Machine Learning Explainers

Last updated: September 7, 2025

Abstract

Summaries of some 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, if it demanded far less rigour. 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 era of deep learning began in 2012 and the mysteries of why its methods are so effective. At their core, many methods in machine learning are statistically-principled, corresponding to 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 are those which fit functions $f_\theta : \Omega_{\mathbf{X}} \to \Omega_{\mathbf{Y}}$ from model variables $\mathbf{X} = (X_1, \ldots, X_q)$ to output variables $\mathbf{Y} = (Y_1, \ldots, Y_r)$ (often $r = 1$ in which case we write $\mathbf{Y} = Y$) using concrete examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ of what the to-be-learned function might yield as output $\mathbf{y} = f_\theta(\mathbf{x})$ for a given input $\mathbf{x}$. These model and output variables can be continuous or discrete in nature, or a mix.

An example of a task for which supervised learning is appropriate is regression. Regression tasks can be thought of as those in which the output variables $\mathbf{Y}$ are continuous. Other than regression, the other vanilla supervised learning task is classification, in which $\mathbf{Y}$ are discrete. A well known example of binary classification is whether or not a given passenger survived the titanic wreck given their ticket class, sex, age, port of embarkation and a ton more. The bread and butter example of multi-class classification is handwritten digit recognition.

Other than the types of tasks which we consider in this section, it's worth mentioning how we deal with our data in general. In training a model on some dataset, we need some idea of how well the model generalises to new data. This is done by splitting the given dataset D into a training set D_{train} and a testing set D_{test}. How exactly this split is done depends on the task at hand but an example is 50/50 uniformly randomly splitting the original dataset D. It can be important that this is done uniformly randomly, e.g. for proper representation of class labels during training. Similar idea for regression tasks but with ensuring a reasonable balance in representation of values belonging to given intervals corresponding to the output space. Think of approximating a piecewise linear function but training only on a region in which it is linear. As long as their is sufficient representation of $\Omega_{\mathbf{Y}}$ then splitting D unifromly randomly is no issue. With a reasonable split we train the model on D_{train}. It is all the model knows — D_{train} is the model's universe. After training, we measure the model's ability to generalise to new samples by seeing how well it performs on D_{test}. Essentially comparing its output given some sample to the ideal/true output value of said sample.

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

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

are the feature values of the i^{th} sample and $y_i \in \Omega_Y \subseteq \mathbb{R}$ is the corresponding output. The reason I chose $|D| = 2n$ is that it results in being able to 50/50 split D into D_{train} and D_{test} with $|D_{\text{train}}| = |D_{\text{test}}| = n$. A linear regression models fits a linear function (in the parameters)

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

where $\theta = (\theta_0, \theta_1 \ldots, \theta_q) \in \mathbb{R}^{q+1}$ are the model parameters, i.e. the values we can tweak to our heart's desire until the corresponding model is sufficiently well according to some metric. 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 acrually refer to?

Casella Berger, as well as other pieces of literature, define linear regression in a way that ensures linearity in parameters. By such a definition, linear regression, as a term, encompasses polynomial regression models and other regression models with non-linear basis functions. This confuses me as '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 for a linear regression model, which we denote by θ^*, is to split $\mathcal{D}$ 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

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

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. 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

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

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. 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, 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

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

and $x_{i,j}$ denotes the jth element of the i^{th} sample. To find the minimiser(s) of MSE(θ), i.e. the optimal parameters θ^*, we compute its gradient with respect to θ, set it to 0 and solve for θ^*. 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 MSE(θ) 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 answer[a] based on a paper[b] published in Nature. The top comment 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} for the parameters of our linear regression model is that which maximise our log-likelihood. That is,

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

As such, the maximum likelihood estimator of the parameters of a 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/multiple (one predictor/multiple predictors) correlation coefficient between Y and $f_\theta(\mathbf{X})$. So for one predictor, i.e. $q = 1$, said Pearson correlation coefficient is given by

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

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

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

where s_e^2 and s_Y^2 denote the sample variances of the residuals and Y respectively. I prefer the ratio of explained variance to underlying model variance though; is more intuitive to me, especially in terms of how its motivated and doesn't feel like it comes outta nowhere. Note that 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 via ANOVA tests.

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

2.1.3 Why call it 'regression'?

Nowadays, regression models are those which predict a value belonging to some continuous space but from where did the term originate? In 1886, Francis Galton authored "Regression Towards Mediocrity in Hereditary Stature" which is where the 'regression towards the mean' term comes from. Loosely speaking, Galton noticed that tall fathers tend to have short sons, short fathers tend to have have tall sons 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 $E[Y|X] = \rho X$ 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. one has regression towards the mean.

To see why this does not apply to all modern 'regression' models, consider logistic regression in which one, roughly speaking, predicts probabilities of binary outcomes. In such cases, the dependent variable is binary, so 'regressing towards the mean' has no meaningful interpretation.

The 'linear' part refers to the output variable being linear in the parameters. This is sometimes confusing in linear regression we typically deal with basis functions that give line-like surfaces (lines, planes, etc.) but we could always have non-linear basis functions. For example, $y = \beta_0 + \beta_1 e^x$ is linear in $\beta = (\beta_0, \beta_1)$ but $y = \beta_0 + e^{\beta_1} x$ is not. A consequence is that an estimate $\hat{\beta}$ of the model parameters can be written as $\hat{\beta} = \sum_{i=1}^{k} w_i y_i$ where w_i are the determined weights and y_i are the chosen basis functions.

What makes MSE so special in general?

Differentiability is a common but unsatisfying answer to this question. Exponents of $2.05, 3$ or π would yield a differentiable loss function but we never see those, so what's up? A more satisfying answer is given by this answer to a Stack Exchange[a]:

> The variance is the mean squared deviation from the average. The variance of the sum of 10,000 random variables is the sum of their variances. That doesn't work for other powers of the absolute value of the deviation. That means if you roll a die 6,000 times, so that the expected number of 1s you get is 1,000, then you also know that the variance of 1s is $6000 \cdot \frac{1}{6} \cdot \frac{5}{6}$ so if you want the probability that the number of 1s is between 990 and 1,020, you can approximate the distribution by the normal distribution with the same mean and variance. You couldn't do that if you didn't know the variance, and you couldn't know the variances without additivity of variances, and if the exponent were anything besides 2, then you don't have that. (Oddly, you do have additivity with cubes of the deviations, but not cubes of the absolute values of the deviations).

[a] `https://math.stackexchange.com/a/63245`

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 $(0, 1)$ in some way and imposing a threshold-based decision rule to output either 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 $(q+1)-$dimensional space is the set of points $(x_1, \ldots, x_q) \in \mathbb{R}^q$ which satisfy

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_q x_q = 0$$

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

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

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 $C_\theta^\delta(\mathbf{x}) = \mathbb{1}(h_\theta(\mathbf{x}) > \delta)$.

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

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

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 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. $\mathbb{R}^{q+1}$, 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 $\theta_+ = (\theta_1, \dots, \theta_q)$ 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

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

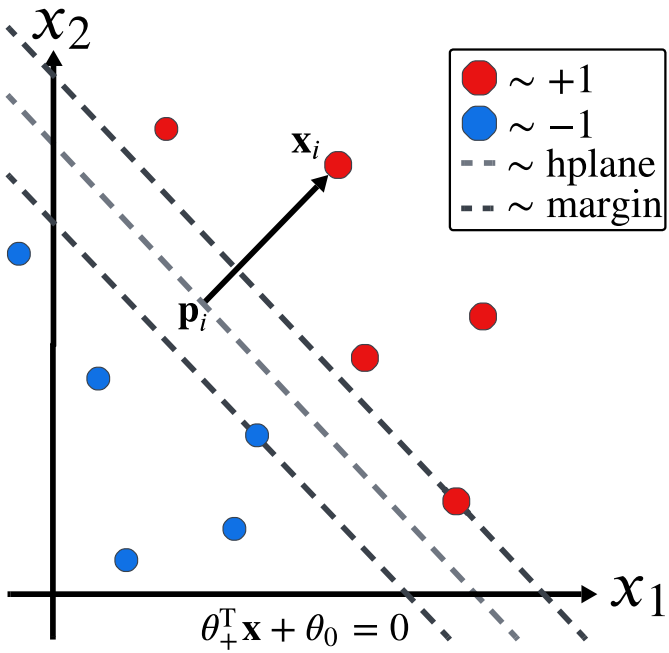


Figure 1: A hard-margin SVM in two dimensions. Samples lying on the margin are referred to as support vectors.

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.

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.

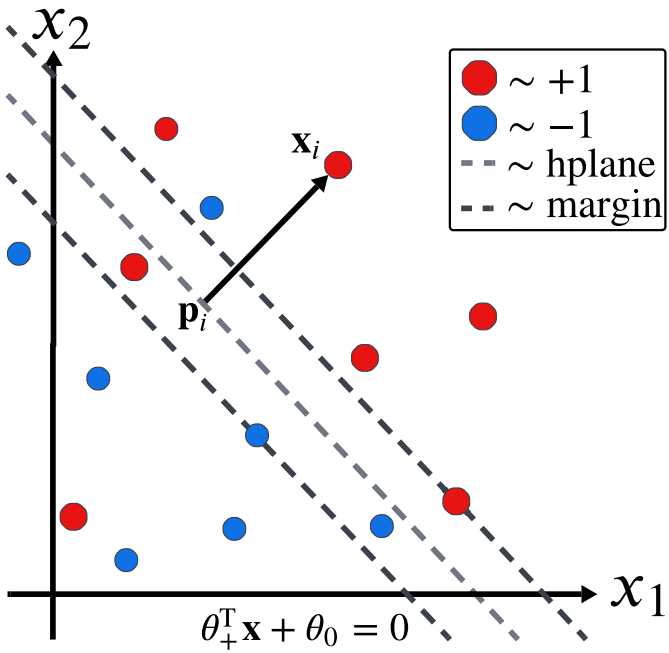


Figure 2: A soft-margin SVM in two dimensions. Samples are labelled according to their true class.

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

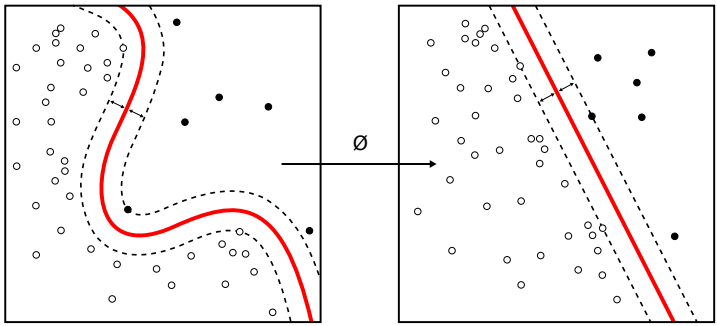


Figure 3: Non-linearly separable samples being transformed in such a way that they become linearly separable. Almost like punching the curve into a line. I'm unsure why the empty set symbol above the arrow is present. **TODO**: Make own figure.

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 RBF kernel. **TODO**: Complete a paragraph on common kernels.

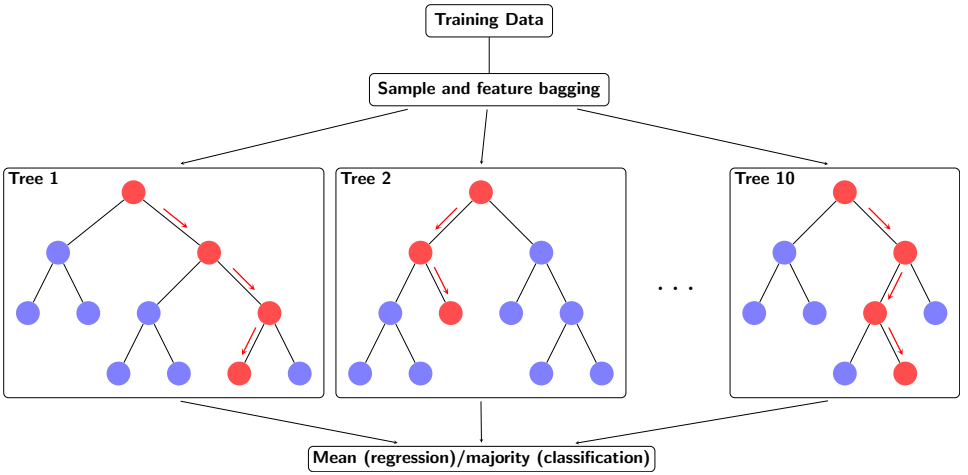


Figure 4: A random forest.

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 include a plot 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 write this subsection properly.

LR > SVM/RF when?

This is a pretty natural question once shown more sophisticated methods which deal with linearly separable and non-linearly separable data. The largest benefit is that it allows for statistical tests on parameters, e.g. statistical significance tests. It also helps that its implementation and interpretation are straightforward.

2.5 REVIEW: The Bias-Variance Tradeoff

The bias-variance tradeoff is a statement pertaining to the goodness of a learning method with respect to its architecture and the data. Ideally, given a fixed architecture, its post-training parameters would vary somewhat with respect to the training data but not too much. This idea leads naturally to underfitting and overfitting.

2.5.1 TODO: Underfitting and Overfitting

Crudely put, if the model family used to fit is too simple to represent the underlying distribution to which the samples belong then the learning method is highly-biased, i.e. the outcome of the learning method is almost invariant with respect to the training data. This is underfitting. Think of fitting a line to a high-degree polynomial.

At the other extreme, if the model family used is too complex (e.g. overparameterised) then the excess parameters often[1] cause the model to fit the noise present in the training data as opposed to the underlying structure. In this case, the outcome of the learning method varies heavily with respect to the training data. Think of polynomial regression models trained on noisy samples.

Finding rigorous definitions of both

I was unable to find rigorous definitions in books like Casella Berger but The Cambridge Dictionary of Statistics offers them. That said, in practice, over and underfitting are often spoken about in terms of the training and validation losses observed during training for a given problem.

2.5.2 Derivation with MSE loss

Recall that sample targets pertaining to datasets from which supervised models are learned are 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

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

We seek to learn an estimate $\hat{f}_D$ of $f(\mathbf{x}) = \mathbb{E}_Y\left[Y|\mathbf{X} = \mathbf{x}\right]$ from the dataset $D \subset (\Omega_{\mathbf{X}} \times \Omega_Y)^n$. In other words, the model/estimate $\hat{f}_D$ is determined by

[1] See the double-descent phenomenon in Figure 6.

the realisation D of $\mathcal{D}$ which is distributed according to $p(\mathbf{x}, y)^{\otimes n}$ for some fixed n. As is common in supervised learning contexts, we make use of the expected risk of an estimate $\hat{f}_D$ which is given by

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

in which mean square error (MSE) is used as loss. The quantity pertaining to the quality of a learning procedure 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

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

in which the tower property of conditional expectations

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

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

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

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

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

This is simply the expected noise, e.g. due to imperfect calibration in the instruments used to obtain samples. It is simply denoted by σ^2.

For 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 $\mathbf{x} \in \Omega_{\mathbf{X}}$ and so

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

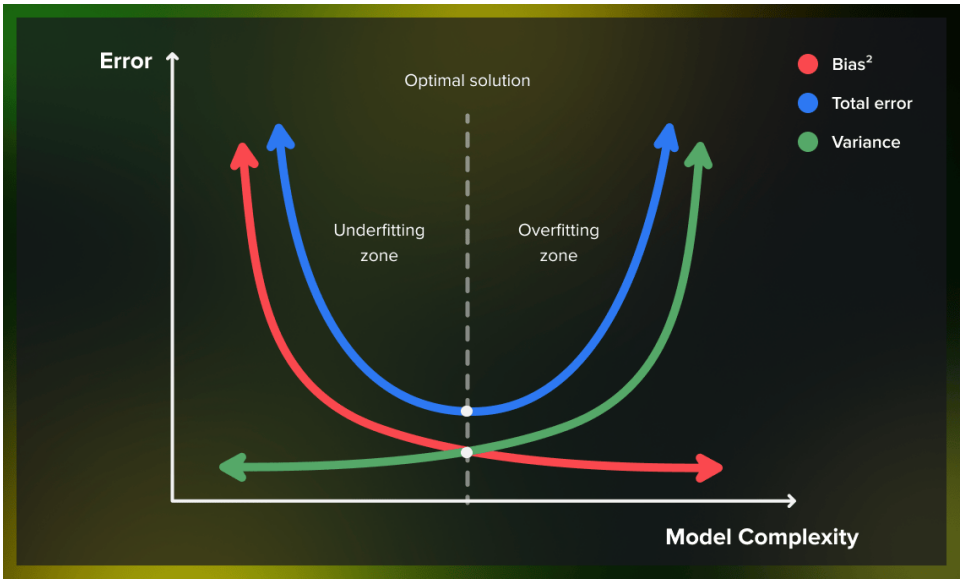


Figure 5: The graph of learning method error against model complexity in terms of bias and variance. **TODO**: Make own figure.

Put together, we obtain

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[R\left(\hat{f}_{\mathcal{D}}\right)\right] \\
&= \mathbb{E}_{\mathbf{X}}\left[\underbrace{\left(f(\mathbf{X}) - \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{X})\right]\right)^2}_{\text{square of bias of estimator}} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathcal{D}}(\mathbf{X}) - \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{X})\right]\right)^2\right]}_{\text{variance of estimator}}\right] + \underbrace{\sigma^2}_{\text{noise}}
\end{aligned}$$

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 the learning method should not seek to minimise square bias or variance individually. Instead, it should seek to find a sweet spot in minimising their sum. An illustration of this idea is given in Figure 5.

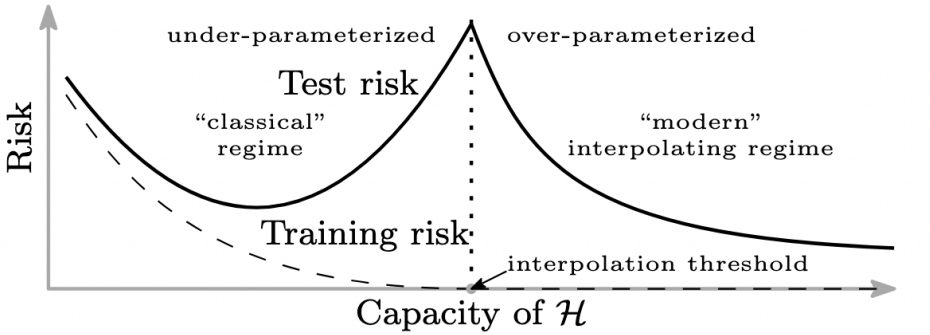


Figure 6: How double descent typically looks. **TODO**: Make own figure.

Beyond MSE

Natural question: MSE is a common choice of loss when dealing with regression but not classification, in which cross-entropies are used for loss, so why does every derivation of the bias-variance tradeoff only use MSE? The quick answer is that it's the only choice which yields a relatively simple derivation of this nice decomposition of learning method error into two orthogonal characteristics of the learning method: bias and variance. It turns out that if you instead consider specifically probabilistic classification methods then using expected cross-entropy as loss yields a decomposition into bias and variance but with a different form.

2.5.3 Double descent

Double descent refers to a behaviour observed in select set ups in which over-parameterised models seemingly fly in the face of the bias-variance tradeoff. As the number of parameters (i.e. the model complexity) increases beyond a certain point, the test error begins to drop. What makes this a phenomenon is that it seems that said threshold is the number of samples used to train the model. This is illustrated nicely in Figure 6.

I like to imagine fitting a third degree polynomial by a polynomial of higher and higher degree. One would expect overfitting with a degree as low as 9. If double descent were observed in this set up, one might expect the learned polynomial to act wildly (as expected) until reaching a degree equal to the number of training samples. At which point, the model begins to fit

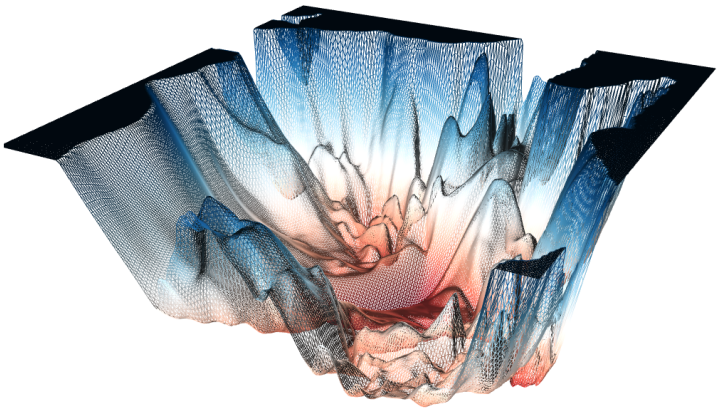


Figure 7: The graph of a complex loss landscape $\{(\theta, R_D(\theta))|\theta \in \mathbb{R}^2\}$ whose minimiser we seek. **TODO**: Make own figure.

the third degree polynomial better and better until it truly fits something very close to a third degree polynomial. Strange.

I'm unaware of anyone who has a good idea of why double descent is sometimes observed. The most intuitive guess I've heard is that gradient descent, as a process, is inherently regulatory.

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 $\mathcal{L}\left(y, f_\theta(\mathbf{x})\right)$. In line with this, let the empirical risk R_D over a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ be

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

In fitting our paremtric model 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, of which many exist.

Tons of 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 machine learning, hence the inclusion of this section.

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 gradient descent, don't necessarily correspond exactly with $R_D(\theta)$ but this isn't usually an issue.

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 minimising $f(\mathbf{x})$. 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 should reach the bottom of the mountain, i.e. a minima 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 $f : \mathbb{R}^q \to \mathbb{R}$ then its second order Taylor expansion about $\mathbf{a} \in \mathbb{R}^q$ 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 $\mathbf{v}$ such that $f(\mathbf{x})$ increases most locally in the direction of $\mathbf{v}$, i.e. we seek to compute

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

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

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

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 symmetries hold then parameters change identically), points of large magnitude (exploding gradients) and points of small magnitude (vanshing gradients). Also, if the function in question is convex then convergence guarantees are held. That said, empirical risk functions are not convex in practice. Simon Price offers an interesting point regarding some quirks of parameter initialisation around 01:08:00 in one of his YouTube interviews.

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

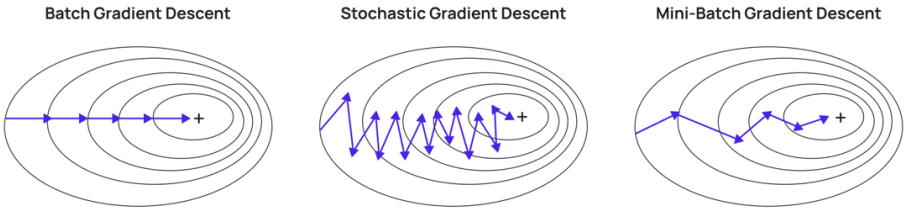


Figure 8: How we might expect the convergence of each method to look. **TODO**: Make own figure.

3.1.3 Batch learning

If your training dataset is huge and you don't have enough memory to store things pertaining to your gradients: 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 in line with being shown the training data. In practice, the model is often shown the data multiple times. Each full cycle of the model having learned from the training data is called an epoch.

1) Full-batch updates

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 updates

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 split into m disjoint subsets $B_1, \ldots, B_m$, referred to as batches, and m updates are made to the parameters 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)

Turns out that doing mini-batch with $m = n$ (so one sample per update), as in

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

for $j = 1, \ldots, n$ is viable. Susprising; I would have thought that it'd produce nonsense parameters. As such, a full epoch during training using SGD corresponds to n parameter updates. Unsurprisingly, SGD yields relatively unstable convergence, a bit like the steps a drunk man would take in walking down the mountain. It's perhaps unsurprising that mini-batch, and thus SGD, are unbiased but high in variance.

Confusing use of terminology

Beware: some refer to what is called mini-batch here as SGD.

3.1.4 Batch + Layer normalisation

These forms of normalisation typically only apply to learning in neural network settings. There's some piece of advice regarding which you should use when batch number is small; was it to use layer norm?

1) Batch norm

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

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

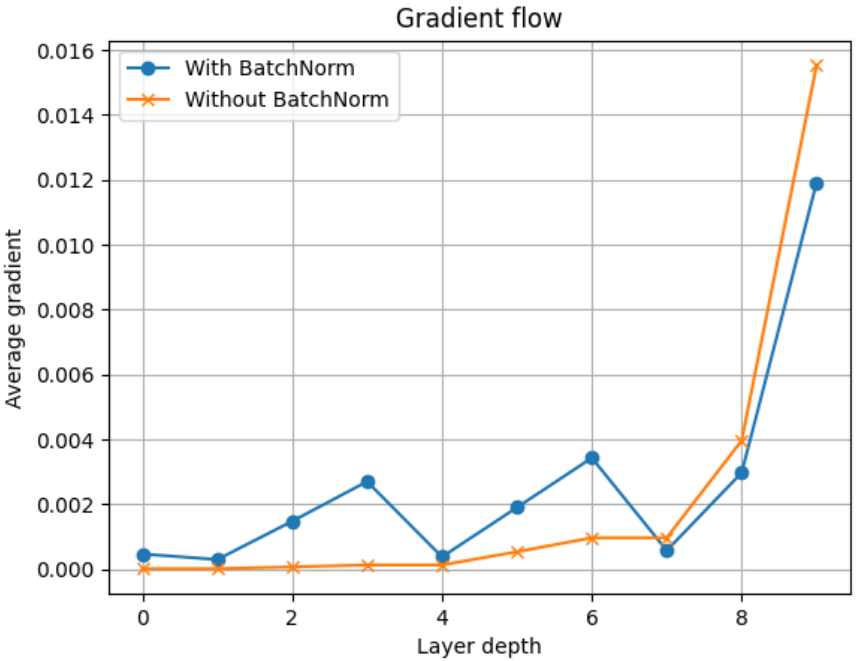


Figure 9: An effect of Batch normalisation. **TODO**: Make own figure.

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 tiny. 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; perhaps helps avoid exploding/vanishing gradients. It was initially thought to reduce something called covariance shift but the consensus has changed and it's now believed that it simply smoothes out the corresponding loss landscape.

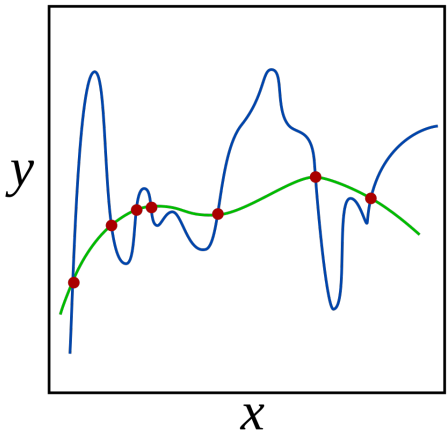


Figure 10: Two fits, both yielding 0 test error. **TODO**: Make own figure.

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 $a_1, \ldots, a_d$, layer norm first normalises the activation values as in

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

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

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

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

3.2 Regularisation

Overfitting is a pain. How might we regulate the learning procedure in a way that seeks to avoid overfitting? Regularisation!

The most well known regularisation techniques 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

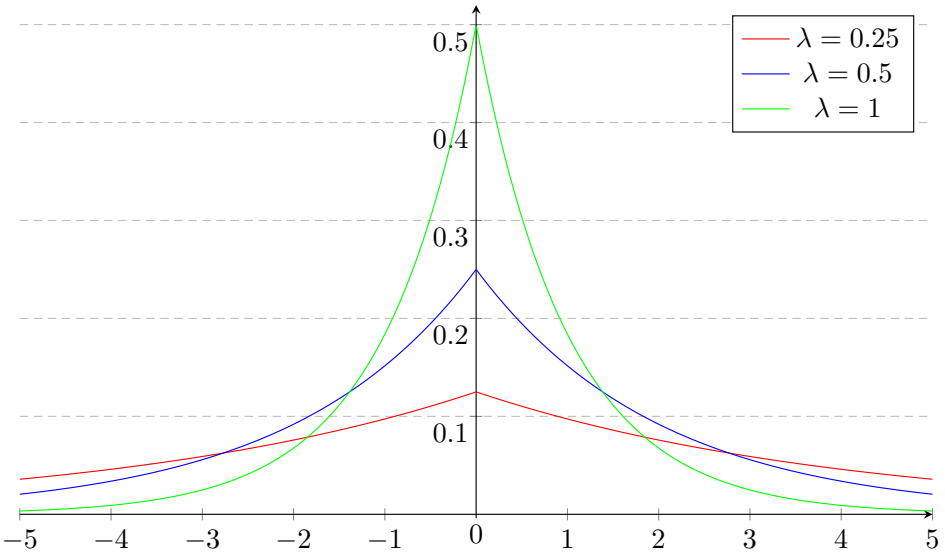


Figure 11: Laplace pdf with means 0 and decreasing variances according to $\lambda \in \{0.5, 1, 2, 3\}$.

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 regular maximum likelihood, compute

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

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 $p(\theta_j) = \lambda \exp(-2\lambda|\theta_j|)$ illustrated in Figure 11.

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 $\theta_1, \ldots, \theta_q \overset{\text{i.i.d.}}{\sim} \text{Lap}(0, 2/\lambda)$, the quantity we seek to compute is given by

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

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. This is achieved by imposing a normal prior and yields L2 regularisation. That is, if $\theta_1, \ldots, \theta_q \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2)$ then the quantity we seek to compute is given by

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

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

Visualising L1 and L2 regularisation

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. After reading the loss function with L1 or L2 regularisation, it's immediately seen how adding either penalty term effectively ....

TODO: make a before and after pic for 3D loss landscape.

It's almost as if you sink every point in the loss landscape according to how far away it is from the origin. The amount by which you sink depends on the hyperparameter λ. It's hopefully clear from the illustration how L1 and L2 reduce variance and increase bias.

3) Early stopping

Split training into training and validation sets. After each epoch (or every few), compute both training and validation losses. If training loss is decreasing while validation loss is not then the model may be overfitting the training data. As illustration, consider Figure 12 in which the training loss continues to decrease in the number of epochs while the validation loss seems to reach its lowest value around epoch 28.

If you observe this for sufficiently many consecutive epochs (the predetermind patience of your early stopping procedure) then you can stop

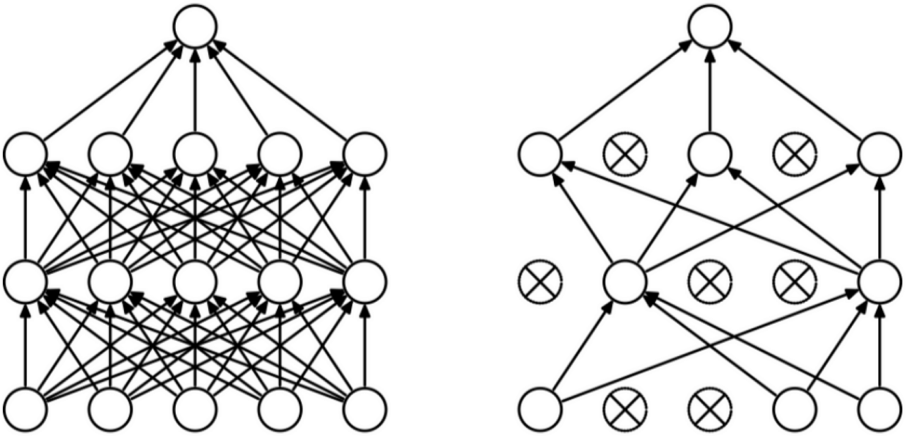
Figure 12: Training and validation losses over 50 epochs for some model. Best validation obtained at epoch 28.

training and use the parameters pertaining to the epoch at which the validation loss was lowest. To do this, at each epoch during training, save the parameters of the current epoch if they offer a lower validation loss than at any epoch before. This is often referred to as a checkpoint within the training process and the parameters are saved in a `.ckpt` file. Alternatively, pick a threshold $\delta > 0$ by which the validation error must have decreased by during the previous however many epochs.

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, during 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 (weights and bias) are not updated during backpropagation. This idea is illustrated in Figure 13.

Make sure to not apply dropout at ineference time!



(a) Standard Neural Net (b) After applying dropout.

Figure 13: Dropout to prevent overdependence on given neurons. **TODO**: Make own figure.

3.3 Momentum + Adaptive Learning Rates

There are some intuitive ideas that accelerate gradient descent. One is momentum, which builds up speed if gradients have pointed in the same direction over multiple steps. The other is an adaptive learning rate, where each parameter has its own step size that adapts over time depending on its past gradients.

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 accelerates. If gradients oscillate, e.g. traversing like a ravine, then momentum dampens the motion.

We impose this 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 switch up the gradient descent update rule to

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

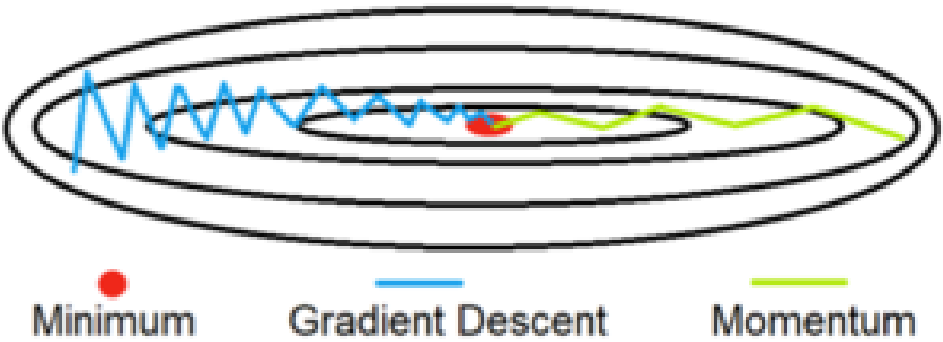


Figure 14: How we might expect the convergence to look with momentum. **TODO**: Make own figure.

where $\mu \in [0, 1)$ is the momentum coefficient. Typically, $\mu \in \{0.9, 0.95, 0.99\}$ are taken. To get a sense of how this refelcts a momentum-like idea, see that $\mathbf{v}^{(1)} = -\eta \nabla R_D(\theta^{(0)})$ and so

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

If $\mu \approx 1$ and the gradients $\nabla R_D(\theta^{(0)})$ and $\nabla R_D(\theta^{(1)})$ are roughly equal then $\mathbf{v}^{(2)}$ will confidently boost $\theta^{(3)}$ in the direction of the first two gradients. If they oppose one another then, as intuition would suggest, $\mathbf{v}^{(0)} = 0$.

2) Adaptive learning rates

Why have a fixed learning rate? Why have the same learning rate for each parameter? Adaptive learning rates looks to solve both. Some intuition to fuel how learning rates might change: if up to the t^{th} parameter update the gradients for a given parameter have been large then it has presumably converged decently so its learning rate would ideally get smaller. The way AdaGrad does this is by storing a per parameter running total of gradients, as in

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

and switch the parameter update to

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

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

3.4 Hyperparameter Tuning

TODO: Review. Mention that the only real reason I included this subsection was to remind myself of cross val Hyperparameters are the tunable parts of our model which are not learned during training. Bad hyperparameters will yield training issues like divergence, overfitting or underfitting. The easiest example of a hyperparameter I can think of is the learning rate in vanilla gradient descent. Some choices of learning rate will yield models which generalise better post-training than other learning rates. For a more sophisticated example, consider architecture-related hyperparameters, e.g. the number of hidden layers in a multi-layer perceptron.

1) Random search

Begin by designating some portion of the training data for validation. Define a finite set of values in which each hyperparmeter may take a value. Since the cardinality of such a set will often be massive, instead of traversing the whole grid of values, consider random search: choose a number of models to train, randomly sample from the finite set for each and choose whichever yielded the lowest validation loss.

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 subsets $D_1, \ldots, D_k$. Then, for $i = 1, \ldots, k$, train a model on $k-1$ of them and validate on the remaining. Average the k validation losses. Choose whichever hyperparameter configuration yields the lowest average validation loss.

More elaborate methods of hyperparameter tuning exist, random search is a straightforward one.

TODO: Why not choose hyperparams which minimise test loss?

4 Neural Networks

It's often the case that someone's intended meaning when stating 'neural networks' is more precisely multi-layer perceptrons (MLPs), perhaps the

simplest class of neural networks beyond single-layer perceptrons. Rigorously formulating MLPs is one of those things that's useful to do at least once; 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 15. 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 the neurons in its preceding 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. 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, MLPs allow for an evaluation of the approximation of the function of interest with 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, 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, note the following:

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

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

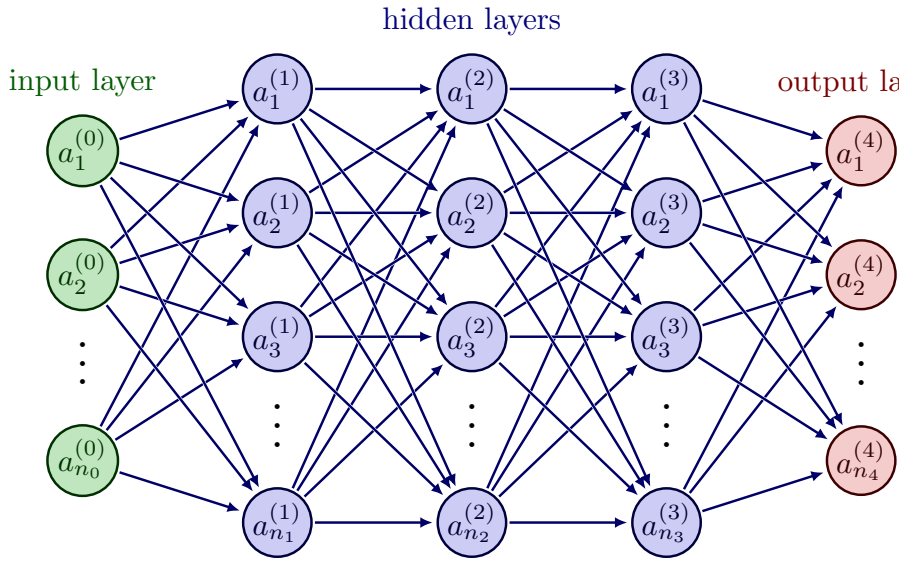


Figure 15: A multi-layer perceptron (MLP) with $k = 3$ hidden layers.

- Let $\sigma_j : \mathbb{R} \to \mathbb{R}$ denote 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.

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 $n_0 = \dots = n_{k+1} = n$ yields a complexity of

$$\mathcal{O}((k+1)n^2),$$

i.e. linear growth in the number of layers and quadratic growth in the number of neurons per layer. A super crude argument for increasing depth over width.

4.1.1 Example

Consider the neural network in Figure 16 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

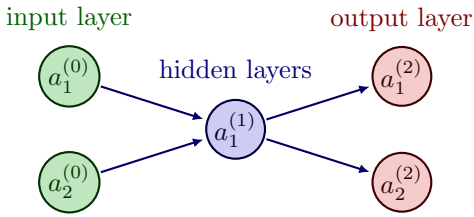


Figure 16: A neural network with one hidden layer ($k = 1$). **TODO**: The figure should read 'hidden layer', not plural.

$n_2 = 2$ 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 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.

4.1.2 The Universal Function Approximation of MLPs

TODO: Add figures. A class of functions, or function class, $\mathcal{F}_S$ is capable of universal function approximation on $S \subset \mathbb{R}^q$ 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 $S \subset \mathbb{R}^q$. 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 $S \subset \mathbb{R}^q$ 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 $\mathbb{R}^q$ and voilà.

Is this idea practical?

What the idea conveyed shows is *that* (and even *how*) an MLP can approximate any function in $C(S)$ for $S \in \mathbb{R}^q$ 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 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.

4.1.3 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

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

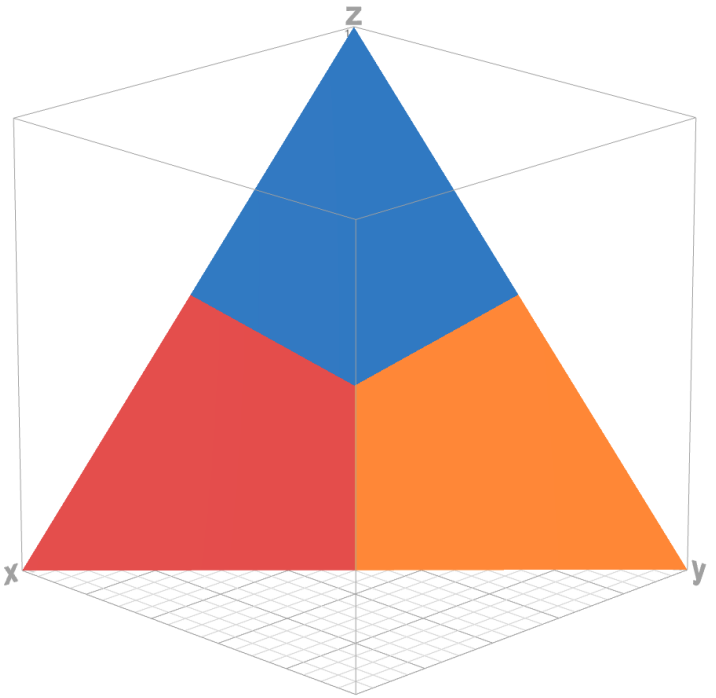


Figure 17: The 2−probability simplex coloured according to argmax in the three coordinates.

The 2−probability simpelx coloured according to argmax is illustrated in Figure 17. From the figure, it is clear how argmax can 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−simplex, as in learning $p(y|\mathbf{X} = \mathbf{x})$.

4.2 Backpropagation for MLPs

Backpropagation is a method of computing the gradient of a loss function pertaining to an MLP (or more generally some neural network) with respect to its weights and biases. Said gradients are used to learn suitable MLP parameters via gradient descent. We'll soon see why it's called backpropagation: it computes the gradients of interest in a way that requires going forwards through the MLP and then backwards. As with formulating MLPs, it's good to rigorously derive backpropagation at least once. 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.

Onto the derivation, which makes use of the notation introduced in Subsection 4.1. 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

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

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

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

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

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

4.3 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) are neural networks whose architectures are designed to process image data. Illustrated in Figure 18, their architecture can be described on a high level as consisting of two parts. The first part performs feature extraction through convolutions and pooling. The second part is an MLP which uses the extracted feature values to produce the overall model's output, e.g. the probability that the input object belongs to a given class. Feature extraction into MLP, pretty intuitive.

Before these deep learning architectures, researchers would have use handcrafted features and feed the feature values to a given model, e.g. an SVM for classification. Humans turn out to be far worse at knowing what features to use than well-trained machine learning models. So how do these earlier layers in CNN architectures perform feature extraction? Convolution and pooling layers. **TODO**: Clarify that Feature maps != Filters/kernel

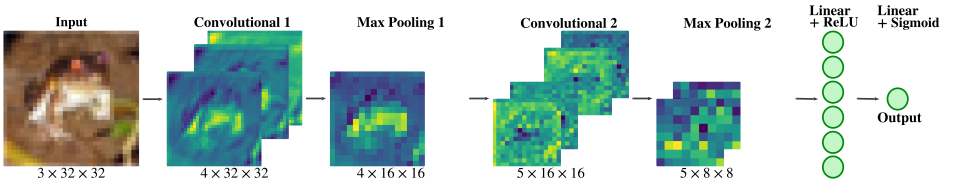


Figure 18: A binary classification CNN processing an image of a frog.

4.3.1 Convolutions

Convolution operations involve using a square matrix to, in some sense, summarise the information embedded in the parts of the imaage which it traverses over. This square matrix is referred to as a filter or a kernel. I prefer referring to it as a filter. You can think of this convolution operation as a sort of dot product between the filter and the grid of pixel values it is on top of at that point.

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 $F = K * I \in \mathbb{R}^{W' \times H'}$ given by the convolution of K over I is given by

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

where b_c is the filter's bias in channel c. 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. For a simpler illustration, consider Figure 19.

Generally, earlier convolutions in a CNN architecture extract higher level features, like edges and textures, while later convolutions extract lower level features which are not human-interpretable. This is observed in Figure 18 where after the first convolutional layer one can still see a resemblance of the input image of a frog. The feature maps produced by the second convolutional layer do not maintain a human-interpretable resemblance of the frog.

4.3.2 Pooling

Pooling operations reduce the spatial dimensions of a given feature map by, in some sense, clustering certain parts of the feature map. There's not much point to writing this mathematically rigorously. Instead, consider Figure 20. Pooling has a ton of flavours: max, min, average and more.

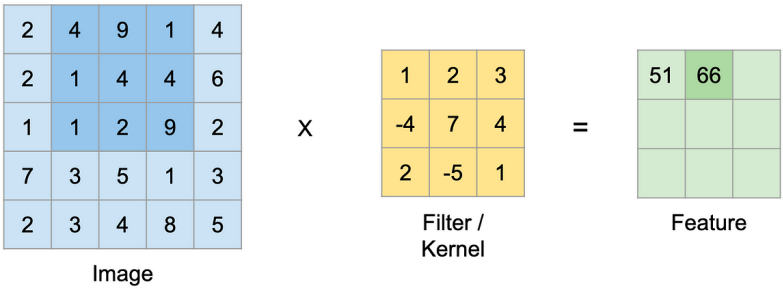


Figure 19: Convolution without bias. **TODO**: Make own figure.

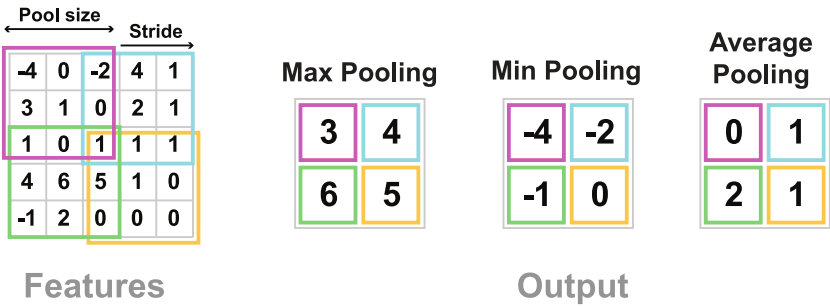


Figure 20: The core flavours of pooling. **TODO**: Make own figure.

A nice benefit of pooling is improved translation-invariance. Ideally, our model would classify an image in a similar enough way to how it'd classify the same image but slightly shifted. What is meant here is very problem-dependent so take it with a grain of salt. Hopefully the idea is clear anyway. Another benefit to pooling the reduced compute.

4.3.3 Output dimensions after convolving and pooling

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. **TODO**: complete.

Why use a CNN over an MLP?

Improved parameter-efficiency, translation invariance and learning-efficiency[a], i.e. fewer samples needed to learn distributions to similar degrees of precision.

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

4.3.4 Inductive biase

Seems like a good point in the document to talk about inductive bias...

4.4 TODO: Recurrent Neural Networks (RNNs)

4.4.1 Long Short-Term Memory (LSTMs)

4.4.2 Gated Recurrent Units (GRUs)

4.5 TODO: Transformers

Just another (very effective) architecture for correlation extraction.

4.5.1 Tokens and Embedding

Tokenisation is the process of slicing a sample up into tokens. Think of splitting a 512×512 image into its four 256×256 quadrants or the sentence 'The chef cooked a meal for the critic.' into

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.

An embedding function $E : \{1, \ldots, |S|\} \to \mathbb{R}^d$, where $d \in \mathbb{N}$ is the embedding dimension, offers high dimensional numeric representations of tokens. What makes token embedding functions interesting is how well they

can capture underlying semantics, e.g. of natural language. For example, an embedding function learned on pieces of text might satisfy

$$E(\text{Germany}) + E(\text{fascist}) - E(\text{Mussolini}) \approx E(\text{Hitler})$$

for which the tokenisations occur.

4.5.2 Positional encoding

Use sinusoids to form a position vector P_k for the kth token in an input sequence of length L. Add P_k to the word embedding E_k for $k = 0, \ldots, L-1$ to obtain the input $[E_0 + P_0, \ldots, E_{L-1} + P_{L-1}]$ fed to the model.

4.5.3 Architecture

...

4.6 Misc. questions

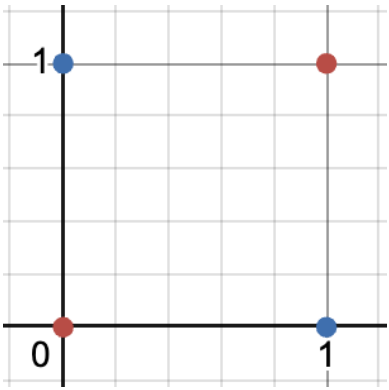
Some questions to challenge one's understanding.

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. Two such cases are illustrated in Figure 21.

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 Figure 22.

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



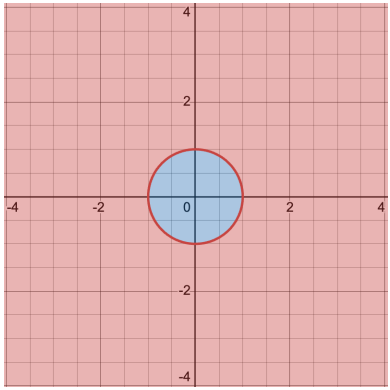


Figure 21: Examples of samples belonging to classes that are not linearly separable. **TODO**: Make properly.

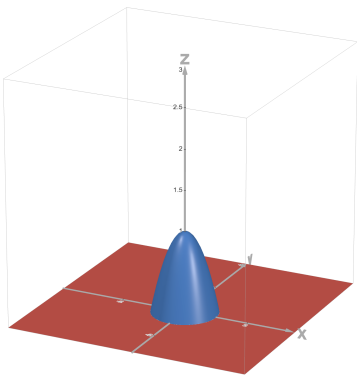


Figure 22: The image of our non-linear transformation. **TODO**: Make properly.

transformation required us to map samples to 3D — how about mapping to just 1D? To do this, a complete decision function is given by

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

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 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 is sometimes undesirable, e.g. if one seeks to minimise the number of hyperparameters for a 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

Worth mentioning that said vanishing behaviour is also somtimes observed in forward passes.

5 Generative Models

How is a mention of modelling the joint not in here? E.g. GPT models sequentially sample from conditional distributions. Almost feels like classification...

TODO: Re-word paragraph and emphasise that it's all about modelling the joint distribution. Also get in how this is what most of the sciences most often do. Also, mention how these approaches are more generally deep generative models but my interests are in vision generative things. Since the era of deep learning began in 2012, tons of generative image, video and text models have arisen. The general idea of these generative models is that if we have a model that fits the data generating process, e.g. the relevant distribution if taking a probabilistic approach, then samples from the model should be sufficiently convincing. That is, if I get a nice fit for the distribution of images of cats lounging in the sun then samples from my fit distirubtion should look convincingly like cats lounging in the sun.

The problem with fitting such a distribution is that it is not at all clear what the distribution function would be. These distributions are complex and generally very hard to get right. Most ways to fit the underlying distribution is to perfrom maximum likelihood in some way. Performing maximum likelihood directly without access to any sort of direct distribution function is pretty hard, so it is often done indirectly. For example, when training a variatioanl autoencoder (VAE), you look to maximise a lower bound of the likelihood. The idea is that if this lower bound is sufficiently tight then maximising it in some sense pushes up the true likelihood, maximising it.

5.1 TODO Bayesian networks

Probabilistic circuits should get an honourable mention

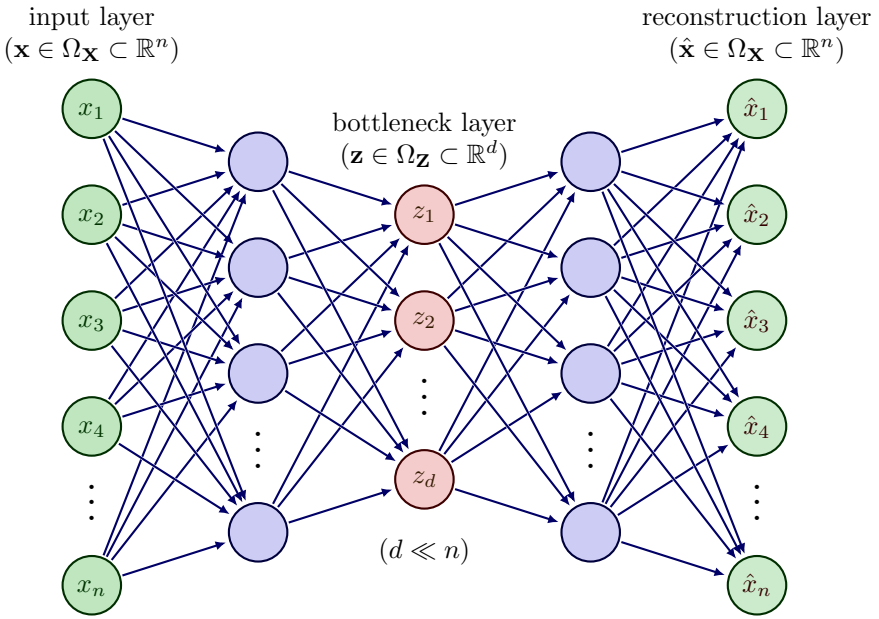


Figure 23: An autoencoder in which θ and ϕ are fit using multi-layer perceptrons (MLPs). **TODO**: Colour missing in text? Also change n to q

5.2 Variational Autoencoders (VAEs)

TODO: re-word whole subsection. In describing variational autoencoders (VAEs), it's natural to first motivate autoencoders and then add some spice in the form of variational inference to be able to do generative things.

5.2.1 Autoencoders: no variation yet!

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

In intuitive terms, one can see the encoder θ 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 θ as the latent space $\Omega_{\mathbf{Z}}$. Similarly, the decoder ϕ 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. $d \ll q$. Of course, when $d \ll q$, learning such mappings θ and ϕ typically involves 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.

Autoencoder example

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 $\theta_2(x, y, z) = (x, z)$. To reconstruct samples from their latent representation (i.e. to decompress) one might employ the decoder $\phi_2(x, z) = (x, x, z)$. The autoencoder $(\Omega_{\mathbf{X}}, 2, \theta_2, \phi_2)$ offers lossless compression on the sample space of interest as $(\phi_2 \circ \theta_2)(\mathbf{x}) = \mathbf{x}$ 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 $\theta_1(x, y, z) = x$ and the decoder $\phi_1(x) = (x, x, x)$. The autoencoder $(\Omega_{\mathbf{X}}, 1, \theta_1, \phi_1)$ offers lossy compression, i.e. some information pertaining to a sample is lost during its compression and decompression as $(\phi_1 \circ \theta_1)(a, a, b) = (a, a, a)$ for all $a, b \in \mathbb{R}$.

One's tolerance for the loss incurred by a given encoder/decoder pair 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 $(\theta^*, \phi^*) \in \Theta \times \Phi$ which minimises the empirical risk over D where Θ and Φ are chosen function families. That is, computing

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

If the function families permit the computation of gradients of $\theta \in \Theta$ and $\phi \in \Phi$ then this computation may be done using gradient descent. For example, one could take Θ and Φ to be the family of MLPs of appropriate input and output dimensions, as illustrated in Figure 23.

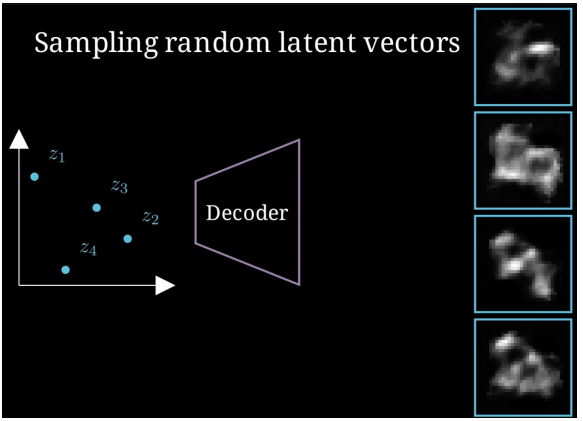


Figure 24: The output of the decoder of an autoencoder with randomly generated latent points as input. Gibberish output. **TODO**: Make own figure.

5.2.2 Motivating VAEs

Autoencoders are great for compression, denoising, disecting LLMs (look up sparse autoencoders), etc. but we're yet to see how we might make use of them for generative tasks, i.e. sampling from the complex underlying distribution p 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 them through the decoder, as in Figure 24. The outputs of the decoder should be similar to 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 (whatever this may mean) results in 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 is 'un-utilised'). This is perhaps unsurprising as at no point during training an autoencoder do we explicitly demand that it interpolates nicely between points.

New idea: how about we feed the decoder latent points that are pretty close to the embedding of a given training sample, as in Figure 25? This should result in decoder outputs that are similar in structure to the input sample but a bit different, right? No.

Distaster strikes again. To get decoder outputs that are meaningful 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

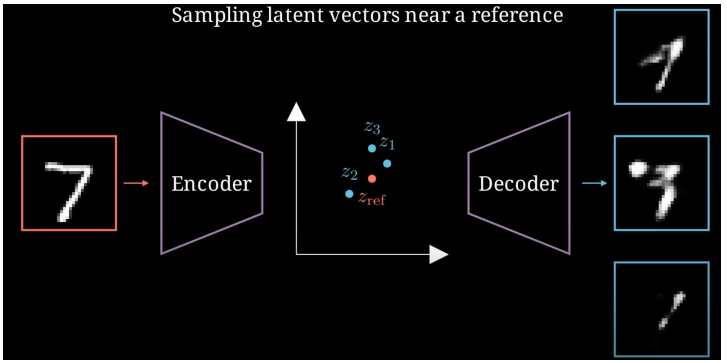


Figure 25: The output of the decoder of an autoencoder points relatively close to the latent embedding of a given training sample. **TODO**: Make own figure.

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

TODO: ensure consistent notation. 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 $(\Omega_{\mathbf{X}}, d, \mathcal{Q}_d, \mathcal{P}_q, \theta, \phi)$ consists of the sample space of the distribution of interest $\Omega_{\mathbf{X}} \subset \mathbb{R}^q$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, the parameter space $\mathcal{Q}_d$ of a family of $d-$dimensional conditional distributions denoted $q_\theta(\mathbf{z}|\mathbf{x})$, the parameter space $\mathcal{P}_n$ of a family of $q-$dimensional conditional distributions denoted $p_\phi(\mathbf{x}|\mathbf{z})$, an encoder $\theta : \mathbb{R}^q \to \mathcal{Q}_d$ and a decoder $\phi : \mathbb{R}^d \to \mathcal{P}_n$.

To illustrate the intended meaning of the newly-introduced parameter spaces $\mathcal{Q}_d$ and $\mathcal{P}_n$, 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 $\mathbb{R}^{d \times d}$.

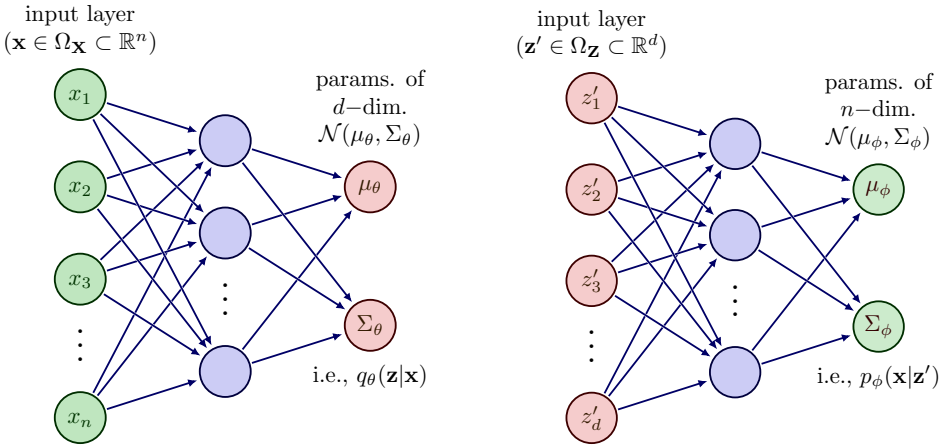


Figure 26: Left: an encoder which outputs parameters of the latent distribution $q_\theta(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\theta, \Sigma_\theta)$. Right: a decoder which outputs parameters of the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}') = \mathcal{N}(\mu_\phi, \Sigma_\phi)$ in which $\mathbf{z}' \sim q_\theta(\mathbf{z}|\mathbf{x})$.

As the encoder of a VAE yields a $d-$dimensional distribution $q_\theta(\mathbf{z}|\mathbf{x})$ 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 $\theta(\mathbf{x}) \in \mathcal{Q}_d$ of $q_\theta(\mathbf{z}|\mathbf{x})$, e.g. a $d-$dimensional Gaussian, via the encoder and samples $\mathbf{z}' \sim q_\theta(\mathbf{z}|\mathbf{x})$. To reconstruct $\mathbf{x}$ from its latent representation $\mathbf{z}'$, one computes the parameters $\phi(\mathbf{z}') \in \mathcal{P}_n$ of the $q-$dimensional reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}')$ via the decoder. Sampling $\mathbf{x}' \sim p_\phi(\mathbf{x}|\mathbf{z}')$ 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 $\mathcal{P}_n$ 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 26.

VAE example

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

$$\Sigma = \begin{bmatrix} 2 & 2 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

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

$$\begin{aligned} \theta : \Omega_{\mathbf{X}} &\to \mathcal{Q}_2 \\ (x_1, x_2, x_3) &\mapsto ((x_1, x_3), \sigma^2 I_2) \end{aligned}$$

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 $q_\theta(\mathbf{z}|\mathbf{x}) = \mathcal{N}((x_1, x_3), \sigma^2 I_2)$. As decoder, if all we desire is accurate reconstructions, we might choose

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

That is, for the latent sample $\mathbf{z}' = (z'_1, z'_2)$, the decoder's output would yield the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}') = \mathcal{N}((z'_1, z'_1, z'_2), \sigma^2 I_3)$. Ideally, sampling $(x'_1, x'_2, x'_3) \sim p_\phi(\mathbf{x}|\mathbf{z}')$ 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 $\mathbb{R}^q$. VAEs seek to remedy this.

To learn a VAE from data, we look to maximise the evidence lower bound (ELBO) over some dataset $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$. 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 ϕ (which parameterises the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z})$), we may express the marginal distribution $p(\mathbf{x})$ using $p_\phi(\mathbf{x}|\mathbf{z})$ as

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z} = \int p_\phi(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}. \tag{1}$$

Using Equation 1, along with the encoder θ (which parameterises the latent distribution $q_\theta(\mathbf{z}|\mathbf{x})$) we derive the ELBO over a single sample $\mathbf{x} \in \Omega_{\mathbf{X}}$:

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

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 Subsection A.7. The effect of maximising

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

over a dataset is often explained term-by-term. 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 $q_\theta(\mathbf{z}|\mathbf{x})$ 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 $\phi(\mathbf{z}')$ of the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}')$ 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 $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$, we learn a VAE by choosing function classes Θ and Φ (e.g. MLPs as in Figure 26) and computing

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

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 $\mathcal{P}_n = \mathbb{R}^n \times \mathcal{S}^n_{++}$, 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[3].

5.2.4 Backpropagation for VAEs

TODO The expectation term in the ELBO requires us to sample from $q_\theta(\mathbf{z}|\mathbf{x})$. Back propping through sampling is not a thing, it ain't differentiable. We need the reparameterisation trick!

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

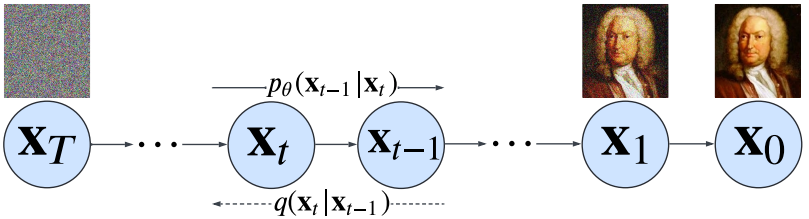


Figure 27: Johann Bernoulli being denoised in line with the backward process of a diffusion model. **TODO**: Improve figure.

5.3 TODO: Generative Adversarial Networks (GANs)

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 $\mathcal{N}(0, I)$.

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

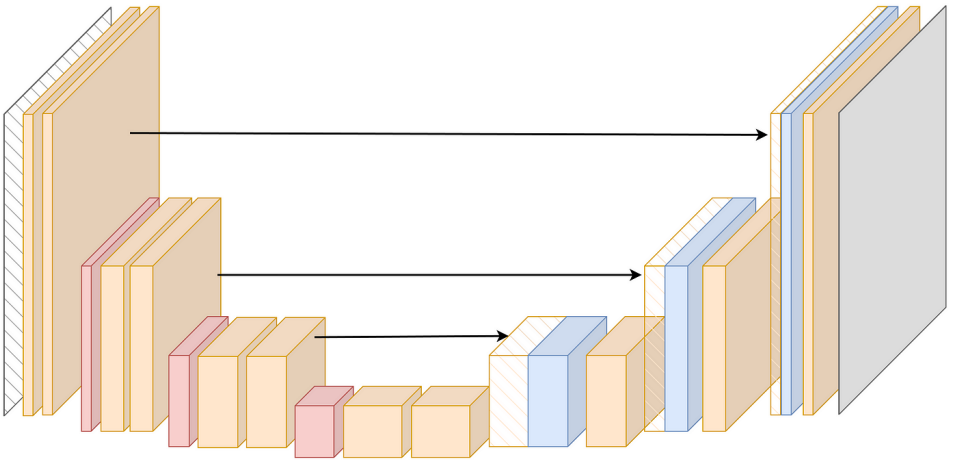


Figure 28: Example U-Net architecture.

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 branch, illustrated in Figure 28. 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 28, 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 TODO: Evaluating Generative Models

Since samples are not labeled, how does one evaluate a generative model? What does under/overfitting mean for generative models?

6 Object Detection Models

Object detection is the task of localisation and classification within an image or video. Before the era of deep learning, it was typically solved using Haar-like filters... As of August 2025, there are three flavours of object detection models: R-CNN, YOLO and DETR.

6.1 TODO: (Fast/Faster) R-CNN

R-CNN models has evolved significantly since its proposal in 2013 with three evolutions worthy of a new name: R-CNN (the original), Fast R-CNN and Faster R-CNN. R-CNN stands for *Regions with CNN features*. Roughly put, R-CNN model consist of two steps:

1. Proposing regions of interest within the image
2. Classifying what is in each region (if there is something in the region)

6.2 TODO: YOLO

6.3 TODO: DETR

Appendices

A Probability Theory and Statistics Things

Here are some probability theory and statistics-related things, e.g. brief statements, long derivations, and so on, which belong in an appendix. I wish I knew more probability/measure theory and classical statistics. In another life.

A.1 Sample Independence

Suppose we'd like to get an idea of the mean population height of men and women. If we randomly sample 100 men and 100 women then we're good to go. This could be done by independently sampling, without replacement, from the categorical corresponding to the ID numbers of individuals. If we instead randomly sample 100 couples then our samples would not be independent since couples' heights correlate.

Sample independence is massively important, e.g. for maximum likelihood estimation. Independence is also assumed in a ton of other statistics-related things like t-tests, ANOVA, etc.

A.2 Derivations Related to Common Distributions

1) From Bernoulli to Binomial

Let's try to extend the Bernoulli distribution to the binomial distribution. If $X \sim \text{Ber}(p)$ then $\Omega_X = \{0, 1\}$ and $\mathbb{P}(X = x) = p^x(1-p)^{1-x}$, so $\mathbb{P}(X = 1) = p$ and $\mathbb{P}(X = 0) = 1 - p$.

Taking n independent Bernoulli random variables and summing them yields the random variable $\mathbf{X} = \sum_{i=1}^{n} X_i \sim \text{Bin}(n, p)$ with $\Omega_{\mathbf{X}} = \{0, \dots, n\}$. Let k denote a realisation of $\mathbf{X}$, i.e. $k = x_1 + \cdots + x_n$. The corresponding mass function is given by

$$\begin{aligned}
\mathbb{P}(\mathbf{X} = k) &= Z \cdot \prod_{i=1}^{n} \mathbb{P}(X_i = x_i) \\
&= Z \cdot \prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i} \\
&= Z \cdot p^{x_1+\cdots+x_n}(1-p)^{n-(x_1+\cdots+x_n)} \\
&= Z \cdot p^k(1-p)^{n-k}
\end{aligned}$$

where Z is a normalising constant such that $\sum_{k=0}^{n} Zp^k(1-p)^{n-k} = 1$. It's more intuitively seen as the number of ways of placing k-many 1s among a series of $n \geq k$ bits, i.e. $Z = \binom{n}{k}$, so

$$\mathbb{P}(\mathbf{X} = k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

2) From Categorical to Multinomial

The categorical distribution is an extension of the Bernoulli distribution and the multinomial distribution is an extension of the binomial distribution. In line with this, we'd hope to be able to extend the categorical distribution to the multinomial distribution.

If $X \sim \text{Cat}(p_1, \ldots, p_C)$ then realisations of X can be represented by single integers in $\{1, \ldots, C\}$ but I prefer to represent them in terms of their $C-$long one-hot encodings. So I denote the presence of the c^{th} class in a realisation as the unit row vector $\mathbf{e}_c^\top$, e.g. $(0, 1, 0 \ldots, 0)$ denotes a sample in which only the second class is present. The corresponding mass function is then given by

$$\mathbb{P}(X = (x_1, \ldots, x_C)) = \prod_{j=1}^{C} p_j^{x_j}.$$

Note that, in this notation, all but one of these x_j terms are zero, so it really boils down to just a single probability value, e.g. $\mathbb{P}(X = (0, 1, \ldots, 0)) = p_2$. The mass function can also be written using indicator functions but the form offered above is the one I find easiest to use.

Applying the same idea used to extend Bernoulli to binomial, consider the random variable $\mathbf{X} = \sum_{i=1}^{n} X_i$ pertaining to n independent categorical trials. Each realisation of $X_1, \ldots, X_n$ can be represented by a $C-$long one-hot encoded vector and so n realisations of $X \sim \text{Cat}(p)$ can be thought of as a matrix $\mathbf{x} \in \{0, 1\}^{n \times C}$ whose rows are unit vectors in $\mathbb{Z}_{\geq 0}^{C}$. As such, letting $k_1, \ldots, k_C$ denote the number of 1s in the columns of $\mathbf{x}$ (so $k_1 + \cdots + k_C = n$) we see immediately that $(k_1, \ldots, k_C)$ is a realisation of $\mathbf{X}$. So what is the corresponding mass function? Let x_{ij} denote the element in the i^{th} row and j^{th} column of $\mathbf{x}$ and let $k_j = x_{1j} + \cdots + x_{nj}$ denote the sum of all elements in the j^{th} column of $\mathbf{x}$. Note that k_j is the number of realisations of $X_1, \ldots, X_n$

in which the j^{th} class is present. We have

$$\begin{aligned}
\mathbb{P}(\mathbf{X} = (k_1, \ldots, k_C)) &= Z \cdot \prod_{i=1}^{n} \mathbb{P}(X_i = (x_{i1}, \ldots, x_{iC})) \\
&= Z \cdot \prod_{i=1}^{n} \prod_{j=1}^{C} p_j^{x_{ij}} \\
&= Z \cdot \prod_{j=1}^{C} p_j^{x_{1j} + \cdots + x_{nj}} \\
&= Z \cdot \prod_{j=1}^{C} p_j^{k_j} \\
&= Z \cdot p_1^{k_1} \cdots p_C^{k_C}.
\end{aligned}$$

So all that's left to do is derive the normalisation constant Z. Note that for each $j \in \{1, \ldots, C\}$ we know that there are k_j-many 1s in column j so k_1 of these n rows must pertain to the first class for which there are $\binom{n}{k_1}$ possible orderings. From here, see that k_2 of the $n - k_1$ remaining rows must pertain to the second class for which there are $\binom{n-k_1}{k_2}$-many orderings. Continuing this line of reasoning, we obtain

$$\begin{aligned}
Z &= \binom{n}{k_1}\binom{n-k_1}{k_2}\binom{n-(k_1+k_2)}{k_3} \cdots \binom{n-(k_1+\cdots+k_{C-1})}{k_C} \\
&= \frac{n!}{k_1!} \cdot \frac{(n-k_1)!}{k_2!(n-(k_1+k_2))!} \cdot \frac{(n-(k_1+k_2))!}{k_3!(n-(k_1+k_2+k_3))!} \cdots \frac{k_C!}{k_C!0!} \\
&= \frac{n!}{k_1! \cdots k_C!}
\end{aligned}$$

from which we obtain

$$\mathbb{P}(\mathbf{X} = (k_1, \ldots, k_C)) = \frac{n!}{k_1! \cdots k_C!} p_1^{k_1} \cdots p_C^{k_C}.$$

3) Negative Binomial and Geometric

We keep flipping our (maybe biased) coin until we observe r successes (r is a parameter) where individual trials are $\text{Ber}(p)$ distribured. So if $X \sim \text{NB}(r, p)$ then

$$\mathbb{P}(X = k) = Z \cdot p^r (1-p)^k.$$

A trial must be of the form $(x_1, \ldots, x_{k+r-1}, 1)$ which means that $r-1$ of the first $k+r-1$ elements must be a success of which there are $\binom{k+r-1}{r-1}$ possibilities, thus

$$\mathbb{P}(X = k) = \binom{k+r-1}{r-1} p^r (1-p)^k.$$

The case $r = 1$ yields the shifted geometric distribution, whose pmf is given by

$$\mathbb{P}(X = k) = p(1-p)^k.$$

4) From Binomial to Poisson

Let $X_n \sim \text{Bin}(n, p)$ and $\lambda = np$. How might X_n look as $n \to \infty$? Why might we be interested in this at all? It can be thought of as ...

To see what X_n converges to in distribution, note that for all $x \in \{0, \ldots, n\}$

$$\begin{aligned} p(X_n = x) &= \binom{n}{x} \left(\frac{\lambda}{n}\right)^x \left(1 - \frac{\lambda}{n}\right)^{n-x} \\ &\approx \frac{n!}{x!(n-x)!} \frac{\lambda^x}{n^x} e^{-\lambda} e^{\lambda x/n} \\ &= e^{\lambda x/n} \frac{n(n-1)\cdots(n-x+1)}{n \cdots n} e^{-\lambda} \frac{\lambda^x}{x!} \\ &\xrightarrow{n\to\infty} e^{-\lambda} \frac{\lambda^x}{x!}. \end{aligned}$$

Showing this statement more rigorously might make use of Stirling's approximation of $n!$. Since X_n is discrete for all $n \in \mathbb{N}$ we conclude that

$$X_n \xrightarrow[d]{n\to\infty} \text{Poi}(\lambda).$$

5) From Poisson to Exponential

When working with a Poisson process, a natural question is the distribution of time between the occurence of events. Let T denote the time at which the first event occurs, so $\Omega_T = [0, \infty)$. See that if $N(t) \sim \text{Poi}(\lambda t)$ is a Poisson process then $p(N(t) = x)$ is the probability of x events occuring in $[0, t]$. We

have

$$\begin{aligned} p(T < t) &= \sum_{x=1}^{\infty} p(N(t) = x) \\ &= 1 - p(N(t) = 0) \\ &= 1 - e^{-\lambda t} \end{aligned}$$

and so $p(t) = \lambda e^{-\lambda t}$.

A.3 The Law of Large Numbers

If $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are i.i.d. and $\mathbb{E}[\mathbf{X}] < \infty$ then

$$\frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i \xrightarrow{n \to \infty} \mathbb{E}[\mathbf{X}]$$

with respect to some type of convergence. The law comes in a weak flavour and a stronger one. The weak flavour pertains to convergence in probability and the latter to asmymptotic convergence.

Frankly, the details aren't important for my purposes, so I simply think of it as meaning "The sample mean of a buncha samples is the population mean.". It's useful for justifying certain approaches, e.g. Monte Carlo integration.

A.4 The (univariate) Central Limit Theorem (CLT)

If $X_1, \ldots, X_n$ are i.i.d. and $\mathrm{Var}(X) < \infty$ then

$$\sqrt{n} \left(\frac{1}{n} \sum_{i=1}^{n} X_i - \mathbb{E}[X] \right) \xrightarrow[d]{n \to \infty} \mathcal{N}(0, \mathrm{Var}(X)).$$

It's one of the most important results in probability theory and statistics: it helps explain why the normal distribution appears so often in real world data. I think of it as meaning "The scaled sum of i.i.d. samples will look normal upon repeatedly sampling.".

Its clearest use-case to me is in justifying the modelling of residuals in linear regression as normally distributed.

A.5 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 of it, so I won't include one.

A.6 Motivating Variance and the Bias of Sample Variance

TODO

A.7 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 less surprising, with some extra conditions. So if $p(\mathbf{x}) = 0.01$ then we 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 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 the 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 of two independent events occurring should be the sum of the surprises of the events individually. Thirdly, the surprise of a given event should be higher than the surprise of any less probable event. So, for all $\mathbf{x}_1, \mathbf{x}_2 \in \Omega_{\mathbf{X}}$, 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 being conveyed is clear without.

A.7.1 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

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

It is often used as a measure of similarity between two distributions: it follows from Gibbs' inequality that it is non-zero and it is often proposed as a loss function when assessing how well a model q encodes an underlying distribution p. In many cases, a loss function is chosen whose minimisation is equivalent to minimising the relevant KL-divergence.

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

Suppose we have the geometric distribution with $p = 1/2$ and would like to fit it using a Poisson distribution. That is, we would like to best approximate

$$\begin{aligned} p : \mathbb{N} &\to [0,1] \\ k &\mapsto 2^{-(k+1)} \end{aligned}$$

by finding a nice λ for

$$\begin{aligned} q : \mathbb{N} &\to [0,1] \\ k &\mapsto \frac{e^{-\lambda}\lambda^k}{k!}. \end{aligned}$$

To find a good λ, it makes sense to first come up with a metric for how good a given value 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 it's minimised by. 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 $C = \sum_{k=1}^{\infty}\frac{\log(k!)}{2^k}$ is independent of λ. In computing the minmum of $H(p,q)$ in λ we obtain

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

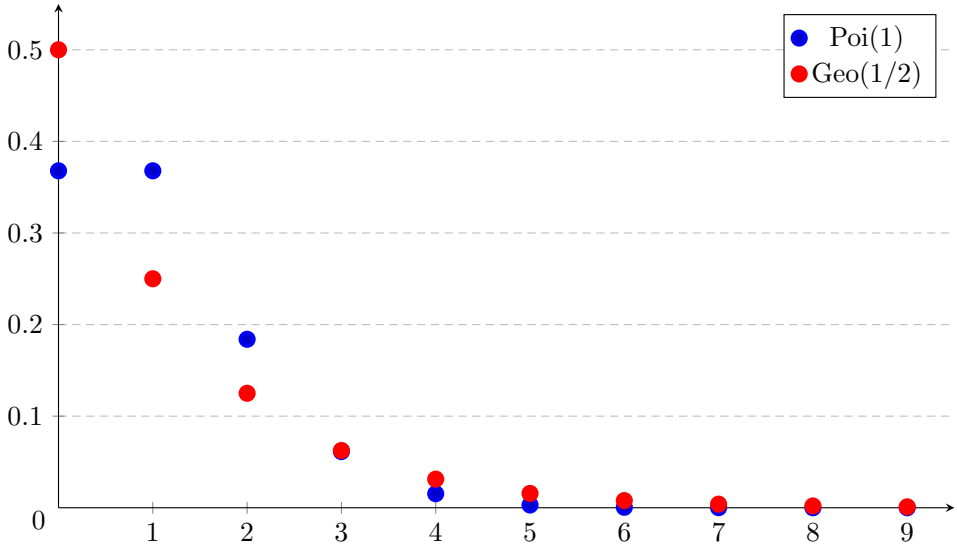


Figure 29: Our geometric distribution p and Poisson fit q.

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

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

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

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:

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

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

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

A.8 Pearson Correlation and Mutual Information

If X and Y are scalar random variables then their Pearson correlation of is given by

$$\rho_{X,Y} = \frac{\mathbb{E}\left[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])\right]}{\sqrt{\mathbb{E}\left[(X - \mathbb{E}[X])^2\right]\mathbb{E}\left[(Y - \mathbb{E}[Y])^2\right]}} = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y} \in [-1, 1]$$

where the expectation in the numerator is over their joint $p_{(X,Y)}$. The overall quantity yields the linear dependence between X and Y and the denominator is for normalisation.

What if we want an idea of the non-linear dependence between random variables? This desire motivates mutual information. The mutual information of the joint over $\mathbf{X}$ and $\mathbf{Y}$ is often spoken of as a property of two random variables $\mathbf{X}$ and $\mathbf{Y}$ and is denoted as such as in

$$I(\mathbf{X}; \mathbf{Y}) = D_{\mathrm{KL}}(p_{\mathbf{X},\mathbf{Y}} || p_{\mathbf{X}} \otimes p_{\mathbf{Y}}).$$

I'm massively in favour of writing it as a KL-divergence as its meaning is immediately clear: its a measure of divergence between the joint and the product of the marginals. Informally, I see it as asking "How well does the product of marginals model the joint?" which is precisely the question we seek to answer when utilising things like Pearson correlation and mutual information.

A.9 Quality of fit ≈ Encoding quality?

TODO: Perhaps worthy of its own section, unsure

A.10 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 $\mathbf{X} = (X_1, \ldots, X_q)$. That is, what if $p(\mathbf{x})$ is simply the marginal of the true joint $p(\mathbf{x}, \mathbf{z})$ where $\mathbf{Z} = (Z_1, \ldots, Z_{q'})$ 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. Variational 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.

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

B Philosophy Things

I don't think about the philosophic aspects of maching learning very often as I'm in the camp of people whose interests lie in solving present problems. I find some topics interesting though.

B.1 Thoughts on reasoning (Summer 2024)

At the time of writing, machine learning, as a term, is used almost interchangeably with artificial intelligence (AI). AI isn't particularly well-defined in itself, even in academic settings, so its use in day-to-day things, e.g. in the news, often causes confusion and debate. I'm not sure how I would define AI in a way that'd satisfy most. Maybe the discipline of studying things that can 'reason', as a human does, given some agreed upon notion of 'reasoning'. A common issue found in discussions surrounding this topic is that a definition of 'reasoning' is left out.

Right now, I'm unaware of any machine learning model that can 'reason' by any agreed upon definition of the term. I suspect that while current models can simulate 'reasoning' to the extent of being able to effectively solve tasks which require some sort of underlying understanding, they do not 'reason' in the human sense, which is grounded in embodied experience. Let's say that an LLM (e.g. GPT-4o) concludes that a continuous real-valued function $f : \mathbb{R} \to \mathbb{R}$ has a root in some interval (a, b) using the intermediate value theorem (IVT) by noting that $f(a) < 0$ and $f(b) > 0$. Did the LLM 'reason' in its application of the IVT or do we instead brush this off as the LLM having been trained on many examples of the application of the IVT? Does it 'understand' what it's concluding? As of right now I'd say that it doesn't. This of course isn't something unique to the IVT — we can construct many such examples. What I think makes the IVT interesting is that we can effectively express what it means to 'understand' its correctness as a human. I'd say that the distinction between the LLM's 'understanding' and a human's here is that a human knows that if they trace their finger on a piece of paper starting from the bottom half, crossing a horizontal line cutting through the middle of the paper, to the top half of the paper without removing their finger then their finger must have touched the horizontal line at some point (ignore the fact that such a traversal is not necessarily a function, you get the idea). You don't need any mathematical training to know and, more importantly, understand why your finger crosses through the horizontal line on the page. LLMs certainly do not 'understand' the correctness of the IVT in this way, or really any way that isn't purely

symbolic. Maybe multimodal LLMs will alleviate this.

A separate but related question is to what extent it matters that a model 'reasons' in order to arrive its conclusions. Somewhat in the same way that it doesn't matter that a calculator has no inherent idea of why it outputs 6 when you input $3 \cdot 2$. From a purely problem solving point of view, these philosophical debates regarding what constitutes reasoning, understanding and knowing aren't too important in the short term. It'd be funny if 50 years from now we're well-taken care of by a fleet of machine learning-powered robots and we still look down on them for not being capable of true reasoning.

B.2 Sufficiency of Internet-Scraped Data

TODO

B.3 The Principled Measure-Explainability Tradeoff

TODO