Machine Learning Explainers

Last updated: August 15, 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   Introduction

I think of machine learning as the discipline of studying methods of fitting parameterised models from data. Why I find machine learning so interesting interesting is due to its rapid development since the era of deep learning began in 2012. At their core, many methods in machine learning are statistically-principled in nature, corresponding to maximum likelihood estimation. The fact that something as simple as maximum likelihood estimation can be used to fit very complex distributions, in practice, to at least a small extent isn't too surprising. What amazes me is how well such complex distributions can be modelled using maximum likelihood estimation to fit deep architectures. So well that we are now able to produce models which take a natural language description as input and output a photo-realistic image matching said description. What right do such simple methods have to fit such complex distributions? What right do the architectures we use have to facilitate such effective learning? The more I've learned, the more I'm amazed that contemporary methods work as well as they do.

For a description of machine learning which clarifies why it is difficult to define as a term, consider the following description written by Herbert Jaeger: "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."

2   Supervised Learning

Supervised learning algorithms are a class of machine learning algorithms in which a model learns some relationship between model variables and

output variables by being shown concrete examples of what a given input (pertaining to the model variables) should yield as output. These model and output variables can be continuous or discrete in nature but usually it's a mix.

An example of a task which supervised learning is appropriate for includes regression predicting life expectancies of a poulation given features such as the population's height, age, BMI etc. You can think of regression tasks as those where the output of the model should belong to some continuously distributed space. Other than regression, the other vanilla supervised learning task is classification. Well known examples of binary classification include 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. It's amazing that something difficult as recently as 2005 is trivial now for various levels of machine learing understanding.

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 $\mathcal{D}$ into a training set $\mathcal{D}_{\text{train}}$ and a testing set $\mathcal{D}_{\text{test}}$. How exactly this split is done depends on the task at hand but an example is 50/50 uniformly randomly splitting the original dataset $\mathcal{D}$ — chronologically or not. It can be important that this is done uniformly randomly. For example, if one is tackling a binary classification task and $\mathcal{D}$ consists of 1000 samples with 500 belonging to class 1 and 500 belonging to class 2 then it's important that one does not split in a way that causes egregiously imbalanced representations of these classes. Training a model on 500 samples of class 1 then testing it on 500 samples of class 2 wouldn't make much sense. Similar idea for regression tasks but with ensuring a reasonable balance in representation of values belonging to given intervals corresponding to the output space. With a reasonable split we train the model on $\mathcal{D}_{\text{train}}$. It is all the model knows — $\mathcal{D}_{\text{train}}$ is the model's universe. After training, we measure the model's ability to generalise to new samples belonging to the same underlying distribution by seeing how well it performs with respect to the samples in $\mathcal{D}_{\text{test}}$. Essentially comparing its output given some sample to the ideal/true output value of said sample. Things get a bit more fancy when we consider cross-validation but that'll come later.

2.1 Linear Regression

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

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

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

$$f_\theta : \Omega_{\mathbf{X}} \to \Omega_Y$$

$$\mathbf{x} \mapsto \theta^{\text{T}} \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} = \theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p$$

where $\theta = (\theta_0, \theta_1 \ldots, \theta_p) \in \mathbb{R}^{p+1}$ 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 $\tilde{\mathbf{x}} \in \Omega_{\mathbf{X}}$ we can predict the corresponding output as $\tilde{y} = f_\theta(\tilde{\mathbf{x}})$.

> **Assumption of normally distributed residuals**
>
> [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."
>
> Another good link: `https://stats.stackexchange.com/a/12266`.
>
> ---
> [a]`https://en.wikipedia.org/wiki/Central_limit_theorem#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 $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ (each consisting of n sample in our case) and minimising some pre-determined loss function of said parameters over $\mathcal{D}_{\text{train}}$. 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 $d : Y \times Y \to \mathbb{R}_{\geq 0}$ is some metric on the set of possible output values and itself. 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 — it's mostly a notational convenience thing. So in our case, we seek to compute

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

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 $\mathcal{D}_{\text{train}}$ 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

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

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

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

for some small $\delta \in \mathbb{R}$. That said, what's given above is not rigorous, so let's derive it. Note that

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

where

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

and $x_{i,j}$ denotes the jth element of the ith sample. To find the minimiser(s) of $\text{MSE}(\theta)$, 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 $\text{MSE}(\theta)$ and see that it is semi-positive definite. On that note, we have

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

and so the optimiser θ^* is given by

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

> **Note:** There are some pieces of literature that define linear regression models as regression models that are linear in their parameters. By this definition, linear regression, as a term, would encompass polynomial regression models and other regression models with non-linear basis functions so long as the model is linear in its parameters. This confuses me as 'linear regression' most often refers to models that fit a hyperplane to data. In fairness, from what I understand, using the least squares method derived below for all regression models linear in their parameters is viable.

2.1.1 Statistical Motivation

This idea of minimising the mean square error of the model over $\mathcal{D}_{\text{train}}$ even has some rigorous statistical motivation. Assume that the residuals corresponding of the model's output over $\mathcal{D}_{\text{train}}$ 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

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

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

$$
\begin{aligned}
\theta_{\text{MLE}} &= \underset{\theta \in \mathbb{R}^{p+1}}{\arg\max} \left[\log\left(\prod_{i=1}^{n} f_{Y|\mathbf{X}}(y_i | \mathbf{x}_i; \theta) \right) \right] \\
&= \underset{\theta \in \mathbb{R}^{p+1}}{\arg\max} \left[\sum_{i=1}^{n} \log(f_{Y|\mathbf{X}}(y_i | \mathbf{x}_i; \theta)) \right] \\
&= \underset{\theta \in \mathbb{R}^{p+1}}{\arg\max} \left[n \log\left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - f_{\mathbf{w},b}(\mathbf{x}_i))^2 \right] \\
&= \underset{\theta \in \mathbb{R}^{p+1}}{\arg\min} \left[\sum_{i=1}^{n} (y_i - f_{\mathbf{w},b}(\mathbf{x}_i))^2 \right].
\end{aligned}
$$

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 $\mathcal{D}_{\text{train}}$.

2.1.2 Example

Age (years)	BMI (kg/m^2)	Cholesterol (mg/dL)
25	22.5	180
30	24.0	190
35	26.5	210
40	28.0	220
45	29.5	240
50	30.0	250
55	31.5	260
60	32.0	270
65	33.5	280
70	34.0	290

Table 1: Patient data pertaining to their age, BMI and cholesterol levels.

Suppose we are given the dataset illustrated in Table 1 pertaining to a set of patients' ages, BMIs and cholesterol levels and that we would like to predict the cholesterol levels of some other set of patients based on their ages and heights. In this context we have two features, age and BMI, which will correspond to the parameters θ_1 and θ_2 respectively. As such, our model's output function will be of the form

$$f_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

where x_1 and x_2 correspond to the input sample patient's age and BMI level respectively. Before fitting a linear regression model to our dataset, it's a good idea to see if a linear underlying relationship exists in the first place. To do this, we can plot the given dataset to obtain Figure 1.

Looks sufficiently linear to me. We'll split the dataset into training and testing datasets via Table 2 and Table 3. Since $\mathcal{D}_{\text{train}}$ consists of only five samples, it's feasible to compute the optimal parameters θ^* by hand. We have

$$y = \begin{bmatrix} 180 \\ 210 \\ 240 \\ 260 \\ 280 \end{bmatrix}, \quad X = \begin{bmatrix} 1 & 25 & 22.5 \\ 1 & 35 & 26.5 \\ 1 & 45 & 29.5 \\ 1 & 55 & 31.5 \\ 1 & 65 & 33.5 \end{bmatrix}$$
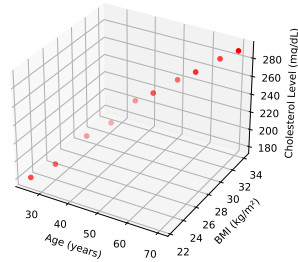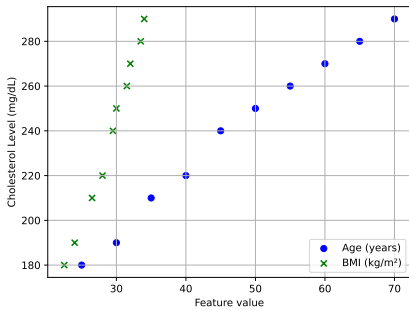
Figure 1: Each feature against the output (left) and the dataset in the feature space (right).

Age	BMI	Cholesterol
25	22.5	180
35	26.5	210
45	29.5	240
55	31.5	260
65	33.5	280

Table 2: Training dataset.

Age	BMI	Cholesterol
30	24.0	190
40	28.0	220
50	30.0	250
60	32.0	270
70	34.0	290

Table 3: Testing dataset.

from which we obtain

$$\theta^* = \left(X^{\mathrm{T}}X\right)^{-1} X^{\mathrm{T}} y = \begin{bmatrix} 25.68 \\ 0.94 \\ 5.79 \end{bmatrix}$$

which corresponds to

$$f(x_1, x_2) = 25.68 + 0.94x_1 + 5.79x_2.$$

What's nice about such a low-dimensional problem like this is that visualising the loss landscape is relatively doable. We have three parameters so I'll fix $\theta_0 = 25.68$ and plot θ_1 and θ against

$$\mathrm{MSE}(25.68, \theta_1, \theta_2) = \sum_{i=1}^{5}(y_i - 25.68 - \theta_1 \mathbf{x}_{i,1} - \theta_2 \mathbf{x}_{i,2})^2$$

where the sum is taken over the training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{5}$.

As for how well the model performs, its MSE on $\mathcal{D}_{\text{train}}$ is 1.26 and its MSE on $\mathcal{D}_{\text{test}}$ is 12.24. In this case, such a difference is fine. To get a visual on how well the model does on the training and testing sets, consider Figure 2. I should improve these at some point.
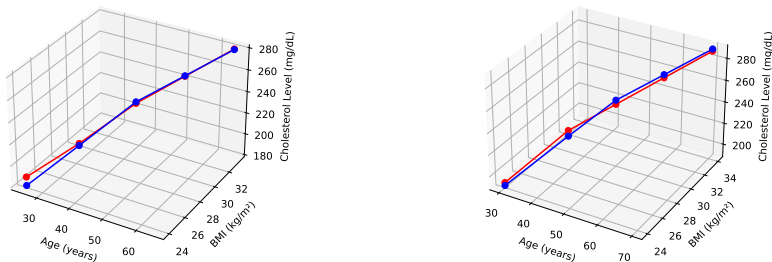
Figure 2: Model output for the training set (left) and testing set (right). Hard to appreciate these 3D figures.

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

The following is taken from an answer on Quora[a] (the top comment is worth reading) based on a paper[b] published in Nature. Related[c].

___

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

2.1.3 Regularisation

The most well known methods of regularisation in the context of regression are Lasso and Ridge regression. These are also known as L1 and L2 regularisation. They are motivated very nicely using Bayesian arguments. L1 regularisation corresponds to a Laplace prior and L2 to a Gaussian prior.

2.1.4 R^2 value

. . .

2.1.5 Terminology

Nowadays, regression models are those which predict a value belonging to some continuous space but where did it come from? In 1886, Francis Galton authored "Regression Towards Mediocrity in Hereditary Stature" which is where the 'regression towards the mean' term comes from. Galton noticed that tall father's have short sons, short fathers have tall sons and average height fathers 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 we first normalise to have RVs of mean 0 and variance 1 then we would have $E[Y|X] = \rho X$ where ρ is the correlation coefficient between X and Y. Since $|\rho| \leq 1$ we know that the expected value of Y is close 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.

2.1.6 Opportunity to Demonstrate High Quality Data

Let's say we had perfect data. That is, no noise and samples truly lie on the underlying hyperplane fit. In such a case, if our samples are of the form $(x_1, \ldots, x_n, y) \in \mathbb{R}^{n+1}$ then the hyperplanes of interest are n-dimensional (assuming independent features?) and each is uniuely identified by $n + 1$

distinct points on the plane. So if our data is perfect then bam, all we need is $n + 1$ linearly independent samples to perfectly fit the data in the linear regression case.

The idea of course extends to things like polynomial regression, just more points are needed depending on the degree.

2.2 Logistic Regression

While linear regression is the staple example of regression models, its equivalent for binary classification is logistic regression. The name is a bit confusing at first since it's fair to expect that something with the name regression would be used for regression tasks, i.e. predicting something continuously distributed, as opposed to something discrete like binary classification. Logistic Regression essentially applies a logistic (or sigmoid) transformation to the output of our model to introduce a notion of confidence of the classification. Understandably, this transformation maps all outputs to $(0, 1)$. This is what makes it regression-like.

Binary classification is the task of assigning one of two classes to some input sample. For example, given some data pertaining to a patient's health, it might be nice to be able to predict whether they are at risk of suffering from a heart attack or not. With this in mind, how does one classify a sample at all? The idea is to fit a hyperplane to the feature space of the distribution which we wish to model. A hyperplane is an $(n - 1)$-dimensional object embedded in n-dimensional space. So in 2D space, a hyperplane is just a line and in 3D space it's a plane etc. A hyperplane in $(p + 1)$−dimensional space consists of the points $(x_1, \ldots, x_p) \in \mathbb{R}^p$ which satisfy

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p = 0$$

where $\theta = (\theta_0, \ldots, \theta_p)$ are the hyperplane's coefficients and $\mathbf{x} = (1, x_1, \ldots, x_p)$ and we can denote the function corresponding to the hyperplane by

$$f_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p = \theta^{\mathrm{T}} \mathbf{x}.$$

The reason for the additional 1 appearing at the beginning of $\mathbf{x}$ is similar to as in linear regression: it accounts for the bias term θ_0 and makes notation a lot cleaner. After fitting the parameters of such a hyperplane, we can 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. $f_\theta(\mathbf{x}) \leq 0$, are classified as 0 and samples 'above', i.e. $f_\theta(\mathbf{x}) > 0$, are classified as 1. With this idea in mind, the hard part of this approach is finding an appropriate hyperplane.

That is, we want $f_\theta(\mathbf{x})$ to be such that sufficiently many samples of each class lie on the correct side of the hyperplane. To regressionify this approach, we apply a logistic transformation to the output $f_\theta(\mathbf{x})$ for a given sample $\mathbf{x}$ yielding

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

and say that sample $\mathbf{x}$ is of class 1 if $h_\theta(\mathbf{x}) > p$, for some pre-determined threshold $p \in (0,1)$, and of class 0 if $h_\theta(\mathbf{x}) \leq p$. So ultimately, $C_\theta(\mathbf{x}) = \mathbb{1}(h_\theta(\mathbf{x}) > p)$. Typically, people take $p = 0.5$ which yields precisely the same classifications as $f_\theta(\mathbf{x})$. To see this, note that if $p = 0.5$ then we have

$$
\begin{aligned}
C_\theta(\mathbf{x}) = 1 &\iff h_\theta(\mathbf{x}) > 0.5 \\
&\iff \frac{1}{1 + \exp(-f_\theta(\mathbf{x}))} > 0.5 \\
&\iff 1 > \exp(-f_\theta(\mathbf{x})) \\
&\iff f_\theta(\mathbf{x}) > 0 \\
&\iff \theta^{\mathrm{T}}\mathbf{x} > 0.
\end{aligned}
$$

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^*.

2.2.1 Statistical Motivation

Suppose that the outputs of the logistic function h_θ are Bernoulli distributed with parameter p. That is, suppose that $h_\theta(\mathbf{x}) \sim \text{Bernoulli}(p)$. In this case, the probability that $\mathbf{x}$ belongs to class $y \in \{0,1\}$ is given by

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

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

$$
\begin{aligned}
l(\theta) &= \sum_{i=1}^n \log(\mathbb{P}(y_i|\mathbf{x}_i; \theta)) \\
&= \sum_{i=1}^n \log\left((h_\theta(\mathbf{x}_i))^{y_i}(1 - h_\theta(\mathbf{x}_i))^{1-y_i}\right) \\
&= \sum_{i=1}^n y \log(h_\theta(\mathbf{x}_i)) + (1 - y_i)\log(1 - h_\theta(\mathbf{x}_i)) \\
&= \sum_{i=1}^n \left[y_i \log\left(\frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right) + (1 - y_i)\log\left(1 - \frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right)\right] \\
&= \sum_{i=1}^n \left[y_i \log\left(\frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right) + (1 - y_i)\log\left(\frac{\exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right)\right] \\
&= \sum_{i=1}^n \left[y_i \log\left(\frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right) + (1 - y_i)\left(-\theta^{\mathrm{T}}\mathbf{x}_i + \log\left(\frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)}\right)\right)\right] \\
&= \sum_{i=1}^n \left[-\log(1 + \exp(-\theta^{\mathrm{T}}\mathbf{x}_i)) - (1 - y_i)\theta^{\mathrm{T}}\mathbf{x}_i\right]
\end{aligned}
$$

which we maximise using something like gradient descent to obtain the optimal paramateres

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

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 samples. So in some sense, logistic regression is a probabilistic approach while SVMs take a 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' is 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.

Note: Terminology surrounding SVMs is inconsistent. 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. Here, 'an SVM' refers to the former and I'll state explicitly when considering the latter.

2.3.1 Hard-margin SVMs

Since we aim to fit a hyperplane in our feature space $\mathbb{R}^{p+1}$, we use the same notation for the function $f_\theta(\mathbf{x})$ corresponding to the linear decision boundary (i.e. hyperplane). To make notation a bit easier, let $\theta_+ = (\theta_1, \ldots, \theta_p)$ so that θ is the ordered concatentation of θ_0 and θ_+. Further, let $\mathbf{x}_i$ denote the ith sample's features and $z_i \in \{-1, 1\}$ its class such that z_i is 1 if $\theta_+^{\text{T}}\mathbf{x}_i + \theta_0 > 0$ and 0 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

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

which, after some rearranging, yields

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

Taking a point $\mathbf{x}_i$ whose class is -1 yields $d_i = -\frac{\theta_+^{\text{T}}\mathbf{x}_i + \theta_0}{||\theta_+||}$ and so a neater way of writing this distance for an arbitrary sample $\mathbf{x}_i$ is $d_i = \frac{z_i(\theta_+^{\text{T}}\mathbf{x}_i + \theta_0)}{||\theta_+||}$. Let $d_{\min} = \min_{i=1,\ldots,n} d_i$. The idea behind hard-margin SVMs is only applicable
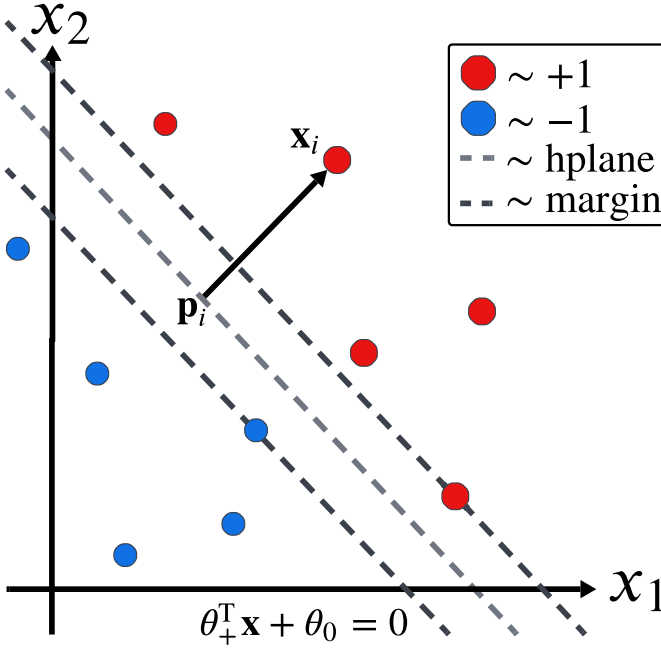
Figure 3: A hard-margin SVM in two dimensions.

to entirely linearly sparable data and effectively boilds down to finding which parameters maximise $d_{\min}$. That is, we look to compute

$$\theta^* = \underset{(\theta_0, \theta_+)}{\arg\max} \left[\min_{i=1,\ldots,n} d_i \right]$$

$$= \underset{(\theta_0, \theta_+)}{\arg\max} \left[\min_{i=1,\ldots,n} \frac{z_i(\theta_+^{\mathrm{T}} \mathbf{x}_i + \theta_0)}{||\theta_+||} \right]$$

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

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

in which r is arbtirarily scalable, so it is equivalent to

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

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

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

which is solved in the usual convex problem solving ways.

2.3.2 Soft-margin SVMs

Hard-margin SVMs are rarely applicable. In practice, classes are not entirely linearly separable, due to noise and outliers, 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

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

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 4, 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 simplified to computing
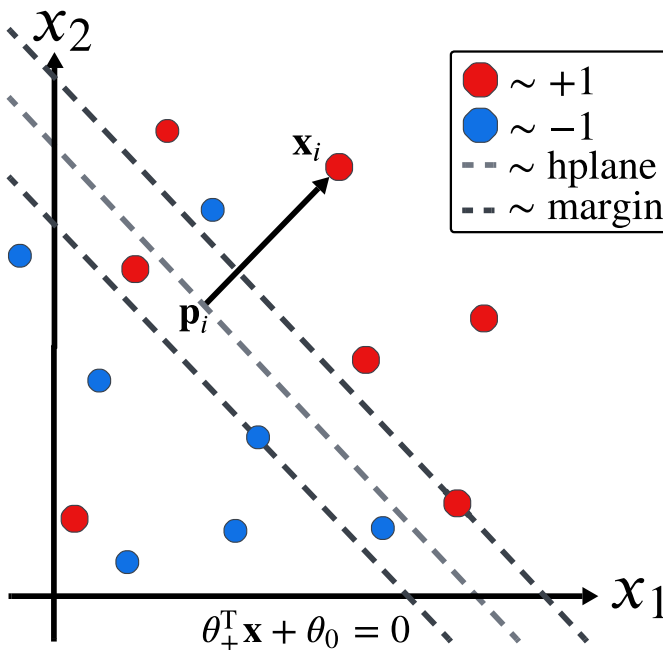
$$\arg\min_{(\theta_0,\theta_+)} \left[||\theta_+||^2 + \lambda \sum_{i=1}^{n} \max\left(0, 1 - z_i\left(\theta_+^{\mathrm{T}}\mathbf{x}_i + \theta_0\right)\right)\right]$$

which can be done using gradient descent.

2.3.3 Non-linear SVMs

- the kernel trick

- standard kernels

- derivation of standard identity

...

Figure 4: A soft-margin SVM in two dimensions. Samples are labelled according to their true class.

2.4   Decision Trees and Random Forests

- split according to some feature

- prune prune prune

2.5   Handcrafted Features

A lost art in the current era :(

2.6   The Bias-Variance Tradeoff

The bias-variance tradeoff is a statement pertaining to how well one can perform supervised learning from datasets. Crudely put, if the model family used to fit is too simple to represent the underlying distributions from which the samples were drawn then our learning method is highly-biased: our method is almost dataset-invariant. Think of fitting a line to a high-degree polynomial. On the other end, if the model family is too complex

(e.g. overparameterised) then one might expect to more-easily overfit the underlying distribution. There are too many parameters to twink and tweam so fitting the individal samples is too easy.

2.6.1 Derivation with MSE loss

Recall that sample targets pertaining to datasets from which supervised models are learned are effected by noise, i.e. $Y|(X = x) = f(x) + \epsilon(x)$ where $\epsilon(x) \sim \mathcal{N}(0, \sigma^2(x))$ and so

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

We seek to learn an estimate $\hat{f}_D$ of $f(x) = \mathbb{E}_Y[Y|X = x]$ from the dataset $D \subset (\Omega_X \times \Omega_Y)^n$. In other words, the model/estimate $\hat{f}_D$ is determined by the realisation D of $\mathcal{D}$ where $\mathcal{D}$ is distributed according to $p(x, y)^{\otimes n}$ 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

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

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

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

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

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

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

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

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

$$= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_X\left[\mathbb{E}_Y\left[(Y - f(X))\left(f(X) - \hat{f}_{\mathcal{D}}(X)\right)|X\right]\right]\right]$$

$$= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_X\left[\left(f(X) - \hat{f}_{\mathcal{D}}(X)\right)\mathbb{E}_Y[Y - f(X)|X]\right]\right]$$

$$= 2\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_X\left[\left(f(X) - \hat{f}_{\mathcal{D}}(X)\right) \cdot 0\right]\right]$$

$$= 0$$

in which the tower property of conditional expectations

$$\mathbb{E}_{(X,Y)}[g(X,Y)] = \iint g(x,y) f_{(X,Y)}(x,y) dx dy$$
$$= \int \left(\int g(x,y) f_{Y|X}(y|x) dy \right) f_X(x) dx$$
$$= \mathbb{E}_X \left[\mathbb{E}_Y \left[g(X,Y)|X \right] \right]$$

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

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

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

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

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

$$\mathbb{E}_{\mathcal{D}} \left[\mathbb{E}_{(X,Y)} \left[(Y - f(X))^2 \right] \right] = \mathbb{E}_{(X,Y)} \left[(Y - f(X))^2 \right]$$
$$= \mathbb{E}_X \left[\mathbb{E}_Y \left[(Y - f(X))^2|X \right] \right]$$
$$= \mathbb{E}_X \left[\mathbb{E}_Y \left[(Y - \mathbb{E}_Y[Y|X])^2 |X \right] \right]$$
$$= \mathbb{E}_X \left[\text{Var}_Y \left(Y|X \right) \right]$$
$$= \mathbb{E}_X [\sigma^2(X)]$$

This is simply the expected noise, e.g. due to imperfect calibration in the instruments used to obtain samples, and is typically taken to be constant. Usually, as a constant, it is simply denoted by σ^2.

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

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

See that the final term vanishes as $\mathbb{E}_{\mathcal{D}}\left[f(\bar{X}) - \hat{f}_{\mathcal{D}}(X)\right] = 0$ and so

$$\mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_X\left[\left(f(X) - \hat{f}_{\mathcal{D}}(X)\right)^2\right]\right]$$

$$= \mathbb{E}_X\left[(f(X) - \bar{f}(X))^2\right] + \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_X\left[\left(\bar{f}(X) - \hat{f}_{\mathcal{D}}(X)\right)^2\right]\right]$$

$$= \mathbb{E}_X\left[(f(X) - \bar{f}(X))^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\bar{f}(X) - \hat{f}_{\mathcal{D}}(X)\right)^2\right]\right]$$

$$= \mathbb{E}_X\left[\left(f(X) - \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(X)\right]\right)^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathcal{D}}(X) - \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(X)\right]\right)^2\right]\right].$$

Put together, we obtain

$$\mathbb{E}_{\mathcal{D}}\left[R\left(\hat{f}_{\mathcal{D}}\right)\right]$$

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

where $\sigma^2 = \mathbb{E}_X\left[\sigma^2(X)\right]$ (ignore the poor choice of notation). We can crudely summarise it as the sum of the weighted average, according to $p(x)$, of the square of the bias of the learning method, the variance of the estimate according to the dataset from which it is determined and irreducible noise. The irreducible noise is invariant to the learning method and so it's usually ignored in discussions surrounding the bias-variance tradeoff.

In brief, if the learned estimate varies a lot as we vary the dataset from which it is determined then the learning method has high variance. This is often due to the estimates in question overfitting the dataset on which they were learned which itself is often due to overparameterisation. If the number of parameters of a model is far higher than is needed then one can easily fit the individal samples from the dataset used during learning. Think of fitting a high degree polynomial to a line. The dataset will consist of points which roughly form a line but the high degree polynomial, if not regularised, can easily fit the samples present in the dataset used to learn (and in doing so fits the noise) resulting in an overfit model.

The flip side of this is an underparameterised model. Since such a model has little hope of fitting the dataset from which it is learned, you'd expect it to rarely overfit (it can't) and to instead be almost invariant to the dataset. That is, no matter which dataset you gave it to learn from, you get a similar
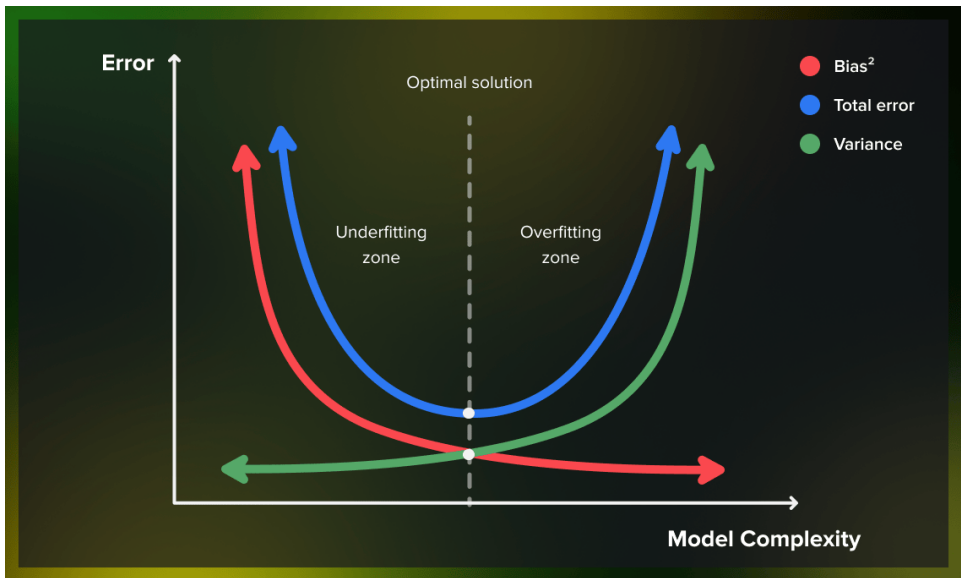
Figure 5: The graph of learning method error against model complexity in terms of bias and variance.

model each time. In other words, the learning method (the method of going from dataset to model) is biased. Think of fitting a line to high degree polynomial — no matter what dataset you use, the best fit line will likely not change too much (which is usually a good thing) but the fit itself will never be anything but linear, i.e. always a bad fit.

With this in mind, when considering which learning method (i.e. model family) to consider when tackling a problem, thought should be given to how complex (i.e. parameter number) the model family should be. Too simple (i.e. too few parameters) and you underfit (high bias, low variance). Too complex (i.e. too many parameters) and you overfit (low bias, high variance). This is nicely illustrated in Figure 5 in which we see that there exists a sweet spot in model complexity. A spot in which said model complexity yields some square of bias and some variance but the minimum some of said quantities, i.e. the lowest learning method error. You can think of this plot as the graph of

$$\left(N\left(\mathcal{A}\right), \mathbb{E}_X\left[R\left(\hat{f}_{\mathcal{D}}(X)\right)\right]\right)$$

where $N(\mathcal{A})$ denotes the number of parameters of models given by $\mathcal{A}$.

2.6.2 TODO: How it motivates regularisation

For ordinary least squares (OLS), methods of regularisation, such as L1 and L2, introduce some bias but reduce variance. The result is better performance according to MSE.

For classification, as the quantity of data increases, the variance of learned models decreases. As such, if we get more and more data, it might be more tempting to use methods which knowingly have minimal bias as the variance will be taken care of by increased amounts of data. On the other hand, if you have few samples then perhaps consider methods with knowingly little variance, e.g. Bayesian methods.

2.6.3 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.
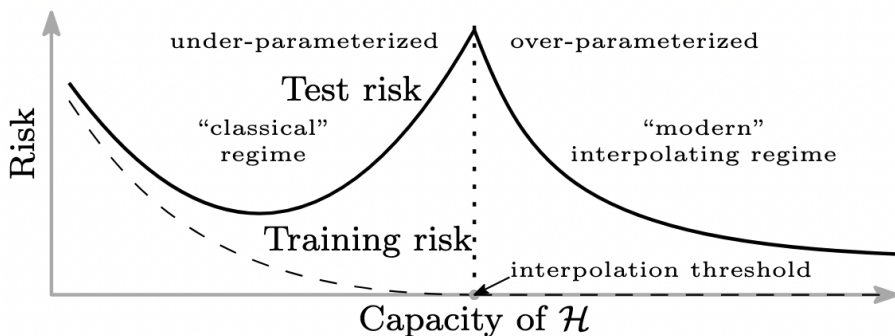
2.6.4 Double descent

Double descent refers to a behaviour in which overparameterised models ignore 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 this threshold is the number of samples used to determine the model. This is illustrated nicely in Figure 6.

A consequence of this is that very deep neural networks learned using gradient descent do exceptionally well. It flies in the face of conventional wisdom. I'm unaware of anyone that this doesn't surprise. It isn't well-understood. It's related to grokking.

2.7 Misc. questions

Some questions to challenge one's understanding.

Figure 6: The double descent phenomenon.

What are the benefits of using logistic regression over more sophisticated methods?

Mostly because statistical tests can be conducted on parameters, e.g. statistical significance tests. It also helps that it offers a notion of classification confidence and that it simplementation and interpretation are straightforward.

Q2: Why not use mean absolute error instead of MSE?

There are many reasons for this, one of which is that often in machine learning and statistics-related contexts one would like to be able to compute the derivative(s) of such a loss function, e.g. in gradient descent, and the presence of an absolute value makes this annoying.

As to why it's useful that the MSE involves the squares of the errors, instead of exponents like $2.05, 3$ or π, it was best put by this answer to a Stack Exchange answer about the same question[1]:

"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 numbner 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

---

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

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)."

Q3: **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[2], 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.

3 Gradient Descent and its Optimisation

. . .

3.1 Batch learning

Loss landscape gets smoothed out. Make sure to make a note of what Welch Labs states about projecting loss landscapes: every step you make in the actual high dimensional loss landscape will change its 2D projection. His example of the actual loss landscape being 2D and illustrating its 1D projection makes for a great example (he mentions that this only applies to the case of training on a single sample? I don't understand entirely as of now). He also makes an interesting point of "if we get stuck in a global minimum then it must be a minima wrt all dimensions, which is less likely to happen the higher the dimension" whatever that might mean precisely.

3.2 Gradient Descent

. . .

3.3 Momentum

. . .

3.4 Hyperparameter Tuning

. . .

---

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

3.5   Misc. questions

Some questions to challenge one's understanding.

Q1: Why does $f(\mathbf{x})$ ascend most in the direction of $\nabla f(\mathbf{x})$?

The following is more of a proof *that* $\nabla f(x)$ yields the direction of steepest ascent not an intuition of *why*. For some reason, it is typically taken for granted that $f(\mathbf{x})$ would both ascend and ascend *most* in the direction of $\nabla f(\mathbf{x})$. I do not consider this obvious at all, so here is an informal physicist-like argument. Note that in this argument, we consider only unit vectors $\mathbf{v}$. This is because we care only about the direction of such vectors, so size does not matter, and an equation later on simplifies quite nicely due to its unit length.

In gradient descent, we are looking for the unit vector $\mathbf{v}$ such that $f(\mathbf{x})$ increases most in the direction of $\mathbf{v}$, i.e. we seek to compute

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

where $\eta > 0$. If η is small then approximating $f(\mathbf{x} + \eta\mathbf{v})$ using its Taylor expansion yields

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

from which we obtain

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

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 $\mathbf{x}$ by a distance $\eta > 0$, 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 where $\nabla f(\mathbf{x}) = 0$ corresponds to $f(\mathbf{x})$ being a local maximum.

Note: This argument, which utilises a Taylor approximation, relies on a sufficiently small learning rate $\eta > 0$. Otherwise, the terms in the Taylor approximation of $f(\mathbf{x} + \eta\mathbf{v})$ beyond $\nabla f(\mathbf{x}) \cdot \eta\mathbf{v}$ are not necessarily negligible.

Q2: Which loss function is appropriate for gradient descent?

There are a ton of machine learning-related scenarios that would need a specially-chosen loss function but the list below is a fine overview. Most are principled in their motivation due to their minimisation corresponding to maximum likelihood estimation in some way.

Task	Loss function
Binary classification	Binary cross-entropy
Multi-class classification	Categorical cross-entropy
Regression	Mean square error
Multi-label classification	Binary cross-entropy over each label

Table 4: Loss function overview.

Q3: How should we choose the learning rate η?

This is entirely problem-dependent but $\eta = 0.1$ is a great staple choice. If convergence is unstable then try $\eta = 0.01$. Other than this line of thinking, there are some 'rules of thumb' relating batch number/size to learning rate, otherwise it's a 'whatever seems to work best' kind of thing.

It's perhaps also worth mentioning the idea of learning over some number of epochs with $\eta = 0.1$ and then for a few more epochs with $\eta = 0.01$. There's some name for this idea. It's a pretty intuitive one and I first saw it used in practice by Karpathy.

Q4: How should we initialise the parameters? TODO

Xavier initialisation is often-used for neural networks. Maybe things have changed in this regard though; things move fast. Unsure about older architectures.

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 kind of neural network. Rigorously formulating MLPs is one of those things that's useful to do at least once. Formulating more sophisticated architectures, like CNNs, RNNs, etc., can be done in a much less rigorous manner if your use-case is solving tasks in an engineery way.

Note: Throughout, the terms 'neuron' and 'node' are synonymous. I use both without realising it at the time of writing.

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 7. Each layer is made up of a number of neurons, each of which has a real-valued activation value. For any node in a non-input layer, its activation value is the output of a non-linear activation function given, as input, the node'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[3].

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 nodes of each layer of the MLP (after the heavy simplifcation of assuming a roughly equal number of nodes in each layer).

For brevity, given $m, n \in \mathbb{N}$ with $m \leq n$ let $[n]_m = \{m, \dots, n\}$ 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 jth 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 $j - 1$.

---

[3]https://stats.stackexchange.com/a/159172
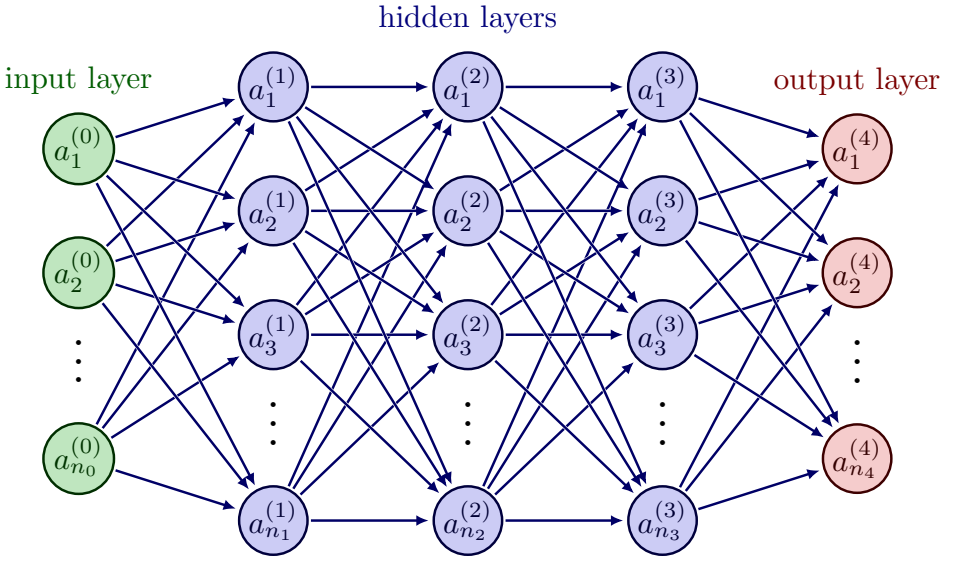
Figure 7: A multi-layer perceptron (MLP) with $k = 3$ hidden layers.

- Let $b_i^{(j)} \in \mathbb{R}$ denote the bias of neuron i in layer $j \in [k+1]$.

- 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

$$a_i^{(j+1)} = \sigma_{j+1}\left(\sum_{l=1}^{n_j} w_{i,l}^{(j+1)} a_l^{(j)} + b_i^{(j+1)}\right)$$

$$= \sigma_{j+1}\left(\left[w_{i,1}^{(j+1)} \ldots w_{i,n_j}^{(j+1)}\right] \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{n_j}^{(j)} \end{bmatrix} + b_i^{(j+1)}\right)$$

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 nodes in a non-input layer in terms of the activation

values of the neurons belonging to its preceding layer as

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

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.

4.1.1 Constructive example: The algebra
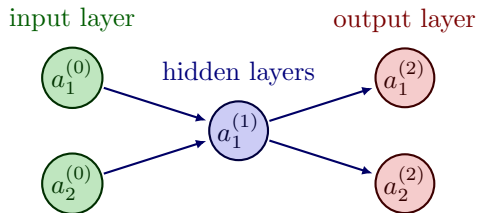
Figure 8: A neural network with one hidden layer ($k = 1$).

Consider the neural network in Figure 8 whose input, hidden and output layers consist of two, one and two neurons respectively. With such a simple neural network, we may explicitly express the output neurons $a_1^{(2)}$ and $a_2^{(2)}$ in terms of the input neurons $a_1^{(0)}$ and $a_2^{(0)}$. We see that $n_0 = 2$, $n_1 = 1$ and $n_2 = 2$ and so $\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \end{bmatrix}$ and $\mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} \\ w_{2,1}^{(2)} \end{bmatrix}$. Noting additionally that $\mathbf{b}^{(1)} = b_1^{(1)}$ and $\mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$ we can explicitly express the activation of

the single neuron in the hidden layer as

$$\mathbf{a}^{(1)} = \sigma_1 \left(\mathbf{W}^{(1)} \mathbf{a}^{(0)} + \mathbf{b}^{(1)} \right)$$

$$= \sigma_1 \left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)} \right)$$

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

$$\mathbf{a}^{(2)} = \sigma_2 \left(\mathbf{W}^{(2)} \mathbf{a}^{(1)} + \mathbf{b}^{(2)} \right)$$

$$= \sigma_2 \left(\begin{bmatrix} w_{1,1}^{(2)} a_1^{(1)} + b_1^{(2)} \\ w_{2,1}^{(2)} a_1^{(1)} + b_2^{(2)} \end{bmatrix} \right)$$

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

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

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

$$a_1^{(2)} = \sigma_2 \left(w_{1,1}^{(2)} \sigma_1 \left(w_{1,1}^{(1)} a_1^{(0)} + w_{1,2}^{(1)} a_2^{(0)} + b_1^{(1)} \right) + b_1^{(2)} \right)$$

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

4.1.2 Constructive example: Subbing values in

Note: The weights and biases of a neural network are typically learned during training, i.e. they are parameters to be tweaked and tuned until the corresponding model performs sufficiently well. They are not taken randomly as is done here for the sake of example.

Consider a neural network as in Figure 8. For the weights and biases, take

$$w_{1,1}^{(1)} = 0.5 \qquad\qquad w_{1,1}^{(2)} = 0.7$$

$$w_{1,2}^{(1)} = -0.5 \qquad\qquad w_{1,2}^{(2)} = 0.3$$

$$b_1^{(1)} = 0.1 \qquad\qquad b_1^{(2)} = 0.2$$

$$b_2^{(2)} = -0.2$$

and for the activation functions σ_1 and σ_2 take ReLU, i.e. $\sigma_i(x) = \max(0, x)$. Finally, take the input neuron values $a_1^0 = 5$ and $a_2^0 = 2$. Explicitly computing the activation values of the output neurons yields

$$
\begin{aligned}
a_1^{(2)} &= \max(0, 0.7 \cdot \max(0, 0.5 \cdot 5 - 0.5 \cdot 2 + 0.1) + 0.2) \\
&= \max(0, 0.7 \cdot 1.6 + 0.2) \\
&= 1.32
\end{aligned}
$$

and

$$
\begin{aligned}
a_2^{(2)} &= \max(0, 0.3 \cdot \max(0, 0.5 \cdot 5 - 0.5 \cdot 2 + 0.1) - 0.2) \\
&= \max(0, 0.3 \cdot 1.6 - 0.2) \\
&= 0.28.
\end{aligned}
$$

To confirm these values against the same neural network set up using TensorFlow, consider `nn_constructive_example.py` found here.

4.1.3 Universal Function Approximation for MLPs

A class of functions, or function class, $\mathcal{F}_S$ is capable of universal function approximation on $S \subset \mathbb{R}^n$ 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 $S \subset \mathbb{R}^n$. 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 starting with a simpler case: functions of the form $f : S \to \mathbb{R}$ where $S \subset \mathbb{N}$ with $\#S < \infty$.

Without loss of generality, let $S = [\#S]$ and so we seek to show that for all $\epsilon > 0$ there exists an MLP with one hidden 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 $x \in [\#S]$,

$$f(x) = \sum_{s=1}^{\#S} f(k) \mathbb{1}(x = s)$$

$$= \sum_{s=1}^{\#S} f(k)(\mathbb{1}(x \geq s) - \mathbb{1}(x \geq s + 1))$$

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

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

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 $2 \cdot \#S$ nodes in the hidden layer approximate

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

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 $2 \cdot \#S - 1$ nodes 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

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

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

approximates the Heaviside function $H(x - n)$ well with some tiny leakage. To hit certain levels of precision, i.e. attaining the desired ϵ bound, simply change the -9 exponent to something smaller. 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 $\left\lceil \frac{s+1}{2} \right\rceil^{\text{th}}$ node 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.

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

If $f \in C(S)$ for $S \subset \mathbb{R}$ compact then (uniform continuity) 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

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

for all c, x in the same subinterval. 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 so 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 nodes 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.

Extending the idea to $S \subset \mathbb{R}^n$ 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 $\mathbb{R}^n$ and voilà.

Note: What this demonstrates is how an MLP *could* approximate functions in $C(S)$ for S 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 nodes. 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.

4.1.4 Advocating for depth > width

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

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

Very crudely supposing $n_0 = \cdots = n_{k+1} = n$ yields a complexity of

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

i.e. complexity grows linearly in the number of layers and quadratically in the number of neurons per layer. So we argue for increasing depth over width.

A perhaps more important result is from using MLPs for classification. Geometrically, they do this by splitting some 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. This generalises to

$$R_d(n) = \sum_{k=0}^{d} \binom{n}{k}$$

for d input nodes. So for $d = 1$ (so in $\mathbb{R}$), each new hyperplane (line) at most splits the subset of the real line into an additional subinterval, i.e. $R_1(n) = n + 1$. In general, $R_d(n) \approx n^d/d!$.

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.

There's something to be said about the fact that these are upper bounds and to what extent the bounds are tight in practice. Another worthwhile question is: to what extent do contemporary methods of learning the parameters of such architectures make use of this improved number of regions? Do gradient descent-based methods utilise this increase in number of regions perfectly? No, but the study of this sort of thing is worthy of a career so I won't go into more detail about it beyond interesting results, once I come across them.

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. The dataset from which the MLP is learned is constant, and so in deriving backpropagation the loss function is treated as a function of the weights and biases only, i.e.

$$\mathcal{L}\left(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(k+1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(k+1)}\right)$$

which we denote by just $\mathcal{L}$. 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

$$
\delta_i^{(k+1)} = \frac{\partial \mathcal{L}}{\partial z_i^{(k+1)}}
$$

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

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

Statement 2. First, note that

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

and so

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

which reduces the to-be-derived statement to

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

Using the chain rule, we deduce that

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

To complete our deduction, see that

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

from which we obtain

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

completing the deduction.

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

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

Statement 4. As before, see that

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

which yields

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

To complete the derivation, consider

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

4.3 Convolutional Neural Networks (CNNs)

Note: The terms 'kernel' and 'filter' are synonymous in this subsection.

Convolutional neural networks (CNNs) are neural networks whose architectures are designed to process image data. Illustrated in Figure 9, 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.
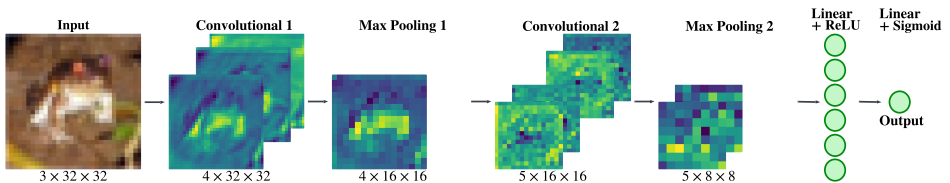
Figure 9: A binary classification CNN processing an image of a frog.

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.
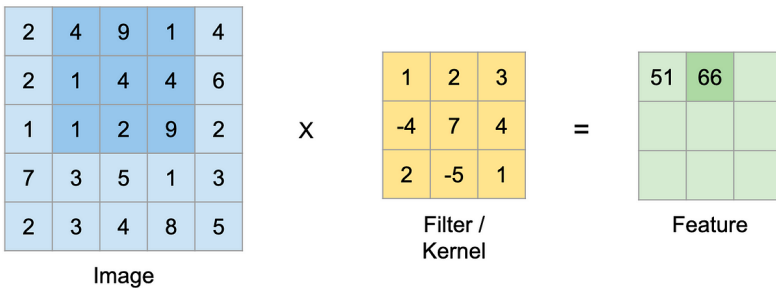
4.3.1 Convolutions

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. It's sometimes referred to as a kernel and I tend to unconsciously switch which I use. 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 kernel $K \in \mathbb{R}^{C \times k \times k}$ 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 kernel'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 10.


Figure 10: Convolution without bias.

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 9 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 11. Pooling has a ton of flavours: max, min, average and more.
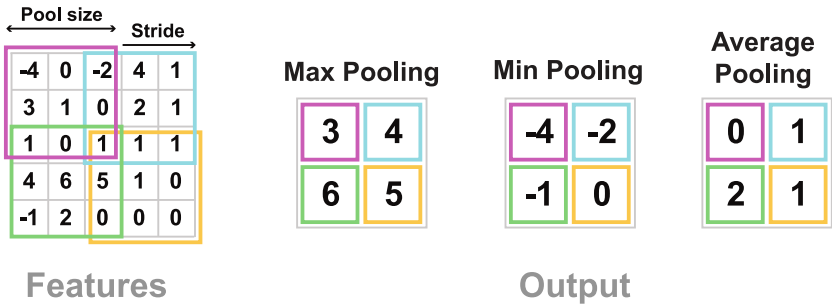
Figure 11: The core flavours of pooling.

I typically just pick max pooling and see how it does. 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 kernel $K \in \mathbb{R}^{k \times k}$ over F with uniform padding P and stride S. The precise dimensions W' and H' of the new feature map O are given by

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

and

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

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

4.4 Recurrent Neural Networks (RNNs)

. . .

4.4.1 Long Short-Term Memory (LSTMs)

. . .

4.4.2 Gated Recurrent Units (GRUs)

. . .

4.5 Autoencoders

An autoencoder $(\Omega_{\mathbf{X}}, d, \theta, \phi)$ consists of a sample space $\Omega_{\mathbf{X}} \subset \mathbb{R}^n$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, an encoder $\theta : \mathbb{R}^n \to \mathbb{R}^d$ and a decoder $\phi : \mathbb{R}^d \to \mathbb{R}^n$. 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. $d \ll n$. Of course, when $d \ll n$, 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.

Example 4.1 *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*
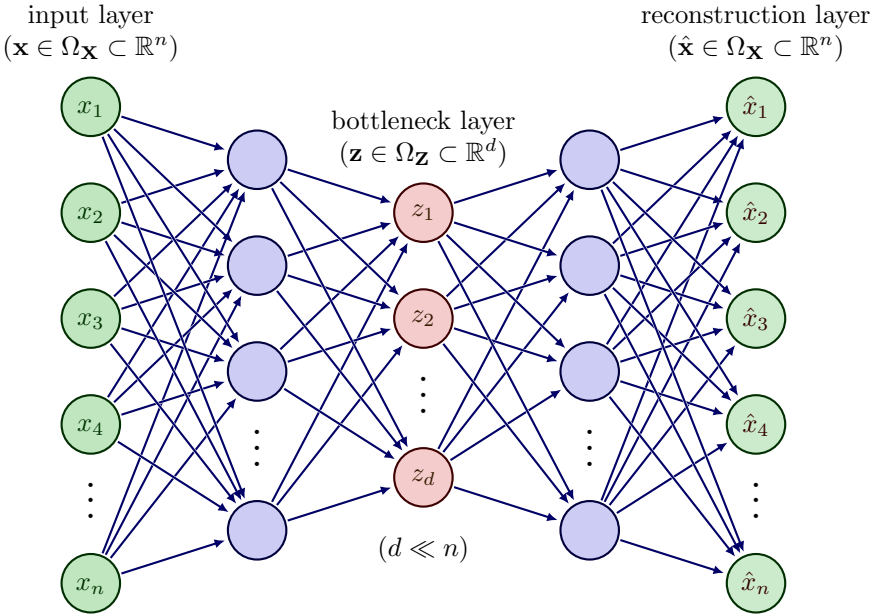
Figure 12: An autoencoder in which θ and ϕ are fit using multi-layer perceptrons (MLPs).

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 in compressing and then decompressing it 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 $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_\mathbf{X}$ and a latent dimension d, one might compute the pair $(\theta^*, \phi^*) \in \Theta \times \Phi$ which minimises the empirical risk over $\mathcal{D}$ where Θ and Φ are chosen function classes. That is, computing

$$(\theta^*, \phi^*) = \operatorname*{arg\,min}_{(\theta, \phi) \in \Theta \times \Phi} \sum_{i=1}^{M} ||\mathbf{x}_i - \phi(\theta(\mathbf{x}_i))||^2.$$
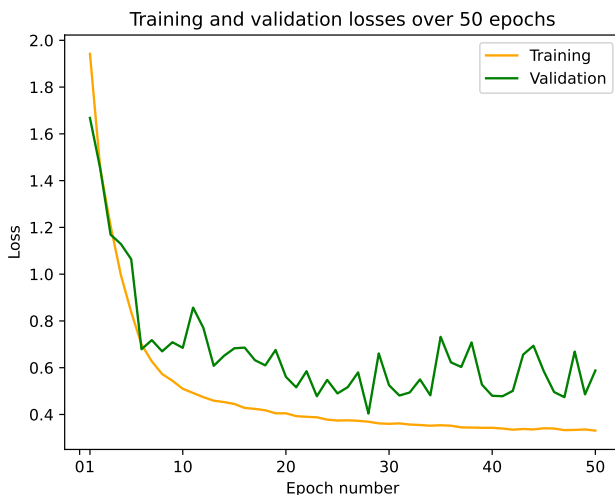
Figure 13: Training and validation losses over 50 epochs for some model.

If the function classes permit for computing the 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 classes of MLPs of appropriate input and output dimensions, as illustrated in Figure 12.

4.6 Misc. questions

Some questions to challenge one's understanding.

Q1: How does one prevent overfitting in NNs?

1) 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 decreasing then you're at risk of overfitting. For illustration, consider Figure 13 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 then you can stop 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.

2) Dropout: We don't want to be overly-reliant on any given subset of neurons. To prevent such dependencies, at each epoch, independently set the activation of each neuron to 0 with probability p. This way some portion of neurons are silenced during the training epoch. As a result, its corresponding parameters (weights and bias) are not updated during backpropagation — it is truly silenced. This idea is illustrated in Figure 14. Do not perform dropout when testing.


(a) Standard Neural Net   (b) After applying dropout.
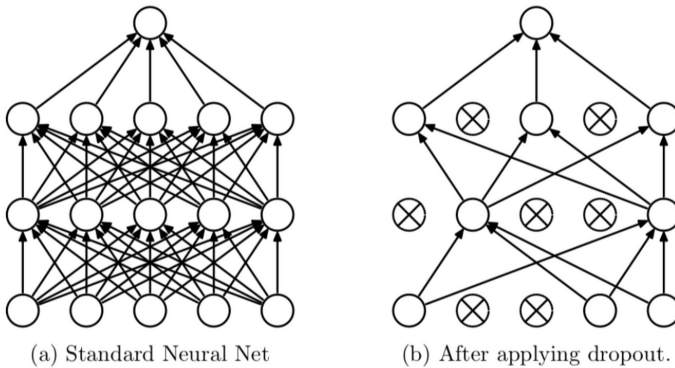
**Figure 14:** Dropout to prevent overdependence on given neurons.

Q2: 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 15.

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 16.
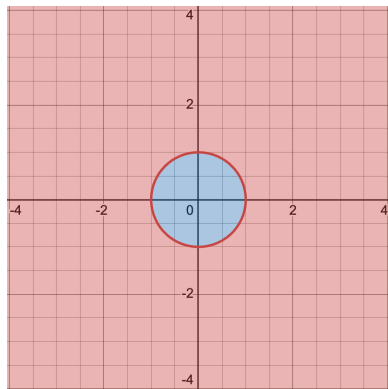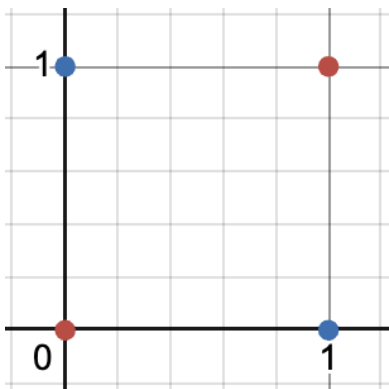
Figure 15: Examples of samples belonging to classes that are not linearly-separable.
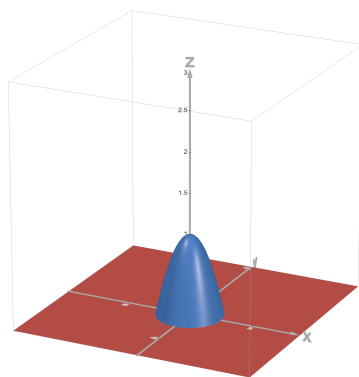
Figure 16: The image of our non-linear transformation.

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

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

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

Q3: 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, the model would effectively be a single linear layer into a non-linear transformation. The only way this could be effective would be if the non-linear transformation were effectively the function we're looking to model in the first place.

TODO: note that this is just a single-layer perceptron: most activation functions are monotonic, so single layer perceptrons reduce to, again, a hyperplane in feature space. like the first paragraph states, if we allow for arbitrary non-linear activation functions then this is possbile in theory (but unrealistic). monotonicity kills it

Q4: Must activation functions be monotonic?

Nope, it just makes things related to optimisation easier while having a sufficiently-small effect on practical performance. Additional minor point: some proofs of the universal function approximation theorem of MLPs necessarite monotonicity but these are typically intentionally short proofs; it isn't neceesary in general.

Q5: Which activation functions should be used when?

Roughly speaking, I wouldn't worry too much about this in practice. They're all nice and differentiable (with small caveats here and there, like with ReLU) allowing for the application of 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 a hyperparameterless model.

Small fun note, the universal function approximation of MLPs has been proven only for some select activation functions. I should look in more detail about this.

Q6: Why use a CNN over an MLP?

Suppose we have an MLP and a CNN, both trained to perform binary classification on images of cats and dogs. Further, suppose they perform equally. It might not be too surprisnig that the MLP will necessarily consist of a number of parameters that is orders of magnitude greater than for the CNN. Also, it's not so clear if a given MLP is translation invariant while

various components of a CNN architecture help alleviate fears of translation-related issues.

Also, improved learning-efficiency[4], i.e. fewer samples needed to lean distributions to similar degrees of precision.

Q7: How does one prevent vanishing/exploding gradients?

Through skip connections. 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

5 Deep Generative Models

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

---

[4]`https://ai.stackexchange.com/questions/27407/what-does-statistical-efficiency`

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 Variational Autoencoders (VAEs)

Motivating VAEs

In Subsection 4.5, autoencoders (AEs) were discussed. Autoencoders are great for data compression/decompression, denoising, interpreting complex models, etc. but we haven't seen yet how we might make use of them for generative tasks, i.e. sampling from the nasty/complex underlying distribution of the data they are trained on. An intuitive first idea is to train a compression/decompression autoencoder, randomly generate points in its latent space and feed them through the decoder, as in Figure 17. The outputs of the decoder should be similar to the training data, right? No.
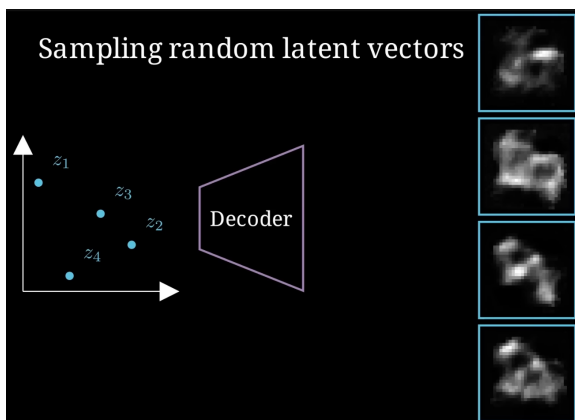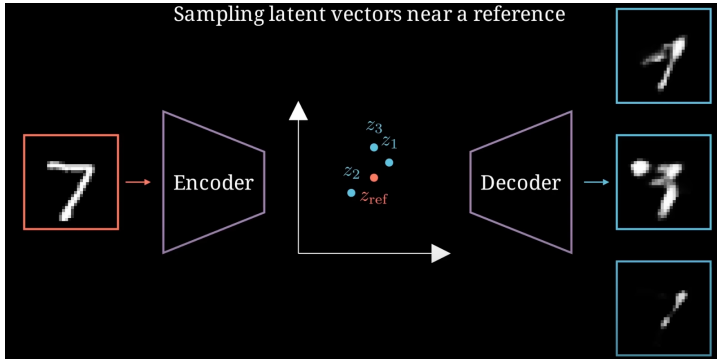

Figure 17: The output of the decoder of an autoencoder with randomly generated latent points as input. Gibberish output.

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 impose any sort of restriction on the structure of its latent space.

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 18? This

should result in decoder outputs that are similar in structure to the input sample but a bit different, right? No.


Figure 18: The output of the decoder of an autoencoder points relatively close to the latent embedding of a given training sample.

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 reconstructing the original training sample each time — certainly not what we mean when we say that we'd like to generate samples. So how do we do autoencoder-like things in a way that yields well-structured latent spaces? Variational autoencoders (VAEs)!

Formulating VAEs

Leaving aside, for now, how to learn one from data, VAEs can informally be seen as an extension of autoencoders in which the encoder and decoder each output parameters of a distribution belonging to some pre-chosen distribution family. Extending the notation used to define autoencoders, a variational autoencoder $(\Omega_{\mathbf{X}}, d, \mathcal{Q}_d, \mathcal{P}_n, \theta, \phi)$ consists of the sample space of the distribution of interest $\Omega_{\mathbf{X}} \subset \mathbb{R}^n$, 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 $n-$dimensional conditional distributions denoted $p_\phi(\mathbf{x}|\mathbf{z})$, an encoder $\theta : \mathbb{R}^n \to \mathcal{Q}_d$ 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
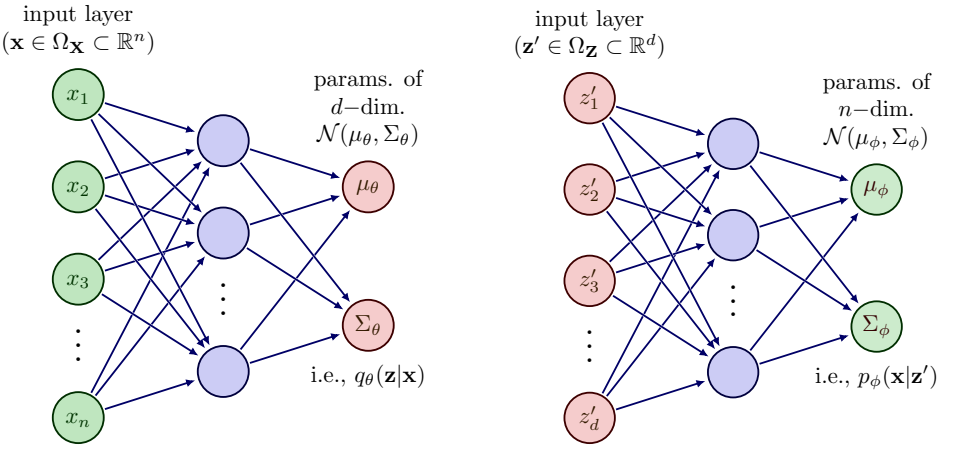
Figure 19: Left: an encoder which outputs parameters of the latent distribution $q_\theta(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\theta, \Sigma_\theta)$. Right: a decoder which outputs parameters of the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}') = \mathcal{N}(\mu_\phi, \Sigma_\phi)$ in which $\mathbf{z}' \sim q_\theta(\mathbf{z}|\mathbf{x})$.

yielding

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

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

As the encoder of a VAE yields a $d-$dimensional distribution $q_\theta(\mathbf{z}|\mathbf{x})$ 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 $n-$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 19.

Example 5.1 *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

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

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

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

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 $\mathcal{D} \subset \Omega_{\mathbf{X}}$.

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 $\mathbb{R}^n$. VAEs seek to remedy this.

To learn a VAE from data, we look to maximise the evidence lower bound (ELBO) over some dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$. 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 $\mathcal{D}$ can be seen as performing approximate maximum-likelihood estimation over $\mathcal{D}$ (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

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

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.5. 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 $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$, we learn a VAE by choosing function classes Θ and Φ (e.g. MLPs as in Figure 19) 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[5].
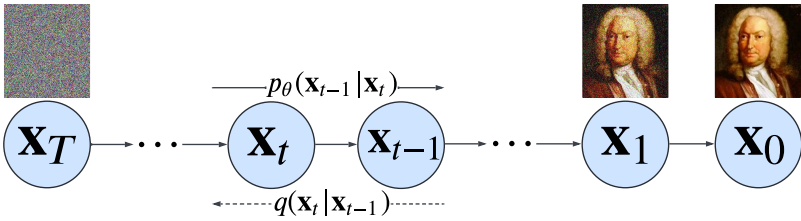
5.1.1 Backpropagation for VAEs

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!

5.2 Generative Adversarial Networks (GANs)

Compete!

5.3 Normalising Flow Models

Turn noise to samples!

Figure 20: Johann Bernoulli being denoised in line with the backward process of a diffusion model.

5.4 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

---

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

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).

5.4.1 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 21. This architecture can be thought of as forming a 'U'-like shape, with the bottleneck lying at the bottom, hence its name.
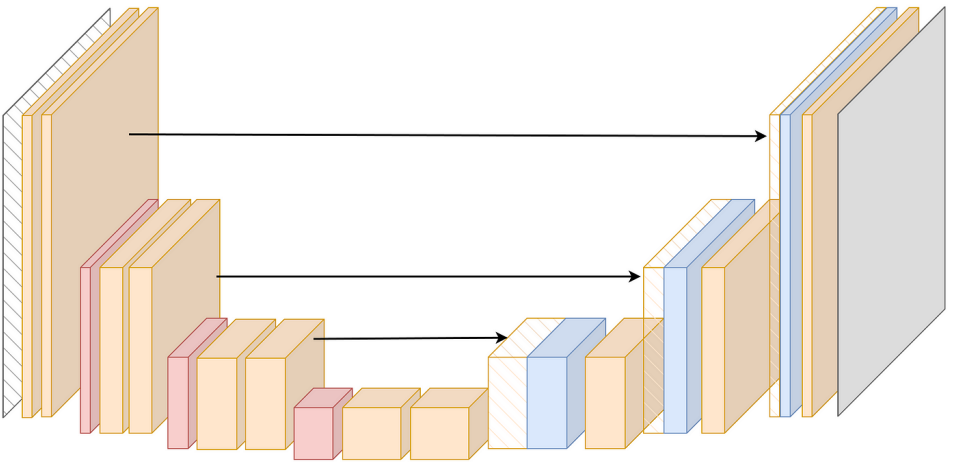
Figure 21: Example U-Net architecture.

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 21, 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.

5.5 Transformers

Stand at attention!

5.5.1 Token embedding

Embed the semantics of tokens in high-dimensional space:

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

So the embedding function E maps from the token vocabulary S to some high-dimensional space. If you first encode the tokens in the token vocabulary, e.g. via enumeration, then

$$E : \{1, \ldots, |S|\} \to \mathbb{R}^d$$

where $d \in \mathbb{N} \backslash \{0\}$ is the embedding dimension. What makes token embedding functions interesting is how well they can capture semantics of language.

5.5.2 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.

5.6 Misc. questions

Some questions to challenge one's understanding.

<div align="center">

Q1: **Blelele?**

</div>

. . .

6 Object Detection Models

CNNs and transformer-based architectures have revolutionised object detection. Three main flavours: R-CNN, YOLO and DETR. Before going into detail about them, what is object detection? What methods came before CNN/transformer-based architectures?

6.1 (Fast/Faster) R-CNN

R-CNN models has three flavours: R-CNN, Fast R-CNN and Faster R-CNN. R-CNN stands for *Regions with CNN features*. Roughly put, detecting objects within a given image using an R-CNN model consist of two parts:

1. Proposing regions of interest within the image

2. Classifying what is in each region (if there is something in the region)

Appendices

A Probability Theory Things

Here are some probability theory-related things, e.g. long derivations, which are well-suited for an appendix.

A.1 From Bernoulli to Binomial

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

$$\mathbb{P}(\mathbf{X} = k) = Z \cdot \prod_{i=1}^{n} \mathbb{P}(X_i = x_i)$$

$$= Z \cdot \prod_{i=1}^{n} p^{x_i}(1 - p)^{1-x_i}$$

$$= Z \cdot p^{x_1 + \cdots + x_n}(1 - p)^{n - (x_1 + \cdots + x_n)}$$

$$= Z \cdot p^k(1 - p)^{n-k}$$

where Z is a normalising constant accounting for the number of ways one can permute the tuple of realisations $(x_1, \ldots, x_n) \in \Omega_X^n$. So Z is just the number of ways of placing k-many 1s among a series of $n > k$ digits, i.e. $Z = \binom{n}{k}$, so

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

A.2 From Categorical to Multinomial

The categorical distribution is a generalisation of the Bernoulli distribution and the multinomial distribution is a generalisation of the binomial distribution. In line with this, we'd hope to be able to generalise 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 cth class in a realisation as the unit row vector $\mathbf{e}_c^{\text{T}}$, 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 easiest to understand for me.

Applying the same idea used to generalise 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 $\mathbb{R}^C$. 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 ith 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 jth column of $\mathbf{x}$. Note that k_j is the number of realisations of $X_1, \ldots, X_n$ in which the jth class is present. We have

$$\mathbb{P}(\mathbf{X} = (k_1, \ldots, k_C)) = Z \cdot \prod_{i=1}^{n} \mathbb{P}(X_i = (x_{i1}, \ldots, x_{iC}))$$

$$= Z \cdot \prod_{i=1}^{n} \prod_{j=1}^{C} p_j^{x_{ij}}$$

$$= Z \cdot \prod_{j=1}^{C} p_j^{x_{1j} + \cdots + x_{nj}}$$

$$= Z \cdot \prod_{j=1}^{C} p_j^{k_j}$$

$$= Z \cdot p_1^{k_1} \cdots p_C^{k_C}.$$

So all that's left to do is derive the normalisation constant Z. Note that the realisation $\mathbf{x}$ has n rows of which there are $n!$-many orderings. Additionally 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 $\binom{n}{k_1}$-many 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

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

from which we obtain

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

A.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 $\mathrm{Ber}(p)$ distribured. So if $X \sim \mathrm{NB}(r, p)$ then

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

An assignment 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 just

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

A.4 The Law of Large Numbers TODO

Suppose you have some i.i.d. samples $x_1, \ldots, x_n \sim p(x)$ then

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

A.5 Entropy

The entropy of a random variable can be motivated by the notion of the surprise of (or information learned from) observing assignments of said random variable. Given a discrete random variable X, an event $E \in \Omega_X$ and a surprise function $S : \Omega_X \to [0, \infty)$, the surprise of observing E is $S(E)$. Before we continue, we need to understand 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 $\mathbb{P}(E) = 0.01$ then we want $S(E)$ to be close to relatively high (strictly speaking it doesn't need to be bounded above) and if $\mathbb{P}(E) = 0.99$ then we want $S(E)$ to be close to 0.

An easy way to achieve this is to take $S(E) = \log(\mathbb{P}(E))$ where log denotes the natural logarithm. Doesn't really matter which base tbh. Quickly see that $\log(0.01) = 4.61$ and $\log(0.99) = 0.01$. From here, we define the entropy of a random variable as its expected surprise

$$H(X) = \mathbb{E}[-\log(\mathbb{P}(X))] = -\sum_{x \in \Omega_X} \mathbb{P}(x) \log(\mathbb{P}(x)).$$

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 $E_1, E_2 \in \Omega_X$, $S : \Omega_X \to [0, \infty)$ must satisfy

- $S(1) = 0$

- $\mathbb{P}(E_1, E_2) = \mathbb{P}(E_1) \cdot \mathbb{P}(E_2) \implies S(E_1, E_2) = S(E_1) + S(E_2)$

- $\mathbb{P}(E_1) > \mathbb{P}(E_2) \implies S(E_1) < S(E_2)$

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

Note 1: It's clear from the second condition that this surprise function is actually a function whose domain is the powerset of Ω_X but addressing this detail isn't worth it — the idea being conveyed is hopefully clear regardless.

Note 2: You can see entropy as pertaining to a random variable or of a distribution of said random variable. I should formalise this at some point.

A.5.1 Kullback–Leibler Divergence (KL-divergence)

Given probability density/mass functions p and q on the same space Ω, their Kullback–Leibler divergence (KL-divergence) is given by

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

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.

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

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

where $H(p,q)$ denotes the cross-entropy between p and q while $H(p)$ 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 $H(p,q)$.

A.5.2 Distribution fitting using cross-entropy

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

$$p : \mathbb{N}\backslash\{0\} \to [0,1]$$
$$k \mapsto 2^{-k}$$

by finding a nice λ for

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

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

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

where $C = \sum_{k=1}^{\infty} \frac{\log(k!)}{2^k}$ is independent of λ. In computing the minmum of $H(p,q;\lambda)$ in λ we compute

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

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

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

which seems pretty low. Good fit!

A.5.3 Distribution fitting using maximum likelihood estimation TODO

Suppose instead of knowing the nice geometric distribution from subsubsection A.5.2 we only had n i.i.d. samples drawn from it denoted $\mathcal{D} = (k_1, \ldots, k_n) \in \mathbb{N}^n$. The MLE of the 'true' k of $\mathrm{Poi}(k)$ is given by

$$
k_{\mathrm{MLE}} = \frac{1}{n} \sum_{i=1}^{n} k_i.
$$

The reason I write 'true' in the previous sentence is that of course we know that these samples were not drawn from a Poisson distribution at all. This is a nice example of imperfections in model assumptions. Is a Poisson distribution what gave rise to our samples? Is the class of functions given by an MLP with a fixed number of layers and nodes in each layer the function that gave rise to our samples? We're not god afterall.

Anyway, by the law of large numbers, we know that

$$
k_{\mathrm{MLE}} \xrightarrow{n \to \infty} \mathbb{E}[K] = 1.
$$

From what I understand, the fact that minimising cross-entropy yields the same estimate as MLE with $n \to \infty$ here is a special case of things learned

in generalised linear models. Some sort of result about distributions from exponential families. Sadly, I never took the GLMs course at my university by Wim Krijnen so it isn't obvious to me.

A.6   The Central Limit Theorem (CLT) TODO

. . .

B   Classical Statistics Things

I should have a far more concrete understanding of classical statistics tbh. Such is life.

B.1   Independent samples

Suppose we'd like to get an idea of the average height of men and women in our population. If we randomly sample 100 men and 100 women then we're good to go: we have independent samples. If instead we randomly sample 100 couples then our samples would not be independent since couples' heights correlate.

This is important because when performing maximum likelihood, the assumption of independence of samples is key to writing the likelihood of the set as a product of individual probabilities. Without this independence, the likelihood would be a horribly complex function that we would not be able to utilise. Independence is also assumed in a ton of other statistics-related things like t-tests, ANOVA, etc.

C   Philosophy Things

I don't think about the philosophic side of maching learning very often but I find some topics interesting.

C.1   Thoughts on reasoning (Summer 2024)

At the time of writing, machine learning, as a term, is used almost inter-changeably 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. This is probably because it's a tricky term to arrive at an agreed upon definition of.

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 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 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 'understands' why it arrived at 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, understand and knowing aren't too important in the short term. While I definitely fall into the bucket of machine learning people who only care about problem solving in the short term, I do sometimes think about these things. 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 complain that they aren't truly capable of reasoning like us.