Machine Learning Explainers

Last updated: May 22, 2025

Abstract

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

Contents

1 Introduction

I personally think of machine learning as modelling the relationship between model variables $\mathbf{X}$ and output variables $\mathbf{Y}$, be the relationship probabilistic or not, in line with the demands of the task to be solved. Vaguely put, this is done by constructing some parameterized representation f_θ of their relationship. Some parameters do a better job than others, according to some metric of goodness, so we'd ideally have the optimal parameters θ^*.

What makes machine learning fun is that, for whatever reason, these learned representations of underlying relationships can be used to infer things about the relationship of $\mathbf{X}$ and $\mathbf{Y}$ very well. The fact that this is possible to at least a small extent isn't too surprising, but, nowadays, this is possible to the extent of being able to do things like input natural language to output a photo-realistic image described by said input. Why is this possible? The more I've learned, the more I'm amazed that our methods work as well as they do. Take denoising diffusion models for image generation. It effectively learns to transform randomly sampled Gaussian noise into something convincingly belonging to the distribution pertainintg to the data that the model was trained on. So if I train such a model on a large dataset of images of cats then it learns to transform Gaussian noise into new images of cats that are sufficiently distinct from those it was trained on. That is, such models do not necessarily need to overfit to be able to output samples convincingly belonging to the target distribution. Why are diffusion, flow-based models, GANs, VAEs, probabilistic circuits or attention-based models so effective? It's sometimes hard to believe, even after working with some of these architectures extensively and understanding them in detail.

Thoughts on 'reasoning'

As a term, machine learning is used almost interchangeably with AI as of now (summer of 2024). AI isn't particularly well-defined in itself, even in academic settings, so its use in day-to-day things, like the news, can be confusing. I think of machine learning as what's written above, effectively in line with statistical learning theory. 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 agreeable notion of 'reasoning'. A common issue found in discussions surrounding this topic is that a definition of 'reasoning' is left out. In fairness, it's a tricky thing to define precisely. For a long time, people considered the Turing test to be a metric in line with

the idea of machines 'reasoning'. Current LLMs are far beyond passing the Turing test but still it is argued that they don't reason, which I think I agree with as of now, but it's worth being conscious of the fact that we have effectively moved the goal posts. A similar example of this is the Automatic Statistician: a model that, given a bunch of data, can effectively output a report on the likely relationships between different parts of the data[1]. Essentially a data scientist or statistician in the form of a machine, which we're much closer to than in say 2020 due to LLMs.

Right now, I'm unaware of any machine learning model that can 'reason' by any agreeable notion of the term. Let's say that 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 $f(b) > 0$. Is this 'reasoning' or do we instead brush this off as the model having been trained on a bunch of examples of the use of 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 IVT - we can construct many such examples. What I think makes IVT interesting is that we can effectively express what it means to understand it as a human. I'd say that the distinction between ChatGPT's 'understanding' here and a human's 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, to the top half of the page without taking their finger off the paper then their finger must have touched the horizontal line at some point, i.e. the function must have intersected the x-axis during this continuous traversal. I believe that humans understand this due to just experiencing the real world. You don't need any mathematical training to know and, more importantly, understand 'why' your finger crosses through this horizontal line on the page. I'm certain that current GPT models do not 'understand' the IVT in this way, or really any way that isn't purely formally mathematical.

A follow up question is to what extent it matters that a model 'understands' how it arrives 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 $3 \cdot 2$. Maybe 50 years from now we'll be well-taken care of by a fleet of machine learning-based robots and we'll still complain that they aren't truly capable of reasoning like us. Hopefully not. I wish industry research in certain areas of machine learning, like LLMs and image/video generation, would be very heavily regulated. Perhaps forcefully stopped entirely. Of course, this will never happen due to countries' desire to be ahead

[1] `https://www.youtube.com/watch?v=aPDOZfu_Fyk`

of their competitors.

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 binary or multi-class 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

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 the data.

In fairness, from what I understand, using the least squares method derived below for all such regression models linear in their parameters is viable.

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

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

are the feature values of the i^{th} 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)

$$\begin{aligned} f_\theta : \Omega_{\mathbf{X}} &\to \Omega_Y \\ \mathbf{x} &\mapsto \theta^{\mathrm{T}} \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} = \theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p \end{aligned}$$

where $\theta = (\theta_0, \theta_1 \dots, \theta_p) \in \mathbb{R}^{p+1}$ 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 $\tilde{y} = f_\theta(\tilde{\mathbf{x}})$.

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

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

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

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

which pertains to taking $d(f_\theta(\mathbf{x}_i), y_i) = (y_i - f_\theta(\mathbf{x}_i))^2$. I'll still refer to it as the 'mean' square error, even though I don't divide by n, simply because minimising it as written above is equivalent to minimising the same expression with division by n included - it's mostly a notational convenience thing. So in our case, we wish to find

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

We know that in the context of linear regression, the optimal paramters θ^* are typically taken to be those which minimise the mean square error over $\mathcal{D}_{\text{train}}$ 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 i^{th} sample. To find the minimiser(s) of $\text{MSE}(\theta)$, i.e. the optimal parameters θ^*, we compute its gradient with respect to θ, 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

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

and so the optimiser θ^* is given by

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

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 via the CLT. Since $f_\theta(\mathbf{x}_i)$ is a constant, we have $y_i|\mathbf{x}_i \sim \mathcal{N}(f_\theta(\mathbf{x}_i), \sigma^2)$ and so

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

By the law of maximum likelihood[2], the maximum likelihood estimator θ_{MLE} for the parameters of our linear regression model is that which maximise our log-likelihood. That is,

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

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

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.

[2] maximum likelihood link

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.

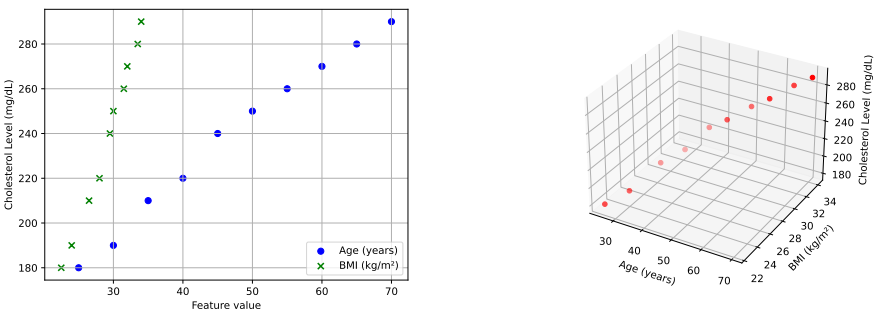

Figure 1: Each feature against the output (left) and the dataset in the feature space (right).

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 write out the optimal parameters θ^* by hand. We have

$$y = \begin{bmatrix} 180 \\ 210 \\ 240 \\ 260 \\ 280 \end{bmatrix}, \quad X = \begin{bmatrix} 1 & 25 & 22.5 \\ 1 & 35 & 26.5 \\ 1 & 45 & 29.5 \\ 1 & 55 & 31.5 \\ 1 & 65 & 33.5 \end{bmatrix}$$

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

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

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

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 **??**. I should improve these at some point.

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/

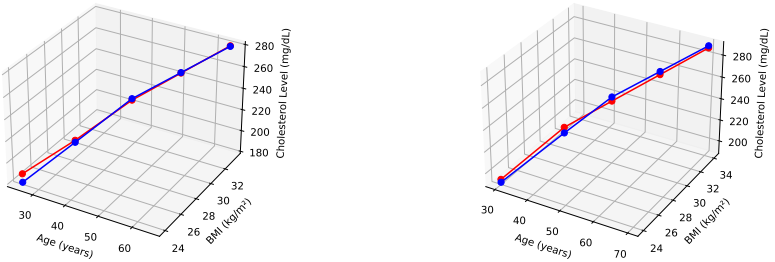

Figure 2: Model performance on the training set (left) and testing set (right).

2.1.3 Regularisation

Ridge regression intentionally introduces bias with the intention of reducing variance and improve accuracy!

2.1.4 R^2 value

...

Assumption of normally distributed residuals

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

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

2.1.5 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 $Y = \alpha + \beta Y + \epsilon$ 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, $C_\theta(\mathbf{x}) = I(h_\theta(\mathbf{x}) > p)$ where I denotes the indicator function. 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

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

Note: 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 in 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 more of a classic approach of 'maximise this thing according to well-defined contraints'.

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 i^{th} sample's features and $z_i \in \{-1, 1\}$ its class such that z_i is 1 if $\theta_+^{\mathrm{T}}\mathbf{x}_i + \theta_0 > 0$ and 0 otherwise.

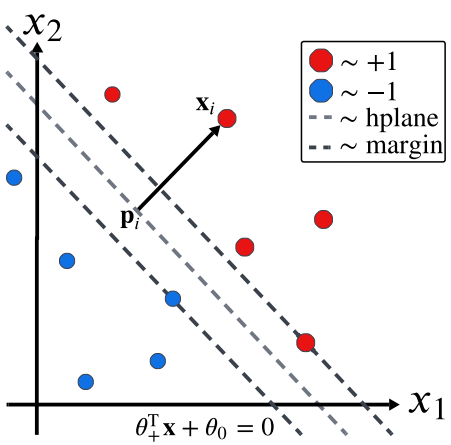

Figure 3: A hard-margin SVM in two dimensions.

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

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

which, after some rearranging, yields

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

Taking a point $\mathbf{x}_i$ whose class is -1 yields $d_i = -\frac{\theta_+^{\mathrm{T}} \mathbf{x}_i + \theta_0}{||\theta_+||}$ and so a neater way of writing this distance for an arbitrary sample $\mathbf{x}_i$ is $d_i = \frac{z_i(\theta_+^{\mathrm{T}} \mathbf{x}_i + \theta_0)}{||\theta_+||}$. Let $d_{\min} = \min_{i=1,\ldots,n} d_i$. The idea behind hard-margin SVMs is only applicable to entirely linearly sparable data and effectively boilds down to finding which parameters maximise $d_{\min}$. That is, we look to compute

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

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

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

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.

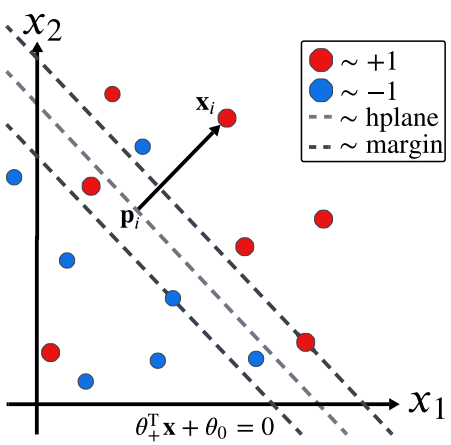

Figure 4: A soft-margin SVM in two dimensions. Samples are labelled according to their true class.

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

...

2.4 Decision Trees and Random Forests

- split according to some feature
- i love democracy, prune them

2.5 Handcrafted Features

...

2.6 The Bias-Variance Trade-Off

...

2.6.1 Double descent

...

2.7 Misc. questions

Some questions to challenge one's understanding.

Q1: What are the benefits of using logistic regression over more sophisticated methods?

- Its implementation and interpretation are straightforward
- If offers a notion of classification confidence which can be useful
- Statistical tests can be conducted on parameters, e.g. statistical significance tests
- mention more classical statistics approaches like LDA and QDA

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

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

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[a], a good method for this is to simply fit a linear and a non-linear model (e.g. a cubic spline smoother model) and see which explains a larger amount of the variance of the output via ANOVA tests.

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

3 Gradient Descent and its Optimisation

...

3.1 Gradient Descent

How do we find such an appropriate set of weights and biases for a neural network? More generally, how do we find an appropriate set of parameters for a machine learning model? A popular method is that of gradient descent.

3.2 Momentum

...

3.3 Hyperparameter Tuning

cross-validation and shiiiiiiieeet

3.4 Misc. questions

Some questions to challenge one's understanding.

Q1: ?

...

3.4.1 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*. That'll come later.

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 are trying to solve the optimisation problem given by

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

where $\eta > 0$. Approximating $f(\mathbf{x}+\eta\mathbf{v})$ by its Taylor approximation

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

for small $\eta > 0$, yields

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

and so 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 is clearly maximised when $\cos(\theta) = 1$, i.e. when $\theta = 0$, and $\mathbf{v}$ has 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 $\nabla f(\mathbf{x}) = 0$ of course corresponds to $f(\mathbf{x})$ being a local maximum.

Note: This argument, which utilises a Taylor approximation, relies on a sufficiently small $\eta > 0$. 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.

3.4.2 Choosing a loss function

Note that these depend on having uniform diversity of classes. If not then use weighted cross-entropy loss functions. There are a ton of machine learning-related scenarios that would need a specially chosen loss function but the list below is a fine start. 123

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 choices.

3.4.3 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.

3.4.4 How should we initialise the parameters?

...

4 Neural Networks

It is often the case that what people mean when they state 'neural networks' they more precisely mean 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 when all we really want to do with them is solve tasks in an engineery way.

4.1 Multi-Layer Perceptrons (MLPs)

MLPs are computational models inspired by the human brain[3] consisting of fully-connected layers of nodes, typically illustrated in a left-to-right fashion, to process data in a way that allows one to solve tasks like classification and regression. They are powerful tools that excel at complex tasks as a result of being relatively expressive-efficient universal function approximators. Their training can be very computationally demanding and they are tricky to interpret despite being perhaps the simplest example of a neural network. Regardless, when something works sufficiently effectively, as engineery people we often care less about why.

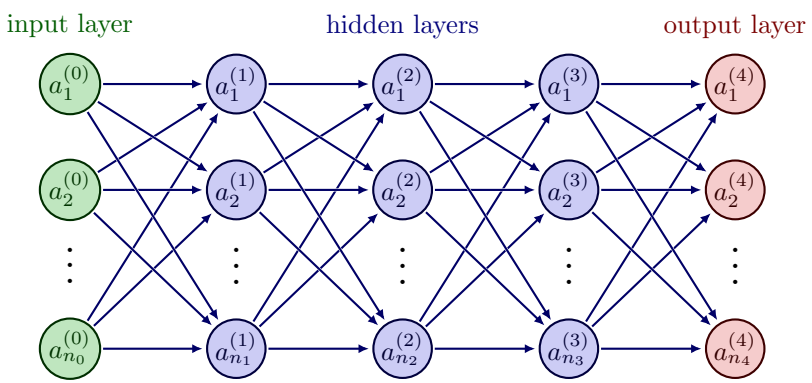

Figure 5: A multi-layer perceptron (MLP) with $k = 3$ hidden layers.

An MLP consists of an input layer (where data is input), hidden layers and an output layer. Each layer is made up of a number of neurons and every neuron in a non-input layer has an associated set of weights, a bias and an activation function.

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

For ease of notation, given an MLP with k hidden layers, denote the index of the input layer by 0, the output layer by $k+1$ and by extension the j^{th} layer by $j \in \{0, \ldots, k+1\}$. Additionally, consider the following -

- Let $n_j \in \mathbb{Z}_{\geq 1}$ denote the number of neurons in layer j.
- Let $a_i^{(j)}$ denote neuron i of layer j.
- Let $w_{i,l}^{(j)}$ denote the weight of neuron i of layer $j \in \{1, \ldots, k+1\}$ associated with neuron l of layer $j-1$. Note that, overall, neuron i of layer j has associated weights $w_{i,1}^{(j)}, \ldots, w_{i,n_{j-1}}^{(j)}$.
- Let $b_i^{(j)}$ denote the bias of neuron i in layer $j \in \{1, \ldots, k+1\}$.
- Let $\sigma_j : \mathbb{R} \to \mathbb{R}$ denote the activation function of layer $j \in \{1, \ldots, k+1\}$.

From here we can express the value of any neuron in a non-input layer in terms of the values of the neurons belonging to its preceding layer as

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

for $j \in \{0, \ldots, k\}$. The reason for writing the second equality above, involving the dot product of two vectors, is that using matrix-vector notation, we can write an elegant and compact expression for the values of all nodes in any non-input layer in terms of the values of the neurons belonging to its preceding layer as

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

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.

Note: We see already that $\mathbf{W}^{(j+1)} \in \mathbb{R}^{n_{j+1} \times n_j}$ and that $\mathbf{a}^{(j+1)}, \mathbf{b}^{(j+1)} \in \mathbb{R}^{n_{j+1}}$ for $j \in \{0, \ldots, k\}$ where, as earlier, k is the number of hidden layers of the neural network.

4.1.1 Constructive example - The algebra

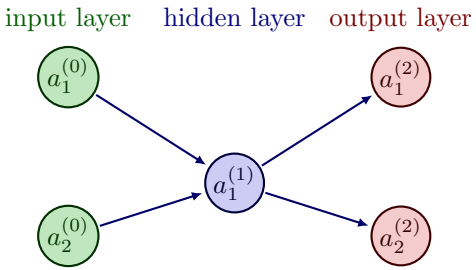

Figure 6: A neural network with $k = 1$ hidden layer.

Consider the neural network in Figure 6 whose input, hidden and output layers consist of two, one and two neurons respectively. With such a simple neural network, we may explicitly express the output neurons $a_1^{(2)}$ and $a_2^{(2)}$ in terms of the input neurons $a_1^{(0)}$ and $a_2^{(0)}$. We see that $n_0 = 2$, $n_1 = 1$ and $n_2 = 2$ and so $\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \end{bmatrix}$ and $\mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} \\ w_{2,1}^{(2)} \end{bmatrix}$. Noting additionally that $\mathbf{b}^{(1)} = b_1^{(1)}$ and $\mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$ we can explicitly express the activation of the single neuron in the hidden layer as

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

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

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

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

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

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 6. For the weights and biases, take

$$\begin{aligned}
w_{1,1}^{(1)} &= 0.5 & w_{1,1}^{(2)} &= 0.7 \\
w_{1,2}^{(1)} &= -0.5 & w_{1,2}^{(2)} &= 0.3 \\
b_1^{(1)} &= 0.1 & b_1^{(2)} &= 0.2 \\
& & b_2^{(2)} &= -0.2
\end{aligned}$$

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.

4.2 Backpropagation

Chain rule/automatic differentiation

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 specially fit to process image data. Illustrated in Figure 7, their architecture can be described on a high level as consisting of two parts. The first part essentially performs feature extraction. The second part, consisting of a small MLP, uses these extracted feature values to produce the overall model's output. Feature extraction into MLP, pretty intuitive.

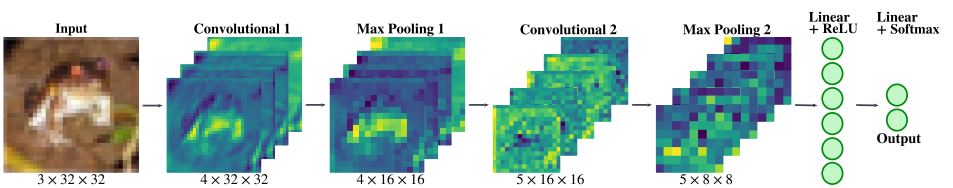

Figure 7: A binary classification CNN processing an image of a frog.

Before these deep learning architectures, researchers would have to perform feature extraction and feed the extracted feature values to a chosen model, e.g. an SVM for classification. The disadvantage of this is that one would have to pick a feature extraction method and a model to go along with it. Which feature extraction algorithms work best for a given model? This is annoying to have to do. To alleviate this, deep learning architectures streamline the process. A huge advantage is that, in some sense, these deep learning models know what features are best for the 'prediction' part of the model.

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 pixels it traverses over. This square matrix is called the kernel or filter (synonymous and I switch a lot). You can think of this convolution operation as a sort of dot product between the filter and the 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 8.

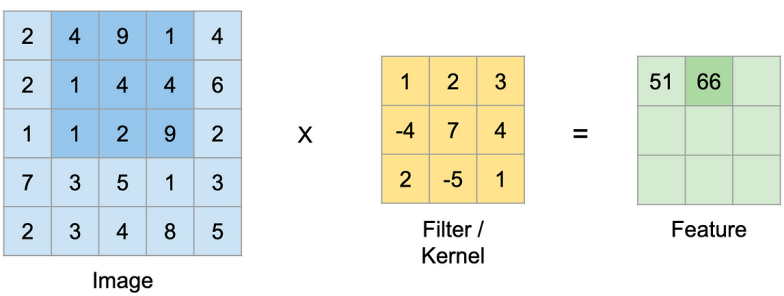

Figure 8: Convolution without bias.

Generally, earlier convolutions in a CNN architecture extract higher level features, like edges and textures, while later convolutions extract lower level features that are usually beyond human interpretation. This is observed in Figure 7 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 9. Pooling has a ton of flavours: max, min, average and more.

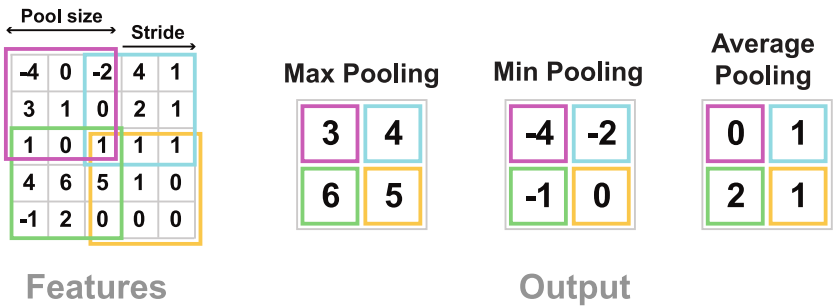

Figure 9: The core flavours of pooling.

I typically just pick max pooling and see how it does. A nice benefit of pooling is improved robustness to translations of images. 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 core benefit is the reduced compute.

Output dimensions of convolutions 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. Maybe worth mentioning that some frameworks allow going beyond the edges of a feature map when convolving, e.g. Pytorch's `MaxPool2d` has a `ceil_mode` flag that determines whether the does precisely this.

4.3.3 Example

CNNs are best understood by example. Consider the nice frog in Figure 7. The input to the model is a 32×32 RBG image, so $3 \times 32 \times 32$ accounting for the three colour channels. This input is fed into the first layer of our CNN which is a convolutional layer. This convolutional layer has 32 filters/kernels each of which is $3 \times 3 \times 3$. I prefer to think of each of these kernels as three separate 3×3 kernels, one for each colour channel. For a given kernel triplet, each is slid over the 32×32 image corresponding to their channel, each yielding a 30×30 output. Since each kernel triplet outputs three such 30×30 outputs, we sum them to obtain the overall feature map corresponding to this kernel.

There are 32 kernel triplets in our case and so you can think of the output of this layer as 32 images each of which are 30×30 in spatial dimension. Alternatively, you can think of the output as a single tensor of dimensions $32 \times 30 \times 30$. Also, each is passed through a non-linear activation function purely for the purpose of sprinkling in some non-linearity.

4.4 Recurrent Neural Networks (RNNs)

...

4.4.1 Long Short-Term Memory (LSTMs)

...

4.4.2 Gated Recurrent Units (GRUs)

...

4.5 Autoencoders

Autoencoders are a nice precursor to variational autoencoders (VAEs). Essentially, autoencoders are neural nets in which you have a bottleneck layer in the middle of the network which compresses inputs in a way that allows for the reconstruction after decoding. So an encoder can be written as a

function $\mathcal{E} : \Omega_{\mathbf{X}} \to \mathbb{R}^d$. Similarly, a decoder can be written as a function $\mathcal{D} : \mathbb{R}^d \to \Omega_{\mathbf{X}}$. Ideally, for all $\mathbf{x} \in \mathbf{X}$ we have $\mathcal{D}(\mathcal{E}(\mathbf{x})) = \mathbf{x}$ but of course in practice it is only practical to achieve $\mathcal{D}(\mathcal{E}(\mathbf{x})) \approx \mathbf{x}$.

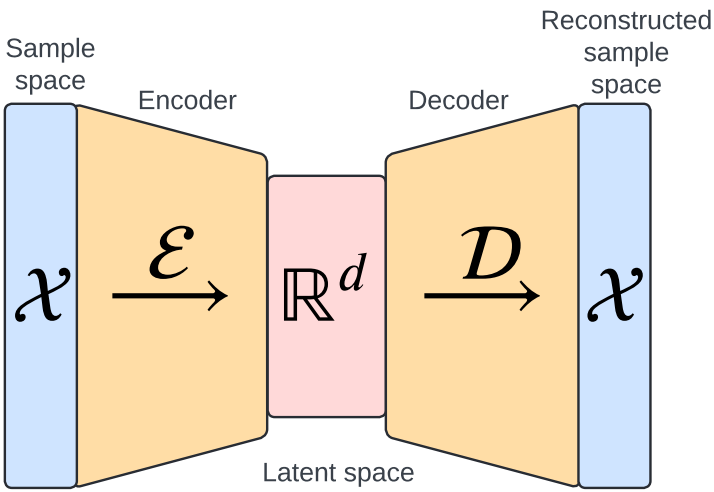

Figure 10: An autoencoder mapping from the sample space $\Omega_{\mathbf{X}}$ to the latent space $\mathbb{R}^d$ then back to $\Omega_{\mathbf{X}}$.

The encoder architecture is typically pretty straightforward: some neural net with an input layer consisting of a number of neurons that matches the dimension of $\Omega_{\mathbf{X}}$, some hidden layers and an output layer with d neurons matching the dimension of the latent space $\mathbb{R}^d$. Similarly, the decoder architecture consists of an input layer of d neurons, some hidden layers and an output layer of $\dim(\Omega_{\mathbf{X}})$ neurons. So the idea is simple: train the encoder $\mathcal{E}$ and decoder $\mathcal{D}$ such that $\mathcal{D}(\mathcal{E}(\mathbf{x})) \approx \mathbf{x}$ for all $\mathbf{x} \in \Omega_{\mathbf{X}}$ where $\dim(\mathrm{Im}(\mathcal{E})) << \dim(\Omega_{\mathbf{X}})$.

4.5.1 Example autoencoder (visualisation bonus)

Consider the MNIST dataset, bunch of handwritten digits. Let's learn an autoencoder in which the latent space is $\mathbb{R}^2$. A nice advantage of picking latent spaces whose dimension is 1, 2 or 3 is that we can visualise the mappings of samples in $\Omega_{\mathbf{X}}$ to the latent space and see if there is some structure

there, e.g. we can inspect if it separate them nicely.

Our encoder architecture will have an input layer consisting of 784 nodes, matching the size of samples in MNIST, two hidden layers of ? neurons each and an output layer consisting of two nodes (tight bottleneck). The decoder will then have an input layer of two nodes, two hidden layers of ? neurons each and an output layer consisting of 784 nodes.

4.6 Skip Connections/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 identity and learning from their is feasible

4.7 Misc. questions

Some questions to challenge one's understanding.

Q1: How does one prevent overfitting in NNs?

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 11 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

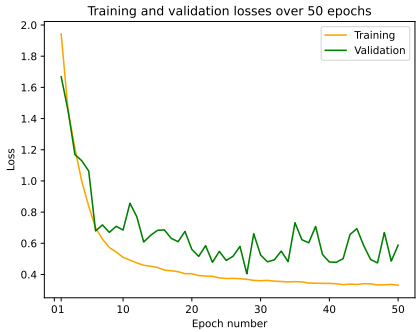

Figure 11: Training and validation losses over 50 epochs for some model.

training process and the parameters are saved in a `.ckpt` file.

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 12. Do not perform dropout when testing.

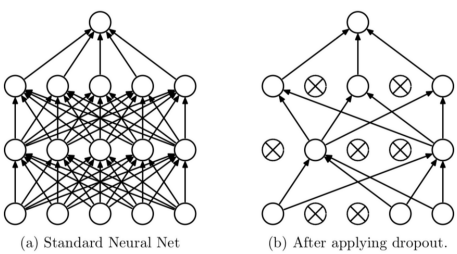

Figure 12: Dropout to prevent overdependence.

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.

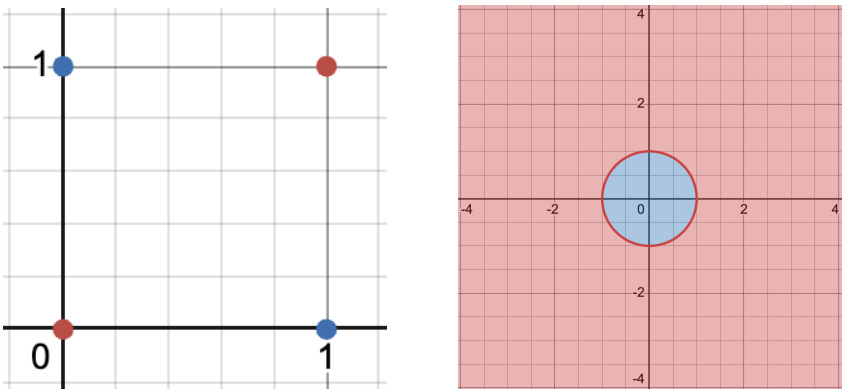

Figure 13: Examples of samples belonging to classes that are not linearly-separable.

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 14.

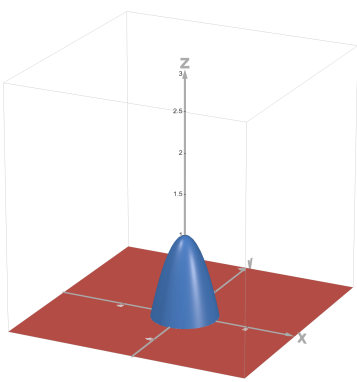

Figure 14: The image of our non-linear transformation.

From here, a natural linear decision boundary is just the plane $z = 0$. 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 just define $D(0,0) = 0$ to avoid the annoying pole.

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.

note that this is just a single-layer perceptron: most activation functions are monotonic, so SLPs 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?

No, just makes it much easier to optimise when they are.

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.

On top of this, pixel values are very usually correlated, so why use an MLP in which many nodes are redundant? CNN's take care of this perfectly. They were handcrafted for processing image data and so they are naturally far superior.

5 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 training data particularly well then samples from the model should be sufficiently convincing. That is, if I get a nice fit for the distribution of images of cats lounging in the sun then samples from my fit distirubtion should look convincingly like cats lounging in the sun.

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

5.1 Variational Autoencoders (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 15. The outputs of the decoder should be similar to the training data, right? No.

Disaster strikes and we begin to see just how unstructured the latent space of an autoencoder is: randomly sampling from it (whatever this may mean) results in nonsense outputs from the decoder because most of the latent space itself is meaningless (nothing is encoded to most of it, so in some sense a lot of it is 'un-utilised'). This is perhaps unsurprising as at no point during training an autoencoder do we 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 16? This should result in decoder outputs that are similar in structure to the input sample but a bit different, right? No.

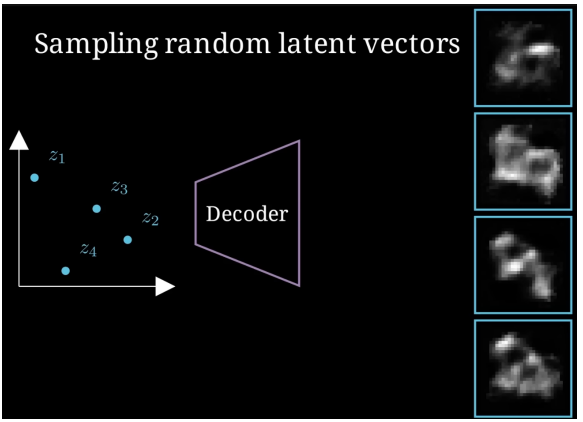

Figure 15: The output of the decoder of an autoencoder with randomly generated latent points as input. Gibberish output.

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.

Queue variational autoencoders (VAEs), illustrated in Figure 17. Loosely put, they have a somewhat similar structure to regular autoencoders in that they have an encoder and a decoder but they are distinguished by two things. First, they impose structure on the latent space which helps very much with making the ideas presented earlier feesable. For example, points that are close in feature space are also close in the latent space and all of the latent space is made use of, none of it meaningless. Second, given fixed inputs, the outputs of the encoder and decoder are not fixed points (hence variational). Instead, the outputs are parameters of sampleable distributions.

5.1.1 think of different subsubsection name

Denote the parameters of the encoder by ϕ and the paramaters of the decoder by θ. Note that sometimes I will write $\phi(\mathbf{x})$ to denote the output of the encoder, which is a little odd as ϕ denotes the encoder's paramteres but it makes notation a little easier. Anyway, the role of the encoder and decoder in VAEs is to output paramters of sampleable distributions. For example, in the vanilla formulation, a VAE's encoder takes in some sample $\mathbf{x}$ and outputs the parameters μ_ϕ and Σ_ϕ (usually diagonal) of a normal

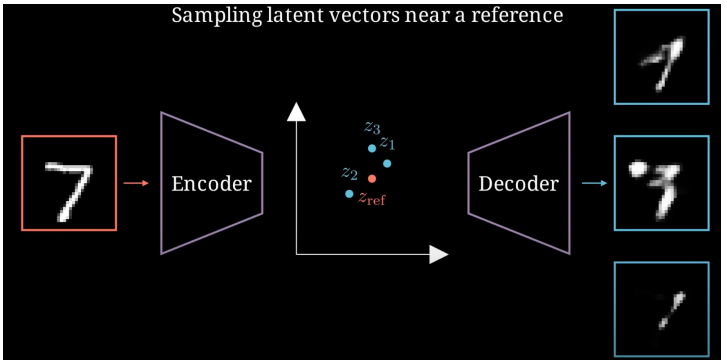

Figure 16: The output of the decoder of an autoencoder points relatively close to the latent embedding of a given training sample.

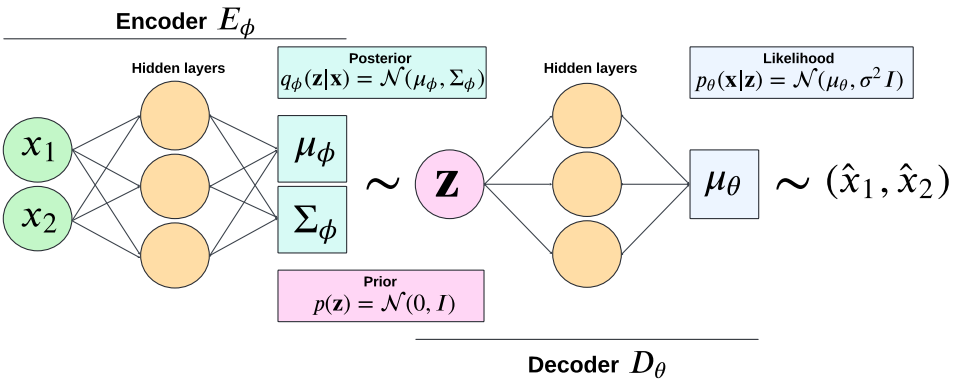

Figure 17: Example architecture of a variational autoencoder in which the input distribution is two-dimensional.

distribution which approximates the posterior distriubtion $\mathbf{Z}|\mathbf{x}$. If we want to reconstruct $\mathbf{x}$, we sample from $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\phi, \Sigma_\phi)$ and feed it as input to the decoder which outputs the mean μ_θ of $p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mu_\theta, \sigma^2 I)$ where σ is a hyperparameter. Sampling from this approximation of the likelihood distribution $\mathbf{X}|\mathbf{z}$ will yield something close to $\mathbf{x}$ itself.

While the idea itself is clear, the difficulty in having it come to fruition is training such an architecture, i.e. finding the parameters ϕ and θ that offer the best reconstruction and smooth latent space, all according to some metric. In line with most other approaches to generative modelling, if we can do maximum likelihood in some way, i.e. ensure that the likelihood of the training data is maximal under our model then samples from the likelihood

distribution, whose parameters are output by the model's decoder, should resemble those in the training set. To do this, we would ideally perform maximum likelihood directly but we don't have access to the true underlying joint distribution function $p(\mathbf{x})$. As such, we instead look to find a lower bound for $p(\mathbf{x})$ which we maximise instead. The idea is that pushing this supposed lower bound up will in turn push up $p(\mathbf{x})$ itself. In line with this, we derive the evidence lower bound (ELBO), first noting that

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z} \approx \int p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}$$

where the parameters of $p_\theta(\mathbf{x}|\mathbf{z})$ are given by VAE's decoder. Note that an almost identical method can be used for discrete random variables, integrals just look prettier. From here, we can utilise Jensen's inequality to obtain

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

where the parameters of $q_\phi(\mathbf{z}|\mathbf{x})$ are given by the VAE's encoder. As such, for a training set $\{\mathbf{x}_i\}_{i=1}^N$, we look to compute

$$\arg\max_{\phi,\theta} \sum_{i=1}^{N} \left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}_i)}\left[\log(p_\theta(\mathbf{x}_i|\mathbf{Z}))\right] - D_{\mathrm{KL}}(q_\phi(\mathbf{Z}|\mathbf{x}_i)||p(\mathbf{Z}))\right].$$

If we can maximise the ELBO then we are doing some sort of pseudo-maximum likelihood but what do the individual terms mean? The first term, $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log(p_\theta(\mathbf{x}|\mathbf{Z}))\right]$ pertains to how well the model reconstructs an input $\mathbf{x}$ given its latent representation. The motive of maximising this is self-explanatory. The second term, however, isn't entirely clear at first

and motivations for minimising it are usually pretty bad. Minimising the KL-divergence term $-D_{\mathrm{KL}}(q_\phi(\mathbf{Z}|\mathbf{x})||p(\mathbf{Z}))$ pertains to encouraging our approximation of the posterior to be close to our chosen prior (usually just $\mathcal{N}(0, I)$). There is good reason to desire this as any usual choice of prior is nicely structured. It ensures that sampling from the prior distribution is sufficiently similar to sampling from the posterior itself. Note that more often than not, even though this ensures that the posterior is well-structured, this does not mean that it looks human-interpretable[4].

It turns out that the ELBO is a very tight bound in many cases - even to the extent of performing very well on density estimation benchmarks.

5.1.2 The Reparameterization Trick

Back propping through sampling is not a thing, it ain't differentiable

5.1.3 VAE architecture for MNIST

For generating new sample of MNIST, the following architecture worked for me. Encoder:

- three hidden layers of $h/4$, $h/2$ and h neurons respectively where $h = 4096$ is the hidden dimension
- ReLU for each hidden layer and just linear maps for parameters
- a latent dimension of 256

5.2 Generative Adversarial Networks (GANs)

Compete!

5.3 Flow-Based Models

Turn noise to samples!

5.4 Diffusion

Turn noise to samples!

5.5 Transformers

Stand at attention!

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

5.5.1 Token embedding

Embed the semantics of tokens in high-dimensional space. “Germany” plus “fascist” minus “Mussolini” equals “Hitler”.

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.

Q1: Why choose normal priors in VAEs?

. . .

Appendices

A Probability Theory Things

Discrete:

- Poisson, Geometric → Negative Binomial **(generic)**
- Bernoulli → Binomial **(binary classification)**
- Categorical → Multinomial **(multi-class classification)**

Continuous:

- Uniform, (Multivariate) Normal, Exponential **(generic)**
- (Multivariate) Student's t, Chi-squared **(classical stats)**
- Beta, Gamma, Dirichlet **(Bayesian stats)**

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 $\Omega_{\mathbf{X}} = \{0, \dots, n\}$. Let k denote a realisation of $\mathbf{X}$, i.e. $k = x_1 + \cdots + x_n$. The distribution function of $\mathbf{X}$ is given by

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

where Z is a normalising constant accounting for the number of ways one can permute the tuple of realisations $(x_1, \dots, x_n)$. 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. As such, 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 c^{th} 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 the second class is present. The distribution function of X 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 distribution function can also be written using indicator functions but this form 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 realisations of a categorical distribution with C classes. 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 distribution function of $\mathbf{X} = \sum_{i=1}^{n} X_i$ with $X_1, \ldots, X_n \sim \text{Cat}(p_1, \ldots, p_C)$? Let x_{ij} denote the element in the i^{th} row and j^{th} column of $\mathbf{x}$ and let $k_j = x_{1j} + \cdots + x_{nj}$ denote the sum of all elements in the j^{th} column of $\mathbf{x}$. Note that k_j is the number of realisations

of $X_1, \ldots, X_n$ in which the j^{th} class is present. We have

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

So all that's left to do is derive this 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 $\text{Ber}(p)$ distribured. So if $X \sim$

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 Central Limit Theorem (CLT)

...

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: 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.

A.5.1 What does the entropy of a random variable tell us?

How surprising it is.

B Statistics Things

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.

B.2 Assumptions of Normality

https://stats.stackexchange.com/a/12266