Machine Learning for High Energy Physics

Notes taken by Manuel Morales & James Moore
at the MLHEP summer school

1 Competition: Dark matter searches **5**
1.1 Introduction to dark matter . . . 5
Background information on dark matter.
1.2 Direct detection of dark matter . . . 5
Dark matter flux on Earth. Directionality and detection. Background events vs signal events. Electron recoil and nuclear recoil.
1.3 Dark matter detection with a time projection chamber . . . 7
Gas tank detection. Images of electron recoil tracks and nuclear recoil tracks. Goal of the school's competition.

I Introduction to machine learning **9**

2 Introduction to supervised learning **9**
2.1 Problem setup . . . 9
Problem setup for supervised learning. Basic examples.
2.2 Feature types . . . 10
Features and the design matrix. Numeric, categorical nominal, categorical ordinal and binary features. One-hot encoding.
2.3 Learning algorithms . . . 12
Definition of a learning algorithm. Loss functions, e.g. least squares error. Examples of k nearest neighbours and linear regression.
2.4 Assumptions about data and model complexity . . . 14
Training and test data. The curse of dimensionality. Model complexity.

3 Linear models **16**
3.1 Linear models and linear regression . . . 16
Linear models and examples. 'Hidden power' of linear models via feature transformation. Examples of loss functions: MSE, MAE, MAPE and MSLE.
3.2 Analytic solution . . . 18
Analytic solution for linear models. Ill-conditioning and flat directions. Bias terms.
3.3 Numerical and stochastic optimisation: gradient descent . . . 20
The gradient and its geometric interpretation. The gradient descent algorithm; convex functions. Overfitting and early stopping. Stochastic gradient descent.
3.4 Feature expansion . . . 22
General description of feature transformation. Examples of polynomial feature transformations.

4 Classification with linear models **23**
4.1 Classification with linear regression . . . 23
Classification problems; naïve approach with MSE loss. Better approach with smooth approximations to 0-1 loss.
4.2 Logistic regression . . . 25
Logistic regression from class probabilities and maximum log likelihood. The sigmoid function and the logistic regression loss function.
4.3 Multinomial logistic regression . . . 27

Logistic regression for many classes using class probabilities; symmetry of the parameters. The multinomial logistic regression loss function.

4.4 Multiclass classification: general approach . . . 28
One-versus-rest multiclass classification. Comparison with logistic regression.

5 Model regularisation **30**
5.1 The problem of overfitting . . . 30
Recap of overfitting and test set.
5.2 Prediction error decomposition . . . 30
Decomposition of error sources: model variance, bias, and irreducible error. The bias-variance tradeoff. Bias and variance of linear models.
5.3 Regularisation . . . 32
Regularisation methods: $L2$ regularisation, $L1$ regularisation, and elastic net regression. Comparison of methods; sparsity vs weight-sharing.
5.4 Probabilistic view . . . 34
Relationship of regularisation to probabilities and maximum likelihood.
5.5 Bayesian view . . . 35
Relationship of regularisation to Bayesian priors.

6 Quality metrics for regression and classification **36**
6.1 Quality metrics for regression models . . . 36
Effect of outliers.
6.2 Quality metrics for classification models . . . 38

7 Model tests **44**
7.1 Train/test split . . . 44
7.2 Cross-validation methods . . . 46
7.3 Quality metric uncertainty estimation . . . 49
7.4 Statistical model comparison . . . 50

8 Decision trees **53**
8.1 Decision tree training . . . 53
8.2 Stopping rules for decision tree learning . . . 55

9 Ensemble methods **56**
9.1 Bagging and random forests . . . 56
9.2 Stacked generalisation . . . 58

10 Gradient boosting **60**
10.1 Boosting . . . 60
10.2 Gradient boosting . . . 62
10.3 Regularisation of gradient boosting machines via shrinkage . . . 64
10.4 XGBoost algorithm . . . 66

II Introduction to neural networks **67**

11 Feature engineering, importance and selection **67**
11.1 Feature engineering . . . 67
11.2 Feature importance . . . 69
11.3 Feature selection . . . 73

12 Clustering **74**
12.1 Clustering vs classification . . . 74
12.2 K-means . . . 75
12.3 Quality metrics . . . 78
12.4 Limitations of the K-means algorithm . . . 80
12.5 Hierarchical clustering . . . 81
12.6 DBSCAN . . . 82

13 Introduction to neural networks **83**
13.1 From linear models to neural networks . . . 83
13.2 Activation functions . . . 85
13.3 Universal approximators and deeper nets . . . 86
13.4 Backpropagation . . . 86
13.5 Optimisation techniques . . . 87

14 Introduction to PyTorch **89**
14.1 Deep learning frameworks . . . 89
14.2 Neural network representation . . . 90
14.3 Building blocks, graph . . . 91
14.4 Inference . . . 94
14.5 Neural network class . . . 94
14.6 PyTorch Lightning . . . 94

15 Network regularisation **95**
15.1 Weight initialisation . . . 95
15.2 Overfitting with neural networks . . . 96
15.3 Normalisation layers . . . 97

16 Convolutional neural networks **99**
16.1 Image recognition . . . 99
16.2 Convolutional layer . . . 101
16.3 Pooling . . . 102
16.4 Convolutional features stacking . . . 104

17 Transformers **105**
17.1 Introduction to sequence to sequence models . . . 105
17.2 Transformers . . . 106
17.3 Inside the encoder block . . . 107

III Bayesian deep learning **108**

18 Introduction to Bayesian methods **108**
18.1 Frequentist vs Bayesian frameworks . . . 108
18.2 Application of the Bayesian framework to machine learning models . . . 111
18.3 Full Bayesian inference . . . 112
18.4 Maximum a posteriori estimation . . . 114
18.5 Approximate Bayesian inference . . . 115

19 Bayesian linear regression **118**
19.1 Review of linear regression . . . 118
19.2 Bayesian approach to linear regression . . . 118

20 Introduction to black-box optimisation **123**
20.1 Optimisation methods categorisation 123
20.2 Simple black-box optimisation methods 124

21 Variational optimisation **126**
21.1 Variational bound 126
21.2 Stochastic gradient descent with variational optimisation 128
21.3 Discrete variables 128

1 Competition: Dark matter searches

1.1 Introduction to dark matter

Less than 5% of the total mass and energy in the universe is the stuff we know about: stars, planets, galaxies, gases. Most of the universe is made of *dark matter* (dark because it does not interact with light), whilst the rest is *dark energy*.

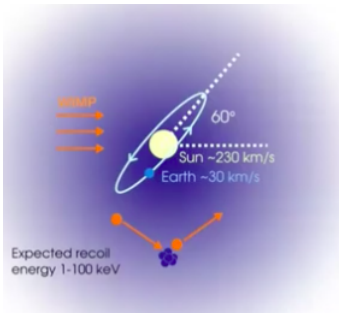
There is significant evidence for this dark matter (from rotational curves of galaxies, the CMB, velocity of galaxies, big bang nucleosynthesis, cluster collisions) at various scales. It was even hypothesised in the 1930s - but we still don't know what it is! We can, however, comment on *where* dark matter is. Our galaxy is surrounded by an approximately spherical halo of dark halo, with the Sun moving through this halo.

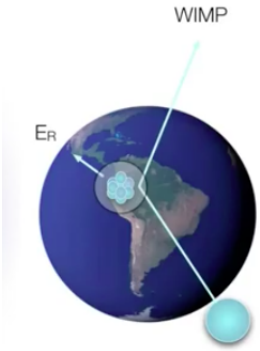
Certainly, dark matter is not one of the Standard Model elementary particles (i.e. dark matter is not a quark, lepton, gauge boson, or the Higgs boson). Quarks, leptons and the W-boson are all charged (dark matter is neutral - it is 'dark'), neutrinos are too light (and move too fast), and the Z and Higgs are too short-lived.

There are many models which have been proposed to explain dark matter. One hypothesis is that dark matter is made of *weakly interacting massive particles* (abbreviated to WIMPs).

1.2 Direct detection of dark matter

Since the Earth is moving through the dark matter halo surrounding the galaxy, we should expect to able to detect dark matter here on Earth.





One can perform a rough calculation to determine how many dark matter particles we expect to detect:

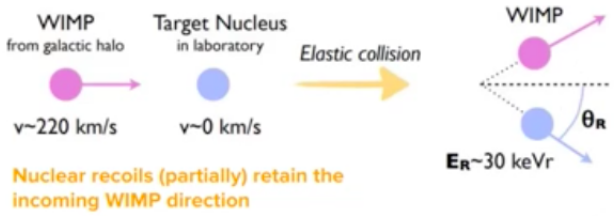
- The dark matter density is given experimentally by $\rho = 0.3\,\mathrm{GeV/cm^3}$
- The average velocity of the Sun is given by $\langle \mathbf{v} \rangle = 220\,\mathrm{km/s}$.
- We don't know the mass of the dark matter particles, but in the WIMP models we expect that the mass is around $m \approx 100\,\mathrm{GeV}$.

We can then determine the *flux* of WIMPs on Earth:

$$\phi = \frac{\rho}{m} \langle \mathbf{v} \rangle \approx 10^5\,\mathrm{cm^{-2}s^{-1}}.$$

Thus tens of millions of dark matter particles are flying through your hand every second!

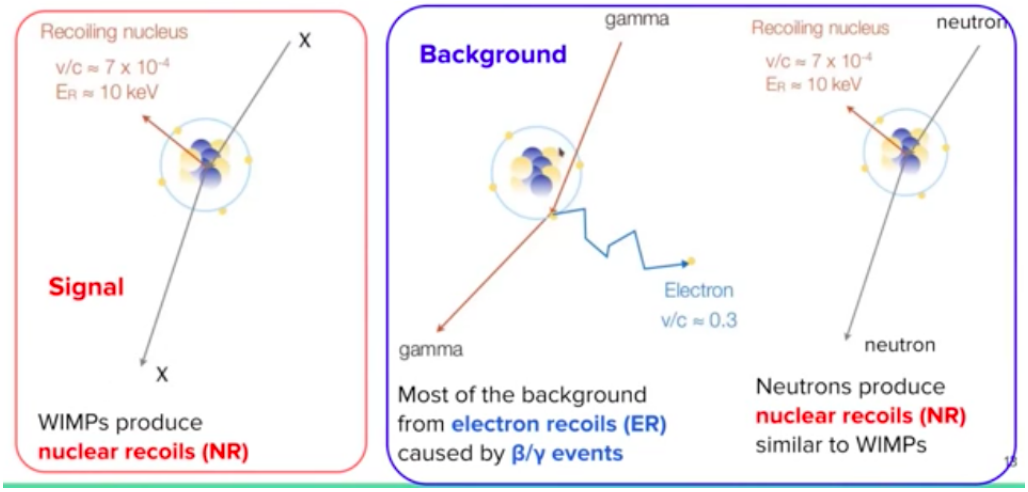
One important strategy we can employ in the detection of dark matter is *directionality*. From our position on Earth, we know that dark matter particles arrive from a *precise direction* in the sky, namely the *Cygnus constellation*. They travel at velocities around $220\,\mathrm{km/s}$. When they hit some special nuclei that forms part of our detector, there is an *elastic collision* that results in movement of the nuclei - this is measurable because the nuclei is visible (unlike the WIMP).



If we see nuclei moving in a direction that is compatible with the direction that we expect dark matter to arrive from, this could be evidence of dark matter detection.

The direction is very important because there are many other things that can hit a nucleus, and the nucleus can move for other reasons unrelated to dark matter. The general problem is that *whilst the flux of WIMPs is high, the probability of interaction is very low*. In a sensitive detector we expect 1 event per kilogram per year (compare this with placing a banana near the detector - this causes around 100 events per kilogram per year, because of the radiation from the potassium in the banana!). Therefore, the experimental challenge is to detect a *tiny signal* over a *large background*.

There are some differences between signal and background events. A dark matter signal comprises only *nuclear recoil*, whilst most of the background is comprised of *electron recoil* caused by photon interactions. On the other hand the background does also include nuclear recoil events caused by neutrons, which are similar to WIMP nuclear recoil events.



There are two main sources of background:

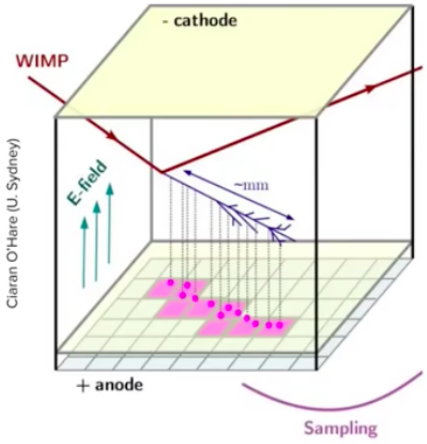
(1) *Cosmic rays*. These are a problem if you would like to do your measurement on the Earth's surface. To mitigate the effect, we can work in underground laboratories.

(2) *Environmental radioactivity* (mainly from potassium, uranium and thorium). This comes even from the walls of your lab! To mitigate this effect, we can shield the equipment, and carefully choose materials with low levels of radioactivity in the setup.

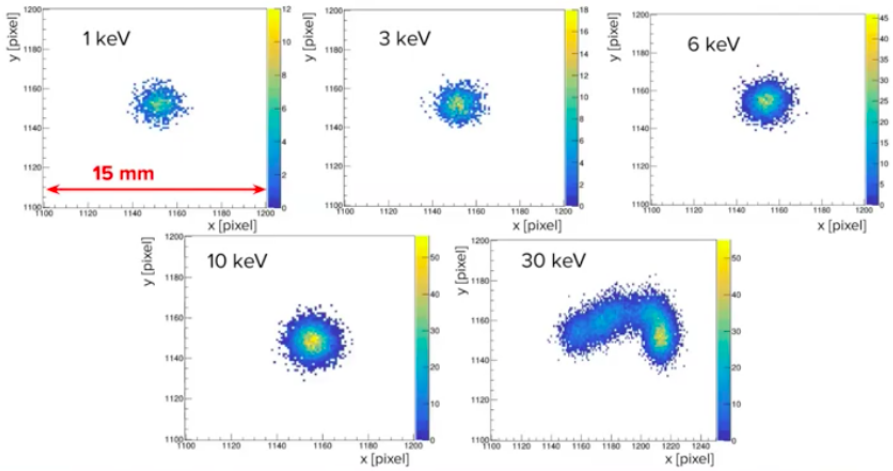
1.3 Dark matter detection with a time projection chamber

We will now describe in more detail the detection of dark matter via nuclear recoil.

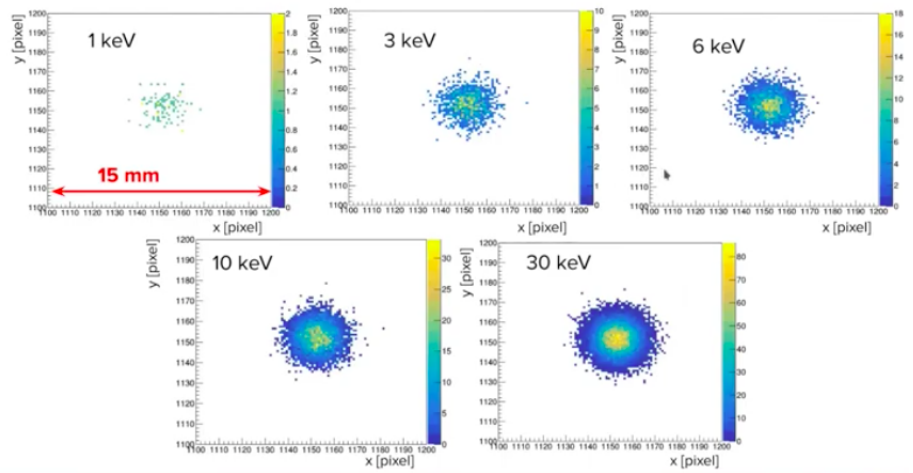
The general idea is to us a *gas target*; for example the gas used in the CYGNO experiment, based in Italy, is a mixture of helium He and carbon tetrafluoride CF_4. The advantage of using a gas target is that it is not very dense, so the recoiling nucleus can travel several millimetres. This direction can be measured.



The way the measurement works is as follows. Electrons from the struck nucleus are ionised, and an electric field is applied to the detector such that the electrons move towards the *anode* of the detector. The amount of electrons that reach the anode is proportional to the released energy, which implies we can measure energy deposits and their position in the xy-plane, giving a 2D projection of the particles' track.
Here are some examples of (simulated, cleaned) tracks from electron recoils in the CYGNO experiment:

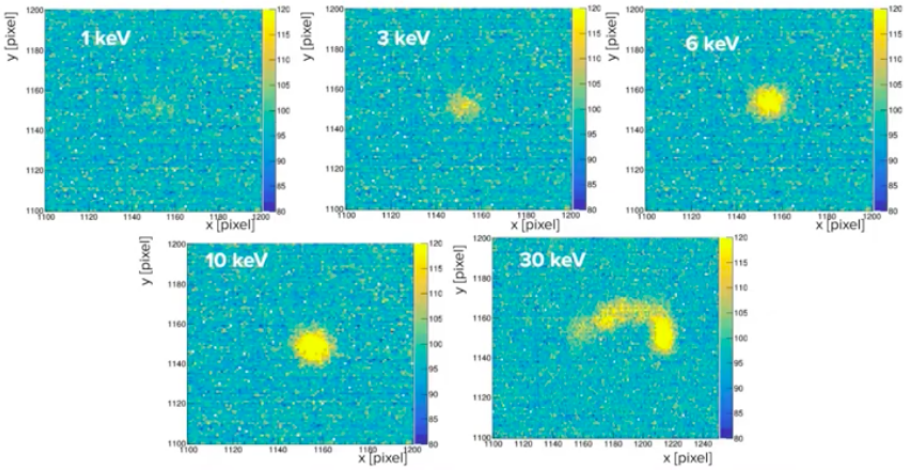


Here are some examples of (simulated, cleaned) tracks from helium recoils in the CYGNO experiment:



As you can see from the figures, there is a somewhat visible difference between electron recoil and helium recoil.

When noise is added, the figures become slightly more complicated:



For smaller energies it can be difficult by eye to even find the track!

The goal of the competition at the school is to devise a machine learning procedure to distinguish between electron recoil (ER) and helium nuclear recoil (NR) in such tracks. The events will have energies ranging from 1 to 30 keV (the most important energy range for dark matter searches, and also the most challenging region for background rejection), together with a random initial angle. There will be precisely one particle for each image in each case.

Part I
Introduction to machine learning

2 Introduction to supervised learning

2.1 Problem setup

Definition 2.1: Consider a set $\mathcal{X}$ of objects and a set of targets $\mathcal{Y}$. Suppose also that there is an unknown function $f : \mathcal{X} \to \mathcal{Y}$ mapping objects into targets (which may be stochastic - by which we mean the objects and targets are drawn from some joint probability distribution).

A *dataset* is a finite collection $D = \{(x_i, f(x_i)) : i = 1, ..., N\} \subseteq \mathcal{X} \times \mathcal{Y}$. The goal of *supervised learning* is to *approximate f given a dataset D* (i.e. learn to recover targets from objects).

Example 2.2:

(i) **Classification of iris flower species.** The objects $\mathcal{X}$ are individual flowers, described by the length and width of their sepals and petals. The targets $\mathcal{Y}$ are the species to which the flowers belong. The mapping $f : \mathcal{X} \to \mathcal{Y}$ takes a flower to its species (and depends on different shapes of sepals, petals, etc corresponding to the different species). Note this is a *non-deterministic* mapping because the growth of a certain flower is associated with a number of random processes.

(ii) **Spam filtering.** The objects $\mathcal{X}$ are emails (sequences of characters). The targets are the categories $\mathcal{Y} = \{\text{spam, not spam}\}$. The mapping $f : \mathcal{X} \to \mathcal{Y}$ tells us whether a particular email $x \in \mathcal{X}$ is spam or not, and is again *non-deterministic* (the style and content can vary from author to author).

(iii) **CAPTCHA recognition.** The set of objects $\mathcal{X}$ is a set of CAPTCHA images (which are really vectors of pixel level brightness). The set of targets $\mathcal{Y}$ are sequences of characters, relating to the correct word that the CAPTCHA shows. The mapping $f : \mathcal{X} \to \mathcal{Y}$ is the inverse of the algorithm used to generate the CAPTCHA - this is *almost* deterministic, but depends on the level of distortion (very high distortion can make the map less deterministic).

(iv) **Particle identification in a HEP experiment.** The objects $\mathcal{X}$ are particles, described by the detector experiments (e.g. track parameters, calorimeter energy, deposit, etc.). The targets $\mathcal{Y}$ are the types of the particles (e.g. electrons, muons, protons, etc.). The mapping $f : \mathcal{X} \to \mathcal{Y}$ is the inverse of the physical process generating the detector response (by definition, this is a stochastic process).

2.2 Feature types

Typically the objects in our set $\mathcal{X}$ will have some structure.

Definition 2.3: Let $\mathcal{X}, \mathcal{Y}$ be the sets of objects and targets in a supervised learning problem. Suppose that $\mathcal{X}$ is a collection of *tuples*, so that given a dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$, for each $(x_i, f(x_i)) \in D$ we can write x_i as:

$$x_i = (x_i^1, x_i^2, ..., x_i^d),$$

for some d (which need not be the same for all $x_i \in \mathcal{X}$). We say that x_i^j is a *feature* of the object x_i (note we use the upper index for the *feature*, and the lower index to label the object in the dataset).

Many algorithms require that the *dimensionality* d of the data is the same for all objects. In this case, the dataset can be represented by a *design matrix*:

$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix}.$$

The column index corresponds to the features, and the row index corresponds to the position of the object in the dataset.

Example 2.4: Consider the problem of classifying species of irises. In this case, features might include *sepal length*, *sepal width*, *petal length* and *petal width*. In this case, all of the features are real numbers.

Since each flower will have these features, we can describe all objects as 4-tuples, and hence we can organise the data into a $N \times 4$ design matrix.

Definition 2.5: In general, features x_i^j might be of varying natures. We give special names to some common cases:

- *Numeric features* take values in (for example) the real numbers. For example sepal length in the classification of iris species, building height, or particle transverse mometnum.
- *Categorical nominal features* take values in a finite set, with no natural ordering. For example colour, city of birth or particle type.
- *Categorical ordinal features* take values in a set equipped with a natural order, for example level of education, age, or particles passing loose, medium or tight selection criteria.
- *Binary features* take values in a set of size 2, for example $\{\text{true}, \text{false}\}$, $\{0, 1\}$, $\{-1, +1\}$.

A natural question that arises in the handling of data is how one might convert a categorical nominal feature into a binary or numeric feature (since binary or numeric features can be more easily handled).

A naïve approach would be to assign each category a number (e.g. red$= 1$, green$= 2$, etc). However, this introduces an ordering on the feature, which could have a negative effect on learning algorithms later on.

A better solution is *one-hot encoding*:

Definition 2.6: Let $x_i = (x_i^1, ..., x_i^{j-1}, x_i^j, x_i^{j+1}, ..., x_i^d)$ be an object in a dataset, and let x_i^j be a categorical nominal feature for the object, taking values in some finite set S. A *one-hot encoding* replaces the feature x_i^j with $|S|$ binary features $\{x_i^{j,s} : s \in S\}$ obeying:

$$x_i^{j,s} = \begin{cases} 0 & \text{if } x_i^j \neq s, \\ 1 & \text{if } x_i^j = s. \end{cases}$$

The new object then takes the form

$$x_i = (x_i^1, ..., x_i^{j-1}, x_i^{j,s_1}, ..., x_i^{j,s_{|S|}}, x^{j+1}, ..., x_i^d),$$

where $s_i \in S$ are the elements of S.

Example 2.7: Suppose x is a categorical nominal feature taking values in the set $\{\text{red, blue, green}\}$. A one-hot encoding of the object is:

$$(x^{\text{red}}, x^{\text{blue}}, x^{\text{green}}),$$

where $x^{\text{red}} = 0$ if $x \neq$ red, 1 if $x =$ red, etc.

For example, $x =$ red is replaced by $(1, 0, 0)$, whilst $x =$ green is replaced by $(0, 0, 1)$.

One-hot encoding can also be applied to categorial ordinal features:

Definition 2.8: Let $x_i = (x_i^1, ..., x_i^{j-1}, x_i^j, x_i^{j+1}, ..., x_i^d)$ be an object in a dataset, and let x_i^j be a categorical *ordinal* feature for the object, taking values in some finite set S with ordering $<$. Suppose that the elements of S are ordered as $s_1 < s_2 < \cdots < s_{|S|}$. A *one-hot encoding* replaces the feature x_i^j with $|S|$ binary features $\{x_i^{j,s} : s \in S\}$ obeying:

$$x_i^{j,s} = \begin{cases} 0 & \text{if } s < x_i^j, \\ 1 & \text{if } x_i^j \leq s. \end{cases}$$

The new object then takes the form

$$x_i = (x_i^1, ..., x_i^{j-1}, x_i^{j,s_1}, ..., x_i^{j,s_{|S|}}, x^{j+1}, ..., x_i^d).$$

Example 2.9: Suppose x is a categorical ordinal feature taking values in the ordered set $\{\text{bachelors, masters, PhD}\}$, with ordering bachelors $<$ masters $<$ PhD. A one-hot encoding of the object is:

$$(x^{\text{bachelors}}, x^{\text{masters}}, x^{\text{PhD}}).$$

For example, $x =$ bachelors is replaced by $(1, 0, 0)$, whilst $x =$ PhD is replaced by $(1, 1, 1)$.

Note that this *retains* the ordering of the original categorical ordinal feature.

2.3 Learning algorithms

Definition 2.10: Let $\mathcal{X}, \mathcal{Y}, f$ be the respective objects, targets and function of a supervised learning problem. Given a dataset $D = \{(x_i, f(x_i)) : i = 1, 2, ..., N\} \subseteq \mathcal{X} \times \mathcal{Y}$, a *learning algorithm* $\mathcal{A}$ returns an approximation $\hat{f} = \mathcal{A}(D)$ (which depends on the dataset) to the true function f.

Definition 2.11: The k *nearest neighbours algorithm* can be applied when the features of the objects have some notion of distance (i.e. a *metric*). We then define:

$$\hat{f}(x) = \frac{1}{k} \sum_{i : x_i \in D_x^k} f(x_i),$$

where D_x^k is the set of k objects in the dataset D closest to x.

Example 2.12: Consider a *classification problem* where the target space $\mathcal{Y}$ consists of only two values. In this case the k nearest neighbours algorithm approximates the function $f : \mathcal{X} \to \mathcal{Y}$ by:

$$\hat{f}(x) = \operatorname*{argmax}_C \sum_{i : x_i \in D_x^k} 1_{\{f(x_i) = C\}},$$

where $1_{\{f(x_i)=C\}}$ is the relevant indicator function. Note that instead of averaging the target, we take the value of C which maximises the sum, either 0 or 1.

In general, how does one find an algorithm $\mathcal{A}$ giving an approximation $\hat{f} = \mathcal{A}(D)$ to the true mapping function? Many algorithms work by solving an *optimisation task* - namely, minimising a *loss function*:

Definition 2.13: Let $\hat{f} = \mathcal{A}(D)$ be the approximation to the true mapping function for a supervised learning problem $f : \mathcal{X} \to \mathcal{Y}$, with dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$ given. A *loss function* is a function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, chosen to measure the quality of predictions. The *loss* of our approximation at the point $x_i \in \mathcal{X}$ in the dataset is given by $\mathcal{L}(f(x_i), \hat{f}(x_i))$.

Example 2.14: An example of a loss function is *least squares error*, given by $\mathcal{L}(f(x_i), \hat{f}(x_i)) = (f(x_i) - \hat{f}(x_i))^2$, where the target space $\mathcal{Y} = \mathbb{R}$.

Definition 2.15: Given a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ for a supervised learning problem, we can obtain an approximation $\hat{f} = \mathcal{A}(D)$ of the true mapping $f : \mathcal{X} \to \mathcal{Y}$ from a datset D via *loss minimisation*:

$$\hat{f} = \operatorname*{argmin}_{\tilde{f}} \mathop{\mathbb{E}}_{(x_i, f(x_i)) \in D} \mathcal{L}(f(x_i), \tilde{f}(x_i)),$$

where the argmin is taken over all functions $\tilde{f} : \mathcal{X} \to \mathcal{Y}$, and the notation $\mathbb{E}$ means the expected value of the loss over the dataset (in the case that the mapping is stochastic, this is important).

Example 2.16: An example is *linear regression*, where we take $\mathcal{X} = \mathbb{R}^d$, $\mathcal{Y} = \mathbb{R}$. In this case, the functions $\tilde{f}$ which could constitute possible mappings $\mathcal{X} \to \mathcal{Y}$ are restricted to simply be *linear functions*, of the form:

$$\tilde{f}_{\mathbf{w},\mathbf{b}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \mathbf{b}.$$

Given a dataset $D \subseteq \mathcal{X} \times \mathcal{Y}$ of size N, minimising the least squared loss corresponds to determining $\mathbf{w}$, $\mathbf{b}$ such that:

$$\frac{1}{N} \sum_{i=1,\ldots,N} \left(f(x_i) - \hat{f}_{\mathbf{w},\mathbf{b}}(x_i) \right)^2$$

is minimised.

2.4 Assumptions about data and model complexity

Given no assumptions about a dataset, there are typically infinitely many solutions to a loss minimisation problem. For example, if we wish to interpolate a finite set of points for a one-dimensional function, infinitely many curves will perfectly minimise the loss (defined in this case as the sum of the distances between the curve and the point at each point).

To combat these, we introduce the concept of *test data*:

Definition 2.17: A set of *test data* is an independent dataset drawn from the same population as the dataset used to construct the solution of our loss minimisation problem (this original set is called the *training data*).

We then demand that our expected loss on the whole population (i.e. both the test and training data) is minimised.

Example 2.18: Consider a linear interpolation problem in $\mathbb{R}^d$ which we solve both with the k nearest neighbour algorithm, and with linear regression. Suppose the true dependence is also linear.

It turns out that if we validate our approaches using a test set, the error on the test set for k nearest neighbour and linear regression agree well for lower dimensions d, but k nearest neighbour becomes much worse for larger numbers of dimensions.

This feature is called the *curse of dimensionality* - as the number of dimensions of the feature space grows, the data becomes very sparse, and the k nearest neighbour approach begins to fail. In this case, to keep the error low, the number of training points would have to increase exponentially with the dimension.

The reason we get the distinction is that the two algorithms have different assumptions:

- The k nearest neighbour (kNN) approach assumes that *similar objects have similar targets*.
- The linear regression approach assumes that *targets are linear in features*.

In this case, both assumptions are correct, but linearity is much stronger which allows us to fight the curse of dimensionality.

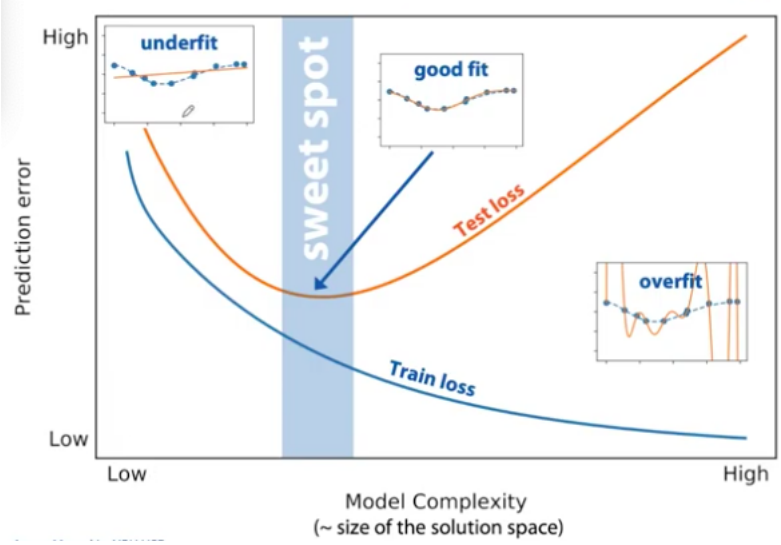
In particular, imposing assumptions about the data *restricts the possible function space of solutions* $\tilde{f}$ for the loss minimisation problem:

$$\hat{f} = \operatorname*{argmin}_{\tilde{f}} \mathop{\mathbb{E}}_{(x_i, f(x_i)) \in D} \mathcal{L}(f(x_i), \tilde{f}(x_i)),$$

Linearity is a much stronger constraint than kNN imposes (kNN allows for a much more flexible functional form), restricting the space of functions much more. Sometimes these restrictions help us to *overcome the curse of dimensionality*.

Definition 2.19: The *complexity* of a model refers to the heuristic size of the function space over which we minimise in the loss minimisation problem. A larger complexity means a larger function space, whilst a smaller complexity means a smaller function space.

In general, changing the model complexity can allow us to find the 'sweet spot' where expected loss on both the training and test data sets is minimised:



The goal is to find the *right level of limitations* - not too strict, not too loose.

3 Linear models

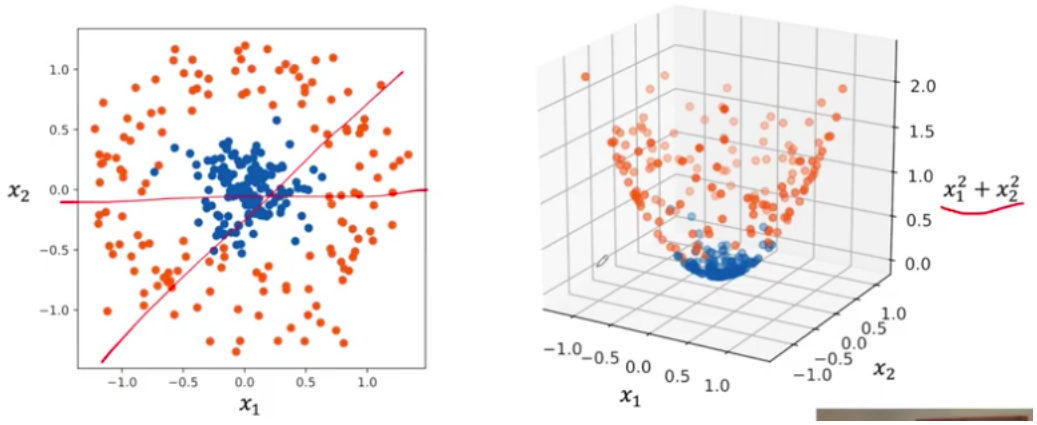
3.1 Linear models and linear regression

Definition 3.1: A *linear model* is a model $\hat{f}(x)$ which is linear in its input (note this assumes a vector space structure on both the objects and targets).

Example 3.2: Examples include:

- Linear regression, for example $\hat{f}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$.
- Classification, for example $\hat{f}(\mathbf{x}) = 1_{\{\boldsymbol{\theta}^T \mathbf{x} > 0\}}$. This is non-linear as a function, but divides the target space by a line.

Linear models have a *hidden power*. Problems which appear linearly inseparable (i.e. cannot be solved with linear models) can be made into problems which are tractable with linear models by *transforming the features*. As an example, the data on the left below can be made linearly separable by introducing a new feature based on the sum of the squares of the coordinates of the points:



Now a hyperplane separates the blue and orange sets.

In general, linear models motivate the construction of *deep models*. Neural networks are just linear models with *activation functions* between them (effecting some transformation). Thus better understanding linear models will help us understand deep neural networks.

We have already introduced linear regression as a linear model. Here, we update the notation:

Definition 3.3: The *model prediction* is $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$, where $\boldsymbol{\theta} \in \mathbb{R}^d$ is our *parameter vector*, $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ is our *features vector*, and:

$$\frac{1}{N} \sum_{i=1,\ldots,N} \left(f(\mathbf{x}_i) - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i) \right)^2$$

is our mean squared error loss, given a dataset. We wish to minimise this over $\boldsymbol{\theta}$.

Other loss functions are available. For example:

- **Mean absolute error (MAE).** This is given by:

$$\frac{1}{N}\sum_{i=1,\ldots,N}\left|f(\mathbf{x}_i)-\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right|.$$

  This grows linearly for large errors, rather than quadratically like mean squared error loss. In particular, it doesn't penalise large errors as much as mean squared error loss - it might be more suitable if there are outliers in your data which you do not wish to be biased against.

- **Mean absolute percentage error (MAPE).** This is given by:

$$\frac{1}{N}\sum_{i=1,..,N}\left|\frac{f(\mathbf{x}_i)-\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)}{f(\mathbf{x}_i)}\right|.$$

- **Mean squared logarithmic error (MSLE).** This is given by:

$$\frac{1}{N}\sum_{i=1,\ldots,N}\Big(\log(f(\mathbf{x}_i)+1)-\log(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)+1)\Big)^2.$$

  The 1's are there just to allow zero values of the targets.

Both MAPE and MSLE provide a means of computing *relative error*, so are useful when the targets span through a large range of magnitudes.

In general, applying different loss functions are also related to different assumptions about the data. A different loss function might be more suitable to a particular dataset.

3.2 Analytic solution

Recall that earlier we defined the design matrix X, a matrix with rows given by the objects and columns given by the features. We can use this to rewrite the mean squared error loss as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1,\dots,N} \left(f(\mathbf{x}_i) - \boldsymbol{\theta}^T \mathbf{x}_i \right)^2 = \frac{1}{N} ||\mathbf{y} - X\boldsymbol{\theta}||^2,$$

where $\mathbf{y} = (f(\mathbf{x}_1), ..., f(\mathbf{x}_N))^T$.

To minimise this function, we can ignore the $1/N$ as it plays no role. Thus we wish to minimise $||\mathbf{y} - X\boldsymbol{\theta}||^2$ over $\boldsymbol{\theta}$. Analytically, the requirements are:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}_{\text{MSE}} = \mathbf{0}, \qquad \text{the matrix } \frac{\partial^2}{\partial \theta_i \partial \theta_j} \mathcal{L}_{\text{MSE}} \text{ of second derivatives should be positive definite.}$$

The first condition ensures we have an extremum, the second condition ensures that extremum is a minimum.

Proposition 3.4: If the columns of X are linearly independent, the solution to the above minimisation problem is $\boldsymbol{\theta} = (X^T X)^{-1} \mathbf{y}$.

Proof: Note that in general, the matrix $X^T X$ is always *positive semi-definite*, since:

$$\mathbf{v}^T X^T X \mathbf{v} = ||X\mathbf{v}||^2 \geq 0$$

for arbitrary $\mathbf{v}$. We see it is *precisely* positive definite when the columns of X are linearly independent (imagine the expression $X\mathbf{v}$ to be a linear combination of the columns of X). This additionally implies X is invertible (e.g. all its eigenvalues are positive).

We can now consider the optimisation problem. The first derivative of $\mathcal{L}_{\text{MSE}}$ is easily computed:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}_{\text{MSE}} = \frac{1}{N} \frac{\partial}{\partial \boldsymbol{\theta}} (\mathbf{y} - X\boldsymbol{\theta})^T (\mathbf{y} - X\boldsymbol{\theta}) = -\frac{2}{N} X^T (\mathbf{y} - X\boldsymbol{\theta}) = \mathbf{0}.$$

This implies that $X^T \mathbf{y} - X^T X \boldsymbol{\theta} = \mathbf{0}$. Since $X^T X$ is invertible, we can compute the solution:

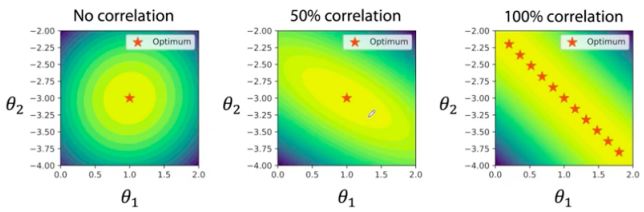
$$\boldsymbol{\theta} = (X^T X)^{-1} X^T \mathbf{y}.$$

To check that this is a minimum, we must look at the second derivative. We find that the matrix of second derivatives is precisely:

$$2X^T X.$$

In particular, this matrix is positive definite by the above. □

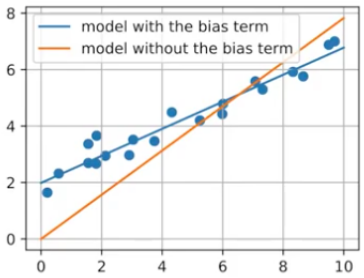
In practice, there may be some linear dependence of the columns of X (i.e. there may be some 'feature correlations'). This affects the invertibility of the matrix $X^T X$ - we get *flat directions*, and hence multiple solutions to the problem.



In the above we also assumed that our model took the form $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$ for ease of calculation, but actually we can consider the more general linear model $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} + \theta_0$, where the term θ_0 is called the *bias term*. There is no need to redo any calculations here; we simply add a constant feature to the design matrix to account for this:

$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix} \quad \rightarrow \quad X = \begin{pmatrix} 1 & x_1^1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix}$$

Now when we multiply $X\boldsymbol{\theta}$, one of the parameters will play the role of the bias term.



3.3 Numerical and stochastic optimisation: gradient descent

We have now discussed the analytic solution for the mean squared error loss minimisation problem in the context of a linear model. However, most loss functions do not allow an analytic solution, so it is important to discuss numerical solutions.

We being with some revision of the *gradient*. Recall that the gradient of a scalar function on $\mathbb{R}^d$ is given by:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), ..., \frac{\partial f}{\partial x_d}(\mathbf{x}) \right).$$

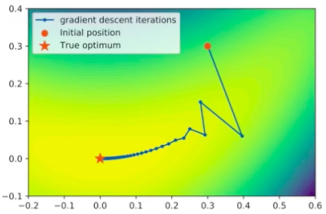
The gradient always points in the *direction of greatest increase* of the function. Hence if we start at a point, and move in the opposite direction to the gradient, we will get closer to a local minimum - this aids optimisation.

Let us formalise this:

Definition 3.5: Given a smooth function $f : \mathbb{R}^d \to \mathbb{R}$ and an initial point $\mathbf{x}^{(0)}$, we define a recursive sequence of points via:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \alpha \nabla f(\mathbf{x}^{(k-1)}),$$

where $\alpha \in \mathbb{R}_{>0}$ is a constant called the *learning rate*. This construction is called *gradient descent iteration*.



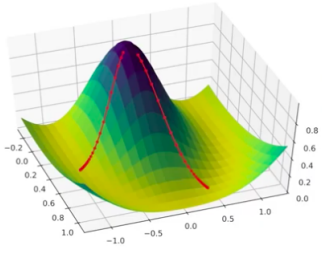
It is possible to show that:

Proposition 3.6: For smooth, *convex* functions with a single minimum $\mathbf{x}^* \in \mathbb{R}^d$, we have:

$$f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*) = O\left(\frac{1}{k}\right),$$

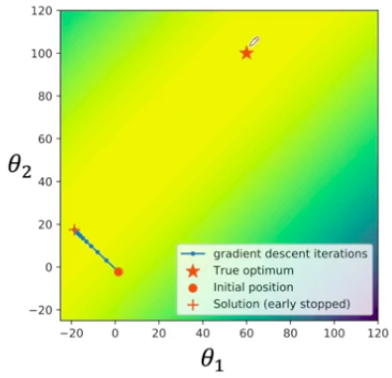
i.e. as we take more and more points, we approach the true minimum of the function.

Gradient descent may also be applied to non-convex functions, but we may reach a minimum which is not *global*, and the result may *depend on the starting point*.



A useful property of gradient descent is that it can be used for *regularisation* when our loss function is ill-defined; for example, when our $X^T X$ matrix above is not invertible (or not stably invertible).

In such cases, the true minimum typically has very large parameter values. These often correspond to *overfitting*. To avoid this, we can start from a point where our initial parameters are *small* and we can stop the gradient descent early (this is called *early stopping*). This will typically be a better solution than the true minimum.



There is an important modification to the gradient descent algorithm called *stochastic gradient descent*. Recall that in machine learning we typically optimise loss functions which are averages over some set of objects:

$$L = \frac{1}{N} \sum_{i=1,\ldots,N} \mathcal{L}\left(f(x_i), \hat{f}_\theta(x_i)\right).$$

For large N, gradient descent is computationally inefficient and may be unfeasible in terms of memory consumption. Stochastic gradient descent deals with this problem:

Definition 3.7: Given a function $\hat{f}_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ which depends on a parameter $\boldsymbol{\theta} \in \mathbb{R}^d$, a starting point $\boldsymbol{\theta}^{(0)}$, and a set of objects $x_1, \ldots, x_N \in \mathcal{X}$, we define a recursive sequence of points as follows:

- At step k of the process, pick some $l_k \in \{1, \ldots, N\}$ at random.
- Define:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(f(x_{l_k}), \hat{f}_{\boldsymbol{\theta}^{(k-1)}}(x_{l_k})\right).$$

This is called *stochastic gradient descent iteration*.

In the case of a smooth convex function, stochastic gradient descent also leads you to the unique minimum, but at a slower rate of $O(1/\sqrt{k})$ instead. This *can* be improved by 'batching' and other tricks.

3.4 Feature expansion

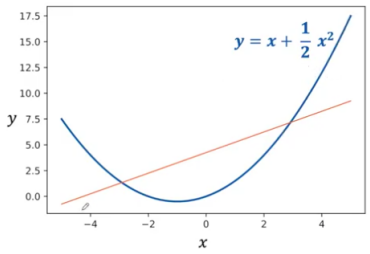
Let us now see how linear models can be made more powerful by transforming their features.

Definition 3.8: Given a design matrix X of dimension $N \times d$, a *feature transformation* via the function $\Phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is the induced map of the design matrix:

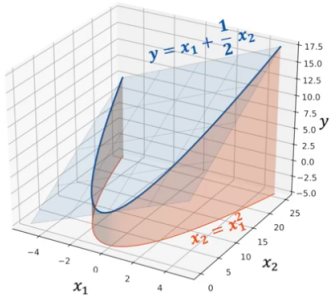
$$X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{pmatrix} \quad \to \quad \Phi(X) = \begin{pmatrix} \Phi^1(x_1^1, ..., x_1^d) & \cdots & \Phi^{d'}(x_1^1, ..., x_1^d) \\ \Phi^1(x_2^1, ..., x_2^d) & \cdots & \Phi^{d'}(x_2^1, ..., x_2^d) \\ \vdots & \ddots & \vdots \\ \Phi^1(x_N^1, ..., x_N^d) & \cdots & \Phi^{d'}(x_N^1, ..., x_N^d) \end{pmatrix}$$

Finding good functions Φ to transform with is called *feature engineering*. It is an important part of machine learning and requires a deep understanding of the underlying problem and the data.

Example 3.9: (**Polynomial features.**) Consider trying to fit the curve $y = x + \frac{1}{2}x^2$. This cannot be solved using only linear regression; a single line will not work.



However, we can introduce a new feature via a feature transformation. Let $(x_1, x_2) = (x, x^2)$. Then in the new feature space, a linear estimate is really an estimate of the form $\hat{f}(\mathbf{x}) = \theta_1 x_1 + \theta_2 x_2 = \theta_1 x + \theta_2 x^2$, which facilitates fitting.



Example 3.10: (**Polynomial features - general case**) Let $(x_i^1, x_i^2, ..., x_i^d)$ be the original features of a dataset. We introduce as new features all unique multiplicative combinations of the form:

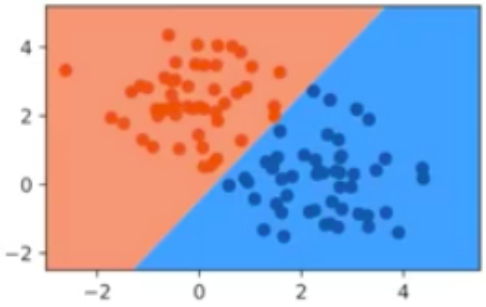
$$(x_i^1)^{p_1}(x_i^2)^{p_2}...(x_i^d)^{p_d},$$

where $p_1 + p_2 + \cdots + p_m \leq p$ (with p the degree we expect to work).

4 Classification with linear models

4.1 Classification with linear regression

Consider a classification problem, with two classes, as shown below:

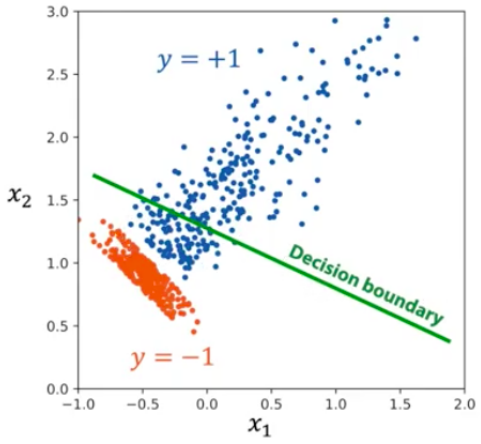


Since there are only two classes, we can assign numeric values to the classes - for our convenience, we choose $y = +1$ for one class (the 'positive' class) and $y = -1$ for the other class (the 'negative' class)

We can try to predict these targets with a linear model. To do so, we solve a linear regression problem using the model $\hat{y}_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$. To obtain the classified function, we then simply take the sign of the result:

$$\hat{f}(\mathbf{x}) = \text{sign}(\hat{y}(\mathbf{x})).$$

This may seem like a reasonable approach, but you can face problems. Consider the following set:



Here, the MSE loss makes the model avoid high errors, at the price of *pushing the decision boundary* towards the class with higher spread.

Therefore, we might try to use a different loss function. A good first example is 0-1 *loss*.

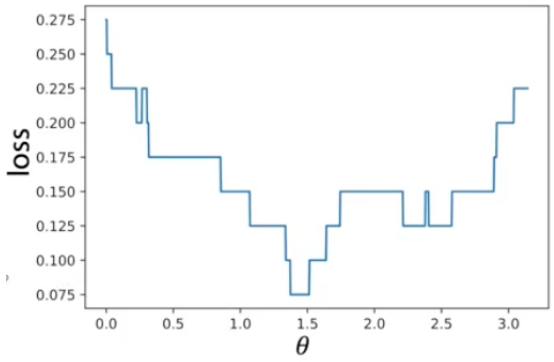
Definition 4.1: For the above classification problem, we define the 0-1 *loss by*:

$$\mathcal{L}_{0\text{-}1} = \frac{1}{N} \sum_{i=1,\ldots,N} 1_{\{\boldsymbol{\theta}^T \mathbf{x}_i y_i < 0\}}$$

where $y_i \in \{-1, +1\}$. We call $M = \boldsymbol{\theta}^T \mathbf{x} y$ the *margin*. When $M > 0$, we have a *correct* classification, but when $M < 0$ we have an *incorrect* classification. The higher the magnitude of M, the further an object is from the decision boundary.

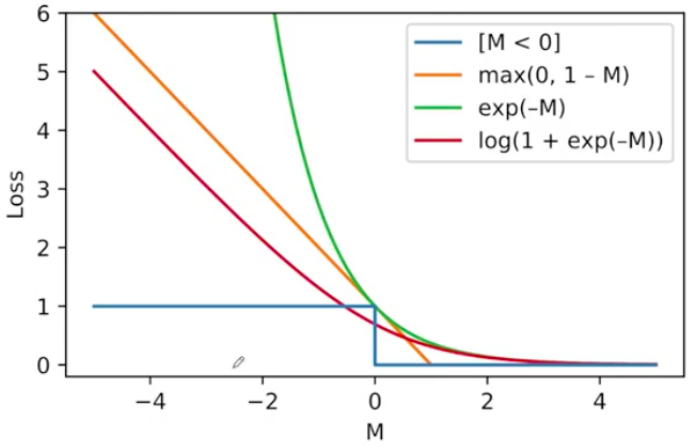
The sum computes the number of times our linear regression misclassifies a point.

Unfortunately, the resulting 0-1 loss function is a piecewise constant function of $\boldsymbol{\theta}$:



This means that we cannot use *gradient descent* to minimise our loss (of course, there are other methods - we will discuss them later).

Instead of using 0-1 loss then, we could replace the sharp 'step' function described by the loss by a *differentiable upper bound*:



Examples are shown in the above figure (the red line can be scaled to be above the step function). We will discuss the red line (*log loss*) in much more detail as we discuss *logistic regression*.

4.2 Logistic regression

Let's depart from the discussion above, and approach things from a different way to start with. Consider modelling the *class probabilities*: we introduce a function $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x})$ that gives the probability of an object $\mathbf{x}$ being in the class $y = +1$, so that:

$$\mathbb{P}(y = +1|\mathbf{x}) = \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}), \qquad \mathbb{P}(y = -1|\mathbf{x}) = 1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}).$$

How can we optimise such a model? We can fit with the *maximum log likelihood*.

Recall that the *likelihood* is defined by:

$$\text{likelihood} = \prod_{i=1,\dots,N} \mathbb{P}(y_i|\mathbf{x}_i) = \prod_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i) + 1_{\{y_i=-1\}} \left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right).$$

Maximising the logarithm likelihood allows us to obtain the best values of the parameter $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right).$$

Note that the indicator functions can be moved out of the logarithm because only one of them is equal to 1 at a time.

Once we have fitted the parameter values, we can for example predict the class the *highest probability* for each $\mathbf{x} \in \mathcal{X}$ in the object space. More generally, we could not just choose the highest probability class but try to use a probability threshold that is suitable for our problem.

Let's try to relate this back to linear models now. We need to ask: how can we map a linear model output to a probability value in $[0, 1]$? A common choice is the *sigmoid function*:

Definition 4.2: The *sigmoid function* is defined by:

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

It has the useful property that $1 - \sigma(x) = \sigma(-x)$.

This can be applied to the above example, since we can define:

$$\mathbb{P}(y = +1|\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x}).$$

where $\boldsymbol{\theta}^T \mathbf{x}$ is our linear model. The linear model $\boldsymbol{\theta}^T \mathbf{x}$ then has the meaning of *log odds ratio* between the two classes:

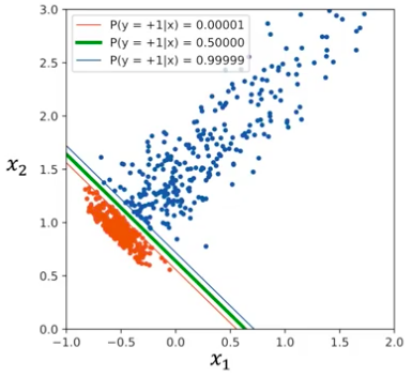
$$\log\left(\frac{\mathbb{P}(y = +1|\mathbf{x})}{\mathbb{P}(y = -1|\mathbf{x})}\right) = \log\left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \cdot \frac{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}{e^{-\boldsymbol{\theta}^T \mathbf{x}}}\right) = \boldsymbol{\theta}^T \mathbf{x}.$$

Therefore to bring everything together, we use the negative log likelihood as our loss function:

$$\begin{aligned}
\mathcal{L} &= -\sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)\right) \\
&= \sum_{i=1,\dots,N} \left(1_{\{y_i=+1\}} \log\left(\sigma(\boldsymbol{\theta}^T \mathbf{x}_i)\right) + 1_{\{y_i=-1\}} \log\left(\sigma(-\boldsymbol{\theta}^T \mathbf{x}_i)\right)\right) \\
&= -\sum_{i=1,\dots N} \log\left(\sigma(\boldsymbol{\theta}^T \mathbf{x}_i y_i)\right) = \sum_{i=1,\dots N} \log\left(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_i y_i}\right).
\end{aligned}$$

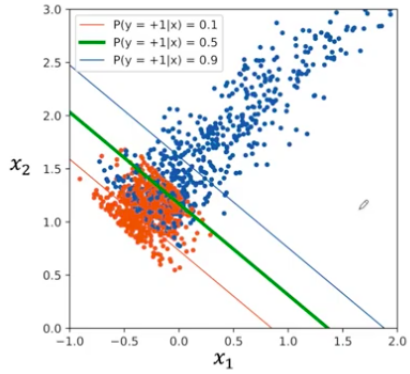
This is the same as the 'red' example we used above when we discussed upper bounds on 0-1 loss. We can apply gradient descent or stochastic gradient descent here.

Applying to the dataset above, the decision boundary is in the correct place with logistic regression:



Note that when the classes are linearly separable as above, for any correct decision boundary, mapping $\boldsymbol{\theta} \to C \cdot \boldsymbol{\theta}$ for some $C > 1$ keeps the boundary at the same place, yet improves the loss. The ideal fit is when the sigmoid turns into a step function (at infinitely large $\boldsymbol{\theta}$.

If the classes overlap however, the predicted class probability changes smooth, and the loss has a finite minimum.



4.3 Multinomial logistic regression

Similarly to the binary case, we can model the class probabilities when there are more classes than 2. Let's model the *unnormalised* class probabilities as:

$$\tilde{\mathbb{P}}(y = k|\mathbf{x}) = \exp\left(\boldsymbol{\theta}_k^T \mathbf{x}\right),$$

where we now have K parameter vectors, $k = 1, ..., K$. The normalised probabilities can be straightforwardly obtained:

$$\mathbb{P}(y = k|\mathbf{x}) = \frac{\exp\left(\boldsymbol{\theta}_k^T \mathbf{x}\right)}{\sum_{k'=1,...,K} \exp(\boldsymbol{\theta}_{k'}^T \mathbf{x})}$$

This function is called *softmax* and is commonly used in neural networks.

Note that there is a transformational symmetry here: transforming $\boldsymbol{\theta}_k \mapsto \boldsymbol{\theta}_k + \mathbf{v}$ by some constant vector $\mathbf{v}$ does not affect the normalised probability. We have:

$$\tilde{\mathbb{P}}(y = k|\mathbf{x}) = e^{\boldsymbol{\theta}_k^T \mathbf{x}} \to e^{\mathbf{v}^T \mathbf{x}} \cdot e^{\boldsymbol{\theta}_k^T \mathbf{x}} = e^{\mathbf{v}^T \mathbf{x}} \cdot \tilde{\mathbb{P}}(y = k|\mathbf{x}).$$

The remaining exponential cancels out in the ratio in the normalised probabilities. Therefore, we have some extra degree of freedom, which we are free to choose - we can therefore set $\boldsymbol{\theta}_K = \mathbf{0}$ without loss of generality for example (i.e. we set the last parameter to zero). We are left with $K - 1$ parameter vectors.

Individual linear outputs $\boldsymbol{\theta}_k^T \mathbf{x}$ now have the meaning of *log odds ratio* between the classes k and K:

$$\log\left(\frac{\mathbb{P}(y = k|\mathbf{x})}{\mathbb{P}(y = K|\mathbf{x})}\right) = \log\left(\frac{\tilde{\mathbb{P}}(y = k|\mathbf{x})}{\tilde{\mathbb{P}}(y = K|\mathbf{x})}\right) = \log\left(\frac{e^{\boldsymbol{\theta}_k^T \mathbf{x}}}{e^0}\right) = \boldsymbol{\theta}_k^T \mathbf{x}.$$

Putting everything into the negative log likelihood function, we again get our loss function:

$$\mathcal{L} = -\sum_{i=1,...,N} \log\left(\frac{\exp(\boldsymbol{\theta}_{y_i}^T \mathbf{x}_i)}{1 + \sum_{k'=1,...,K-1} \exp\left(\boldsymbol{\theta}_{k'}^T \mathbf{x}_i\right)}\right),$$

where $\boldsymbol{\theta}_K = \mathbf{0}$. Again, this can be optimised *numerically* using gradient descent or stochastic gradient descent.

4.4 Multiclass classification: general approach

In fact, you are not limited to use logistic regression for multiclass classification - you are able to use any linear model.

For a problem with K classes, introduce K predictors:

$$\hat{f}_k(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$$

for $k = 1, ..., K$ each of which outputs a corresponding *class score*. We can use this to predict the class with the *highest score*:

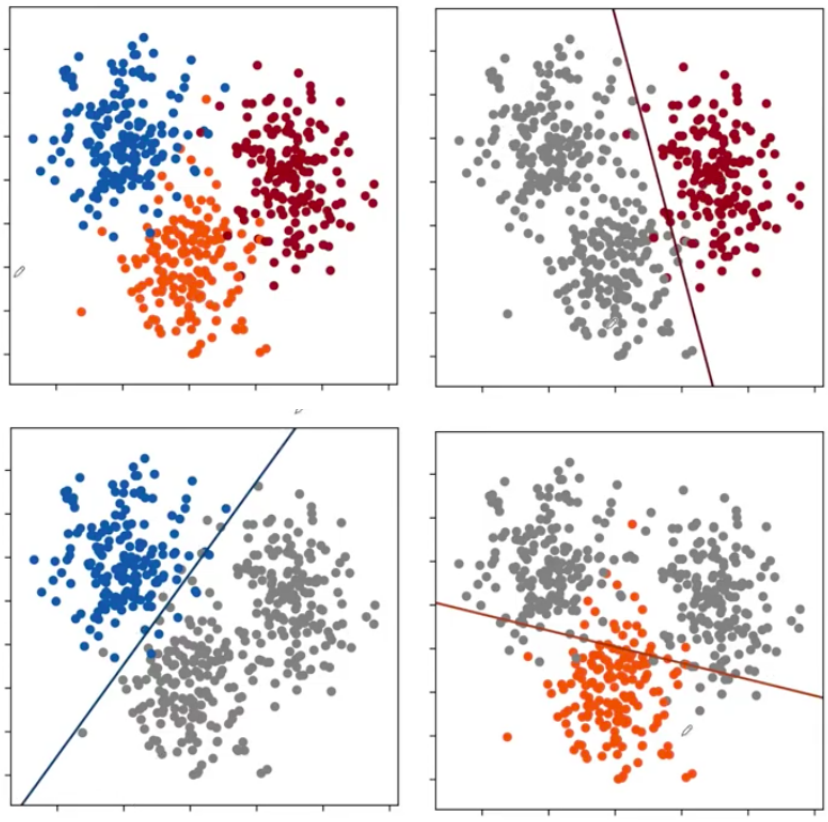
$$\hat{y}_i = \underset{k}{\operatorname{argmax}} \hat{f}_k(\mathbf{x}_i).$$

For example, any binary linear classification model can be converted to a multiclass classification with the *one-versus-rest* strategy:

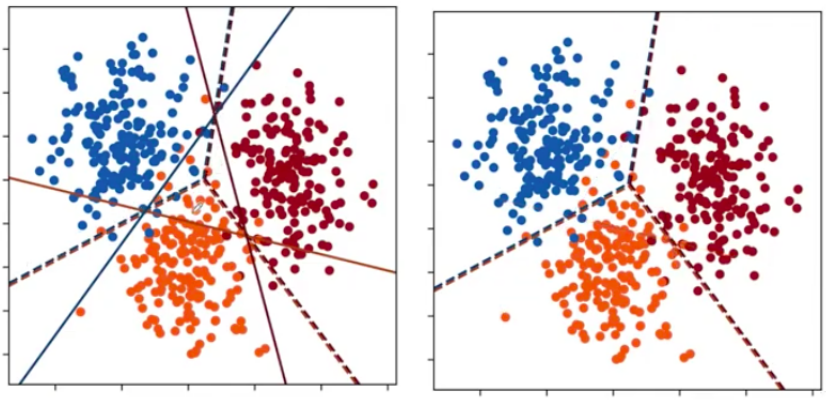
- For each class k train a binary model $\hat{f}_k(\mathbf{x}) = \boldsymbol{\theta}_{(k)}^T \mathbf{x}$ separating the given class from all others, $\hat{y}_{(k)}^{\text{one-vs-rest}} = \operatorname{sign}(\hat{f}_k(\mathbf{x}))$.
- Use the outputs of $\hat{f}_k$ as class scores for multiclass classification:

$$\hat{y}_i = \underset{k}{\operatorname{argmax}} \hat{f}_k(\mathbf{x}_i).$$

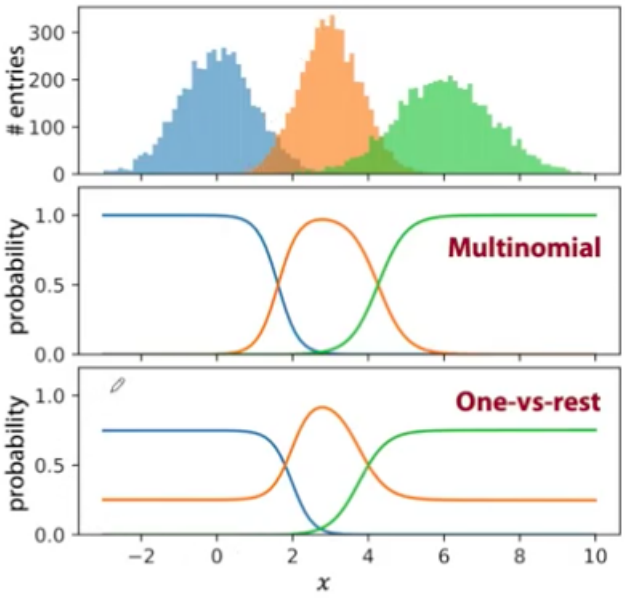
Example 4.3: In the following three-class classification, the one-versus-rest splits given by using the appropriate binary classifiers are shown below:



The resulting decision boundaries can be built up as follows:



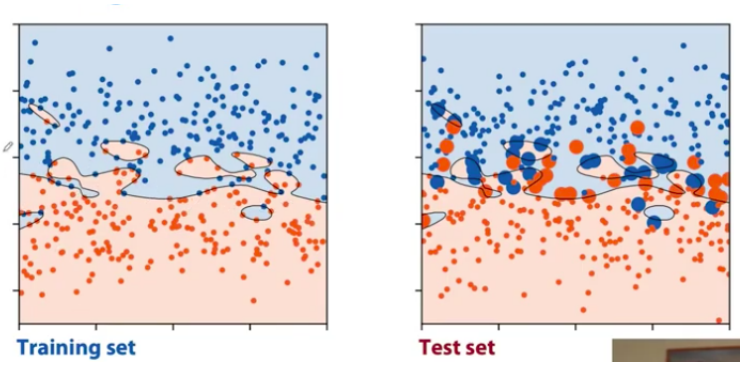
Is the one-vs-rest approach better than multinomial logistic regression however? Some problems are not linearly separable, so one-vs-rest results in biased class probabilities whilst logistic regression is still quite accurate:



5 Model regularisation

5.1 The problem of overfitting

Overfitting is the tendency of a model to adjust to random fluctuations in the data. This can be detected by a model making poor predictions on the *test set*.



Recall that we need to use a model complexity such that we don't underfit, and we don't overfit - we need to find the 'sweet spot' in between.

5.2 Prediction error decomposition

Before talking about regularisation, let's discuss the prediction error of a model. Assume that there's the following unknown relation between features and targets:

$$y = f(x) + \epsilon,$$

where ϵ is some random noise, $\mathbb{E}[\epsilon] = 0$ and $\mathrm{Var}[\epsilon] = \sigma_\epsilon^2$. Let's denote a training set as τ. We want to study the *expected squared error* for the model $\hat{f}_\tau$ trained on the dataset τ:

$$\mathrm{exp.sq.err}(x) = \mathop{\mathbb{E}}_{\tau,y}\left[(\hat{f}_\tau(x) - y)^2 | x\right].$$

The target y is sampled independently of the training set.

We can rewrite this expression in the following form, by adding and subtracting the 'prediction of the expected model' and the 'ground truth' $f(x)$ (without the noise):

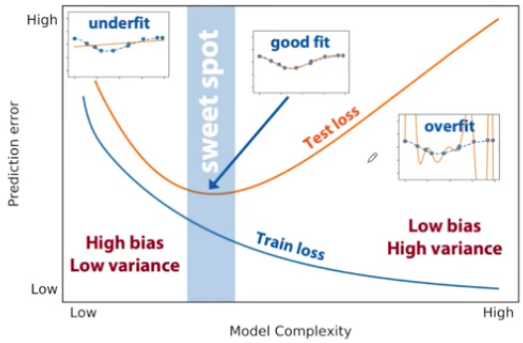
$$\begin{aligned}
\mathrm{exp.sq.err}(x) &= \mathop{\mathbb{E}}_{\tau,y}\left[(\hat{f}_\tau(x) - y)^2 | x\right] \\
&= \mathop{\mathbb{E}}_{\tau,y}\left[\left((\hat{f}_\tau(x) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)]) + (\mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)] - f(x)) + (f(x) - y)\right)^2 \middle| x\right] \\
&= \mathop{\mathbb{E}}_{\tau}\left[\left(\hat{f}_\tau(x) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)]\right)^2\right] + \left(\mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(x)] - f(x)\right)^2 + \mathop{\mathbb{E}}_{y}[(f(x) - y)^2 | x].
\end{aligned}$$

In the final step, we have expanded everything and used the fact that the cross-term expectation are 0 (this follows from the fact that τ, y are sampled independently).

The remaining terms have important interpretations:

- The first term is the *variance of the model*. It quantifies how 'unstable' the model is with respect to the noise in the training data.
- The second term is the *squared bias*. It is how much the 'expected model' differs from the ground truth.
- The third term is the *irreducible error*. Substituting the true dependence y, we find it is just given by σ_ϵ^2. This error cannot be eliminated.

The first two errors on the other hand, can be removed by changing the model. Typically there's a *tradeoff* between the two types of errors:



For low model complexity, the *bias* is high because we are artificially simplifying the model - it doesn't depend much on the data (it has 'low variance'). The opposite is true for high model complexity. This tradeoff is called the *bias-variance tradeoff*.

Example 5.1: Let's compute the bias and variance of a linear model. For each expectation $\mathbb{E}$ with respect to τ, we will assume that the features are *fixed*, i.e. $X_\tau = X$ (i.e. the design matrix is constant), and only the target vector y_τ is random. This simplification allows us to analytically compute the result.

Recall the solution for the linear regression model with the MSE loss:

$$\hat{f}_\tau(\mathbf{x}) = \boldsymbol{\theta}_\tau^T \mathbf{x}, \qquad \boldsymbol{\theta}_\tau = (X^T X)^{-1} X^T \boldsymbol{y}_\tau.$$

The *bias term* from the above error decomposition is:

$$\text{bias}(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}[\hat{f}_\tau(\mathbf{x})] - f(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}\left[\mathbf{x}^T (X^T X)^{-1} X^T \mathbf{y}_\tau\right] - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}}$$

We also assume that the true dependence is indeed linear, i.e. $f(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}_{\text{true}}$ for some $\boldsymbol{\theta}_{\text{true}}$.

Since X does not depend on the sampling dataset τ, we can move lots of things out of the expectation, leaving:

$$\mathbf{x}^T (X^T X)^{-1} X^T \mathop{\mathbb{E}}_{\tau}[\mathbf{y}_\tau - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = \mathbf{x}^T (X^T X)^{-1} X^T X \boldsymbol{\theta}_{\text{true}} - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} - \mathbf{x}^T \boldsymbol{\theta}_{\text{true}} = 0.$$

Hence we see that the linear regression model is *unbiased*, provided the true dependence is linear.

Now let's consider the *variance term*:

$$\text{variance}(\mathbf{x}) = \mathop{\mathbb{E}}_{\tau}\left[\left(\hat{f}_{\tau}(\mathbf{x}) - \mathop{\mathbb{E}}_{\tau'}[\hat{f}_{\tau'}(\mathbf{x})]\right)^2\right].$$

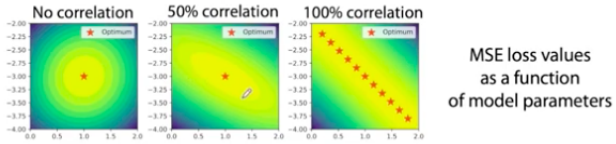
It can be shown that:

$$\text{variance}(\mathbf{x}) = \sigma_{\epsilon}^2 \mathbf{x}^T (X^T X)^{-1} \mathbf{x},$$

so that the variance error component is a *quadratic form*, defined by the $(X^T X)^{-1}$ matrix. We can diagonalise $X^T X$ giving:

$$\text{variance}(\mathbf{x}) = \sigma_{\epsilon}^2 \tilde{\mathbf{x}}^T \Lambda^{-1} \tilde{\mathbf{x}},$$

where $\Lambda = \text{diag}\{\lambda_1, ..., \lambda_d\}$ is the diagonal matrix of eigenvalues of $X^T X$. This means that *small eigenvalues amplify the model variance*. This happens when $X^T X$ is ill-defined, e.g. when the features are correlated.



In a high variance model, we expect that a small perturbation data will lead to a large change in the prediction.

5.3 Regularisation

How can we reduce the variance of a model? If only we could *increase the eigenvalues* of the matrix $X^T X$. In fact, we can do this manually:

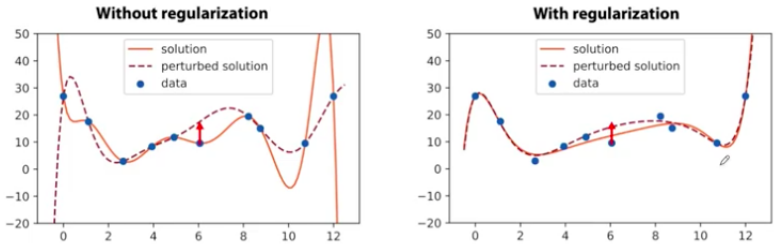
$$X^T X \to X^T X + \alpha I,$$

for $\alpha > 0$, with I a unit $d \times d$ matrix. This is called $L2$ *regularisation*.

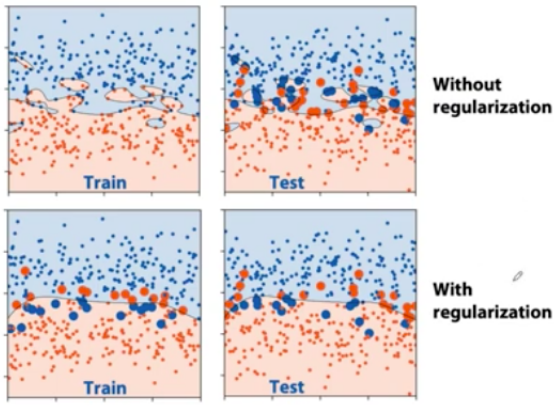
In this case, we change the solution to:

$$\hat{f}_{\tau}(\mathbf{x}) = \mathbf{x}^T (X^T X + \alpha I)^{-1} X^T \mathbf{y}_{\tau}.$$

Now the model is *no longer biased*. We *increased bias to reduce variance* (we no longer get zero for the bias term of this model.



By regularising the model, we increase the training loss and decrease the test loss. This improves the *generalisability* of the model.



In order to understand this trick better, let's reverse engineer the loss function it optimises. We have the solution $\hat{f}_\tau(\mathbf{x}) = \mathbf{x}^T(X^TX + \alpha I)^{-1}X^T\mathbf{y}_\tau$, so the $\boldsymbol{\theta}_\tau$ vector should be given by:

$$\boldsymbol{\theta}_\tau = (X^TX + \alpha I)^{-1}X^T\boldsymbol{y}_\tau.$$

Regrouping the terms, we have:

$$(XT^X + \alpha I)\boldsymbol{\theta}_\tau = X^T\boldsymbol{y}_\tau \qquad \Rightarrow \qquad X^T(X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau) + \alpha\boldsymbol{\theta}_\tau = \mathbf{0}.$$

In fact, this is the equation $\partial\mathcal{L}/\partial\boldsymbol{\theta}_\tau = \mathbf{0}$ for the loss function:

$$\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2.$$

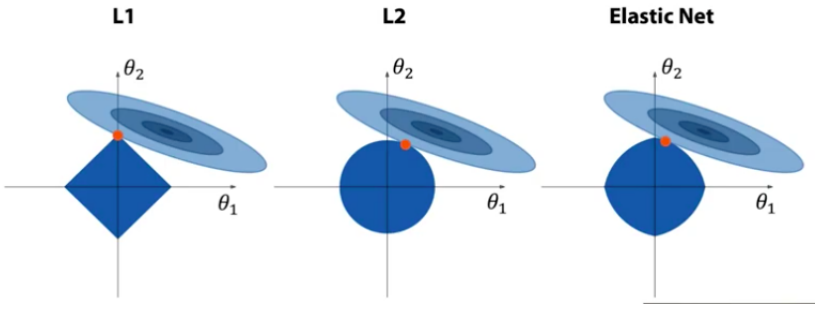
This explains the origin of the term $L2$ *regularisation*. The model minimises the MSE loss with an $L2$ *penalty theorem* (this model is also called *ridge regression*).

There are other similar regularisation methods:

- $L2$ *regularisation* or *ridge regression* takes as the loss function $\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2$.
- $L1$ *regularisation* or *lasso regularisation* takes as the loss function $\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||_1$ (where $||\cdot||_1$ denotes the $L1$ norm.
- *Elastic net regression* is a combination of the two:

$$\mathcal{L} = ||X\boldsymbol{\theta}_\tau - \mathbf{y}_\tau||^2 + \alpha||\boldsymbol{\theta}_\tau||^2 + \beta||\boldsymbol{\theta}_\tau||_1.$$

The introduction of regularisation methods drive the parameters towards smaller values, yet they *induce different properties* of the solution.



For example, the $L1$ regularisation term has unit ball given by a diamond in θ_1, θ_2 space (i.e. the surface where $||\boldsymbol{\theta}||_1 = 1$). It's quite likely that among all the points at the perimeter of the diamond (where the penalty term does not change), the intersection is most likely at the coordinate axes. This means that this loss tends to *sparsify the solution* - it tends to set some parameters to zero.

For the $L2$ regularisation, we instead have *weight sharing* - it uniformly drags the values of the parameters towards zero.

The elastic net is a compromise between the two approaches.

5.4 Probabilistic view

Let's revisit our assumption about the data, $y = f(x) + \epsilon$. Let's now assume that the label noise is *normally distributed*, $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. This means that the targets, given the objects, are also normally distributed:

$$y|x \sim \mathcal{N}(f(x), \sigma_\epsilon^2).$$

We want our model $\hat{f}_\theta(x)$ to fit the true dependence $f(x)$, i.e. we define a *probabilistic model*:

$$y|x \sim \mathcal{N}(\hat{f}_\theta(x), \sigma_\epsilon^2),$$

where the mean of the normal distribution is described by our function.

Our model can be fitted with the maximum likelihood approach:

$$L = \prod_{i=1,\dots,N} \mathcal{N}(y_i|\hat{f}_\theta(x_i), \sigma_\epsilon^2).$$

Maximising this with respect to θ is equivalent to minimising the negative log likelihood,

$$-\log(L) = \sum_{i=1,\dots,N} \log(\mathcal{N}(y_i|\hat{f}_\theta(x_i), \sigma_\epsilon^2)) = -\sum_{i=1,\dots,N} \left[\log\left(\exp\left(-\frac{(y_i - \hat{f}_\theta(x_i))^2}{2\sigma_\epsilon^2}\right)\right) - \log(\sqrt{2\pi}\sigma_\epsilon^2)\right]$$

$$= C \cdot \sum_{i=1,\dots,N} (y_i - \hat{f}_\theta(x_i))^2 + \text{constant}.$$

Thus we just get MSE loss! Thus MSE loss is equivalent to modelling the probability with a normal label noise.

You can do similar tricks for other loss functions and show that different probabilistic models lead to different loss functions.

5.5 Bayesian view

In the Bayesian view, we treat both data (X, y) and model parameters (θ) as random variables. We estimate the parameter distribution given the observed data via Bayes' rule:

$$p(\theta|X, y) = \frac{p(y|\theta, X)p(\theta)}{\int (p(y|\theta, X)p(\theta))\ d\theta}$$

We assume that θ, X are independent variables here. Here, $p(\theta)$ is our prior knowledge about the model parameters (our belief about how they are distributed before we see any data). The function $p(y|\theta, X)$ is our likelihood function. The distribution $p(\theta|X, y)$ is called the *posterior distribution*, and is our knowledge about the model after seeing data. The denominator is called 'evidence' (the probability of observing the data when the parameter uncertainty is integrated out).

To get some estimate of the parameter, we can calculate the *maximum a posteriori* (where we ignore the denominator since is it is integrated over θ - the maximum is the same as the maximum of the numerator):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, p(y|\theta, X)p(\theta) = \underset{\theta}{\operatorname{argmin}}\, [-\log(p(y|\theta, X) - \log(p(\theta))]$$

Similarly, the maximum is the same as the minimum negative log likelihood. The term $\log(p(\theta))$ is called a *regularising* term.

Example 5.2: Suppose we model the data with a normal distribution $y|x \sim \mathcal{N}(\hat{f}_\theta(x), \sigma_\epsilon^2)$. Suppose the prior is normal too $\theta \sim \mathcal{N}(0, \sigma_\theta^2 I)$, where I is the unit matrix so that the parameters are uncorrelated. Then maximum a posteriori estimate corresponds to minimising the following loss:

$$\mathcal{L} = -\log(p(y|\theta, X)) - \log(p(\theta)) = C_1 \sum_{i=1,\ldots,N} (\hat{f}_\theta(x_i) - y_i)^2 + C_2||\theta||^2 + \text{constant.}$$

In other words a normal prior is equivalent to $L2$ regularisation of the parameters.

Choosing different priors will lead to different regularisation of the parameters.

6 Quality metrics for regression and classification

6.1 Quality metrics for regression models

Consider a dataset described by design matrix X and target values $\mathbf{y}$, and a linear regression model:

$$\hat{\mathbf{y}} = X\mathbf{w},$$

where $\mathbf{w}$ are the parameters (weights) of the model (previously denoted as $\boldsymbol{\theta}$). Our goal is to *measure the quality* of this model - in other words, estimate how close predictions $\hat{\mathbf{y}}$ are to the real target values. This is measured by a *loss function*, also called a *quality metric* in this section of the course.

We have already seen some examples of quality metrics: for example, mean squared error (MSE) and mean absolute error (MAE). Sometimes MSE is replaced by *root mean squared error* (RMSE):

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}.$$

With these metrics, it can be hard to tell if a model is good: for example, if RMSE$= 1$, the model quality can be different for mean target values of $\overline{y} = 100$ and $\overline{y} = 1$.

Another popular metric is MAPE, which we also saw before. It is defined by:

$$\frac{100}{N}\sum_{i=1}^{N}\left|\frac{\hat{y}_i - y_i}{y_i}\right|.$$

It measures the relative error of the prediction. It is easy to understand the quality of the model, but it is sensitive to the y scale.

Two other quality metrics arE:

- **Relative square error (RSE).** This is given by:

$$\sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}}.$$

  This metric measures how the average square error differs from the squared standard deviation of the real target values.

- **Relative absolute error (RAE).** This is given by:

$$\frac{\sum_{i=1}^{N}|y_i - \hat{y}_i|}{\sum_{i=1}^{N}|y_i - \overline{y}|}.$$

  Similarly, this metric measures how the average absolute error of the model differs from the the median value of the real target values.

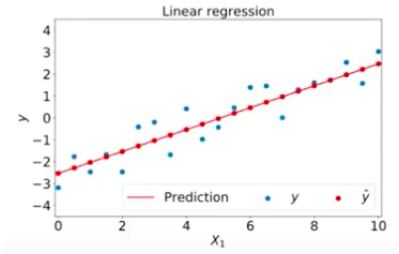
The main advantage of these metrics is that they are robust to the scale of the target values y.

The final quality metric we will consider is *root mean squared logarithmic error* (RMSLE), which we also saw before:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(y_i+1)-\log(\hat{y}_i+1))^2}.$$

It is similar to root mean square error, but uses logarithmic values instead. It is used in cases when the target changes by several orders of magnitude - in such cases, the other quality metrics will measure quality less accurately.

Example 6.1: Consider the linear regression shown below:

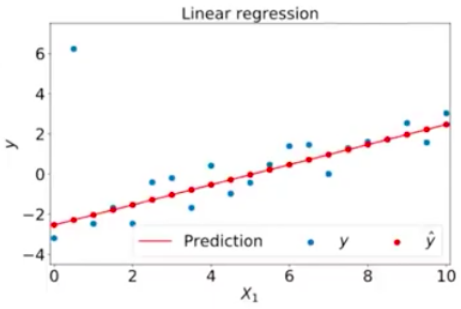


Computing all of the quality metrics in this case, we have:

Metric	No outliers
RMSE	0.67
MAE	0.59
MAPE, %	1035
RSE	0.39
RAE	0.40

Note that MAPE fails because of the y scale, and the fact that some y_i are close to zero.

Now consider adding a single outlier to the dataset:



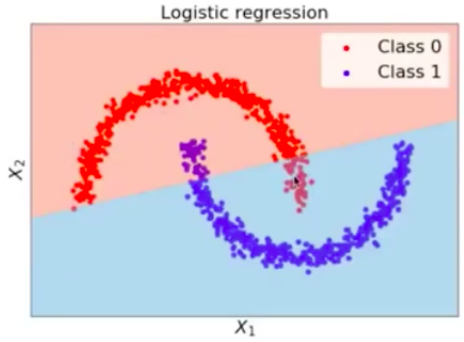
The outlier significantly affects the metrics:

Metric	No outliers	With outlier
RMSE	0.67	1.93
MAE	0.59	0.96
MAPE, %	1035	1040
RSE	0.39	0.92
RAE	0.40	0.58

However, we note that MAE and RAE are the most robust against the addition of the outlier.

6.2 Quality metrics for classification models

Consider a binary classification problem with a data sample and a classifier (into two classes 'positive' and 'negative'), as shown in the figure. We use a logistic regression model to find the decision boundary.



The goal is to measure the quality of the classifier, i.e. to estimate how well it separates objects of different classes.

To measure the model's quality, we introduce a *confusion matrix*:

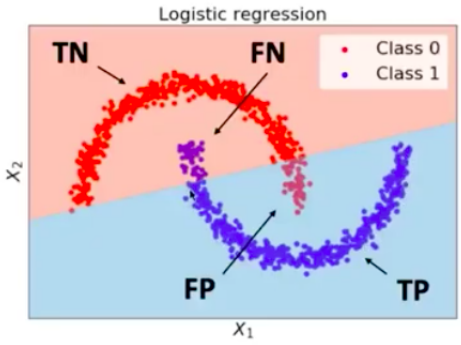
Definition 6.2: A *confusion matrix* is a matrix of the form:

$$\begin{pmatrix} \text{TP} & \text{FN} \\ \text{FP} & \text{TN} \end{pmatrix}.$$

The element TP represents the number of objects which are positive and are correctly predicted as positive (*true positives*). The element FN represents the number of objects which are positive and are incorrectly predicted as negative (*false negatives*).

The element FP represents the number of objects which are negative and are incorrectly predicted as positive (*false positives*). The element TN represents the number of objects which are negative and are correctly predicted as negative (*true negatives*).

In the example above, we have the following assignments:



We also define some additional quantities:

- The number of all positives is $\text{Pos} = \text{TP} + \text{FN}$.
- The number of all negatives is $\text{Neg} = \text{TN} + \text{FP}$.
- The number of all positive predictions is $\text{PosPred} = \text{TP} + \text{FP}$.
- The number of all negative predictions is $\text{NegPred} = \text{TN} + \text{FN}$.

Using these quantities, we can define different quality metrics. The first metric we define is *accuracy*:

Definition 6.3: The *accuracy* of a classifier is defined by:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{TN} + \text{FP}} = \frac{\text{TP} + \text{TN}}{\text{Pos} + \text{Neg}}.$$

A complementary quality metric is *error rate*, defined by:

$$\text{Error rate} = 1 - \text{Accuracy}.$$

These metrics are symmetric and measure quality of classification of both classes equally.

Other metrics measure how well a classifier identifies positive objects only (i.e. they are asymmetric). Examples include:

Definition 6.4:

- *Precision* of a classifier is defined by:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{PosPred}}.$$

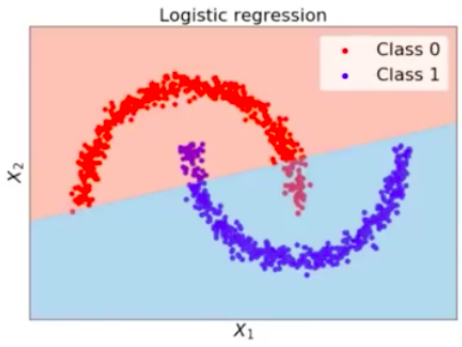
- *Recall* of a classifier is defined by:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{Pos}}.$$

- The F_1*-score* of a classifier is the harmonic mean of precision and recall, defined by:

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Example 6.5: Consider the classifier example we showed at the start of this section:



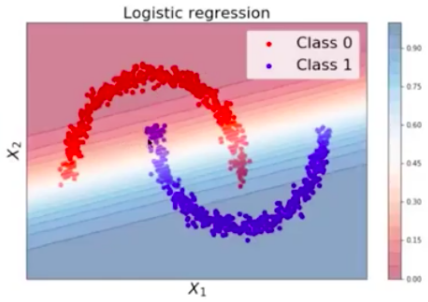
Computing all of the above metrics on this example, we find:

Metric	Value
Accuracy	0.89
Precision	0.89
Recall	0.89
F_1	0.89

All them metrics give the same value in this case, because of the symmetry of the problem. Later we will see other cases.

Recall that when we did classification problems with logistic regression earlier, we could also have based our ideas on *probabilities*. If an object has a probability p of being in the positive class, we predict it to have value 1 if $p \geq 0.5$ and value 0 if $p < 0.5$.

The probability in the above example is distributed as a gradient instead of a hard cutoff like the decision boundary:



We can use these probabilities to define quality metrics.

Definition 6.6: The *receiver operating characteristic curve* (ROC curve) is a curve relating the *true positive rate* $\mathrm{TPR}(\mu)$ and the *false positive rate* $\mathrm{FPR}(\mu)$ for different 'thresholds' μ of the predicted probabilities p. We make the following definitions first:

- The *true positive rate* is defined by:

$$\mathrm{TPR}(\mu) = \frac{1}{\mathrm{Pos}} \sum_{i \in \mathrm{Pos}} 1_{\{p_i \geq \mu\}} = \frac{\mathrm{TP}(\mu)}{\mathrm{Pos}},$$

  where $\mathrm{TP}(\mu)$ is the number of true positives which have predicted probability of being positive greater than μ.

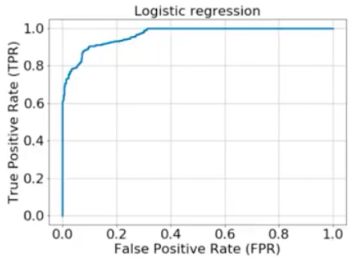
- The *false positive rate* is defined by:

$$\mathrm{FPR}(\mu) = \frac{1}{\mathrm{Neg}} \sum_{i \in \mathrm{Neg}} 1_{\{p_i \geq \mu\}} = \frac{\mathrm{FP}(\mu)}{\mathrm{Neg}},$$

  where $\mathrm{FP}(\mu)$ is the number of false positives which have predicted probability of being positive greater than μ.

The ROC curve is the curve in the TPR, FPR plane parametrised by the threshold μ.

Example 6.7: For our example above, with the 'gradient of probabilities' shown in the figure, the ROC curve looks like:

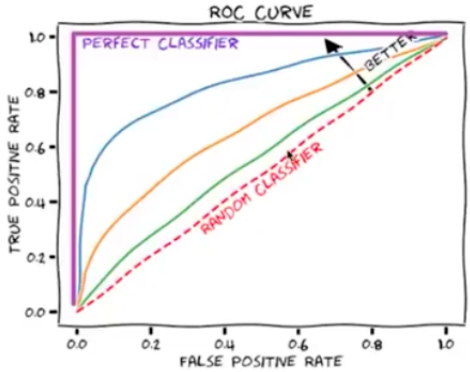


In high energy physics, we very often plot dependency of *background reduction* from *signal efficiency*. In this case, *signal efficiency* is to be interpreted as the true positive rate and *background reduction* is to be interpreted as one minus the false positive rate. The resulting curves are very similar to the ROC curve.

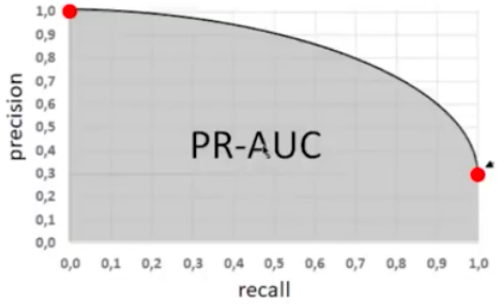
Next, we should comment on the interpretation of the ROC curve. The quality of a classifier is related to *area under the ROC curve*.

Definition 6.8: We define the quality metric *ROC AUC* to be the *area under the ROC curve*. We note that:

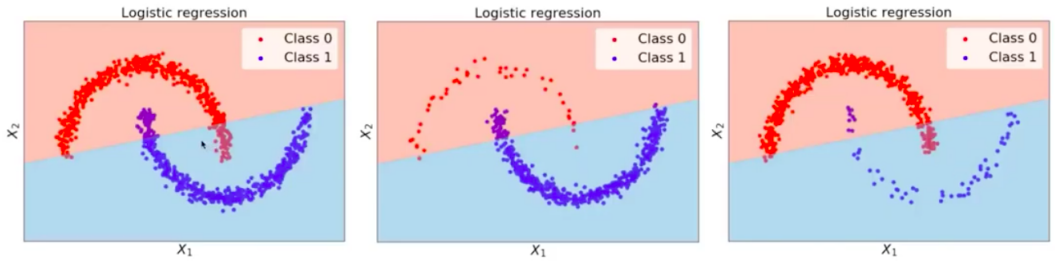
- ROC AUC $\in [0, 1]$.
- ROC AUC $= 0.5$ corresponds to random guessing.
- ROC AUC $= 1$ means ideal classification.
- ROC AUC $= 0$ also means ideal classification, but for opposite labels!



Similarly to the ROC curve, we can also define a *precision-recall curve* (PR), the curve with coordinates $(\text{Recall}(\mu), \text{Precision}(\mu))$ as the probability threshold μ is varied. This also induces a quality metric using the area under the curve (PR AUC).



Example 6.9: We can investigate each of these quality metrics by *fixing a model* but *changing the class balance* in the test sample. Let's choose to make the balances $1:1, 1:10$ and $10:1$ as shown below:



The resulting quality metrics are given by:

Metric	1:1	1:10	10:1
Accuracy	0.89	0.89	0.89
Precision	0.89	0.99	0.47
Recall	0.89	0.89	0.89
F_1	0.89	0.94	0.61
ROC AUC	0.97	0.97	0.97

With the class balance changing, some metrics change. We note that:

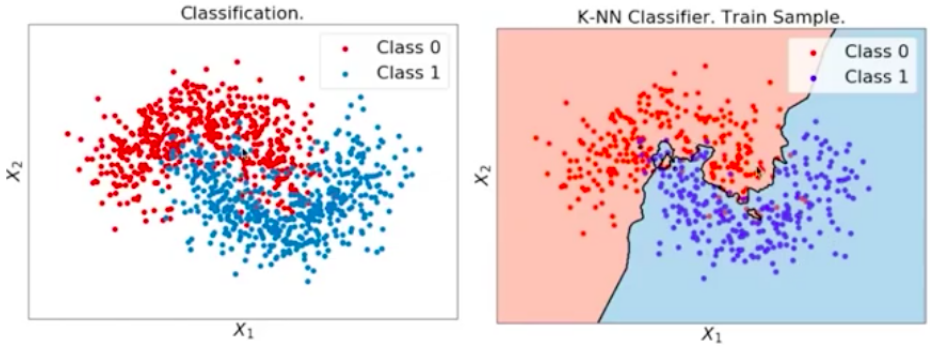
- Recall and ROC AUC do not change with the class balance changing. This is a general feature.
- Accuracy did not change with the class balance changing - however, this isn't true in general.

7 Model tests

7.1 Train/test split

We have already discussed the need for splitting our data into a training set and a test set. This is a quick reminder of how we can use this in regression and classification.

Example 7.1: Consider a binary classification problem. We randomly divide our data into two equal samples: *train* and *test*. We use the train sample to fit a classifier.



Note that in our case, the fitted model has quite an irregular decision boundary - this suggests that some overfitting has taken place, and the model may not be very generalisable.

After fitting the model, we can evaluate quality metrics on both the train and test sets. We find:

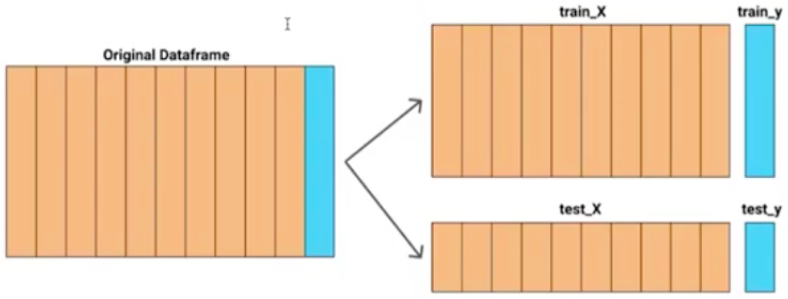
Metric	Train	Test
Accuracy	0.93	0.89
Precision	0.92	0.90
Recall	0.95	0.88
F_1	0.93	0.89
ROC AUC	0.99	0.94

The values on the training set are greater than on the test set. The reason that this has occurred is that the classifier has learned the training sample specifics, i.e. it has *overfitted*.

As a result, the quality metrics are larger on the training set, but since these specifics are *not* properties of the classes, they are not observed in the test sample - thus the classifier makes more mistakes on the test set, and the quality metrics are smaller on that set.

As a general rule: the larger differences between the metrics on the train and test set, the greater the extent of model overfitting.

Let's consider train/test splits in general. We would like to divide our original dataset (written as (X, y) where X is our design matrix and y is our tuple of predictions) into a train set (used to fit and tune a model) and a test set (used to measure the quality of the model).



We can then tune *hyperparameters* in our model, for example coefficients in some regularisation method, refit the model on the training and measure its quality on the test. Overall our aim is to increase quality on the test.

How do we come up with a strategy for dividing data between a train and test set? In general we do the following:

- Data is divided between the train and test *randomly* to ensure distributions on the train and test are similar.
- Usually we take the train size greater than or equal to the test size. Typical proportions include $1:1$ or $2:1$.
- The larger the training size, the better the model will fit. However, if the test size is larger, our quality measurements on the test set will be more accurate.
- The test size can also be *estimated* based on a desired acceptable uncertainty of the quality metrics.

In general, a large part of the data is not used for the model fit. This might be crucial for small dataset.

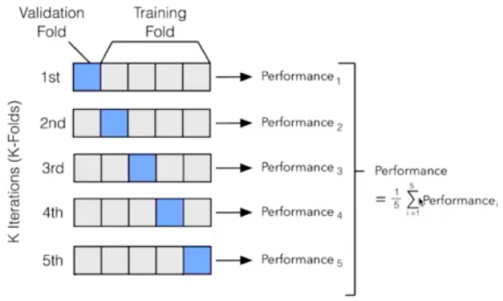
Even when we have a train/test split, if we turn our model too long using the same *test* sample, we can find values of the model's hyperparameters such that we *overfit* on the *test* set! To prevent this, we can split our data into *train*, *validation* and test sets.

As before, the train set is used to fit and tune the model. The *validation* set is used to measure the quality of the model during its tuning when we are searching for the best hyperparameter values. The *test* set is used only for the final quality measurement (after tuning). This quality is considered the quality of the model.

7.2 Cross-validation methods

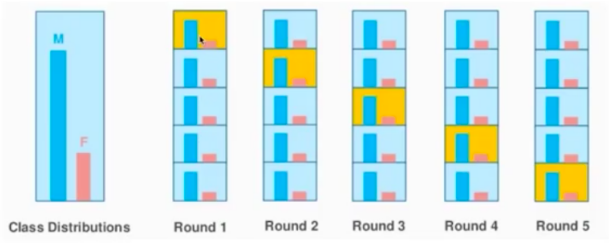
We will now discuss various methods of *cross-validation*. The first we consider is k-*folding*.

Definition 7.2: In the method of k-*folding*, the dataset is split into k parts, or k 'folds'. One fold is used to measure quality of the model (as a *validation set*), whilst the other $k-1$ folds are used to train the model. This procedure is repeated k times, and the average quality metric on the validation sets is computed to estimate the final quality; we can also collect the standard deviation of the quality metrics (or different confidence levels for example).

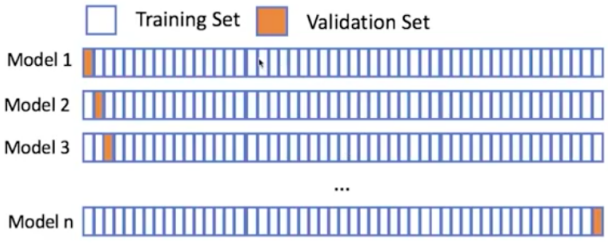


The main disadvantage of this procedure is that it is very time-consuming - it requires fitting the model k-times. Typical numbers of folds are around 5-10.

A useful modification is *stratified* k-*folding*. This works exactly the same as normal k-folding, but when the folds are created, the *same* class balance is used across all folds. This method is useful when we have large class imbalances and small datasets.



An extreme case of k-folding is *leave one out* cross-validation. This is k-folding where the number of folds is equal to the size of the dataset, and can be useful for very small sample sizes.



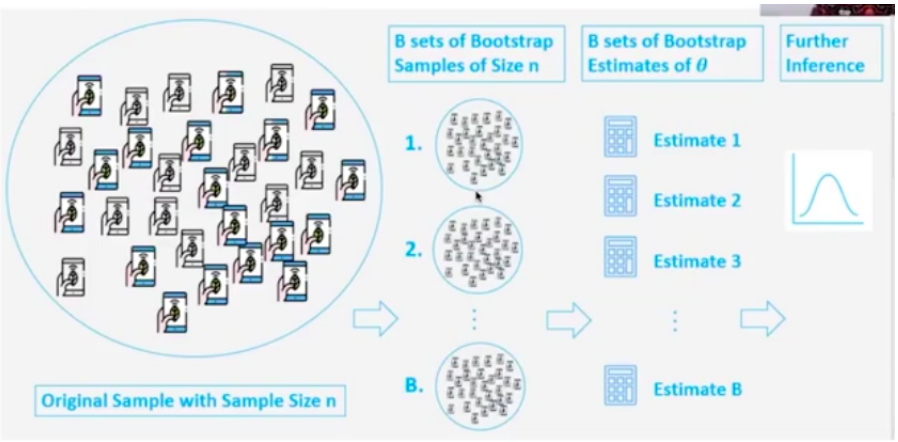
In detail, consider a sample with N objects. For N iterations, remove the ith object from the sample, and fit the model using the remaining $N-1$ objects. Using the model, make a prediction for the ith object and save it. Collect all predictions and measure the quality of the model.

Note that the models from each iteration will be approximately the same (since we hardly change the dataset). Therefore, the predictions will demonstrate a very small variance.

Yet another similar approach is *leave p out* cross-validation. Here, we simply remove p randomly selected objects instead of one, for each iteration. As a result, fewer iterations are required.

The *bootstrap method* is described as follows:

- Consider a sample of size n as shown in the figure below. We generate B sets of 'bootstrap samples' also of size n; these are generated by randomly sampling objects from the original sample *with replacement*. Thus an object can appear in a bootstrap sample several times.
- For each of the B bootstrap samples, a target is estimated (e.g. if we wanted to estimate the mean of the whole sample, we could compute the mean on each of the B boostrap sets).
- Compile a list of the B target estimates. Make further inference from these values, e.g. compute their mean, standard deviation, or even plot their distribution.



The bootstrap method can be used for a cross-validation method called *out-of-bag bootstrap*. The method is as follows:

- Given a dataset of size n, we create B random bootstrap samples of size n by sampling *with replacement* as usual. These samples become *training sets*.
- For each of the B bootstrap samples, determine which elements of the original dataset are not present in the particular sample. These objects comprise *test sets* for each of the corresponding training sets; we can compute required quality metrics on these sets.

Once we are done, we can estimate the mean and standard deviation of the quality metrics.

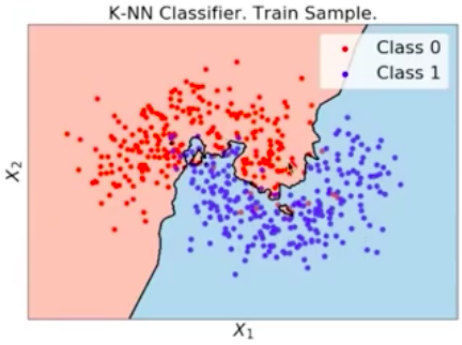
7.3 Quality metric uncertainty estimation

Our goal in this section is to estimate uncertainty of the quality metric on the test set. Again, we can use the bootstrap method:

- Suppose we are given a model fitted on a training set. Suppose that there are N objects in a corresponding test set.
- For $i = 1, ..., B$ bootstrap sample (typically of the order of hundreds or thousands):
  - Sample *with replacement* a subsample with N objects from the test sample.
  - Calculate quality metrics on this subsample.
- Estimate statistics of this metrics, for example *mean*, *variance*, *confidence intervals*.

The main advantage of this approach is that it is completely universal - it can be used for any model and any metric. It also does not required fitting the model times.

Example 7.3: In the binary classification example before, with model:



applying the bootstrap method gives the following uncertainties:

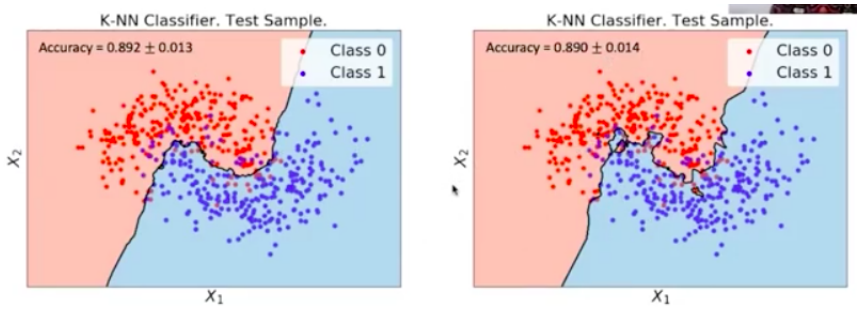
Metric	Train	Test	Bootstrap
Accuracy	0.93	0.89	0.89 ± 0.01
Precision	0.92	0.90	0.90 ± 0.02
Recall	0.95	0.88	0.88 ± 0.02
F_1	0.93	0.89	0.89 ± 0.02
ROC AUC	0.99	0.94	0.94 ± 0.01

The means are the same as those computed on the test set, but now we have uncertainties on these values.

We can also use cross-validation methods as previously described to estimate the quality metric uncertainties. For example, k-folding or out of bag bootstrap can be applied in similar ways - however, these require fitting a model several times.

7.4 Statistical model comparison

Consider two different models fitted on the same training sample:



How do we know which model is statistically better? Visually, the model in the left figure has more of a regular separation surface. Additionally, the mean accuracy of the model on the left is higher, but it has a lower uncertainty on accuracy than the model on the right hand side.

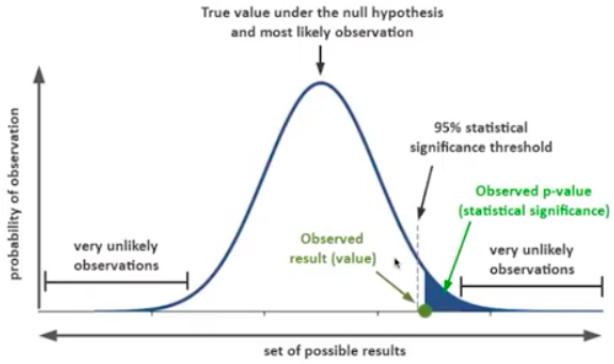
To distinguish between the models, we use *hypothesis testing*. We have two hypotheses in this case:

- H_0: the models have the same quality (the *null hypothesis*).
- H_1: the models have different qualities.

To test whether to reject the null hypothesis H_0 or not, we do the following:

- Select a test statistic, and calculate its value under the null hypothesis.
- Estimate the p-value for the observed statistic value.
- Compare the p-value with the given *significance level*.
- Based on a decision rule, reject H_0 and accept H_1 *or* do not reject H_0.

These steps are illustrated in the following image:



Examples of tests and statistics are described as follows. First, consider the *resampled paired t-test*:

- Given two models A and B, for $i = 1, ..., k$:
  - Randomly split the data into training and test samples.
  - Fit the models A and B on the training sample.
  - Compute quality metrics q_{Ai} and q_{Bi} for the models on the test sample.
- Compute the following t-statistic with $k-1$ degrees of freedom under the null hypothesis that the models have equal quality:

$$t = \frac{\overline{q}\sqrt{k}}{\sqrt{\sum_{i=1}^{k}(q_i - \overline{q})^2/(k-1)}},$$

  where $q_i = q_{Ai} - q_{Bi}$ and $\overline{q} = \sum_{i=1}^{k} q_i$.
- Given the estimate t-statistic value, we compute the p-value for this statistic using *Student's distribution* for $k-1$ degrees of freedom. We compare this p-value with the chosen *significance level* (e.g. $\alpha = 0.05$).
- If the p-value is less than α, the null hypothesis is rejected. This means that there is a significant difference between the models.

Example 7.4: For the two models at the start of this section, we can compute the t-statistic to be $t = 1.930$. The p-value is then given by 0.063. This is larger than 0.05, so the null hypothesis is not rejected. The models are *not* significantly different in terms of accuracy.

Another test is *Dietterich's 5×2-fold CV paired test*:

- Given two models A and B, for $i = 1, ..., 5$:
  - Randomly split the data into training and test samples. Fit the models A and B on the training sample. Compute quality metrics $q_{Ai}^{(1)}$ and $q_{Bi}^{(1)}$ for the models on the test sample.
  - *Rotate* the training and test samples: fit the models on the test, compute the metrics $q_{Ai}^{(2)}$ and $q_{Bi}^{(2)}$ on the training sample.
  - Compute the following values:

$$q_i^{(1)} = q_{Ai}^{(1)} - q_{Bi}^{(1)}, \qquad q_i^{(2)} = q_{Ai}^{(2)} - q_{Bi}^{(2)}, \qquad \overline{q}_i = \frac{q_i^{(1)} + q_i^{(2)}}{2}, \qquad s_i^2 = (q_i^{(1)} - \overline{q}_i)^2 + (q_i^{(2)} - \overline{q}_i)^2.$$

- Then calculate the following t-statistic with 5 degrees of freedom and the null hypothesis that the models A and B have equal qualities:

$$t = \frac{q_1^{(1)}}{\sqrt{\frac{1}{5}\sum_{i=1}^{5} s_i^2}}.$$

- Compute the p-value for this statistic using Student's distribution for 5 degrees of freedom. Compare the p-value with the chosen significance level α. If the p-value is less than α, the null hypothesis is rejected and there is a significant difference between the two models.

Example 7.5: Applying this test to the above example, we find that $t = 2.073$ and the p-value is 0.093. We see that the null hypothesis is *not* rejected, and again we conclude that the models are not significantly different in terms of accuracy.

There are many other available tests, including:

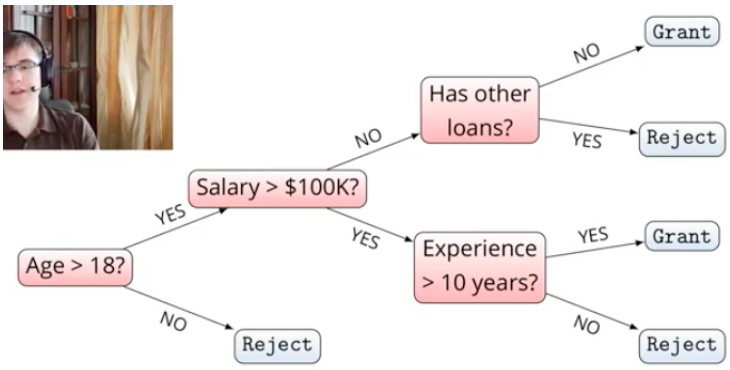
- Alpaydin's combined 5×2 CV F-test.
- Cochran's Q-test.
- Friedman's test.

However, a detailed discussion is beyond the scope of these lectures.

8 Decision trees

8.1 Decision tree training

Decision trees are a machine learning algorithm for classification and regression. Decision trees themselves predate machine learning, and can are used to make a decision e.g.



In our case, we would like to build a decision tree for example for classification problems, to decide which class a given element belongs to (hence a decision tree determines a decision boundary).

In general, building an optimal decision tree is an NP-complete problem. Therefore, in practice we use a *greedy algorithm* to build a decision tree - on each iteration, we partition the data as if the split was final (we don't think ahead at all). There are two key components:

- A split criterion to compare different ways to partition the data.
- A stopping condition to decide when to stop building the tree.

Formally:

Definition 8.1: We define *greedy decision tree learning* as follows. Suppose we are given an input training set $D = \{(x_i, y_i) : i = 1, ..., N\}$. For each node in the tree u, we write $R_u \subseteq D$ as the data subset associated with this node. To build each R_u we do the following:

(1) Set the first node in the tree $R_{\text{start}} = D$ to be equal to the whole dataset.

(2) Given R_u left unsplit and not a leaf, we greedily split R_u into R_l and R_r given by:

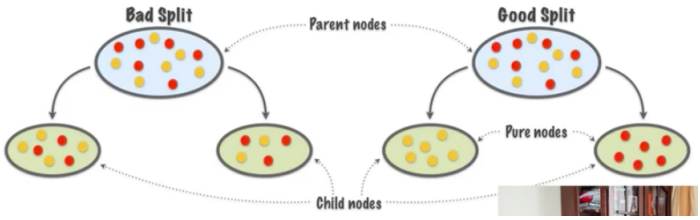
$$R_l(j,t) = \{\mathbf{x} \in R_u : x^j \leq t\}, \qquad R_r(j,t) = \{\mathbf{x} \in R_u : x^j > t\},$$

where j is a particular feature, optimising a given loss $Q(R_u, j, t)$ over (j, t).

(3) If a stopping criterion is satisfied for u, declare it a *leaf* of the true.

(4) If not, create an internal node u corresponding to the predicate $x^j \leq t$ and repeat from Step 2 for $R_u = R_l(j,t)$ and $R_u = R_r(j,t)$.

A good split at any point in the building of a decision tree is one that maximises *purity*; i.e. objects in the split are as uniform as possible:



We can formalise the purity by carefully discussing the loss function $Q(D, j, t)$ above. Recall that R_u is the subset of D corresponding to the node u. With the current split, let $R_l \subseteq R_u$ go left and $R_r \subseteq R_u$ go right. We choose a predicate to maximise:

$$Q(R_u, j, t) = H(R_u) - \frac{|R_l|}{|R_u|} H(R_l) - \frac{|R_r|}{|R_u|} H(R_r),$$

where $H(R)$ is some impurity criterion. Generally, we take:

$$H(R) = \min_{c \in \mathcal{Y}} |R|^{-1} \sum_{(x_i, y_i) \in R} \mathcal{L}(y_i, c),$$

where $\mathcal{L}$ is a loss function, penalising wrong predictions.

Example 8.2: Some examples of impurity criterion include:

- For regression, we might choose MSE:

  $$H(R) = \min_{c \in \mathcal{Y}} |R|^{-1} \sum_{(x_i, y_i) \in R} (y_i - c)^2.$$

  This is minimised by $c = |R|^{-1} \sum_{(x_j, y_j) \in R} y_j$, which makes the impurity the variance of the target.

- For classification, we can enumerate $\mathcal{Y} = \{y_1, ..., y_N\}$. We then let:

  $$p_k = |R|^{-1} \sum_{(x_i, y_y) \in R} 1_{\{y_i = y_k\}}$$

  be the share of examples belonging to the kth class.

  We could choose the impurity to be the *miss rate* (inaccuracy):

  $$H(R) = \min_{c \in \mathcal{Y}} |R|^{-1} \sum_{(x_i, y_i) \in R} 1_{\{y_i \neq c\}}.$$

  We could also choose the impurity to be the *Gini index*:

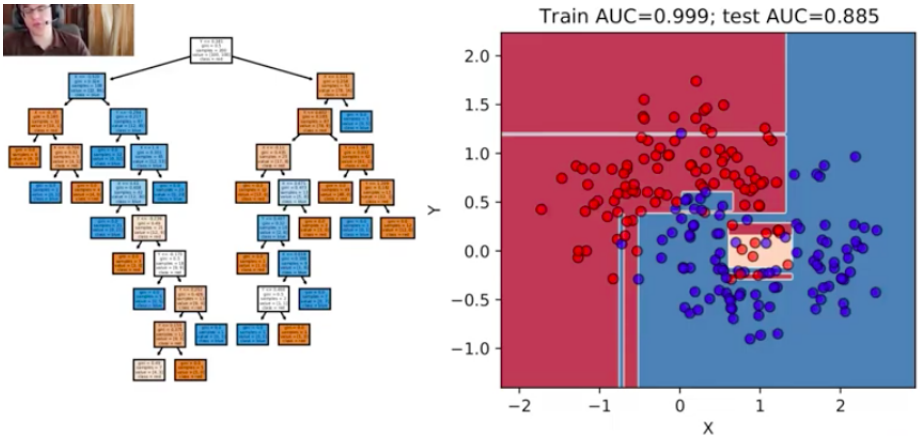
  $$H(R) = \sum_{k=1}^{N} p_k (1 - p_k).$$

  Finally, we could choose the impurity to be the *cross-entropy*:

  $$H(R) = -\sum_{k=1}^{N} p_k \log(p_k).$$

8.2 Stopping rules for decision tree learning

What will happen if we build a decision tree to minimise the impurity to the greatest possible extent? The tree will be built until all leaves are 100% pure, with some leaves only containing a single example. This corresponds to a *dramatic overfitting*. For example:

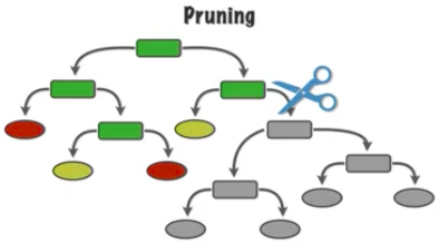


Clearly we need to know when to stop the training to avoid overfitting. There are multiple choices available:

- Impose a maximum tree depth.
- Impose a minimum number of objects in a leaf.
- Impose a maximum number of leaves in a tree.
- Stop if all objects fall into the same leaf.
- Constrain *quality improvement* (stop when improvement gains drop below a defined threshold).

There is no general way to select the best choice. It is typically decided by exhaustive search and cross-validation.

A useful technique that can help to avoid overfitting is *decision tree pruning*.



We train a very large tree (effectively overfitting the training set), then remove the least important nodes. This often leads to better result than simply training a small tree.

The reason why is as follows. When we build a tree, we use a greedy algorithm - we never look ahead or plan smart moves. In particular, pruning allows us to recover some of this missing foresight.

9 Ensemble methods

Ensemble methods can be used to improve model performance.

9.1 Bagging and random forests

The root of all evil in machine learning is the finite amount of data. When a learning algorithm trains a model, it either trusts the data too much and overfits, or trusts the data too little and underfits.

The idea of *bagging* is to fight this problem by having many version of the model trained on different subsets of the dataset (each overfitting), hoping that the biases cancel each other out.

The idea of bagging rests on the bootstrap procedure:

Definition 9.1: The *bootstrap procedure* is defined as follows. Given a dataset $D = \{(x_i, y_i)\}$, we generate new samples X_j^* of (x_i, y_i) by drawing from D uniformly at random *with replacement* (therefore it is possible that we get copies of a single (x_i, y_i) in the sample).

The *bagging method* (also called *bootstrap aggregating*) proceeds as follows:

(1) Given a dataset D, generate N bootstrapped samples $X_1^*, ..., X_n^*$.

(2) Train models $h_1, ..., h_n$ to each of the bootstrapped samples.

(3) Average the predictions to obtain an 'average model':

$$h(x) = \frac{1}{N} \sum_{j=1}^{N} h_j(x).$$

The *random forest algorithm* is a variant of bagging for decision trees. We reduce error by additionally *averaging over instances and features*.

Definition 9.2: Let $D = \{(\mathbf{x}_i, y_i)\}$ be a dataset, with $\mathbf{x}_i \in \mathcal{X} = \mathbb{R}^d$ and $y_i \in \mathcal{Y}$. The *random forest algorithm* is as follows:

(1) For $j = 1, ..., N$:

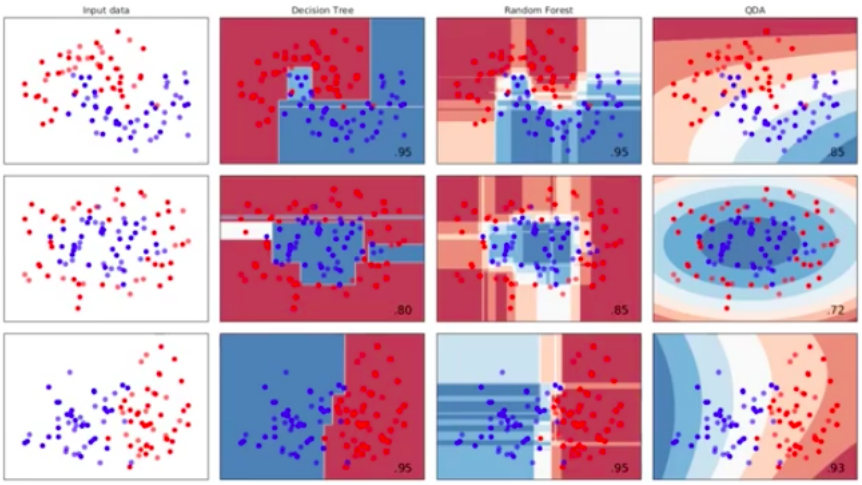
  (i) Pick p random features out of d.

  (ii) Bootstrap a sample $D_j = \{(\mathbf{x}_i, y_i)\}$ for $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i \in \mathcal{Y}$.

  (iii) Train a decision tree $h_j(\mathbf{x})$ using the bootstrapped dataset D_j.

(2) Proceed as normal - take the average of the resulting models as in the standard bagging procedure.

Below is a comparison of three input datasets, three decision trees trained to the datasets, and three random forests trained to the datasets (i.e. applying the random forest algorithm using decision trees). The use of a random forest nicely reduces overfitting.



We will now prove something about ensemble methods' bias and variance. Recall the bias-variance decomposition of mean square error we saw earlier in the course:

$$\text{MSE}(\mathbf{x}) = \underbrace{\mathbb{E}_y[(y - \mathbb{E}[y|x])^2]}_{\text{noise}} + \underbrace{(\mathbb{E}_D[f_D(x)] - \mathbb{E}[y|x])^2}_{\text{bias}} + \underbrace{\mathbb{E}_D\left[(f_D(x) - \mathbb{E}_D[f_D(x)])^2\right]}_{\text{variance}}$$

The error of a model is its noise (a constant that cannot be removed), the *bias* and the *variance*.

We first claim that bias is not made any worse by bagging:

Proposition 9.3: The bias of a model is equal to the bias of the model under bagging.

Proof: The bias of the ensemble is given by:

$$\mathbb{E}_y\left[\left(\mathbb{E}_D\left[\frac{1}{N}\sum_{n=1}^{N}\tilde{f}_D(x)\right] - \mathbb{E}[y|x]\right)^2\right] = \mathbb{E}_y\left[\left(\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_D[\tilde{f}_D(x)] - \mathbb{E}[y|x]\right)^2\right]$$
$$= \mathbb{E}_y\left[\left(\mathbb{E}_D[\tilde{f}_D(x)] - \mathbb{E}[y|x]\right)^2\right],$$

which is just the bias of the individual model. This holds since $\mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B]$ holds for any two random variables A, B regardless of their dependence. □

Variance is more complicated:

Proposition 9.4: The variance of a bagged model is N times lower for uncorrelated hypotheses, and is unchanged for fully-correlated hypotheses.

Proof: Define:

$$F = \frac{1}{N}\sum_{n=1}^{N} \tilde{f}_n(x).$$

The variance of the bagged model is then given by:

$$\mathrm{Var}(F) = \frac{1}{N^2}\sum_{i,j}\mathrm{Cov}(\tilde{f}_i, \tilde{f}_j) = \frac{1}{N^2}\sum_{i}\left[\mathrm{Var}(\tilde{f}_i) + \sum_{j\neq i}\mathrm{Cov}(\tilde{f}_i, \tilde{f}_j)\right].$$

All the models $\tilde{f}_i$ use the same algorithm, so we can assume that they all have the same variance, hence:

$$\mathrm{Var}(F) = \frac{1}{N}\mathrm{Var}(\tilde{f}) + \frac{1}{N^2}\sum_{i}\sum_{j\neq i}\mathrm{Cov}(\tilde{f}_i, \tilde{f}_j).$$

The result follows. □

Therefore, the practical gain of ensembling depends highly on whether the hypotheses are highly correlated or not. However, we won't make the variance any worse (we just might fail to make it better).

9.2 Stacked generalisation

The point of *stacking* is to correct errors made by a previous algorithm: the idea is to run a new algorithm B which is designed to correct the mistakes of algorithm A.

A naïve precursor to the idea of stacking is *blending*:

Definition 9.5: The *blending* procedure is defined as follows:

(1) Partition the training dataset D into D_1, D_2.

(2) Train models $\tilde{f}_i(x)$ on D_1.

(3) Compute predictions $Z_i = \tilde{f}_i(D_2)$.

(4) Train a *meta-model* $\phi(Z_1, ..., Z_n, D_2)$ on the predictions obtained on the previous step and features.

Unfortunately, there is a glaring issue with this approach. Both levels are trained on half of the dataset - this is unacceptable waste in the quest for, say, 1% performance gain!

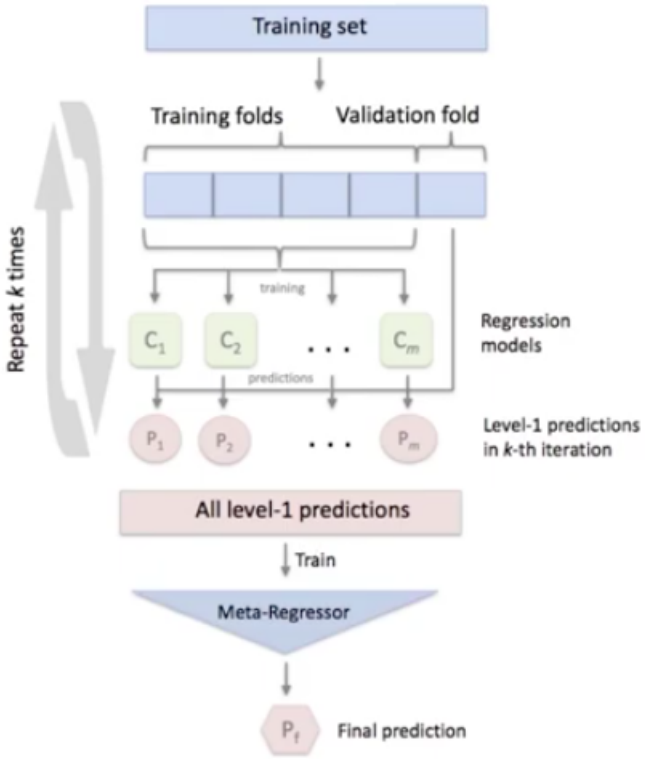
Fortunately, there is a better approach, based on k-folding - this is *stacking*:

Definition 9.6: The *stacking* procedure is defined as follows:

(1) Partition the training set into k folds.

(2) As in cross-validation, train k 'level 1' models leaving a fold out in each case, and make predictions for each of these 'level 1' models on the left-out fold.

(3) Fit the meta-model on all the level 1 predictions, optionally concatenated with features.



10 Gradient boosting

10.1 Boosting

The idea of *boosting* is to build a sequence of models, where each model corrects the error of the previous ones (similar to stacking as discussed in the previous chapter).

More mathematically, consider a regression problem:

$$\text{minimise} \qquad \frac{1}{2}\sum_{i=1}^{l}(h(x_i)-y_i)^2 \qquad \text{over } h.$$

We search for solutions in the form of a *weak learner composition*:

$$a_N(x)=\sum_{n=1}^{N}h_n(x),$$

with weak learners $h_n \in \mathbb{H}$ in some function space $\mathbb{H}$. The *boosting approach* add weak learners greedily:

(1) Start with the 'trivial' weak learner given by:

$$h_0(x)=\frac{1}{l}\sum_{i=1}^{l}y_i.$$

(2) At each step N, compute the residuals:

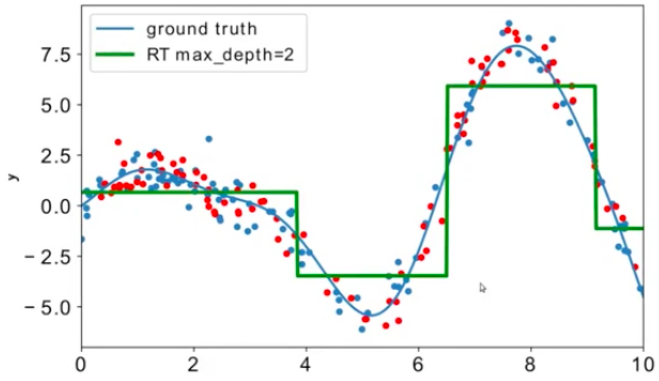
$$s_i^{(N)}=y_i-\sum_{n=1}^{N-1}h_n(x_i)=y_i-a_{N-1}(x_i),$$

for $i=1,...,l$.

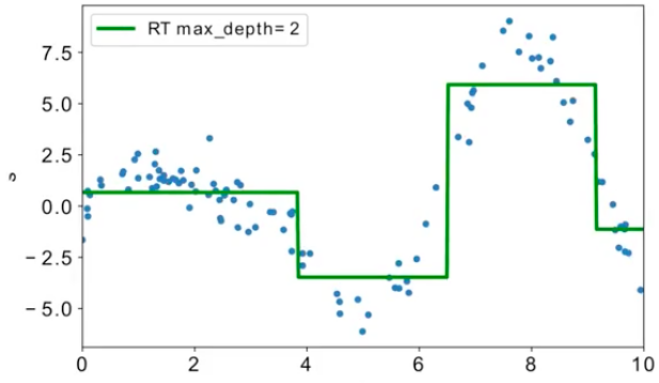
(3) Train the next weak algorithm using:

$$a_N(x):=\underset{h\in\mathbb{H}}{\operatorname{argmin}}\,\frac{1}{2}\sum_{i=1}^{l}(h(x_i)-s_i^{(N)})^2.$$

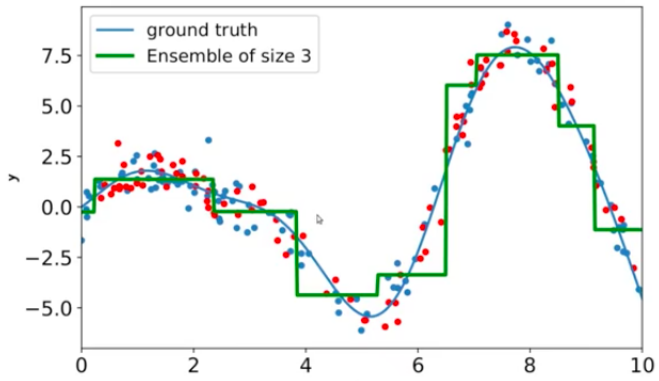
Example 10.1: Here is an example of boosting a decision tree:



Above, we have fitted a decision tree. Next, we compute the difference between the predictions of this tree and the data, then we fit a new tree to the differences:



We repeat this procedure until we arrive at a final boosted tree:



10.2 Gradient boosting

In the above form of boosting, we used mean squared error loss. In order to handle arbitrary losses, we need to define the residuals using *gradient descent*.

The scheme of *gradient boosting* is the following. With $a_{N-1}(x)$ already built, we wish to compute:

$$\text{minimise} \qquad \sum_{i=1}^{l} \mathcal{L}(y_i, a_{N-1}(x_i) + \gamma h_(x_i)) \qquad \text{over } \gamma, h.$$

We view the loss function $\mathcal{L}(y, z)$ as a function of $z = a_N(x_i)$ and execute gradient descent on z. We search for such $s_1, ..., s_l$ that minimise:

$$\sum_{i=1}^{l} \mathcal{L}(y_i, a_{N-1}(x) + s_i).$$

We choose:

$$s_i = -\frac{\partial \mathcal{L}(y, z)}{\partial z}\bigg|_{z=a_{N-1}(x_i)}$$

and approximate the s_i's by $h_N(x_i)$.

All in all then, we have the following. Our inputs include a *training set* $D = \{(x_i, y_i)\}_{i=1}^{l}$, a number of boosting iterations N, a loss function $\mathcal{L}(y, z)$ with gradient $\partial\mathcal{L}/\partial z$, a family $\mathbb{H} = \{h(x)\}$ of weak learners and their associated learning procedures, and additional hyperparameters for the weak learners (tree depth, etc). Gradient boosting then proceeds as follows:

(1) Initialise the gradient boosting machine using some simple rule (e.g. zero, most popular class, etc).

(2) Execute the boosting iteration $t = 1, ..., N$:

(i) Compute the *pseudo-residuals*:

$$s_i = -\frac{\partial \mathcal{L}(y_i, z)}{\partial z}\bigg|_{z=a_{N-1}(x_i)},$$

for $i = 1, ..., l$.

(ii) Train $h_N(x_i)$ by regressing onto $s_1, ..., s_l$:

$$h_N(x) = \operatorname*{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{l} (h(x_i) - s_i)^2.$$

(iii) Find the optimal γ_N using plain gradient descent:

$$\gamma_N = \operatorname*{argmin}_{\gamma \in \mathbb{R}} \sum_{i=1}^{l} \mathcal{L}(y_i, a_{N-1}(x) + \gamma h_N(x_i)).$$

(iv) Update the gradient boosting machine by $a_N(x_i) = a_{N-1}(x) + \gamma_N h_N(x)$.

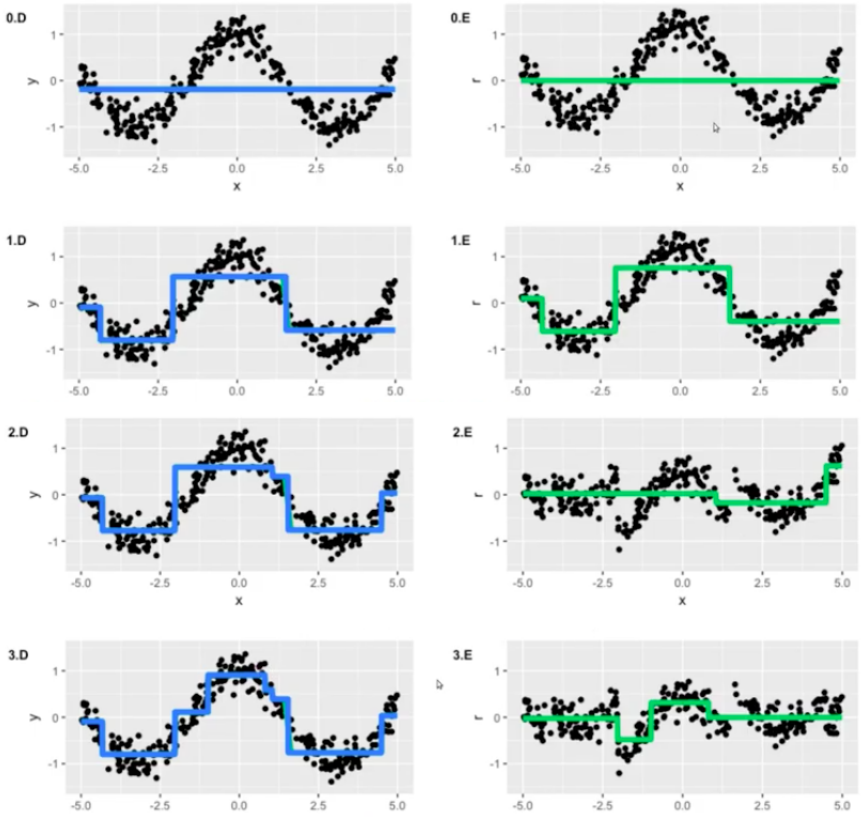
(3) Compose the final gradient boosting machine learner:

$$a_N(x) = \sum_{t=0}^{N} \gamma_i h_i(x).$$

Example 10.2: Let's consider an example of gradient boosting for a regression problem. Let $X^{300} = \{(x_i, y_i)\}_{i=1}^{300}$ be a training set with $x_i \in [-5, 5]$, $y_i = \cos(x_i) + \epsilon_i$ and $\epsilon_i \sim N(0, 1/5)$ normally distributed.

Pick $N = 3$ boosting iterations, quadratic loss $\mathcal{L}(y, z) = (y - z)^2$. The gradient of the quadratic loss is $\partial L(y, z)/\partial z = y - z$, i.e. the residuals. Choose our weak learners as decision trees $h_i(x)$, and set 2 as the maximum depth for decision trees.

On the left are the values of the model, and on the right are the values of the model at each iteration:



10.3 Regularisation of gradient boosting machines via shrinkage

Like all machine learning models, gradient boosting machines can overfit. We need to regularise them to make them practical.

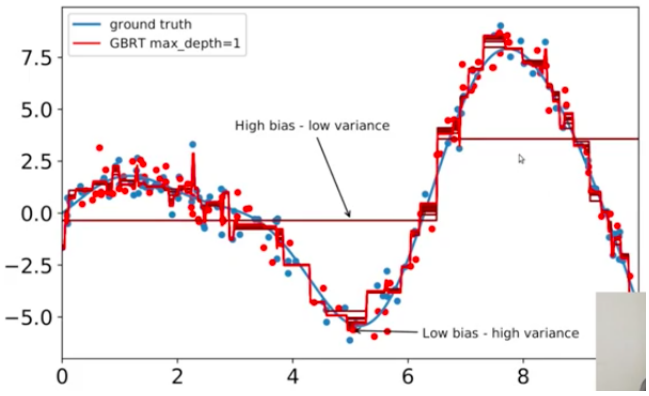
For *too simple weak learners* the negative gradient is approximated badly - thus we have a random walk in the space of samples. For *too complex weak learners* a few boosting steps may be enough for overfitting.

One solution is to use *shrinkage*: we make the steps shorter using a *learning rate* $\eta \in (0, 1]$:

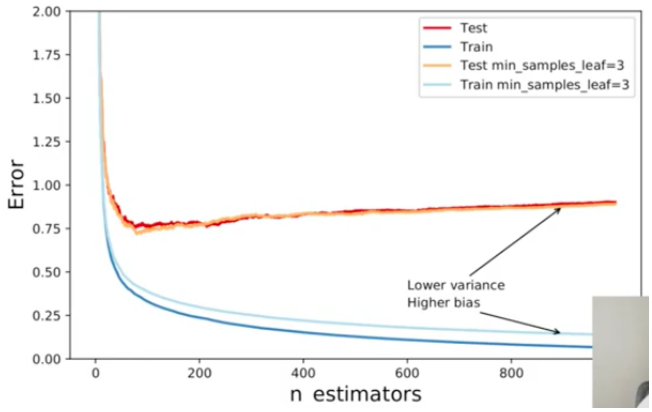
$$a_N(x) = a_{N-1}(x) + \eta\gamma_N h_N(x).$$

Effectively, we distrust the gradient direction estimated via the weak learners.

Example 10.3: Consider the following regression model, fitted with too many trees (light red) and too few trees (dark red).

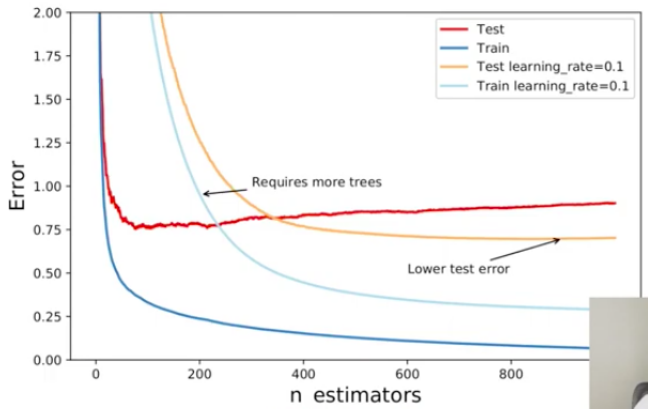


As always, we want to estimate the sweet spot where the test error is the lowest:



If we perform some regularisation by simplifying the decision trees as above, then we find that the train error increases a little bit (the model is less able to model the training dataset), and the test error slightly decreases.

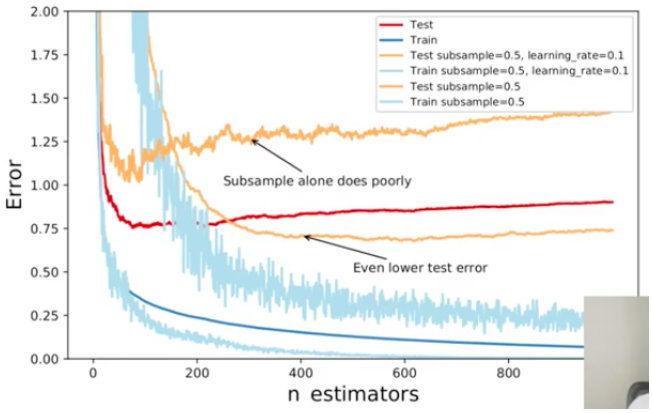
Consider instead reducing the learning rate:



We get much better quality predictions on the test data. However this does require more trees.

In principle, you can do this more or less indefinitely - divide the learning rate by 10, multiply the number of trees by 10, etc, but eventually we get diminishing returns.

Now consider combining two of our previous ideas. In the figure below, we compare with subsampling. We combine the idea we had with random forests, where the next tree was not fitted on the whole dataset but on a subsample instead, together with gradient boosting.



Changing the subsampling without the learning rate makes thing even worse - quality decreases. However, with a change in learning rate, we can achieve super performance.

10.4 XGBoost algorithm

XGBoost stands for *extreme gradient boosting*. This adds some features to the standard algorithm:

- The gradient descent direction using *second order derivatives* rather than just single derivatives; we minimise over h:

$$\sum_{i=1}^{l}\left(-s_i h(x_i)+\frac{1}{2}t_i h^2(x_i)\right) \qquad t_i=\frac{\partial^2}{\partial z^2}\mathcal{L}(y_I,z)\bigg|_{a_{N-1}(x_i)}.$$

- We penalise *large leaf counts* J and *large leaf coefficient norms* $||\mathbf{b}||_2^2=\sum_{j=1}^{J}b_j^2$; we minimise over h:

$$\sum_{i=1}^{l}\left(-s_i h(x_i)+\frac{1}{2}t_i h^2(x_i)\right)+\gamma J+\frac{\lambda}{2}\sum_{j=1}^{J}b_j^2,$$

  where $b(x)=\sum_{j=1}^{J}b_j[x\in R_j]$.

- We choose splits $[x_j<t]$ at the node R to maximise:

$$Q=H(R)-H(R_l)-H(R_r),$$

  where the impurity criterion is given by:

$$H(R)=-\frac{1}{2}\left(\sum_{(t_i,s_i)\in R}s_j\right)^2\Bigg/\left(\sum_{(t_i,s_i)\in R}t_j+\lambda\right)+\gamma.$$

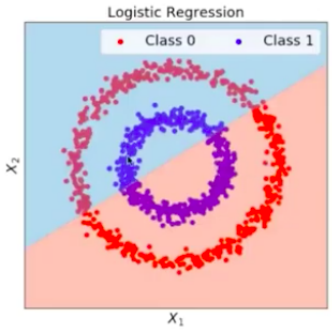
- We add the stopping criterion: declare the node a leaf if even the best split gives negative Q.

Part II
Introduction to neural networks

11 Feature engineering, importance and selection

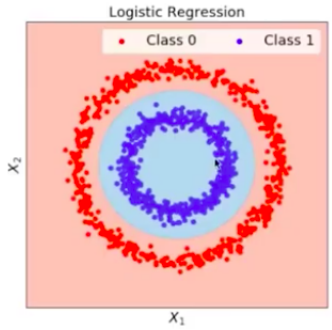
11.1 Feature engineering

Example 11.1: Consider a binary classification problem in 2D with a logistic regression classifier:



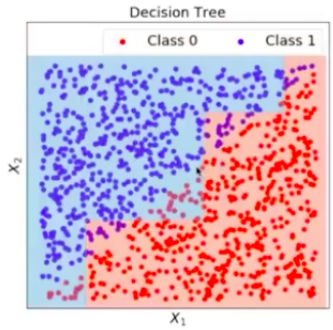
The classes are concentric circles, and the classifier *cannot* separate them using only the input features X_1, X_2 - they are not linearly separable.

Let's create the new feature $X_3 = X_1^2 + X_2^2$. This feature helps to separate the circles by a straight line. Now, logistic regression can solve the classification problem ideally using all three features X_1, X_2 and X_3.



The above example shows how 'feature engineering' can improve the predictions of linear models.

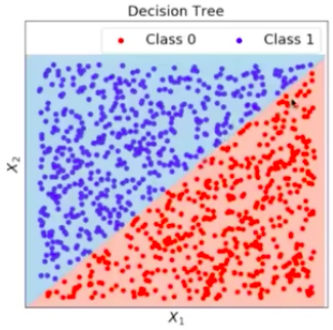
Example 11.2: Consider another example where the two classes are separated by the surface $X_2 - X_1 = 0$.



These classes are linearly separable, and can be found using a logistic regression model.

However, the problem is difficult to solve with a decision tree classifier (as shown above - it splits input feature space with *rectangles*). It requires larger depth of the tree to separate the classes properly.

If we introduce the new feature $X_3 = X_2 - X_1$, we help the classifier. Now the classes can be separated using just one predicate $X_3 > 0$. It requires a decision tree with depth 1 to solve the problem ideally.



These two examples show that creating new features can:

- Improve the quality of a model.
- Reduce the complexity of a model (see the shorter decision tree in the second example).
- Speed up model training (due to reduced complexity).
- Reduce dimensionality of a problem by removing less informative features (e.g. X_1, X_2 in previous examples - these can be removed without losing the model's qualtiy!).

But how do we choose which new features will be useful? There are some general principles:

- Use any information about the problem we have. In the above examples the information we had was the shape of the classes and the border between them.
- Create features with physical meaning (e.g. $\sqrt{X_1^2 + X_2^2}$ is radius). For example in high energy physics, we might like to use momenta, mass, angles, etc.
- Remove limitations of a model (like with the decision tree in example 2).

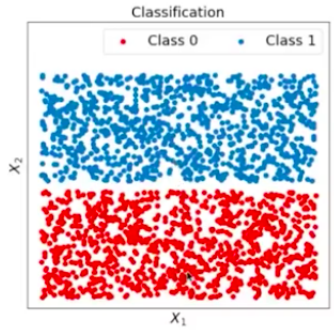
Given an initial set of input features $X_1, ..., X_n$, typical feature combinations include:

- Powers and products, e.g. X_i^p, $X_1 X_2$.
- Sums and differences of squares, e.g. $X_1^2 \pm X_2^2$.
- Sums and differences, e.g. $X_1 \pm X_2$.
- Sum and difference quotients, e.g.
$$\frac{X_1 \pm X_2}{X_1 \mp X_2}$$
- More complicated functions, e.g.
$$\sin(X_1), \qquad \cos(X_1).$$

11.2 Feature importance

Let us try to address the problem of which features are important more carefully.

A dataset can have a lot of features, but not all features are equally useful for a given regression or classification task. Some of them are more informative than others. Consider the example below:



We see that the feature X_1 is completely uninformative for the classification problem, and can be skipped. We can easily spot this from the plot of the data, but how can we do this in multi-dimensional cases? Our goal should be to measure the importance of each feature in the general case somehow.

There are several different methods to achieve this:

- Correlations between features and target values.
- Probabilistic distances.
- Decision tree-based feature importance.
- Linear model-based feature importances.

Finally we will consider a completely general algorithm which can be applied for any classification or regression problem.

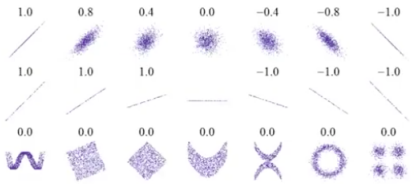
We begin with correlations:

Definition 11.3: Let f be a feature and let y be a target. Denote by f_i the value of the feature for the ith object in a dataset, and by y_i the value of the target for the ith object in a dataset (this corresponds to labels 0 or 1 for example in a binary classification problem, or to the target values in a regression problem). We define the *Pearson correlation coefficient* by:

$$\rho(f, y) = \frac{\sum_i (f_i - \overline{f})(y_i - \overline{y})}{\sqrt{\sum_i (f_i - \overline{f})^2 \sum_i (y_i - \overline{y})^2}},$$

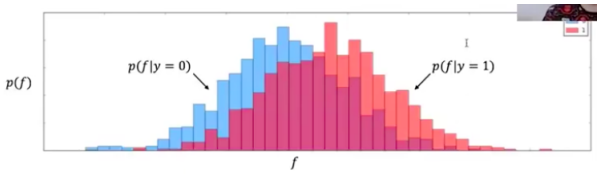
where $\overline{f}$ is the mean of the feature over the dataset, and $\overline{y}$ is the mean of the target values over the dataset.

This measurement of feature importance is very easy to compute, but captures only linear dependencies in dataset:



Another simple approach is based on *probabilistic distance*:

Definition 11.4: Let f be a feature, and let y be a target taking values 0 or 1 (i.e. a binary classifier).

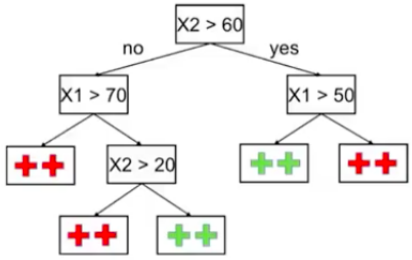


We define the *total variation* between the probability distributions $\mathbb{P}(f|y=0)$ and $\mathbb{P}(f|y=1)$ to be:

$$\mathrm{Imp}(f) = \int |\mathbb{P}(f|y=1) - \mathbb{P}(f|y=0)| \, df.$$

There are other methods of measuring the distance between probability distributions - we will consider these later in the school.

Another approach is based on *decision trees*. Recall from the previous lectures that a decision tree is a binary tree where each node t has two children. Each node uses one feature to make a split into the two children notes. For each node, we compute the number of objects n_t which satisfy the criteria in the node; then for each node we can compute *impurity* functions $I(t)$ (e.g. Gini, cross-entropy, MSE) for the node.



Definition 11.5: Let $T(f)$ be the set of all nodes which use feature f to make a split. Then feature importance can be estimated by:

$$\mathrm{Imp}(f) = \sum_{t \in T(f)} n_t \Delta I(t),$$

where the *information* $\Delta I(t)$ contained in the node t is defined by:

$$\Delta I(t) = I(t) - \sum_{c \in \mathrm{children}(t)} \frac{n_c}{n_t} I(c).$$

This statistic is one of the most popular and important for feature estimation. It takes into account dependencies between different input features, and can capture complicated dependencies between features and the target. It is also implemented in almost all decision tree libraries.

Another method is *linear-model based*.

Definition 11.6: Consider a linear model with regularisation (e.g. an $L1$ or $L2$ penalty). The model makes predictions:

$$\hat{y} = w_0 + w_1 f_1 + w_2 f_2 + \cdots + w_k f_k.$$

If features are normalised (have the same ranges, e.g. by scaling), the feature importance of f_i is defined to be equal to:

$$\mathrm{Imp}(f_i) = |w_i|.$$

This method is very simple, but has the same limitations as Pearson correlation - it cannot capture complicated dependencies.

Now let's consider a completely general approach. Suppose we have a model used for classification or regression.

- We train the model to some training set.
- We calculate the quality measure Q_0 on some validation set.
- For a given feature f, we:
  - Replace the given values of this feature with random values from the same distribution (for example, you can perform random shuffling of the value of the feature).

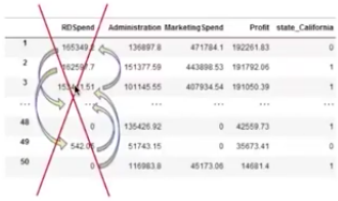
  - Compute the new quality measure Q_f on the validation set, and estimate the feature importance as the difference between the initial and current quality metric values:

$$\mathrm{Imp}(f) = Q_0 - Q_f.$$

This is the most powerful method for computing feature importance. It can capture complicated dependencies, and works for any model - e.g. neural networks, which we shall see soon.

There is a slight modification to this method which also works:

- We train the model on the *full set of features* (as normal)
- We calculate the quality measure Q_0 on some validation set.
- For a given feature f, we:
  - Remove the feature f from the dataset, then *refit the model without this feature*.



  - Compute the new quality measure Q_f on the validation set, and estimate the feature importance as the difference between the initial and current quality metric values:

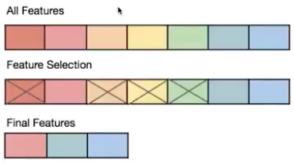
$$\mathrm{Imp}(f) = Q_0 - Q_f.$$

The advantage of this approach is that it estimates the feature importances more precisely, and works better for highly correlated input features. However, it requires refitting the model several times - this might be difficult for large models and datasets.

There are many other approaches to estimate feature importance which we don't have time to discuss here - for example the method of *Shapley values*, which has a Python implementation.

11.3 Feature selection

The goal of *feature selection* is to reduce the number of features with minimal loss of model quality. For example, we might want to keep the best K of D features, or we might want to remove as many features as possible but keeping the quality $Q \geq Q_{\text{min}}$ for some Q_{min}.



Three of the most popular methods are the *filter method*, *embedded methods* and *recursive feature elimination*.

Definition 11.7: The *filter method* involves estimating the importance of individual features $\text{Imp}(f_1), \text{Imp}(f_2), ..., \text{Imp}(f_D)$ via some method (e.g. correlation, probability distance). We then simply select the required number of features with the highest importances.

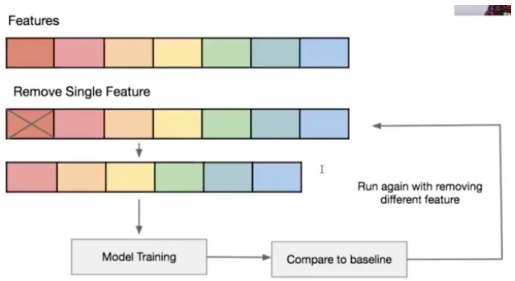
This is simple it implement and is quite fast. However, it is bad for highly correlated features, since it may take many redundant features.

Definition 11.8: An *embedded method* selects features based on the feature importance of a model.

- In a linear model, an embedded method selects the best features using the weights of the model (as above).
- For a decision tree, the best features are selected using their decision tree importance (as above).

Embedded methods is widely used and take into account correlation between features.

Finally, we have *recursive feature elimination*. Uninformative features are removed one by one, based on deleting the least important feature at each step.



The steps are as follows:

- Train a model on the full set of features, and estimate the importance of each feature based on the model.
- Remove the least importance features (or several features).
- Repeat until we have the desired number of features remaining.

In combination with the general method for feature importance estimation, this is one of the most powerful methods for feature selection. It is applicable to any regression or classification problem.

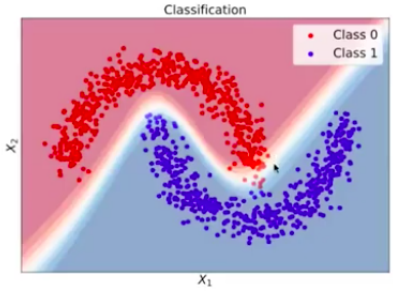
12 Clustering

So far, we have studied classification and regression models. These are models of 'supervised learning' where we know exactly what to predict - class labels for classification or real labels for regression.

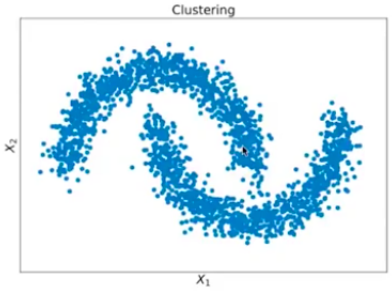
In this section, we will discuss *clustering*, where we do not have any target values for predictions. Therefore clustering is a topic of 'unsupervised machine learning'.

12.1 Clustering vs classification

Consider a binary classification problem, where we have object features X and class labels $y \in \{0, 1\}$. A classifier learns a decision rule f, so that $f(X) \approx y$. The trained classifier predicts class labels for the new objects.



In clustering, we don't have class labels y. The goal is to divide all objects into separate groups using only object features X - these groups are called *clusters*.



We would like to cluster the objects such that *objects inside clusters are similar*, and *objects from different clusters are dissimilar*. In the above figure, 'similarity' can be defined as proximity to the other objects.

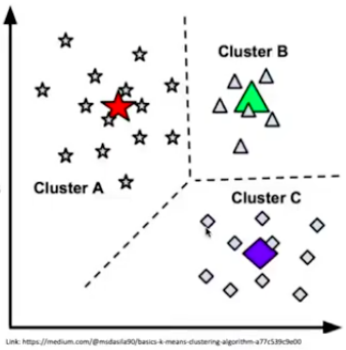
Most clustering algorithms are based on the following assumptions:

- Objects form *dense* clusters. This means that the object density inside one cluster is larger than between the clusters.
- Objects from one cluster are somehow similar, whilst objects from different clusters are somehow dissimilar. Often similarity between objects is based on 'distance' between them.
- Distances between neighbours within one cluster are smaller than between objects from different clusters.

12.2 K-means

The most popular clustering approach is the approach of *K-means*. We develop intuition for this approach as follows.

We suppose that each cluster is represented by a *centre*. All objects are assigned to a cluster according to the closest centre. The goal of the problem is therefore to *find centres that form the most compact clusters*.



Let's define some notation which we will use throughout this section. Consider a sample with N objects, written $\{x_n\}_{n=1}^N$ (where x_n denotes a tuple of features). We will search for K clusters with centres $\{\mu_1, ..., \mu_K\}$. The criterion to find the best centres is minimising the *within-cluster distance* over $\mu_1, ..., \mu_K$:

$$Q = \sum_{n=1}^{N} \min_k \rho(x_n, \mu_k).$$

Each object is then assigned to a cluster $z_n \in \{1, 2, ..., K\}$ via:

$$z_n = \operatorname*{argmin}_k \rho(x_n, \mu_k).$$

A general algorithm that achieves this is as follows (called the *K-centroid algorithm*):

(1) Initialise the centres $\mu_1, ..., \mu_K$ by randomly selecting k objects from the sample.

(2) Assign each object of the sample to the nearest centre:

$$z_n = \operatorname*{argmin}_k \rho(x_n, \mu_k),$$

where ρ is the 'distance' between two objects.

(3) Update the centres by minimising the within cluster distance, i.e. determining:

$$\mu_k = \operatorname*{argmin}_\mu \sum_{n:z_n=k} \rho(x_n, \mu).$$

(4) If the algorithm has not converged to within some tolerance, go to Step 2 and repeat. Otherwise return $z_1, ..., z_N$.

The above algorithm can use difference distances ρ:

Definition 12.1: If $\rho(x_n, \mu_k) = ||x_n - \mu_k||_2^2$, the above algorithm is called the *K-means algorithm*. If $\rho(x_n, \mu_k) = ||x_n - \mu_k||_1$, the above algorithm is called the *K-medians algorithm*.

The K-means algorithm can be implemented efficiently as follows:

(1) Initialise the centres $\mu_1, ..., \mu_K$ by randomly selecting k objects from the sample.

(2) Assign each object of the sample to the nearest centre:

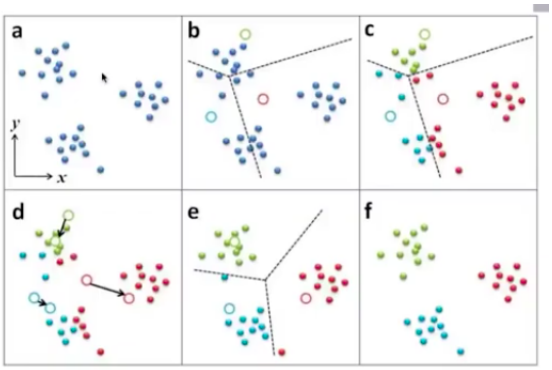
$$z_i = \underset{j \in \{1,2,...,K\}}{\operatorname{argmin}} ||x_i - \mu_k||_2^2.$$

(3) Update the centres using the following minimisation (this choice minimises the within-cluster distance):

$$\mu_j = \frac{\sum_{n=1}^{N} 1_{\{z_n = j\}} x_i}{\sum_{n=1}^{N} 1_{\{z_n = j\}}}.$$

(4) If the algorithm has not converged to within some tolerance, go to Step 2 and repeat. Otherwise return $z_1, ..., z_N$.

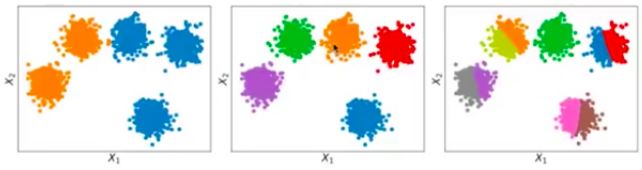
Below is an example of the convergence of the algorithm:



Let's discuss the properties of the K-means algorithm in more detail:

- **Initialisation.** The centres $\{\mu_k\}_{k=1}^K$ are usually initialised randomly from training objects. The number of clusters (and centres) K is usually fixed.
- **Convergence criteria.** There are various methods we could use to check for convergence. For example, we could set some limit on the number of iterations of the algorithm, we could demand that the centres stop changing significantly within some tolerance, or we could ask that the cluster assignments $\{z_n\}_{n=1}^N$ stop changing.
- **Solution.** The solution depends on the starting positions of the centres. In particular, this means that the algorithm is sensitive to outliers, and may create single-object clusters. To prevent this it is recommended to run the algorithm with several different initialisations, and select the solution with the minimal within-cluster distance Q.

Now we understand how K-means works, we can ask some interesting questions about it. For example - how do we know how many clusters K to split the data into? Using the wrong number of clusters can lead to bad results, e.g.

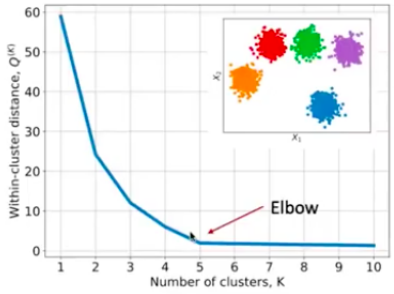


Let $Q^{(K)}$ denote the within-cluster distances for each possible number of clusters K,

$$Q^{(K)} = \sum_{n=1}^{N} ||x_n - \mu_{z_n}||_2^2.$$

We wish to minimise this over $z_1, ..., z_N, \mu_1, ..., \mu_K$.

The dependency of $Q^{(K)}$ on K is demonstrated in the figure below for the above example.

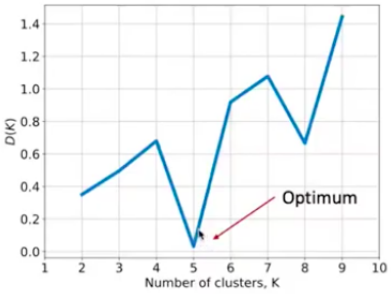


The dependence has an 'elbow' at the optimal number of clusters, $K = 5$. Let's try to formalise this notion.

We define the function $D(K)$ by:

$$D(K) = \frac{|Q^{(K+1)} - Q^{(K)}|}{|Q^{(K)} - Q^{(K-1)}|}.$$

This function takes its smallest value for the optimal number of clusters.



12.3 Quality metrics

Now we know one clustering algorithm. How do we measure the quality of a clustering algorithm?

There are two kinds of quality metrics for clustering models: *supervised* and *unsupervised*.

- *Supervised* quality metrics are based on the ground truth of the object labels. They are invariant to cluster naming. Unfortunately, very often the true labels are unknown and hence it is impossible to use these metrics in this case.
- *Unsupervised* quality metrics are based on intuition about 'good' clusters. They assume that objects from the same cluster are similar to one another, and objects from different clusters are dissimilar to one another.

One supervised quality metric is the *Rand index*:

Definition 12.2: The *Rand index* is the supervised quality metric defined as:

$$\mathrm{RI} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}.$$

Here, TP denotes *true positives*, the number of pairs in the same cluster in predictions and the ground truth. TN denotes *true negatives*, the number of pairs from different clusters in predictions and the ground truth. FP denotes *false positives*, the number of pairs in the same cluster in predictions, but from different clusters in the ground truth. FN denotes *false negatives*, the number of pairs in the same cluster in the ground truth, but from different clusters in predictions.

The Rand index takes values in $[0, 1]$, where 1 means perfect clustering. It is similar to accuracy score, but for pairs.

A slight modification is:

Definition 12.3: The *adjusted Rand index* (ARI) is defined by:

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathrm{RI}_{\mathrm{expected}}}{\mathrm{RI}_{\mathrm{max}} - \mathrm{RI}_{\mathrm{expected}}}.$$

The expected Rand index is taken by averaging the Rand index over all possible object assignments to the clusters.

The ARI has a value close to 0 for random labelling, independently of the number of clusters and sample, and exactly 1 when the clustering is ideal.

As we have just seen Rand index is very similar to accuracy score. We can compute similar metrics like precision, recall, or F_1-score in the same sorts of ways by just considering pairs.

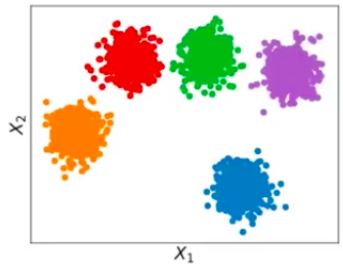
One of the most popular unsupervised metrics is the *silhouette metric*:

Definition 12.4: The *silhouette metric* is defined by:

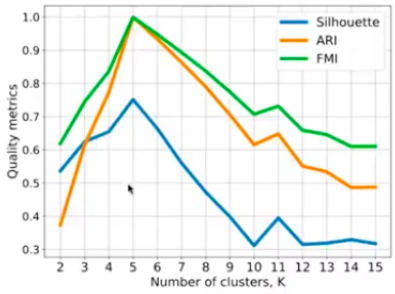
$$\mathrm{Silhouette} = \frac{1}{N} \sum_{i=1}^{N} \frac{d_i - s_i}{\max(d_i, s_i)}$$

where s_i is the mean distance between the ith object and all objects in the same cluster, and d_i is the mean distance between the ith object and all objects in the nearest (distinct) cluster. It is based on the idea that objects in one cluster should be closer to one another than objects in different clusters.

Example 12.5: Consider clustering of the following data with K clusters:



The quality metrics for the clustering are given below:



Each quality metric predicts the optimum number of clusters is $K = 5$.

12.4 Limitations of the K-means algorithm

12.5 Hierarchical clustering

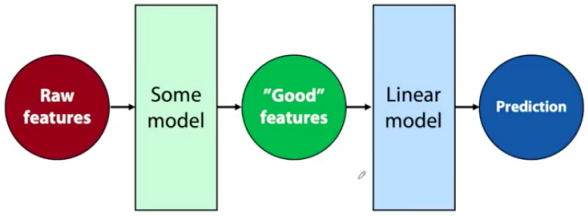
12.6 DBSCAN

13 Introduction to neural networks

13.1 From linear models to neural networks

Neural networks can be thought of as a natural extension of linear models. Recall how for linear models, we introduced *new features* to make the model *more powerful*. Finding good features (aka feature engineering) is a *highly non-trivial task*. We therefore ask the question: *can we automate feature engineering?*

One idea is to add another model that takes in the raw features, and output some 'good' features such that our problem can be solved with a linear model.

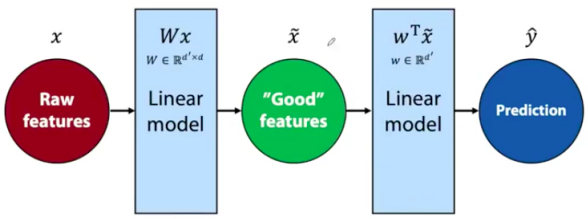


However, it's not quite clear how we can optimise this model, since we don't have a criterion for 'good' features.

One possible solution is to *train everything simultaneously*. We can use (stochastic) gradient descent for this procedure if both models are *differentiable*. In such a case, we can just treat the whole thing as a single model.

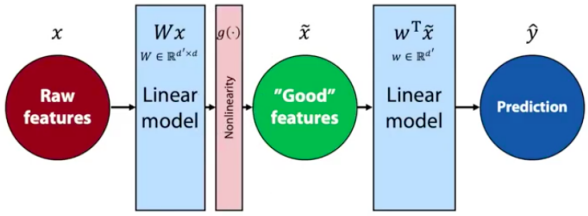
However, we should note that stacking models in this manner typically makes the loss function non-convex, thus there are *no* convergence guarantees if we choose to do this. We'll discuss this problem a bit later.

The simplest additional model we could add is another linear model:



Here, we have $\hat{y} = w^T\tilde{x} = w^T(Wx) = (w^TW)x = (w')^Tx$. In particular, the result is just another linear model with slightly different parameters. This isn't more expressive than a single linear model.

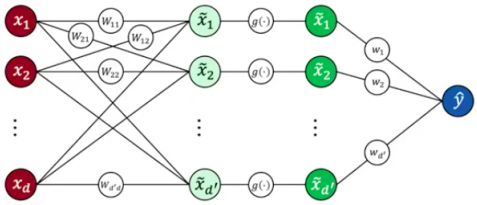
To fix this, we *remove* the linearity so the above manipulation is invalid. We can do this simply by adding some *nonlinear scalar function* (applied elementwise) after the first linear model.



In this case, we have $\hat{y} = w^T\tilde{x} = w^T g(Wx)$, which is non-linear in x. Explicitly writing the indices, we have:

$$\hat{y} = w^T\tilde{x} = w^T g(Wx) = \sum_j \left[w_j g\left(\sum_i W_{ji} x_i \right) \right].$$

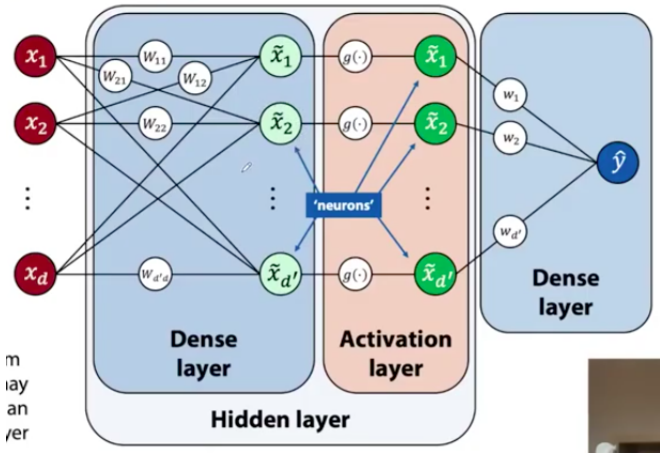
This can be visualised as the following diagram:



Models of this form are called *artificial neural networks*.

Let's introduce some more terminology associated with these networks:

Definition 13.1: In the diagram above, we label the following features:



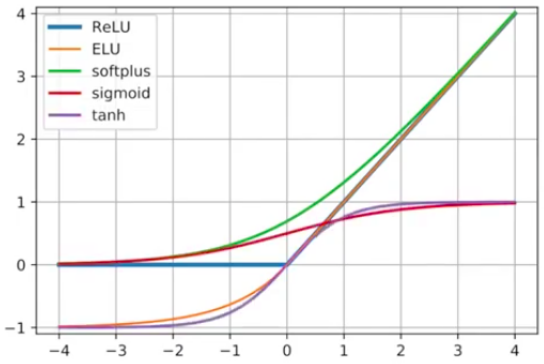
The function g is often called an *activation function*.

13.2 Activation functions

Let's begin by discussing the possible activation functions we could use to introduce non-linearity.

Definition 13.2: Possible activation functions include:

- The *rectified linear unit*, $\mathrm{ReLU}(x) = \max(0, x)$.
- The *exponential linear unit*,
$$\mathrm{ELU}(x) = \begin{cases} x & x \geq 0, \\ e^x - 1 & x < 0. \end{cases}$$
- The *soft-plus* activation:
$$\mathrm{softplus}(x) = \log(1 + e^x).$$
- The *sigmoid* function:
$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$
- The *hyperbolic tangent* function $\tanh(x)$.



The final two functions are slightly tricky to use due their 'saturational' properties for very large positive and negative units. Typically the rectified linear unit is a good first choice, but other functions can be tried if desired.

Sigmoid and hyperbolic tangent functions are typically reserved for the final layers, where we want our output to be in a specific range.

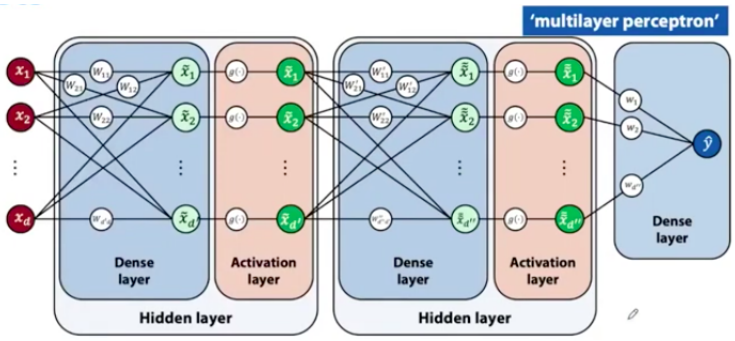
13.3 Universal approximators and deeper nets

This simple modification to a linear model (stacking linear models with only a single non-linearity in between them) is very powerful. In fact, there is a theorem that tells us that a single hidden linear layer with a nonlinearity makes our model a *universal approximator*:

Proposition 13.3: Any function can be approximated *arbitrarily closely* by an artificial neural network with a single hidden layer, given a sufficiently wide hidden layer (i.e. large enough dimension d' for the output of the first linear model).

In practice however, in may be impossible to find this approximation. This might be due to a heavily non-convex loss function, an infeasibly large dimension d', or overfitting.

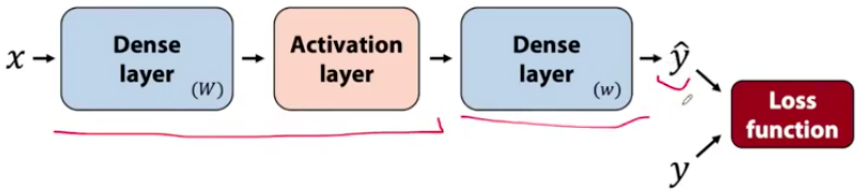
In practice, we can instead stack more hidden layers, reducing the number of neurons required to represent a given function. Such a layer is called a *multilayer perceptron*:



13.4 Backpropagation

Now let's discuss in more detail how to compute the derivatives to perform the gradient descent.

A schematic view of the flow of information through our model is given below.



Our loss function could be, for example, mean squared error loss:

$$L = \frac{1}{N} \sum_{i=1,\dots,N} \left(y_i - w^T g(W x_i)\right)^2 .$$

If we denote our layers as functions $\mathrm{Dense1}$, $\mathrm{Activation1}$ and $\mathrm{Dense2}$, the model can be written as:

$$\hat{f}(x; w, W) = \mathrm{Dense2}(\mathrm{Activation1}(\mathrm{Dense1}(x))).$$

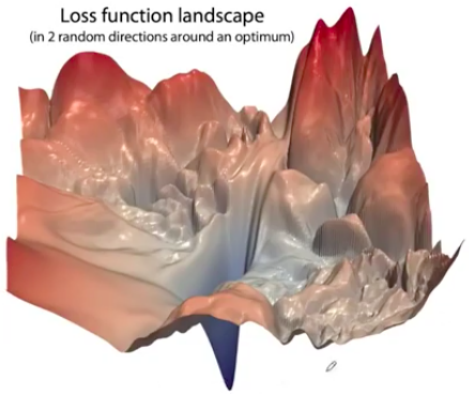
The loss can be viewed as a function $L(w, W) \equiv L(y, \hat{f}(x, w, W))$. By the chain rule then, we have:

$$\frac{\partial L}{\partial W} = \frac{\partial L(y, \hat{f})}{\partial \hat{f}} \cdot \frac{\partial \hat{f}}{\partial W} = \underbrace{\frac{\partial L(y, \hat{f})}{\partial \hat{f}}}_{\text{scalar}} \cdot \underbrace{\frac{\partial \mathrm{Dense2}}{\partial \mathrm{Activation1}}}_{d'\text{-vector}} \cdot \underbrace{\frac{\partial \mathrm{Activation1}}{\partial \mathrm{Dense1}}}_{d' \times d' \text{ diagonal matrix}} \cdot \underbrace{\frac{\partial \mathrm{Dense1}}{\partial W}}_{d' \times d' \times d\text{-tensor}}.$$

Overall, the result is a $d' \times d$ matrix. The *backpropagation algorithm* is essentially just applying the chain rule repeatedly. The actual algorithm states how to do this in an efficient way.

13.5 Optimisation techniques

Stacking linear models with a non-linearity in between makes the resulting problem highly non-convex. For example, a loss function landscape might look like:



There is no convergence guarantee for (stochastic) gradient descent when we don't have convexity! However, there's a number of modifications we can do to improve the situation.

First, let's recap stochastic gradient descent. Recall that at each step k in the procedure, we pick $l_k \in \{1, ..., N\}$ at random, then update:

$$\theta^{(k)} = \theta^{(k-1)} - \eta \nabla_\theta \mathcal{L}(y_{l_k}, \hat{f}_\theta(x_{l_k}))\Big|_{\theta=\theta^{(k-1)}}.$$

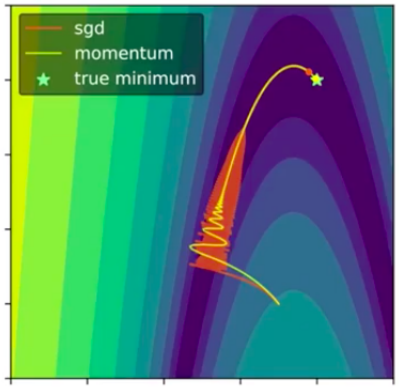
A modification is called *mini-batch stochastic gradient descent*. Here, we shuffle the training set, then iterate through it in chunks (batches) of fixed size. At each iteration, we evaluate the loss gradients on the given chunk B:

$$g = \sum_{i \in B} \nabla_\theta \mathcal{L}(y_i, \hat{f}_\theta(x_i)).$$

We then update the parameters as usual: $\theta^{(k)} = \theta^{(k-1)} - \eta \cdot g$.

Another modification is *momentum stochastic gradient descent*. The basic idea is to introduce *inertia* into the procedure (like a ball rolling down a hill). This method smooths out fast oscillations and helps get out of small local minima; it also allows for larger ranges of learning rates. The relevant equations are:

$$m^{(k)} = \beta m^{(k-1)} + (1-\beta)\frac{\partial L}{\partial \theta}\bigg|_{\theta=\theta^{(k-1)}}, \qquad \theta^{(k)} = \theta^{(k-1)} - \eta m^{(k)}.$$



Another modification to stochastic gradient descent is called RMSprop. The idea is to adjust the learning rate separately for different components of the parameter vector. As the gradients get smaller, we increase the learning rate (we scale by the inverse root mean square of the gradient):

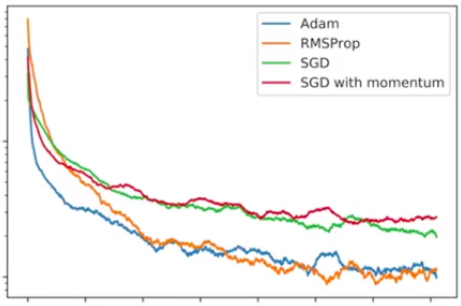
$$\mathbb{E}[g^2]_{(k)} = \beta \mathbb{E}[g^2]_{(k-1)} + (1-\beta)\left(\frac{\partial L}{\partial \theta}\right)^2\bigg|_{\theta=\theta^{(k-1)}}.$$

We then update via:

$$\theta^{(k)} = \theta^{(k-1)} - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_{(k)} + \epsilon}}\frac{\partial L}{\partial \theta}\bigg|_{\theta=\theta^{(k-1)}}.$$

Note that we must introduce a stabilisation constant ϵ to avoid division by zero in some cases.

A final modification to stochastic gradient descent, called Adam, combines both these ideas (momentum and RMSprop). This is typically a good first choice for an optimising algorithm. A comparison is given below:

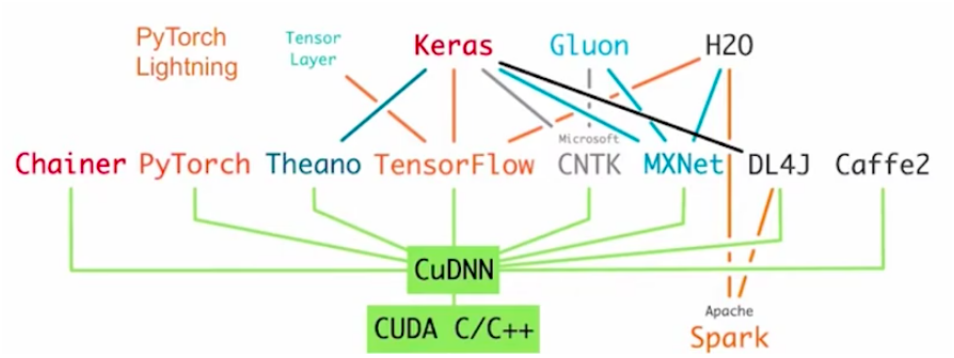


14 Introduction to PyTorch

In this section, we discuss architectural details of *PyTorch*.

14.1 Deep learning frameworks

There are many deep learning frameworks available today:



The lower on the diagram, the lower the level of programming. They differ with how they deal with hardware, etc, but also with how they organise *automatic differentiation* - this is the main mechanism that underlies computation of gradients needed during training of the network.

The most popular is probably Google's *TensorFlow*. It's mainly known for good production quality.

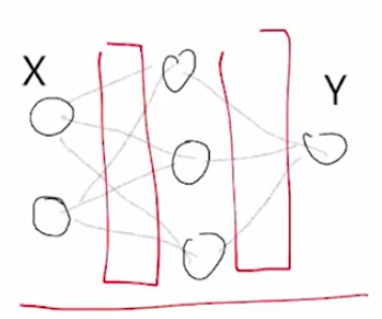
PyTorch is backed up by Facebook's AI research and is nicely accepted in the research community. It helps to write concise code, it is flexible and transparent. There is also *PyTorch Lightning* which simplifies some of the lower-level aspects of PyTorch. We will focus on PyTorch and PyTorch Lightning only.

Highlights of PyTorch include:

- Simple, transparent development and debugging.
- A rich ecosystem: plenty of pretrained models, NLP, Vision, interpretation, hyper-optimisation.
- Production ready (export of the trained model into different formats).
- Distributed training, declarative data parallelism.
- Cloud deployment support.
- Choice of many industry leaders and researchers.

14.2 Neural network representation

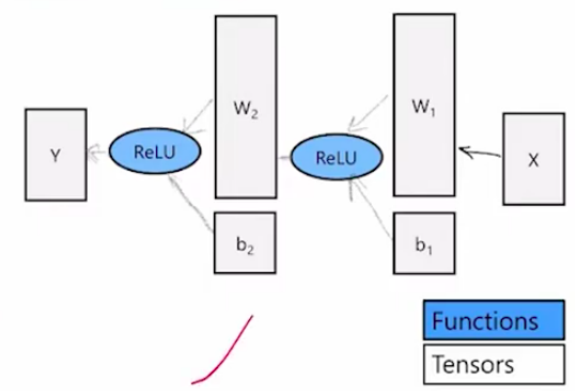
Let's take a look at a lower level description of the neural network design. We start with an 'abstract representation':



This can be written in 'linear algebra' form as:

$$Y = \mathrm{ReLU}\left(W_2 \cdot \mathrm{ReLU}(W_1 X + b_1) + b_2\right).$$

We can of course represent this diagramatically as a graph consisting of tensors and functions:



These building blocks are implemented in the code. Tensors are created either for nested lists or from numpy arrays:

```
torch.Tensor(L)                    # Create tensor from (nested) list or numpy array L.
torch.[ones|zeros](*size)          # Tensor with all ones or all zeros.
x.clone()                          # Clone an existing tensor.
```

You can explicitly specify which device on which you wish a tensor to be placed (either CPU, GPU or TPU), or later on you can move a tensor between devices:

```
torch.ones([2,4], dtype=torch.float64, device=cuda0)

# These methods move tensors between different devices.
x.cuda()
x.cpu()
mytensor.to(device)
```

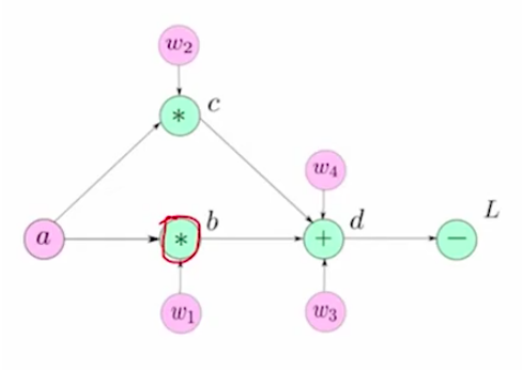
Operations can be performed only if arguments are on the same device.

14.3 Building blocks, graph

When we have all our tensors, we need a good way to compute gradients using backpropagation (i.e. the chain rule). Let's consider a toy example:

$$b = w_1 * a, \qquad c = w_2 * a, \qquad d = w_3 * b + w_4 * c, \qquad L = 10 - d.$$

for some operation $*$. This is represented using the graph:



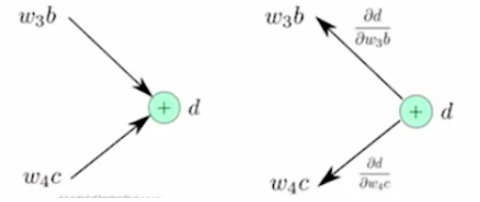
Mathematical operations are implemented in PyTorch as:

```
A.mm(B)          # Matrix multiplication.
A.mv(x)          # Matrix–vector multiplication (for some reason different to matrix multiplication).
x.t()            # Matrix transpose.
```

To perform backpropagation, we need to compute:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}.$$

Graphically, we swap the directions of arrows:



The gradient can be computed *automatically* in PyTorch (this is the most important and behind-the-scenes thing).

Each `Tensor` object has an attribute `grad_fn`, which refers to the mathematical operation which created the tensor. If `Tensor` is a leaf node (initialised by the user), then the `grad_fn` is `None`. Each tensor also has an argument `requires_grad`:

```
a = torch.randn((3,3), requires_grad = True)
```

If set to `True` (which is *not* the default argument), we can compute intermediate gradients for this tensor.

All mathematical steps a represented by classes inherited from `torch.autograd.Function`. Every `Function` entering the graph has two methods:

- The `forward` method, which does the obvious thing - computing the forward direction node output and saving it to an internal buffer.
- The `backward` method, which stores the incoming gradient from the downstream node, and multiplies by a local gradient, before sending it to the upstream node.

Inside the backward method, the details are:

```
def backward(incoming_gradients):
        self.Tensor.grad = incoming_gradients

        for inp in self.inputs:
                if inp.grad_fn is not None:
                        new_incoming_gradients = // incoming_gradient * local_grad(self.Tensor, inp)

                        inp.grad_fn;backward(new_incoming_gradients)
                else:
                        pass
```

Consider the following example:

```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1 * a
c = w2 * a

d  = w3 * b + w4 * c

# Replace L = 10 - d by
L = (10 - d).sum()

L.backward()
```

When we use the `backward` method at the end, all the gradients in the network involved in the computation of L are updated, in the way described above. Optimisation via gradient descent then becomes very easy:

```
learning_rate = 0.5
w1 = w1 - learning_rate * w1.grad
```

Usually, this step happens within an optimiser inside the PyTorch platform.

In general, PyTorch does a lot of work behind the scenes:

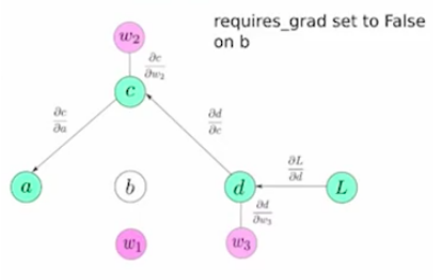
- Calling `forward` creates:
  - A graph with the intermediate note output values.
  - Buffers for the non-leaf nodes.
  - Buffers for intermediate gradient values.
- Calling `backward`:
  - Computes gradients.
  - Frees the buffers and destroys the graph

When we next call `forward`, the leaf node buffers from the from the previous run are shared, and non-leaf nodes' buffers are recreated.

Due to the flexibility of the network architecture, it is not obvious when an iteration of gradient descent stops, so `backward`'s gradients are accumulated each time a variable (`Tensor`) occurs in the graph. This is usually desired for *recurrent neural networks*. If you do not need to accumulate these, you *must clean previous gradient values* at the end of each iteration:

- Either use `x.data.zero_()` for every model tensor `x`.
- Or, use optimiser's `zero_grad()` method, which is much more preferable.

As discussed above, the `requires_grad` attribute of the `Tensor` class is by default `False`. This comes in handy if you must freeze some layers and stop them from updating parameters while training. Thus, no gradient is propagated to them, or to those layers which depend upon these layers for gradient flow.



When set to `True`, `requires_grad` is contagious - even if one operand of an operation has `requires_grad` set to `True`, so will the result.

This can be useful when we use a pre-trained model:

```
model = torchvision.models.resnet18(pretrained=True)

# Freeze some layers in the existing model
for param in model.parameters():
        param.requires_grad = False

# Replace the last fully-connected layer with our own
# Parameters of newly constructed modules have requires_grad = True by default
model.fc = nn.Linear(512,100)

# Optimise only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

14.4 Inference

When we are computing gradients, we need to cache input values, and intermediate features as they may be required to compute the gradient later. The gradients of $b = w_1 * a$ with respect to its inputs w_1 and a are a and w_1 respectively.

We need to store these values for gradient computation during the backward pass. This affects the memory footprint of the network. However, when we are performing inference, we don't compute gradients:

```
with torch.no_grad:
        # inference code goes here
```

14.5 Neural network class

The main class that wraps everything in PyTorch is `torch.nn.Module`.

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):

        # We write all the tensors that need to be trained in the constructor
        def __init__(self):
                super(Model, self).__init__()
                self.conv1 = nn.Conv2d(1, 20, 5)
                self.conv2 = nn.Conv2d(20, 20, 5)

        # We also define how all the building blocks need to be connected in the forward procedure
        def forward(self, x):
                x = F.relu(self.conv1(x))
                return F.relu(self.conv2(x))
```

We must also write loss functions and optimisers (the optimisers are all already implemented in PyTorch - just pick one).

14.6 PyTorch Lightning

PyTorch Lightning is the lightweight PyTorch wrapper for high-performance AI research. It has maximal flexibility, whilst removing the boilerplate code. It has self-contained models and modular code.

We will take a closer look at this during the seminars.

15 Network regularisation

15.1 Weight initialisation

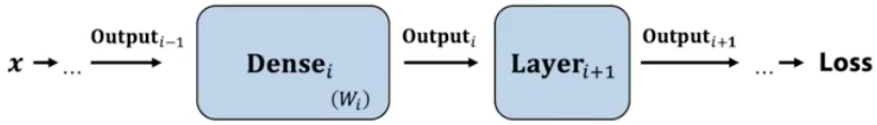
Consider a neural network. Is it possible to initialise all weights with the same value? If we do so, within each layer the gradients for each of the weights will be the same as well, hence the *updates will be the same*, and thus the network degrades! All the weights will continue being the same.

Thus initialisation with a constant is a bad idea. Perhaps we should use a random initialisation instead. For simplicity, we omit the activation functions for now. Then the output of a neural network composed of dense layers only is:

$$\hat{y} = W_{\text{out}} \cdot \ldots \cdots W_{h2} \cdot W_{h1} x.$$

Note that the gradient with respect to any of the weight matrices W_{hk} is proportional to the *product* of all other matrices. For example, for 1×1 matrices, if all are of scale $S \in \mathbb{R}$, the gradient g is proportional to $g \sim S^{m-1}$, where m is the *depth* of the network. For S too large, the gradients will *explode*; for S too small, they will *vanish*.

More realistically, consider a network with hidden layers.



We have in this case:

$$\frac{\partial \text{Loss}}{\partial W_i} = \frac{\partial \text{Loss}}{\partial \text{Output}_i} \cdot \frac{\partial \text{Dense}_i}{\partial W_i} = \frac{\partial \text{Loss}}{\partial \text{Output}_{i+1}} \cdot \frac{\partial \text{Layer}_{i+1}}{\partial \text{Output}_i} \cdot \text{Output}_{i-1}.$$

Continuing this process, we accumulate the product of the gradients for the subsequent layers. Therefore, the idea is that for stable learning we would like to keep the scale of the gradients the same at each step:

$$\text{Var}\left(\frac{\partial \text{Layer}_{i+1}}{\partial \text{Output}_i} \cdot \frac{\partial \text{Layer}_i}{\partial \text{Output}_{i-1}}\right) \approx \text{Var}\left(\frac{\partial \text{Layer}_{i+1}}{\partial \text{Output}_i}\right).$$

Similarly, we would like the output to remain at the same scale at each step of the forward pass:

$$\text{Var}(\text{Layer}_{i+1}(\text{Layer}_i(\text{Output}_{i-1}))) \approx \text{Var}(\text{Layer}_i(\text{Output}_{i-1})).$$

This requirement ensures that the activation of each layer is not too small and not too high.

These two requirements may in fact contradict each other. For example, for ReLU activation, they result in the initialisation requirements:

$$\text{Var}(W_{ij}) = \frac{2}{\text{number of outgoing connections}}, \qquad \text{Var}(W_{ij}) = \frac{2}{\text{number of incoming connections}}.$$

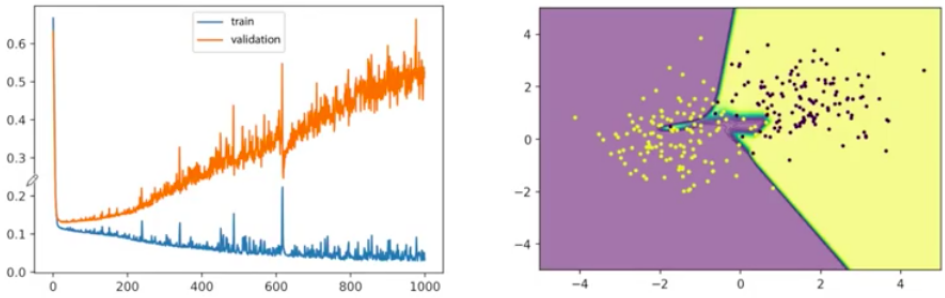
Typically we can just choose one of them, or we can try to average them out:

$$\text{Var}(W_{ij}) = \frac{4}{\text{number of outgoing connections} + \text{number of incoming connections}}.$$

This type of initialisation is called *He* initialisation.

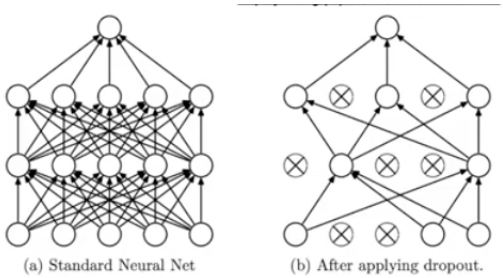
15.2 Overfitting with neural networks

Since neural networks are highly complex models, they are prone to overfitting.



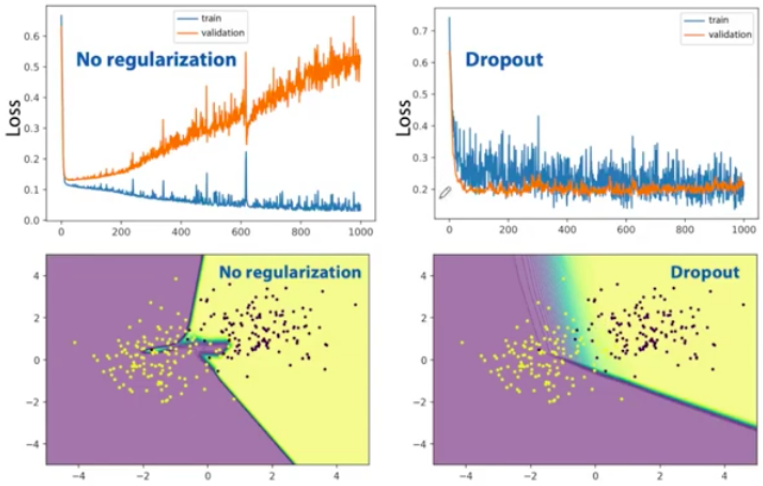
Regularisation techniques like $L1$ and $L2$ regularisation are also available for neural networks. We could also use *early stopping* (i.e. stop the training before the validation error grows).

There are some alternative methods that are only available for neural networks. The first we will discuss is *dropout*: at train time, set neuron activations to 0 with a given probability p. At test time, multiply the activation by p (i.e. set it to the *expected value*). This makes the neurons learn to work with a *randomly chosen sample* of other neurons. It drives individual neurons towards *creating useful features* rather than relying on other neurons to correct its mistakes.



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

Applying dropout to the above problem, we see a big difference:



Note however, we still see some signs of overfitting - early stopping might be more useful in this case.

15.3 Normalisation layers

This technique was originally proposed to mitigate *internal covariate shift* (the updates in one layer changing the input distributions of the subsequent layers drastically). The proposed solution was to compute the *mean* and *variance* of inputs on a single batch B:

$$\mu_B = \frac{1}{|B|}\sum_{i\in B} x_i, \qquad \sigma_B^2 = \frac{1}{|B|}\sum_{i\in B}(x_i - \mu_B)^2.$$

We then *normalise* the input, and *scale and shift* (with the trainable parameters γ, β):

$$y_i = \gamma \cdot \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta.$$

This technique turned out to be *extremely powerful* in many cases. It resulted in faster and more stable convergence.

However, it was also proved to *not* reduce the internal covariate shift! It is not quite well understood why it works, but it works so people use it.

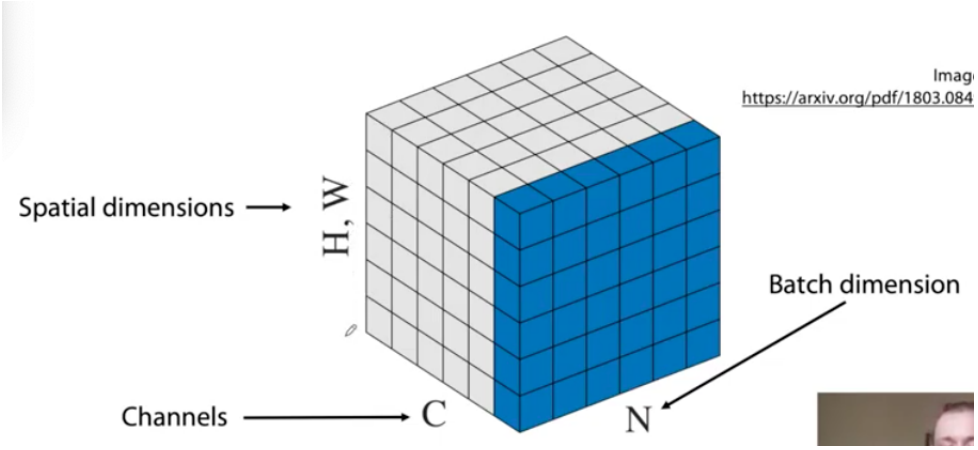
Effectively, we *remove* the 'shift' and 'scale' degrees of freedom from the previous layer. We transfer these degrees of freedom to the new batch normalisation layer. This separation seems to make the learning more stable.

We now discuss some more practical considerations related to this method. First, we ask: which dimension should we normalise over? Typically, in a fully-connected layer we work with either:

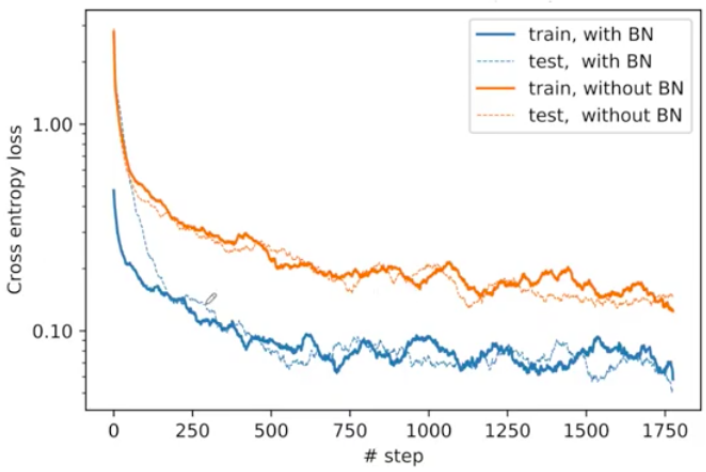
- Batches of one-dimensional vectors (which overall are of the shape batch dimension $\times$ features dimension). In this case, we normalise separately for each component in the features dimension, i.e. we normalise over the batch dimensions.
- For tensors, the dimensions are:

  $$\text{batch dimension} \times \text{spatial dimension } 1 \times \ldots \times \text{channel dimensiona}.$$

  We normalise separately for each component of the channel dimension, i.e. over batch dimension $\times$ spatial dimension 1 ... etc.

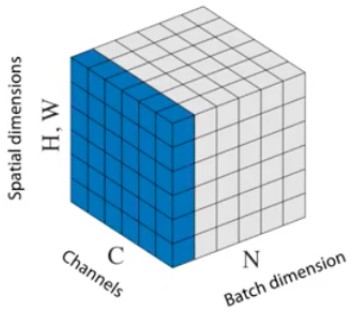


Calculating batch statistics at test time may be problematic, e.g. when there's a single object to predict. Instead, we can calculate the running mean and variance during training, and apply batch normalisation at test time.



In some cases, batch normalisation is a bit problematic. Batch normalisation imposes limits on the batch size - if this is too small, the variance of the sample statistics will be too high. In particular, it is problematic to use in recurrent networks.

There is an alternative in this case: *layer normalisation*. The maths is the same, except statistics are calculated over channels rather than batch elements. The effect is quite different though - layer normalisation 'entangles' different neurons within a layer.



16 Convolutional neural networks

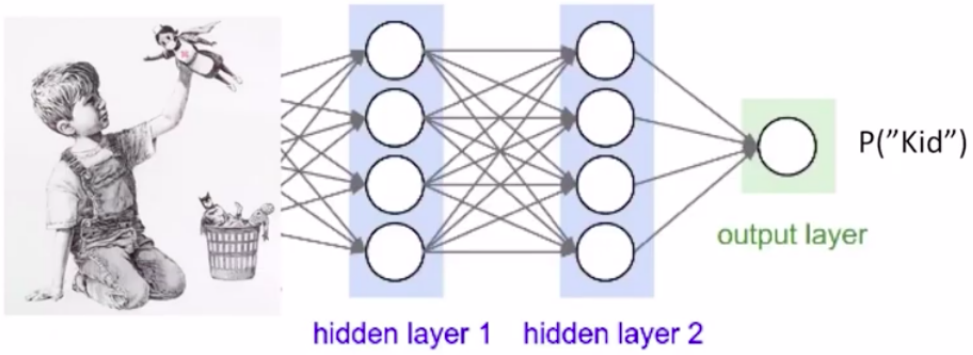
In this section, we overview computer vision tasks. Often these are best tackled using *convolutional neural networks*.

16.1 Image recognition

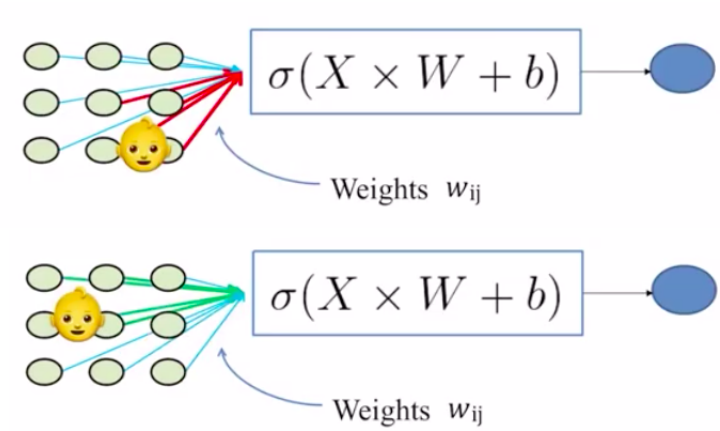
A grayscale image is a matrix of pixels of dimensions $H \times W$ (heigh times width). Pixel is an abbreviation of 'picture elements'. Each pixel sores a number in the range $[0, 255]$ corresponding to its brightness.

On the other hand, a colour image is a three-dimensional array of dimensions $H \times W \times 3$ or $3 \times H \times W$. The three channels correspond to red, green and blue.

In order to recognise an image, we must send this multi-dimensional object into a neural network:

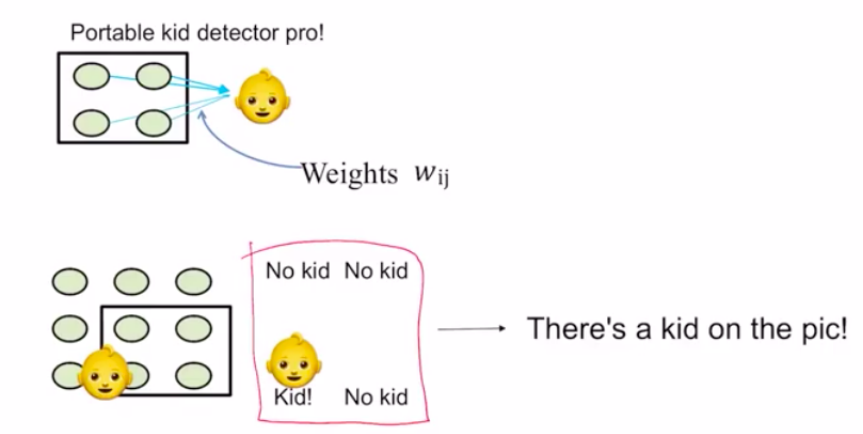


One challenge is that images are very big. Another challenge is that images are not stuck in one particular location in the matrix, as in the figure below:



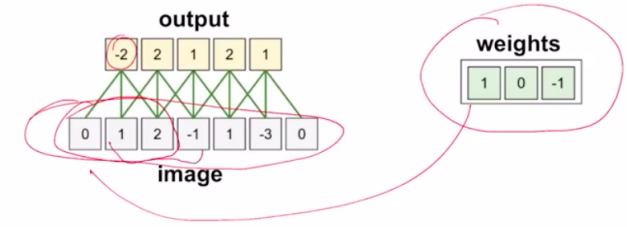
In the worst case, your network will have to learn about each of these cases separately - i.e. we might have to have one neuron per position!

The main idea to overcome these challenges is to encode the output 'kid' as a weight tensor that we can shift across the image:

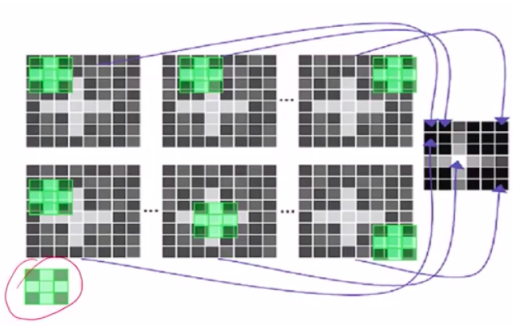


We move this weight tensor around the image to form a matrix of detections: 'no kid', 'no kid', 'kid', 'no kid' etc. The output is both a position of a feature and an identification of the image.

The mathematical procedure that allows us to achieve this is *convolution*. Given a vector of weights $(1, 0, -1)$ for example and an image, we systematically apply the weights to the image using a dot product. In the below figure, this process is described in the one-dimensional case:



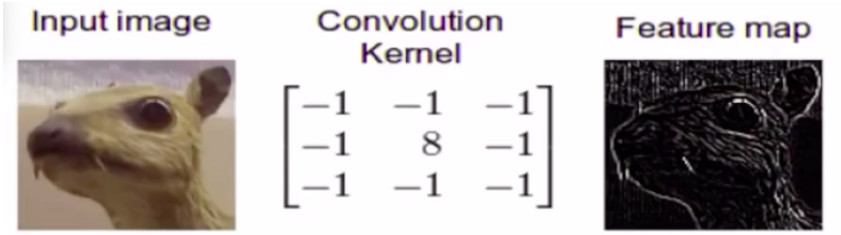
In the two-dimensional case, things are the same:



We apply the relevant *filter* to all patches of the image. The result is to highlight regions of the image where things are similar to the filter. Therefore: our intuition should be 'how kid-like is this filter'?

Example 16.1: A well-known convolutional kernel used for *edge detection* is:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}.$$



We can add a few tricks to the convolution to make it more flexible and useful in a greater variety of cases:

- **Padding.** We pad the edges of the image with extra, 'fake' pixels (usually of value 0, hence the term 'zero padding'). This way when we slide the filter around, we can allow the original edge pixels to be at its centre (while extending into the fake pixels beyond the edge), producing an *output the same size as the input*.
- **Striding and dilation.** The idea of *striding* is to skip some of the slide locations on the image; the *stride* is the number of pixels that we skip.

  Dilated convolution is a basic convolution only applied to the input volume with defined gaps. This is useful when we are working with high resolution images, but fine-grained details are still important.

16.2 Convolutional layer

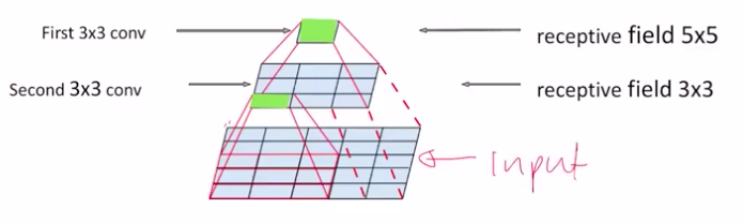
A convolutional layer for a neural network learns the weights of a filter by *gradient descent*. We may specify as many as we want, one per *output channel*.

The output value of the layer with input size (N, C_{in}, H, W) and output size $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ is given by:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) * \text{input}(N_i, k).$$

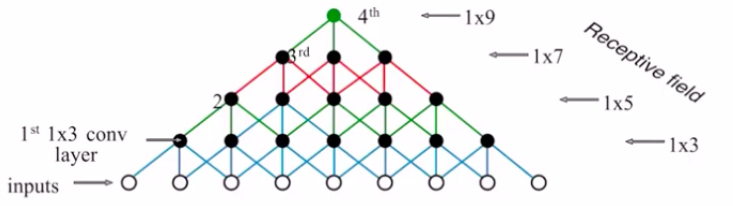
Here N is the batch size, and C is the number of channels. The total number of learnable parameters is $C_{\text{out}} \cdot (C_{\text{in}} \cdot \text{size}(\text{kernel}) + 1)$.

A very important aspect of the convolutional layer is how much information it is capable it is seeing, which corresponds to a single feature at the end.



Adding multiple convolutions increases the 'field of view'. Stacking convolutions extends the *receptive field*, and allows us to send the information to a higher level classification algorithm.

How many 3×3 convolutions should we use to recognise a 100×100 pixel object? Around 50 - this is too much, we need to increase the receptive field faster!

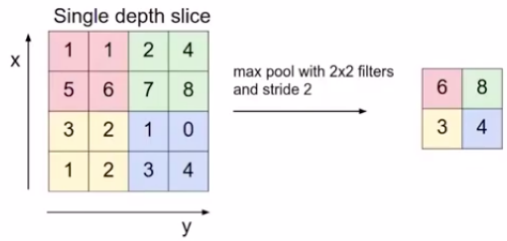


How do we pick the size of the kernel/filter in general? First, we should have an odd number number of dimensions for the kernel (i.e. there should be a centre of the kernel).

If the input image is larger than 128×128 pixels, say, use 5×5 or 7×7 filters, and then quickly reduce the spatial dimensions - use 3×3 afterwards. If our images are smaller, then we should start with 3×3 to start with.

16.3 Pooling

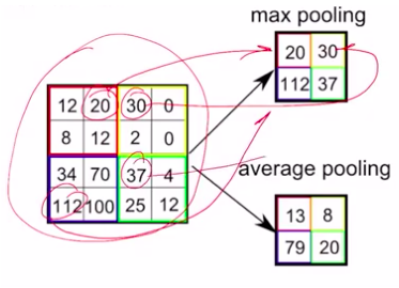
One technique used to extend the receptive field is *pooling*.



Here, we select patches of the same colour that are the size of the kernel (2×2 in this case) and we average over this part of the image. This removes redundant information and compresses it to highly responsive parts that have shown previously that location of this patch corresponds to the presence of an object.

In this way, we reduce the layer size by a factor of the size of the kernel. This makes the NN less sensitive to small image shifts. This procedure also has no parameters to train!

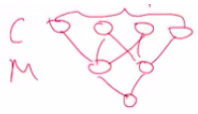
Popular types of pooling include: max pooling and mean pooling.



Let's check we understand some sizes in each of the following cases:

Example 16.2: How large is the receptive field for a module consisting of a convolution $\mathrm{Conv1d}(3 \times 1, \text{stride } = 1)$, and pooling $\mathrm{MaxPool1d}(2 \times 1, \text{stride } = 2)$?

We draw the module:



We see that the size of the receptive field is 4.

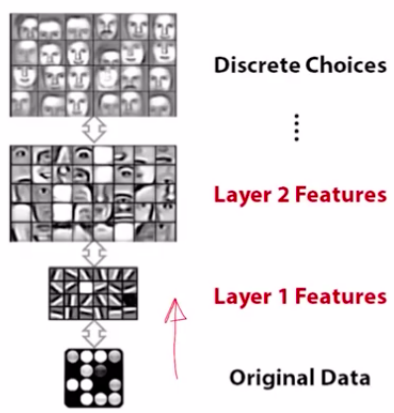
Example 16.3: How many parameters should a network consisting of $\mathrm{Input}(100 \times 100 \times 3)$, $\mathrm{Conv2d}(3 \times 3, 2$ out channels, bias $=$ True$)$, $\mathrm{MaxPool2d}(3 \times 3, \text{stride } = 3, \text{dilation } = 1)$?

It is important to remember that the number of training parameters for convolution layers is:

$$\text{number of input channels } (3) \times \text{ size of kernel } (3 \times 3) \times \text{ number of output channels } (2)$$
$$+ \text{ number of output channels to account for bias } (2) = 56.$$

16.4 Convolutional features stacking

The intuition for stacking many convolutional layers is to identify simpler features at each layer (e.g. straight lines on the first, combinations of straight lines on the second, etc). Eventually, we should arrive at a set of discrete choices (e.g. faces of people).



There are many ways to stack these layers together. The state of the art is *Google Vision*.

However, there are many problems that can happen with large networks:

- Memory can run out.
- Gradients can vanish or explode.
- Activations can vanish or explode.

Possible solutions include using *pre-trained networks*, or using different normalisation techniques.

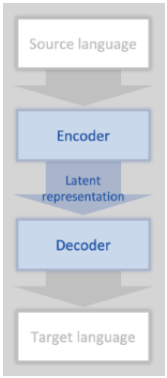
17 Transformers

17.1 Introduction to sequence to sequence models

Machine translation is a useful example of a sequence to sequence problem. The aim is to train a computer to translate a sentence from one language into another.

A *sequence to sequence model* for machine translation consists of two parts: one part relates to *natural language understanding* - reading the source sentence - and the other part relates to *natural language generation* - creating the output.

The basic architecture is as follows. The *encoder* inputs a sentence in the source language, and the *decoder* outputs a sentence in the target language. The *latent representation* is a vector representation of the input sentence, created by the encoder and used by the decoder (it serves as a bridge between the encoder and decoder).



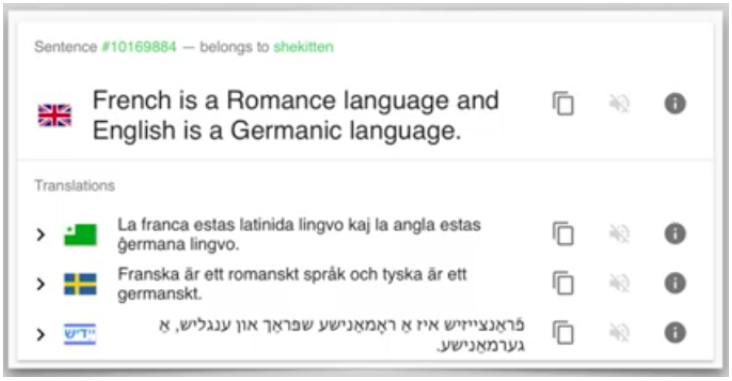
The encoder and decoder are trained jointly with the same loss functions, to create an overall model.

We write the source and target sentences as tuples:

- We begin with a *source sentence* $\mathbf{x}_{\text{source}} = (x_1, ..., x_n), x_i \in V_{\text{source}}$, where V_{source} is the finite *vocabulary* of the source language.
- We aim to recover a *target sentence* $\mathbf{y}_{\text{target}} = (y_1, ..., y_m), y_i \in V_{\text{target}}$.

To create a useful machine translation system, both $\mathbf{x}_{\text{source}}$ and $\mathbf{y}_{\text{target}}$ should have similar *semantic meaning*.

A *parallel corpus* consists of a collection of sentences written in different languages, all with the same meaning. We can use such parallel sentences to train a machine translation system.



Our goal is to maximise $p(\mathbf{y}|\mathbf{x})$, the probability of the target sentence given the source sentence. Our objective function is the log likelihood:

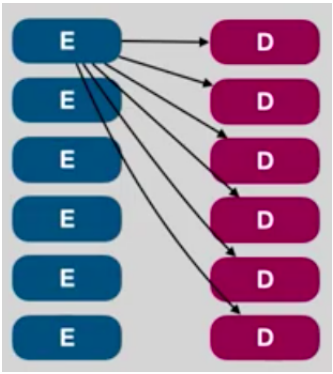
$$\mathcal{L}_\theta = \sum_{x,y \in C} \log(p(\mathbf{y}|\mathbf{x};\theta)),$$

where C is the dataset (namely the parallel corpus).

17.2 Transformers

Transformers provide a faster and more efficient model for machine translation than previous ones. The core ideas include: *multi-head attention mechanism* and *two stacks of layers*. We will discuss the multi-head attention mechanism in detail.

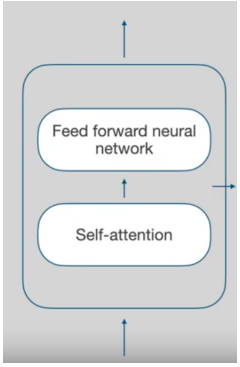
The encoder and decoder and both stacks of blocks, organised sequentially with their own inputs and outputs.



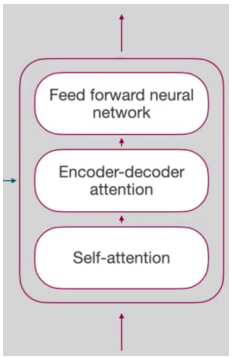
The first layer of the encoder inputs the source sentence. The last layer of the decoder outputs the sentence in the target language. All other blocks output something intermediate.

The encoder block consists of two sublayers:

- The self-attention layer helps to discover relations between words within sentences.
- The feed forward neural network layer aggregates outputs of the self-attention layer.



The decoder block has the same two layers, but between them there is an encoder-decoder attention layer. An encoder-decoder attention layer helps the decoder to focus on different input words:



In total the encoder consists of N identical blocks. Each block inputs the outputs of the previous block except the first one. The first block inputs *words embeddings*.

A word embedding is a vector; each word is represented by a single vector of size 512. The vectors are *dense* which means that there are not many zeroes in them.

The more similar words are, the closer their embeddings are, measured by the cosine similarity $\cos(x_i, x_j)$. The embeddings are trainable, i.e. get updated when the whole model is trained.

The embeddings are actually the sum of word embeddings and positional encodings, $x_i = e_i + (pe)_i$.

17.3 Inside the encoder block

Step 1: Self-attention. Each word embedding is transformed into a query, a key and a value vector:

- Queries: $q_i = W^Q x_i$.
- Keys: $k_i = W^K x_i$.
- Values: $v_i = W^V x_i$.

The weight matrices W^Q, W^K, W^V are trainable (i.e. updated during the training).

In matrix notation, the word embeddings X are multiplied by weight matrices:

- Queries: $Q = XW^Q$.
- Keys: $K = XW^K$.
- Values: $V = XW^V$.

Part III
Bayesian deep learning

18 Introduction to Bayesian methods

18.1 Frequentist vs Bayesian frameworks

Begin by recalling the definitions of conditional probabilities:

Definition 18.1: Let x, y be random variables. The *conditional probability of x given y* is:

$$p(x|y) = \frac{p(x,y)}{p(y)},$$

where $p(x,y)$ is the probability of x *and* y.

This satisfies some nice properties:

Proposition 18.2: We have the following:

- The *product rule*: any joint distribution can be expressed as a product of one-dimensional conditional distributions:
$$p(x,y,z) = p(x|y,z)p(y|z)p(z)$$
- The *sum rule*: any marginal distribution can be obtained from the joint distribution by integrating out unnecessary variables:
$$p(y) = \int p(x,y)\,dx.$$

Proof: Both follow from the definitions of conditional probability (and the fact that the distribution $p(x|y)$ integrates to 1 over x).

Example 18.3: Suppose we have a joint distribution over three groups of variables $p(x,y,z)$. We observe x and are interested in predicting y. The values of z are unknown and irrelevant to us. How do we estimate $p(y|x)$ from $p(x,y,z)$?

We use the product and sum rules:

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{\int p(x,y,z)\,dz}{\int p(x,y,z)\,dzdy}.$$

In general, the product and sum rules allow us to obtain arbitrary conditional distributions from the joint one. However, to actually *compute* the distribution in practice, the integrals may not be tractable - in one dimension the integral can be taken numerically, but in multiple dimensions it can be difficult to take the integral numerically.

From the above simple rules, we can obtain *Bayes' theorem*:

Theorem 18.4: Bayes' theorem states:

$$p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)\,dy}.$$

Proof: By application of the product and sum rules:

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\int p(x|y)p(y)\,dy}. \qquad \square$$

Bayes' theorem defines the rule for uncertainty conversion when new information arrives; the appropriate interpretation is:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}.$$

This framework is different from the framework we normally use in machine learning, namely *frequentism*. We can see the difference when we discuss *statistical inference*. Consider independent identically distributed data $X = (x_1, ..., x_n)$ generated from some distribution $p(x|\theta)$. Suppose we must estimate θ (the 'parameters' of our model). There are two approaches:

- The *frequentist framework* is the standard method in machine learning. We use maximum likelihood estimation (MLE):

$$\theta_{\text{ML}} = \operatorname{argmax} p(X|\theta) = \operatorname{argmax} \prod_{i=1}^{n} p(x_i|\theta) = \operatorname{argmax} \sum_{i=1}^{n} \log\left(p(x_i|\theta)\right).$$

- In the *Bayesian framework*, we instead encode uncertainty about θ in a prior $p(\theta)$ and apply Bayesian inference:

$$p(\theta|X) = \frac{\prod_{i=1}^{n} p(x_i|\theta)p(\theta)}{\int \prod_{i=1}^{n} p(x_i|\theta)p(\theta)\,d\theta}.$$

There are two key differences. First, instead of a point estimate for θ as in the frequentist framework, the Bayesian framework provides us with the distribution of θ - this is good as it gives us more information, allowing us to make uncertainty estimates for example. Second, in order to perform Bayesian inference we need to choose a prior distribution $p(\theta)$ - this is sometimes a very difficult task, and we need to think carefully how to choose a prior in each specific problem.

Example 18.5: Consider tossing a coin which may be biased. The task is to estimate the probability θ of landing heads up. We are given the following data: 2 tosses with results heads in both cases, (H, H).

In the frequentist framework, we find the most probable value of θ is $\theta_{\mathsf{ML}} = 1$. Thus the coin is unfair and always lands heads up. Real world experience tells us something is wrong here!

In the Bayesian framework, we need to first choose a prior. We will suppose that the prior is symmetric about $1/2$, so that the coin is most likely to be fair. We find the posterior distribution is given by:

We make less of a strong conclusion in this case - our data suggests it is more probable that the coin has a bias, but it is not conclusive.

Of course, if we increase the amount of data to say 1000 tosses, both frameworks give similar results.

So which method should we pick? In general:

- The frequentist framework is applicable when the number of data points n is much larger than the number of parameters d which we wish to estimate, $n \gg d$.
- The Bayesian framework is applicable for all numbers of data points n, regardless of the number of parameters.

The number of tunable parameters in modern machine learning models is comparable with the sizes of the training data - therefore in these cases, the Bayesian framework may also be useful.

The frequentist framework is a limit case of the Bayesian one:

$$\lim_{n/d \to \infty} p(\theta | x_1, ..., x_n) = \delta(\theta - \theta_{\mathsf{ML}}).$$

In particular, the Bayesian framework just provides an alternative point of view, it *does not* contradict or deny the frequentist framework. They agree completely in the case of infinite data.

There are several advantages of the Bayesian framework to bear in mind:

- We can encode our *prior knowledge* (if we have some) or *desired properties* of the final solution into a prior distribution.
- The prior can act as a form of regularisation. If we have small amounts of data, the prior helps us to overtrain (as we saw in the example with coin-tossing above).
- Additionally to the point estimate of θ, the posterior distribution in Bayesian inference contains information about the uncertainty of the estimate.

18.2 Application of the Bayesian framework to machine learning models

Consider a machine learning task where we have observed variables x (*features*) and a set of hidden/latent variables y (*class labels*, for example). Suppose that our model has parameters θ (e.g. the weights of a linear model).

There are two main types of machine learning models: *generative* and *discriminative*.

Definition 18.6: In a *generative model* we try to model the joint distribution $p(x, y, \theta) = p(x, y|\theta)p(\theta)$. Such models can be used to generate new objects, i.e. new pairs (x, y). Examples of such tasks include generation of text, speech and images.

This may be quite difficult to train, since the observed space is usually much more complicated than the hidden one.

Definition 18.7: In a *discriminative model* we try to model the distribution $p(y, \theta|x)$. We will not be able to generate new objects; we need x as an input. This usually assumes that the prior over θ does not depend on x:

$$p(y, \theta|x) = p(y|x, \theta)p(\theta).$$

Examples include classification and regression tasks (here the hidden space is much simpler than the observed one) and machine translation (where the hidden and observed spaces have equal complexity).

We will focus mostly on discriminative models.

In order to train a Bayesian machine learning model, we start with some given training data $(X_{\text{tr}}, Y_{\text{tr}})$ and a discriminative model $p(y, \theta|x)$ (by a model, we mean that we choose the form of the distribution).

- **Training stage.** At the training stage, we use Bayesian inference over θ:

$$p(\theta|X_{\text{tr}}, Y_{\text{tr}}) = \frac{p(Y_{\text{tr}}|X_{\text{tr}}, \theta)p(\theta)}{\int p(Y_{\text{tr}}|X_{\text{tr}}, \theta)p(\theta)\, d\theta}. \tag{1}$$

- **Result.** Once we have obtained our result, we have an 'ensemble' of algorithms rather than a single one θ_{ML}. By this, we mean we have a model for each value of θ, with probability described by the posterior distribution.

  The ensemble usually outperforms the single best model. The posterior captures all dependencies from the training data that the model could extract, and could be used as a new prior later (for example, if we acquired new data).

- **Testing stage.** We now have a posterior distribution $p(\theta|X_{\text{tr}}, Y_{\text{tr}})$. Suppose we are given a new data point x; we need to compute the predictive distribution on its hidden value y.

  We ensemble with respect to the posterior over the parameters θ:

$$p(y|x, X_{\text{tr}}, Y_{\text{tr}}) = \int p(y|x, \theta)p(\theta|X_{\text{tr}}, Y_{\text{tr}})\, d\theta. \tag{2}$$

We now see that equations (1) and (2) are very important in the Bayesian model framework. However, the integrals present in (1) and (2) may be intractable - it is important to ask when the integrals can be performed, and what we can do when they cannot.

18.3 Full Bayesian inference

In the previous section, we established that in order to apply Bayesian methods to machine learning we need to consider the integrals:

$$\int p(Y_{\text{tr}}|X_{\text{tr}},\theta)p(\theta)\,d\theta, \qquad \int p(y|x,\theta)p(\theta|X_{\text{tr}},Y_{\text{tr}})\,d\theta.$$

In the case that these integrals are tractable, we say that we can use *full Bayesian inference* - we don't use any approximate methods to complement Bayesian inference.

We start by defining *conjugacy* of distributions:

Definition 18.8: The distributions $p(\theta)$ and $p(x|\theta)$ are called *conjugate* if $p(\theta|x)$ belongs to the same parametric family as $p(\theta)$:

$$p(\theta) \in \mathcal{A}(\alpha), \quad p(x|\theta) \in \mathcal{B}(\theta) \qquad \Rightarrow \qquad p(\theta|x) \in \mathcal{A}(\alpha').$$

For example, $p(\theta)$ and $p(\theta|x)$ could both be normal distributions, but with different mean and variance.

The intuition for this definition is Bayes' theorem:

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{\int p(x|\theta)p(\theta)\,d\theta}.$$

If $p(x|\theta)$ and $p(\theta)$ are conjugate, then $p(\theta|x)$ belongs to some specific parametric family of distributions, the same as the prior. In particular, we can compute the normalisation constant for this distribution - indeed, this is what the integral on the denominator is! This is possible for standard distributions - just look at Wikipedia. Hence all we need to do is to compute α', the parameter describing where $p(\theta|x)$ is in the parametric family.

The same is true at the testing stage, since the integral takes the same form (just replacing $p(Y_{\text{tr}}|X_{\text{tr}},\theta)$ with $p(y|x,\theta)$ and $p(\theta)$ with $p(\theta|X_{\text{tr}},Y_{\text{tr}})$. We conclude that:

Both of our integrals are tractable if the prior and likelihood are conjugate.

Full Bayesian inference is very useful because we can use analytical formulas for both the training and testing stages. However, it makes strong assumptions on the model - *conjugacy of the prior and likelihood*. We must choose a conjugate prior, and this is only possible for simple models (a conjugate prior is typically not flexible enough in most cases).

Example 18.9: Consider coin-tossing again, where our coin may be biased. The task is to estimate a probability θ of landing heads up. Suppose we are given data $X = (x_1, ..., x_n)$ where $x \in \{0, 1\}$ (with heads 0 and tails 1).

Our probabilistic model is given by $p(x,\theta) = p(x|\theta)p(\theta)$ for some prior distribution $p(\theta)$. The likelihood distribution is a *Bernoulli distribution*:

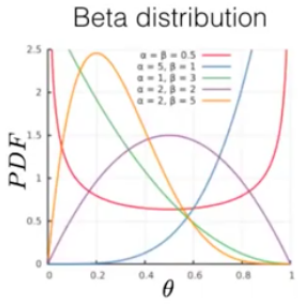
$$p(x|\theta) = \theta^x(1-\theta)^{1-x}.$$

How should we choose a prior for our probabilistic model? Since we are trying to use full Bayesian inference, we should try to use a conjugate prior.

First note that the prior should have domain $\theta \in [0, 1]$ since θ is a probability. Next, the prior should include our knowledge that the coin is most likely fair (from everyday experience). Finally, our prior should be conjugate to the Bernoulli distribution. The *beta distribution* matches all of these requirements:

$$\text{Beta}(\theta|a,b) = \frac{1}{B(a,b)}\theta^{a-1}(1-\theta)^{b-1},$$

where B is the beta function.



Choosing $a = b$ gives a symmetric distribution as required. However, if we think the coin is most likely unfair, we could choose $a \neq b$.

Let's check that our likelihood and prior are conjugate. Recall that:

$$p(x|\theta) = \theta^x(1-\theta)^{1-x}, \qquad p(\theta) = \frac{1}{B(a,b)}\theta^{a-1}(1-\theta)^{b-1}.$$

We need to check that the prior and posterior lie in the same parametric family. We note that for some normalisation constant C, we have:

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{C} = \frac{1}{CB(a,b)}\theta^{x+a-1}(1-\theta)^{x+b}$$

So for an appropriate choice of the constant C, this is a beta distribution with parameters $x + a, x + b + 1$. Thus we have conjugacy as required.

We can now compute the posterior distribution on θ, given a dataset X. We have, for some normalisations Z, Z':

$$p(\theta|X) = \frac{1}{Z}p(X|\theta)p(\theta) = \frac{1}{Z}\left[\prod_{i=1}^{n} p(x_i|\theta)\right]p(\theta) = \frac{1}{Z}\left[\prod_{i=1}^{n}\theta^{x_i}(1-\theta)^{1-x_i}\right]\frac{1}{B(a,b)}\theta^{a-1}(1-\theta)^{b-1}$$

$$= \frac{1}{Z'}\theta^{a+\sum_{i=1}^{n}x_i-1}(1-\theta)^{b+n-\sum_{i=1}^{n}x_i-1}.$$

This is a beta distribution $\mathrm{Beta}(\theta|a', b')$ where:

$$a' = a + \sum_{i=1}^{n} x_i, \qquad b' = b + n - \sum_{i=1}^{n} x_i.$$

In practice, we don't need to compute the normalisation constants C, Z, Z' above (although note that they are easy to compute, for example:

$$1 = \frac{1}{Z'}\int_0^1 \theta^{a'-1}(1-\theta)^{b'-1} = \frac{B(a',b')}{Z'} \qquad \Rightarrow \qquad Z' = B(a',b'). \qquad)$$

18.4 Maximum a posteriori estimation

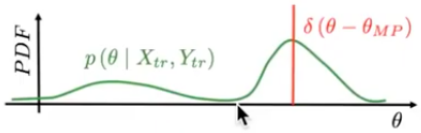
In the case where we do not have conjugacy, we need to make some approximations: a technique that we can use is *maximum a posteriori estimation*. We approximate the posterior distribution with a delta function in θ_{MP}, where:

$$\theta_{\text{MP}} = \operatorname{argmax} p(\theta|X_{\text{tr}}, Y_{\text{tr}}) = \operatorname{argmax} p(Y_{\text{tr}}|X_{\text{tr}}, \theta)p(\theta).$$

At the testing stage, the integral is rendered tractable:

$$p(y|x, X_{\text{tr}}, Y_{\text{tr}}) = \int p(y|x, \theta)p(\theta|X_{\text{tr}}, Y_{\text{tr}})\, d\theta \approx p(y|x, \theta_{\text{MP}}).$$

However, this approximation might be quite poor, e.g.



Note that whilst we are using a point estimate for maximum a posteriori estimation, this is *not* the same as θ_{ML}. It still uses some information from the prior distribution $p(\theta)$.

In summary, we have the following methods:

Approximation		Inference
Exact	$p(\theta \mid x)$	Full Bayesian inference
More advanced techniques		
Delta function	$p(\theta \mid x) \approx \delta(\theta - \theta_{MP})$	MP inference
No prior	θ_{ML}	MLE

18.5 Approximate Bayesian inference

We continue with our discussion of Bayesian inference in the approximate case. We now wish to improve on the simple method of maximum a posteriori estimation (MP estimation) described in the previous section.

There are two main approximate inference methods for Bayesian inference. Consider a probabilistic model $p(x, \theta) = p(x|\theta)p(\theta)$, and suppose we wish to compute $p(\theta|x)$. In the case we do not have conjugacy, we estimate:

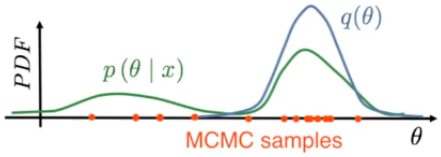
- **Variational inference.** We estimate the posterior distribution by approximating $p(\theta|x) \approx q(\theta) \in \mathcal{Q}$ where $\mathcal{Q}$ is some parametric family of distributions.

  This method is *biased* but is faster and more scalable.

- **Markov Chain Monte Carlo.** Sample from the unnormalised posterior $p(\theta|x)$. This is possible because we can compute the numerator in Bayes' theorem, but not the denominator.

  This method is *unbiased* but needs a lot of samples.

A comparison of the two methods is given in the diagram below:



Let's discuss *variational inference* in more detail. In order to find a posterior approximation $p(\theta|x) \approx q(\theta) \in \mathcal{Q}$, we use the following criterion function:

$$F(q) := KL(q(\theta)||p(\theta)|x)),$$

where KL is the *Kullback-Leibler divergence* (a good mismatch measure between two distributions over the same domain). We minimise this function over $q(\theta) \in \mathcal{Q}$.

In detail:

Definition 18.10: The *Kullback-Leibler divergence* is defined by:

$$KL(q(\theta)||p(\theta|x)) = \int q(\theta) \log\left(\frac{q(\theta)}{p(\theta|x)}\right) d\theta.$$

It has the following properties:

Proposition 18.11: The Kullback-Leibler divergence has the following properties:

(i) $KL(q||p) \geq 0$

(ii) $KL(q||p) = 0$ if and only if $q = p$.

(iii) $KL(q||p) \neq KL(p||q)$ in general (i.e. it is asymmetric).

There are two difficulties with this minimisation task:

- We can't compute the posterior distribution $p(\theta|x)$ in the first place! Thus we need to rewrite our problem to remove $p(\theta|x)$ from it.
- How can we perform a minimisation with respect to a *distribution* $q(\theta) \in \mathcal{Q}$? We need to change the optimisation problem to an optimisation problem with respect to parameters instead.

First, to rewrite $KL(q(\theta)||p(\theta|x))$ we use some 'mathematical magic':

$$\begin{aligned}\log(p(x)) &= \int q(\theta)\log(p(x))\,d\theta = \int q(\theta)\log\left(\frac{p(x,\theta)}{p(\theta|x)}\right)\,d\theta \\ &= \int q(\theta)\log\left(\frac{p(x,\theta)q(\theta)}{p(\theta|x)q(\theta)}\right)\,d\theta \\ &= \int q(\theta)\log\left(\frac{p(x,\theta)}{q(\theta)}\right)\,d\theta + \int q(\theta)\log\left(\frac{q(\theta)}{p(\theta|x)}\right)\,d\theta \\ &= \mathcal{L}(q(\theta)) + KL(q(\theta)||p(\theta|x)).\end{aligned}$$

The second term is the KL-divergence. The first term is usually called the *evidence lower bound* (ELBO); note that the ELBO does not depend on the posterior distribution, only on the joint distribution, which we know how to compute.

Before we discuss how to apply this to change our optimisation function, let's mention why the ELBO is a *lower bound*. Recall Bayes' theorem tells us that:

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)} = \frac{p(x|\theta)p(\theta)}{\int p(x|\theta)p(\theta)\,d\theta} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}.$$

The denominator, given by $p(x)$, is called the *evidence*. It is the total probability of observing the data. Since the KL-divergence is non-negative, we have:

$$\log(p(x)) = \mathcal{L}(q(\theta)) + KL(q(\theta)||p(\theta|x)) \geq \mathcal{L}(q(\theta)),$$

which explains why the ELBO is a lower bound for the (logarithm of the) evidence.

Now let's return to the optimisation problem. In the equation $\log(p(x)) = \mathcal{L}(q(\theta))$ and $KL(q(\theta)||p(\theta|x))$, we note that $\log(p(x))$ does not depend on q. In particular, changing q, we must have that $\log(p(x))$ is constant with respect to q, so decreasing KL-divergence increases the evidence lower bound. It follows that the following problem is equivalent to our original optimisation:

> *Maximise* $\mathcal{L}(q(\theta))$ with respect to $q(\theta) \in \mathcal{Q}$.

Before moving on, let's note that we can write:

$$\begin{aligned}\mathcal{L}(q(\theta)) &= \int q(\theta)\log\left(\frac{p(x,\theta)}{q(\theta)}\right)\,d\theta = \int q(\theta)\log\left(\frac{p(x|\theta)p(\theta)}{q(\theta)}\right)\,d\theta \\ &= \int q(\theta)\log(p(x|\theta))\,d\theta + \int q(\theta)\log\left(\frac{p(\theta)}{q(\theta)}\right)\,d\theta \\ &= \mathbb{E}_{q(\theta)}\log(p(x|\theta)) - KL(q(\theta)||p(\theta)).\end{aligned}$$

The first term here is called the *data term* and the second term is called the *regulariser*. The first term corresponds to the log likelihood of our data; the second term corresponds to the fact that we would like our posterior to be similar to our prior.

We still need to decide how to optimise with respect to a distribution. There is a simple solution: *parametric approximation*. We assume that $q(\theta) = q(\theta|\lambda)$ where λ is a set of parameters describing some parametric family of distributions.

This is slightly restrictive since:

- It may be too simple and insufficient to model the data.
- If it is complex enough, then there is no guarantee we can train it well enough to fit the data.

With this approximation, variational inference reduces to the parametric optimisation problem of *maximising*

$$\mathcal{L}(q(\theta|\lambda)) = \int q(\theta|\lambda) \log \left(\frac{p(x, \theta)}{q(\theta|\lambda} \right) \, d\theta$$

over λ. If we're able to compute derivatives of the ELBO with respect to λ, we can solve this problem using some numerical optimisation solver (e.g. gradient descent).

To summarise, we now have the four inference methods:

Approximation		Inference
Exact	$p(\theta \mid x)$	Full Bayesian inference
Parametric	$p(\theta \mid x) \approx q(\theta) = q(\theta \mid \lambda)$	Parametric VI
Delta function	$p(\theta \mid x) \approx \delta(\theta - \theta_{MP})$	MP inference
No prior	θ_{ML}	MLE

19 Bayesian linear regression

19.1 Review of linear regression

Recall that in the problem of linear regression, we are given a design matrix $X \in \mathbb{R}^{N \times d}$ of input data, and a tuple of target values $Y \in \mathbb{R}^N$, where N is the number of objects and d is the number of features. Our task is to model how Y depends on X.

Linear regression assumes the relationship is of the form $Y \approx Xw$ where w is a collection of weights. Each target value is given by $y_i = x_i^T w$, where x_i is each data point. To train the model, we wish to minimise the MSE:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i^T w - y_i)^2 = \frac{1}{N}||Xw - Y||^2.$$

over $w \in \mathbb{R}^d$. To compute predictions on a new object x_*, we evaluate $a(x_*) = x_*^T w$.

The objective function in this case is simply a quadratic function of the weights w. In most cases, there is a unique optimum which can be found analytically:

$$w_{\mathsf{ML}} = (X^T X)^{-1} X^T Y,$$

provided $\operatorname{rank}(X^T X) = d$. If $\operatorname{rank}(X^T X) < d$, there are infinitely many solutions (there are *flat directions*).

We can also add a *regularisation* term to the objective function:

$$L(w) = \frac{1}{N}||Xw - Y||^2 + \lambda||w||^2,$$

for $\lambda > 0$. In this case, $L(w)$ is always *strongly convex*, and there always exists a unique solution:

$$w_{\mathsf{MP}} = (X^T X + \lambda I)^{-1} X^T Y.$$

The use of regularisation also prevents overfitting by ensuring that the weights are close to zero.

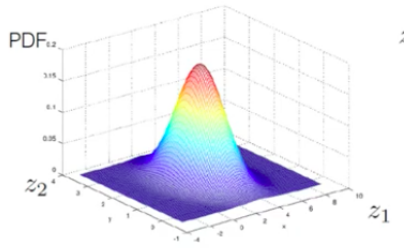
19.2 Bayesian approach to linear regression

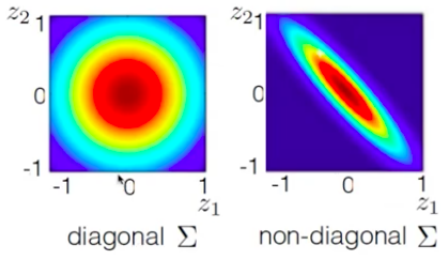
Now let's take a Bayesian view of linear regression. We start by recalling the definition of the multivariate normal distribution:

Definition 19.1: The *multivariate normal distribution* of dimension d has the probability density function:

$$\mathcal{N}(z|\mu, \Sigma) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2}(z-\mu)^T \Sigma^{-1}(z-\mu)\right),$$

where $z \in \mathbb{R}^d$, $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d \times d}$.





The vector μ is the *mean vector*. It reflects the location about which the distribution is concentrated. It also corresponds to the *mode* of the distribution.

The matrix Σ is the *covariance matrix*. If it is diagonal, the different components of z are uncorrelated. On the other hand, for non-diagonal Σ, the components of z are correlated (see above).

In Bayesian linear regression, we are given data $X \in \mathbb{R}^{N \times d}$ and $Y \in \mathbb{R}^N$ as follows. Our Bayesian model is:

$$p(Y, w|X) = p(Y|X, w)p(w),$$

where $p(Y|X, w)$ is the likelihood (how does the target Y depend on the input X?) and $p(w)$ is the prior (what weights w do we expect?).

We assume the likelihood to be a product of independent normal distributions:

$$p(Y|X, w) = \prod_{i=1}^{N} \mathcal{N}(y_i|x_i^T w, 1) = \mathcal{N}(Y|Xw, I).$$

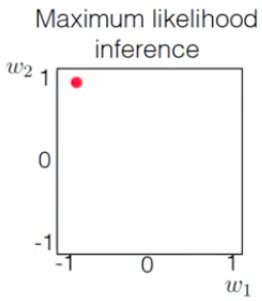
In particular, we can write it as a multivariate Gaussian distribution with covariance matrix I.

The prior is chosen to be *conjugate* with the likelihood. It turns out that the correct choice is a *normal distribution*:

$$p(w) = \mathcal{N}(w|0, \alpha I),$$

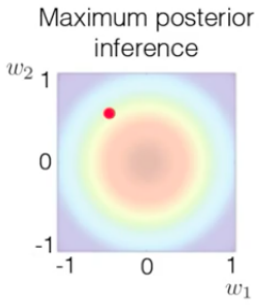
for $\alpha > 0$ (this is not tuned during training - it is a hyperparameter which must be determined during cross-validation). This works because the product of normal distributions remains normal.

The simplest option for training of Bayesian linear regression is to use *maximum likelihood inference*. In this case, we ignore the prior distribution and maximise $p(Y|X, w)$ with respect to the weights w. The result is a single point:



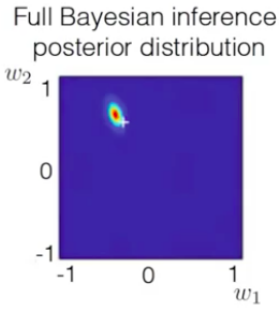
In fact, it can be shown that this is completely equivalent to usual linear regression.

If we instead take into account the prior distribution, we should maximise $p(Y|X,w)p(w)$ over w. This is *maximum posterior inference*, as we saw earlier. Again we get a point estimate, but it will be influenced by the prior:



It can be shown in this case that this is completely equivalent to usual linear regression including a regularisation parameter.

Finally, we can use *full Bayesian inference*. The posterior distribution takes the form:



In this case we have that the posterior distribution is given by Bayes' theorem, $p(w|X,Y) \propto p(Y|X,w)p(w)$.

Let's derive the posterior distribution in the case of full Bayesian inference. The likelihood is given by $P(Y|W,w) = \mathcal{N}(Y|Xw, I)$ and the prior is given by $p(w) = \mathcal{N}(w|0, \alpha I)$ for $\alpha > 0$, hence:

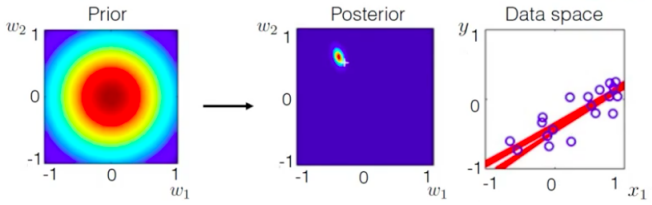
$$p(w|X,Y) = \text{Const} \exp\left(-\frac{1}{2}(Y - Xw)^T(Y - Xw)\right) \exp\left(-\frac{1}{2\alpha}w^T w\right)$$

$$= \text{Const} \exp\left(-\frac{1}{2}w^T\left(X^T X + \frac{1}{\alpha}I\right)w + w^T X^T Y\right).$$

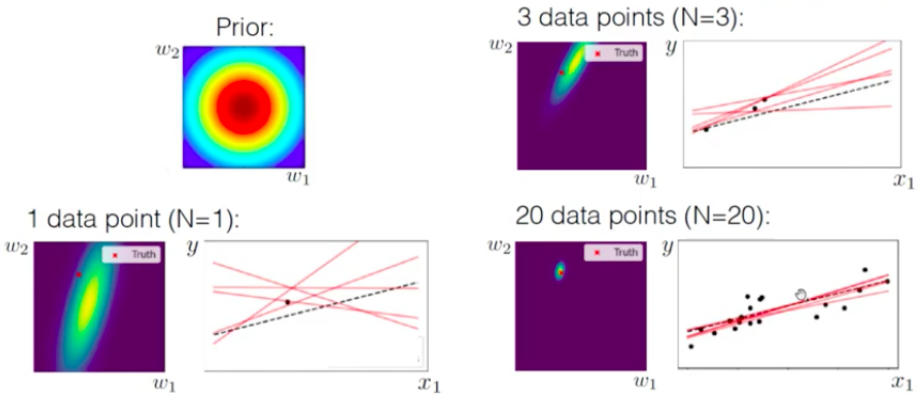
Completing the square in the exponent, we find that the posterior distribution is normal, $p(w|X,Y) = \mathcal{N}(w|w_{\text{MP}}, \Sigma)$ where:

$$w_{\text{MP}} = \left(X^T X + \frac{1}{\alpha}I\right)^{-1} X^T Y, \qquad \Sigma = X^T X + \frac{1}{\alpha}I.$$

The posterior is much narrower than the prior.



Increasing the amount of data improves things as expected:



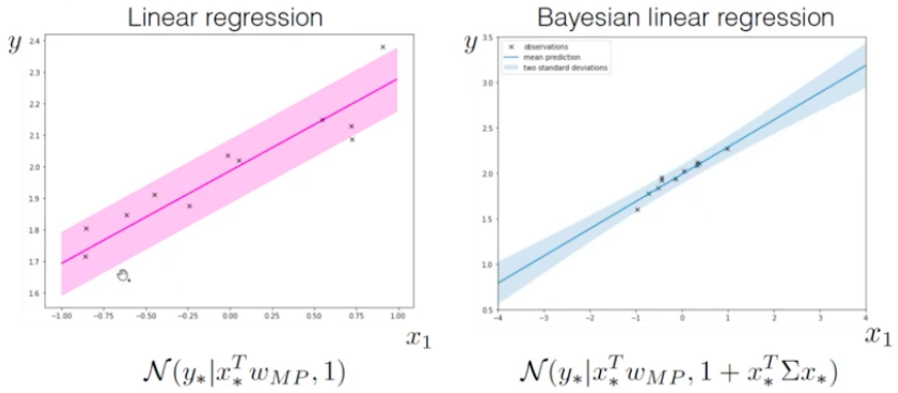
We have now completely discussed the training stage for linear Bayesian regression. For the testing stage, given a new object x_*, we should compute:

$$p(y_*|x_*, X, Y) = \int p(y_*|x_*, w)p(w|X, Y)\, dw = \mathbb{E}_{p(w|X,Y)}\left[p(y_*|x_*, w)\right].$$

It is not difficult to compute this. We find that:

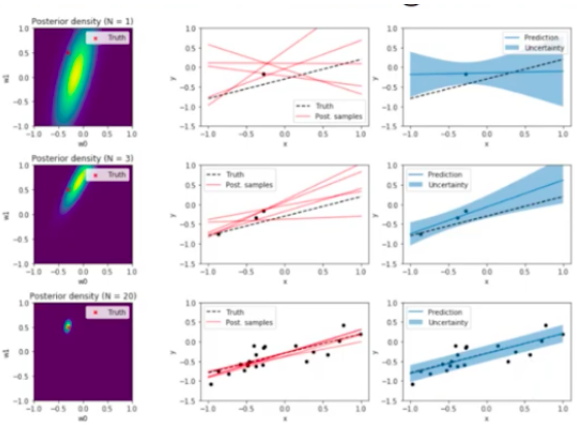
$$p(y_*|x_*, X, Y) = \mathcal{N}(y_*|x_*^T w_{\text{MP}}, 1 + x_*^T \Sigma x_*).$$

The variance of the prediction is the most interesting thing. It is $1 + x_*^T \Sigma x_*$, where the quadratic form $x_*^T \Sigma x_*$ is always non-negative. Thus the variance is always at least 1.



We see that Bayesian regression provides us with accurate error bands. They are smaller closer to the data and wider away from the data.

As we increase the amount of data, the variance reduces:


Posterior density (N = 1)
Posterior density (N = 3)
Posterior density (N = 20)
Truth
Post. samples
Prediction
Uncertainty
w0
w1
x
y

Part IV
Advanced optimisation methods

20 Introduction to black-box optimisation

20.1 Optimisation methods categorisation

Optimisation methods can be categorised into three main types:

- **Gradient methods:** For example stochastic gradient descent, Adam, and others. These have access to the value of the function $f(x)$ at a point and the *gradient* too, $\nabla f(x)$.
- **Second-order and quasi-Newton methods:** For example, Newton's method and BFGS. These additionally have access to the value of the Hessian H (e.g. Newton's method updates values according to $\mathbf{x} \leftarrow \mathbf{x} - H^{-1}\nabla f(\mathbf{x})$).
- **Black-box methods:** In this case, we can only access the values of the function $f(x)$ - we *don't* know the gradient. Examples include Bayesian optimisation, variational optimisation, evolutionary algorithms and many others.

In these lectures we will consider mainly Bayesian and variational optimisation.

Example 20.1: Examples of black-box optimisation problems include the following:

- Consider *car aerodynamics*. This is usually modelled by solving differential equations using finite difference methods. The gradients might be very expensive and potentially unstable (possible amplifying errors). Here it is reasonable to use black-box optimisation methods.
- Consider minimising:
$$\text{background}(\theta) = \mathop{\mathbb{E}}_{\text{event}} 1_{\{\text{muons} > 0 | \text{event}, \theta\}},$$
in an experiment where we wish to detect muons. In this case, we cannot compute the expectation exactly and have to use a Monte Carlo approximation. The expectation is being taken over a non-differentiable function, so it is unclear what we mean by the gradient in this case - thus black-box is necessary.
- Consider a chess-bot which wants to maximise:
$$\text{win rate}(\theta) = \mathop{\mathbb{E}}_{\text{event}} 1_{\{\text{win} \mid \text{opponent}, \theta\}}.$$
Again, only Monte Carlo estimates are available of the objective function. Furthermore, we have no estimate of the gradient.

To summarise: black-box optimisation methods can asses value of the objective in any point, $f(x)$, but have no other information available.

We can make some assumptions about the function: for example, we can assume bounds on the parameters, assumptions of (Lipschitz) continuity or smoothness, or even restricting the objective function to a family of functions.

Black-box optimisation is usually appleid to heavy objectives which are difficult to compute.

20.2 Simple black-box optimisation methods

One of the simplest black-box optimisation methods is a *reduction to a gradient method*. Here, we can approximate:

$$\frac{\partial}{\partial x} f(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

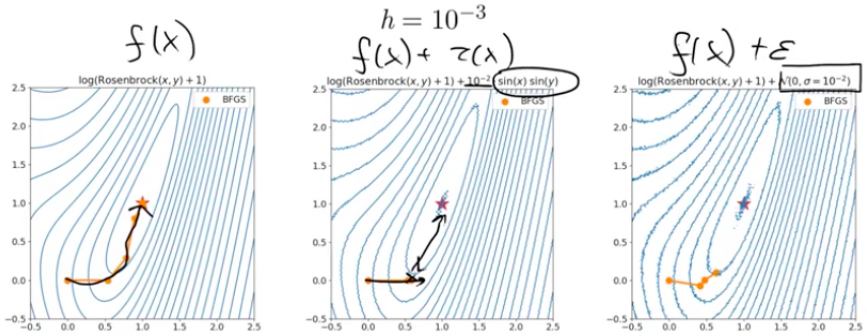
This requires $\mathcal{O}(d)$ evaluations of the function f, where d is the dimension of the feature space. For this reason, fast convergence algorithms like quasi-Newton algorithms are recommended.

The main drawback of these methods is the high sensitivity to noise. Consider the case where we have a noisy function obtained from Monte Carlo estimation:

$$\frac{f(x+h) + \epsilon_1 - f(x-h) - \epsilon_2}{2h} \approx \frac{\partial}{\partial x} f(x) + \mathcal{O}\left(\frac{\epsilon}{h}\right).$$

Here, ϵ_1, ϵ_2 are noise, and ϵ is a characteristic length scale for the noise. Note that for small h, the noise on the gradient can be very large. For large h, we could have an unreliable gradient since the function f could change a lot over a region of our step-size.

Here is a comparison of three objective functions with gradient estimated as above using a quasi-Newton method. The first is a 'pure' objective function with no noise. The second has small irregularities of order $1/100$. The third has a very small noise term.



The situation is improved if we simply use a gradient method rather than a quasi-Newton method, but this can take a very long time.

Another possible black-box optimisation method is *grid search*:

1. Make a grid.
2. Evaluate f at every point of the grid.
3. Search for the minimum.

The main drawback is this is extraordinarily slow in high dimensions. The main advantage is that it is a *global optimisation* rather than a local optimisation. Furthermore, it makes essentially no assumptions on the function f.

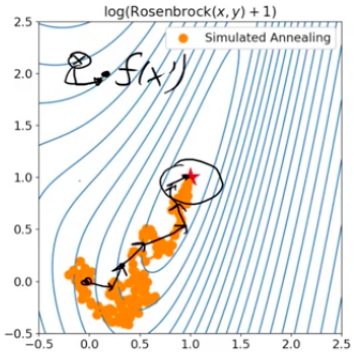
Another similar method is *random search*:

1. Draw uniformly multiple points (rather than making a grid).
2. Evaluate f at every point of the grid.
3. Search for the minimum.

This improve slightly on grid search, since we can insert some prior knowledge of the minimum by picking a different distribution centred on our assumed minimum (this can also be tackled by grid search though by using non-uniform bids).

A more complicated black-box optimisation method is called *simulated annealing*:

- For $i = 1, ..., N$, do the following steps:
  1. Set $x_i' = x_{i-1} + \epsilon \cdot \text{normal}(), y_i' = f(x_i'), T = T_0 \cdot (N - i + 1)/N$ and $P = \exp((y_{i-1} - y_i)/T)$.
  2. If $P > \text{uniform}(0, 1)$, then $x_i, y_i = x_i', y_i'$. Otherwise, $x_i, y_i = x_{i-1}, y_{i-1}$.
- The parameter T is called the *temperature*. It decreases with time which reduces the region which can be explored. The *initial temperature* is T_0, which is a hyperparameter.



This algorithm has lots of advantages. It is a *global optimisation*, since it can always leave local minima. Furthermore, it is very robust to the introduction of noise.

21 Variational optimisation

21.1 Variational bound

Gradient-based methods for black-box optimisation are not useful in the case that our objective function is non-differentiable or x is a discrete variable. The idea of *variational optimisation* is to replace our problem of minimising:

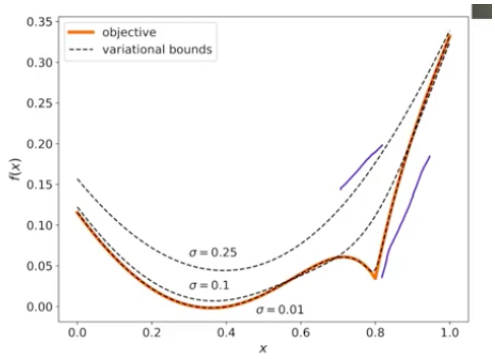
$$f(\theta)$$

over θ with a *differentiable surrogate*:

$$J(\psi) = \mathop{\mathbb{E}}_{\theta \sim P(\cdot|\psi)} f(\theta),$$

and minimise this over ψ. Here, $J(\psi)$ is called the *variational bound*, and $P(\cdot|\psi)$ is some *search distribution* that we choose.

Here are some examples of the variational bound and an objective function for different search distributions (chosen to be normal with different standard deviations σ):



For lower σ, we see that get much closer to the true objective function. For noisy functions this can be a problem.

The main property of the variational bound is that it is an *upper bound*:

$$J(\psi) \geq \min_\theta(f(\theta))$$

for all ψ. If $P(\cdot|\psi)$ is allowed to (nearly) collapse into the delta function, then $P(\cdot|\psi^*) \approx \delta(\theta^*)$.

The gradient of the variational bound can be computed as follows:

$$\begin{aligned}
\frac{\partial}{\partial \psi} J(\psi) &= \frac{\partial}{\partial \psi} \mathop{\mathbb{E}}_{\theta \sim P(\cdot|\psi)} f(\theta) \\
&= \frac{\partial}{\partial \psi} \int d\theta \, f(\theta) P(\theta|\psi) \\
&= \int d\theta \, f(\theta) \frac{\partial}{\partial \psi} P(\theta|\psi) \\
&= \int d\theta \, f(\theta) P(\theta|\psi) \frac{\partial}{\partial \psi} \log(P(\theta|\psi)) \\
&= \mathop{\mathbb{E}}_{\theta \sim P(\cdot|\psi)} f(\theta) \frac{\partial}{\partial \psi} \log(P(\theta|\psi)).
\end{aligned}$$

Hence we have established:

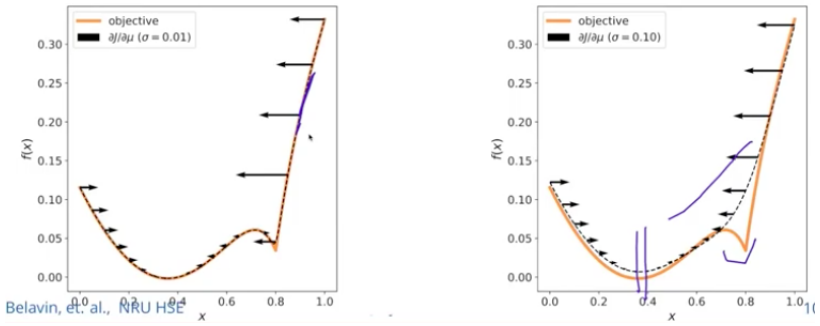
$$\nabla_\psi J(\psi) = \mathop{\mathbb{E}}_{\theta \sim P(\cdot|\psi)} f(\theta) \nabla_\psi \log(P(\theta|\psi)).$$

The main property is that $\nabla_\psi J(\psi)$ does not depend on $\nabla_\theta f(\theta)$. We can therefore apply the gradient to some non-differentiable functions, e.g. $f(x) = H(x)$, the Heaviside step-function. Choosing the search distribution to be $N(x; \psi, 1)$, we find:

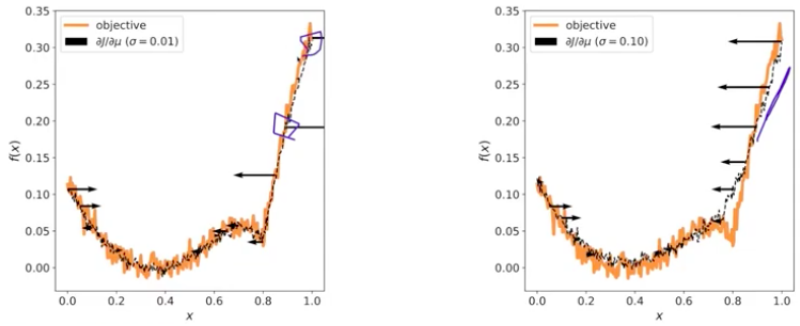
$$\nabla_\psi J = e^{-\psi^2/2}/\sqrt{2\pi}$$

in this case.

There is a tradeoff in the choice of parameters in the variational distribution. Higher variance gives smoother gradients, and the ability to avoid local minima. Lower variance gives results that are much closer to the original objective, but are much more prone to noise.



Higher variance can also negate sampling noise effects:



The only drawback of high variance is an increased *bias*. Fortunately, in practice we usually minimise over both *mean* and *variance* of the distribution. Therefore we can set σ large at the start of our optimisation, and let it tend to smaller values over time.

21.2 Stochastic gradient descent with variational optimisation

We can implement variational optimisation via the following algorithm:

1. Initialise the search distribution $P(\cdot|\psi)$.
2. While the algorithm has not converged, do the following:
   (i) Sample θ from $P(\cdot|\psi)$.
   (ii) Update $\nabla_\psi J(\psi) \leftarrow f(\theta)\nabla_\psi \log(P(\theta|\psi))$.
   (iii) Update $\psi \leftarrow \psi - \gamma\nabla_\psi J(\psi)$ for some learning rate γ.

We could use other gradient descent methods, e.g. Adam.

Variational optimisation allows usage of stochastic gradient descent methods for black-box problems, but is much slower in contrast to using analytical gradient descent.

Usually the search distribution is chosen to be simple, e.g. a normal distribution.

The dimensionality of the problem can be retained; at least 1 additional parameter is included to allow the search distribution to collapse (σ). $2n$ parameters are present for a normal distribution with a diagonal covariance, $O(n^2)$ for a full covariance matrix.

21.3 Discrete variables

Consider the following objective function where df/dx does not exist:

$$f(x) = \begin{cases} 0.45 & x = 0, \\ 0.53 & x = 1. \end{cases}$$

To optimise this function, we choose $p(x|\psi) = \text{Bernoulli}(x|\psi)$, the Bernoulli distribution with one search parameter. In this case:

$$J(\psi) = \mathop{\mathbb{E}}_{x\sim P(x|\psi)} f(x) = 0.53\psi + 0.45(1-\psi) = 0.08\psi + 0.45.$$

The gradient is 0.08. Voila! Even though $f(x)$ cannot be differentiated, we are still able to compute the gradient of the variational bound with respect to ψ and optimise the upper bound.

Neural networks with discrete random variables are a powerful technique for representing processes encountered in language modelling, attention mechanisms, and robotics control. Discrete representations are often more interpretable. However, networks with discrete variables are hard to train.

The drawback of variational optimisation in this case is that Monte-Carlo estimation of $\nabla_\psi \mathop{\mathbb{E}}_{x\sim P(x|\psi)} f(x)$ has high variance, and consequently, slow convergence. The variance scales linearly with the number of dimensions of the sample vector, making it especially challenging to use for categorical distributions.

One trick that can help us out is the *Gumbel maximum trick*. Given probabilities $\{\pi_i\}_{i=1}^n$ such that $\sum_i \pi_i = 1$, suppose that we want to sample from this discrete distribution.

Proposition 21.1: The following procedure:

$$z = \text{argmax}\{\log(\pi_i) + G_i\}, \qquad G_i = -\log(-\log(u_i)), \qquad u_i \sim U[0,1]$$

generates samples from the discrete probability distribution defined by $\{\pi_i\}_{i=1}^n$.

However, we can't backpropagate through z with repsect to π, but it's a first step.

To make this procedure differentiable, we need a differentiable approximation of the argmax function. One possible choice is the *softmax operation*:

$$y_i = \frac{\exp((\log(\pi_i) + G_i)/\tau)}{\sum_j \exp((\log(\pi_j) + G_j)/\tau)}, \qquad G_i = -\log(-\log(u_i)), \qquad u_i \sim U[0,1].$$

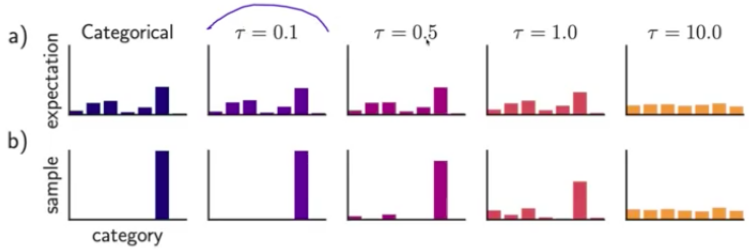
The variable τ is called the *temperature*. This distribution is called the *Gumbel softmax distribution*.

During the forward pass, we apply $z = \text{one-hot}(\text{argmax}\{y_i\})$. During the backward pass, we substitute:

$$\frac{dy}{d\pi} \to \frac{dz}{d\pi}.$$

In the limit $\tau \to 0$, it can be shown that these two gradients will be equal.

The temperature parameter controls how closely the samples approximate the discrete one-hot vectors.



Good practice is to start with a high temperature and gradually decrease it during the training.