions

$$p_\theta(x_{t-1}|x_t) \text{ for each } t = 1, 2, \dots, T.$$

Idealistically, we want to train to maximize the (log-)likelihood of the real data $\mathcal{D}$ under the model p_θ, or equivalently minimize cross-entropy loss of the data against

$$p_\theta(x_0) = \int p_\theta(x_0|x_1)p_\theta(x_1|x_2)\dots p(x_{T-1}|x_T)p(x_T)\, dx_1 \dots\, dx_{T-1}dx_T.$$

As in the discussion of VAEs (§10.1), such integrals are intractable, so we will instead use the Evidence Lower Bound as a proxy for $p_\theta(x_0)$. To formulate it, we need to have a collection of probability distributions

$$q_\phi(x_t|x_{t-1}) \text{ for each } t = 1, 2, \dots, T$$

which collectively play the role of the encoder of a VAE.

Remark 11.1.1. Starting with the 2020 work of Ho et al [HJA20], the learnable parameters were removed from q_ϕ.

Collectively, these are analogous to the term $q_\phi(z|x)$ in the discussion of VAEs, except without any learnable parameters. Thus, we can think of (11.0.1) as the "decoder half" to an encoder which implements

$$x = x_0 \to x_1 \to x_2 \to \dots \to x_{t-1} \to x_t \to \dots \to x_T \tag{11.1.1}$$

In particular, as is clear from the form of q_ϕ, (11.1.1) is a Markov process. However, (11.1.1) should *not* be thought of as an embedding/representation. Rather, this Markov process should be thought of as "successively adding noise" so that x_T tends towards a Gaussian distribution $\mathcal{N}(0, \mathrm{Id})$ as $T \to \infty$. To implement this, x_t is obtained from x_{t-1} by adding a small amount of Gaussian noise at each step.

11.1.1. *Functional form of q_ϕ.* As with VAE, we fix the form of $q_\phi(x_t|x_{t-1})$ to be Gaussian. Namely, we take x_t to be of the form

$$x_t = a_t x_{t-1} + b_t \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(0, \mathrm{Id}) \tag{11.1.2}$$

In the original 2015 work of Sohl-Dickstein et al [SDWMG15], the a_t and b_t were learned parameters, but in later work they actually became fixed hyperparameters (thus removing all learning from q). We would like this to converge to a fixed normal distribution, say $\mathcal{N}(0, \mathrm{Id})$ as $t \to \infty$. A choice of algebra that makes for convenient expressions ends up being $a_t = \sqrt{1-\beta_t}$ and $b_t = \sqrt{\beta_t}$; this choice ensures that if x_{t-1} has unit variance, then x_t also has unit variance. The transitions are thus defined by a Markov process

$$x_t = \sqrt{1-\beta_t}\; x_{t-1} + \sqrt{\beta_t}\; \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(0, \mathrm{Id}) \tag{11.1.3}$$

so that

$$q_\phi(x_t|x_{t-1}) \sim \mathcal{N}(\sqrt{1-\beta_t}x_{t-1}, \beta_t\, \mathrm{Id}).$$

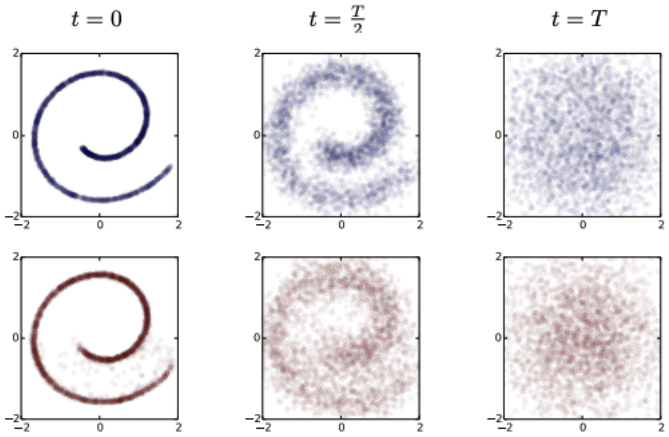


FIGURE 11.1.1. The top row depicts (from left to right) the result of successively adding noise to an initial data sample x_0, using the Markov chain $q_\phi(x_t|x_{t-1})$. The bottom row depicts (from right to left) the model's prediction for each time step, obtained by successively drawing from $p(x_{t-1}|x_t, t)$. From [SDWMG15, Figure 1].

Exercise 11.1.2. To help see these calculations, prove the "Law of Total Variance": for any two random variables X, Y, we have

$$\mathrm{Var}[X] = \mathbb{E}[\mathrm{Var}[X|Y]] + \mathrm{Var}[\mathbb{E}[X|Y]].$$

In the analogy to statistical physics, the β_t correspond to the *diffusion rate*. In the early Diffusion Models of [SDWMG15], these were (the only) learned parameters of q_ϕ. Thus the ϕ stands for the collection of parameters $\beta_1, \beta_2, \ldots, \beta_T$.

The form of this Markov process means that $q_\phi(x_t|x_0)$ is also Gaussian. Explicitly, defining[23] $\bar{\alpha}_t := \prod_{s=1}^{t}(1-\beta_s)$, we have

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon_t^{\text{accum}}, \qquad \epsilon_t^{\text{accum}} \sim \mathcal{N}(0, \mathrm{Id}). \tag{11.1.4}$$

In (11.1.4), the $\epsilon_t^{\text{accum}}$ is not "the same" as the ϵ_t from (11.1.3), but it is identically distributed. It instead represents the accumulated noise relative to the original x_0.

Now, a key observation is that the conditional distribution of x_{t-1} given x_t and x_0 is also Gaussian.

Lemma 11.1.3. *The conditional distribution $q(x_{t-1}|x_t, x_0)$ is Gaussian.*

Proof. According to Bayes' rule we will have

$$q(x_{t-1}|x_t, x_0) = \frac{q(x_t|x_{t-1}, x_0)q(x_{t-1}|x_0)}{q(x_t|x_0)}.$$

For the assertion in question, we can ignore the denominator, since it is a normalization constant that independent of x_{t-1}. The factor $q(x_t|x_{t-1}, x_0) = q(x_t|x_{t-1})$ in the numerator is Gaussian by construction. The factor $q(x_{t-1}|x_0)$ is also Gaussian, as is clear from (11.1.4). Then we conclude using that the product of two Gaussian densities is Gaussian. □

[23]It is common in the literature to define $\alpha_t = 1 - \beta_t$.

Exercise 11.1.4. Show that $q(x_{t-1}|x_t, x_0)$ has mean

$$m_t(x_0, x_t) = \frac{\sqrt{\alpha_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{(1-\alpha_{t-1})\sqrt{1-\beta_t}}{1-\bar{\alpha}_t}x_t \tag{11.1.5}$$

and variance

$$s_t^2 = \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}. \tag{11.1.6}$$

Notice that the variance s_t^2 does not depend on x_0 or x_t.

To summarize, from Lemma 11.1.3 we have

$$q(x_{t-1}|x_t, x_0) \sim \mathcal{N}(m_t(x_0, x_t), s_t^2) \tag{11.1.7}$$

where $m_t(x_0, x_t)$ and s_t^2 are as in Exercise (11.1.4).

11.1.2. *Functional form of p_θ.* As in the discussion of VAEs, we will also take

$$p_\theta(x_{t-1}|x_t) \sim \mathcal{N}(\mu_\theta(x_t, t), \sigma_t^2 I)$$

to be Gaussian, with mean $\mu_\theta(x_t, t)$ to be learned by the neural network. The variance is taken to be the same $\sigma_t^2 = s_t^2$ where s_t^2 is as in (11.1.6). It is not necessary to make this particular choice; we could instead set σ_t^2 to be an independently learned parameter, but this is a reasonable choice which simplifies the objective function (as we will see later).

11.2. **The Evidence Lower Bound.** Let's derive the form of the Evidence Lower Bound $\mathrm{ELBO}(x|p_\theta, q)$ with these choices. For any probability density function $\hat{q}(x_1, \dots, x_T)$ in $x_1, \dots, x_T$, the modeled probability distribution can be written as

$$\begin{aligned} p_\theta(x_0) &= \int p_\theta(x_0, x_1, \dots, x_T)\, dx_1 \dots dx_T \\ &= \int \frac{p_\theta(x_0, x_1, \dots, x_T)}{\hat{q}(x_1, \dots, x_T)} \hat{q}(x_1, \dots, x_T)\, dx_1 \dots dx_T \\ &= \mathbb{E}_{\hat{q}(x_1, \dots, x_T)}\Big[\frac{p_\theta(x_0, x_1, \dots, x_T)}{\hat{q}(x_1, \dots, x_T)}\Big] \end{aligned}$$

so the ELBO is

$$\begin{aligned} \log p_\theta(x) &= \log \mathbb{E}_{\hat{q}(x_1, \dots, x_T)}\Big[\frac{p_\theta(x_0, x_1, \dots, x_T)}{\hat{q}(x_1, \dots, x_T)}\Big] \\ &\geq \mathbb{E}_{\hat{q}(x_1, \dots, x_T)}\Big[\log \frac{p_\theta(x_0, x_1, \dots, x_T)}{\hat{q}(x_1, \dots, x_T)}\Big] =: \mathrm{ELBO}(x|p_\theta, \hat{q}). \end{aligned}$$

11.2.1. This holds for any probability density $\hat{q}(x_1, \dots, x_T)$, so we can in particular take $\hat{q}$ to be q_ϕ from §11.1.1. This gives

$$q_\phi(x_1, \dots, x_T|x_0) = q_\phi(x_T|x_0) \prod_{t=1}^{T} q_\phi(x_{t-1}|x_t, x_0), \qquad q(x_{t-1}|x_t, x_0) \sim \mathcal{N}(m_t(x_0, x_t), s_t^2 \,\mathrm{Id}). \tag{11.2.1}$$

By Lemma 11.1.3, this is a product of Gaussian distributions. We can factorize $p_\theta(x_0, x_1, \dots, x_T)$ in an analogous way:

$$p_\theta(x_0, x_1, \dots, x_T) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t), \qquad p_\theta(x_{t-1}|x_t) \sim \mathcal{N}(\mu_\theta(x_t, t), \sigma_t^2 \,\mathrm{Id}). \tag{11.2.2}$$

11.2.2. In summary, we have

$$\frac{p_\theta(x_0, x_1, \dots, x_T)}{q(x_1, \dots, x_T|x_0)} = \frac{p_\theta(x_T)}{q(x_T|x_0)} \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)}.$$

Therefore the ELBO takes the form

$$\mathrm{ELBO}(x|p_\theta, q) = \mathbb{E}_{q(x_1,\dots,x_T|x_0)}\Big[\log \frac{p(x_T)}{q(x_T|x_0)} + \sum_{t=1}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)}\Big].$$

We will express it analytically in terms of the learnable parameters.

Example 11.2.1. Consider the first term,

$$\mathbb{E}_{q(x_1,\dots,x_T|x_0)}\Big[\log \frac{p(x_T)}{q(x_T|x_0)}\Big] = \int q(x_1, \dots, x_T|x_0) \log \frac{p(x_T)}{q(x_T|x_0)}\, dx_1 \dots\, dx_T.$$

Since $\log \frac{p(x_T)}{q(x_T|x_0)}$ doesn't depend on $x_1, \dots, x_{T-1}$, the marginalization over those variables does nothing, so this can be simplified to

$$\mathbb{E}_{q(x_T|x_0)}\Big[\log \frac{p(x_T)}{q(x_T|x_0)}\Big] = -\,\mathrm{KL}(q(x_T|x_0) \;||\; p(x_T)).$$

Since $p(x_T) \sim \mathcal{N}(0, \mathrm{Id})$ and $q(x_T|x_0) \sim \mathcal{N}(\sqrt{\bar\alpha_T}x_0, (1-\bar\alpha_T)\,\mathrm{Id})$ by (11.1.4), from Example 10.2.6 we have

$$\mathrm{KL}(q(x_T|x_0) \;||\; p(x_T)) = \frac{d}{2}\Big(\frac{\bar\alpha_T + 1}{1-\bar\alpha_T} - \log \frac{1}{1-\bar\alpha_T} - 1\Big)$$

Actually, since this term doesn't depend on learnable parameters θ, it can be ignored for the purpose of optimization.

11.2.3. Treating the other terms similarly, we see that the ELBO decomposes in terms of KL divergences between Gaussians, and can therefore be expressed cleanly in closed form.

$$\begin{aligned}
\mathbb{E}_{q(x_1,\dots,x_T|x_0)}\Big[\log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)}\Big] &= \int q(x_1, \dots, x_T|x_0) \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)}\, dx_1 \dots dx_T \\
&= \int \Big(\int q(x_{t-1}|x_t, x_0) \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_{t-1}|x_t, x_0)}\, dx_{t-1}\Big)\, q(x_t|x_0)\, dx_t \\
&= \mathbb{E}_{q(x_t|x_0)}[-\,\mathrm{KL}(q(x_{t-1}|x_t, x_0) \;||\; p_\theta(x_{t-1}|x_t))]
\end{aligned}$$

Since $q(x_{t-1}|x_t, x_0)$ and $p_\theta(x_{t-1}|x_t)$ are chosen to be Gaussian with the same variance s_t^2, Example 10.2.6 gives

$$\mathrm{KL}(q(x_{t-1}|x_t, x_0) \;||\; p_\theta(x_{t-1}|x_t)) = \frac{(m_t(x_t, x_0) - \mu_\theta(x_t, t))^2}{2s_t^2} + \text{constant}. \tag{11.2.3}$$

where $m_t(x_t, x_0)$ is as in (11.1.5) and $\mu_\theta(x_t, t)$ is learned mean of $p_\theta(x_{t-1}|x_t)$. The constant does not affect the formation of gradients, so we can ignore it.

Example 11.2.2. For $t = 1$, $q(x_{t-1}|x_t, x_0)$ is a bump function at x_0, so we simply have

$$\mathrm{KL}(q(x_0|x_1, x_0) \;||\; p_\theta(x_0, x_1)) = -\log p_\theta(x_0|x_1).$$

The corresponding contribution to $\mathrm{ELBO}(x|q, p_\theta)$ is

$$-\mathbb{E}_{q(x_1|x_0)}[\log p_\theta(x_0|x_1)],$$

which is analogous to the "Reconstruction Term" for a VAE (§10.2.2).

11.2.4. *The loss function.* Putting things together (and sorting out the signs), we find that maximizing the ELBO is equivalent to the following optimization problem:

$$\arg\min_\theta \Big(- \mathbb{E}_{q(x_1|x_0)}[\log p_\theta(x_0|x_1)] + \sum_{t=2}^T \frac{(m_t(x_t,x_0) - \mu_\theta(x_t,t))^2}{2s_t^2}\Big). \tag{11.2.4}$$

The appropriate loss function in terms of data x_0 is therefore

$$\mathcal{L}_{x_0}(\theta) = \mathbb{E}_{q(x_1,\dots,x_T|x_0)}\Big[- \log p_\theta(x_0|x_1) + \sum_{t=2}^T \frac{(m_t(x_t,x_0) - \mu_\theta(x_t,t))^2}{2s_t^2}\Big]. \tag{11.2.5}$$

11.3. **Comparison to VAEs.** If we took $T = 1$, then our initial discussion of Diffusion Models would be formally similar to that of a VAE, except that the form of $q_\phi(x_1|x_0) \leftrightarrow q_\phi(z|x)$ is different. However, the intuitive meaning of x_1 and z is still very different: z is meant to be a latent representation, whereas x_1 is meant to be a noise-obscured of x_0.

Note also that despite the more complicated mathematical derivation, the training objective of a Diffusion Model turned out to be simpler: a straightforward mean squared error regression.

left for presentations instead of lecture

11.4. **Improving diffusion models.** So far, we have described Diffusion Models in their original (2015) conception [SDWMG15]. It was not until the 2020 work of Ho et al [HJA20] that the efficacy of Diffusion Models for image generation was convincingly demonstrated. Compare the images in Figure 11.4.1, which were generated by the Diffusion Model [HJA20], with the images generated by GAN and VAE in Figure 10.4.1.

This achievement involved several modifications to the "vanilla" setup described so far, most of which were *simplifications*. These simplifications are mathematically anticlimactic, in the sense that they throw away hard-earned mathematical derivations in favor of simpler variants that would have been obvious to guess from the beginning, but they work better in practice. Such is life...

11.4.1. *Removing the encoder.* In the original formula of Diffusion Models, the functions $q_\phi(x_t|x_{t-1})$ depended upon learnable parameters β_t. Ho et al fix β_t to be hyperparameters, thus fixing $q(x_t|x_{t-1}) = q_\phi(x_t|x_{t-1})$ to be independent of training. Specifically, they took $T = 1000$, and scheduled β_t to increase linearly from $\beta_1 = 10^{-4}$ to $\beta_{1000} = 0.02$. These were deliberately chosen to be small compared to the values of the input data, which were RGB intensities scaled to the range $[-1, 1]$.

11.4.2. *Simplifying the loss function.* The objective (11.2.5) was formulated for a model that predicts $p_\theta(x_{t-1}|x_t)$, by predicting its mean and variance.

Recall from (11.1.4) that x_t could be written in the form

$$x_t = \sqrt{\bar\alpha_t} x_0 + \sqrt{1-\bar\alpha_t}\varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0, \mathrm{Id}). \tag{11.4.1}$$

Ho et al change the neural network to predict the noise vector ε_t instead than $p_\theta(x_{t-1}|x_t)$. This is philosophically reminiscent of residual learning, where we predict the "residual" instead of the desired function. Therefore, we will rewrite the objective (11.2.5) in these terms. First, we can solve for $m_t(x_0, x_t)$ in terms of ε_t,

$$m_t(x_0, x_t) = \frac{1}{\sqrt{1-\beta_t}}\Big(x_t - \frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\varepsilon_t\Big).$$

FIGURE 11.4.1. Images generated by diffusion, from [HJA20, Figure 1]. We see marked improvements compared to the images generated by GAN and VAE in Figure 10.4.1.

Therefore, recalling that $p_\theta(x_{t-1}|x_t) \sim \mathcal{N}(\mu_\theta(x_t,t), \sigma_t^2 \operatorname{Id})$, we will define $\varepsilon_\theta(x_t,t)$ in terms of $\mu_\theta(x_t,t)$ by the formula

$$\mu_\theta(x_t,t) = \frac{1}{\sqrt{1-\beta_t}}\Big(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t,t)\Big).$$

Instead of outputting $\mu_\theta(x_t,t)$, the neural network will output $\varepsilon_\theta(x_t,t)$. We can view $q(x_t|x_0)$ as defining a distribution on ε_t. Then for $t \geq 2$, we write the KL divergence contribution (11.2.3) as

$$\mathcal{L}^t_{x_0}(\theta) = w_t \mathbb{E}_{q(x_t|x_0)}[||\varepsilon_t - \varepsilon_\theta(x_t,t)||^2]$$

where the weighting factor is $w_t = \frac{\beta_t}{2(1-\bar{\alpha}_{t-1})(1-\beta_t)}$. A calculation shows that this is in fact also true for $t = 1$, as is also clear by "pure thought" since that can be seen as a continuation of the sequence where $x_{-1} = x_0$ and $\beta_0 = 0$. Notice that the inner term has a simple interpretation as the MSE between the sampled noise ε_t and the predicted noise $\varepsilon_\theta(x_t,t)$.

In summary, we have rewritten (11.2.5) as

$$\mathcal{L}_{x_0}(\theta) = \sum_{t=1}^{T} \mathcal{L}^t_{x_0}(\theta) = \sum_{t=1}^{T} w_t \mathbb{E}_{q(x_t|x_0)}[||\varepsilon_t - \varepsilon_\theta(x_t,t)||^2. \tag{11.4.2}$$

In theory, this is the function which we should try to maximize. But practice is another matter. After reaching (11.4.2), Ho et al [HJA20] suggest throwing away the weighting

factors w_t, which simplifies the math and actually works better in their experiments. A possible intuition here is that w_t downweights the contributions from large t where the noise level is high, whereas in practice the small t regime tends to be less important than the mathematics suggests. Anyway, this gives a simplified loss function,

$$\widetilde{\mathcal{L}}_{x_0}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{q(x_t|x_0)}[||\varepsilon_t - \varepsilon_\theta(x_t, t)||^2. \tag{11.4.3}$$

Furthermore, the interpretation of (11.4.3) is clear: it is just the usual mean squared loss between the "ground-truth" noise ε_t and the predicted noise $\varepsilon_\theta(x_t, t)$, summed over all t. Intuitively, the Diffusion Model learns to guess what noise was added at each step of the process $x_0 \to x_1 \to x_2 \to \ldots \to x_T$.

In fact, since T is large in practice (e.g., $\approx 10^3$), the summation over t is pretty costly. Therefore, we replace it by an estimator which is faster to evaluate: we will simply choose a random t and evaluate the term $\mathbb{E}_{q(x_t|x_0)}[||\varepsilon_t - \varepsilon_\theta(x_t, t)||^2]$ by Monte Carlo sampling. To do this, we sample $\varepsilon_t \sim \mathcal{N}(0, \mathrm{Id})$, then write x_t in terms of x_0 and ε_t as

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon_t,$$

and feed this into the NN to compute $\varepsilon_\theta(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon_t, t)$.

In conclusion, we have arrived at a "simple loss function",

$$\mathcal{L}_{x_0}^{\text{simple}}(\theta) = \mathbb{E}_{\substack{t \sim \mathcal{U}[1,T] \\ \varepsilon_t \sim \mathcal{N}(0,\mathrm{Id})}} [||\varepsilon_t - \varepsilon_\theta(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon_t, t)||^2] \tag{11.4.4}$$

where $\mathcal{U}[1, T]$ is the uniform distribution on $1, 2, \ldots, T$. Note that the sampling of $\varepsilon_t \sim \mathcal{N}(0, \mathrm{Id})$ effects the sampling of $x_t \sim q_t(x_t|x_0)$, and the sampling of $t \sim \mathcal{U}[1, T]$ is an unbiased estimator for the average over T (Example 3.5.2). Hence (11.4.4) can be viewed as an unbiased estimator for (11.4.3). This comes at a cost, however: (11.4.4) is noisier than (11.4.3), and therefore requires more samples to converge.

left for presentation instead of lecture

11.5. **Latent Diffusion.** Although Diffusion Models can produce better results than GANs and VAEs, they also come at a significant price: both training and evaluation are much more costly, (e.g., hundreds of GPU days). Indeed, Diffusion operates in an ambient space of the same dimension as the data, which for images is very high. This was in fact the reason we emphasized the importance of representation learning, which could be thought of as compression to latent space.

There is an intuition that representation learning has two qualitatively different phases. The first stage is "perceptual compression", which removes essentially imperceptible details that have no semantic consequence; and the second stage, which occurs when the modeled reaches meaningful tiers of understanding, is "semantic compression". The intuition is that diffusing in pixel space wastes the bulk of computation on semantically insignificant bits, which are compressed away in the first stage; therefore, we should really be performing diffusion in the intermediate latent space reached after "perceptual compression".

In 2021, Rombach et al [RBL+21] introduced *Latent Diffusion*, implementing diffusion in latent space to achieve high-quality image generation at lower cost (some samples can be seen in Figure 11.5.1). This project later developed into Stable Diffusion, which is known to be one ingredient behind OpenAI's DALL-E 2; it is hard to say more due to scarcity of publicly available information about the details of current proprietary image generation technology.

FIGURE 11.5.1. Samples of images generated by the Latent Diffusion Models trained on a dataset of celebrity images, from [RBL+21, Figure 4].

11.5.1. *Latent compression.* Rombach et al use an autoencoder (Remark 10.2.4) to compress to latent space. The autoencoder downsamples the $H \times W \times C$ image to $(H/f) \times (W/f) \times C$. Choices of $f \in \{1, 2, 4, 8, 16, 32\}$ were investigated. The values $f = 1$ and $f = 2$ demonstrated slow training progress, while $f = 32$ already exhibits stagnation (failure to improve after a certain point). The best performance occurs with $f = 4$ or $f = 8$. Notably, this is far less downsampling than occurs over the course of a typical CNN.

11.5.2. *Cross-Attention conditioning.* Rombach et al also introduce "Cross-Attention blocks" into the latent Diffusion model architecture, in anticipation of application to *conditional* generation in the sense of §10.5. For example, suppose we want to generate an image x based on a caption c. Then the model should learn $\varepsilon_\theta(x_t, t|E(c))$ where $E(c)$ is a domain-specific embedding, e.g., a text embedding for caption generation. Attention is used to interface the diffusion process with $E(c)$.

More precisely, for each x_t generated by the model during denoising, we take a flattened intermediate representation $\varphi(x_t)$ generated by the neural network and use it to compute a Query vector $Q(x_t) = \varphi(x_t)W_Q$ where W_Q is a learned parameter. Then we compute Attention against Key and Value vectors coming from $E(c)$:

$$K = E(c)W_K, \quad V = E(c)W_V,$$

where W_K and W_V are also learned. This is called Cross-Attention, as it is a form of Attention that extends across different modalities. These Cross-Attention layers are inserted into the U-net (a CNN with downsampling convolutions followed by upsampling convolutions)

that calculates $\varepsilon_\theta(x_t, t|E(c))$. The architecture of Latent Diffusion is displayed in Figure 11.5.2.

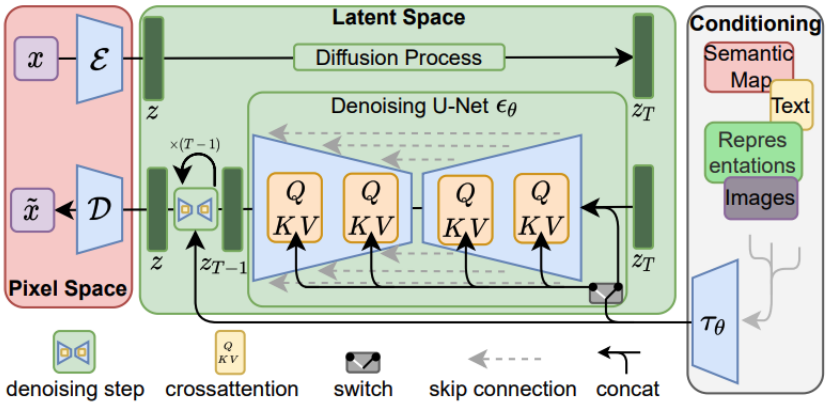


FIGURE 11.5.2. Latent Diffusion Model architecture from [RBL+21, Figure 3]. Reading from top left to top right, then bottom right to bottom left: the encoder $\mathcal{E}$ embeds the data x in latent space. Diffusion is implemented in latent space, with Cross-Attention blocks interleaved in the denoising process for conditional generation. Finally, the decoder transforms the generated latent vector back to data space $\mathcal{D}$.

Remark 11.5.1. Why are the image pixels are used to compute the Query, while the text caption is used to compute the Key and Value? The way to think about things is that the image is the output, and the output needs to know what to attend to. Therefore, *the output issues the query.*

Part 4. Reinforcement learning

12. Value function methods

13. Policy optimization

References

[ABGM13] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma, *Provable Bounds for Learning Some Deep Representations*, International Conference on Machine Learning, 2013.

[AZ20] Samira Abnar and Willem Zuidema, *Quantifying Attention Flow in Transformers*, Annual Meeting of the Association for Computational Linguistics, 2020.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, CoRR **abs/1409.0473** (2014).

[BKH16] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton, *Layer Normalization*, ArXiv **abs/1607.06450** (2016).

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Ma teusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei, *Language Models are Few-Shot Learners*, ArXiv **abs/2005.14165** (2020).

[CB90] George Casella and Roger L. Berger, *Statistical Inference*, The Wadsworth & Brooks/Cole Statistics/Probability Series, Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1990.

[CvMG+14] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, Conference on Empirical Methods in Natural Language Processing, 2014.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby, *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, ArXiv **abs/2010.11929** (2020).

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, North American Chapter of the Association for Computational Linguistics, 2019.

[DRD+24] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness, *Language Modeling Is Compression*, 2024.

[DYY+19] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov, *Transformer-XL: Attentive Language Models beyond a Fixed-Length Context*, ArXiv **abs/1901.02860** (2019).

[FS97] Yoav Freund and Robert E. Schapire, *A Decision-theoretic Generalization of On-line Learning and an Application to Boosting*, European Conference on Computational Learning Theory, 1997.

[Goo17] Google Research Blog, *Transformer: A Novel Neural Network Architecture for Language Understanding*, `https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/`, aug 2017, Accessed: 2025-08-30.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio, *Generative Adversarial Nets*, Neural Information Processing Systems, 2014.

[GSS14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy, *Explaining and Harnessing Adversarial Examples*, CoRR **abs/1412.6572** (2014).

[HBM+22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and L. Sifre, *Training Compute-Optimal Large Language Models*, ArXiv **abs/2203.15556** (2022).

[HHS17] Elad Hoffer, Itay Hubara, and Daniel Soudry, *Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks*, ArXiv **abs/1705.08741** (2017).

[HJA20] Jonathan Ho, Ajay Jain, and P. Abbeel, *Denoising diffusion probabilistic models*, ArXiv **abs/2006.11239** (2020).

[HMP+16] Irina Higgins, Loïc Matthey, Arka Pal, Christopher P. Burgess, Xavier Glorot, Matthew M. Botvinick, Shakir Mohamed, and Alexander Lerchner, *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*, International Conference on Learning Representations, 2016.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber, *Long Short-Term Memory*, Neural Computation **9** (1997), 1735–1780.

[HS06] Geoffrey E. Hinton and Ruslan R. Salakhutdinov, *Reducing the Dimensionality of Data with Neural Networks*, Science **313** (2006), no. 5786, 504–507.

[HSK+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, *Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors*, ArXiv **abs/1207.0580** (2012).

[HZRS15a] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun, *Deep Residual Learning for Image Recognition*, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015), 770–778.

[HZRS15b] ———, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet*

Classification, 2015 IEEE International Conference on Computer Vision (ICCV) (2015), 1026–1034.

[iF89] Ken ichi Funahashi, *On the Approximate Realization of Continuous Mappings by Neural Networks*, Neural Networks **2** (1989), 183–192.

[IS15] Sergey Ioffe and Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, ArXiv **abs/1502.03167** (2015).

[Kap] Jared Kaplan, *Notes on Contemporary Machine Learning for Physicists*.

[KMH+20] Jared Kaplan, Sam McCandlish, T. J. Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu, and Dario Amodei, *Scaling Laws for Neural Language Models*, ArXiv **abs/2001.08361** (2020).

[KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang, *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*, ArXiv **abs/1609.04836** (2016).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Communications of the ACM **60** (2012), 84–90.

[KW13] Diederik P. Kingma and Max Welling, *Auto-Encoding Variational Bayes*, CoRR **abs/1312.6114** (2013).

[LBOM12] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller, *Efficient BackProp*, Neural Networks, 2012.

[PS24] Tim Pearce and Jinyeop Song, *Reconciling Kaplan and Chinchilla Scaling Laws*, Trans. Mach. Learn. Res. **2024** (2024).

[PWJ+24] Tomer Porian, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon, *Resolving Discrepancies in Compute-Optimal Scaling of Language Models*, ArXiv **abs/2406.19146** (2024).

[RBL+21] Robin Rombach, A. Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, *High-resolution image synthesis with latent diffusion models*, 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2021), 10674–10685.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei, *ImageNet Large Scale Visual Recognition Challenge*, International Journal of Computer Vision (IJCV) **115** (2015), no. 3, 211–252.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, CoRR **abs/1511.06434** (2015).

[RN18] Alec Radford and Karthik Narasimhan, *Improving Language Understanding by Generative Pre-Training*, 2018.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, *Language Models are Unsupervised Multitask Learners*, 2019.

[SDWMG15] Jascha Narain Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli, *Deep unsupervised learning using nonequilibrium thermodynamics*, ArXiv **abs/1503.03585** (2015).

[Sha48] C. E. Shannon, *A Mathematical Theory of Communication*, Bell System Tech. J. **27** (1948), 379–423, 623–656.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, D. Erhan, Vincent Vanhoucke, and Andrew Rabinovich, *Going Deeper with Convolutions*, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2014), 1–9.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le, *Sequence to Sequence Learning with Neural Networks*, ArXiv **abs/1409.3215** (2014).

[SZ14] Karen Simonyan and Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, CoRR **abs/1409.1556** (2014).

[VSP+17] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin, *Attention is All you Need*, Neural Information Processing Systems, 2017.