Machine Learning Explainers

Last updated: February 12, 2026

Abstract

Summaries of machine/deep learning-related topics. My main motive in writing these summaries is as a reminder for my future self.

Contents

1 What is Machine Learning?

I think of machine learning as the broad discipline of studying methods of fitting models to data; like statistics, as a discipline, with less rigour but more messy creativity. For a description of machine learning which helps to clarify why it's difficult to define as a term, consider the following excerpt from Herbert Jaeger's lecture notes for his machine learning course at the University of Groningen during the academic year 2023/24:

> ML as a field, which perceives itself as a field under this name, is relatively young, say, about 40 years (related research was called "pattern recognition" earlier). It is interdisciplinary and has historical and methodological connections to neuroscience, cognitive science, linguistics, mathematical statistics, AI, signal processing and control; it uses mathematical methods from statistics (of course), information theory, signal processing and control, dynamical systems theory, mathematical logic and numerical mathematics; and it has a very wide span of applications. This diversity in traditions, methods and applications makes it difficult to study "Machine Learning". Any given textbook, even if it is very thick, will reflect the author's individual view and knowledge of the field and will be partially blind to other perspectives. This is quite different from other areas in computer science, say for example formal languages/theory of computation/computational complexity where a widely shared repertoire of standard themes and methods cleanly define the field.

My interests in machine learning lie in its rapid development since the deep learning boom from 2012 (AlexNet) and the mysteries of why its methods are so effective. At their core, many methods in machine learning are statistically-principled, relying on maximum likelihood estimation. The fact that something as simple as maximum likelihood estimation can be used to fit very complex distributions to at least a small extent isn't too surprising. What fascinates me is the notable extent to which it does so in practice. It performs so well that we are able to produce models which take natural language as input and output entirely realistic corresponding images/videos. What right does maximum likelihood estimation have to facilitate such effective fitting of complex distributions in practice? Why are deep learning architectures able to encode such rich function classes? We only have partial answers to the many natural questions like these and so our understanding is far from complete. What an interesting time to be alive. :)

2 Supervised Learning

Supervised learning methods fit parameterised functions $f_\theta : \Omega_{\mathbf{X}} \to \Omega_{\mathbf{Y}}$ from model variables $\mathbf{X} = (X_1, \ldots, X_m)$ to output variables $\mathbf{Y} = (Y_1, \ldots, Y_r)$ (often $r = 1$ in which case we write $\mathbf{Y} = Y$) by tweaking θ in line with concrete examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ of how the function may behave in practice. The two most prominent examples of supervised learning tasks are regression (continuous output variables) and classification (discrete output variables).

2.1 Linear Regression

Perhaps the simplest example of supervised learning is linear regression. Suppose we are given a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{2n}$ where

$$\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,m}) \in \Omega_{X_1} \times \cdots \times \Omega_{X_m} =: \Omega_{\mathbf{X}} \subseteq \mathbb{R}^m$$

are the feature values of the i^{th} sample and $y_i \in \Omega_Y \subseteq \mathbb{R}$ is its corresponding output. The reason I chose $|D| = 2n$ is that it yields a convenient partition of the data into D_{train} and D_{test} each of n samples. A linear regression models fits a model of the form

$$\begin{aligned} f_\theta : \Omega_{\mathbf{X}} &\to \Omega_Y \\ \mathbf{x} &\mapsto \theta^\top \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} = \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m \end{aligned}$$

where $\theta = (\theta_0, \theta_1 \ldots, \theta_m) \in \mathbb{R}^{m+1}$ are the model parameters, i.e. the values we can tweak to our heart's content until testing error is sufficiently low. Some people refer to the paramter θ_0, which dictates the elevation of the hyperplane corresponding to $f_\theta(\mathbf{x}) = 0$, as the bias of the model which I find very confusing as there are a bunch of other intended meanings of the term 'bias' in statistics and machine learning. I prefer to refer to it as the elevation. Anyway, once such a linear function has been fit, given feature values $\mathbf{x} \in \Omega_{\mathbf{X}}$ our model predict the corresponding output as $y = f_\theta(\mathbf{x})$.

What does the 'linear' in linear regression actually refer to?

Casella Berger, as well as other pieces of literature, define linear regression models as being linear in their parameters. By such a definition, linear regression, as a term, encompasses polynomial regression models and other regression models with non-linear basis functions. This previously confused me because, in my experience, 'linear regression' is most often intended to mean models that fit a hyperplane to $\Omega_{\mathbf{X}} \times \Omega_Y$.

The first paragraph of the Wikipedia page[a] for polynomial regression reiterates this potential confusion.

[a] `https://en.wikipedia.org/wiki/Polynomial_regression`

An intuitive approach to finding the 'optimal' parameters of a linear regression model, which we denote by θ^*, is to split D into training and testing datasets D_{train} and D_{test} (each consisting of n sample in our case) and minimising some pre-determined loss function of said parameters over D_{train}. Essentially, minimising something like

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} d(f_\theta(\mathbf{x}_i), y_i)$$

where $f_\theta(\mathbf{x}_i)$ is the model's prediction for feature values $\mathbf{x}_i$ and $d : \Omega_Y \times \Omega_Y \to \mathbb{R}_{\geq 0}$ is some goodness-of-prediction metric or loss function. Said loss function gives one an idea of how well θ fits the true underlying relationship which we wish to model. A common choice for the loss function is the mean square error

$$\text{MSE}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(\mathbf{x}_i))^2$$

which pertains to taking $d(f_\theta(\mathbf{x}_i), y_i) = (y_i - f_\theta(\mathbf{x}_i))^2$. Note that the factor of $1/n$ is often left out when discussing MSE as minimising the expression in θ is invariant to the inclusion of the factor. So in our case, we seek to compute

$$\theta^* = \underset{\theta \in \mathbb{R}^{m+1}}{\arg\min} \left[\sum_{i=1}^{n}(y_i - f_\theta(\mathbf{x}_i))^2\right].$$

We know that in the context of linear regression, the optimal paramters θ^* are typically taken to be those which minimise the mean square error over D_{train} but how do we actually compute θ^*? This could be done through

numerical methods, which is often the case in machine learning, e.g. using gradient descent in computing the optimal parameters of a logistic regression model, but linear regression has a closed form solution. This is pretty cool since it's not so common for such closed form solutions to exist in machine learning-related contexts (though stats people might not like linear regression being referred to as machine learning-related). The informal method which I use to remember the closed form solution for the optimal paramters of a linear regression model is

$$X\theta^* = y \implies X^\top X\theta^* = X^\top y \implies \theta^* = (X^\top X)^{-1}X^\top y.$$

In practice, if this matrix $X^\top X$ is singular then just add some small values to its diagonal. That is, instead compute

$$\theta^* = (X^\top X + \delta I)^{-1}X^\top y$$

for some small $\delta \in \mathbb{R}$. That said, what's given above is not rigorous as it assumes the existance of θ^* and does not make use of MSE, so let's derive it. Note that

$$\begin{aligned}
\text{MSE}(\theta) &= \sum_{i=1}^{n}(y_i - f_\theta(\mathbf{x}_i))^2 \\
&= \begin{bmatrix} y_1 - f_\theta(\mathbf{x}_1) & \cdots & y_n - f_\theta(\mathbf{x}_n) \end{bmatrix} \begin{bmatrix} y_1 - f_\theta(\mathbf{x}_1) \\ \vdots \\ y_n - f_\theta(\mathbf{x}_n) \end{bmatrix} \\
&= (y - X\theta)^\top (y - X\theta) \\
&= ||y - X\theta||^2
\end{aligned}$$

where

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n, X = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,m} \end{bmatrix} \in \mathbb{R}^{n\times(m+1)}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_m \end{bmatrix} \in \mathbb{R}^{m+1}$$

and $x_{i,j}$ denotes the jth element of the i^{th} sample. To find the minimiser(s) of $\text{MSE}(\theta)$, i.e. the optimal parameters θ^*, we compute its gradient with respect to θ and find its root(s). This of course only works when our function is both differentiable and convex, which is the case here. To see convexity, simply compute the Hessian of $\text{MSE}(\theta)$ and see that it is semi-positive

definite. On that note, we have

$$\begin{aligned}\nabla \text{MSE}(\theta) &= \nabla ||y - X\theta||^2 \\ &= \nabla (y - X\theta)^\top (y - X\theta) \\ &= \nabla \left[\theta^\top X^\top X\theta - \theta^\top X^\top y - y^\top X\theta + y^\top y\right] \\ &= \nabla \left[\theta^\top X^\top X\theta - 2y^\top X\theta + y^\top y\right] \\ &= 2X^\top X\theta - 2X^\top y\end{aligned}$$

and so the optimiser θ^* is given by

$$\theta^* = (X^\top X)^{-1} X^\top y.$$

Funny misuse of linear regression: Momentous sprint at the 2156 Olympics?

Read this Quora post[a] based on a paper[b] published in Nature. The top comment of the post is worth reading too. Related xkcd comic[c].

[a] https://qr.ae/pslbEN
[b] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3173856/
[c] https://xkcd.com/1007/

2.1.1 Statistical Motivation

The idea of minimising the mean square error of the model over D_{train} has a rigorous statistical motivation, corresponding to maximum likelihood estimation. Assume that the residuals corresponding of the model's output over D_{train} are independent and identically normally distributed with mean 0. That is, assume

$$E_i = Y_i - f_\theta(\mathbf{X}_i) \sim \mathcal{N}(0, \sigma^2)$$

for $i = 1, \ldots, n$ are i.i.d. This assumption is reasonable in practice due to the central limit theorem (CLT). We have $Y_i | \mathbf{X}_i \sim \mathcal{N}(f_\theta(\mathbf{x}_i), \sigma^2)$ and so

$$p_\theta(y_i | \mathbf{x}_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - f_\theta(\mathbf{x}_i))^2}{2\sigma^2}\right).$$

The maximum likelihood estimate θ_{MLE} of the parameters of our linear regression model is that which maximise the relevant log-likelihood. That

is,

$$\begin{aligned}
\theta_{\text{MLE}} &= \arg\max_{\theta \in \mathbb{R}^{m+1}} \left[\log \left(\prod_{i=1}^{n} p_\theta(y_i | \mathbf{x}_i) \right) \right] \\
&= \arg\max_{\theta \in \mathbb{R}^{m+1}} \left[\sum_{i=1}^{n} \log(p_\theta(y_i | \mathbf{x}_i)) \right] \\
&= \arg\max_{\theta \in \mathbb{R}^{m+1}} \left[n \log \left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2 \right] \\
&= \arg\min_{\theta \in \mathbb{R}^{m+1}} \left[\sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2 \right].
\end{aligned}$$

As such, the maximum likelihood estimate of the parameters of our linear regression model are precisely those which minimise the mean square error of the model over D_{train}.

Are residuals really normally distributed?

[a]Central limit theorem to the rescue: "Regression analysis, and in particular ordinary least squares, specifies that a dependent variable depends according to some function upon one or more independent variables, with an additive error term. Various types of statistical inference on the regression assume that the error term is normally distributed. This assumption can be justified by assuming that the error term is actually the sum of many independent error terms; even if the individual error terms are not normally distributed, by the central limit theorem their sum can be well approximated by a normal distribution."

[a]`https://en.wikipedia.org/wiki/Central_limit_theorem#Regression`

2.1.2 Goodness of fit: R^2

If we'd like a way to measure the goodness-of-fit of a linear regresison model beyond test MSE, a natural avenue is to assess what portion of the sample variance is explained by the model. As usual, we do this over some sample $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n \subset \Omega_{\mathbf{X}} \times \Omega_Y$. That is, computing

$$\frac{\text{Var}(f_\theta(\mathbf{X}))}{\text{Var}(Y)} \approx \frac{\frac{1}{n}\sum_{i=1}^{n} \left(f_\theta(\mathbf{x}_i) - \bar{y}\right)^2}{\frac{1}{n}\sum_{i=1}^{n} \left(y_i - \bar{y}\right)^2} =: R^2$$

where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. The reason that this quantity is denoted by R^2 is that it can be shown that it is equal to the square of the Pearson correlation between Y and $f_\theta(\mathbf{X})$. For one predictor, i.e. $m = 1$, said Pearson correlation coefficient is given by

$$R = \frac{\sum_{i=1}^{n} \left(f_\theta(\mathbf{x}_i) - \bar{f}_\theta\right)(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} \left(f_\theta(\mathbf{x}_i) - \bar{f}_\theta\right)^2}\sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}}.$$

The derivation of this statement is boring, so I'll leave it out. It's worth noting that R^2 is typically expressed as

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} = 1 - \frac{s_e^2}{s_Y^2}$$

where s_e^2 and s_Y^2 denote the sample variances of the residuals and Y respectively. I find the representation of R^2 as the ratio of explained variance to response variance to be more intuitive and feel less like it just comes out of nowhere.

This notion of the portion of explained variance can be extended to logistic regression but the details are a bit much for this document.

How can we get an idea of the extent to which our features and output are linearly related without plotting?

To be honest, I thought a streamline test for this existed. As per this Stack Exchange answer[a], a good method for this is to simply fit a linear and a non-linear model (e.g. a cubic spline smoother model) and see which explains a larger amount of the variance of the output.

[a] `https://stats.stackexchange.com/a/239142`

2.1.3 Why call it 'regression'?

Nowadays, regression models refer to those which predict a value belonging to some continuous space, but where does the term come from? In 1886, Francis Galton authored "Regression Towards Mediocrity in Hereditary Stature" which is where the 'regression towards the mean' phrase comes from. Loosely speaking, Galton noticed that tall fathers tend to have sons which are taller than average but shorter than them, short fathers tend to have sons which are shorter than average but taller than them and average height fathers tend to have average height sons. Taken from a Stack Exchange comment:

Galton derived a linear approximation to estimate a son's height from the father's height in that paper. His equation was fitted so an average height father would have an average height son, but a taller than average father would have a son that is taller than average by 2/3 the amount his father is. Same with shorter than average. This could be argued to be a simple linear regression.

Put mathematically, suppose random variables X and Y are related via $Y = \alpha + \beta X + \epsilon$ where $\alpha, \beta \in \mathbb{R}$ are regression coefficients and ϵ is noise with mean 0. Then $\mathbb{E}[Y|X] = \alpha + \beta X$, so if X and Y are centred then $\mathbb{E}[Y|X] = \rho X$ where ρ is the correlation coefficient between X and Y. Since $|\rho| \leq 1$ we know that the expected value of Y is closer to the mean than X unless $\rho = 1$. So extreme values of X tend to correspond to values of Y that are closer to the mean, i.e. $Y|X$ regresses towards the mean.

2.2 Logistic Regression

Logistic regression is to binary classification what linear regression is to regression. That is, both are the first method learned when introduced to regression and binary classification. The name might seem strange at first as regression models predict continuously distributed things and binary classification models predict either 0 or 1. The reason regression appears in its name is that it classifies samples by transforming them to $(0, 1)$ and imposing a threshold-based decision rule to yield 0 or 1.

Before detailing precisely how a logistic regression model classifies samples, how might one classify a sample at all? A natural idea is to fit a hyperplane to feature space which separates samples of the two classes. A hyperplane is an $(n-1)$-dimensional plane-like object embedded in n-dimensional space. For example, a hyperplane in $\mathbb{R}^2$ is a line and in $\mathbb{R}^3$ it is a plane. As an object, a hyperplane in $(m+1)-$dimensional space is the set of points $(x_1, \ldots, x_m) \in \mathbb{R}^m$ which satisfy

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m = 0$$

where $\theta = (\theta_0, \ldots, \theta_m) \in \mathbb{R}^{m+1}$ are parameters which characterise the hyperplane. For brevity, we denote the function whose set of roots is the hyperplane by

$$\begin{aligned} f_\theta : \{1\} \times \mathbb{R}^m &\to \mathbb{R} \\ \mathbf{x} &\mapsto \theta_0 + \theta_1 x_1 + \cdots + \theta_m x_m =: \theta^\top \mathbf{x}. \end{aligned}$$

The purpose of the 1 in the first index of $\mathbf{x}$ is similar to its purpose in linear regression: it accounts for the elevation term θ_0 and makes notation a lot cleaner. After learning the parameters of a hyperplane, we may classify a sample $\mathbf{x}$ according to which side of the hyperplane $f_\theta(\mathbf{x}) = 0$ it lies. For example, samples 'below' the hyperplane, i.e. $f_\theta(\mathbf{x}) \leq 0$, could be classified as 0 and samples 'above', i.e. $f_\theta(\mathbf{x}) > 0$, could be classified as 1. That is, classify samples according to $C_\theta(\mathbf{x}) = \mathbb{1}(f_\theta(\mathbf{x}) > 0)$.

To obtain a notion of the probability that the class label of a sample is 1, consider how far it deviates from the plane. To make this idea concrete, apply a logistic (sigmoid) transformation to the output $f_\theta(\mathbf{x})$ as in

$$h_\theta(\mathbf{x}) = \sigma(f_\theta(\mathbf{x})) = \frac{1}{1 + \exp(-f_\theta(\mathbf{x}))} \in (0, 1).$$

From here, the class label of a sample $\mathbf{x}$ is 1, according to the model, if $h_\theta(\mathbf{x}) > \delta$ for some threshold $\delta \in (0, 1)$. Note that $\delta = 0.5$ corresponds precisely to the 'above/below the hyperplane' approach above. The advantage of this broad decision rule is that we can select δ in a way that improves precision at the expense of recall and vice versa. For example, for higher precision and lower recall, $\delta > 0.5$ and classify samples according to $C^\delta_\theta(\mathbf{x}) = \mathbb{1}(h_\theta(\mathbf{x}) > \delta)$.

See that for the model to match the underlying probability that the class label of a given sample is 1, we require that

$$\begin{aligned}
& h_\theta(\mathbf{x}) = p(Y = 1|\mathbf{X} = \mathbf{x}) \\
\iff & \frac{1}{1 + \exp(-f_\theta(\mathbf{x}))} = p(Y = 1|\mathbf{X} = \mathbf{x}) \\
\iff & f_\theta(\mathbf{x}) = \log\left(\frac{p(Y = 1|\mathbf{X} = \mathbf{x})}{1 - p(Y = 1|\mathbf{X} = \mathbf{x})}\right) \\
\iff & f_\theta(\mathbf{x}) = \log\left(\frac{p(Y = 1|\mathbf{X} = \mathbf{x})}{p(Y = 0|\mathbf{X} = \mathbf{x})}\right)
\end{aligned}$$

and so logistic regression models can be seen as fitting a linear regression model to the log-odds of each sample being of class 1.

While their motivation is intuitive, how might we learn suitable parameters of logistic regression models from data? Fortunately, as with linear regression, a statistically-grounded method exists.

2.2.1 Statistical Motivation

Like in linear regression, for a statistical motivation we'll make some assumptions regarding the class labels to derive a way of finding the optimal

parameters θ^*. Suppose that $Y|(\mathbf{X} = \mathbf{x}) \sim \text{Bernoulli}(h_\theta(\mathbf{x}))$. In this case

$$p(y|\mathbf{x};\theta) = (h_\theta(\mathbf{x}))^y(1 - h_\theta(\mathbf{x}))^{1-y}.$$

We construct the log-likelihood over $D_{\text{train}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, assuming its samples are i.i.d., as

$$\begin{aligned}
l(\theta) &= \sum_{i=1}^{n} y \log(h_\theta(\mathbf{x}_i)) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i)) \\
&= \sum_{i=1}^{n} \left[y_i \log\left(\frac{1}{1 + e^{-\theta^\top \mathbf{x}_i}}\right) + (1 - y_i) \log\left(1 - \frac{1}{1 + e^{-\theta^\top \mathbf{x}_i}}\right)\right] \\
&= \sum_{i=1}^{n} \left[y_i \log\left(\frac{1}{1 + e^{-\theta^\top \mathbf{x}_i}}\right) + (1 - y_i) \left(-\theta^\top \mathbf{x}_i + \log\left(\frac{1}{1 + e^{-\theta^\top \mathbf{x}_i}}\right)\right)\right] \\
&= \sum_{i=1}^{n} \left[- \log\left(1 + e^{-\theta^\top \mathbf{x}_i}\right) - (1 - y_i)\theta^\top \mathbf{x}_i \right]
\end{aligned}$$

which we maximise numerically in θ, e.g. via gradient descent, to obtain the optimal paramateres

$$\theta^* = \underset{\theta \in \mathbb{R}^{m+1}}{\arg\max} \left[\sum_{i=1}^{n} \left[- \log\left(1 + e^{-\theta^\top \mathbf{x}_i}\right) - (1 - y_i)\theta^\top \mathbf{x}_i \right] \right].$$

Note that, in practice, it's unlikely that any samples will lie on the hyperplane $f_{\theta^*}(\mathbf{x}) = 0$ itself.

2.3 Support Vector Machines (SVMs)

Logistic regression and SVMs both fit a hyperplane to feature space. The distinction is the metric of goodness of said hyperplanes. In logistic regression, the metric of goodness is how much the parameters of said hyperplane maximise the relevant log-likelihood. SVMs, however, look to maximise the distance between the hyperplane and the nearest samples. So in some sense, logistic regression is a probabilistic approach while SVMs take a raw constraint-based approach.

The authors' insight came from structural risk minimization in which instead of focusing on minimising training error (as neural networks and decision trees were doing at the time), one focuses on minimising an upper bound on the generalisation error. The inclusion of 'support vector' becomes clear from their construction but 'machine' stood out to me as a bit odd. It

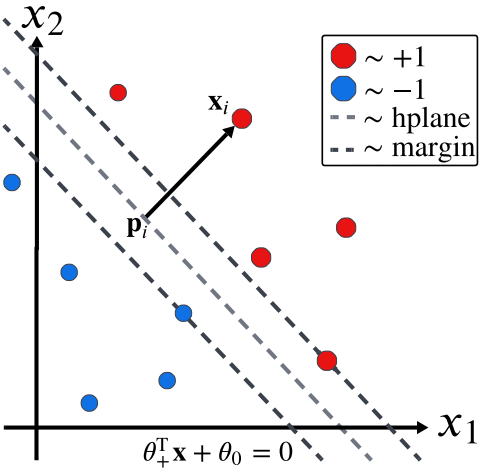


Figure 1: A hard-margin SVM in two dimensions. Samples lying on the margin are referred to as support vectors.

turns out that at time of their development, around 1960, it was common to use 'machine' when referring to algorithms that learned from data, i.e. algorithms belonging to statistical learning theory. This reflects Arthur Samuel's coining of machine learning as a term in 1959 while working at IBM.

Inconsistent terminology surrounding SVMs

Some pieces of literature describe the decision boundary learned by an SVM as strictly linear, others allow for a non-linear decision boundary. It'd be nice if authors stuck to the former and explicitly stated 'non-linear SVM' when describing the latter. In this section, the term refers to those which correspond to linear decision boundaries. I'll state explicitly when considering non-linear SVMs.

2.3.1 Hard-margin SVMs

Since we aim to fit a hyperplane to feature space, i.e. $\mathbb{R}^{m+1}$, we use the same notation for the function $f_\theta(\mathbf{x})$ corresponding to the linear decision boundary (i.e. hyperplane) used to motivate logistic regression. To make notation a bit easier, let $\theta_+ = (\theta_1, \ldots, \theta_m)$ so that θ is the ordered concatentation of θ_0 and θ_+. Further, let $\mathbf{x}_i$ denote the i^{th} sample's features and $y_i \in \{-1, 1\}$ its class such that y_i is 1 if $\theta_+^\top \mathbf{x}_i + \theta_0 > 0$ and -1 otherwise.

Let $\mathbf{p}_i$ denote the projection of $\mathbf{x}_i$ onto the hyperplane and let d_i denote the distance between $\mathbf{p}_i$ and $\mathbf{x}_i$. Noting that the normal to the hyperplane

is θ_+, we have $\mathbf{p}_i = \mathbf{x}_i - \frac{\theta_+}{||\theta_+||} d_i$ and so

$$\theta_+^\top \left(\mathbf{x}_i - \frac{\theta_+}{||\theta_+||} d_i \right) + \theta_0 = 0$$

which, after some rearranging, yields

$$d_i = \frac{\theta_+^\top \mathbf{x}_i + \theta_0}{||\theta_+||}.$$

Similarly, taking a point $\mathbf{x}_i$ whose class is -1 yields $d_i = -\frac{\theta_+^\top \mathbf{x}_i + \theta_0}{||\theta_+||}$ and so a neater way of writing this distance for an arbitrary sample $\mathbf{x}_i$ is $d_i = \frac{y_i(\theta_+^\top \mathbf{x}_i + \theta_0)}{||\theta_+||}$. Hard-margin SVMs are only applicable to linearly sparable data and their construction, from data, effectively boils down to finding which parameters maximise $\min_{i=1,\ldots,n} d_i$. That is, we seek to compute

$$\begin{aligned} \theta^* &= \arg\max_{(\theta_0,\theta_+)} \left[\min_{i=1,\ldots,n} d_i \right] \\ &= \arg\max_{(\theta_0,\theta_+)} \left[\min_{i=1,\ldots,n} \frac{y_i(\theta_+^\top \mathbf{x}_i + \theta_0)}{||\theta_+||} \right] \end{aligned}$$

which we'd like to translate into a convex optimisation problem. First, notice that computing θ^* is equivalent to solving

$$\max_{(\theta_0,\theta_+)} \frac{r}{||\theta_+||} \text{ s.t. } y_i \left(\theta_+^\top \mathbf{x}_i + \theta_0 \right) \geq r \ (i = 1, \ldots, n).$$

in which r may be scaled arbitrarily by positives, so it is equivalent to

$$\max_{(\theta_0,\theta_+)} \frac{1}{||\theta_+||} \text{ s.t. } y_i \left(\theta_+^\top \mathbf{x}_i + \theta_0 \right) \geq 1 \ (i = 1, \ldots, n).$$

This problem is still non-convex so we make a convenient switcheroo in realising that it is equivalent to solving

$$\min_{(\theta_0,\theta_+)} ||\theta_+||^2 \text{ s.t. } y_i \left(\theta_+^\top \mathbf{x}_i + \theta_0 \right) \geq 1 \ (i = 1, \ldots, n)$$

which is solved in the usual convex problem solving ways.

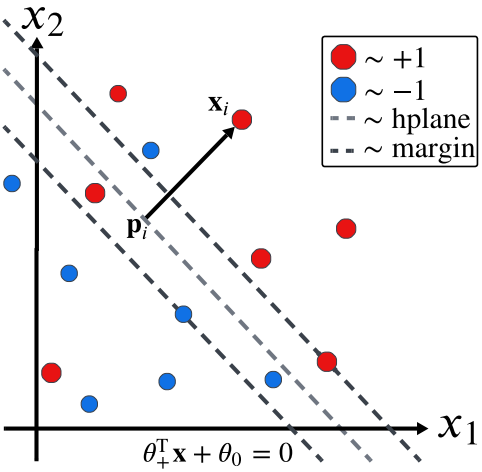


Figure 2: A soft-margin SVM in two dimensions. Samples are labelled according to their true class.

2.3.2 Soft-margin SVMs

Hard-margin SVMs are rarely applicable. In practice, samples are not entirely linearly separable and so allowing for some misclassification is pragmatic. Going from hard-margin to soft-margin is pretty straightforward, just include some slack variables $\xi = (\xi_1, \ldots, \xi_n)$ that ultimately allow the model to violate the constraints while penalising said violations. More precisely, it involves solving

$$\min_{(\theta_0, \theta_+, \xi)} \left[||\theta_+||^2 + \lambda \sum_{i=1}^{n} \xi_i \right] \text{s.t. } y_i \left(\theta_+^\top \mathbf{x}_i + \theta_0 \right) \geq 1 - \xi_i, \ \xi_i \geq 0$$

for $i = 1, \ldots, n$ where $\lambda \geq 0$ is a regularisation parameter that influences the tradeoff of margin size and misclassification rate. Larger λ corresponds to prioritising a larger margin while smaller λ corresponds to prioritising the minimisation of misclassification.

As illustrated in Figure 2, it allows for samples to be closer to the decision boundary than the margin as well as outright misclassifications. Again, it is typically solved in the usual convex problem solving ways. Going a step further, it turns out that this can be reduced to computing

$$\arg\min_{(\theta_0, \theta_+)} \left[||\theta_+||^2 + \lambda \sum_{i=1}^{n} \max \left(0, 1 - y_i \left(\theta_+^\top \mathbf{x}_i + \theta_0 \right) \right) \right]$$

which can be done using gradient descent.

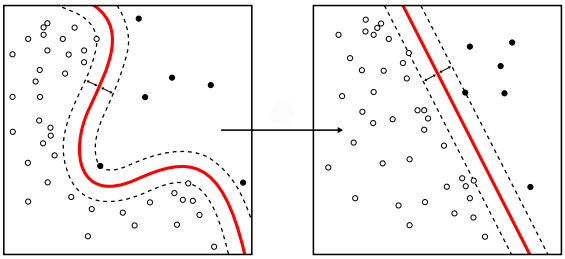


Figure 3: Non-linearly separable data being transformed into linearly separable data.

2.3.3 Non-linear SVMs

Motivating non-linear SVMs is straightforward: we'd sometimes like to separate samples by a non-linear decision boundary. With this in mind, a super intuitive approach is to find a transformation ϕ which maps samples to a space in which they are linearly separable. In said space, employ a linear SVM.

With this idea in mind, we seek to solve

$$\min_{(\theta_0,\theta_+,\xi)} \left[\frac{1}{2}||\theta_+||^2 + \lambda\sum_{i=1}^{n}\xi_i\right] \text{ s.t. } y_i\left(\phi(\theta_+)^\top \mathbf{x}_i + \theta_0\right) \geq 1-\xi_i,\ \xi_i \geq 0$$

which is difficult if the dimension of the image of ϕ is large. To make things easier, the dual of problem is optimised instead. My understanding of primal problems and their dual problems isn't great, so I'll just give the dual outright without deriving it:

$$\max_{(\alpha_1,\ldots,\alpha_n)\in[0,\lambda]^n} \left[\sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j)\right] \text{ s.t. } \sum_{i=1}^{n}\alpha_i y_i = 0$$

where $K(\mathbf{x},\mathbf{z}) = \langle\phi(\mathbf{x}),\phi(\mathbf{z})\rangle$ is a pre-chosen kernel. See how it never requires an explicit computation involving ϕ as long as a closed form of $K(\mathbf{x},\mathbf{z})$ not involving ϕ is available. This is referred to as the kernel trick and reduces having to solve an optimisation problem in a space whose dimension is the number of features to a space whose dimension is the number of training samples.

At inference time, given some $\mathbf{x}$, we seek to compute

$$\begin{aligned} y &= \text{sign}\left(\theta_+^\top\phi(\mathbf{x}) + \theta_0\right) \\ &= \text{sign}\left(\sum_{i=1}^{n}\alpha_i y_i K(\mathbf{x}_i,\mathbf{x}) + b_k\right) \end{aligned}$$

where $b_k = y_k - \sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_k)$ for some $(\mathbf{x}_k, y_k)$ that satisfies $\alpha_k \in (0, \lambda)$. Note that the second line in the equation above is derived using to argument made in deriving the dual.

The two simplest kernels are polynomial kernels and the radial basis function (RBF) kernel, the latter of which is far more popular. That said, polynomial kernels have had their place in applying SVMs to natural language processing tasks. The polynomial kernels

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z} + c)^d$$

capture feature interactions up to degree d, which is useful when the separation of classes depends on combinations of input variables, e.g. quadratic boundaries in $\mathbb{R}^2$. Therein lies their weakness: polynomial kernels can be limited if said decision boundary is particuarly irregular. The RBF kernel

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{z}||^2}{2\sigma^2}\right)$$

is more flexible as a result of it measuring similarity based on distance, effectively producing localised bumps in feature space. From this, the model is able to form very non-linear decision boundaries. So, all in all, polynomial kernels provide interpretable higher-order interactions, while RBF kernels offer universal approximation.

2.4 Decision Trees and Random Forests

I find decision trees and random forests so boring that I'm not going to write about them and will instead point to Figure 4 which illustrates random forests well. This section mostly exists so that the list of subsections in this section form a well-rounded list of supervised learning methods.

Despite my disinterest in them, learning about decision trees encourages you to understand entropy which is good. There are other topics which encourage the same thing though, e.g. Variational Autoencoders, through the use of KL-divergences. Also, random forests are a nice introduction to ensemble methods which nicely demonstrate how to prevent overfitting by reducing variance — directly illustrating the importance of the bias-variance tradeoff! Oh, and bootstrapping. One day I'll do them justice and write this subsection properly.

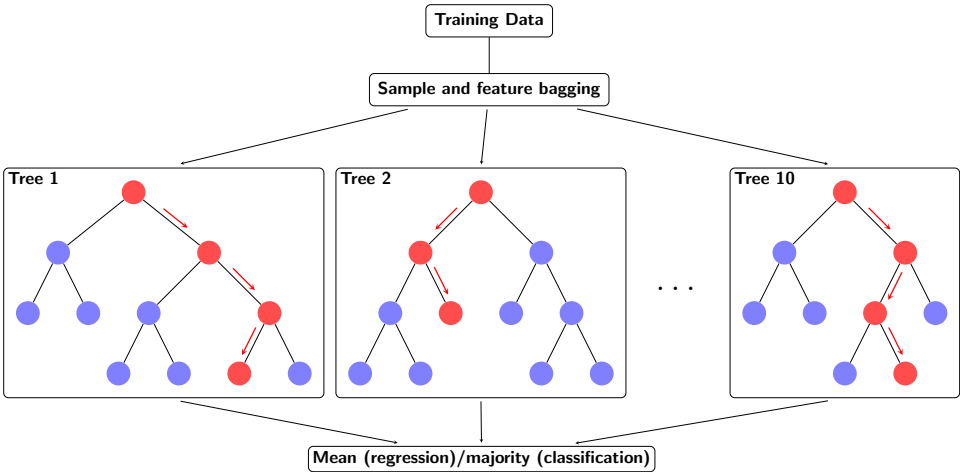


Figure 4: A random forest.

Logistic regression > SVM/RF when?

This is a pretty natural question once shown more sophisticated methods which deal with linearly separable and non-linearly separable data. A significant benefit of logistic regression is that it allows for statistical significance tests on parameters. It also helps that its implementation and interpretation are straightforward.

2.5 The Bias-Variance Tradeoff

The bias-variance tradeoff is a statement pertaining to the goodness of estimators (learning algorithms) according to mean square error (MSE) in terms of their variance — with respect to training data — and the square of their bias — due to architecture-related assumptions. It turns out that the optimal estimator, according to mean square error, is a careful tradeoff of both. Before deriving the tradeoff, let's consider the consequences of high squared bias and high variance: underfitting and overfitting.

2.5.1 Underfitting and Overfitting

Crudely put, if the architecture pertaining to an estimator is too simple to represent the point estimate then the estimator's bias (and thus its square) will be large in magnitude. Such an estimator underfits that which it is intended to model. I like to think of such an estimator as being almost

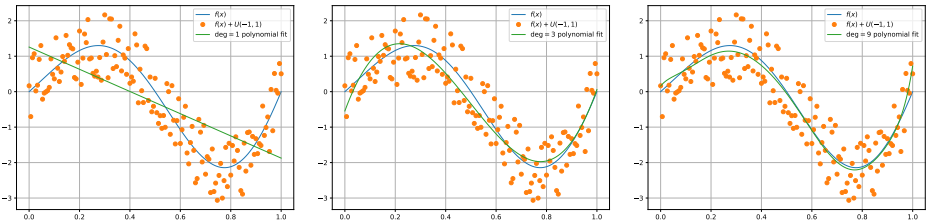


Figure 5: Lots of data, little noise.

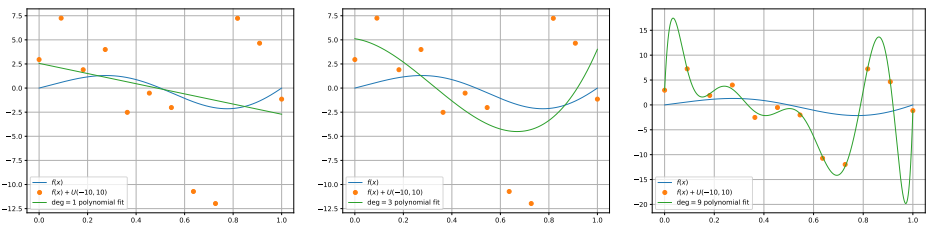


Figure 6: Little data, lots of noise.

invariant to the training data. As a practical example, think of fitting a line to a very non-linear function as in the leftmost plots of Figure 5 and Figure 6. Underfitting is reflected during training by both high training error and high validation error.

At the other extreme, if the architecture pertaining to an estimator is more complex than is necessary (e.g. overparameterised) to represent the point estimate then the excess parameters often[1] cause the model to fit the noise in the training data. In this case, the estimator is highly variant with respect to the training data. Such an estimator overfits the training data instead of fitting the point estimate. As a practical example, think of fitting a polynomial from small and heavily-noised training data, as in the rightmost plot of Figure 6. Overfitting is reflected during training by low training error and high validation error (relative to the training error).

2.5.2 Derivation with MSE loss

Recall that target values are often effected by noise, i.e. $Y|(\mathbf{X} = \mathbf{x}) = f(\mathbf{x}) + \epsilon(\mathbf{x})$ where $\epsilon(\mathbf{x}) \sim \mathcal{N}(0, \sigma^2(\mathbf{x}))$ and so

$$Y|(\mathbf{X} = \mathbf{x}) \sim \mathcal{N}\left(f(\mathbf{x}), \sigma^2(\mathbf{x})\right).$$

[1] See the double-descent phenomenon in Figure 8.

Given a fixed architecture, the estimate $\hat{f}_D$ of f is a purely function of the training dataset $D \subset (\Omega_{\mathbf{X}} \times \Omega_Y)^n$. Such a dataset is a realisation of the random variable $\mathcal{D}$ distributed according to $p(\mathbf{x}, y)^{\otimes n}$. In line with this, the bias and variance derived are that of the estimator $f_{\mathcal{D}}$.

As is common in regression contexts, the goodness of an estimate $\hat{f}_D$ is given by its expected risk

$$R\left(\hat{f}_D\right) = \mathbb{E}_{(\mathbf{X},Y)}\left[\left(Y - \hat{f}_D(\mathbf{X})\right)^2\right]$$

in which mean square error (MSE) is used as loss. Thus, the quality of the corresponding estimator (or learning algorithm) used to determine $\hat{f}_D$ from $D \in \Omega_{\mathcal{D}}$ is given by

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[R\left(\hat{f}_{\mathcal{D}}\right)\right] \\
&= \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[\left(Y - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[\left(Y - f(\mathbf{X}) + f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[(Y - f(X))^2\right]\right] + \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&+ \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[2(Y - f(\mathbf{X}))\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\right]\right].
\end{aligned}$$

Let's address this term-by-term beginning with the term in red. See that

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[2(Y - f(\mathbf{X}))\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\right]\right] \\
&= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_{Y|\mathbf{X}}\left[(Y - f(\mathbf{X}))\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\right]\right]\right] \\
&= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\mathbb{E}_{Y|\mathbf{X}}\left[Y - f(\mathbf{X})\right]\right]\right] \\
&= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\cdot 0\right]\right] \\
&= 0
\end{aligned}$$

in which the tower property of conditional expectations

$$\begin{aligned}
\mathbb{E}_{(\mathbf{X},Y)}[g(\mathbf{X}, Y)] &= \iint g(\mathbf{x}, y) f_{(\mathbf{X},Y)}(\mathbf{x}, y) dx dy \\
&= \int \left(\int g(\mathbf{x}, y) f_{Y|\mathbf{X}}(y|\mathbf{x}) dy\right) f_{\mathbf{X}}(\mathbf{x}) dx \\
&= \mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_{Y|\mathbf{X}}\left[g(\mathbf{X}, Y)\right]\right]
\end{aligned}$$

is applied. As such, the quantity of interest reduces to only two terms as in

$$\mathbb{E}_{\mathcal{D}}\left[R\left(\hat{f}_{\mathcal{D}}\right)\right] = \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[(Y - f(\mathbf{X}))^2\right]\right] + \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right]$$

of which we will now address the first in orange. Noting that

$$\mathbb{E}_{Y|(\mathbf{X}=\mathbf{x})}\left[Y - f(\mathbf{x})\right] = 0$$

for all $\mathbf{x} \in \Omega_{\mathbf{X}}$ and that what is inside the expectation over $\mathcal{D}$ is independent of $\mathcal{D}$, we see that the first term reduces to

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{(\mathbf{X},Y)}\left[(Y - f(\mathbf{X}))^2\right]\right] &= \mathbb{E}_{(\mathbf{X},Y)}\left[(Y - f(\mathbf{X}))^2\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_{Y|\mathbf{X}}\left[(Y - f(\mathbf{X}))^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\mathbb{E}_{Y|\mathbf{X}}\left[\left(Y - \mathbb{E}_{Y|\mathbf{X}}[Y]\right)^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\mathrm{Var}\left(Y|\mathbf{X}\right)\right] \\
&= \mathbb{E}_{\mathbf{X}}[\sigma^2(\mathbf{X})]
\end{aligned}$$

This is simply the expected noise, e.g. due to imperfect calibration in the instruments used to obtain samples.

To address the term in green, let $\bar{f}(\mathbf{x}) = \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{x})\right]$ for all $\mathbf{x} \in \Omega_{\mathbf{X}}$ and see that

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \bar{f}(\mathbf{X}) + \bar{f}(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \bar{f}(\mathbf{X})\right)^2\right] + \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(\bar{f}(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&+ 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X}) - \bar{f}(\mathbf{X})\right)\left(\bar{f}(\mathbf{X}) - \hat{f}_{\mathcal{D}}(\mathbf{X})\right)\right]\right].
\end{aligned}$$

See that the final term vanishes as $\mathbb{E}_{\mathcal{D}}\left[\bar{f}(\mathbf{x}) - \hat{f}_{\mathcal{D}}(\mathbf{x})\right] = 0$ for all $\mathbf{x} \in \Omega_{\mathbf{X}}$

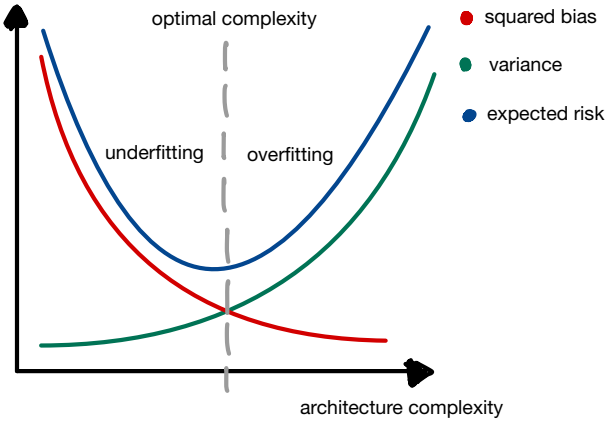


Figure 7: Expected risk, squared bias and variance against architecture complexity.

and so

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(f(X)-\hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X})-\bar{f}(\mathbf{X})\right)^2\right] + \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\mathbf{X}}\left[\left(\bar{f}(\mathbf{X})-\hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X})-\bar{f}(\mathbf{X})\right)^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\bar{f}(\mathbf{X})-\hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2\right]\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X})-\mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(X)\right]\right)^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathcal{D}}(\mathbf{X})-\mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{X})\right]\right)^2\right]\right].
\end{aligned}$$

Put together, we obtain

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[R\left(\hat{f}_{\mathcal{D}}\right)\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\left(f(\mathbf{X})-\mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{X})\right]\right)^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathcal{D}}(\mathbf{X})-\mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{X})\right]\right)^2\right]\right] + \sigma^2 \\
&= \mathbb{E}_{\mathbf{X}}\left[\mathrm{Bias}\left(\hat{f}_{\mathcal{D}}(\mathbf{X})\right)^2 + \mathrm{Var}\left(\hat{f}_{\mathcal{D}}(\mathbf{X})\right)\right] + \sigma^2
\end{aligned}$$

where $\sigma^2 = \mathbb{E}_{\mathbf{X}}\left[\sigma^2(\mathbf{X})\right]$ (ignore the poor choice of notation). With this in mind, it's clear that we do not seek the estimator $\hat{f}_{\mathcal{D}}$ which minimises expected square bias or expected variance individually. Instead, we seek the estimator which hits the sweet spot in minimising their sum. An illustration of this statement, as the estimator's architecture is made more complex, is given in Figure 7.

Beyond MSE

Natural question: MSE is a common choice of loss when dealing with regression but not classification, in which cross-entropies are typically used as loss, so why do derivations of the bias-variance tradeoff use MSE?

The quick answer is that it's the only choice which yields a relatively simple derivation of this elegant decomposition of estimator error into the square of its bias and its variance. I read online somewhere that some other choices of loss also yield decompositions into squared bias and variance but that their derivation is less elegant. There seems to be a good amount of work in this area but it's too in-depth for this document.

2.5.3 Double descent

Double descent refers to a behaviour observed in select set ups in which overparameterised models seemingly fly in the face of the bias-variance tradeoff. As the number of parameters (i.e. the model complexity) increases, the test error increases in line with the usual bias-variance tradeoff until reaching the number of training samples n. At this point, the test error spikes and training error hits 0. That is, the model is able to entirely interpolate the training data. The phenomenon begins as the number of parameters exceeds n. Not only does training error remain 0 (expected) but the test error begins to decrease and, in many set ups, reduces to an amount below the minimum test error observed during the traditional bias-variance region.

The phenomenon is illustrated nicely in Figure 8 and motivates the massively overparameterised contemporary models used these days. AlexNet is an example of an overparameterised model lying in the second descent region with around 60 million parameters but only 1.2 million training samples.

I like to imagine fitting a third degree polynomial by polynomials of higher and higher degree. If double descent were observed in this set up, one might expect wild polynomial fits until the polynomial's degree matches the number of training samples. At which point, the model has interoplated the training data perfectly and begins to regulate how it fits what's inbetween.

There has been a good amount of work investigating the phenomenon and some of the mystery has reduced as a result. The most convincing hand-wavy explanation of the behaviour is that parameter exploration via gradient descent is inherently regulatory. Not only that but the use of explcit regularisation, like SGD and Adam, on top of the self-regularisation

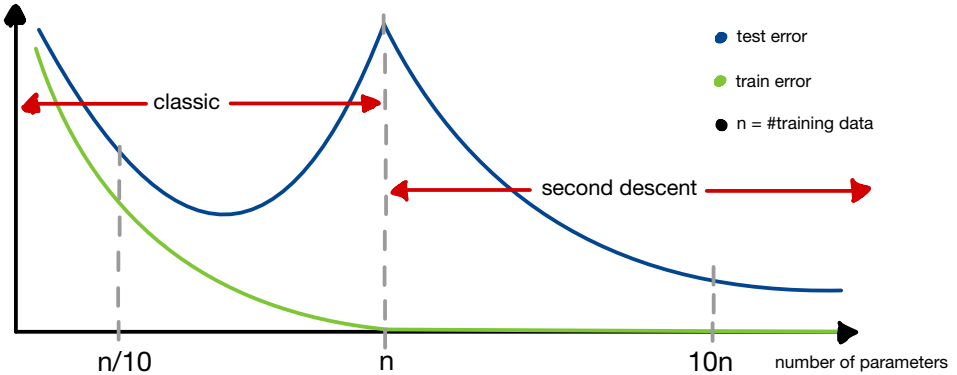


Figure 8: The double descent phenomenon, $n = |D_{\text{train}}|$.

bias toward parameters which offer 'simpler' representations.

3 Parameter Exploration and its Optimisation

In fitting a parametric model $f_\theta : \Omega_{\mathbf{X}} \to \Omega_Y$, the loss induced by a choice of model parameters θ for a sample $(\mathbf{x}, y)$ is given by $\mathcal{L}\left(y, f_\theta(\mathbf{x})\right)$ for some loss function $\mathcal{L} : \Omega_Y \times \Omega_Y \to \mathbb{R}_{\geq 0}$. In line with this, let the empirical risk R_D over a given dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ be given by

$$\begin{aligned} R_D : \Theta &\to \mathbb{R}_{\geq 0} \\ \theta &\mapsto \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, f_\theta(\mathbf{x}_i)) \end{aligned}$$

In fitting f_θ, we seek to compute

$$\theta^* = \arg\min_{\theta \in \Theta} R_D(\theta),$$

i.e. to minimise the empirical risk over the dataset in the parameters. As such, methods of function minimisation are of interest.

Many function minimisation methods exist: Newton's method, genetic algorithms, gradient descent, etc. but most are horribly slow/unstable in high dimensions; not gradient descent though! As a result, gradient descent is *the* method for computing good parameters numerically in deep learning, hence the inclusion of this section.

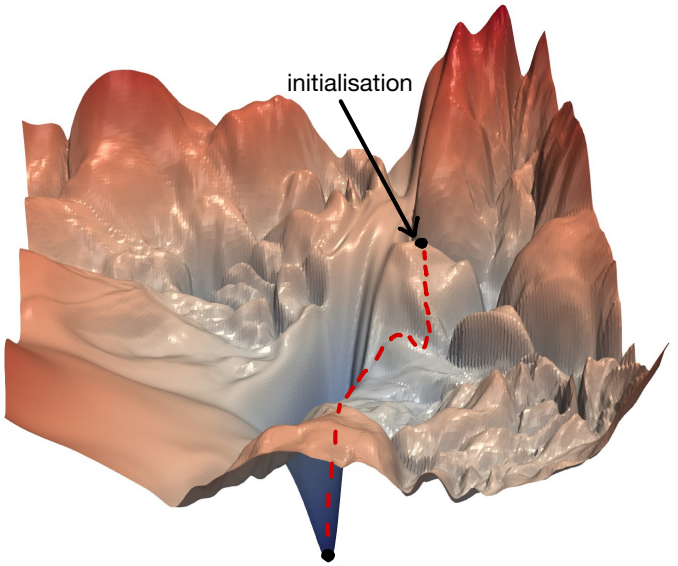


Figure 9: The graph $\{(\theta, R_D(\theta))|\theta \in \mathbb{R}^2\}$ of a complex loss landscape and a visualisation of how me might hope gradient descent looks.

Illustrations of loss landscapes are dataset-dependent!

When loss landscapes are illustrated, they are really plots of the graph $\{(\theta, R_D(\theta))|\theta \in \Theta\}$. That is, they are shaped by the fixed dataset D. As such, computations of the form $R_{D'}(\theta)$ for $D' \subsetneq D$, e.g. computations related to mini-batch or stochastic gradient descent, don't necessarily coincide with $R_D(\theta)$.

3.1 Gradient Descent

Since gradient descent is a general function minimisation method and not specific to machine learning, I'll use more general mathsy notation than θ, R_D, etc.

Say we're interested in finding minima of a function f whose co-domain is $\mathbb{R}_{\geq 0}$. A natural idea is to start with a guess $\mathbf{x}^{(0)}$ and send it in the direction in which f most decsends locally from $\mathbf{x}^{(0)}$, or equivalently the opposite of the direction in which f most ascends locally from $\mathbf{x}^{(0)}$, i.e. computing

$$\underset{||\mathbf{v}^{(0)}||=1}{\arg\max} f\left(\mathbf{x}^{(0)} + \eta \mathbf{v}^{(0)}\right),$$

for some small $\eta > 0$, and updating our guess to $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \eta \mathbf{v}^{(0)}$. Iteratively applying this idea yields the usual gradient descent rule

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \mathbf{v}^{(t)}.$$

Eventually, after taking a ton of these small steps, we hope to reach a minimum of f.

3.1.1 In which direction does f most ascend locally from $\mathbf{x}^{(t)}$?

In the following physicist-like argument, we consider only unit vectors $\mathbf{v}$. This is because we care only about the direction of the vector in question and an equation later on simplifies quite nicely due to its unit length.

Multivariate Taylor expansion

Recall that if $f : \mathbb{R}^m \to \mathbb{R}$ then its second order Taylor expansion about $\mathbf{a} \in \mathbb{R}^m$ is given by

$$f(\mathbf{x}) = f(\mathbf{a}) + \nabla f(\mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) + \frac{1}{2}(\mathbf{x} - \mathbf{a}) \cdot \nabla^2 f(\mathbf{a})(\mathbf{x} - \mathbf{a}).$$

In gradient ascent, we are looking for the unit vector $\mathbf{v}$ whose direction f increases most locally from $\mathbf{x}$, i.e. we seek

$$\arg\max_{||\mathbf{v}||=1} \Big[f(\mathbf{x} + \eta \mathbf{v}) - f(\mathbf{x}) \Big]$$

where $\eta > 0$ is small. Approximating $f(\mathbf{x} + \eta \mathbf{v})$ using its Taylor expansion about $\mathbf{x}$ yields

$$f(\mathbf{x} + \eta \mathbf{v}) - f(\mathbf{x}) \approx \nabla f(\mathbf{x}) \cdot \eta \mathbf{v}.$$

Thus, the problem translates to finding the unit vector $\mathbf{v}$ that maximises

$$\nabla f(\mathbf{x}) \cdot \eta \mathbf{v} = ||\nabla f(\mathbf{x})|| \cdot ||\eta \mathbf{v}|| \cdot \cos(\theta) = \eta ||\nabla f(\mathbf{x})|| \cdot \cos(\theta)$$

where $\theta \in [0, \pi)$ denotes the angle between $\nabla f(\mathbf{x})$ and $\mathbf{v}$. This expression is maximised when $\cos(\theta) = 1$, i.e. when $\theta = 0$, which necessitates $\mathbf{v}$ having the same direction as $\nabla f(\mathbf{x})$. So $\mathbf{v}$ is a unit vector in the direction of $\nabla f(\mathbf{x})$, i.e. $\mathbf{v} = \frac{\nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||}$. To see that f ascends after being sent in the direction of $\mathbf{v} = \frac{\nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||}$ from $\mathbf{x}$, note that

$$f(\mathbf{x} + \eta \mathbf{v}) - f(\mathbf{x}) \approx \nabla f(\mathbf{x}) \cdot \eta \mathbf{v} = \eta \frac{\nabla f(\mathbf{x}) \cdot \nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||} = \eta ||\nabla f(\mathbf{x})|| > 0.$$

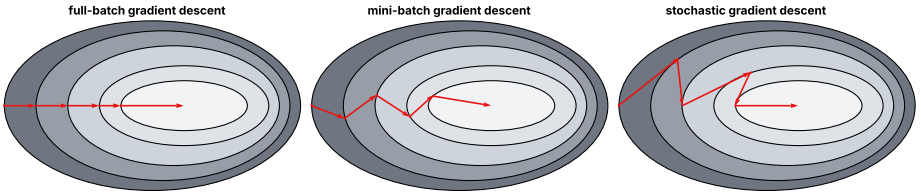


Figure 10: How the convergence of each method might look.

The edge case in which $\nabla f(\mathbf{x}) = 0$ corresponds to $f(\mathbf{x})$ being a local maximum.

3.1.2 Sensitivity to $\mathbf{x}^{(0)}$

I don't know much about initialisation. I know that it's a bad idea to start at 0 (if certain symmetries hold then parameters may change identically), points of large magnitude (exploding gradients) and points of small magnitude (vanshing gradients). Simon Price speaks of some interesting quirks of parameter initialisation around 01:08:00 in a YouTube interview.

One day when my interest in initialisation is sparked, I'll write this part properly.

3.1.3 Batch learning

If your training dataset is huge and you don't have enough VRAM to store gradient information then do not fear! It turns out that we can approximate full gradient updates reasonably well using many smaller updates over subsets of the dataset. How well mini-batch learning and stochastic gradient descent perform (and even just *that* they perform) surprised me when I first learned about them.

Epoch terminology

During training, the model tweaks its parameters based on training data. In practice, the model does so for the same sample many times. Each full cycle of the model having learned from the training data is called an epoch.

1) Full-batch

Full-batch gradient descent involves one parameter update per epoch as in

$$\theta^{(t+1)} = \theta^{(t)} - \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \mathcal{L}\left(y_i, f_{\theta^{(t)}}(\mathbf{x}_i)\right).$$

2) Mini-batch

The information stored in order to compute gradients during training might be too much for your PC's VRAM. If so, instead perform mini-batch gradient descent in which the dataset D is partitioned into m disjoint subsets $B_1, \ldots, B_m$, referred to as batches, and m parameter updates are made according to

$$\theta^{(t+j)} = \theta^{(t+j-1)} - \frac{1}{|B_j|}\sum_{(\mathbf{x},y)\in B_j} \nabla_\theta \mathcal{L}\left(y, f_{\theta^{(t+j-1)}}(\mathbf{x})\right).$$

We see that if we split into m batches then a full epoch during training corresponds to m parameter updates (or optimisation steps).

3) Stochastic gradient descent (SGD)

It turns out that updating based on a single sample $(\mathbf{x}, y)$ (uniformly randomly sampled), as in

$$\theta^{(t+1)} = \theta^{(t)} - \nabla_\theta \mathcal{L}\left(y, f_{\theta^{(t)}}(\mathbf{x})\right)$$

is viable. This surprised me a ton when I first read about it: I would have thought that it'd produce nonsense parameters. Unsurprisingly, SGD yields relatively unstable convergence, a bit like the steps that a drunk man would take in walking down a hill.

Terminology warning

What we call mini-batch gradient here is referred to by some as stochastic gradient descent. Confusing.

3.1.4 Batch + Layer + RMS normalisation

I've only seen these forms of normalisation apply to training neural networks. I'm unaware of them being grounded in theory. They're very widley used so I've included them.

1) Batch norm

For a batch $B = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$, batch normalisation is the act of normalising the post-activation values of some layer of d neurons in a neural network. Denote the d activation values for the sample $\mathbf{x}_i \in B$ as $(a_{i,1}, \dots, a_{i,d})$. The entire batch's activation values for said layer can be written as a matrix as in

$$\begin{bmatrix} a_{1,1} & \dots & a_{1,d} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,d} \end{bmatrix}.$$

Batch norm first normalises the columns of A as in

$$\begin{bmatrix} \hat{a}_{1,1} & \dots & \hat{a}_{1,d} \\ \vdots & \ddots & \vdots \\ \hat{a}_{m,1} & \dots & \hat{a}_{m,d} \end{bmatrix} = \begin{bmatrix} (a_{1,1} - \mu_1)/\sqrt{\sigma_1^2 + \epsilon} & \dots & (a_{1,d} - \mu_d)/\sqrt{\sigma_d^2 + \epsilon} \\ \vdots & \ddots & \vdots \\ (a_{m,1} - \mu_1)/\sqrt{\sigma_1^2 + \epsilon} & \dots & (a_{m,d} - \mu_d)/\sqrt{\sigma_d^2 + \epsilon} \end{bmatrix}$$

where $\mu_j = \frac{1}{m}\sum_{i=1}^m a_{i,j}$, $\sigma_j^2 = \frac{1}{m}\sum_{i=1}^m (a_{i,j} - \mu_j)^2$ and $\epsilon > 0$ is small, appearing for the sake of computational stability. At this point, the columns have mean 0 and variance 1. The output of batch norm for sample $i \in \{1, \dots, m\}$ is given by

$$(\gamma_1 \hat{a}_{i,1} + \beta_1, \dots, \gamma_d \hat{a}_{i,d} + \beta_d)$$

where $\gamma_1, \dots, \gamma_d$ and $\beta_1, \dots, \beta_d$ are learnable parameters.

I don't think that the effect of batch norm is immediately clear and the informal explanations given in the paper which introduced it have been largely discarded since. Perhaps it helps avoid exploding/vanishing gradients, I honestly don't know lol.

2) Layer norm

If batch statistics aren't particularly meaningful or batch size is small then layer norm may be preferable. Layer norm applies to individual samples, as opposed to entire batches, and so normalises over the features instead. Given activation values $a_1, \dots, a_d$, layer norm first normalises the activation values as in

$$(\hat{a}_1, \dots, \hat{a}_d) = \left(\frac{a_1 - \mu}{\sqrt{\sigma^2 + \epsilon}}, \dots, \frac{a_d - \mu}{\sqrt{\sigma^2 + \epsilon}} \right)$$

where $\mu = \frac{1}{d}\sum_{j=1}^d a_j$ and $\sigma^2 = \frac{1}{d}\sum_{j=1}^d (a_j - \mu)^2$. Then, as in batch norm, we scale and shift to obtain

$$(\gamma_1 \hat{a}_1 + \beta_1, \dots, \gamma_d \hat{a}_d + \beta_d)$$

where $\gamma_1, \ldots, \gamma_d$ and $\beta_1, \ldots, \beta_d$ are learnable parameters. As with batch norm, I honestly don't know the influence layer norm has on training.

3) RMS norm

RMS norm is a slight simplification of layer norm in which the root mean square is used to normalise activations rather the standard deviation, i.e. we do not subtract the mean. As a result, the normalised activations don't need to have mean 0. Given activation values $a_1, \ldots, a_d$, RMS norm normalises as in

$$(\hat{a}_1, \ldots, \hat{a}_d) = \left(\frac{a_1}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} a_j^2 + \epsilon}}, \ldots, \frac{a_d}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} a_j^2 + \epsilon}} \right)$$

where $\epsilon > 0$ is small for computational stability. Finally, as before, we scale (typically without an additive shift) to obtain

$$(\gamma_1 \hat{a}_1, \ldots, \gamma_d \hat{a}_d)$$

where $\gamma_1, \ldots, \gamma_d$ are learnable parameters. I also don't really know why this works as well as it does, but empirically it seems to behave similarly to layer norm while being a bit cheaper. I first came accross RMS norm in reading about OpenAI's open weight GPT-OSS models. Seems they favour RMS norm in more modern architectures over layer norm.

3.2 Regularisation

Overfitting is a pain. How might we regulate the learning procedure in a way that (hopefully) prevents overfitting? Regularisation!

The most well known methods of regularisation are L1 and L2 regularisation. Both involve adding a penalty term to the loss function used during training with the intention of encouraging some behaviour in the learning process. There are tons of ways of motivating both L1 and L2 but the most intuitive to me (by a long shot) is a Bayesian approach: assume independence of parameters and impose a prior distribution on them according to whatever bias we hope to bake into the learning process. Then, instead of maximum likelihood estimation, compute

$$\begin{aligned} \arg\max_{\theta \in \mathbb{R}^{m+1}} \log(p(\theta|D)) &= \arg\max_{\theta \in \mathbb{R}^{m+1}} \Big[\log(p(D|\theta)) + \log(p(\theta)) \Big] \\ &= \arg\min_{\theta \in \mathbb{R}^{m+1}} \left[-\sum_{i=1}^{n} \log(p(y_i|\mathbf{x}_i, \theta)) - \sum_{j=1}^{m} \log(p(\theta_j)) \right]. \end{aligned}$$

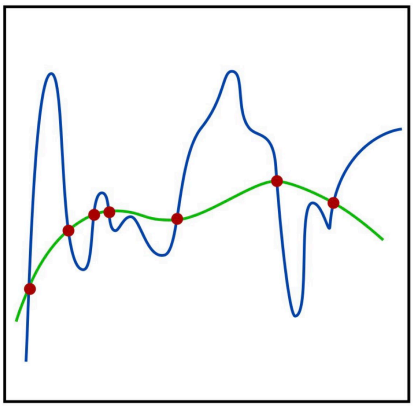


Figure 11: Motivating regularisation. Two fits, both with 0 error on selected datapoints.

3.2.1 L1 regularisation (LASSO)

If we would like our to-be-learned parameters to be sparse, then it makes sense to choose a prior which has a lot of mass around 0 and tapers off rather harshly at the tails. A great choice for this is a Laplace prior with mean 0 and variance $2/\lambda$ which yields L1 regularisation. The corresponding density function is given by $p(\theta_j) = \lambda \exp(-2\lambda|\theta_j|)$ and explicit examples are illustrated in Figure 12.

See how as λ increases, the variance of the corresponding Laplace distribution decreases and more of the mass becomes centred around 0. The result is that the to-be-learned parameters are further encouraged to flatten out around 0. In line with this, assuming $\theta_1, \ldots, \theta_m \overset{\text{i.i.d.}}{\sim} \text{Lap}(0, 2/\lambda)$, the quantity we seek to compute is given by

$$\arg\max_{\theta \in \mathbb{R}^{m+1}} \log(p(\theta|D)) = \arg\min_{\theta \in \mathbb{R}^{m+1}} \left[-\sum_{i=1}^{n} \log(p(y_i|\mathbf{x}_i, \theta)) + \lambda \sum_{j=1}^{m} |\theta_j| \right].$$

3.2.2 L2 regularisation (Ridge)

If we would like no given parameter to be too large then imposing some sort of MSE penalty on the parameters makes sense. From the Bayesian point of view, this is achieved by imposing a normal prior on the parameters and yields L2 regularisation. That is, if $\theta_1, \ldots, \theta_m \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2)$ then the quantity we seek is

$$\arg\max_{\theta \in \mathbb{R}^{m+1}} \log(p(\theta|D)) = \arg\min_{\theta \in \mathbb{R}^{m+1}} \left[-\sum_{i=1}^{n} \log(p(y_i|\mathbf{x}_i, \theta)) + \lambda \sum_{j=1}^{m} \theta_j^2 \right]$$

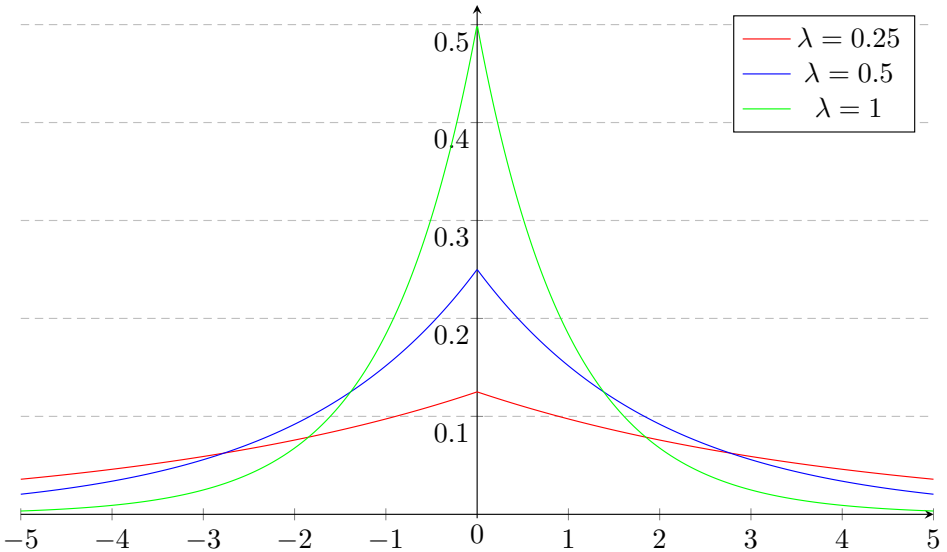


Figure 12: Laplace pdf with means 0 and decreasing variances according to $\lambda \in \{0.5, 1, 2, 3\}$.

where $\lambda = 1/2\sigma^2$.

There's a particularly elegant way to visualise the effects of L1 and L2 reglularisation in terms of how they change the shape of the corresponding loss landscape. It's as if each point in the landscape is sunk according to its distance from the origin and the hyperparameter λ. The sharpness or curvature of said sinking is prior-dependent: for L1 regularisation it is far sharper, hence the origin acts as more of a sinkhole than in L2 regularisation, inducing sparser minima.

3.2.3 Early stopping

To motivating early stopping, consider the familiar scenario of training via gradient descent. A natural thought when logging the training loss is whether or not a decrease in training loss from one epoch to the next genuinely pertains to a model that better generalises. How can we be confident that our model isn't simply overfitting the training data if all we see is decrease in training loss?

Early stopping partially alleviates this fear by splitting the training dataset into a smaller training set and a validation set (perhaps 80%/20%). After each epoch (or every few), compute both training and validation losses.

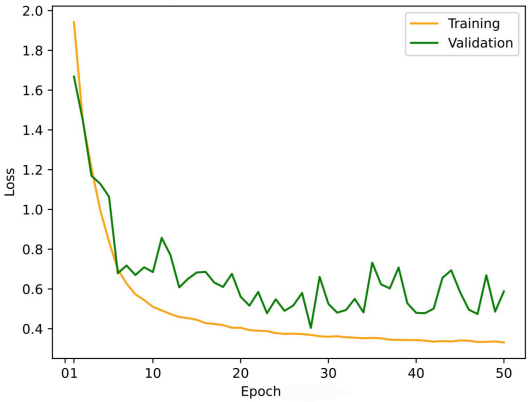


Figure 13: Training and validation losses over 50 epochs. Lowest validation obtained at epoch 28.

Said validation loss acts as some measure of the model's ability to generalise. If training loss is decreasing while validation loss is not then the model may be overfitting. As illustration, consider Figure 13 in which the training loss continues to decrease in the number of epochs while the validation loss seems to obtain its lowest value around epoch 28.

3.2.4 Dropout

Dropout is a form of regularisation specific to neural network architectures (I think). Its rough motivation is that we'd prefer not to be overly-reliant on any given subset of neurons at inference time. To prevent such an over-reliance, at the beginning of each epoch, independently set the activation of each neuron to 0 with some probability p. This way some portion of neurons are silenced during training for said epoch. As a result, its corresponding parameters are not updated during backpropagation. This idea is illustrated in Figure 14. Make sure to not apply dropout at ineference time!

3.3 Momentum + Adaptive Learning Rates

There are some intuitive ideas which aim to accelerate the gradient descent process. One is momentum, which increases steps made when gradients consistently pointed in the same direction over previous steps. Another is an adaptive learning rate, where each parameter has its own step size that adapts over time to its past gradients.

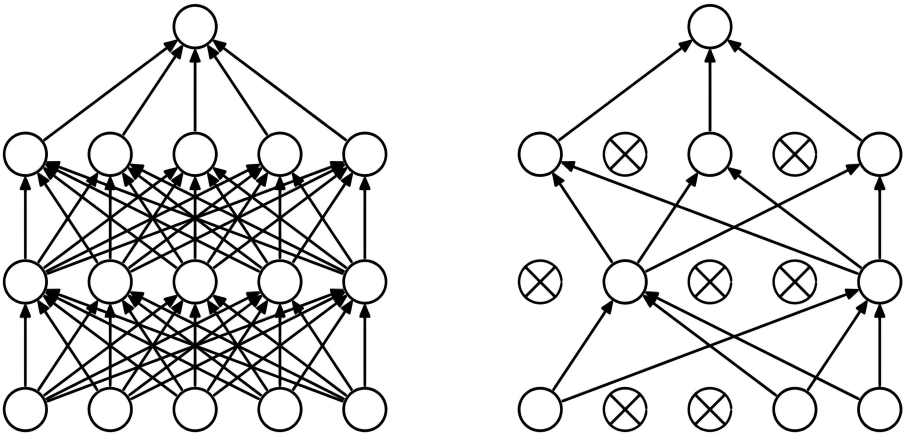


Figure 14: Dropout.

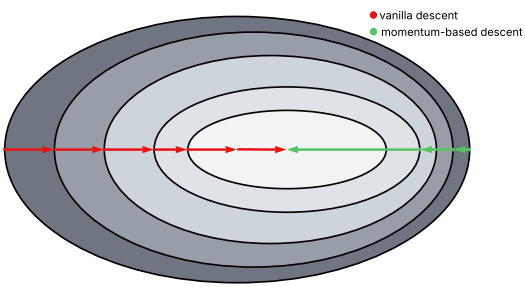


Figure 15: How we might hope convergence would look using momentum.

3.3.1 Momentum

Think of a ball rolling downhill: momentum lets it keep moving in the same direction which smoothes out noisy gradients. If gradients keep pointing in the same direction then momentum builds. If gradients oscillate, e.g. traversing like a ravine, then the motion is dampened and momentum is lost.

We impose a momentum-like idea by introducing a velocity vector

$$\mathbf{v}^{(t+1)} = \mu \mathbf{v}^{(t)} - \eta \nabla R_D \left(\theta^{(t)} \right)$$

with $\mathbf{v}^{(0)} = 0$ and change the gradient descent update rule to

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla R_D \left(\theta^{(t)}\right) + \mu \mathbf{v}^{(t)}$$

where $\mu \in [0, 1)$ is the momentum coefficient. Typical values include, $\mu \in \{0.9, 0.95, 0.99\}$. To get a sense of how this refelcts a momentum-like idea, see that

$$\mathbf{v}^{(1)} = -\eta \nabla R_D \left(\theta^{(0)}\right)$$

and so

$$\mathbf{v}^{(2)} = -\eta \left(\mu \nabla R_D \left(\theta^{(0)}\right) + \nabla R_D \left(\theta^{(1)}\right)\right).$$

If $\mu \approx 1$ and the gradients $\nabla R_D \left(\theta^{(0)}\right) \approx \nabla R_D \left(\theta^{(1)}\right)$ then $\mathbf{v}^{(2)}$ will confidently boost $\theta^{(3)}$ in the direction of the first two gradients. If $\nabla R_D \left(\theta^{(0)}\right) \approx -\nabla R_D \left(\theta^{(1)}\right)$ then, as intuition would suggest, $\mathbf{v}^{(2)} = 0$.

3.3.2 Adaptive learning rates

Why have a fixed learning rate? Why have the same learning rate for each parameter? Adaptive learning rates aim to alleviate the oversimplications these questions pertain to. For some intuition as to how learning rates might change: if up to the t^{th} optimisation step the gradients for a given parameter have been large then it has presumably converged decently so its learning rate would ideally get smaller. AdaGrad does this is by storing a per parameter running total of gradients, as in

$$G_i^{(t)} = \sum_{\tau=1}^{t} \left(\nabla_{\theta_i} R_D \left(\theta_i^{(\tau)}\right)\right)^2$$

and switching the optimisation step to

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta_i \nabla_{\theta_i}^{(t)} R_D \left(\theta_i^{(t)}\right)$$

where $\eta_i^{(t)} = \frac{\eta}{\sqrt{G_i^{(t)}} + \epsilon}$ is the i^{th} parameter's learning rate at step t. There are more elaborate adaptive learning rate methods than this, e.g. RMSProp and Adam, but AdaGrad conveys the idea sufficiently well.

3.3.3 NEXT: Warmup + decay

...

3.3.4 Gradient accumulation + clipping

...

3.4 Hyperparameter Tuning

Hyperparameters are the tunable parts of an architecture which are not learned during training. Bad hyperparameters will yield training issues like divergence, overfitting or underfitting. A simple example of a hyperparameter I can think of is the learning rate in vanilla gradient descent. Some choices of learning rate will result in better generalisation than other choices. For a more sophisticated example, consider architecture-related hyperparameters, e.g. the number of hidden layers in a multi-layer perceptron.

3.4.1 Random search

Begin by designating some portion of the training data for validation (maybe 80% training, 20% validation).

Each hyperparameter may take on a set of values, i.e. the m hyperparameters $h_1, \ldots, h_m$ induce the samples spaces $\Omega_{h_1}, \ldots, \Omega_{h_m}$ and the grid of all possible hyperparameter configurations is thus $\Omega_1 \times \cdots \times \Omega_m$. In a perfect world, you'd be able to train a model for each hyperparameter configuration and choose whichever yields the lowest validation loss. We don't live in a perfect world and the cardinality of said grid is often too large. In line with this, instead of traversing it entirely, uniformly randomly sample from it a number of times and train with said hyperparameter configuration. Choose whichever configuration yielded the lowest validation loss.

3.4.2 $k-$fold cross-validation

For a better idea of how a configuration of hyperparameters will help generalise post-training, consider partitioning the training data into k disjoint subsets (or folds) $D_1, \ldots, D_k$. Then, for $i = 1, \ldots, k$, train on $D \backslash D_i$ and validate on D_i. Take the mean of the k validation losses. Choose whichever hyperparameter configuration yields the lowest mean validation loss.

3.4.3 Honourable mention: benchmark overfitting

Suppose research groups A and B independently publish results on nearly identical single hidden layer MLPs for the same task. The only difference is the number of neurons in the hidden layer: 99 for group A and 100 for

group B. Their training setup is otherwise identical: same data, same splits, same optimisation. Group A reports a slightly lower benchmark test error, and so a company independent of either group adopts the architecture of group A. How is this fundamentally different from a single group training both 99- and 100-neuron models and then choosing the one with the lower test loss?

It isn't: once the test set influences which model gets selected, the reported test error for the selected model becomes an optimistically biased estimate of generalisation. Repeated across papers, reviews, and adoption, the field effectively uses the benchmark as a selection signal. That is, an implicit, distributed form of tuning on the test set. This is referred to as benchmark overfitting and it motivates strict separation of validation and test as well as the evolution (or refreshing) of benchmarks over time.

4 Neural Network Architectures

Perhaps the simplest class of a neural network architecture, beyond a single-layer perceptron, is a multi-layer perceptron (MLP). Rigorously formulating MLPs is one of those things that's useful to do once and never again; consistent notation is 90% of the effort.

4.1 Multi-Layer Perceptrons (MLPs)

Multi-layer perceptrons (MLPs) are fully-connected feed-forward networks consisting of an input layer (where data is input), hidden layers and an output layer. An MLP with three hidden layers is illustrated in Figure 16. Each layer is made up of a number of neurons, each of which has a real-valued activation value. For any neuron in a non-input layer, its activation value is the output of a non-linear activation function given, as input, the neuron's real-valued bias plus a weighted sum of the activation values of all neurons in its preceding layer. Their intended use is as arbitrary function approximators. The output of the MLP, i.e. the output of one's approximation of the relevant underlying function, is given by the activation values of the neurons in the output layer. Regarding their inspiration from the human brain, consider this footnote[2].

The application of MLPs to learning functions from data is due to their expressivity, expressive-efficiency and tractability. Their expressivity is known due to a proof of their universal function approximation by Cybenko.

[2] `https://stats.stackexchange.com/a/159172`

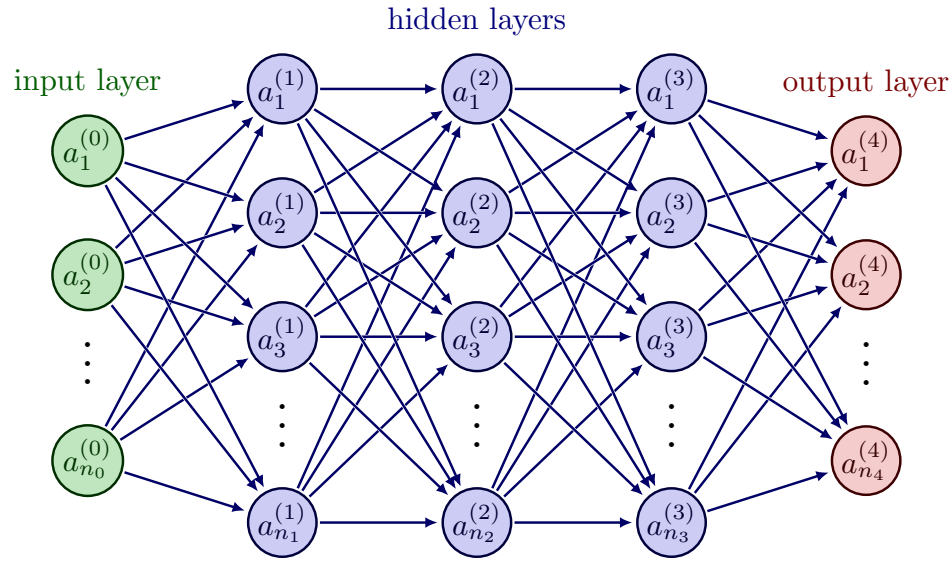


Figure 16: A multi-layer perceptron (MLP) with $k = 3$ hidden layers.

To add to this, their high expressive-efficiency has been demonstrated empirically and ensures that the number of model components (hidden layers and neurons) needed to represent arbitrary functions is far lower than competing model classes. As for their tractability, evaluating an MLP given input node values has complexity quadratic in the number of neurons of each layer of the MLP (after the heavy simplifcation of assuming a roughly equal number of neurons in each layer).

For brevity of notation, given $m, n \in \mathbb{N}$ with $m \leq n$ let $[n]_m = \{m, \ldots, n\}$ and let $[n] = [n]_1$. For further ease of notation, given an MLP with k hidden layers, denote the index of the input layer by 0, the output layer by $k+1$ and, by extension, the j^{th} layer by $j \in [k+1]_0$. Additionally, consider the following denotations

- $n_j \in \mathbb{Z}_{\geq 1}$ is the number of neurons in layer j.
- $a_i^{(j)} \in \mathbb{R}$ is the activation value of neuron i in layer j.
- $w_{i,l}^{(j)} \in \mathbb{R}$ is the weight associated with the edge to neuron i in layer $j \in [k+1]$ from neuron l in layer $j-1$.
- $b_i^{(j)} \in \mathbb{R}$ is the bias of neuron i in layer $j \in [k+1]$.
- $\sigma_j : \mathbb{R} \to \mathbb{R}$ is the non-linear activation function of layer $j \in [k+1]$.

From here we can express the activation value of any neuron in a non-input layer in terms of the activation values of the neurons in the layer which precedes it as

$$\begin{aligned} a_i^{(j+1)} &= \sigma_{j+1}\left(\sum_{l=1}^{n_j} w_{i,l}^{(j+1)} a_l^{(j)} + b_i^{(j+1)}\right) \\ &= \sigma_{j+1}\left(\begin{bmatrix} w_{i,1}^{(j+1)} \dots w_{i,n_j}^{(j+1)} \end{bmatrix} \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{n_j}^{(j)} \end{bmatrix} + b_i^{(j+1)}\right) \end{aligned}$$

for $j \in [k]_0$. The reason for writing the second equality above, involving the dot product of two vectors, is that it helps us to see how using matrix-vector notation allows us to write an elegant and compact expression for the activation values of all neurons in a non-input layer in terms of the activation values of the neurons belonging to its preceding layer as

$$\begin{bmatrix} a_1^{(j+1)} \\ \vdots \\ a_{n_{j+1}}^{(j+1)} \end{bmatrix} = \sigma_{j+1}\left(\begin{bmatrix} w_{1,1}^{(j+1)} & \dots & w_{1,n_j}^{(j+1)} \\ \vdots & \ddots & \vdots \\ w_{n_{j+1},1}^{(j+1)} & \dots & w_{n_{j+1},n_j}^{(j+1)} \end{bmatrix} \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{n_j}^{(j)} \end{bmatrix} + \begin{bmatrix} b_1^{(j+1)} \\ \vdots \\ b_{n_{j+1}}^{(j+1)} \end{bmatrix}\right)$$

which we abbreviate to

$$\mathbf{a}^{(j+1)} = \sigma_{j+1}\left(\mathbf{W}^{(j+1)}\mathbf{a}^{(j)} + \mathbf{b}^{(j+1)}\right)$$

where the activation function σ_{j+1} is applied element-wise.

If we fix the structure and choice of activation functions of an MLP then all that is left to learn are its weights and biases. This is often done using gradient descent to minimise some loss function in which gradients are computed via back-propagation. Such a loss function can be a principled measure, e.g. corresponding to maximum likelihood, but this is not always necessary: ad-hoc loss functions are sometimes employed.

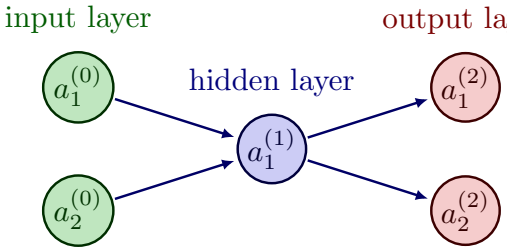


Figure 17: A neural network with one hidden layer ($k = 1$).

Crudely advocating for depth over width

The complexity of a forward pass of an MLP can be expressed in terms of the number of neurons in each layer as

$$\mathcal{O}\left(\sum_{j=1}^{k+1} n_{j-1} \cdot n_j\right).$$

Very crudely supposing $n_0 = \cdots = n_{k+1} = n$ yields a complexity of $\mathcal{O}((k+1)n^2)$ which grows linearly in the number of layers and quadratically in the number of neurons per layer.

Example 4.1 *Consider the neural network in Figure 17 whose input, hidden and output layers consist of two, one and two neurons respectively. With such a simple neural network, we may explicitly express the output neurons* $a_1^{(2)}$ *and* $a_2^{(2)}$ *in terms of the input neurons* $a_1^{(0)}$ *and* $a_2^{(0)}$*. We see that* $n_0 = 2$*,* $n_1 = 1$ *and* $n_2 = 2$ *and so* $\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \end{bmatrix}$ *and* $\mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} \\ w_{2,1}^{(2)} \end{bmatrix}$*. Noting additionally that* $\mathbf{b}^{(1)} = b_1^{(1)}$ *and* $\mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$ *we can explicitly express the activation of the single neuron in the hidden layer as*

$$\begin{aligned} \mathbf{a}^{(1)} &= \sigma_1\left(\mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}\right) \\ &= \sigma_1\left(w_{1,1}^{(1)}a_1^{(0)} + w_{1,2}^{(1)}a_2^{(0)} + b_1^{(1)}\right) \end{aligned}$$

which is just the scalar value $a_1^{(1)}$*. For the output layer, we have*

$$\begin{aligned}
\mathbf{a}^{(2)} &= \sigma_2\left(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\right) \\
&= \sigma_2\left(\begin{bmatrix} w_{1,1}^{(2)} a_1^{(1)} + b_1^{(2)} \\ w_{2,1}^{(2)} a_1^{(1)} + b_2^{(2)} \end{bmatrix}\right) \\
&= \sigma_2\left(\begin{bmatrix} w_{1,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_1^{(2)} \\ w_{2,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_2^{(2)} \end{bmatrix}\right) \\
&= \begin{bmatrix} \sigma_2\left(w_{1,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_1^{(2)}\right) \\ \sigma_2\left(w_{2,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_2^{(2)}\right) \end{bmatrix}.
\end{aligned}$$

As such, the activations of the output neurons are given by

$$\begin{aligned}
a_1^{(2)} &= \sigma_2\left(w_{1,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_1^{(2)}\right) \\
a_2^{(2)} &= \sigma_2\left(w_{2,1}^{(2)} \sigma_1\left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)}\right) + b_2^{(2)}\right).
\end{aligned}$$

Further crudely advocating for depth over width

A perhaps more important result is how, geometrically, MLPs split the input space into a bunch of regions. It turns out that the number of regions by which a single-hidden layer MLP with two input neurons and n neurons in its hidden layer splits the space is bounded above by $\frac{1}{2}(n^2+n+2)$ which is quadratic in n. For an MLP with k hidden layers of n neurons each, the number of regions by which the space is split is bounded above by

$$\frac{1}{2}(n^2+n+2)\left(\frac{n}{n_0}+1\right)^{n_0(k-1)}$$

which is exponential in k. Again, we argue for increasing depth k over width n. A natural quesiton is to what extent these bounds are tight in practice.

4.1.1 Universal Function Approximation with MLPs

A class of functions, or function class, $\mathcal{F}_S$ is capable of universal function approximation on $S \subset \mathbb{R}^m$ compact (closed and bounded) if for all continuous

$f : S \to \mathbb{R}$ and $\epsilon > 0$ there exists some $\hat{f} \in \mathcal{F}_S$ such that

$$\max_{x \in S} \left| f(x) - \hat{f}(x) \right| < \epsilon.$$

Note that such a function family can be used to approximate functions whose codomain is $\mathbb{R}^m$ by performing coordinate-wise approximation.

The family of polynomials is capable of universal function approximation as shown by Weierstrass in 1885. A directly equivalent statement to a function class satisfying the universal function approximation is the function ckass being dense in $C(S)$ with respect to the supremum norm. Here, $C(S)$ denotes the set of all continuous real-valued functions with compact domain $S \subset \mathbb{R}^m$. Quick note: usually the domain of the to-be-approximated function is denoted by K but I denote the number of layers in an MLP using a k so I'd like to avoid the confusion by using S instead. Instead of offering a rigorous proof, I'll motivate the overall idea in three steps starting with a simpler case: functions of the form $f : S \to \mathbb{R}$ where $S \subset \mathbb{N}$ with $\#S < \infty$.

Step 1: Motivating the idea for S finite

Without loss of generality, let $S = [\#S]$. We seek to show that for all $\epsilon > 0$ there exists an MLP with one hidden layer and one node in its output layer, whose output is denoted by $\hat{f}(x)$, which satisfies

$$\max_{x \in [\#S]} \left| f(x) - \hat{f}(x) \right| < \epsilon.$$

First note that for all $x \in [\#S]$,

$$\begin{aligned} f(x) &= \sum_{s=1}^{\#S} f(x) \mathbb{1}(x = s) \\ &= \sum_{s=1}^{\#S} f(x)(\mathbb{1}(x \geq s) - \mathbb{1}(x \geq s+1)) \end{aligned}$$

so we already see that a single neuron in the ouput layer of a single hidden layer MLP ($2 \cdot \#S$ neurons in hidden layer) with weights

$$\mathbf{W}^{(2)} = \begin{bmatrix} f(1) & -f(1) & \dots & f(\#S) & -f(\#S) \end{bmatrix},$$

i.e. $w_{1,s}^{(2)} = (-1)^{s+1} f\left(\left\lfloor \frac{s+1}{2} \right\rfloor\right)$, and bias $\mathbf{b}^{(2)} = 0$ is sufficient if the $2 \cdot \#S$ neurons in the hidden layer approximate

$$\mathbb{1}(x \geq 1), \mathbb{1}(x \geq 2), \mathbb{1}(x \geq 2), \mathbb{1}(x \geq 3), \dots, \mathbb{1}(x \geq \#S), \mathbb{1}(x \geq \#S + 1)$$

with the absolute error of each approximation bounded by ϵ. Note that the final indicator $\mathbb{1}(x \geq \#S + 1)$ is redundant so you only really need $2 \cdot \#S - 1$ neurons but I'll keep it for the nicer number. Also note that these are just renditions of the Heaviside function but I prefer sticking with the indicator function notation. This exact idea can be realised elegantly using the sigmoid as activation for the hidden layer. To see this, note that

$$\begin{aligned}\sigma\left(10^{\#S}(x-n)+10^{\#S-9}\right) &= \frac{1}{1+\exp\left(-10^{\#S}(x-n)+10^{\#S-9}\right)} \\ &= \frac{1}{1+\exp\left(n-x+10^{-9}\right)^{10^{\#S}}}\end{aligned}$$

approximates the Heaviside function $H(x-n)$ with some small leakage. To hit certain levels of precision, i.e. attaining the desired ϵ bound, simply decrease the exponent in which -9 currently lies. You can probably express the exponent in terms of ϵ explicitly (maybe ϵ^{-1}) but the idea is the point here. So what we want is for the $\left\lceil \frac{s+1}{2} \right\rceil^{\text{th}}$ neuron in the hidden layer to approximate $H\left(x - \left\lceil \frac{s+1}{2} \right\rceil\right)$ for $s \in [\#S]$. This is straightforward using a sigmoid activation function in the hidden layer given the approximation above. Simply set the weights and biases accordingly: $w_{s,1}^{(1)} = 10^{\#S}$ and $b_s^{(1)} = -10^{\#S}\left(\left\lceil \frac{s+1}{2} \right\rceil - 10^{-9}\right)$ and we're done.

Step 2: Extending the idea to $S \subset \mathbb{R}$ compact

If $f \in C(S)$ for $S \subset \mathbb{R}$ compact then f is uniformly continuous and so for all $\epsilon > 0$ there exists $\delta > 0$ such that

$$|x - y| < \delta \implies |f(x) - f(y)| < \epsilon.$$

Note that if we have a Lipschitz constant L for a given f then $\delta = \epsilon / L$ suffices. The reason this $\epsilon - \delta$ statement is useful is that if we split S into N subintervals whose width is bounded by δ then for all c, x within the same subinterval we have

$$|f(x) - f(c)| < \epsilon.$$

To realise this idea, if $S = [a, b]$ then split S according to

$$a = x_0 < x_1 < \cdots < x_N = b$$

such that $x_{i+1} - x_i < \delta$ and take $c = (x_i + x_{i+1})/2$, the midpoint of $[x_i, x_{i+1}]$. The number of subintervals N needed depends on a, b and δ, e.g. $N = \left\lceil \frac{b-a}{\delta/2} \right\rceil$

works but the denominator just needs to be less than δ so $\delta/1.1$ would give a tighter N. From this we see that

$$N \approx \frac{b-a}{\delta} = \frac{L(b-a)}{\epsilon}.$$

The higher we take N the more accurate of an approximation we get. This idea of splitting S into N intervals is nice because it requires us to approximate only one point $c_i = (x_i + x_{i+1})/2$ on each subinterval, i.e. we are back to the simpler case which started our motivation of approximating at a fixed number of points.

With this idea in mind, the number of required neurons to realise our idea

$$2N = 2\frac{L(b-a)}{\epsilon}$$

grows linearly in L, the interval length $b-a$ and the reciprocal of the desired level of precision $1/\epsilon$. In practice, you only have control of the desired level of precision ϵ.

Step 3: Extending the idea to $S \subset \mathbb{R}^m$ compact

The idea is really not very different. Imagine $S \subset \mathbb{R}^2$ compact. The compactness is very hepful for visualisation. As long as we can create indicator-approximate functions on separate regions of S then use exactly the idea described above: split S into separate regions and approximate f on a single point in each. To see that we can approximate the indicator on regions of $S \subset \mathbb{R}^2$ simply see that we can create wall-like objects using the same idea of creating a 'steep' plane and sigmoiding it. As such, to isolate some region, produce three walls which enclose a region and you're done. If you want a more accurate enclosing of the region then use more walls. Without boundedness, which is given by compactness, this idea wouldn't work. Extend this idea to $\mathbb{R}^m$ and voilà.

Is the idea used here practical?

What the idea conveyed shows is *that* (and even *how*) an MLP can approximate any function in $C(S)$ for $S \subset \mathbb{R}^m$ compact. It is not at all an indicator of how MLPs learned via gradient descent, in practice, go about approximating functions from data.

An interesting addition to this note is how this formulation gives an idea of how the weights from the hidden layer to the output explicitly encode the evaluation of the to-be-approximated function. Again, this is not what

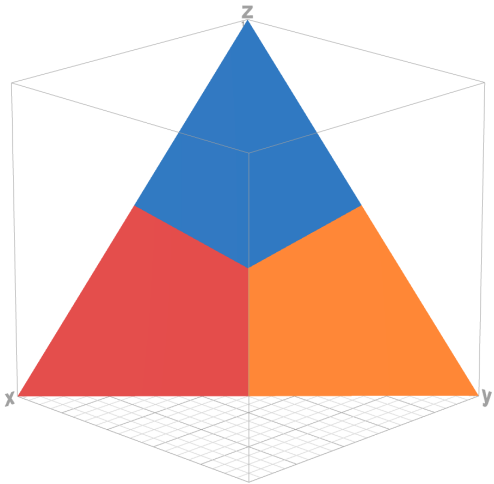


Figure 18: The 2−probability simplex coloured according to argmax in the three coordinates.

we'd expect MLPs to do in practice, instead they find a much more efficient representation, sharing information between neurons. Reminds me a lot of the research 3Blue1Brown discussed in one of his videos about transfromers and LLMs in which we spoke about how the LLM 'knows' to assign a high probability to the token `Jordan` when given `My favourite basketballer is Michael` as input. How does it 'know' to do that? It isn't at all implied by the grammar/semantics/whatever of the sentence. Said research concluded things about such pieces of knowledge being stored in the weights of the MLPs of the transformer blocks.

4.1.2 Posing classification as a regression problem

Which space does one transform their samples into in order to linearly separate them? In the case of neural net approaches, for $K-$classification, it's the $(K-1)$-probability simplex given by

$$\left\{(x_1, \ldots, x_K) \in [0,1]^K | x_1 + \cdots + x_K = 1\right\}.$$

The 2−probability simpelx coloured according to argmax is illustrated in Figure 18. From the figure, it is clear how argmax can be computed by splitting the 2-probability simplex via two planes, e.g. the intersection of $x > y$ and $x > z$ on the simplex yield the red region. As such, neural nets used for classification can be seen as performing regression on the 2−probability simplex.

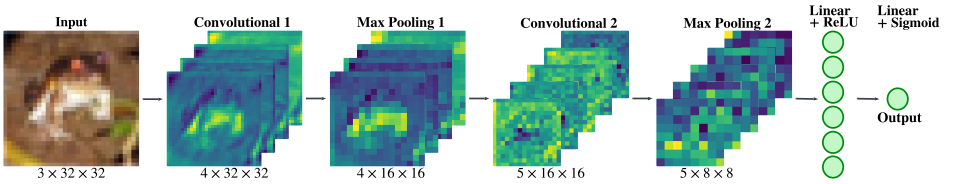


Figure 19: A binary classification CNN processing an image of a frog.

4.2 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) were initially designed to process image and, by extension, video data. Over time, they have been applied to processing far richer forms of data. A forward pass through a CNN designed for image processing is illustrated in Figure 19, in which we see that their architecture can be described on a high level as consisting of two parts. The first part performs feature extraction through convolution and pooling operations. The second part employs an MLP to produce the overall model's output given the earlier-extracted features, e.g. the probability that an input image is of a given class.

Before deep learning architectures became mainstream, researchers used handcrafted features (SIFT, HOG, SURF, etc.) for feature extraction and fed extracted features to a given model, e.g. an SVM for classification. That is, feature extraction wasn't learned, nor was it entwined with the inference component of their models. For image processing tasks, humans turn out to be far worse than deep learning architectures at knowing which features are most informative of what is being predicted. Hence, CNNs are now the go-to architecture for image and video processing. There's something to be said here about vision transformers and the extent to which they are competitive with CNNs but this can wait. Relevant interactive visualisation: `adamharley.com/nn_vis/cnn/3d.html`.

4.2.1 Convolutions

Convolution operations involve using a (typically square) matrix to, in some sense, summarise the information embedded in parts of the object over which it traverses. This matrix is referred to as a filter or a kernel — I prefer using filter as the term kernel has so many meanings elsewhere in maths. You can think of convolution operations as a sort of dot product between the given filter and the grid of pixel values over which lies during its traversal. For a simple illustration of this idea, consider Figure 20. Since CNNs consists

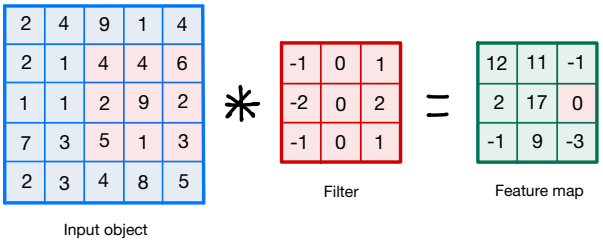


Figure 20: Convolution with a vertical edge detection filter with 0 bias.

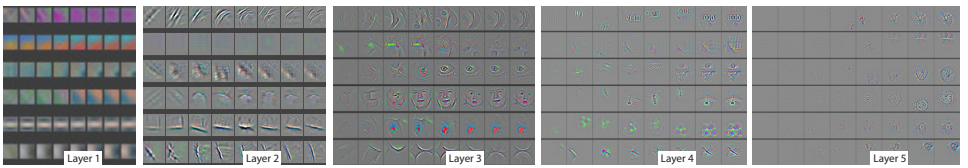


Figure 21: The level of abstraction of a CNN's learned filters. Earlier layers pertain to low-level features like edges or textures. Later layers pertain to higher-level features like faces or wheels.

of multiple convolution layers, the output of a convolution layer is often referred to as a feature map.

More formally, given a feature map $I \in \mathbb{R}^{C \times W \times H}$ and a filter $K \in \mathbb{R}^{C \times k \times k}$ with $k < W$ and $k < H$, the feature map $F = K * I \in \mathbb{R}^{W' \times H'}$ is the convolution of K over I is as in

$$F_{i,j} = \sum_{c=1}^{C} \sum_{x=1}^{k} \sum_{y=1}^{k} K_{c,x,y} \cdot I_{c,x+i,y+j} + b_c$$

where b_c is the filter's bias in channel c. Nowadays, $k = 3$ is a very popular choice. Note that this notation is intended to illustrate what a convolution looks like algebraically. As it stands, what's written above does not account for padding, stride, etc.

Generally, earlier filters in a CNN architecture extract lower-level features, like edges and textures, while later convolutions extract higher-level features like entire components of object, e.g. wheels. This is illustrated in Figure 21.

Post-convolution spatial dimensions

Given a feature map $F \in \mathbb{R}^{W \times H}$, suppose we compute the convolution $O \in \mathbb{R}^{W' \times H'}$ of the filter $K \in \mathbb{R}^{k \times k}$ over F with uniform padding P and stride S. The precise dimensions W' and H' of the new feature map O are

given by

$$W' = \left\lfloor \frac{W + 2P - k}{S} \right\rfloor + 1$$

and

$$H' = \left\lfloor \frac{H + 2P - k}{S} \right\rfloor + 1.$$

It's not too tricky to derive these. It's the kinda thing you do once and never again.

4.2.2 Pooling

Pooling operations reduce the spatial dimensions of a given feature map by applying relatively straightforward transformations: max pooling, min pooling, mean pooling, etc. There's not much point to expressing them rigorously. Instead, consider Figure 22. Benefits of pooling are improved translation-invariance and reduced required compute.

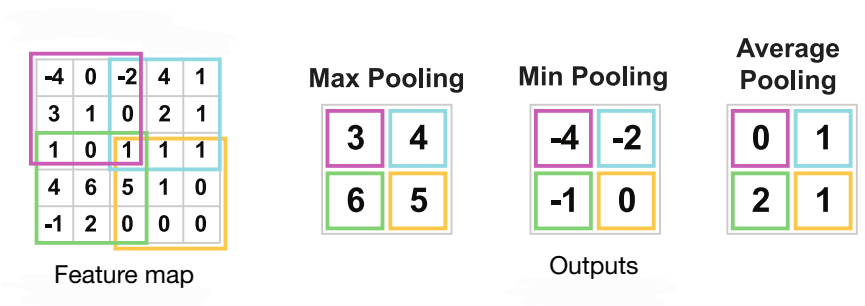


Figure 22: Common flavours of pooling.

Why use a CNN over an MLP?

Massively improved parameter-efficiency, translation invariance and learning efficiency[a] (fewer samples needed to learn distributions to similar degrees of precision). These are in part due to the inductive bias present of their architecture.

[a] ai.stackexchange.com/questions/27407

4.3 TODO: Recurrent Neural Networks (RNNs)

Schmidhuber!

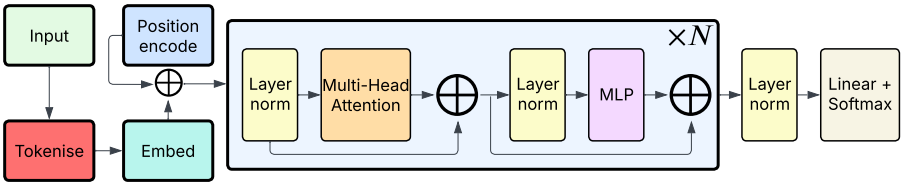


Figure 23: A decoder-only language transformer consisting of N transformer blocks.

4.4 NEXT: Transformers

Transformers are a relatively modern architecture but their applications are already broad. They're now found in language models (e.g. GPTs), image models (vision transformers), denoising models (diffusion) and so on. They offer many advantages over earlier competing architectures, e.g. RNNs, due to not being bottlenecked by input sequence length.

The journey of a given input sequence through a transformer looks something like

$$\texttt{tokenise} \to \texttt{embed} \to \texttt{encode position} \to \texttt{trf blocks} \to \texttt{output proj}$$

as illustrated in Figure 23. Each step is detailed in this subsection.

4.4.1 Tokenisation

Tokens can be thought of as the units of information of a data structure, e.g. sequences of text. The tokenisation of a sequence is simply the process of slicing it up into its token make-up. Think of splitting the sentence 'The chef cooked a meal for the critic.' according to the following colouring (or tokenisation)

or splitting up an image as in Figure 24. To play around with one of OpenAI's text tokenisers, consider `platform.openai.com/tokenizer`.

The purpose of tokenisation is self-explanatory: we'd like an idea of dependence between different parts of the input as to aid our output prediction. As such, being able to break a sequence into chunks, as opposed to processing it in its entirety, is desirable. Notationally, let V denote the set of all tokens in our model's vocabulary and $T : \Sigma^* \to \{1, \dots, |V|\}^*$ the

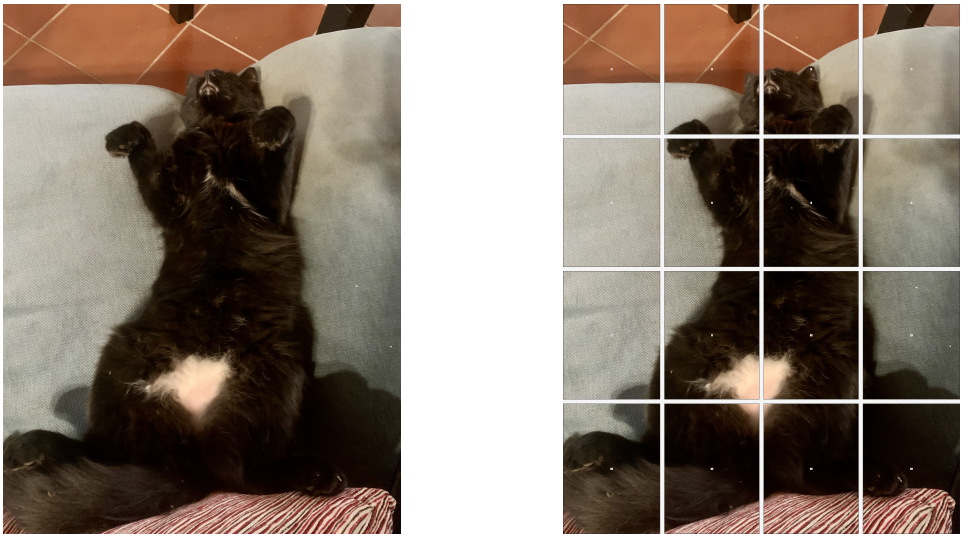


Figure 24: The tokenisation (or patchification) of a sleeping Augusto.

tokenisation function where S^* is the set of all finite strings whose characters are in S.

For text data, the tokeniser T is often learned according to byte pair encoding (BPE). Roughly put, start with a vocabulary V consisting of the first 256 UTF-8 encoding characters. Then, until the vocabulary size reaches some threshold, say $50,000$, repeat the following for $i = 0, \ldots, V_{\max} - 1$:

1. Tokenise the text according to the tokeniser
2. Compute the most frequent token pair in the corpus and append it to the tokensier with index $256 + i$.

The result is an ordered list of merge rules. For example, the most frequent character pair in English text

4.4.2 Embedding

A linear embedding map $L_E : \{e_1, \ldots, e_{|V|}\} \to \mathbb{R}^d$, where $d \in \mathbb{N}$ is the embedding dimension and e_i denotes the i^{th} standard basis vector, offers high dimensional numeric representations of tokens belonging to the vocabulary V. What makes token embedding functions interesting is how well they capture underlying semantics, e.g. of natural language. Such an embedding map learned on pieces of text, like Word2Vec, might satisfy

$$L_E(\text{“Germany”}) + L_E(\text{“fascist”}) - L_E(\text{“Mussolini”}) \approx L_E(\text{“Hitler”})$$

or

$$L_E(\text{“Wales”}) + L_E(\text{“Cardiff”}) - L_E(\text{“Italy”}) \approx L_E(\text{“Rome”})$$

where “word” denotes the basis vector corresponding to the word according to some indexing over the vocabulary, e.g. perhaps $|V| = 50,000$ and “Germany”$= e_{35,629}$. The prupose of taking the domain of L_E to be this set of standard basis vectors is that then it can be seen as a look-up table of the rows of $E \in \mathbb{R}^{d\times|V|}$.

So how is the embedding matrix $E \in \mathbb{R}^{d\times|V|}$ learned? In the context of embedding maps like Word2Vec, where the embedding map is learned in isolation (not in conjunction with a larger architecture), it is learned by repeatedly sampling a center word from the dataset (a large corpus of text in our example), a context word which lies near the center word in the text and l negative-context words which do not lie near the center word. The loss function used to learn the parameters (the elements of our linear embedding map) intends to reward the alignment of the context word and the center word while punishing the alignment of the center word and the l negative-context words. In maths notation, the loss looks something like

$$-\sigma(w_{\text{cent}}, w_{\text{cont}}) + \sum_{i=1}^{l} \sigma\left(w_{\text{cent}}, w_{\text{neg}}^{(i)}\right)$$

where σ denotes the sigmoid function.

Embedding maps are less interesting when learned in conjunction with a larger architecture, e.g. GPT architectures, as they are learned according to backpropagation like any other parameter. If the loss is not such that alignments of token embeddings in $\mathbb{R}^d$ mean anything, some of the nice properties discussed earlier are less likely to be observed.

4.4.3 Positional encoding/embedding

The position of tokens in a given sequence of words is critical to understanding the intended meaning of the sequence. For example, the highlighted sequence above regarding the chef and the critic is distinct from

The critic cooked a chef for his meal .

In the latter sequence, the intended meaning of cooked is akin to ‘criticised’, while in the former it is simply meant to convey that the chef literally cooked. So how do transformers go about accounting for the positions of tokens in given input sequences?

Unlike RNNs, transformers have no inherent notion of the order of tokens in a given input without this additional positional encoding step. That is because the attention and MLP components are, standalone, independent of position. Due to this, sprinkling in position-related information to the embedded input is necessary. I like to think of positional encoding as a soft prior pertaining to which tokens are relevant to others. That is, the prior is relevant but contextual information baked into the text itself will override its inclusion if sufficiently relevant.

Let $\mathbf{x} = (x_1, \ldots, x_n) \in \{e_1, \ldots, e_{|V|}\}^n$ denote a sequence of n one-hot encoded tokens (so the post-tokenisation input) and

$$E(\mathbf{x}) = \begin{bmatrix} L_E(x_1) \\ \vdots \\ L_E(x_n) \end{bmatrix} \in \mathbb{R}^{n \times d}$$

their embedding. Given an appropriate positional embedding matrix $P(\mathbf{x}) \in \mathbb{R}^{n \times d}$ many set-ups form the input to the first transformer block by computing its sum with the embedding matrix $E(\mathbf{x})$. That is, they compute

$$X = E(\mathbf{x}) + P(\mathbf{x}) \in \mathbb{R}^{n \times d}$$

and feed it to the first transformer block.

Before detailing which positional embedding function P was used in the paper that made transformers famous, it's worth attending to the validity of arguably natural approaches and the ways in which they fall short:

- **Why not simply add the index of the token to its embedding?** For long sequences, the index value of later tokens would overpower the value of the presence of the token itself. In line with this, we'd like the positional encoding to be bounded in magnitude.

- **Why not add the index divided by the sequence length to ensure getting something in [0,1]?** For sequences of varying length, we'd like the positional information for a given index to be the same. So attributing 0.5 to the third token in a sequence of five would be an issue as it would also be attributed to the 11th token in a sequence of 20.

- **Why not encode via the binary of the index?** Change in positional encoding should be smooth. We want $||P_{k+1,:} - P_{k,:}||$ to be bounded but for binary encodings it is simply equal to the square root of the number of bit flips: unbounded in k.

To address both issues, the authors of the now famous Attention Is All you Need proposed alternating sinusoids[3]. Sinusoidal positional encoding feels unnatural to me, though it has a nice motivation as a continuous analog of binary encoding. Regardless, far more intuitive methods of positional encoding exist such as learned positional encoding (used by OpenAI for GPT-2) or the now-popular rotary positional embeddings (RoPE).

Learned positional encoding is just as the name suggests: the positional embedding map $P \in \mathbb{R}^{n \times d}$ is set up as a matrix of parameters which are learned according via gradient descent in conjunction with all other model parameters. RoPE is bit more involved than learned positional encoding as it's applied during the attention mechanism and not between embedding and feeding the input to the first transformer block. Despite this complication, RoPE is quite elegant, relying on the linearity of rotation transformations in $\mathbb{R}^d$. I'll detail them here at some point but, for now, knowing just that positional embedding is a core component of transformer architectures is sufficient.

4.4.4 NEXT: Attention + MLP component

...

4.4.5 Example: GPT-2 small (∼123M parameters)

GPT-2 small acts as a great introduction to how a decoder-only language transformer may look and the distribution of parameter counts over different components in their architecture. Its architecture is illustrated in Figure 23 and its hyperparameters are given in Table 1 From the hyperparameters

Hyperparameter	Value
$\lvert V \rvert$	50,256
n_{heads}	12
n_{blocks}	12
d_{model}	768
d_{ff}	3072

Table 1: Hyperparameters for GPT-2-small.

and the GPT-2-small's architecture, we can compute the total number of parameters of the model as in Table 2.

[3]Great video about it: `www.youtube.com/watch?v=dWkm4nFikgM`.

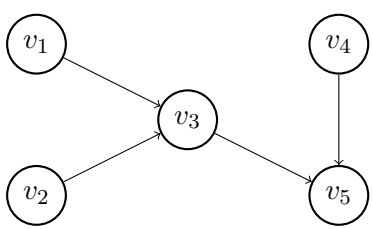
Parameter	Count (in millions)
E	$768 \cdot 50{,}256 \approx 38.6$
P	$1024 \cdot 768 \approx 0.8$
W_Q	$12 \cdot 768 \cdot 768 \approx 7.1$
W_K	$12 \cdot 768 \cdot 768 \approx 7.1$
W_V	$12 \cdot 768 \cdot 768 \approx 7.1$
W_O	$12 \cdot 768 \cdot 768 \approx 7.1$
W_1	$12 \cdot 768 \cdot 3072 \approx 28.3$
W_2	$12 \cdot 768 \cdot 3072 \approx 28.3$
Total	~ 124.4

Table 2: Parameter count for GPT-2-small.

4.5 Backpropagation

So we have all of these fancy neural network architectures, which can be seen loosely as arbitrary function approximators with powerful inductive biases, but how might one compute their gradient in order to undergo gradient descent? Backpropagation (reverse-mode auto-differentiation) is a method of computing the gradient of a scalar function represented as a computational directed acyclic graph (DAG). The motivation of obtaining gradients in this fashion in deep learning contexts is clear: for finding suitable parameters via gradient descent. It relies heavily on the chain rule of calculus.

The following explanation will distinguish between partial derivatives and total derivatives. The total derivative is intended to account for the dependence arguments of a function may have with respect to a variable while partial derivatives are such that all arguments are taken as constant except for the one with which we are computing the derivative. To illustrate the point, consider the following computational DAG representing the function $L(\theta_1, \theta_2; x_1, x_2, x_3) = \theta_2 + x_3 + \theta_1 x_1 x_2$



with $v_1 = x_1$, $v_2 = x_2$, $v_4 = x_3$, $v_3 = \theta_1 v_1 v_2$ and $v_5 = \theta_2 + v_3 + v_4$. In this representation, each node is a primitive function of its parents and the

parameters. As such, we have $\frac{\partial v_5}{\partial \theta_1} = 0$ but $\frac{dv_5}{d\theta_1} = v_1 v_2$. This distinction between partial and total derivatives is important for our formulation of reverse-mode auto-differentiation, i.e. backpropagation.

4.5.1 Reverse-mode auto-differentiation (backprop)

First, suppose the loss function we wish to compute gradients with is represented as a computational DAG whose nodes $v_1, \ldots, v_N$ are topologically ordered. That is, $v_N = L(\theta_1, \ldots, \theta_m)$ and we wish to compute

$$\nabla_\theta v_N = \left(\frac{d}{d\theta_1} v_N, \ldots, \frac{d}{d\theta_m} v_N \right).$$

So how might we compute these individual total derivatives for a given configuration of the input nodes? If each node is represented in terms of its parent nodes and the parameters then applying the chain rule yields

$$\frac{d}{d\theta_i} v_N = \sum_{k=1}^{N} \frac{dv_N}{dv_k} \frac{\partial v_k}{\partial \theta_i}$$

with

$$\frac{d}{dv_k} v_N = \sum_{v_c \in \mathbf{Ch}(v_k)} \frac{dv_N}{dv_c} \frac{\partial v_c}{\partial v_k}$$

where $\mathbf{Ch}(v)$ denotes the set of children of the node v in the computational DAG. As a result, if $\partial v_c / \partial v_k$ and $\partial v_k / \partial \theta_i$ have closed forms for all $v_c \in \mathbf{Ch}(v_k)$ and $k \in \{1, \ldots, N\}$ then the desired total derivatives can be computed by simply computing dv_N / dv_k for $k = N-1, \ldots, 1$ (in said reversed order) and plugging them in to the equation above. This makes it immediately clear why it is named reverse-mode auto-differentation or backpropagation.

As an explicit example, consider the DAG with $v_1, \ldots, v_5$ above. After some algebra, we obtain

$$\frac{dv_5}{dv_4} = \frac{\partial v_5}{\partial v_4} \qquad \frac{dv_5}{dv_3} = \frac{\partial v_5}{\partial v_3} \qquad \frac{dv_5}{dv_2} = \frac{\partial v_5}{\partial v_3}\frac{\partial v_3}{\partial v_2} \qquad \frac{dv_5}{dv_1} = \frac{\partial v_5}{\partial v_3}\frac{\partial v_3}{\partial v_1}$$

and the only non-zero partial derivatives with respect to parameters are

$$\frac{\partial v_3}{\partial \theta_1} = v_1 v_2 \quad \text{and} \quad \frac{\partial v_5}{\partial \theta_2} = 1.$$

From these, we obtain closed forms of the desired total derivatives

$$\frac{dv_5}{d\theta_i} = \sum_{k=1}^{5} \frac{dv_5}{dv_k}\frac{\partial v_k}{\partial \theta_i} = \frac{\partial v_3}{\partial \theta_i} + \frac{\partial v_5}{\partial \theta_i}$$

which yields $dv_5/d\theta_1 = v_1 v_2$ and $dv_5/d\theta_2 = 1$. Noting that

$$v_5 = \theta_2 + v_4 + \theta_1 v_1 v_2,$$

we see that these total derivatives are correct. This example helps to illustrate the need for closed forms of the partial derivatives $\partial v_k/\partial \theta_i$ and $\partial v_c/\partial v_k$ for all $v_c \in \mathbf{Ch}(v_k)$ and $k \in \{1, \ldots, N\}$. Due to this, backpropagation relies on each node being a primitive function of its inputs and parameters.

A non-exhaustive list of such primitives often-used in deep learning is as follows:

- Linear layers
- Non-linear activation functions
- Normalisations
- Reduction operations, e.g. sum, mean, etc.
- Dropout
- Losses
- Convolutions/pooling
- Indexing and reshaping
- Attention-related operations

Computing closed forms of the relevant partial derivatives can be fun. There are simple ones, e.g. if $v_2 = \sigma(v_1)$ where σ is the sigmoid activation function then $\partial v_2/\partial v_1 = \sigma'(v_1)$ has a simple closed form. For an example of the parameters, consider a simple linear layer as in $v_2 = \theta_1 v_1 + \theta_2$ where θ_1 and θ_2 are scalar parameters. In this case, $\partial v_2/\partial \theta_1 = v_1$. Simple enough.

Why not forward-mode auto-differentiation?

Forward-mode auto-differentiation is a similar idea with the relevant derivatives computed from the input nodes to the output nodes. For $f : \mathbb{R}^n \to \mathbb{R}^m$, computing the full Jacobian using forward-mode auto-differentiation requires $O(n)$ memory and $O(n\#\text{DAG})$ time. For comparison, backpropagation requires $O(\#\text{DAG})$ memory and $O(m\#\text{DAG})$ time. It's immediately clear that in machine learning contexts, where n far exceeds $m = 1$, backpropagation is massively favourable if you have sufficient memory.

Other methods exist, e.g. numerical differentiation and symbolic differentiation. The former suffers for many reasons, like instability due to high-precision requirements, and the latter suffers due to the massive blow up of function representations.

4.5.2 Backpropagation for MLPs

As with formulating MLPs, it's good to rigorously derive backpropagation for them at least once[4]. It's sort of strange that I haven't done that yet (August 2025 is time of writing) and have just trusted that it holds up.

Recall that a loss function is of the form

$$\mathcal{L} : \Omega_Y \times \Omega_Y \to \mathbb{R}_{\geq 0}$$

and that the loss induced by a choice of model parameters θ for a sample $(\mathbf{x}, y)$, which we will henceforth crudely denote by $\mathcal{L}$, is given by

$$\mathcal{L}\left(y, f_\theta(\mathbf{x})\right),$$

where f_θ is the model itself. Note that taking the gradient of $\mathcal{L}(y, f_\theta(\mathbf{x}))$ with respect to θ is perfectly valid even though the function $\mathcal{L}$ itself is not a function of θ. For an MLP,

$$\theta = \left(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(k+1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(k+1)}\right).$$

Let $\mathbf{z}^{(j)} = \mathbf{W}^{(j)}\mathbf{a}^{(j-1)} + \mathbf{b}^{(j)}$ for $j \in [k+1]$ from which we have $\mathbf{a}^{(j)} = \sigma_j\left(\mathbf{z}^{(j)}\right)$. Further, let

$$\delta_i^{(j)} = \frac{\partial \mathcal{L}}{\partial z_i^{(j)}}$$

[4] I wrote the following before writing about reverse-mode auto-differentiation above and so the notation here is in line with the formulation of MLPs.

for $j \in [k+1]$ and $i \in [n_j]$ where $\left(z_1^{(j)}, \ldots, z_{n_j}^{(j)}\right) = \mathbf{z}^{(j)}$. We seek to derive four statements:

1. $\delta_i^{(k+1)} = \sigma'_{k+1}\left(z_i^{(k+1)}\right) \frac{\partial \mathcal{L}}{\partial a_i^{(k+1)}}$ for $i \in [k+1]$
2. $\delta_i^{(j)} = \sigma'_j\left(z_i^{(j)}\right)\left(\left(\mathbf{W}^{(j+1)}\right)^\top \delta^{(j+1)}\right)_i$ for $j \in [k]$ and $i \in [n_j]$
3. $\frac{\partial \mathcal{L}}{\partial b_i^{(j)}} = \delta_i^{(j)}$ for $j \in [k+1]$ and $i \in [n_j]$
4. $\frac{\partial \mathcal{L}}{\partial w_{l,i}^{(j)}} = \delta_l^{(j)} a_i^{(j-1)}$ for $j \in [k+1]$, $i \in [n_{j-1}]$ and $l \in [n_j]$

Before deriving each statement in order, we can see already what was meant earlier by propagating backwards through the MLP: to compute the gradients relevant to performing gradient descent, backpropagation requires the computation of $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(k+1)}$ from which $\delta^{(k+1)}$ can be computed. Note that the $\frac{\partial \mathcal{L}}{\partial a_i^{(k+1)}}$ term in the first statement is often easily computed, e.g. for MSE loss we have

$$\frac{\partial \mathcal{L}}{\partial a_i^{(k+1)}} = a_i^{(k+1)} - y_i.$$

From there, one can compute $\delta^{(k)}, \ldots, \delta^{(1)}$ in that order, effectively propagating through the MLP backwards. With $\delta^{(1)}, \ldots, \delta^{(k+1)}$ and $\mathbf{a}^{(0)}, \mathbf{a}^{(1)} = \sigma_1\left(\mathbf{z}^{(1)}\right), \ldots, \mathbf{a}^{(k+1)} = \sigma_{k+1}\left(\mathbf{z}^{(k+1)}\right)$, computing the relevant gradients is straightforward.

Statement 1. Using the chain rule, see that

$$\begin{aligned}
\delta_i^{(k+1)} &= \frac{\partial \mathcal{L}}{\partial z_i^{(k+1)}} \\
&= \frac{\partial \mathcal{L}}{\partial a_i^{(k+1)}} \frac{a_i^{(k+1)}}{\partial z_i^{(k+1)}} \\
&= \sigma'_{k+1}\left(z_i^{(k+1)}\right) \frac{\partial \mathcal{L}}{\partial a_i^{(k+1)}}.
\end{aligned}$$

Statement 2. First, note that

$$\left(\mathbf{W}^{(j+1)}\right)^{\top}\delta^{(j+1)} = \begin{bmatrix} w_{1,1}^{(j+1)} & \dots & w_{n_{j+1},1}^{(j+1)} \\ \vdots & \ddots & \vdots \\ w_{1,n_j}^{(j+1)} & \dots & w_{n_{j+1},n_j}^{(j+1)} \end{bmatrix} \begin{bmatrix} \delta_1^{(j+1)} \\ \vdots \\ \delta_{n_{j+1}}^{(j+1)} \end{bmatrix}$$
$$= \sum_{l=1}^{n_{j+1}} \delta_l^{(j+1)} \begin{bmatrix} w_{l,1}^{(j+1)} \\ \vdots \\ w_{l,n_j}^{(j+1)} \end{bmatrix}$$

and so

$$\left(\left(\mathbf{W}^{(j+1)}\right)^{\top}\delta^{(j+1)}\right)_i = \sum_{l=1}^{n_{j+1}} \delta_l^{(j+1)} w_{l,i}^{(j+1)}$$

which reduces the to-be-derived statement to

$$\delta_i^{(j)} = \sigma_j'(z_i^{(j)}) \sum_{l=1}^{n_{j+1}} \delta_l^{(j+1)} w_{l,i}^{(j+1)}.$$

Using the chain rule, we deduce that

$$\delta_i^{(j)} = \frac{\partial \mathcal{L}}{\partial z_i^{(j)}} = \sum_{l=1}^{n_{j+1}} \frac{\partial \mathcal{L}}{\partial z_l^{(j+1)}} \frac{\partial z_l^{(j+1)}}{\partial z_i^{(j)}} = \sum_{l=1}^{n_{j+1}} \delta_l^{(j+1)} \frac{\partial z_l^{(j+1)}}{\partial z_i^{(j)}}.$$

To complete our deduction, see that

$$z_l^{(j+1)} = \sum_{i=1}^{n_j} w_{l,i}^{(j+1)} \sigma_{j+1}\left(z_i^{(j)}\right) + b_l^{(j+1)}$$

from which we obtain

$$\frac{\partial z_l^{(j+1)}}{\partial z_i^{(j)}} = \sigma_{j+1}'(z_i^{(j)}) w_{l,i}^{(j+1)}$$

completing the deduction.

Statement 3. This one's a simple application of the chain rule, as in

$$\frac{\partial \mathcal{L}}{\partial b_i^{(j)}} = \sum_{l=1}^{n_{j+1}} \frac{\partial \mathcal{L}}{\partial z_l^{(j)}} \frac{\partial z_l^{(j)}}{\partial b_i^{(j)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(j)}} = \delta_i^{(j)}.$$

Statement 4. As before, see that

$$z_l^{(j)} = \sum_{i=1}^{n_j} w_{l,i}^{(j)} a_i^{(j-1)} + b_l^{(j)}$$

which yields

$$\frac{\partial z_l^{(j)}}{\partial w_{l,i}^{(j)}} = a_i^{(j-1)}.$$

To complete the derivation, consider

$$\frac{\partial \mathcal{L}}{\partial w_{l,i}^{(j)}} = \frac{\partial \mathcal{L}}{\partial z_l^{(j)}} \frac{\partial z_l^{(j)}}{\partial w_{l,i}^{(j)}} = \delta_l^{(j)} a_i^{(j-1)}.$$

4.6 Misc. questions

Q1: What is the purpose of activation functions?

Without the application of at least one activation function, the values of the output neurons would simply be the result of matrix-vector multiplication and vector addition. That is to say, the output of the neural network, without an any activation function, would be a purely linear transformation of the input. The issue with this is that most problems require some degree of non-linearity. The staple example of this is the classification of two-dimensional samples which are not linearly separable. An example is illustrated in the left of Figure 25.

The point is that there is no line that would separate the classes in either case: non-linearity is needed! In the latter case, it is desirable to be able to somehow punch the interior of the unit disc to form a blue mountain with the surrounding ground covered in red. This can be achieved by the non-linear transformation $\phi(x, y) = \max(0, 1-(x^2+y^2))$ which is illustrated in the right of Figure 25.

From here, a natural linear decision boundary is the plane $z = 0$ (i.e. the red ground surrounding the blue mountain). Any sample above the plane, i.e. any sample whose $z-$coordinate is positive, is classified as blue. It is otherwise classified as red. We can be more clever than this though. This transformation required us to map samples to 3D — how about mapping to just 1D? To do this, a complete decision function is given by

$$D(x, y) = \left\lceil \frac{\lfloor x^2 + y^2 \rfloor}{x^2 + y^2} \right\rceil$$

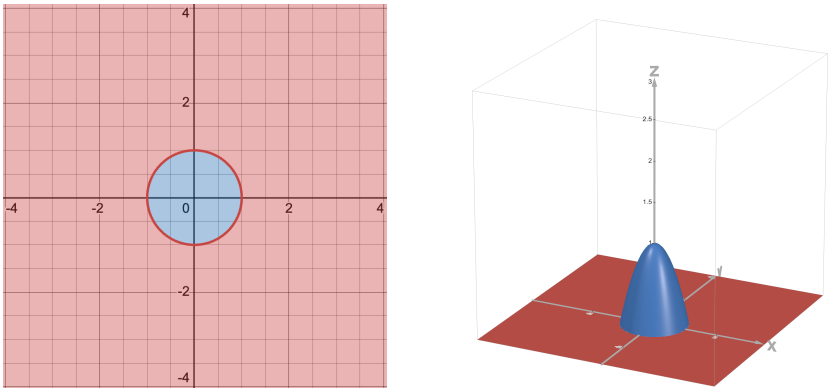


Figure 25: Non-linearly separable data (left) and its image under a non-linear transformation (right).

and define $D(0,0) = 0$ to account for the removable singularity.

Q2: Why not just one activation function in the output layer?

If there were a single activation function towards the end of the architecture then everything that came before would be a series of linear transformations of the input. The composition of linear transformations is just a linear transformation. As such, said model would be equivalent to a single layer perceptron, i.e. linear transformation + non-linear activation function, which would yield linear decision boundaries only.

To be super pedantic, single-layer perceptrons yielding only linear decision boundaries is the case if the activation function used is monotonic, which is often the case. If you allow for weird, non-monotonic activations then this linear decision boundary result doesn't necessary hold but, again, this is a pretty pedantic detail.

Q3: Must activation functions be monotonic?

Nope, it just makes things related to optimisation easier while having a sufficiently-small effect on performance in practice. Some results surrounding neural networks necessitate monotonic activation functions though.

Q4: Which activation functions should be used and when?

Roughly speaking, I wouldn't worry too much about this in practice. The usual choices are all nice and differentiable (with small caveats, like with

ReLU at 0) facilitating backpropagation. You should sometimes care about the output given the problem at hand. For example, if you need outputs in $[-1, 1]$ then tanh is a natural choice.

Note that some choices, like Leaky ReLU, introduce hyperparameters to the model which can be undesirable, e.g. if one seeks to minimise the number of hyperparameters for the strength of their result (if there's no hyperparameter to cherry pick then your result is more trustworthy).

Q5: How does one prevent vanishing/exploding gradients?

Skip connections help. Architectures which use them are sometimes called residual networks.

- Most intuitive benefit: it helps prevent some layer in the architecture from degrading the input entirely. It's like a nice reminder to the model what it was working from before it got to this point
- Reduces vanishing or exploding gradients: opens the door to far deeper architectures
- Ensures that the model has to learn the residual as opposed to the full underlying function: faster convergence usually
- Takes pressure off parameter initialisation: if you set parameters to zero then initially model is just the identity and learning from there is feasible

It's worth mentioning that said vanishing behaviour can be observed during forward passes too.

5 Deep Generative Modelling

Broadly speaking, a generative model seeks to offer ways to reason about the data-generative process to which it corresponds. Reasoning here could mean simulating the data-generative process (sampling) or answering questions pertaining to the frequency of certain events. The simplest example of a generative model I can think of with respect to this broad definition is a histogram: a distribution of the frequencies of events, as in Figure 26. A histogram allows one to see which event is most frequent, the portion of events which satistfy some condition (e.g. exceeding some quantitative threshold) or to simulate the data-generative process. So how do we go

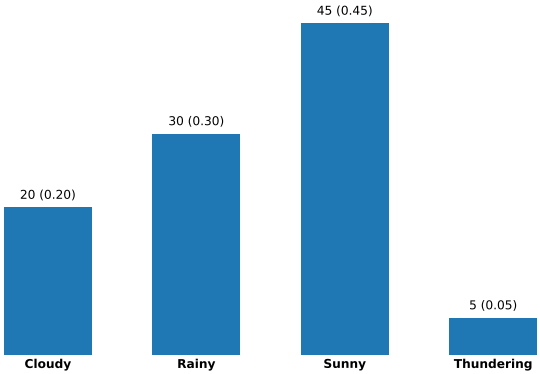


Figure 26: A histogram pertaining to the frequency of states of weather. Corresponding probabilities in parentheses.

from generative models to generative probabilistic models? Well, where frequency-based reasoning goes, probabilistic reasoning follows!

A generative probabilistic model is a tuple $(\mathbf{X}, p_\Theta)$ consisting of model variables $\mathbf{X} = (X_1, \ldots, X_m)$ and a probability function $p_\Theta : \Omega_\mathbf{X} \to \mathbb{R}_{\geq 0}$ parameterised according to the parameter set Θ. Given a generative probabilistic model, we might hope to answer evidence queries $p_\Theta(\mathbf{x})$, marginal queries $p_\Theta(\mathbf{x}')$, where $\mathbf{x}'$ is a realisation of some subset of the model variables, and sampling queries $\hat{\mathbf{x}} \sim p_\Theta(\mathbf{x})$, in which one simulates the data-generative process in turn producing samples of the underlying distribution.

The holy grail of generative probabilistic modelling is to find an exact, efficiently computable and algebraically manipulable representation p_Θ of the underlying joint probability function of interest. If obtained, one can reason probabilistically with ease but, as we will see, the purpose, limitations and learning of generative probabilistic models from data varies heavily. To illustrate this, consider probabilistic graphical models (PGMs) which offer human-interpretable representations of conditional (in)dependencies between model variables according to a directed acyclic graph (DAG). While PGMs are not employed in deep learning contexts, they act as an important stepping stone in understanding contemporary methods. The most well-known class of PGMs are Bayesian networks.

5.1 Bayesian networks

A natural approach to representing the joint probability function of model variables is to decompose it into a product of probability functions over subsets of the model variables. For example, you might decompose the

probability function according to the chain rule of probability as in

$$\begin{aligned} p(x_1, \ldots, x_m) &= p(x_1)p(x_2|x_1)\cdots p(x_m|x_1, \ldots, x_{m-1}) \\ &= \prod_{i=1}^{m} p(x_i|x_1, \ldots, x_{i-1}). \end{aligned}$$

From here, the probabilistic query of sampling, for example, may be done autoregressively: sample x_1 according to $p(x_1)$, x_2 according to $p(x_2|x_1)$ and so on until an entire realisation of $\mathbf{X} = (X_1, \ldots, X_m)$ is obtained. This idea leads to the natural motivation of Bayesian networks: why stop at decomposition according to the chain rule? In image date, for example, nearby pixel values may correlate but far away pixels may not, so why incoroporate distant pixels in a given conditional in the learned decomposition? Why not learn the conditional (in)dependencies of the model variables of interest and decompose accordingly? That's exactly what Bayesian networks do!

With their motivation out of the way, we can move on to their formulation. The parameter set $\Theta = (\mathcal{G}, \theta)$ of a Bayesian network $(\mathbf{X}, p_\Theta)$ consists of a directed acyclic graph (DAG) $\mathcal{G}$ and a parameter set θ. The graph $\mathcal{G}$ has a single node for each model variable $X_1, \ldots, X_m$ and directed edges represent conditional (in)dependencies. The parameter set θ pertains to the parameters of the conditional distributions (the smaller distributions in the decomposition) made explicit by $\mathcal{G}$. The representation of the joint probability function is given by

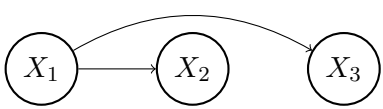
$$p_\Theta(x_1, \ldots, x_m) = \prod_{i=1}^{m} p_{\theta_i}(x_i|\mathbf{pa}(X_i))$$

where $\mathbf{pa}(X_i)$ denotes the random variables which are parents of X_i in $\mathcal{G}$.

Example 5.1 *For model variables $\mathbf{X} = (X_1, X_2, X_3)$, if $X_3|X_1$ is independent of X_2 then their joint probability function decomposes according to*

$$p_\Theta(x_1, x_2, x_3) = p_{\theta_1}(x_1)p_{\theta_2}(x_2|x_1)p_{\theta_3}(x_3|x_1)$$

whose corresponding DAG is given by



Notice that the choice of X_1 as the model variable upon which nothing is conditioned is entirely arbitrary in this example. As a result, for a given decomposition, DAGs are not unique. Here, $\theta = (\theta_1, \theta_2, \theta_3)$ are parameters learned from data.

5.1.1 Learning them from data

I won't go into particular detail regarding how one learns a Bayesian network (its graph $\mathcal{G}$ and parameter set θ) from data as the nitty gritty details aren't relevant to the generative models relevant to deep learning. That said, there are some interesting takeways, especially from methods of learning a Bayesian network's structure $\mathcal{G}$ as they contrast the almost entirely-heuristic approaches taken in deep learning contexts.

Unlike with neural network architectures, both the parameter set θ and the structure $\mathcal{G}$ of a Bayesian network can be motivated in a principled manner via maximum likelihood estimation. In brief, the likelihood of a given parameter set and DAG can be computed separately, and so in learning a Bayesian network from a dataset $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \overset{\text{i.i.d.}}{\sim} p(\mathbf{x})^{\otimes n}$ one might hope to compute

$$\begin{aligned}\arg\max_{\mathcal{G}} \left[\text{Score}(\mathcal{G})\right] &= \arg\max_{\mathcal{G}} \left[\arg\max_{\theta} \left[\log\left(p_{(\mathcal{G},\theta)}(D)\right)\right] - \text{penalty}(\mathcal{G})\right] \\ &= \arg\max_{\mathcal{G}} \left[\arg\max_{\theta} \left[\sum_{i=1}^{n} \log\left(p_{(\mathcal{G},\theta)}(\mathbf{x}_i)\right)\right] - \text{penalty}(\mathcal{G})\right]\end{aligned}$$

where the penalty term has a regularisation-like intention, For example, we might seek to penalise the complexity of $\mathcal{G}$ in the form of its number of edges according to its Bayesian information criterion (BIC). The reason that this is computationally feasible is precisely due to the decomposition of

$$p_{(\mathcal{G},\theta)}(\mathbf{x}) = \prod_{i=1}^{m} p_{\theta_i}(x_i|\mathbf{pa}(X_i))$$

according to the local dependencies present in $\mathcal{G}$.

A natural approach to compute the graph which maximises the relevant score is the greedy approach: start with an empty graph and repeat the following steps:

1. Compute the scores of all graphs which add or remove an edge from the current graph (or reverse an edge).
2. Edit the graph according to whichever addition, removal or reversal furthest increases the score.
3. No change in score? Stop, return $\mathcal{G}$ and recover θ accordingly.

With this approach in mind, the purpose of the penalty term becomes immediately clear: without it, the graph would continue to grow until the chain rule graph described earlier would be realised, which is undesirable.

5.1.2 Why aren't they employed in deep learning?

Learning their structure and parameters for high-dimensional distributions becomes quickly computationally infeasible (say from ~ 100). Regardless of this 'weakness', they serve as a significant, and often overlooked, step in the evolution of probabilistic reasoning and, more broadly, generative probabilistic modelling.

Another important note is that inference is exponential in the tree-width of the Bayesian network. Without going into a lot of detail, a Bayesian network's tree-width can be thought of as the number of dependencies among model variables. As a result, when fitting a high-dimensional Bayesian network in which the number of dependencies is high, inference is computationally infeasible. Unfortunate.

5.2 Variational Autoencoders (VAEs)

In motivating variational autoencoders (VAEs), it's natural to first motivate regular autoencoders, illustrate issues pertaining to intuitive ideas of doing generative things with them and fixing those issues with variational inference.

Note that VAEs can be motivated in a ton of other ways, e.g. Kingma (their creator) himself describes them in terms of Bayesian networks and latent variable models. Such a motivation can be found in `https://arxiv.org/pdf/1312.6114` which acts as a great source for learning about VAEs.

5.2.1 Autoencoders: no variation yet!

An autoencoder $(\Omega_{\mathbf{X}}, d, \phi, \theta)$ consists of a sample space $\Omega_{\mathbf{X}} \subset \mathbb{R}^m$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, an encoder $\phi : \mathbb{R}^m \to \mathbb{R}^d$ and a decoder $\theta : \mathbb{R}^d \to \mathbb{R}^m$. Using a subset of $\Omega_{\mathbf{X}}$, one typically learns the encoder ϕ and the decoder θ with the goal of approximating the identity on $\Omega_{\mathbf{X}}$ via $\theta \circ \phi$. That is, roughly put, one seeks to obtain an encoder/decoder pair such that $\theta(\phi(\mathbf{x})) \approx \mathbf{x}$ for all $\mathbf{x} \in \Omega_{\mathbf{X}}$.

In intuitive terms, one can see the encoder ϕ as a compressor of samples in $\Omega_{\mathbf{X}}$ to their compressed (or latent) representation in $\mathbb{R}^d$ and so one might refer to the image of ϕ as the latent space $\Omega_{\mathbf{Z}}$. Similarly, the decoder θ can be seen as a decompressor of compressed representations $\mathbf{z} \in \Omega_{\mathbf{Z}}$ yielding the original sample $\mathbf{x}$. In line with this notion of an autoencoder as a compressor/decompressor, the latent dimension d is typically taken to be far smaller than the dimension of the distribution of interest, i.e. $d \ll m$. Of course, when $d \ll m$, learning such mappings ϕ and θ typically involves

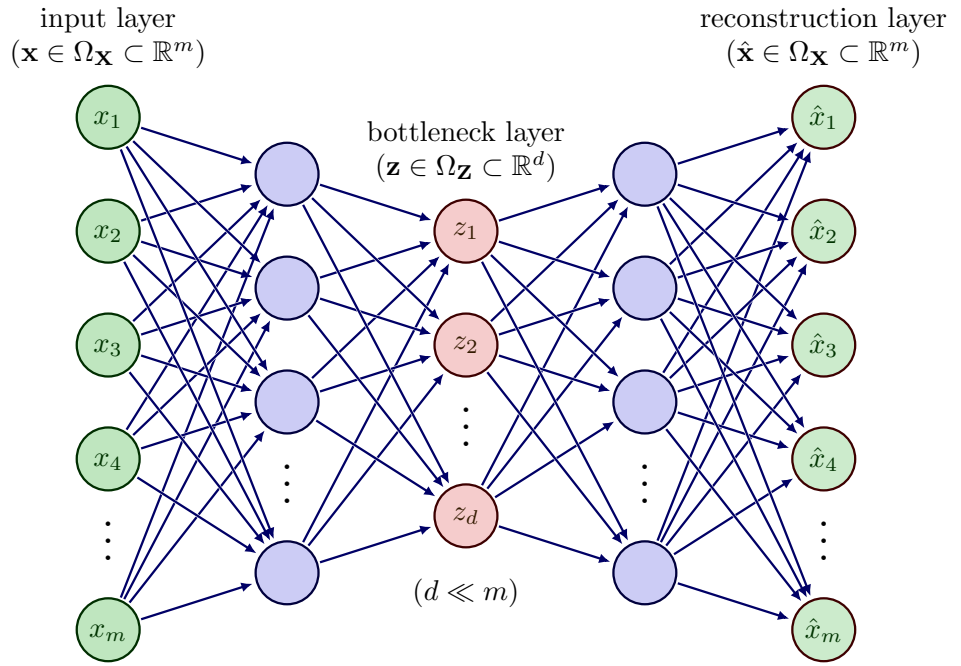


Figure 27: An autoencoder in which ϕ and θ are fit using multi-layer perceptrons (MLPs).

some loss of information if $\Omega_{\mathbf{X}}$ is a manifold whose intrinsic dimension is greater than d. That is, autoencoders are typically lossy compressors.

Example 5.2 *Suppose* $\Omega_{\mathbf{X}} = \{(a, a, b) \in \mathbb{R}^3 : ||(a, a, b)|| \leq 1\}$ *and* $d = 2$. *One immediately notices that* $\Omega_{\mathbf{X}}$ *is a two-dimensional surface embedded in* $\mathbb{R}^3$ *as it is the intersection of the unit ball and the plane* $\{(a, a, b) : a, b \in \mathbb{R}\}$. *To produce representations of* $\mathbf{x} = (a, a, b) \in \Omega_{\mathbf{X}}$ *in* $\mathbb{R}^2$ *(i.e. to compress a sample) one might employ the encoder* $\phi_2(x, y, z) = (x, z)$. *To reconstruct samples from their latent representation (i.e. to decompress) one might employ the decoder* $\theta_2(x, z) = (x, x, z)$. *The autoencoder* $(\Omega_{\mathbf{X}}, 2, \phi_2, \theta_2)$ *offers lossless compression on the sample space as* $(\theta_2 \circ \phi_2)(\mathbf{x}) = \mathbf{x}$ *for all* $\mathbf{x} \in \Omega_{\mathbf{X}}$.

If we instead desire latent representations of $\mathbf{x} = (a, a, b) \in \Omega_{\mathbf{X}}$ *in* $\mathbb{R}$, *i.e. if* $d = 1$, *one might employ the encoder* $\phi_1(x, y, z) = x$ *and the decoder* $\theta_1(x) = (x, x, x)$. *The autoencoder* $(\Omega_{\mathbf{X}}, 1, \phi_1, \theta_1)$ *offers lossy compression, i.e. some information pertaining to a sample is lost during its compression and decompression as* $(\theta_1 \circ \phi_1)(a, a, b) = (a, a, a)$ *for all* $a, b \in \mathbb{R}$.

One's tolerance for the loss incurred by a given encoder/decoder pair

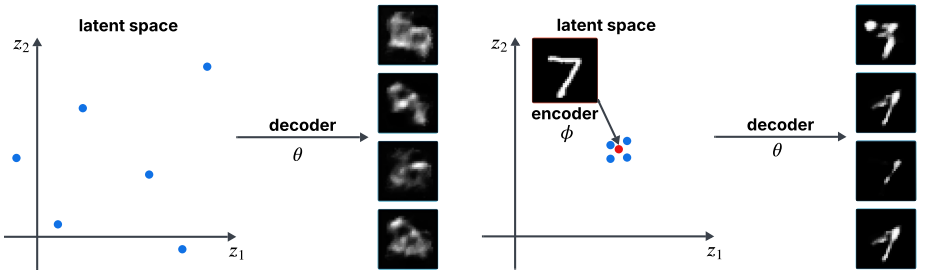


Figure 28: Intuitive ideas for using an autoencoder as a generative model which fail.

is task-dependent. Given a dataset $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \Omega_{\mathbf{X}}$ and a latent dimension d, one might compute the pair $(\phi^*, \theta^*) \in \Phi \times \Theta$ which minimises the empirical risk over D where Φ and Θ are chosen function families. That is, computing

$$(\phi^*, \theta^*) = \underset{(\phi,\theta)\in\Phi\times\Theta}{\arg\min} \sum_{i=1}^{n} ||\mathbf{x}_i - \theta(\phi(\mathbf{x}_i))||^2.$$

If the function families permit the computation of gradients of $\phi \in \Phi$ and $\theta \in \Theta$ then this computation may be done using gradient descent. For example, one could take Φ and Θ to be the family of MLPs of appropriate input and output dimensions, as illustrated in Figure 27.

5.2.2 Motivating VAEs

Autoencoders are great for compression, denoising, disecting LLMs (sparse autoencoders), etc. but we're yet to see how we might apply them to generative tasks, e.g. sampling from the complex underlying distribution $p(\mathbf{x})$ from which their training dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ was sampled. An intuitive first idea is to train an autoencoder, randomly sample from its latent space and feed the sampled latent through the learned decoder, as in Figure 28. The outputs of the decoder should be similar to some subet of the training data, right? No.

Disaster strikes and we begin to see just how unstructured the latent space of an autoencoder is: randomly sampling from it and decoding yields nonsense outputs from the decoder because most of the latent space itself is meaningless (nothing is encoded to most of it, so in some sense a lot of it

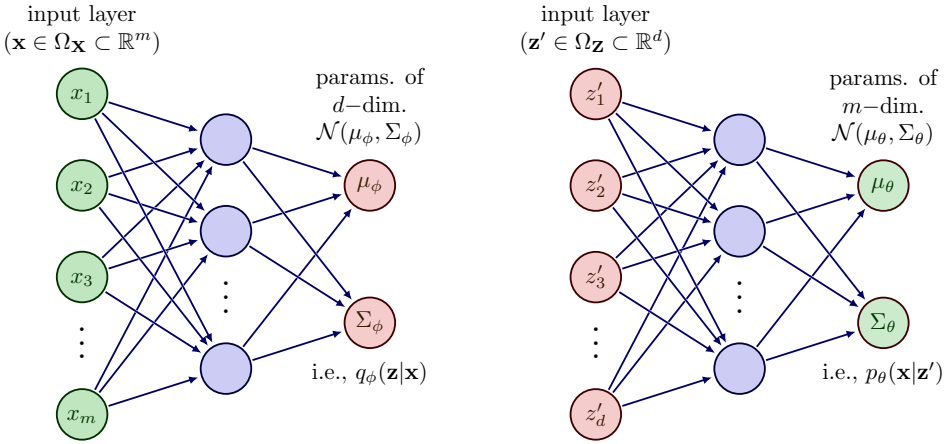


Figure 29: Left: an encoder which outputs parameters of the latent distribution $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\phi, \Sigma_\phi)$. Right: a decoder which outputs parameters of the reconstruction distribution $p_\theta(\mathbf{x}|\mathbf{z}') = \mathcal{N}(\mu_\theta, \Sigma_\theta)$ in which $\mathbf{z}' \sim q_\phi(\mathbf{z}|\mathbf{x})$.

is 'un-utilised'). This is perhaps unsurprising as at no point during training an autoencoder do we encourage it to interpolate nicely between encodings.

An intuitive second idea is to feed the decoder points in the latent space which are close to the encoding of a given training sample, as in Figure 28. The outputs of the decoder should be a sample of the class of said encoding's sample, right? No. Distaster strikes again. For meaningful decodings using this idea, one must take points in the latent space which are ridiculously close to this chosen sample's latent embedding. So close that you'd be practically reconstructing the original training sample each time — certainly not what we mean when we say that we'd like to generate new samples. So how do we do autoencoder-like things in a way that yields well-structured latent spaces? Variational autoencoders (VAEs) to the rescue!

5.2.3 Formulating VAEs

Leaving aside, for now, how to learn one from data, VAEs can informally be seen as an extension of autoencoders in which the encoder and decoder each output parameters of a distribution belonging to some pre-chosen distribution family. Extending the notation used to define autoencoders, a variational autoencoder $(\Omega_{\mathbf{X}}, d, \mathcal{Q}_d, \mathcal{P}_m, \phi, \theta)$ consists of the sample space of the distribution of interest $\Omega_{\mathbf{X}} \subset \mathbb{R}^m$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, the parameter space $\mathcal{Q}_d$ of a family of $d-$dimensional conditional distributions

denoted $q_\phi(\mathbf{z}|\mathbf{x})$, the parameter space $\mathcal{P}_m$ of a family of $m-$dimensional conditional distributions denoted $p_\theta(\mathbf{x}|\mathbf{z})$, an encoder $\phi : \mathbb{R}^m \to \mathcal{Q}_d$ and a decoder $\theta : \mathbb{R}^d \to \mathcal{P}_m$.

To illustrate the intended meaning of the newly-introduced parameter spaces $\mathcal{Q}_d$ and $\mathcal{P}_m$, one's encoder might yield the expectation vector and covariance matrix of a $d-$dimensional Gaussian $\mathcal{N}(\mu, \Sigma)$, i.e. $\mathcal{Q}_d$ could be the parameter space of the family of $d-$dimensional Gaussian distributions yielding

$$\mathcal{Q}_d = \{(\mu, \Sigma) : \mu \in \mathbb{R}^d, \Sigma \in \mathcal{S}^d_{++}\} = \mathbb{R}^d \times \mathcal{S}^d_{++}$$

where $\mathcal{S}^d_{++}$ denotes the set of all positive-definite matrices in $\mathbb{R}^{d\times d}$.

As the encoder of a VAE yields a $d-$dimensional distribution $q_\phi(\mathbf{z}|\mathbf{x})$ given the sample $\mathbf{x} \in \Omega_\mathbf{X}$, to obtain a latent $d-$dimensional representation $\mathbf{z}' \in \mathbb{R}^d$ of $\mathbf{x}$, one computes the parameters $\phi(\mathbf{x}) \in \mathcal{Q}_d$ of $q_\phi(\mathbf{z}|\mathbf{x})$, e.g. a $d-$dimensional Gaussian, via the encoder and samples $\mathbf{z}' \sim q_\phi(\mathbf{z}|\mathbf{x})$. To reconstruct $\mathbf{x}$ from its latent representation $\mathbf{z}'$, one computes the parameters $\theta(\mathbf{z}') \in \mathcal{P}_m$ of the $q-$dimensional reconstruction distribution $p_\theta(\mathbf{x}|\mathbf{z}')$ via the decoder. Sampling $\mathbf{x}' \sim p_\theta(\mathbf{x}|\mathbf{z}')$ yields (ideally) a sufficiently-accurate reconstruction of the original sample $\mathbf{x}$. Achieving accurate reconstructions is done in a similar manner to autoencoders: by including a penalty term pertaining to reconstruction quality in the loss function used to learn the encoder and decoder.

Note that one's choice of $\mathcal{Q}_d$ and $\mathcal{P}_m$ should take into account the need to sample efficiently and so they should correspond to families of distributions which offer efficient means of sampling, e.g. Gaussians, as in Figure 29.

Example 5.3 *Suppose $X_1 \sim \mathcal{N}(1, 2)$ and $X_3 \sim \mathcal{N}(-1, 1)$ are independent. Let $X_2 = X_1 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$ is independent of X_1 and X_3. If $X = (X_1, X_2, X_3)$ then $X \sim \mathcal{N}(\mu, \Sigma)$ with $\mu = (1, 1, -1)'$ and the relevant covariance matrix Σ. As such, $\Omega_\mathbf{X} = \mathbb{R}^3$. To obtain two-dimensional representations of samples $\mathbf{x} \in \Omega_\mathbf{X}$ (so $d = 2$), one might choose $\mathcal{Q}_2 = \mathbb{R}^2 \times \mathcal{S}^2_{++}$ and $\mathcal{P}_3 = \mathbb{R}^3 \times \mathcal{S}^3_{++}$. That is, we could fit a two-dimensional Gaussian to the latent space and a three-dimensional Gaussian to the reconstruction space. Knowing $X_2 = X_1 + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$, one might choose the encoder*

$$\begin{aligned} \phi : \mathbb{R}^3 &\to \mathcal{Q}_2 \\ (x_1, x_2, x_3) &\mapsto ((x_1, x_3), \sigma^2 I_2) \end{aligned}$$

where $\sigma > 0$ is small. That is, for a sample $(x_1, x_2, x_3) \in \Omega_\mathbf{X}$, the encoder's output would yield the latent distribution $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}((x_1, x_3), \sigma^2 I_2)$. As

decoder, if all we desire is accurate reconstructions, we might choose

$$\begin{aligned}\theta : \mathbb{R}^2 &\to \mathcal{P}_3 \\ (z_1, z_2) &\mapsto ((z_1, z_1, z_2), \sigma^2 I_3).\end{aligned}$$

That is, for the latent sample $\mathbf{z}' = (z'_1, z'_2)$, the decoder's output would yield the reconstruction distribution $p_\theta(\mathbf{x}|\mathbf{z}') = \mathcal{N}((z'_1, z'_1, z'_2), \sigma^2 I_3)$. Ideally, sampling $(x'_1, x'_2, x'_3) \sim p_\theta(\mathbf{x}|\mathbf{z}')$ would yield a sample sufficiently similar to the original sample $\mathbf{x} = (x_1, x_2, x_3)$.

Note that in practice, one does not know the true distribution $p(\mathbf{x})$ and so hand-picking the encoder and decoder as in this example is infeasible. Typically, the encoder and decoder are learned from a dataset $D \subset \Omega_{\mathbf{X}}$.

At this point, a natural question arises: for which tasks is learning a VAE more appropriate than learning an autoencoder? The answer lies in the purpose of VAEs which is two-fold: 1) to perform sufficiently-accurate compression/decompression and 2) to produce a sufficiently regularised approximation of the latent space $\Omega_{\mathbf{Z}}$. The latter is ensured by how one learns a VAE from data, which we soon consider. In brief, when learning an autoencoder one never imposes restrictions on the latent space beyond encouraging the model to yield sufficiently-accurate reconstructions. As a result, the latent space of an autoencoder is not well-structured. For example, for latent samples $\mathbf{z}_1, \mathbf{z}_2 \in \Omega_{\mathbf{Z}}$ which are 'close' in the latent space, their reconstructions are not necessarily 'close' in $\mathbb{R}^m$. VAEs seek to remedy this.

To learn a VAE from data, we look to maximise the evidence lower bound (ELBO) over some dataset $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \Omega_{\mathbf{X}}$. Over a single sample $\mathbf{x} \in \Omega_{\mathbf{X}}$, the ELBO is a tight lower bound of $\log(p(\mathbf{x}))$. As such, maximising the ELBO over D can be seen as performing approximate maximum-likelihood estimation over D (sometimes referred to as evidence maximisation). To derive the ELBO, first note that given a decoder θ (which parameterises the reconstruction distribution $p_\theta(\mathbf{x}|\mathbf{z})$ and by extension the marginal distribution $p_\theta(\mathbf{x})$), we may express the marginal distribution $p_\theta(\mathbf{x})$ as

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z} = \int p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}. \quad (1)$$

Using Equation 1, along with the encoder ϕ (which parameterises the latent

distribution $q_\phi(\mathbf{z}|\mathbf{x})$) we derive the ELBO over a single sample $\mathbf{x} \in \Omega_{\mathbf{X}}$:

$$\begin{aligned}
\log(p_\theta(\mathbf{x})) &= \log\left(\int p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})d\mathbf{z}\right) \\
&= \log\left(\int q_\phi(\mathbf{z}|\mathbf{x})\frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}d\mathbf{z}\right) \\
&= \log\left(\mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\frac{p_\theta(\mathbf{x}|\mathbf{Z})p_\theta(\mathbf{Z})}{q_\phi(\mathbf{Z}|\mathbf{x})}\right]\right) \\
&\geq \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_\theta(\mathbf{x}|\mathbf{Z})p_\theta(\mathbf{Z})}{q_\phi(\mathbf{Z}|\mathbf{x})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] - \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{q_\phi(\mathbf{Z}|\mathbf{x})}{p_\theta(\mathbf{Z})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] - D_{\mathrm{KL}}(q_\phi(\mathbf{Z}|\mathbf{x})||p_\theta(\mathbf{Z})) \\
&=: \mathrm{ELBO}
\end{aligned}$$

in which the inequality arises due to Jensen's inequality, as in $\mathbb{E}[\log(f(\mathbf{Z}))] \leq \log(\mathbb{E}[f(\mathbf{Z})])$, and $D_{\mathrm{KL}}(Q||P)$ denotes the KL-divergence between distributions Q and P, which is detailed in the appendix. It's worth noting that equivalent representations of the ELBO exist and are useful in motivating VAEs, as in

$$\begin{aligned}
\mathrm{ELBO} &= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] - \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{q_\phi(\mathbf{Z}|\mathbf{x})}{p_\theta(\mathbf{Z})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_\theta(\mathbf{x},\mathbf{Z})}{p_\theta(\mathbf{Z})}\right) - \log\left(\frac{q_\phi(\mathbf{Z}|\mathbf{x})}{p_\theta(\mathbf{Z})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x},\mathbf{Z})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right]
\end{aligned}$$

or

$$\begin{aligned}
\mathrm{ELBO} &= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x},\mathbf{Z})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{Z}|\mathbf{x})p_\theta(\mathbf{x})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x})) - \log\left(\frac{q_\phi(\mathbf{Z}|\mathbf{x})}{p_\theta(\mathbf{Z}|\mathbf{x})}\right)\right] \\
&= p_\theta(\mathbf{x}) - D_{\mathrm{KL}}\left(q_\phi(\mathbf{Z}|\mathbf{x})||p_\theta(\mathbf{Z}|\mathbf{x})\right).
\end{aligned}$$

The effect of maximising the ELBO can be motivated in a number of ways. Its statement directly above makes it immediately clear that its maximisation corresponds to maximum likelihood and the minimisation of the KL-divergence between the model posterior $q_\phi(\mathbf{Z}|\mathbf{x})$ and the induced posterior

$p_\theta(\mathbf{Z}|\mathbf{x})$. That said, $p_\theta(\mathbf{Z}|\mathbf{x})$ is intractable (which is why we learn the model posterior $q_\phi(\mathbf{Z}|\mathbf{x})$), and so I prefer to motivate the maximisation of the ELBO term-by-term with respect to the following form

$$\text{ELBO} = \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] - D_{\text{KL}}(q_\phi(\mathbf{Z}|\mathbf{x})||p_\theta(\mathbf{Z})).$$

The first term is a principled measure of the VAE's ability to reconstruct latent representations, as with autoencoders. The second term is a principled measure of the similarity of the latent distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and the prior $p(\mathbf{z})$. The prior is chosen before training and the most common choice is a $d-$dimensional standard Gaussian, i.e. $p(\mathbf{z}) = \mathcal{N}(0, I_d)$. As such, minimising the negative KL-divergence between the latent distribution and said prior is often interpreted as encouraging the learning of the encoder such that latent representations are distributed according to the prior. This is particularly useful in the case that one is learning a VAE to generate new samples from $\Omega_{\mathbf{X}}$: post-training, sample $\mathbf{z}' \sim p(\mathbf{z})$, compute the parameters $\theta(\mathbf{z}')$ of the reconstruction distribution $p_\theta(\mathbf{x}|\mathbf{z}')$ via the decoder and sample from it. Note that using a VAE for generative purposes does not invoke the use of the encoder, only the decoder is required post-training.

Given a dataset $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \Omega_{\mathbf{X}}$, we learn a VAE by choosing function classes Φ and Θ (e.g. MLPs as in Figure 29) and computing

$$\underset{(\phi,\theta)\in\Phi\times\Theta}{\arg\max}\left[\sum_{i=1}^{n}\mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x}_i)}\left[\log(p_\theta(\mathbf{x}_i|\mathbf{Z}))\right] - D_{\text{KL}}(q_\phi(\mathbf{Z}|\mathbf{x}_i)||p_\theta(\mathbf{Z}))\right].$$

In practice, after choosing a latent dimension d, we often take $p(\mathbf{z}) = \mathcal{N}(0, I_d)$, $\mathcal{Q}_d = \mathbb{R}^d \times \mathcal{S}^d_{++}$ and $\mathcal{P}_m = \mathbb{R}^m \times \mathcal{S}^m_{++}$, i.e. Gaussians for the latent and reconstruction distribution families and the standard $d-$dimensional Gaussian for the prior. A benefit of these choices is that it yields a differentiable and easy-to-implement closed form for the KL-divergence term in the ELBO. Additionally, the expectation pertaining to the reconstruction term is typically approximated via a single sample, boiling down to a mean square error term[5].

5.2.4 Backpropagation for VAEs (the reparameterisation trick)

To find suitable parameters using gradient descent, we require estimates of ∇_θELBO and ∇_ϕELBO. The former is straightforwardly estimated using

[5] `https://n8python.github.io/mnistLatentSpace/`

Monte Carlo estimation as in

$$\begin{aligned}\nabla_\theta \text{ELBO} &= \nabla_\theta \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] \\ &= \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\nabla_\theta \log(p_\theta(\mathbf{x}|\mathbf{Z}))\right] \\ &\approx \frac{1}{k}\sum_{j=1}^{k} \nabla_\theta \log(p_\theta(\mathbf{x}|\mathbf{z}_j)).\end{aligned}$$

where $\mathbf{z}_1, \ldots, \mathbf{z}_k \sim q_\phi(\mathbf{z}|\mathbf{x})$. A closed form is often obtained by taking convenient distributions, e.g. $p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \mu_\theta(\mathbf{z}), \sigma_\theta(\mathbf{z})I)$.

Estimating ∇_ϕELBO, however, is tricky. The expectation itself is with respect to $q_\phi(\mathbf{z}|\mathbf{x})$ which explicitly depends on ϕ and so the gradient ∇_ϕ can not be brought inside. That is,

$$\nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})] \neq \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\nabla_\phi f(\mathbf{Z})].$$

Before we proceed by describing the reparameterisation trick, it's worth noting that tehre is a closed form for $\nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})]$ referred to as the score function estimator given by

$$\nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})] = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})\nabla_\phi \log\left(q_\phi(\mathbf{Z}|\mathbf{x})\right)]$$

which can be MC-estimated. The reason that this estimator is not used is its variance being very high. A lot of research has been made into reducing the variance of the estimator but, ultimately, the estimate used in practice makes use of the reparameterisation trick.

Suppose instead that you MC-estimated $\nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})]$. The issue here is that you would then have to compute the gradient through the process that yielded the samples $\mathbf{z}_1, \ldots, \mathbf{z}_k$ contributing to your estimate. For example, if

$$\mathbf{z}_i = F_\phi^{-1}(u_i)$$

where $u_1, \ldots, u_k \sim \text{Unif}(u; 0, 1)$ then the process by which our latents are obtained is not deterministic but explicitly depends on ϕ. How then does one backpropagate through this process? Auto-differentiation isn't a fan. The idea of the reparameterisation trick is very natural: entirely separate the stochasticity obtaining said samples from the parameters ϕ. This can be done by taking $\mathbf{Z}|\mathbf{x} = g(\boldsymbol{\epsilon}; \phi, \mathbf{x})$ for invertible g in which $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I_d)$ is the only stochastic component of sampling. With such a function $\mathbf{Z} = g(\boldsymbol{\epsilon}; \phi, \mathbf{x})$ we have (via the law of the unconscious statistician)

$$\begin{aligned}\nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{Z})] &= \nabla_\phi \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(\mathbf{Z})] \\ &= \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla_\phi f(\mathbf{Z})]\end{aligned}$$

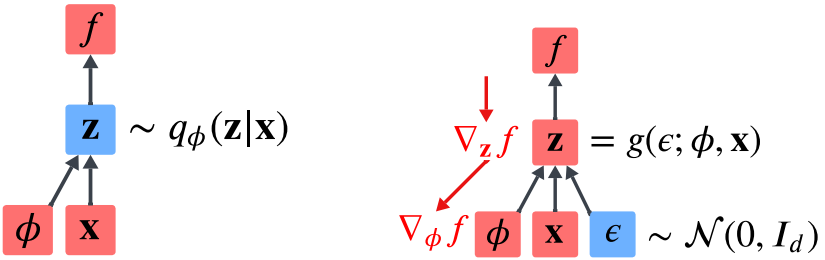


Figure 30: Backpropagation before and after applying the reparameterisation trick. Deterministic nods in blue and stochastic nodes in red.

and so relevant estimate is given by

$$\begin{aligned}
\nabla_\phi \text{ELBO} &= \nabla_\phi \mathbb{E}_{\mathbf{Z}\sim q_\phi(\mathbf{z}|\mathbf{x})} \left[\log(p_\theta(\mathbf{x}|\mathbf{Z})) + \log(p_\theta(\mathbf{Z})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right] \\
&= \nabla_\phi \mathbb{E}_{p(\boldsymbol{\epsilon})} \left[\log(p_\theta(\mathbf{x}|\mathbf{Z})) + \log(p_\theta(\mathbf{Z})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right] \\
&= \mathbb{E}_{p(\boldsymbol{\epsilon})} \left[\nabla_\phi \left(\log(p_\theta(\mathbf{x}|\mathbf{Z})) + \log(p_\theta(\mathbf{Z})) - \log(q_\phi(\mathbf{Z}|\mathbf{x}))\right)\right] \\
&\approx \frac{1}{k} \sum_{i=1}^{k} \nabla_\phi \left(\log(p_\theta(\mathbf{x}|\mathbf{z}_i)) + \log(p_\theta(\mathbf{z}_i)) - \log(q_\phi(\mathbf{z}_i|\mathbf{x}))\right)
\end{aligned}$$

where $\mathbf{z}_i = g(\boldsymbol{\epsilon}_i; \phi, \mathbf{x})$ and $\boldsymbol{\epsilon}_1, \ldots, \boldsymbol{\epsilon}_k \sim \mathcal{N}(\boldsymbol{\epsilon}; 0, I_d)$. Ignore my abuse of notation of conflating the random variable $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I_d)$ with corresponding samples. This estimator is unbiased and all that good stuff. For an illustration of how the reparameterisation helps alleviate the earlier issue with backpropagation, consider Figure 30.

An explicit example of such a reparameterisation is when the posterior is a facorised Gaussian. That is, $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu_\phi(\mathbf{x}), \text{diag}(\sigma^2_\phi(\mathbf{x})))$. In this case, take $g(\boldsymbol{\epsilon}; \phi, \mathbf{x}) = \mu_\phi(\mathbf{x}) + \sigma_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}; 0, I_d)$ and $\odot$ denotes element-wise multiplication. Typically, the prior $p_\theta(\mathbf{z})$ and the likelihood $p_\theta(\mathbf{x}|\mathbf{z})$ are also taken such that their logarithm is nice to compute and backproppable.

Computing $\log(q_\phi(\mathbf{z}|\mathbf{x}))$

Though nice distributions are often attributed to the posterior, it's worth noting that there's a convenient manner of computing $\log(q_\phi(\mathbf{z}|\mathbf{x}))$ which relies on the invertibility of $\mathbf{Z} = g(\boldsymbol{\epsilon}; \phi, \mathbf{x})$. Note that

$$\begin{aligned}\log(q_\phi(\mathbf{z}|\mathbf{x})) &= \log\left(p(\boldsymbol{\epsilon})\left|\det\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}\right)\right|^{-1}\right) \\ &= \log(p(\boldsymbol{\epsilon})) - \log\left(\left|\det\left(\begin{bmatrix}\frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_d}{\partial \epsilon_1} & \cdots & \frac{\partial z_d}{\partial \epsilon_d}\end{bmatrix}\right)\right|\right)\end{aligned}$$

where $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_d)$. This choice is particularly useful for complicated-looking choices of the posterior.

5.2.5 Blurry reconstructions

Blur is often observed in reconstructions via the decoder of VAEs. Its presence can be motivated by noting that the maximisation of the ELBO is equivalent to the minimisation of

$$D_{\mathrm{KL}}(q_{D,\phi}(\mathbf{X}, \mathbf{Z})||p_\theta(\mathbf{X}, \mathbf{Z})) = \mathbb{E}_{(\mathbf{X},\mathbf{Z})\sim q_{D,\phi}(\mathbf{x},\mathbf{z})}\left[\log\left(\frac{q_{D,\phi}(\mathbf{X}, \mathbf{Z})}{p_\theta(\mathbf{X}, \mathbf{Z})}\right)\right]$$

where $q_{D,\phi}(\mathbf{x}, \mathbf{z})$ is induced by $q_\phi(\mathbf{z}|\mathbf{x})$ and the empirical data distribution $q_D(\mathbf{x})$. The KL-divergence clearly illustrates that

1. If $(\mathbf{x}, \mathbf{z})$ is unlikely according to $q_{D,\phi}$ then the KL-divergence (which corresponds to the loss) is relatively low regardless of how likely it is under p_θ.

2. If $(\mathbf{x}, \mathbf{z})$ is likely according to $q_{D,\phi}$ but unlikely according to p_θ then the KL-divergence blows up massively.

As such, the model is encouraged to ensure that $p_\theta(\mathbf{x}, \mathbf{z})$ is never small near observed data. This boils down to ensuring that the reconstruction likelihood $p_\theta(\mathbf{x}|\mathbf{z})$ is not small (since $q_\phi(\mathbf{z}|\mathbf{x})$ is regularised toward $p_\theta(\mathbf{z})$, sampled latents typically lie in reasonably high prior-density regions). With a unimodal likelihood (e.g. Gaussian with fixed variance), the safest fit when distinct $\mathbf{x}$ share similar $\mathbf{z}$ is to place the conditional mean between

them which results in blur. Note also that, even though the model is not penalised for what $p_\theta(\mathbf{x}|\mathbf{z})$ does for $(\mathbf{x}, \mathbf{z})$ such that $q_{D,\phi}(\mathbf{x}, \mathbf{z}) \approx 0$, p_θ is still subject to normalising to 1 under its support. Due to this, p_θ can't get too wild in interpolating between the data.

As illustration, suppose two samples $\mathbf{x}_1$ and $\mathbf{x}_2$ map to the same latent under the encoder. That is, suppose $\mathbf{z}_1 \approx \mathbf{z}_2$ for $\mathbf{z}_1 \sim q_\phi(\mathbf{z}|\mathbf{x}_1)$ and $\mathbf{z}_2 \sim q_\phi(\mathbf{z}|\mathbf{x}_2)$. Denote both by $\mathbf{z}_0$. Note that this is entirely plausible as the maximisation of the ELBO encourages the encoder to resemble the simple prior, e.g. the standard multivariate Gaussian with low latent dimension, and so it may be the case that there simply isn't enough room in the latent space to assign unique latents to distinct samples. In such a case, if $p_\theta(\mathbf{x}|\mathbf{z}_0)$ is unimodal, e.g. a Gaussian (which is often the case for vanilla VAEs), then where should its mean lie? That is, which of the samples which are sent to it by the encoder should it favour for reconstruction? If $\mathbf{X}|\mathbf{z} \sim \mathcal{N}(\mu_\theta(\mathbf{z}), \sigma^2 I)$ then maximising the reconstruction term in the ELBO requires us to compute

$$\begin{aligned}\arg\max_\theta \left[\sum_{i=1}^{2} \log(p_\theta(\mathbf{x}_i|\mathbf{z}_0))\right] &= \arg\max_\theta \left[-\frac{1}{2\sigma^2}\sum_{i=1}^{2} ||\mathbf{x}_i - \mu_\theta(\mathbf{z}_0)||^2\right] \\ &= \arg\min_\theta \left(||\mathbf{x}_1 - \mu_\theta(\mathbf{z}_0)||^2 + ||\mathbf{x}_2 - \mu_\theta(\mathbf{z}_0)||^2\right)\end{aligned}$$

which corresponds to $\mu_\theta(\mathbf{z}_0) = \arg\min_\mu \sum_{i=1}^{2} ||\mathbf{x}_i - \mu||^2 = (\mathbf{x}_1 + \mathbf{x}_2)/2$.

5.2.6 Working in latent space

...

5.3 Generative Adversarial Networks (GANs)

To motivate generative adversarial networks (GANs), first consider what we want a generative model to do: produce convincing samples. As such, if there is some (imperfect but decent) method of distinguishing real samples from those produced by a generative model, a perfect generative model would be able to fool the discriminative method. That is, the decisions made by the discriminative model should be akin to random guessing if the generative model is perfect. This idea was realised by Ian Goodfellow while at a bar (the story is detailed in Genius Makers).

This is precisely the idea that motivates GANs with inspiration taken from game theory. A GAN consists of a generative component $G : \Omega_\mathbf{Z} \to \Omega_\mathbf{X}$ and a discriminative component $D : \Omega_\mathbf{X} \to (0, 1)$, which are trained via

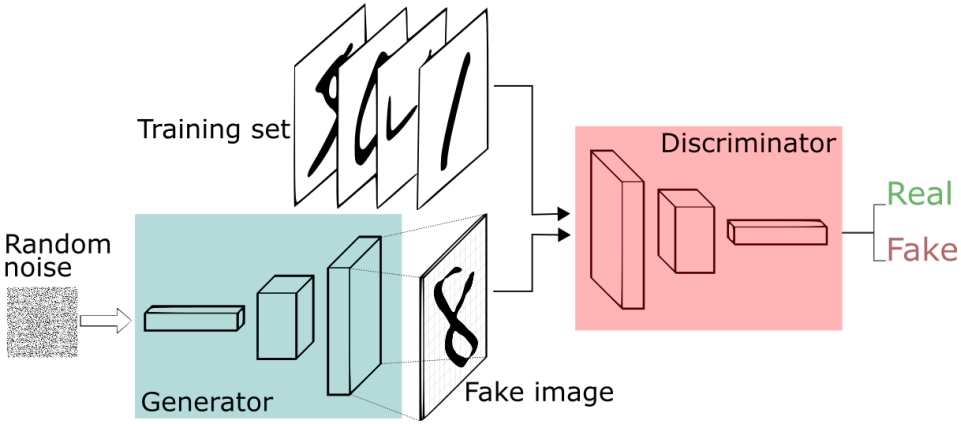


Figure 31: Example GAN architecture.

alternating gradient updates in the style of a minimax game. That is, G produces a new sample given a latent $\mathbf{z} \in \Omega_{\mathbf{Z}}$ while D learns to distinguish between training samples and outputs of G. Essentially, the models compete. In terms of loss functions, the discriminator employs binary cross entropy, as in

$$\begin{aligned}\mathcal{L}_D &= -\mathbb{E}_{\mathbf{X}\sim p}\left[\log(D_\phi(\mathbf{X}))\right] - \mathbb{E}_{\mathbf{Z}\sim\mathcal{N}(0,I)}\left[\log\left(1 - D_\phi(G_\theta(\mathbf{Z}))\right)\right] \\ &\approx -\frac{1}{k}\sum_{i=1}^{k}\Big[\log(D_\phi(\mathbf{x}_i)) + \log(1 - D_\phi(G_\theta(\mathbf{z}_i)))\Big] \\ &= -\frac{1}{k}\sum_{i=1}^{k}\log(D_\phi(\mathbf{x}_i)(1 - D_\phi(G_\theta(\mathbf{z}_i)))),\end{aligned}$$

for samples $\mathbf{x}_1, \ldots, \mathbf{x}_k$ and sampled noise $\mathbf{z}_1, \ldots, \mathbf{z}_k \sim \mathcal{N}(\mathbf{z}; 0, I)$, while the generator makes use of

$$\begin{aligned}\mathcal{L}_G &= -\mathbb{E}_{\mathbf{Z}\sim\mathcal{N}(0,I)}\left[\log(D_\phi(G_\theta(\mathbf{Z})))\right] \\ &=\approx -\frac{1}{k}\sum_{i=1}^{k}\log(D_\phi(G_\theta(\mathbf{z}_i)))\end{aligned}$$

for sampled noise $\mathbf{z}_1, \ldots, \mathbf{z}_k \sim \mathcal{N}(\mathbf{z}; 0, I)$.

As for explicit architectures, the usual choices are natural: the discriminator is fit using a CNN and the generator is fit using a transposed convolutional network. So, as with other image generative models, to generate a new sample, we sample Gaussian noise and run it through the transposed convolutional network.

5.3.1 NEXT: Flavours of GANs

A major contribution of GANs is not only their game theory-like training paradigm but the many flavours in which they come: conditional GANs, cycle GANs, multiple player GANs, style GANs, etc.

Despite their many flavours, GANs come with quirks that other model avoid:

- Training them is often a headache, e.g. the discriminator can quickly overfit making the generative side of things diffcult. This necessitates a lot of data.
- Mode collapse, in which the diversity of samples at inference, is low.
- No approximate likelihood computations according to a GAN so evaluation is difficult.
- Training stability is very architecture-dependent, interestingly.

5.4 TODO: Normalising Flows

...

5.5 Diffusion Models

The purpose of diffusion models is to facilitate the generation of samples from complex distributions from which sampling is typically intractable. While this overarching motive is not unique to diffusion models, how a diffusion model learns and how it generates new samples is quite distinct from other well-known generative models like VAEs and GANs (though some ideas are certainly comparable). A diffusion model can be described by its forward (noising) and backward (denoising) processes. Its forward process iteratively noises a sample $\mathbf{x}_0 \in \mathcal{X}$, belonging to the distribution of interest, T-many times to obtain its noised equivalent $\mathbf{x}_T$. Formally, this is done via the Markov process

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}\left(\mathbf{x}_t \middle| \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t I\right)$$

where $\beta_1, \ldots, \beta_T \in [0,1]$ are hyperparameters satisfying $\beta_i < \beta_{i+1}$, often referred to as the noise schedule of the model. With an appropriately chosen final time T, these noised equivalents $\mathbf{x}_T$ are akin to random noise sampled from $\mathcal{N}(0, I)$.

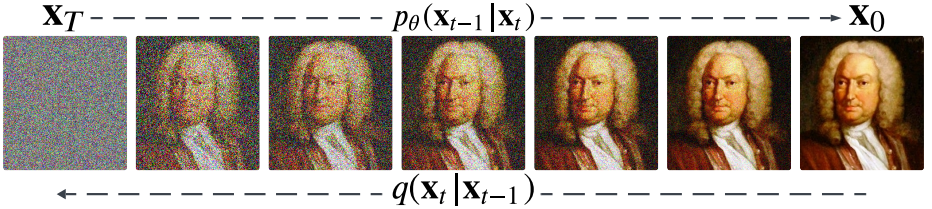


Figure 32: Johann Bernoulli being denoised in line with the backward process of a diffusion model.

Letting $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, we can describe the backward process of a diffusion model again as a Markov process in which we sample some random noise $\mathbf{x}_T$ from $\mathcal{N}(0, I)$ and obtain the $(t-1)$st denoised sample from $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ via

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$$

where $\sigma_1, \ldots, \sigma_T$ are hyperparameters, ϵ_θ is the diffusion model's denoiser and $\mathbf{z}$ is sampled from $\mathcal{N}(0, I)$. The purpose of the denoiser ϵ_θ, where θ is its tuple of parameters, is akin to its name: it is used to iteratively turn sampled noise $\mathbf{x}_T \in \mathcal{N}(0, I)$ into something resembling a sample $\mathbf{x}_0 \in \mathcal{X}$ from the distribution of interest. If the architecture of its denoiser can be backpropagated through then training a diffusion model can be done in the usual manner of choosing an appropriate loss function and performing gradient descent in which gradients are computed via backpropagation through the entire model. We leave further details of training a diffusion model out for the sake of brevity but these can easily be found in literature.

So, what is an appropriate choice for the architecture of a diffusion model's denoiser? Before considering transformers for this task, we consider a more often-used choice, U-Net: a class of convolutional neural networks (CNNs).

5.5.1 U-Net denoisers

Despite not being the choice of denoiser made in the seminal paper introducing diffusion models, U-Net became the go to choice of denoiser architecture for contemporary diffusion models. To understand U-Net's popularity in this regard, it is worth understanding its architecture which consists of a contracting branch, terminating at its bottleneck, followed by an expansion

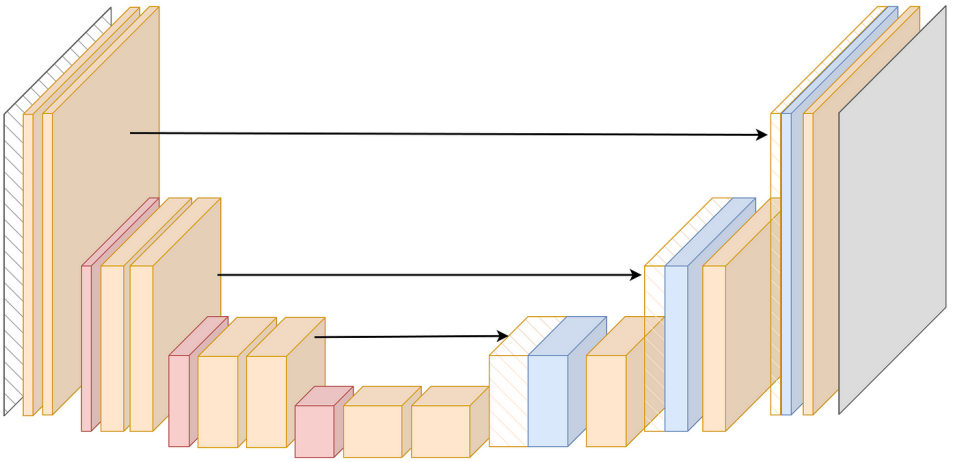


Figure 33: Example U-Net architecture.

branch, illustrated in Figure 33. This architecture can be thought of as forming a 'U'-like shape, with the bottleneck lying at the bottom, hence its name.

In the context of denoising, the contraction branch takes a noised image as input and iteratively completes a series of convolution operations followed by max pooling until reaching the bottleneck layer. At the bottleneck, the model has heavily reduced the spatial dimension of the input image while extracting varying levels of feature abstractions. For example, after the first iteration of convolution and max pooling, the spatial dimension of the input image may be reduced from 1024×1024 to 512×512 but edges and textures within the original image may be encoded in the feature maps at this stage. Then, during expansion, the model looks to upscale from the bottleneck in a way that retains the underlying image while removing noise. This is done by iteratively completing a series of deconvolutions, which correspond to upscaling, and using skip connections from its corresponding component in the contraction branch, seen in Figure 33, in order to retain the underlying image. With this in mind, it is clear why U-Net has been the go to choice of denoiser when developing a diffusion model.

5.6 NEXT: Evaluating Generative Models

Evaluating a supervised learning model is simple enough: leave aside a validation/test set and see how well prediction is done by the model. For

generative models, evaluation is a lot less straightforward, e.g. generated samples are not labeled, ...? Also, what might under/overfitting mean for a generative model? Is the distribution of classes of samples representative of the underlying distribution or do they concentrate on an over-represented subset of classes?

5.6.1 Density estimation

May correspond to evaluations of the learned joint probability function $p_\Theta(\mathbf{x})$.

5.6.2 Sampled images

...

5.6.3 Sampled text

...

Appendices

A Probability Theory & Statistics Things

Here are some probability theory and statistics things relevant to the main body, e.g. brief statements, long derivations, and so on, which are more appendix-appropriate.

I wish my understanding of probability theory, measure theory, stochastic processes and (classical/Bayesian/asymptotic) statistics were better. In another life.

A.1 Jensen's Inequality

If $f : \mathbb{R} \to \mathbb{R}$ is convex and $\mathbb{E}[\mathbf{X}] < \infty$ then

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

If f is concave then

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

I've never come across a particularly inciteful proof, so I won't include one.

The way I remember which way around the inequalities go is by considering the univriate case with $f(x) = \log(x)$. We have $\log(1 + 1) > \log(1) + \log(1)$ and log is concave, from which we recall that

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

for f concave. Other way around for f convex.

A.2 Entropy and its Friend KL-divergence

The entropy of a distribution can be motivated by the notion of the surprise of (or information learned from) observing samples drawn from it. Given a discrete random variable $\mathbf{X}$, an event $\mathbf{x} \in \Omega_{\mathbf{X}}$ and a surprise function $S : \Omega_{\mathbf{X}} \to [0, \infty)$, the surprise of observing $\mathbf{x}$ is $S(\mathbf{x})$. Before continuing, it's useful to lay out what we want out of our surprise function. Following the use of 'surprising' in day-to-day communication, we want events with low probability to be highly surprising and events with high probability to be unsurprising, with some additional natural conditions. For example, if $p(\mathbf{x}) = 0.01$ then we might want $S(\mathbf{x})$ to be relatively high (strictly speaking it doesn't need to be bounded above) and if $p(\mathbf{x}) = 0.99$ then we might want $S(\mathbf{x})$ to be close to 0.

An easy way to achieve this is to take $S(\mathbf{x}) = -\log(p(\mathbf{x}))$ where log denotes the natural logarithm unless stated otherwise. Quickly see that $\log(0.01) = 4.61$ and $\log(0.99) = 0.01$. From here, we define the entropy of the distribution p as its expected surprise

$$H(p) = \mathbb{E}_{\mathbf{X}\sim p}[-\log(p(\mathbf{X}))] = -\sum_{\mathbf{x}\in\Omega_{\mathbf{X}}} p(\mathbf{x})\log(p(\mathbf{x})).$$

More precisely, Claude Shannon wanted such a surprise function to satisfy three intuitive properties. Firstly, the surprise of an event with probability 1 should be 0. Secondly, the surprise invoked by the occurence of independent events should be the sum of the surprise invoked by the individual events. Thirdly, less probable events should be more surprising. In maths speak, we desire

1. $p(\mathbf{x}) = 1 \implies S(\mathbf{x}) = 0$
2. $p(\mathbf{x}_1, \mathbf{x}_2) = p(\mathbf{x}_1)\cdot p(\mathbf{x}_2) \implies S(\mathbf{x}_1, \mathbf{x}_2) = S(\mathbf{x}_1) + S(\mathbf{x}_2)$
3. $p(\mathbf{x}_1) > p(\mathbf{x}_2) \implies S(\mathbf{x}_1) < S(\mathbf{x}_2)$

It's straightforward to see that $S(\mathbf{x}) = -\log(p(\mathbf{x}))$ satisfies these three properties but it turns out that it is unique in satisfying these properties, up to its base.

In literature, entropies are measured in nats (natural units of information) if the natural logarithm is used for their computation and bits if $\log_2$ is used.

Technical note regarding $\mathrm{dom}(S)$

It's clear from the second condition that S is actually a function whose domain is the powerset of $\Omega_{\mathbf{X}}$ but accommodating this detail isn't worth it — the idea conveyed is clear despite the informality.

A.2.1 Kullback-Leibler Divergence (KL-divergence)

Given probability density/mass functions p and q on the same space $\Omega_{\mathbf{X}}$, their Kullback-Leibler divergence (KL-divergence) is given by

$$D_{\mathrm{KL}}(p||q) = \mathbb{E}_{\mathbf{X}\sim p}\left[\log\left(\frac{p(\mathbf{X})}{q(\mathbf{X})}\right)\right].$$

To show that it is non-negative, employ Jensen's inequality to see that

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \mathbb{E}_{\mathbf{X}\sim p}\left[\log\left(\frac{p(\mathbf{X})}{q(\mathbf{X})}\right)\right] \\ &= -\mathbb{E}_{\mathbf{X}\sim p}\left[\log\left(\frac{q(\mathbf{X})}{p(\mathbf{X})}\right)\right] \\ &\geq \log\left(\mathbb{E}_{\mathbf{X}\sim p}\left[\frac{q(\mathbf{X})}{p(\mathbf{X})}\right]\right) \\ &= -\log(1) \\ &= 0. \end{aligned}$$

It is most often interpreted as a measure of similarity between two distributions and it is used as a loss function when assessing how well a model q encodes an underlying distribution p. In many cases, loss functions can be expressed as KL-divergences which can help to motivate the use of said loss function.

The KL-divergence of two distributions can be expressed in terms of a self-entropy and a cross-entropy as in

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \mathbb{E}_{\mathbf{X}\sim p}\left[\log\left(\frac{p(\mathbf{X})}{q(\mathbf{X})}\right)\right] \\ &= \mathbb{E}_{\mathbf{X}\sim p}\left[-\log(q(\mathbf{X}))\right] - \mathbb{E}_{\mathbf{X}\sim p}\left[-\log(p(\mathbf{X}))\right] \\ &= H(p,q) - H(p) \end{aligned}$$

where $H(p,q)$ denotes the cross-entropy between p and q and $H(p)$ denotes the self-entropy of p. It follows that minimising the KL-divergence $D_{\mathrm{KL}}(p,q)$ in q corresponds to minimising the cross-entropy $H(p,q)$.

Example: fitting via cross-entropy/KL-divergence

Let $K \sim \mathrm{Geo}(1/2)$ and denote its probability function by $p : \mathbb{N} \to [0,1]$. Suppose we'd like to fit p via a Poisson distribution whose probability function is $q : \mathbb{N} \to [0,1]$. That is, we would like to best approximate $p(k) = 2^{-(k+1)}$ via $q(k) = \frac{e^{-\lambda}\lambda^k}{k!}$ by tweaking the parameter λ.

To find a suitable value for λ, it makes sense to first come up with a metric for how good a given choice is. For this, we can use the cross-entropy of the distributions p and q given by

$$H(p,q) = -\mathbb{E}_{K\sim p}\left[\log(q(K))\right] = -\sum_{k=1}^{\infty} p(k)\log(q(k))$$

which can be seen as the incurred cost of using the model q in place of the true underlying model p. There are a bunch of different ways of describing/interpreting this quantity. The most interesting is perhaps the one related to encodings. Anyway, in our case, if we can find a closed form of this expression in terms of λ then we can look to compute which value of λ minimises it. In line with this, we compute

$$\begin{aligned} H(p,q) &= -\mathbb{E}_{K\sim p}\left[\log(q(K))\right] \\ &= -\sum_{k=1}^{\infty} p(k)\log(q(k)) \\ &= -\sum_{k=1}^{\infty} 2^{-k}\left(-\lambda + k\log(\lambda) - \log(k!)\right) \\ &= \lambda\sum_{k=1}^{\infty}\frac{1}{2^k} - \log(\lambda)\sum_{k=1}^{\infty}\frac{k}{2^k} + \sum_{k=1}^{\infty}\frac{\log(k!)}{2^k} \\ &= \lambda - \log(\lambda) + C \end{aligned}$$

where $C = \sum_{k=1}^{\infty}\frac{\log(k!)}{2^k}$ does not depend on λ. In computing the minmum of $H(p,q)$ in λ we obtain

$$\frac{\partial}{\partial\lambda}H(p,q) = 1 - \frac{1}{\lambda}$$

which yields $\lambda = 1$. So according cross-entropy, the best-fitting Poisson distribution to our geometric distribution is Poi(1). An illustration is offered in Figure 34.

For $\lambda = 1$ we compute a cross-entropy of

$$H(p,q) = 1 + \sum_{k=1}^{\infty}\frac{\log(k!)}{2^k} \approx 2.01567.$$

On its own, this quantity isn't too useful. To get a sense of goodness-of-fit we desire the KL-divergence, i.e. the cross-entropy minus the self-entropy of p. Let's compute said self-entropy:

$$\begin{aligned} H(p) &= -\sum_{k=1}^{\infty} p(k)\log(p(k)) \\ &= -\sum_{k=1}^{\infty} 2^{-k}\log(2^{-k}) \\ &\approx 1.386 \end{aligned}$$

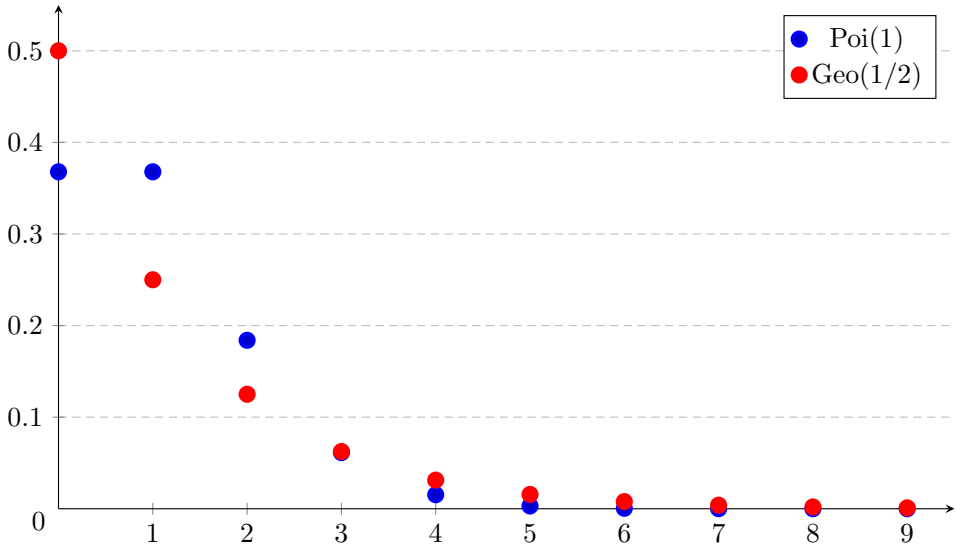


Figure 34: Our geometric distribution p and Poisson fit q.

from which we know that the KL-divergence between the underlying geometric distribution p and our Poisson fit q is given by

$$\mathrm{KL}(p||q) \approx 2.016 - 1.386 = 0.63.$$

Without a point of comparison, the number doesn't tell us much.

A.3 The Expectation-Maximisation (EM) Algorithm

Maximum likelihood estimation (MLE) is great but what do we do when the data generating process is not determined entirely by observed model variables $\mathbf{X} = (X_1, \ldots, X_m)$. That is, what if $p(\mathbf{x})$ is simply the marginal of the true joint $p(\mathbf{x}, \mathbf{z})$ where $\mathbf{Z} = (Z_1, \ldots, Z_{m'})$ are hidden variables?

It turns out that there's a clever way of perfmorming MLE while accounting for hidden variables called the expectation-maximisation (EM) algorithm. It involves obtaining a lower bound for the log-likelihood of a given sample of the observed variables and instead maximising the lower bound in the model's parameters. If certain conditions are met for said lower bound to in fact reach equality with the sample's log-likelihood then the application of EM is MLE itself. If the bound is not exact then we refer to the process as variational EM.

In obtaining a lower bound, let θ denote the parameters of the true joint

$p_\theta(\mathbf{x}, \mathbf{z})$ and see that if q is a probability mass function with domain $\Omega_{\mathbf{Z}}$ then

$$\begin{aligned}
\log(p_\theta(\mathbf{x})) &= \log\left(\sum_{\mathbf{z}\in\Omega_{\mathbf{Z}}} p_\theta(\mathbf{x}, \mathbf{z})\right) \\
&= \log\left(\sum_{\mathbf{z}\in\Omega_{\mathbf{Z}}} q(\mathbf{z})\frac{p_\theta(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})}\right) \\
&= \log\left(\mathbb{E}_{\mathbf{Z}\sim q}\left[\frac{p_\theta(\mathbf{x}, \mathbf{Z})}{q(\mathbf{Z})}\right]\right) \\
&\geq \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(\frac{p_\theta(\mathbf{x}, \mathbf{Z})}{q(\mathbf{Z})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{x}, \mathbf{Z})\right)\right] + H(q) \\
&=: \mathrm{ELBO}(q, \theta)
\end{aligned}$$

in which 'ELBO' stands for evidence lower bound (swap the sums for integrals if you'd like continuously distributed hidden variables). Note that this is not the same ELBO dervied in motivating variational autoencoders (VAEs).

The EM algorithm performs coordinate ascent (component-wise equivalent of gradient ascent) on the ELBO: iteratively maximising $\mathrm{ELBO}(q, \theta)$ in q (with θ fixed) and then in θ (with q fixed) until some convergence crtieria is met. With q fixed, maximising the ELBO in θ amounts to computing

$$\begin{aligned}
\arg\max_\theta \mathrm{ELBO}(q, \theta) &= \arg\max_\theta \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{x}, \mathbf{Z})\right)\right] \\
&= \arg\max_\theta \left(\mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{x}|\mathbf{Z})\right)\right] + \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{Z})\right)\right]\right) \\
&= \arg\max_\theta Q(\theta; q)
\end{aligned}$$

where $Q(\theta; q) = \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{x}|\mathbf{Z})\right)\right] + \mathbb{E}_{\mathbf{Z}\sim q}\left[\log\left(p_\theta(\mathbf{Z})\right)\right]$. Note that equality is offered by the bound precisely when $q(\mathbf{z}) = p_\theta(\mathbf{z}|\mathbf{x})$ and so, with θ fixed, maximising the ELBO in q amounts to computing

$$\arg\max_q \mathrm{ELBO}(q, \theta) = p_\theta(\mathbf{z}|\mathbf{x}),$$

i.e. computing the posterior.

With both statements in mind, the EM algorithm amounts to initialising the parameters $\theta^{(0)}$ and repeating the following two steps from $t = 0$ until stopping criteria is met:

1. (E step: $\theta^{(t)} \to q_t$) Obtain a closed form for $q_t(\mathbf{z}) = p_{\theta^{(t)}}(\mathbf{z}|\mathbf{x})$ and in turn a closed form for $Q(\theta; q_t)$
2. (M step: $q_t \to \theta^{(t+1)}$) Compute $\theta^{(t+1)} = \arg\max_\theta Q(\theta; q_t)$

Variational EM

Computing the posterior exactly is often infeasible in practice. Varational EM seeks to alleviate this by instead restricting q to a family of functions $\mathcal{F}$, e.g. some family of MLPs, and replacing the E-step with the computation

$$q_t = \arg\max_{q \in \mathcal{F}} \text{ELBO}\left(q, \theta^{(t)}\right).$$

Of course, if the posterior $p_{\theta^{(t)}}(\mathbf{z}|\mathbf{x})$ belongs to $\mathcal{F}$ for all relevant $t \in \mathbb{N}$ then variational EM is simply EM.

Regarding the correctness of the EM algorithm, we seek to show that

$$\log(p_{\theta^{(t+1)}}(\mathbf{x})) \geq \log(p_{\theta^{(t)}}(\mathbf{x}))$$

for all $t \in \mathbb{N}$, i.e. individual steps in the loss landscape are neglible at worst and otherwise increase the likelihood of the observed data. Showing this turns out to be elegant, as in

$$\log(p_{\theta^{(t+1)}}(\mathbf{x})) \geq \text{ELBO}\left(q_t, \theta^{(t+1)}\right) \geq \text{ELBO}\left(q_t, \theta^{(t)}\right) = \log(p_{\theta^{(t)}}(\mathbf{x})).$$

The leftmost inequality is due to how the ELBO is defined. The second inequality is seen from

$$\theta^{(t+1)} = \arg\max_\theta \text{ELBO}\left(q_t, \theta\right).$$

Finally, the equality follows from the 'matching the true posterior with q_t' argument offered earlier, i.e.

$$q_t(\mathbf{z}) = p_{\theta^{(t)}}(\mathbf{z}|\mathbf{x}) \implies \text{ELBO}\left(q_t, \theta^{(t)}\right) = \log(p_{\theta^{(t)}}(\mathbf{x})).$$