CO395 - Introduction to Machine Learning (70050)

Module 1 (Introduction to ML)

Module 2 (Instance-based Learning + Decision Trees)

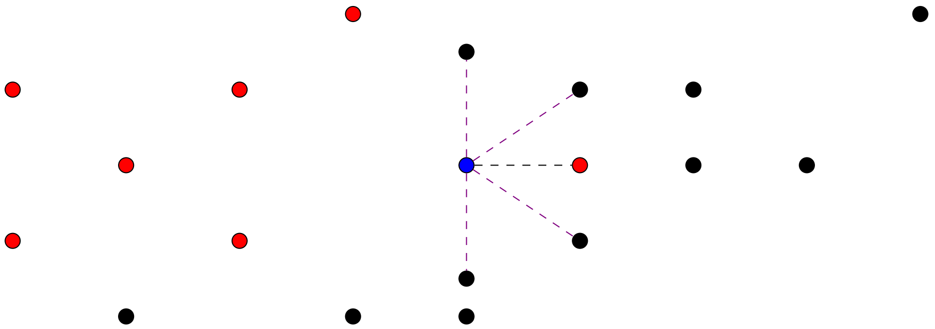
The **k Nearest Neighbours (k-NN)** classifier is classified as a **lazy learner**. A lazy learner stores all the training examples in the data set, and postpone any processing until a request is made (such as a prediction). On the other hand, **decision trees** are classified as a **eager learner**. An eager learner will attempt to construct a general target decision function, which is prepared prior to a query being made.

Classification with Instance-based Learning

The concept behind instance-based learning is that we will use samples in a training data set in order to make inference on a query.

The **Nearest Neighbour** classifier is a specific example, where it classifies a test instance to the label of the nearest training instance, where nearest is subject to some distance metric. This is a **non-parametric model**, which means it naturally emerges from the training set. Note in the example below, an issue with this is that it can be sensitive to noise, as it would classify the blue point to be red, as it is the closest instance in the training set, even though it's more likely to be black - it is very sensitive to noise, and can **overfit** to the training data.

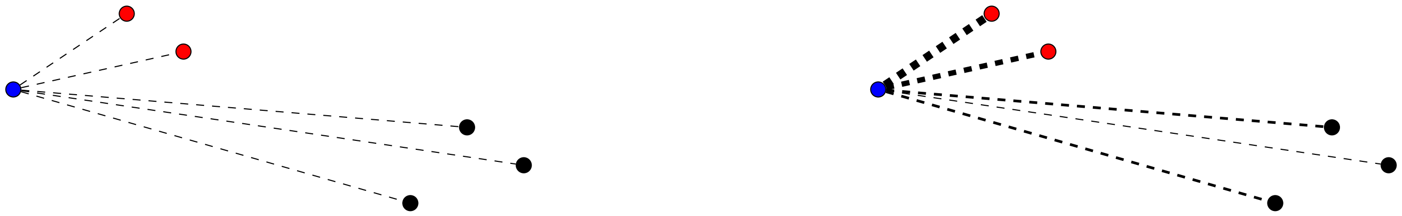
On the other hand, if we consider the **k Nearest Neighbours**, highlighted by the lines in violet, we get the class to be black, as we have 4 against 1. Usually, we need k to be odd, to ensure a winner for the decision task.



Increasing k will give the classifier have a smoother decision boundary (higher bias), and less sensitive to training data (lower variance). Choosing k is dependant on the dataset, normally with a validation dataset.

The distance metric can be defined in many different ways, including the ℓ_1, ℓ_2 and ℓ_∞-norms as seen in **CO233**. Other metrics exist such as the **Mahalanobis distance** for non-isotropic spaces, typically used for Gaussian distributions, or the **Hamming distance** for binary strings.

Another variation is the **Distance Weighted k-NN**. For example, we may not want to trust neighbours which are further away, such as in the example below.



The idea is that we add weights to each neighbour (depending on distance), typically a higher weight for closer neighbours. We then assign the class based on which class has the largest sum. This metric, $w^{(i)}$, is any measure favouring the votes of nearby neighbours, such as;

- inverse of distance
$$w^{(i)} = \frac{1}{d(x^{(i)}, x^{(q)})}$$
- Gaussian distribution
$$w^{(i)} = \frac{1}{\sqrt{2\pi}} e^{-\frac{d\left(x^{(i)}, x^{(q)}\right)^2}{2}}$$

The value of k is less important in the weighted case, as distant examples won't greatly affect classification. If $k = N$, where N is the size of the training set, it is a global method, otherwise it is a local method (only considering the samples close by). This method is also more robust to noisy training data, however it can be slow for large datasets.

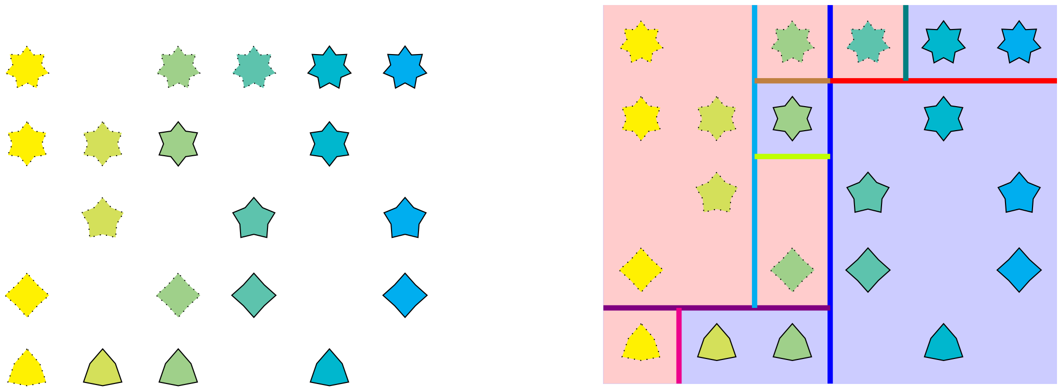
As this method relies on distance metrics, it may not work well if using all features in high dimensional spaces. If these features are irrelevant, instances in the same class may be far from each other. One solution to this is to weight features differently.

k-NN can also be used for regression, either by computing the mean value across k nearest neighbours (which leads to a very rough curve), or by using locally weighted regression, which computes the weighted mean value across k nearest neighbours, leading to a smoother curve.

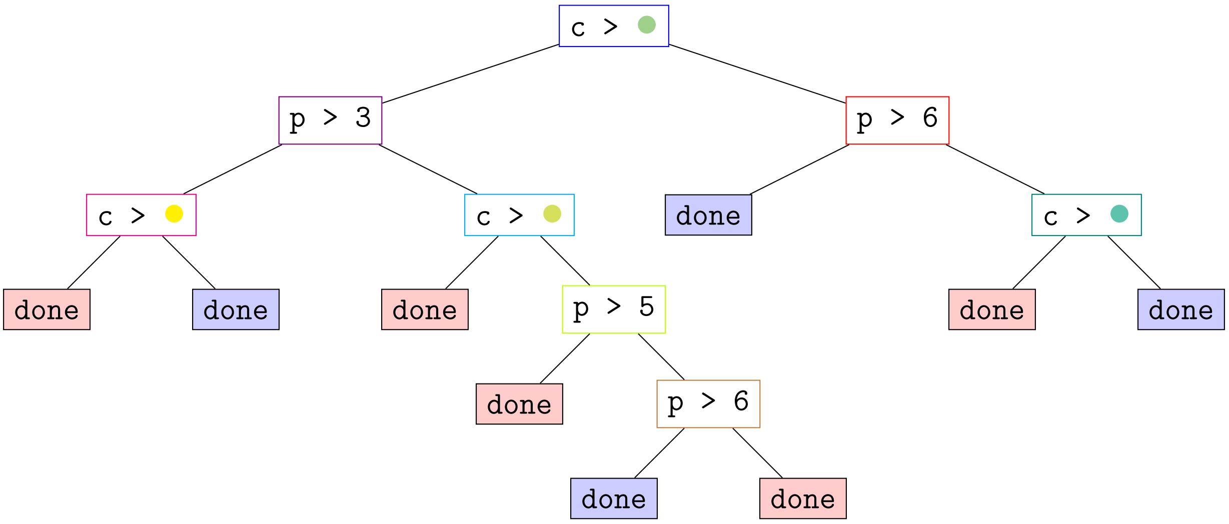
Classification with Decision Trees

Decisions trees are the principal of focusing on a subset or single feature of each sample and then make a decision whether it's true or false (for each feature), and repeat this process to finer decisions until we manage to classify the sample that we want to check.

In decision trees, we learn a succession of linear decision boundaries that we can use to eventually correctly classify samples.



In the example above, we repeatedly choose divisions that result in the fewest number of errors, until we are able to classify everything. This results in the following decision tree, when we are using the attributes of colour and number of points. For brevity, the left branch is the `false` branch, `p` means points, and `c` means colour.



Decision trees are a method of approximating discrete classification functions, by representing them as a tree (a set of if-then rules). The general algorithm (ID3) for constructing a decision tree is as follows;

1. search for the optimal splitting rule on training data
2. split data according to rule
3. repeat 1 and 2 on each subset until each subset is pure (only containing a single class)

How to select the 'optimal' split rule

Intuitively, we want to partition the datasets such that they are more pure than the original set. To do this, we have several metrics;

- **Information gain** ID3, C4.5

  quantifies the reduction of **entropy**

- **Gini impurity** CART

  if we randomly select a point in the feature space and randomly classify it according to the class label distribution, what is our probability of getting it incorrect?

- **Variance reduction** CART

  mostly used for regression trees, with a continuous target variable

To do this, we need to understand information entropy. Entropy is a measure of uncertainty of a random variable. It can also be seen as the average amount of information needed to define a random state / variable. If something has low entropy, it's predictable, and vice versa for high entropy.

Imagine we have two boxes, with something stored in one of the two, with an equal probability in each. To be fully certain, we need a single bit of information, if it's in the left box, the bit is `0`, otherwise (if it's in the right box), it's `1`. Similarly, if we have four boxes, with a uniform distribution, we would need 4 bits to encode the 4 states. In general;

$$\begin{aligned}
2^B &= K \text{ states} \\
B &= \log_2(K) \\
I(x) &= \log_2(K) && \text{amount of information to determine the state of a random variable} \\
P(x) &= \frac{1}{K} && \Rightarrow \\
K &= \frac{1}{K} && \Rightarrow \\
I(x) &= -\log_2(P(x))
\end{aligned}$$

As such, we can say;

$$I(x = \text{box}_1) = I(x = \text{box}_2) = I(x = \text{box}_3) = I(x = \text{box}_4) = -\log_2(P(x)) = 2 \text{ bits}$$

However, assume a non-uniform distribution, with the probabilities being 97%, 1%, 1%, and 1% respectively. If we were told it was in box 1, we do not get a lot of new information (low entropy); however if we were told it was in one of the other three, we high entropy (represents very important information).

$$\begin{aligned} I(x = \text{box}_1) &= -\log_2(0.97) \\ &\approx 0.0439 \text{ bits} \\ I(x = \text{box}_2) &= -\log_2(0.1) \\ &\approx 6.6439 \text{ bits} \end{aligned}$$

Entropy is defined as the average amount of information;

$$H(X) = -\sum_{k}^{K} P(x_k)\log_2(P(x_k))$$

In our example, we therefore have;

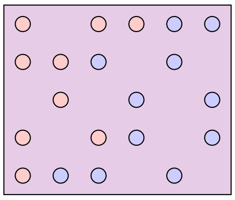
$$H(X) = -(0.97 \cdot \log_2(0.97) + 0.01 \cdot \log_2(0.01) + 0.01 \cdot \log_2(0.01) + 0.01 \cdot \log_2(0.01)) \approx 0.2419 \text{ bits}$$

We therefore need, on average, less information to know where the key is (compared to the uniform distribution).

For continuous entropy, we can use the probability density function $f(x)$ - this is imperfect (it can have negative values), but is still often used in Deep Learning.;

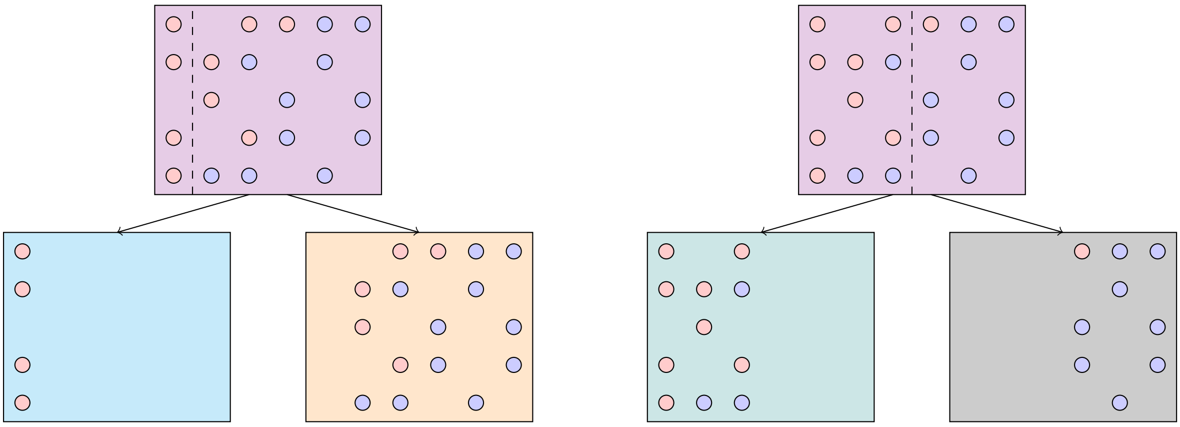
$$H(X) = -\int_x f(x)\log_2(f(x))\,\mathrm{d}x$$

Consider the following example;



$$\begin{aligned} P(\bullet) &= \frac{11}{20} \\ P(\bullet) &= \frac{9}{20} \\ H(\blacksquare) &= -\left(\frac{11}{20} \cdot \log_2\left(\frac{11}{20}\right) + \frac{9}{20} \cdot \log_2\left(\frac{9}{20}\right)\right) \\ &\approx 0.9928 \end{aligned}$$

An entropy value close to 1 would indicate a maximum amount of information needed.



$$
\begin{aligned}
H(\text{🟦}) &= 0 \\
H(\text{🟧}) &\approx 0.896 \\
H(\{\text{🟦}, \text{🟧}\}) &\approx \frac{4}{20} \cdot 0 + \frac{16}{20} \cdot 0.896 \\
&\approx 0.7168 \\
H(\text{🟪}) - H(\{\text{🟦}, \text{🟧}\}) &\approx 0.276 && \text{information gain}
\end{aligned}
$$

$$
\begin{aligned}
H(\text{🟩}) &\approx 0.8454 \\
H(\text{◼}) &\approx 0.5033 \\
H(\{\text{🟩}, \text{◼}\}) &\approx \frac{11}{20} \cdot 0.8454 + \frac{9}{20} \cdot 0.5033 \\
&\approx 0.6915 \\
H(\text{🟪}) - H(\{\text{🟩}, \text{◼}\}) &\approx 0.3013 && \text{information gain}
\end{aligned}
$$

As the second split has the larger information gain, that is the one we will end up selecting (and generally we want to split to maximise information gain). A formulation of this is as follows;

$$
\begin{aligned}
IG(\text{dataset}, \text{subsets}) &= H(\text{dataset}) - \sum_{S \in \text{subsets}} \frac{\mid S \mid}{\mid \text{dataset} \mid} H(S) \\
\mid \text{dataset} \mid &= \sum_{S \in \text{subsets}} \mid S \mid
\end{aligned}
$$

We can have the following types of input;

- **ordered values**
  - attribute and split point
  - for each attribute, sort the values and consider split points between two examples with different classes
- **categorical / symbolic values**
  - search for the most informative feature and create as many branches as there are values for this feature

Worked example for construction decision tree

Skipped, as this is basically done for the coursework.

Summary and other considerations with decision tree

Note that in general, if we have real-valued attributes, we will end up with a binary tree, with an attribute and threshold at each node. On the other hand, if we have categorical values, we can end up with a **multiway tree**.

Decision trees will **overfit**, like with many machine learning algorithms. This means the algorithm will take into account every sample in the dataset, to the point where it picks up the noise in the dataset. On the other hand, we have an underfitted algorithm, which has low variance and high bias (in contrast).

In decision trees, to deal with overfitting, we can employ the following strategies;

- **early stopping**

  basically stop the algorithm when a condition is met, rather than when the subset is pure (such as maximum depth of tree, or a minimum number of examples in the subset)

- **pruning** will be covered more next week

  1. identify internal nodes connected to only leaf nodes
  2. turn each into a leaf node (with the majority class label)
  3. if the validation accuracy of the pruned tree is greater, we keep it, and then repeat the process until no other pruning can improve the accuracy

To test this, we can reserve part of the dataset for training, and another part for validation. This is called **cross-validation**.

Another approach is to use a random forest. This involves training multiple decision trees, each with a subset of the training dataset, with a random subset of the features, and therefore each focuses on one subset of the features. We then take the majority vote by each of the decision trees as the final outcome.

Decision trees can also be used for regression (**regression trees**). Instead of class labels, each leaf node predicts a real-valued number.

Module 3 (Machine Learning Evaluation)

Evaluation Set-up

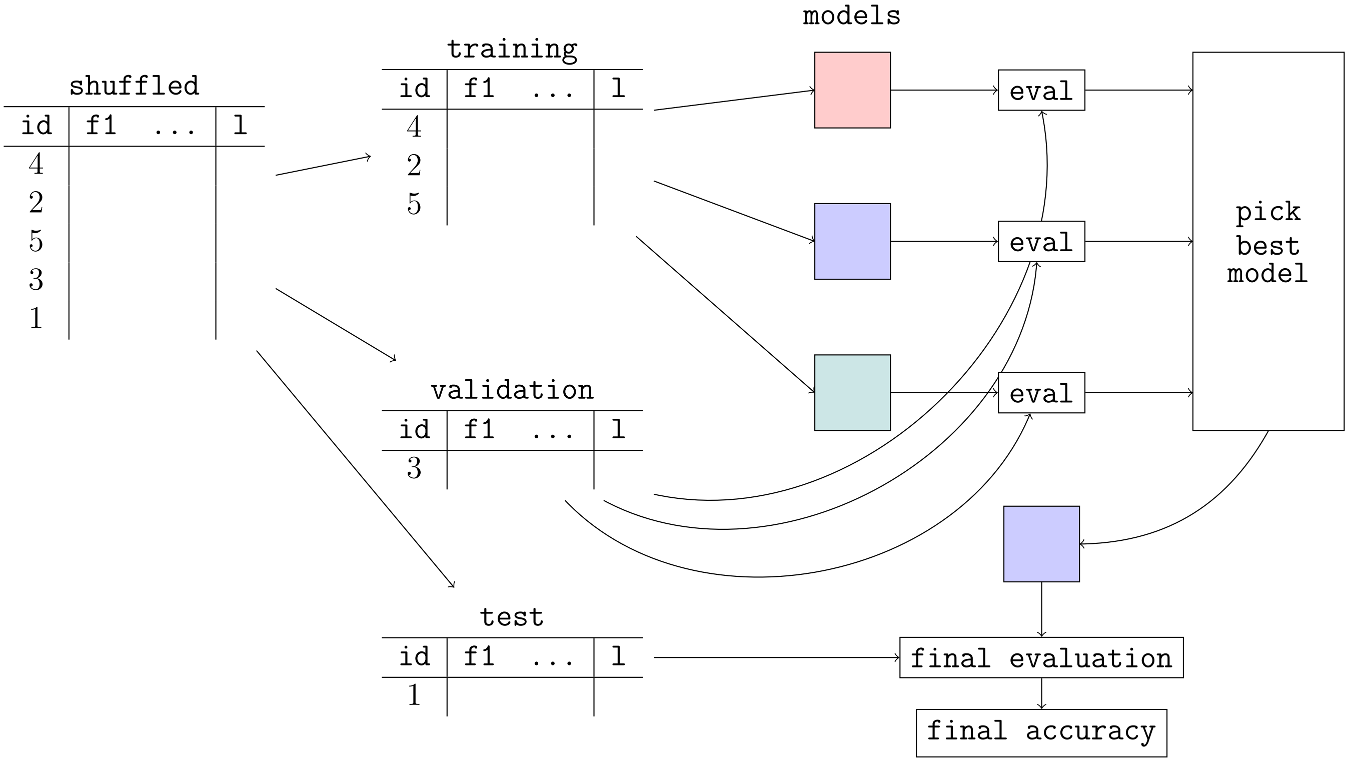
The goal is to create models and algorithms that can generalise to unseen data. We have good accuracy for the training set, since we trained the model for that, however we care more about the accuracy of unknown data.

To ensure meaningful evaluation, we need to split the training dataset from the test dataset (the test dataset should **never** be used to train, as it needs to simulate unknown data). This is done by first shuffling the dataset, and then splitting it into training and test datasets. The training dataset is used to train the model, and the test dataset is then fed into the trained model for final evaluation.

Hyperparameters are model parameters chosen before the training, such as the k value for k-NN algorithm. Our overall objective is to find the values that lead to best performance for unknown data. An incorrect approach for hyperparameter tuning is to try different values on the training dataset, and then select the ones that lead to the best accuracy on the test dataset. This is incorrect because we now use the test dataset as part of the training process, and therefore we cannot say that it is unknown.

As such, the correct approach is to split the dataset into 3; training, validation, and test. The splits for this are between somewhere between $60 : 20 : 20$ and $80 : 10 : 10$. Different hyperparameter values are attempted on the **training** set, and then the result with the best accuracy on the **validation** set is chosen. The final evaluation is still done on the **test** dataset.

We want to keep the classifier that leads to the maximum performance on the validation test. We can extend this even further by adding the validation dataset to the training dataset and training it on the model with the best parameters to give it more data. Once again, the final evaluation is still done with the test dataset.



Cross-validation

The idea of cross-validation is that the dataset can be divided into k (usually 10) equal splits. $k - 1$ of these folds can be used for training and validation, and the remaining split can be used for testing. This is done k times, each time testing on a different portion of the data, in which we test on all of the data (but notice we never train and test on the same data at the same time). The performance on all k held-out test sets can be averaged;

$$\text{global error estimate } = \frac{1}{N} \sum_{i=1}^{N} e_i$$

Note that this is used to evaluate an algorithm, not a particular model.

This method needs to be slightly modified when doing parameter tuning, in which we have the following options;

option 1: At each iteration, we use 1 fold for testing, 1 for validation, and the remaining $k - 2$ folds for training. However, this will give us a different set of optimal parameters in each fold.

option 2: Another approach is to do cross-validation within cross-validation. As before, we still separate 1 fold for testing, however we run another internal cross-validation over the remaining $k - 1$ folds to obtain the optimal hyperparameters. Once we obtain the best hyperparameters, we can then test it against the fold reserved for testing to obtain the final evaluation. This isn't always practical, as it requires a lot of computation for complex models.

When we go into production (not as common in academia), we may use all the remaining reserved test data for training as well (once we have the optimal parameters). However, this comes with the downside that we are no longer able to estimate the performance of the final trained model.

Performance Metrics

Once we have a model, we want to have a quantifiable way to judge the quality of a model against another. Consider the following results from the test dataset;

id	label	prediction
1	+	+
2	+	+
3	+	-
4	+	+
5	-	-
6	-	+
7	-	-
8	-	+

	class 1 (predicted)	class 2 (predicted)
class 1 (actual)	true positive (3)	false negative (1)
class 2 (actual)	false positive (2)	true negative (2)

This confusion matrix highlights the risk of each prediction - sometimes it can be more important to have fewer false negatives than fewer false positives (such as diagnosing a disease) It also allows for easy identification of confusion between classes (when one class is commonly mislabelled as another). Many other measures can be computed from the confusion matrix. In our example, we have two classes (positive and negative). The common measures are as follows;

- **accuracy** $\frac{TP+TN}{TP+TN+FP+FN}$

  This is simply the number of correctly classified examples divided by the total number of examples. The classification error is $1 -$ accuracy.

- **precision** $\frac{TP}{TP+FP}$

  This is the number of correctly classified positive examples divided by the total number of predicted positive examples. We can also think about it as;

  $$P(\texttt{positive} \mid \texttt{example classified as positive})$$

  A high precision implies that an example labelled as positive is actually positive (few false positives).

- **recall** $\frac{TP}{TP+FN}$

  This can be considered as the inverse of precision. It is the number of correctly classified positive examples divided by the number of actual positive examples. It can be thought of as

  $$P(\texttt{correctly classified as positive} \mid \texttt{actually positive})$$

  A high recall implies that the class is correctly recognised (therefore a small number of false negatives).

- **F-measure / F-score** $F_1 = \frac{2 \cdot \texttt{precision} \cdot \texttt{recall}}{\texttt{precision} + \texttt{recall}}$

  Sometimes it is useful to measure the performance of the classifier with a single number. More generally it can be written as (with more emphasis on precision for higher β);

  $$F_\beta = (1 + \beta^2) \cdot \frac{\texttt{precision} \cdot \texttt{recall}}{(\beta^2 \cdot \texttt{precision}) + \texttt{recall}}$$

For something to be high recall and low precision, most of the positive examples are correctly recognised, but with many false positives. On the other hand if something has low recall and high precision, we miss a lot of positive examples, but the ones that we predict as positive are more likely to be actually positive.

The macro-averaged recall is the mean of the recalls for all the classes. The same can be done for precision and F-measure. In the multi-class case, precision, recall, and F-measure are computed for each class separately (we define one class each time as being the positive class). Note that macro-averaging is done on the class level, and is the average of the metrics for each class. On the other hand, micro-averaging does it on the item level (adding the TP, FP, TN, FN values for each class before calculating the metrics). Note that micro-averaged P, R and F1 will be equal to accuracy.

Another measure is regression tasks, where a lower mean squared error (MSE) is better (where Y_i is a sample from the dataset and $\hat{Y}_i$ is the prediction from the model);

$$\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$$

However, we don't only care about accuracy, our models should be;

- **accurate**	makes correct predictions
- **fast**	fast to train and query
- **scalable**	works with large datasets
- **simple**	understandable and robust
- **interpretable**	can explain predictions

Imbalanced Datasets

In a balanced dataset, we have an equal number of positive and negative datapoints. However, this will not always be the case, and we may have an unbalanced dataset where classes are not equally represented. The accuracy goes down, as it tends to follow the majority class. Additionally, the precision may also go down for the minority class. Consider the following case;

	class 1 (predicted)	class 2 (predicted)
class 1 (actual)	700	300
class 2 (actual)	100	0

From this, we obtain the following metrics where the accuracy may be high, but class 2 is completely misclassified;

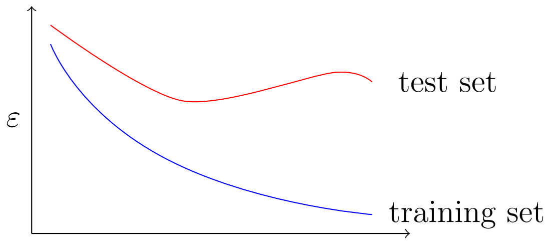
$$\begin{aligned} R(c1) &= 0.7 \\ P(c1) &= 0.875 \\ F_1(c1) &\approx 0.778 \\ R(c2) &= 0 \\ P(c2) &= 0 \\ F_1(c2) &= \texttt{N/A} \\ A &\approx 0.636 \end{aligned}$$

In conclusion, we need to look at different metrics, as well as the confusion matrix as a single metric may be misleading by itself.

We can normalise the confusion matrix by dividing each member of a row by the sum of that row (such that each row adds to one). We can also downsample the majority class, by picking less examples to get the two classes equal, or upsample the minority class by duplicating data. Neither will reflect how the model will generalise.

Overfitting

An overfitted model has good performance on training data, but poor generalisation to other data. On the other hand, underfitting has poor performance on the training data, as well as poor generalisation.



In the example above, it starts off with an underfitted model, and then ends up overfitted. The point where it's correct is just as the error of the test set begins to increase again.

Overfitting can occur under these scenarios (and how we could avoid it);

- model used is too complex (learns too many fine details)

  use the validation set to decide the complexity
- examples in the training set are not representative of all possible situations

  obtain more data
- learning is performed for too long (such as neural networks)

  stopping the training earlier (using the validation set to decide when)

Confidence Intervals

The amount of data used in our test set also affects our confidence of the performance evaluation. A 90% accuracy score on a test set with 10 samples is still less trustworthy than an 84% accuracy score on a test set with 10,000 samples.

We define the true error of the model h as the probability that it will misclassify a randomly drawn example x from distribution D;

$$error_D(h) \equiv P[f(x) \neq h(x)]$$

The **sample error** of the model h based on a data sample S is as follows;

$$\begin{aligned} n &= \text{number of samples} \\ \delta(f(x), h(x)) &= \begin{cases} 1 & f(x) \neq h(x) \\ 0 & f(x) = h(x) \end{cases} \\ error_S(h) &\equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x)) \end{aligned}$$

We can say an N% confidence interval for some parameter q is an interval with probability N to contain the true value of q. Given a sample S, with more than 30 examples;

$$error_S(h) \pm Z_N \underbrace{\sqrt{\frac{error_S(h) \cdot (1 - error_S(h))}{n}}}_{\text{est. standard deviation of sample error}}$$

Due to the n in the estimation of the standard deviation, if we have a very large n, we can obtain a very tight confidence interval. An example of this applied is as follows;

Emotion recognition results for 3 samples, using 156 training and 50 testing samples.

	attributes	number of classes	classifier	correctly classified
face	67 * 8	C4.5	78%	
body	140	6	BayesNet	90%

We want to classify the 95% confidence interval for this error ($Z_N = 1.96$)

$$\begin{aligned} error_S(h) &= 0.22 \\ n &= 50 \\ Z_N &= 1.96 \\ \text{interval} &= \left[0.22 - 1.96\sqrt{\frac{0.22 \cdot (1 - 0.22)}{50}}, 0.22 + 1.96\sqrt{\frac{0.22 \cdot (1 - 0.22)}{50}}\right] \\ &= [0.11, 0.33] \end{aligned}$$

Significance Testing

Statistical tests can tell us if the means of two sets are significantly different;

- **randomisation test**

  Randomly switch some predictions between two models and measure if the performance difference that we get is greater than or equal to the original difference.

- **two-sample T-test**

  This is used to estimate if two metrics from different populations are actually different. This has lower computational requirements and is easier to calculate.

- **paired T-test**

  Examining significance over multiple matched results, such as classification error over the same folds in cross-validation.

The **null hypothesis** (see **CO245**) is the hypothesis that the two algorithms / models perform the same and the differences are only due to sampling error. These tests return a **p-value**, which is the probability of obtaining the differences we see, assuming the null hypothesis is correct. A small p-value implies that we can be more confident that one system is actually different.

We consider a performance difference to be **statistically significant** of $p < 0.05$. However, $p > 0.05$ does not mean the algorithms are similar, just that we cannot observe a statistical difference.

There's a fairly long bit on **P-hacking**, but not sure why. A way to protect against P-hacking is to use an adaptive p-value;

1. rank p-values from M experiments;

$$p_1 \leq p_2 \leq p_3 \leq \cdots \leq p_M$$

2. calculate the **Benjamini-Hochberg** critical value for each experiment;

$$z_i = 0.05\frac{i}{M}$$

3. significant results are the ones where the p-value is smaller than the critical value

Module 4 (Artificial Neural Networks I)

The Rise of Neural Networks

Artificial neural networks are a class of machine learning algorithms, similar to **kNN** or **decision trees**. They consist of connected neurons, normally optimised with **gradient descent**. On the other hand **deep learning** refers to the use of neural network models with multiple hidden layers (hence deep) - they are usually trained on larger datasets for longer periods of time.

Using neural network models, there was a large improvement in speech recognition. *AlphaGo* allows for board analysis to beat human players (an exhaustive search isn't feasible compared to chess). Another example is realistic text generation, as well as video editing (see *DeepFakes*) - automating what would take hours to do manually. The aforementioned example of video editing works by sharing an encoder, and feeding it to a decoder which specialises in generating a single face. To change a face, the encoded output from the first person is put into the decoder of the second. Another application is combining different information mediums (such as images and text) - for example generating descriptive outputs for a given image.

The theory for neural networks had already existed (perceptrons in 1958, backpropagation in 1964, convolutional neural networks and LSTMs in the 1990s). However, today there is more data for training, as well as improved methods for storing and managing data. Similarly, neural networks benefit from faster hardware for computation (especially GPUs, since the matrix computation can be easily parallelised on GPUs). Finally, better software (such as automatic differentiation libraries) reduces the amount of work required (compared to manual calculation).

Linear Regression

We can think of linear regression as a very simplified method of a neural network model. This is a form of **supervised learning**, where we assume we have a dataset of input and output pairs;

- **dataset** $\{\langle x^{(1)}, y^1\rangle, \langle x^{(N)}, y^N\rangle, \dots, \langle x^{(N)}, y^N\rangle\}$
- **input features** $X = \{x^{(1)}, x^{(2)}, \dots, x^{(N)}\}$
- **known (desired) outputs** $Y = \{y^{(1)}, y^{(2)}, \dots, y^{(N)}\}$

Our aim is to learn the mapping $f : X \to Y$, such that $\forall i \in [1, N]\ f(x^{(i)}) = y^{(i)}$. In linear regression, f is a linear function.

It's important to note the difference between continuous and discrete problems;

- **classification** desired labels are discrete
  - obtaining digit labels from handwriting
  - obtaining sentiment from text
- **regression** desired labels are continuous
  - obtaining price of a house depending on some features
  - determining stock price from data on a company

An example of simple linear regression, with one input variable can be modelled as $y = ax + b$. a denotes the slope, which controls the angle, b denotes the intercept / bias (which controls the height) - with respect to a graph with the standard axes.

Consider a dataset which contains GDP per capita (x) and enrolment rate (y). To find the best values for a and b in $\hat{y} = ax + b$, we must first determine what we mean by best.

Loss Function

A loss / cost function determines how well we are doing on our dataset, where a smaller value of E means our predictions are close to our real values. Consider the **sum-of-squares** loss function;

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{N}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}\left(ax^{(i)} + b - y^{(i)}\right)^2 \end{aligned}$$

Note that the division by two is optional, however it allows for easier differentiation since it cancels with the two that will come from the square. By using a squared function, we are allowing for small errors that are close to the actual value, but we want to penalise the model for large errors.

Updating Parameters with Derivatives

Working out the partial derivative of the previously mentioned loss function with respect to each of the parameters, we get the following;

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{N}\left(ax^{(i)} + b - y^{(i)}\right)^2 \\ \frac{\partial E}{\partial a} &= \frac{\partial}{\partial a}\frac{1}{2}\sum_{i=1}^{N}\left(ax^{(i)} + b - y^{(i)}\right)^2 \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}\sum_{i=1}^{N} \frac{\partial}{\partial a}\left(ax^{(i)} + b - y^{(i)}\right)^2 \\
&= \sum_{i=1}^{N} \left(ax^{(i)} + b - y^{(i)}\right) x^{(i)} \\
&= \sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right) x^{(i)} \\
\frac{\partial E}{\partial b} &= \frac{\partial}{\partial b}\frac{1}{2}\sum_{i=1}^{N} \left(ax^{(i)} + b - y^{(i)}\right)^2 \\
&= \frac{1}{2}\sum_{i=1}^{N} \frac{\partial}{\partial b}\left(ax^{(i)} + b - y^{(i)}\right)^2 \\
&= \sum_{i=1}^{N} \left(ax^{(i)} + b - y^{(i)}\right) \\
&= \sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)
\end{aligned}
$$

Gradient Descent

Gradient descent is the process of repeated updating our parameters by taking small steps (of learning rate / step size α) in the negative direction of the partial derivative (worked out above). Note that we use := to denote a reassignment (not equality) in our **update rules**.

$$
\begin{aligned}
a &:= a - \alpha\frac{\partial E}{\partial a} \\
&:= a - \alpha\sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right) x^{(i)} \\
b &:= b - \alpha\frac{\partial E}{\partial b} \\
&:= b - \alpha\sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)
\end{aligned}
$$

This can be implemented manually as follows;

```
X = data["GDP"].values
Y = data["Enrolment Rate"].values

a = 0.0
b = 0.0
learning_rate = 1e-11

for epoch in range(5): # an epoch is a single iteration
    update_a = 0.0
    update_b = 0.0
    error = 0.0
    for i in range(len(Y)):
        y_pred = a * X[i] + b
        update_a += (y_predicted - Y[i]) * X[i]
        update_b += (y_predicted - Y[i])
        error += np.square(y_predicted - Y[i])

    # Update rule for gradient descent
    a = a - learning_rate * update_a
    b = b - learning_rate * update_b

    # the lines above can be shortened to the following
    y_predicted = a * X + b
    a = a - learning_rate * ((y_predicted - Y) * X).sum()
    b = b - learning_rate * (y_predicted - Y).sum()
    rmse = np.sqrt(np.square(y_predicted - Y).mean())
```

It can often be more convenient to work with vector notation. The gradient is a vector of all the partial derivatives. For a function $f : \mathbb{R}^K \rightarrow \mathbb{R}$ (where there are K parameters) the gradient is;

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_K} \end{bmatrix}$$

Using this, there is an **analytical solution** for solving a single variable linear regression;

$$\begin{aligned} \boldsymbol{X} &= \begin{bmatrix} x^{(1)} & 1.0 \\ x^{(2)} & 1.0 \\ \vdots & \vdots \\ x^{(N)} & 1.0 \end{bmatrix} \\ \boldsymbol{y} &= \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} \\ \boldsymbol{\theta} &= \begin{bmatrix} a \\ b \end{bmatrix} \\ \nabla_\theta E(\boldsymbol{\theta}) &= \boldsymbol{X}^\top (\boldsymbol{X}\boldsymbol{\theta} - y) \\ &= 0 & \Rightarrow \\ \boldsymbol{\theta}^* &= (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \end{aligned}$$

While this requires no iteration to directly find the optimal parameter values, it's not great for large problems as the matrix inversion has cubic complexity. The analytical solution, presented here with *Scikit-Learn*, has another benefit over gradient descent;

```
from sklearn.linear_model import LinearRegression

model = LinearRegression(fit_intercept=True)
X = data["GDP"].values.reshape(-1, 1)
Y = data["Enrolment Rate"]
model.fit(X, Y)

mse = np.square(Y - model.predict(X)).mean()
```

The analytical method manages to find the **global minimum**, whereas gradient descent finds a **local minimum**.

Multiple Linear Regression

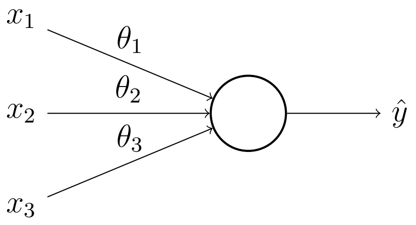
The previous example only took into account a single feature, which likely is insufficient for most complex problems. **multiple linear regression** considers many input features as follows, where each input feature has its own parameter as well as an extra parameter which acts as a bias;

$$y^{(i)} = \sum_{j=1}^{K} \theta_j x_j^{(i)} + \theta_{K+1}$$

Note that compared to before, when this is graphed on a two dimensional plane, it will be discontinuous, as it will exist in higher dimensions.

Artificial Neuron

In the example below, we have the **features** x_i and a corresponding weight (**parameter**) θ_i.



The output value of the neuron above is as follows, where g is the **activation function** (this is what makes it a neuron instead of just linear regression) and b is the **bias**;

$$\hat{y} = g(\underbrace{\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + b}_{\text{linear regression}})$$

Note that the bias is often implicit, since we can simply add an extra input feature that has a value of 1. Note that it's also possible to represent the weights as a vector $\boldsymbol{W}$, which simplifies the equation to be the following;

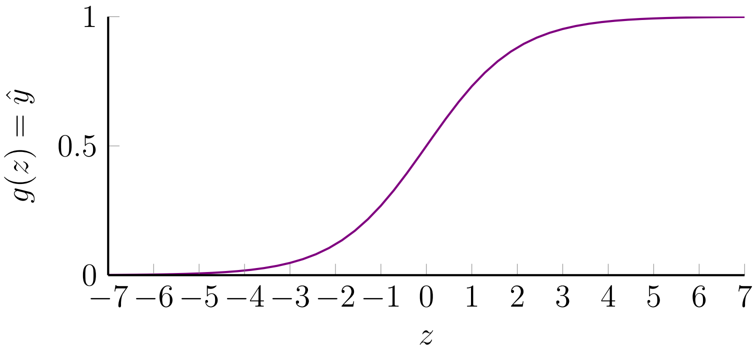
$$\hat{y} = g(\boldsymbol{W}^\top \boldsymbol{x})$$

Note that the part inside the activation function, concerning W and x, can be written in any way (depending on the representation), as long as the dimensions line up to give a single scalar result.

Logistic Activation Function

The **logistic function** is also known as the **sigmoid function**;

$$g(z) = \frac{1}{1 + e^{-z}}$$
$$z \in \mathbb{R}$$
$$\hat{y} \in [0, 1]$$



In logistic regression (which actually isn't regression), we can pass the output of linear regression through a logistic function, allowing us to get either 0 or 1 (binary classification). The model is optimised using gradient descent.

Perceptron

This is another algorithm for supervised binary classification (similarly has the two classes 0 and 1). It uses a threshold function as the activation function;

$$h(x) = f(\boldsymbol{W}^\top \boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{W}^\top \boldsymbol{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The learning rule is as follows;

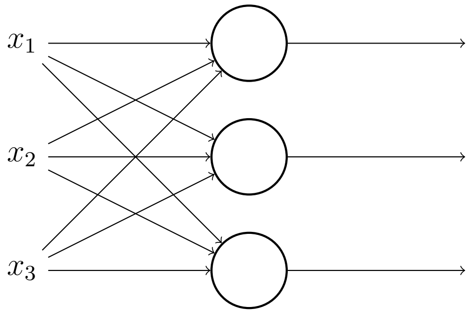
$$\theta_i \leftarrow \theta_i + \alpha(y - h(x))x_i$$

The reasoning is as follows; if the desired output (y) is equal to our prediction ($h(x)$), then the right hand side of the summation becomes 0, thus the parameter is unchanged - we don't fix it if it isn't broken. On the other hand, if the true value is 1 and the prediction is 0, we want to make $\boldsymbol{W}^\top \boldsymbol{x}$ bigger since the desired output is bigger than our prediction (therefore our prediction is too small to be set to 1 by the activation function). Since $y - h(x) = 1$ in this case, we increase θ_i if x_i is positive, and decrease it if it is negative. The reasoning holds the other way around, when the true value is 0 but we predict 1.

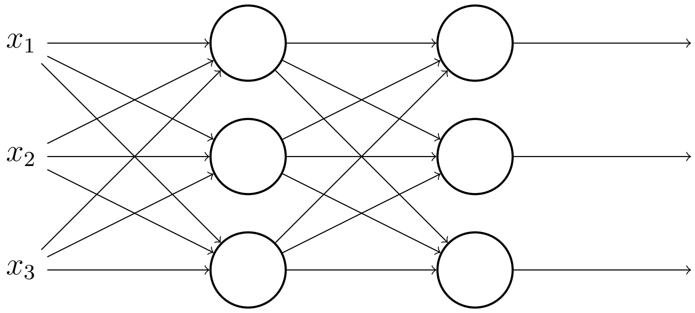
Any linearly separable function can be learnt with a perceptron, such as logical `OR` or `AND`, however something like `XOR` cannot be learnt. The activation function is also very sharp (and also not differentiable) so it's not used in most complex neural networks.

Connecting Neurons

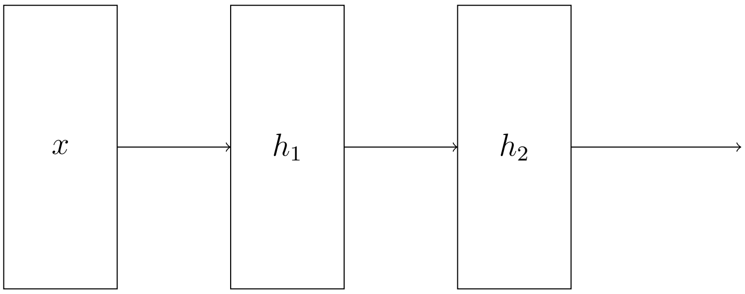
We can connect multiple neurons in parallel and consider each one as a **feature detector**;



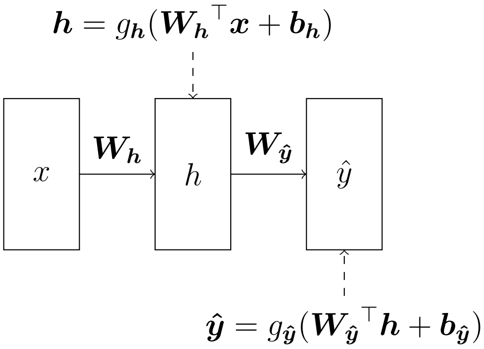
Similarly, we can connect neurons in sequence to learn from higher-order features (this is a multilayer perceptron, which isn't actually a perceptron);



A multilayer perceptron with a sufficient number of neurons can theoretically model an arbitrary function over an input. However, drawing this can be tedious (and isn't even feasible for large networks), we can represent each layer as a block, and each fully connected matrix of weights as an arrow (the diagram below represents the diagram above);



We typically refer to the first layer as the **input layer**, the final layer as the **output layer**, and everything in between the two as **hidden layers**. Consider a network with an input layer x, an output layer $\hat{y}$, and a hidden layer h;


$\boldsymbol{h} = g_{\boldsymbol{h}}(\boldsymbol{W_h}^\top \boldsymbol{x} + \boldsymbol{b_h})$
x
$\boldsymbol{W_h}$
h
$\boldsymbol{W_{\hat{y}}}$
$\hat{y}$
$\hat{\boldsymbol{y}} = g_{\hat{\boldsymbol{y}}}(\boldsymbol{W_{\hat{y}}}^\top \boldsymbol{h} + \boldsymbol{b_{\hat{y}}})$

When something doesn't work, it's often a good idea to verify the dimensions match up (possibly missing a transpose);

$$\begin{aligned}
\boldsymbol{x} &\in \mathbb{R}^{K\times 1}\\
\boldsymbol{h} &\in \mathbb{R}^{H\times 1}\\
\boldsymbol{W_h} &\in \mathbb{R}^{K\times H}\\
\boldsymbol{b_h} &\in \mathbb{R}^{H\times 1}\\
\hat{\boldsymbol{y}} &\in \mathbb{R}^{C\times 1}\\
\boldsymbol{W_{\hat{y}}} &\in \mathbb{R}^{H\times C}\\
\boldsymbol{b_{\hat{y}}} &\in \mathbb{R}^{C\times 1}
\end{aligned}$$

Learning Representations and Features

It's also important to note the difference between traditional pattern recognition and end-to-end training. For example with image recognition, traditional pattern recognition would require someone to manually craft a feature extractor, which then goes into a trainable classifier to give an output. This is in contrast to end-to-end training, where useful features are learnt from the data and trained with the classifier. This is useful as the feature extractor (typically at the lower levels of the network) can change to help the higher levels of the network make better decisions.

For example (at a very abstract level), a face detector would be structured as follows - note how the lower levels generally learn individual features, and the higher levels learn features of the features from lower levels);

- initially works on individual pixels
- detects edges
- detects components of a face
- detects full faces

Activation Functions

If we are able to use a linear model to capture all the features of our data, then we should do so (use the simplest model we can). However, more likely than not, our data will not be linearly separable, and therefore complex patterns cannot be captured with linear models. For multilayer networks, activation functions become more important.

- **linear activation** equivalent to having no activation function

$$f(x) = x$$

We cannot only use linear activation as multi-layer linear networks are equivalent to a single layer;

$$\begin{aligned} \boldsymbol{U} &= \boldsymbol{W_1}\boldsymbol{W_2} \\ \hat{\boldsymbol{y}} &= \boldsymbol{W_1}(\boldsymbol{W_2}\boldsymbol{x}) \\ &= \boldsymbol{U}\boldsymbol{x} && \text{becomes a single layer} \end{aligned}$$

- **sigmoid activation** smoothly compresses the output into $[0, 1]$

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

- **tanh activation** similar shape to sigmoid, but range in $[-1, 1]$

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **ReLU activation** rectified linear unit (linear in positive part)

$$f(x) = \text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{otherwise} \end{cases} = \max(0, x)$$

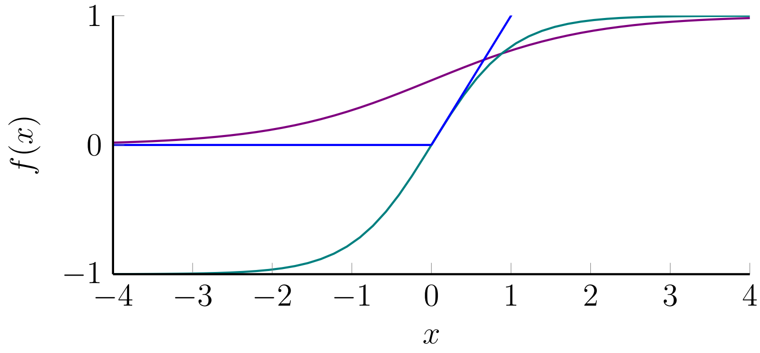
- **softmax activation**

scales inputs into a probability distribution (largest will be large, rest small), all output values sum to 1

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum\limits_k e^{z_k}}$$

This acts as a differentiable version of the max function, as it pushes the highest values close to 1 and the lower (remaining) values to 0. It identifies the value with the highest confidence and assigns more probability to it - very useful for image classification.

We can see the functions graphically as follows;



Note that most activation functions are applied **element-wise**, where each element is passed through the function independently (with the exception of softmax).

ReLU is used commonly for very deep networks, however tanh and sigmoid also work well and can be argued to be more robust. Those functions are more robust as ReLU is unbounded (if something is broken in the network, ReLU will give a large value which leads to more problems in the higher levels of the network). Therefore, the activation functions used are also a hyperparameter. The activation of the output layer should depend on the task (since it determines what the model can actually output);

• classifying into two classes	sigmoid or tanh
• predicting an unbounded score	linear
• predicting a probability distribution	softmax

Feedforward Network in PyTorch

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer_h = nn.Linear(10, 5) # input->hidden (input of size 10)
        self.layer_y = nn.Linear(5, 1) # hidden->output (hidden layer of size 5, gives 1 output)

    def forward(self, x):
        h = torch.tanh(self.layer_h(x)) # hidden layer with tanh activation
        y = torch.sigmoid(self.layer_y(h)) # output with sigmoid activation

net = Net()
input = torch.FloatTensor([x for x in range(10)]) # sample input
output = net(input) # execution
```

Module 5 (Artificial Neural Networks II)

Loss Functions

We need to define a loss function which we aim to minimise, a smaller loss function tells us that we are doing better on a particular task. To optimise a neural network, we want to iteratively update parameters with gradient descent (taking small steps in the negative direction of the loss function), aiming to reduce the loss function. The general formula is as follows;

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \frac{\partial E}{\partial \theta_i^{(t)}}$$

Loss Function for Regression

When we aim to predict a continuous variable, we often use linear activation (to avoid restricting the output). This is also often optimised with **mean squared error** (or quadratic / L2 loss).

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

If the value is 0, then we know that our predictions are equal to our target data.

Loss Function for Classification

In contrast, for classification the model needs to choose between discrete (categorical) options;

- **binary classification** one of two possible choices (e.g. disease diagnosis)
- **multi-class classification** an object can take one of many classes (e.g. detecting digits)
- **multi-label classification** an object can take multiple classes (e.g. detecting objects in an image)

In **cross-entropy**, we aim to maximise the likelihood of the network assigning the correct labels to all the inputs in our dataset;

$$\prod_{i=1}^{N} P(y^{(i)} \mid x^{(i)}; \theta)$$

Also assume that the examples are independent and identically distributed;

$$P(A \wedge B) = P(A)P(B)$$

For binary classification, we can consider the output of the network as a parameter of a **Bernoulli distribution**;

$$\prod_{i=1}^{N} \left(\hat{y}^{(i)}\right)^{y^{(i)}} \left(1 - \hat{y}^{(i)}\right)^{\left(1-y^{(i)}\right)}$$

If we always correctly assign all the probability to the correct label, we end up with an output of 1, and if we always incorrectly assign the probability we get 0.

However, dealing with multiplication of small values (since all of our values will be between 0 and 1) can lead to rounding errors in computers, we want to take logarithms. Since maximising the logarithm is identical;

$$\sum_{i=1}^{N} y^{(i)} \log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)$$

This can be turned into a loss (**binary cross-entropy**), which we aim to minimise. The division by N is optional, but it avoids the magnitude of the loss being affected by the number of datapoints;

$$L = -\frac{1}{N} \sum_{i=1}^{N} (\underbrace{y^{(i)} \log\left(\hat{y}^{(i)}\right)}_{\text{class 1}} + \underbrace{\left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)}_{\text{class 0}})$$

Notice that the above refers to two distinct classes - we can generalise this to multiple classes in **categorical cross-entropy**;

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_c^{(i)} \log\left(\hat{y}_c^{(i)}\right)$$

Here C is the set of possible classes, and $\hat{y}_c^{(i)}$ is the predicted probability of class c for datapoint i.

For multi-class classification, it's common to use softmax activation with categorical cross-entropy loss. Some examples are as follows;

problem	**type**	**output activation**	**loss**
future stock prices	regression	linear	MSE
stock prices up or down	binary	sigmoid	binary cross-entropy
speech recognition	multi-class	softmax	categorical cross-entropy
chemical properties	multi-label	sigmoid	binary cross-entropy

Note that for the final example, the activation is sigmoid for **each** individual output and we optimise each output with binary cross-entropy (such that each output acts as its own binary classifier for that property).

Batching

A common process is to combine the vectors of several datapoints into one matrix to improve speed and reduce noise. The speed improvement comes from GPUs being able to handle parallel workloads such as matrix multiplication well, whereas more time will be spent copying vectors to the GPU without batching. See the example below where datapoint 1 ($x_{1,i}$) and datapoint 2 ($x_{2,i}$) are multiplied by the same weight matrix;

$$\begin{bmatrix} x_{1,1} & x_{1,2} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{2,1} & x_{2,2} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} = \begin{bmatrix} x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} \end{bmatrix}$$

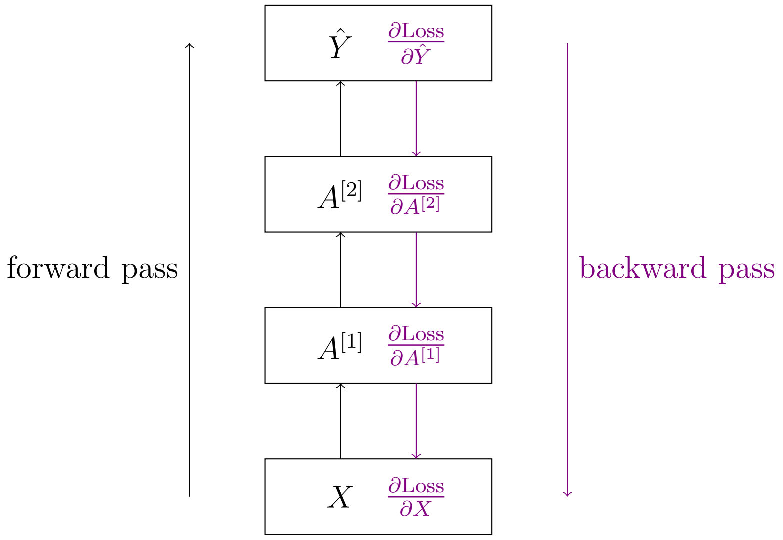
This method is applied to the majority of neural network operations, such as addition, subtraction as well as application of activation functions.

Forward and Backward Pass

The following process applies from the bottom up (X progresses to $\hat{Y}$);

$$\begin{aligned}
\text{Loss} &= L(Y, \hat{Y}) \\
\hat{Y} &= g_o\left(Z^{[3]}\right) \\
Z^{[3]} &= A^{[2]}W^{[3]} + B^{[3]} \\
A^{[2]} &= g_{h_2}\left(Z^{[2]}\right) \\
Z^{[2]} &= A^{[1]}W^{[2]} + B^{[2]} \\
A^{[1]} &= g_{h_1}\left(Z^{[1]}\right) \\
Z^{[1]} &= XW^{[1]} + B^{[1]} \\
X &\in \mathbb{R}^{N \times K} && N \text{ datapoints and } K \text{ features per datapoint}
\end{aligned}$$

The idea of backpropagation reduces the work needed by splitting up the workload for calculating partial derivatives. The partial derivative is obtained by using the partial derivative from the next layer (the next layer in the forward pass), which obtains it from the partial derivative of the loss function.



$\hat{Y}$ $\frac{\partial \text{Loss}}{\partial \hat{Y}}$
$A^{[2]}$ $\frac{\partial \text{Loss}}{\partial A^{[2]}}$
forward pass
backward pass
$A^{[1]}$ $\frac{\partial \text{Loss}}{\partial A^{[1]}}$
X $\frac{\partial \text{Loss}}{\partial X}$

By recalling the **chain rule** (for calculating derivatives of composite functions);

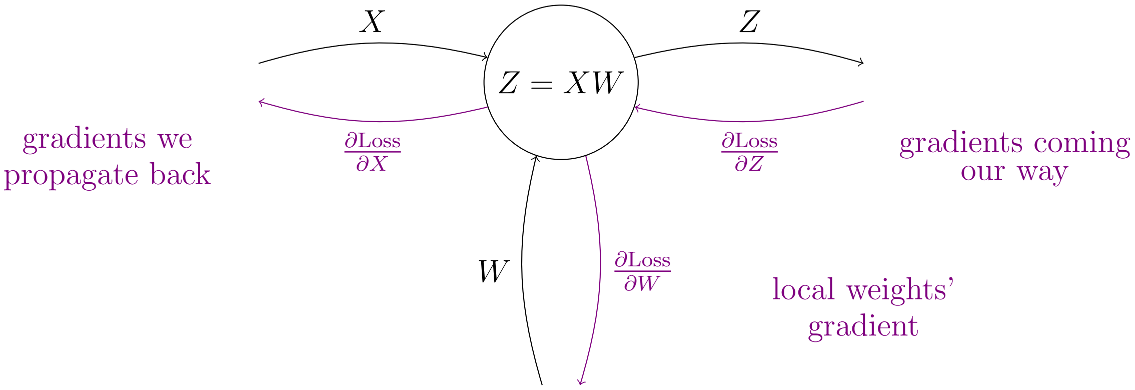
$$z = f(x)$$

$$
\begin{aligned}
y &= g(z) \\
\frac{\mathrm{d}y}{\mathrm{d}x} &= \frac{\mathrm{d}y}{\mathrm{d}z} \cdot \frac{\mathrm{d}z}{\mathrm{d}x}
\end{aligned}
$$

For example, assume we want to find the partial derivative with respect to the weight matrix $W^{[1]}$ (which we would want to do for gradient descent). We can break down our derivatives as follows;

$$
\begin{aligned}
\frac{\partial \text{Loss}}{\partial Z^{[1]}} &= \frac{\partial \text{Loss}}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \\
\frac{\partial \text{Loss}}{\partial W^{[1]}} &= \frac{\partial \text{Loss}}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\
&= \frac{\partial \text{Loss}}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z^{[3]}} \cdot \frac{\partial Z^{[3]}}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}
\end{aligned}
$$

We can see the partial derivatives in teal are the output of an activation function with respect to the inputs, whereas the ones in violet are the output of a linear transformation with respect to the input of a linear transformation. We cannot directly apply this on matrices, as they are 4D tensors.



X
$Z = XW$
Z
gradients we propagate back
$\frac{\partial \text{Loss}}{\partial X}$
$\frac{\partial \text{Loss}}{\partial Z}$
gradients coming our way
W
$\frac{\partial \text{Loss}}{\partial W}$
local weights' gradient

$$
\begin{aligned}
X &\in \mathbb{R}^{N \times D} \\
W &\in \mathbb{R}^{D \times M} \\
B &\in \mathbb{R}^{N \times M} \\
Z &\in \mathbb{R}^{N \times M} \\
Z &= XW + B
\end{aligned}
$$

consider a particular case, where;

$$
\begin{aligned}
N &= 2 \\
D &= 2 \\
M &= 3 \\
X &= \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} \\
W &= \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} \\
B &= \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{b} \end{bmatrix} && \text{same bias} \\
&= \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix} \\
Z &= \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}
\end{aligned}
$$

assume we already receive the following through backpropagation (notice it has the same shape as Z - generally the derivative of a scalar with respect to a matrix has the same shape as the matrix)

$$\frac{\partial \text{Loss}}{\partial Z} = \begin{bmatrix} \frac{\partial \text{Loss}}{\partial z_{1,1}} & \frac{\partial \text{Loss}}{\partial z_{1,2}} & \frac{\partial \text{Loss}}{\partial z_{1,3}} \\ \frac{\partial \text{Loss}}{\partial z_{2,1}} & \frac{\partial \text{Loss}}{\partial z_{2,2}} & \frac{\partial \text{Loss}}{\partial z_{2,3}} \end{bmatrix}$$

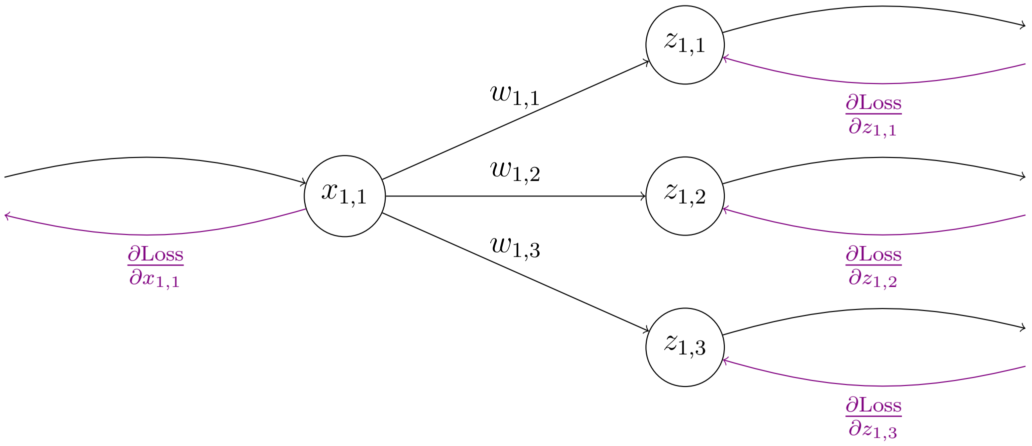
to update weights, we need to calculate the following;

$$\frac{\partial \text{Loss}}{\partial W} = \frac{\partial \text{Loss}}{\partial Z} \cdot \frac{\partial Z}{\partial W}$$
$$\frac{\partial \text{Loss}}{\partial b} = \frac{\partial \text{Loss}}{\partial Z} \cdot \frac{\partial Z}{\partial b}$$

the following is needed to pass the gradient on to the lower layer;

$$\frac{\partial \text{Loss}}{\partial X} = \frac{\partial \text{Loss}}{\partial Z} \cdot \frac{\partial Z}{\partial X}$$

Looking at calculating this for a single value, we can visualise it as follows;


$z_{1,1}$
$w_{1,1}$
$\frac{\partial \text{Loss}}{\partial z_{1,1}}$
$w_{1,2}$
$x_{1,1}$
$z_{1,2}$
$w_{1,3}$
$\frac{\partial \text{Loss}}{\partial x_{1,1}}$
$\frac{\partial \text{Loss}}{\partial z_{1,2}}$
$z_{1,3}$
$\frac{\partial \text{Loss}}{\partial z_{1,3}}$

This gives us the following result;

$$\frac{\partial L}{\partial x_{1,1}} = \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3}$$

Note that the chain rule is already applied here, as we can think can think of the weights as derivatives;

$$w_{1,1} = \frac{\partial z_{1,1}}{\partial x_{1,1}}$$

Therefore, for all of X, using the result above for a single element;

$$\begin{aligned} \frac{\partial \text{Loss}}{\partial X} &= \begin{bmatrix} \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3} & \frac{\partial L}{\partial z_{1,1}} w_{2,1} + \frac{\partial L}{\partial z_{1,2}} w_{2,2} + \frac{\partial L}{\partial z_{1,3}} w_{2,3} \\ \frac{\partial L}{\partial z_{2,1}} w_{1,1} + \frac{\partial L}{\partial z_{2,2}} w_{1,2} + \frac{\partial L}{\partial z_{2,3}} w_{1,3} & \frac{\partial L}{\partial z_{2,1}} w_{2,1} + \frac{\partial L}{\partial z_{2,2}} w_{2,2} + \frac{\partial L}{\partial z_{2,3}} w_{2,3} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial \text{Loss}}{\partial z_{1,1}} & \frac{\partial \text{Loss}}{\partial z_{1,2}} & \frac{\partial \text{Loss}}{\partial z_{1,3}} \\ \frac{\partial \text{Loss}}{\partial z_{2,1}} & \frac{\partial \text{Loss}}{\partial z_{2,2}} & \frac{\partial \text{Loss}}{\partial z_{2,3}} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{bmatrix} \\ &= \frac{\partial \text{Loss}}{\partial Z} W^\top \end{aligned}$$

A similar process can be applied to calculate the derivatives with respect to the weights;

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial z_{1,1}} x_{1,1} + \frac{\partial L}{\partial z_{2,1}} x_{2,1}$$

$$\begin{aligned}\frac{\partial \text{Loss}}{\partial W} &= \begin{bmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{bmatrix} \begin{bmatrix} \frac{\partial \text{Loss}}{\partial z_{1,1}} & \frac{\partial \text{Loss}}{\partial z_{1,2}} & \frac{\partial \text{Loss}}{\partial z_{1,3}} \\ \frac{\partial \text{Loss}}{\partial z_{2,1}} & \frac{\partial \text{Loss}}{\partial z_{2,2}} & \frac{\partial \text{Loss}}{\partial z_{2,3}} \end{bmatrix} \\ &= X^\top \frac{\partial \text{Loss}}{\partial Z}\end{aligned}$$

Finally, we still need to calculate the bias, which can be done as follows (note that $\mathbf{1}$ is a column vector of ones, hence $\mathbf{1}^\top$ is a row vector of ones);

$$\frac{\partial \text{Loss}}{\partial b} = \mathbf{1}^\top \frac{\partial \text{Loss}}{\partial Z}$$

Partial Derivatives for Vectors and Matrices

Note that we do not need to remember these, but it can be helpful;

$$\begin{aligned}\frac{\partial y}{\partial \boldsymbol{x}} &= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix} && \text{scalar-by-vector} \\ \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} &= \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} && \text{vector-by-vector}\end{aligned}$$

$$\begin{aligned} z &= Wx \\ \frac{\partial z}{\partial x} &= W \end{aligned}$$

$$\begin{aligned} z &= x \\ \frac{\partial z}{\partial x} &= I \end{aligned}$$

$$\begin{aligned} z &= xW \\ \frac{\partial z}{\partial x} &= W^\top \end{aligned}$$

$$\begin{aligned} z &= Wx \\ \delta &= \frac{\partial J}{\partial z} \\ \frac{\partial J}{\partial W} &= \delta^\top x \end{aligned}$$

$$\begin{aligned} z &= xW \\ \delta &= \frac{\partial J}{\partial z} \\ \frac{\partial J}{\partial W} &= x^\top \delta \end{aligned}$$

Backpropagation of Activation Functions

Consider the following elementwise application of activation functions;

$$\begin{aligned} A &= g(Z) \\ &= \begin{bmatrix} g(z_{1,1}) & g(z_{1,2}) \\ g(z_{2,1}) & g(z_{2,2}) \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

It's important to note that since the activation function doesn't take any other inputs, we can use the derivative g' of our activation function g in the following;

$$\frac{\partial \text{Loss}}{\partial z_{1,1}} = \frac{\partial \text{Loss}}{\partial a_{1,1}} \cdot \frac{\partial a_{1,1}}{\partial z_{1,1}} = \frac{\partial \text{Loss}}{\partial a_{1,1}} g'(z_{1,1})$$

Using this, we can generalise to matrix form (note that $\circ$ is the Hadamard product, which is element-wise multiplication);

$$\begin{aligned} \frac{\partial \text{Loss}}{\partial Z} &= \frac{\partial \text{Loss}}{\partial A} \cdot \frac{\partial A}{\partial Z} \\ &= \begin{bmatrix} \frac{\partial \text{Loss}}{\partial a_{1,1}} g'(z_{1,1}) & \frac{\partial \text{Loss}}{\partial a_{1,2}} g'(z_{1,2}) \\ \frac{\partial \text{Loss}}{\partial a_{2,1}} g'(z_{2,1}) & \frac{\partial \text{Loss}}{\partial a_{2,2}} g'(z_{2,2}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial \text{Loss}}{\partial a_{1,1}} & \frac{\partial \text{Loss}}{\partial a_{1,2}} \\ \frac{\partial \text{Loss}}{\partial a_{2,1}} & \frac{\partial \text{Loss}}{\partial a_{2,2}} \end{bmatrix} \circ \begin{bmatrix} g'(z_{1,1}) & g'(z_{1,2}) \\ g'(z_{2,1}) & g'(z_{2,2}) \end{bmatrix} \\ &= \frac{\partial \text{Loss}}{\partial A} \circ g'(Z) \end{aligned}$$

A reference of derivatives for common activation functions are as follows;

$$\begin{aligned} g(z) &= z & \text{linear} \\ g'(z) &= 1 \end{aligned}$$

$$\begin{aligned} g(z) &= \frac{1}{1 + e^{-z}} & \text{sigmoid} \\ g'(z) &= g(z)(1 - g(z)) \end{aligned}$$

$$\begin{aligned} g(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} & \text{tanh} \\ g'(z) &= 1 - g(z)^2 \end{aligned}$$

$$\begin{aligned} g(z) &= \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} & \text{ReLU} \\ g'(z) &= \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Softmax is usually applied with cross-entropy loss in the output layer - their joint derivative has a nice form (note that y_i is a vector and y is a matrix / batch of vectors);

$$\begin{aligned} \hat{y}_i &= \text{softmax}(z_i) \\ \text{softmax}(z_{i,c}) &= \frac{e^{z_{i,c}}}{\sum\limits_k e^{z_{i,k}}} \\ L &= -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log\left(\hat{y}_{i,c}\right) \\ \frac{\partial L}{\partial z} &= \frac{1}{N} \left(\hat{y} - y\right) \end{aligned}$$

Gradient Descent

As before, the general formula for applying gradient descent is as follows (note that the equals does not refer to equality but rather an update of value);

$$W = W - \alpha \frac{\partial L}{\partial W}$$

However, this assumes that all gradients can be computed, therefore all network functions and the losses need to be differentiable. The general algorithm is as follows;

1. initialise weights randomly
2. loop until convergence;
    - compute gradient based on **whole dataset**
    - update weights
3. finish

It's important to ensure we calculate **all** gradients **before** updating weights, otherwise we may have errors arising from the weights being changed before they are used in backpropagation. Additionally, this algorithm for gradient descent isn't feasible for large datasets.

In contrast, **stochastic** gradient descent updates the weights after each datapoint. However, this can be noisy since there is very restricted information from a single point. A balance between the two approaches is **mini-batched** gradient descent, which is what is mostly used in practice (this has less noise as the gradient is averaged out in a batch);

1. initialise weights randomly
2. loop until convergence;
    - shuffle dataset and split into batches
    - loop over batches of datapoints;
        - compute gradient based on batch
        - update weights
3. finish

Learning Rates

It's also important to consider the learning rate - if it is too low, it will take too long to converge. On the other hand, if the learning rate is too high, we will likely step over the optimal values.

An adaptive learning rate follows the intuition that each parameter has a different learning rate. It takes bigger steps if a parameter hasn't been updated much, and smaller steps if it has been getting big updates. Examples of this method that work quite well are *Adam* and *AdaDelta*.

A very simple approach to an adaptive learning rate is to decay it, such that we have $\alpha \leftarrow \alpha d$, where $d \in [0, 1]$ is a decay factor. This follows the intuition that we want to take smaller and smaller steps as we get closer to the minimum. This can be updated on every epoch, after a certain number of epochs, or when performance on the validation set hasn't improved for multiple epochs.

Weight Initialisation

In order to iteratively update our weights, we need to start them at some value. Some approaches are as follows;

- **zeroes**

  It's common to set the biases to 0, since it makes sense for our network to have no bias to begin with. However, we shouldn't set all weights to 0.

- **normal**

  We want to avoid setting all weights to 0, otherwise they will all learn the same updates. One option is to draw them **randomly** from a normal distribution, commonly with a mean of 0 and a variance of 1 or 0.1.

- **Xavier Glorot**

  In this we draw values from a uniform distribution, where n_j is the number neurons in the previous layer, and n_{j+1} is the number in the next layer;

  $$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

  If we have fewer neurons, we draw larger weights and vice versa.

Randomness

Different random initialisations can lead to different results. One solution would be to set the seeds for all random number generators. However, another problem is that GPUs can finish jobs in a random order, which may lead to rounding issues adding up (something we can't control). To solve this, we can embrace the randomness and report the average.

Normalisation

We often want to normalise our values, since weight updates are proportional to the input. These scaling values are calculated on the **training set only** (can lead to incorrect results if we normalise on test data too) and typically differently for each feature.

- **min-max** scale smallest value to a and largest to b

  $$X' = a + \frac{(X - X_{\text{min}})(b - a)}{X_{\text{max}} - X_{\text{min}}}$$

- **standardisation (z-normalisation)** scale to have mean 0 and standard deviation 1

  $$X' = \frac{X - \mu}{\sigma}$$

Gradient Checking

We can check if our gradient descent is implemented correctly as follows;

- from the weight difference before and after gradient descent

  $$w^{(t)} = w^{(t-1)} - \alpha\frac{\partial L(w)}{\partial w} \Rightarrow \frac{\partial L(w)}{\partial w} = \frac{w^{(t-1)} - w^{(t)}}{\alpha}$$

- actually changing the weight and measuring a change in loss

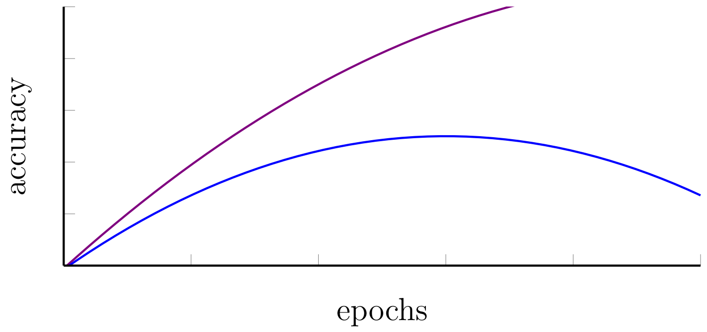
  $$\frac{\partial L(w)}{\partial w} = \lim_{\epsilon \to 0}\frac{L(w + \epsilon) - L(w - \epsilon)}{2\epsilon}$$

Both should result in similar values.

Overfitting

If a neural network has enough capacity, it can easily overfit to the training set. For example, consider a network with 100 neurons and 100 datapoints - each neuron can simply memorise the answer for each datapoint. For this reason, it's important to use **held-out** validation and test sets.

There exists a correlation between the capacity of a neural network and its ability to overfit - if there are many parameters then the model can memorise the data instead of learning patterns. If the network is underfitting, we can attempt to fix it by increasing the number of neurons / layers and vice versa for overfitting. Ideally, we can fix overfitting by having **more data**, but this isn't always feasible.



Note that the model will continue to become more accurate on the training data, but accuracy on the validation data will likely get worse as it overfits. We want to validate on the validation data every epoch, storing the best model so far, and stop training when the performance hasn't improved for a number of epochs.

Another approach is **regularisation**, where we add information or constraints to stop the model from overfitting. An example of this is to restrict the weights of the model. The forms of regularisation are as follows;

- **L2 regularisation**

  We add the squared weights to the loss function (our new loss function is now $J(\theta)$). We therefore penalise large weights, and push them towards 0 - this encourages sharing between features since small weights aren't penalised as much.

  $$J(\theta) = \text{Loss}(y, \hat{y}) + \lambda \sum_w w^2 \Rightarrow w \leftarrow w - \alpha \left(\frac{\partial \text{Loss}}{\partial w} + 2\lambda w \right)$$

  Since the weights are in the update, it reduces the weight on every update.

- **L1 regularisation**

  Similar to the above, but using absolute weights - this encourages sparsity as the model only keeps the most important features (by encouraging unnecessary weights towards 0);

  $$J(\theta) = \text{Loss}(y, \hat{y}) + \lambda \sum_w |w| \Rightarrow w \leftarrow w - \alpha \left(\frac{\partial \text{Loss}}{\partial w} + \lambda \cdot \text{sign}(w) \right)$$

Another approach is **dropout**. This sets some neural activations to 0, and typically we drop around half of the activations in a layer. This prevents the network from relying on any one node.

Module 6 (Unsupervised Learning)

Unsupervised Learning

In supervised learning, our training data is in the form of $\left\{\left(X^{(1)}, y^{(1)}\right), \dots, \left(X^{(N)}, y^{(N)}\right)\right\}$, where we have a pair consisting of a set of features X and a label y, hence we have **ground truth**. On the other hand, in unsupervised learning, we only have sets of features $\left\{X^{(1)}, \dots, X^{(N)}\right\}$. Here the task becomes more challenging, as we want to determine groups from the features alone.

Some examples of unsupervised learning are as follows;

- **clustering**
- **density estimation**
- **dimensionality reduction**

Clustering

A cluster is a set of instances that are **similar** (low intra-cluster variance) to each other, and **dissimilar** to instances in other clusters. Clustering is the process of defining these sets.

This can be applied for image compression (by clustering pixel values together), as well as for **vector quantisation**, which groups continuous features into prototypes. For example, in a video there may be very similar frames of wheels on a car, which can be replaced with a single instance. Another application is to create **superpixels**, which group similar-looking pixels as an intermediate stage of processing.

K-means

Similar to k-NN, the k is a hyperparameter which refers to the number of clusters. The general algorithm is as follows;

1. **initialise**

   The idea is to randomly initialise the locations of the k cluster centres (referred to as **centroids**). An improvement to this is to randomly select k instances, which avoids the issue of empty clusters.

2. **assignment** minimise L with respect to c, by fixing μ

   Here we can assign each of the instances to be part of the nearest cluster, based on some distance metric.

   For each $i \in \{1, \dots, N\}$, we want to do the following (where $c^{(i)} \in \{1, \dots, K\}$ refers to which cluster training example $x^{(i)}$ is assigned to), by looking at the distance between the training example (i) and the cluster mean μ_k;

   $$c^{(i)} = \underset{k \in \{1, \dots, K\}}{\text{argmin}} \; ||x^{(i)} - \mu_k||^2$$

3. **update** minimise L with respect to μ, by fixing c

   Move the position of the centroid by computing the average location of the instances associated with a given centroid.

   For each cluster $k \in \{1, \dots, K\}$, we want to compute the mean of the cluster k (μ_k), using the indicator function $1(a)$, which is 0 when a is false, and 1 if a is true;

   $$\mu_k = \frac{\sum\limits_{i=1}^{N} 1(c^{(i)} = k) \cdot x^{(i)}}{\sum\limits_{i=1}^{N} 1(c^{(i)} = k)}$$

4. finish if **converged**, otherwise go back to step 2

   Observe the distance the centroid moves - if it is below a given threshold, we can conclude it has converged.

   To test for convergence, we simply check all values against some threshold ϵ;

   $$\forall k \left| \mu_k^t - \mu_k^{(t-1)} \right| < \epsilon$$

This creates decision boundaries, in this case a **Voronoi diagram**, where every line in the diagram is at equal distance between two points. We can also look at this as an optimisation objective, where we aim to minimise;

$$L(c^{(1)}, \dots, c^{(N)}, \mu_1, \dots, \mu_k) = \frac{1}{N} \sum_{i=1}^{N} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Therefore, we are aiming to minimise the average distance between each training sample to the mean of its assigned cluster.

However, the choice of k is important, and we can select it with a number of methods;

- **elbow method**

  The idea here is to run k-means multiple times, with different k values, and track the cost $L(\theta)$ for each k. We want to select the k value where the rate of decrease sharply shifts (becomes slower). One important consideration is that the score alone cannot be used, since it will **always** converge to 0, when k is equal to the number of datapoints.

- **cross-validation**

  This is used less often than the elbow method, but is able to capture behaviour on unseen data. As before, with cross validation, the dataset is split into N folds, with $N-1$ folds being used to compute the centroid positions. The average score is then computed as before on the validation datasets, and the best value of k is picked. We want to select a k where any further increase in the number of clusters leads to a only a small improvement in the average score.

This is a simple algorithm, both to understand and implement, and is also popular. It's also efficient (linear complexity since $K << N$ and $T << N$), with complexity $O(TKN)$, where T is the number of iterations, K is the number of clusters, and we have N examples.

However, there are a few weaknesses, one in that we have to define k, it finds a local optimum, and is also sensitive to the initial centroid positions (and also to outliers, which can be solved with k-medoid).

Additionally, k-means is only applicable if there is some distance function - a variant, k-mode, exists for categorical data, where the centroids represent the most frequent values. It is also not suitable for discovering clusters that are not hyper-ellipsoids / hyper-spheres (has a continuous and convex shape).

Probability Density Estimation

A probability density function $p(x)$ models how likely a continuous variable is to be observed within a particular **interval**. This has the following constraints, note that there is only a lower bound (and therefore can be greater than 1);

$$p(x) \geq 0$$

$$\int_a^b p(x) \, \mathrm{d}x = 1$$

Density estimation is to estimate $p(x)$ from data.

An example of this is for anomaly detection, where we we calculate the probabilities for a given task (for example CPU usage versus memory usage in a data-centre), and if a point lies in a region of low probability (below a given threshold) - we can classify it as abnormal.

Another application is a **generative model**, where we model the distribution of a given class, as $p(X \mid y)$ - this allows us to generate new samples (such as images). This is in contrast to **discriminative models**, which directly model $p(y \mid X)$, which is used in neural networks, linear regression, decision trees, etc.

Using the following, we can convert a generative model into a discriminative classifier;

$$p(y \mid X) = \frac{p(X \mid y)p(y)}{p(X)}$$

There are two approaches for these models;

- **non-parametric approach**

  A non-parametric model makes no assumptions about the form. This simply follows the data (has low bias, therefore we get high variance), and the number of parameters grow with data.

  One example of a density estimation, which we have likely encountered, is a histogram. Here data is grouped into contiguous bins, and the number of occurrences in each bins is counted (and then normalised, to ensure a sum to 1). By decreasing the number of bins, we are able to smooth out $p(x)$ - it will suffer from high variance, but will not be biased.

  Another approach is **kernel / Parzen density estimation**, where we compute $\hat{p}(x)$ by observing training examples within a kernel function H (or a Parzen window);

  $$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{h^D}H\left(\frac{x - x^{(i)}}{h}\right)$$

  In the example above, we have h as the bandwidth / window size, and D as the number of dimensions (and h^D as the volume of the window). The Parzen window we can use is defined as follows;

  $$H(u) = \begin{cases} 1 & \text{if } |u_j| < \frac{1}{2}; j = 1, \dots, D \\ 0 & \text{otherwise} \end{cases}$$

  Another way would be to define the kernel function H as a Gaussian, which gives us a smoother $\hat{p}(x)$;

  $$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{(2\pi h^2)^{\frac{D}{2}}}e^{-\frac{||x-x^{(i)}||^2}{2h^2}}$$

  Increasing h gives a smoother $\hat{p}(x)$.

- **parametric approach**

  In contrast, a parametric method makes simplified assumptions about the form (which leads to high bias, and low variance), and the number of parameters is fixed (doesn't grow with the data).

  Similar to before, we have likely encountered such an example - a **univariate Gaussian (normal) distribution**;

  $$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  $\hat{p}(x)$ is computed by fitting the parameters to the training examples;

  $$\begin{aligned} \hat{\mu} &= \frac{1}{N}\sum_{i=1}^{N}x^{(i)} \\ \hat{\sigma}^2 &= \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \hat{\mu})^2 \\ \hat{p}(x) &= \mathcal{N}(x \mid \hat{\mu}, \hat{\sigma}^2) \end{aligned}$$

An extension to this is a multivariate Gaussian distribution. This is largely similar, however we now instead have a covariance matrix $\boldsymbol{\Sigma}$;

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}$$

Similarly, the parameters are fitted to the training example as follows;

$$\begin{aligned}
\hat{\boldsymbol{\mu}} &= \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^{(i)} \\
\hat{\boldsymbol{\Sigma}} &= \frac{1}{N}\sum_{i=1}^{N} (\boldsymbol{x}^{(i)} - \hat{\boldsymbol{\mu}})(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{\mu}})^\top \\
\hat{p}(\boldsymbol{x}) &= \mathcal{N}(\boldsymbol{x} \mid \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}})
\end{aligned}$$

To quantify how good a fitted model is, we use **likelihood**. Our goal is to have a model that captures the probability of generating or observing data x within an interval. The likelihood measures the probability of observing data x from a dataset $\mathcal{X}$, where θ are the model parameters, assuming iid;

$$p(\mathcal{X} \mid \theta) = \prod_{i=1}^{N} p(x^{(i)} \mid \theta)$$

We often use the **negative log-likelihood**, which becomes a minimisation problem;

$$\mathcal{L} = -\log p(\mathcal{X} \mid \theta) = \sum_{i=1}^{N} \log p(x^{(i)} \mid \theta)$$

By taking logarithms, we improve numerical stability as the multiplication of small probabilities is bad. The proof for a univariate Gaussian distribution is as follows;

$$\begin{aligned}
\mathcal{L} &= -\sum_{i=1}^{N} \log p(x^{(i)} \mid \theta) \\
&= -\sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x^{(i)}-\mu)^2}{2\sigma^2}} \\
&= \frac{N}{2}\log 2\pi + \frac{N}{2}\log\sigma^2 + \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x^{(i)} - \mu)^2
\end{aligned}$$

We know that $\mathcal{L}$ is minimum when the derivatives are zero;

$$\begin{aligned}
\frac{\partial\mathcal{L}}{\partial\mu} &= 0 \\
\frac{\partial\mathcal{L}}{\partial\sigma^2} &= 0 \\
\frac{\partial\mathcal{L}}{\partial\mu} &= \frac{1}{\sigma^2}\sum_{i=1}^{N}(x^{(i)} - \mu) \\
&= \frac{1}{\sigma^2}\left(\sum_{i=1}^{N} x^{(i)} - N\mu\right) \\
\frac{\partial\mathcal{L}}{\partial\sigma^2} &= \frac{N}{2}\frac{1}{\sigma^2} - \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{N}(x^{(i)} - \mu)^2
\end{aligned}$$

When these are set to zero, it gives us the solutions as $\hat{\mu}$ and $\hat{\sigma}^2$.

However, a Gaussian distribution is insufficient for modelling densities, for example if we have multiple classes. An improvement on this is to use **mixture models**;

$$p(x) = \sum_{k=1}^{K} \pi_k p_k(x)$$

$$0 \leq p_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

A Gaussian Mixture Model (GMM) is defined as the following;

$$p(\boldsymbol{x} \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$$

Where we have θ defined as; $\theta = \{\pi_k, \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k} : k = 1, \dots, K\}$

GMM-EM

As previously mentioned, a **Gaussian Mixture Model** is a weighted mixture of Gaussian distributions. Because of the constraints we have on our weights, the mixture of probability density functions stays as a PDF.

GMM's can model complicated data distributions, however due to the increased complexity, there are more parameters to optimise. Unlike the univariate or multivariate case, we cannot maximise the likelihood since the update of each parameter depends on other parameters - hence we do not have a closed form for GMMs (this is similar to how the clustering changes when a centroid is updated in k-means).

The **Expectation-Maximisation (EM)** algorithm is as follows;

1. **initialisation**

   First pick a K value, and then randomly initialise the parameters;

$$\theta = \{\pi_k, \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k} : k = 1, \dots, K\}$$

$$0 \leq p_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

2. **E-Step**

   Compute the **responsibilities** for each training example $\boldsymbol{x^{(i)}}$, and each mixture component k (note that the following is done for each i **and** k);

$$r_{i,k} = \frac{\pi_k \mathcal{N}(\boldsymbol{x^{(i)}} \mid \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})}{\sum\limits_{j=1}^{K} \pi_j \mathcal{N}(\boldsymbol{x^{(i)}} \mid \boldsymbol{\mu_j}, \boldsymbol{\Sigma_j})}$$

3. **M-Step**

   Note that for the following part, we will be using the following;

$$N_k = \sum_{i=1}^{N} r_{i,k}$$

- Update the **mean** of all the mixture components

  $$\hat{\boldsymbol{\mu}}_{\boldsymbol{k}} = \frac{1}{N_k} \sum_{i=1}^{N} r_{i,k} \boldsymbol{x}^{(\boldsymbol{i})}$$

  This changes the centres of the Gaussians (if we consider this with two features, looking top-down at contours).

- Update the **covariance** of all the mixture components

  $$\boldsymbol{\Sigma}_{\boldsymbol{k}} = \frac{1}{N_k} \sum_{i=1}^{N} r_{i,k} \left(\boldsymbol{x}^{(\boldsymbol{i})} - \hat{\boldsymbol{\mu}}_{\boldsymbol{k}}\right) \left(\boldsymbol{x}^{(\boldsymbol{i})} - \hat{\boldsymbol{\mu}}_{\boldsymbol{k}}\right)^\top$$

  Similar to the visual above, this changes the shapes of the Gaussians.

- Update the **mixing proportion** of all mixture components

  $$\pi_k = \frac{N_k}{N}$$

  This changes the height of the Gaussians (assuming two features as before, but with the y axis being p).

Similar to k-means, this will converge to a local optimum, and therefore should be repeated, finally taking the model with the minimum negative log likelihood.

4. finish if **converged**, otherwise go back to step 2

   Define convergence as the parameters not changing, or the likelihood stagnates.

One approach to selecting the number of components K is to minimise the **Bayesian Information Criterion (BIC)**;

$$BIC_K = \mathcal{L}(K) + \frac{P_k}{2} \log(N)$$

Note that the above consists of two parts;

- violet - negative log likelihood, which encourages fitting of data
- teal - penalises complex models; Occam's razor suggests we should pick the simplest of all models that fit

  Note that P_K denotes the number of parameters, and N denotes the number of examples. The the number of parameters for a 2D Gaussian is $P_K = 6K - 1$, where we have 2 for mean, 3 for covariance, and 1 for the mixing proportion. It's important to note that while the covariance matrix is $\mathbb{R}^{2\times 2}$, it is symmetrical hence we do not need the entire matrix to define it.

Another approach is to perform cross-validation with the same procedure as k-means, bu splitting into training and validation sets, then running GMM-EM on the training sets.

There are a fair number of similarities between k-means and GMM-EM;

- need to select K
- convergence happens when changes are sufficiently small
- also sensitive to initialisation (often initialise the means of GMM-EM from the result of k-means)

We can consider GMM-EM as a softer k-means clustering. On one hand, with hard clustering (k-means), every point belongs to exactly one cluster, whereas with soft clustering (GMM), every point belongs to several clusters in certain degrees.

Each component in the mixture represents a different cluster, and the responsibilities of each data point define the probability of a data point belonging to a certain cluster. Another difference in GMM is that the distance to the centroids (the means) is not isotropic and varies during the learning process, due to the covariance matrix (the distance is related to the Mahalanobis distance).

We can summarise the differences between the two as follows;

- **objective function**
  - *k*-means	minimise average mean squared distance
  - GMM-EM	maximise log likelihood
- **algorithm**
  - *k*-means
    - assignment	assign points to cluster
    - update	optimise cluster
  - GMM-EM
    - E-step	compute posterior probability of membership
    - M-step	optimise parameters
- **hardness**
  - *k*-means	performs hard assignment during assignment step
  - GMM-EM	performs soft assignment during E-step
- **other**
  - *k*-means

    assumes spherical clusters with equal probability for each cluster
  - GMM-EM

    can be used for non-spherical clusters and can generate clusters with different probabilities

Module 7 (Genetic Algorithms)