Neural Networks

Slides are available at muli.doc.ic.ac.uk:9000 with live lecture updates and PDF version on CATE. For the live version you can use any username and password, *do not use your college password*. These slides are based on the full lecture notes that are available online.

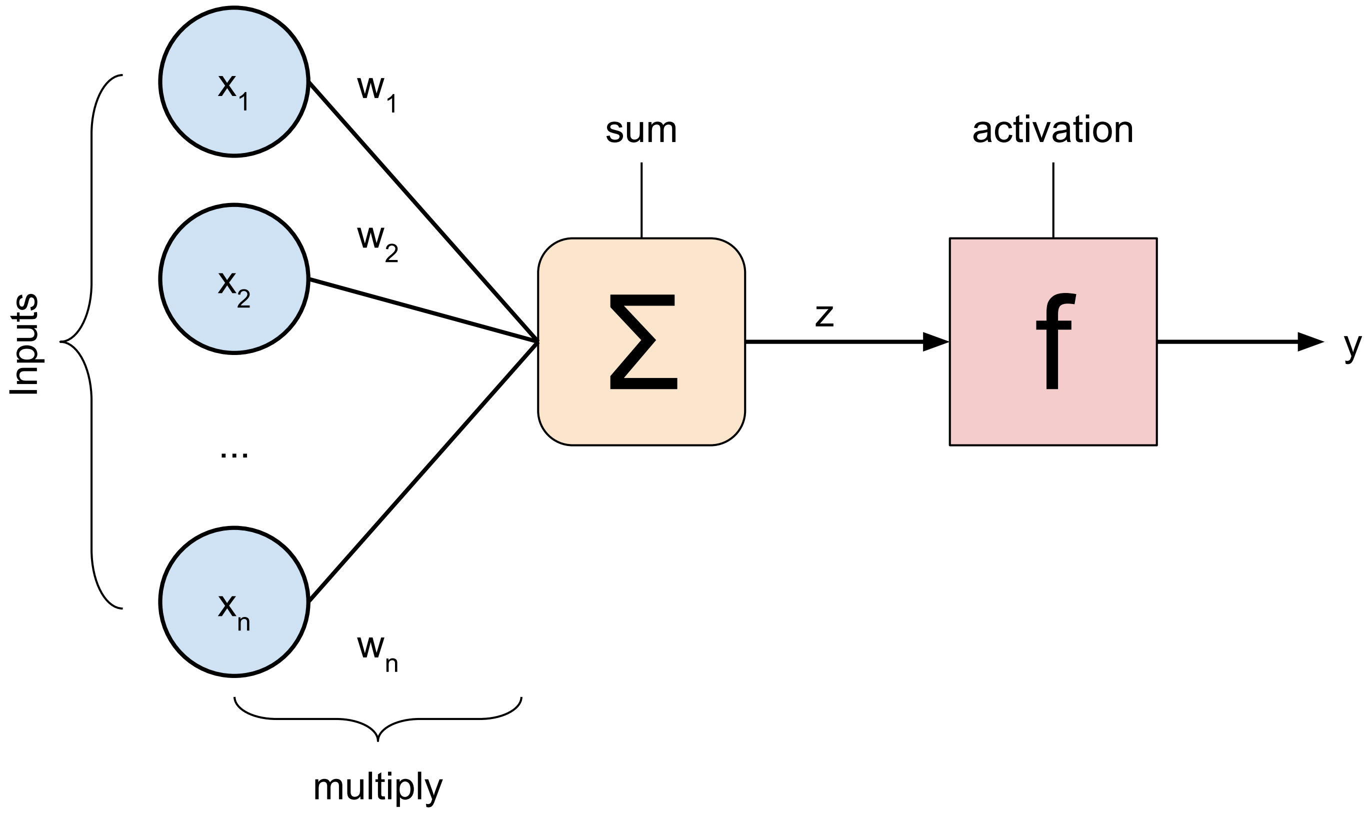
Inspired by / based on / referenced from: Artificial Intelligence: A Modern Approach (Peter Norvig and Stuart J. Russell), Deep Learning (Aaron C. Courville, Ian Goodfellow, and Yoshua Bengio), Course 6.036 Massachusetts Institute of Technology, Courses CS231 & CS224n Stanford University

Office Hours: Thursdays 14th and 21st of February from 2pm to 4pm (before lab tutorial) in either 558B or 555 on the 5th floor. These are for *non-coursework* based discussions, questions and headaches surrounding neural networks.

The learning outcomes:

- **Describe** the module keywords in the context of machine learning and neural networks.
- **Implement and train** a neural network to solve a machine learning task
- **Summarise** the steps of learning with neural networks
- **Assess and improve** the suitability of a neural network for a given task

Neuron



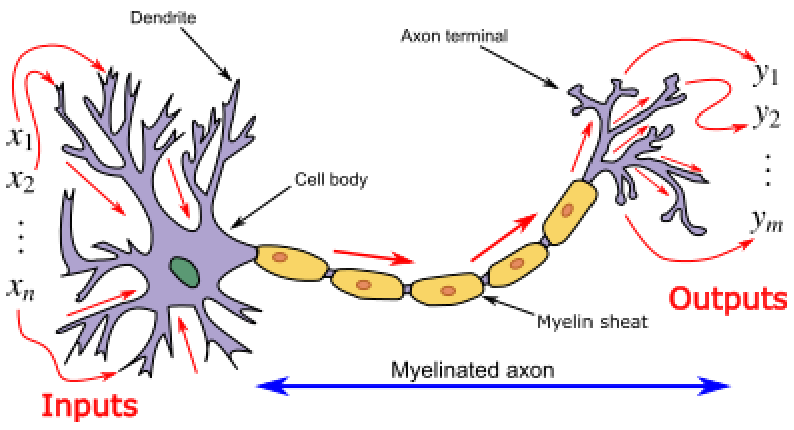
$$y = f(z) = f(\sum_i w_i x_i) = f(w^T x)$$

which reads every *input* is multiplied by a corresponding **weight** and then summed to get z, the pre-activation value is then passed to the **activation function** f. In vector form, $w \in \mathcal{R}^{n\times 1}$ and $x \in \mathcal{R}^{n\times 1}$.

On the opposite end, we can simplify things to a single input and the equation becomes somewhat familiar:

$$y = f(wx)$$

This looks like the function of a line $y = wx + b$ but there is something we omitted: **the bias**. But if $x' = [x, 1]; w = [w_1, w_0]$ and we get $y = w_1 x + w_0$ which is exactly the same as having an explicit bias value.



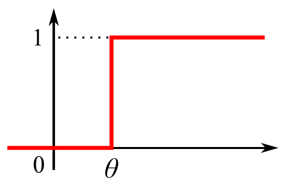
- A brain-inspired network that learns some functions. This view is common with the public who think we are building artificial brains of some sort and some day the machines are going to take over the world.
- An algebraic learnable transformation of inputs with respect to some outputs.

> **Pro Tip:** Evolution had a lot of time to randomly explore solutions to problems we face, roughly 4 billion years ago life has started on Earth. In contrast the term personal computer was coined in 1975. So if there is a problem, you might need to get out of labs to find a solution.

Perceptron

Perceptron is *an algorithm* for supervised binary classification. We have some labelled data $x^{(1)}, x^{(2)}, x^{(3)}, \ldots$ which has desired outputs $y^{(1)}, y^{(2)}, y^{(3)}, \ldots$ and we are trying to do binary classification, so there are 2 classes we want to predict, either class 0 or 1. Let's take the activation function to be a *threshold function*:

$$h_w(x) = f(w \cdot x) = 1 \text{ if } w \cdot x \geq 0 \text{ else } 0$$

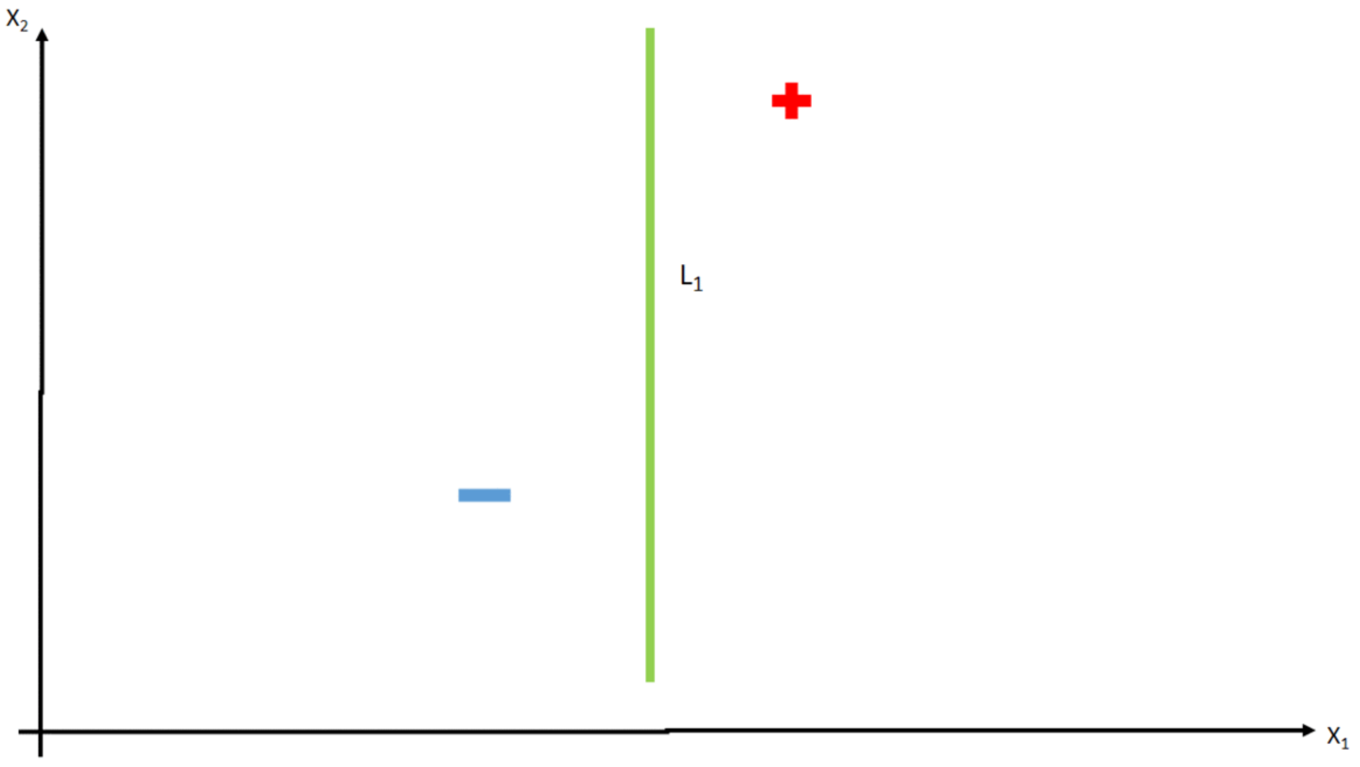


Then the **perceptron learning rule** (equation 18.7, chapter 18 from the textbook Artificial Intelligence: A Modern Approach) to update the weights in order to learn this classification is:

$$w_i \leftarrow w_i + \alpha(y - h_w(x)) \times x_i$$

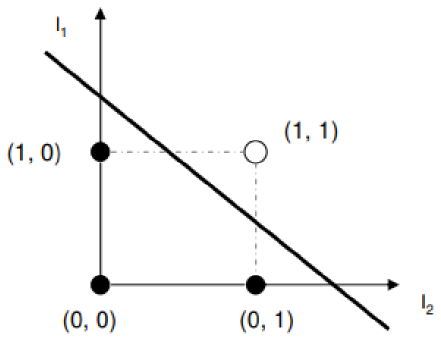
The α is the **learning rate** that allows us make smaller incremental changes rather than large jumps which can be unstable. Let's analyse this equation to see what happens:

- if the desired output is equal to our prediction (perceptron output), $y = h_w(x)$ then the right hand side of the summation becomes 0 and the weights stay the same. Intuitively, don't fix it if it isn't broken.
- if $y = 1; h_w(x) = 0$ then the corresponding weight is increased when corresponding input is x_i is positive and decreased when it's negative. By doing so we want to make $w \cdot x$ bigger since the desired output is larger than our prediction.
- if $y = 0; h_w(x) = 1$ the opposite of the previous situation happens because we want to decrease the summation.

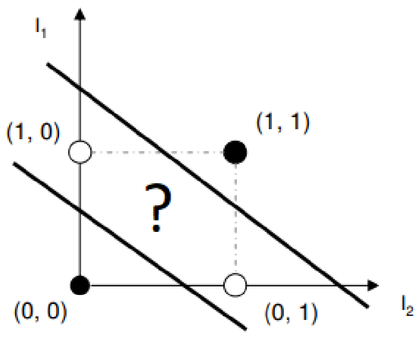


We can *learn any linearly separable function* using perceptron:

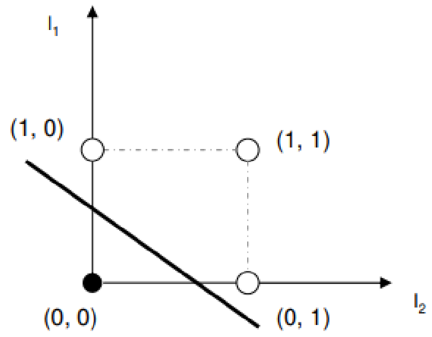
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



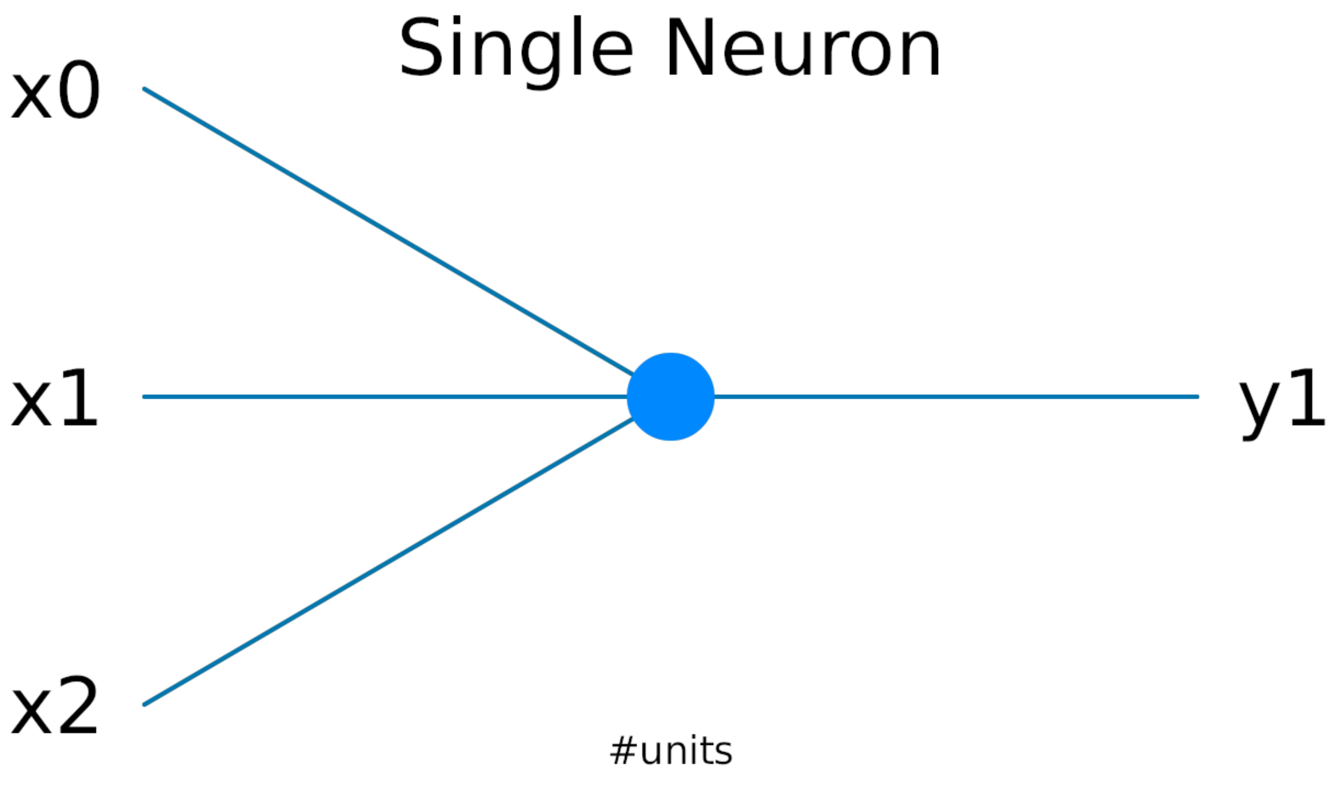
OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1



> **Pro Tip:** These days you only hear about the success of neural networks and deep learning but not much about their failures. A failed result is as important and valuable as the successful

> one.

Layer



The weights are collected into a matrix $W \in \mathcal{R}^{N \times M}$ with bias $b \in \mathcal{R}^{M \times 1}$ producing $y \in \mathcal{R}^{M \times 1}$ outputs. The equation is just the vector form of a single neuron:

$$y = f(W^T x + b)$$

> **Pro Tip:** When you find yourself writing vector form equations and dealing with matrices, vectors, tensors, just checking the shapes *match* is a good starting point to make sure things work.

The scalar activation function is applied **element-wise** to a vector: $f(x) = f([x_1, x_2, \dots]) = [f(x_1), f(x_2), \dots]$ where each element is passed through the function individually.

$$z = W^T x + b$$
$$y = f(z)$$

We can separate the activation from the linear transformation. But why?

```
# From https://github.com/keras-team/keras/blob/master/keras/layers/core.py
# This is the source code of the library, I didn't make it up :)
class Dense(Layer):
  # ... some initialisation here, like creating weights etc.
  def call(self, inputs):
    # Your Wx, the weights are also referred to as the kernel
    # So self.kernel is your W
    output = K.dot(inputs, self.kernel)
    # Add the bias b if requested
    if self.use_bias:
      output = K.bias_add(output, self.bias, data_format='channels_last')
    # Apply activation function if any
    if self.activation is not None:
      output = self.activation(output)
    return output
```

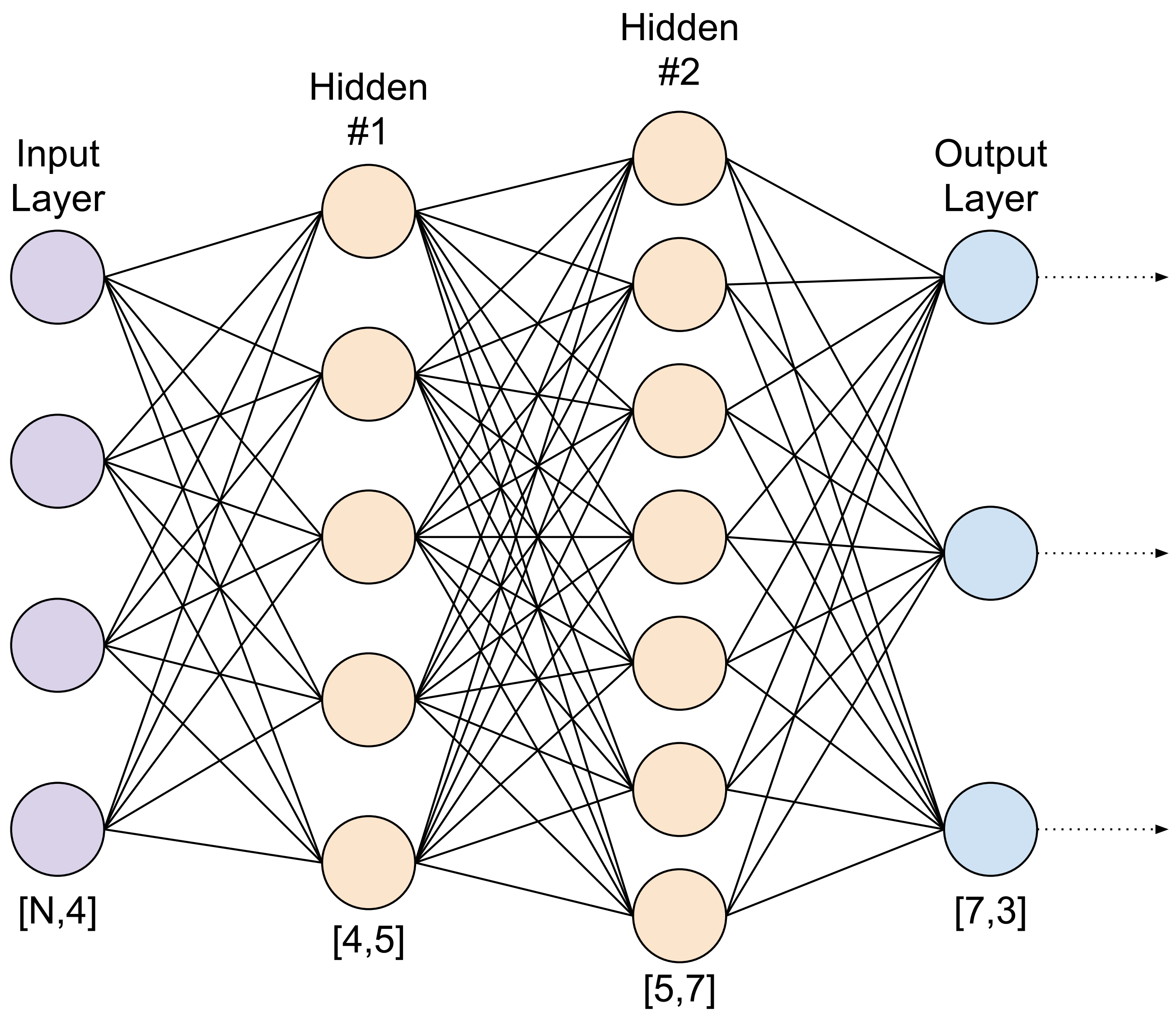
```
# From Keras library
keras.layers.Dense(units, activation='[the activation function]', ...)
# From Tensorflow, which is identical to Keras (literally it calls Keras)
tf.layers.Dense(units, activation='[the activation function]', ...)
# From PyTorch, opts to put activation as a separate layer
torch.nn.Linear(num_inputs, num_outputs, bias=True)
# From Chainer, yes there are many other libraries
chainer.links.Linear(num_inputs, num_outputs, nobias=False,  ...)
### These all compute y = f(Wx+b) or y = Wx+b (if no activation)
```

Feed-Forward Networks

We can now define what is referred to as a **feed-forward**, **deep feed-forward** or **multi-layer perceptron (MLP)** network. In the context of deep learning, these terms are used interchangeably. They are **a collection of layers chained together**.

$$y = h_3(h_2(h_1(x)))$$

The **depth** of the network is often referred to the number of layers while the **width** is the number of neurons. This is where the term **deep learning** comes from -> more layers more depth -> deeeeeeep learning. The **architecture** of a network refers to the overall structure: how many units the layers have, how these units are connected together (densely, recurrently etc). The layers in between the input and output are called **hidden layers**.



In fact let's implement this! Here is the Keras version of that *exact* diagram:

```
network = keras.models.Sequential() # one layer comes after the other
network.add(keras.layers.Dense(5, input_dim=4) # 4 inputs, 5 outputs
network.add(keras.layers.Dense(7)) # note we don't have to specify input size again, why?
network.add(keras.layers.Dense(3))
# that's it, we have a feed-forward network, let's also use it
network.predict([[41,42,42,43]]) # we get a vector of size 3
```

To be more rigorous let's break down the mathematical steps of a network:

$$A^{(l)} = f^{(l)}(Z^{(l)}); Z^{(l)} = W^{(l)} \cdot A^{(l-1)}; A^{(0)} = x$$

Initialising Weights

When we create a new network, what are the values of weights initially? Most deep learning libraries have weight initialising in built, for example Keras has numerous initialising functions:

- **Zeros:** just sets the corresponding parameters to 0. This is commonly used for the bias since initially you might not want any value for the threshold of a neuron. Zeros is the default initialisation function for the bias in the `keras.layers.Dense` layer.
- **Normal:** sets the parameters from a normal distribution $W \sim \mathcal{N}(\mu, \sigma^2)$ often setting the mean to 0 and variance to 1.

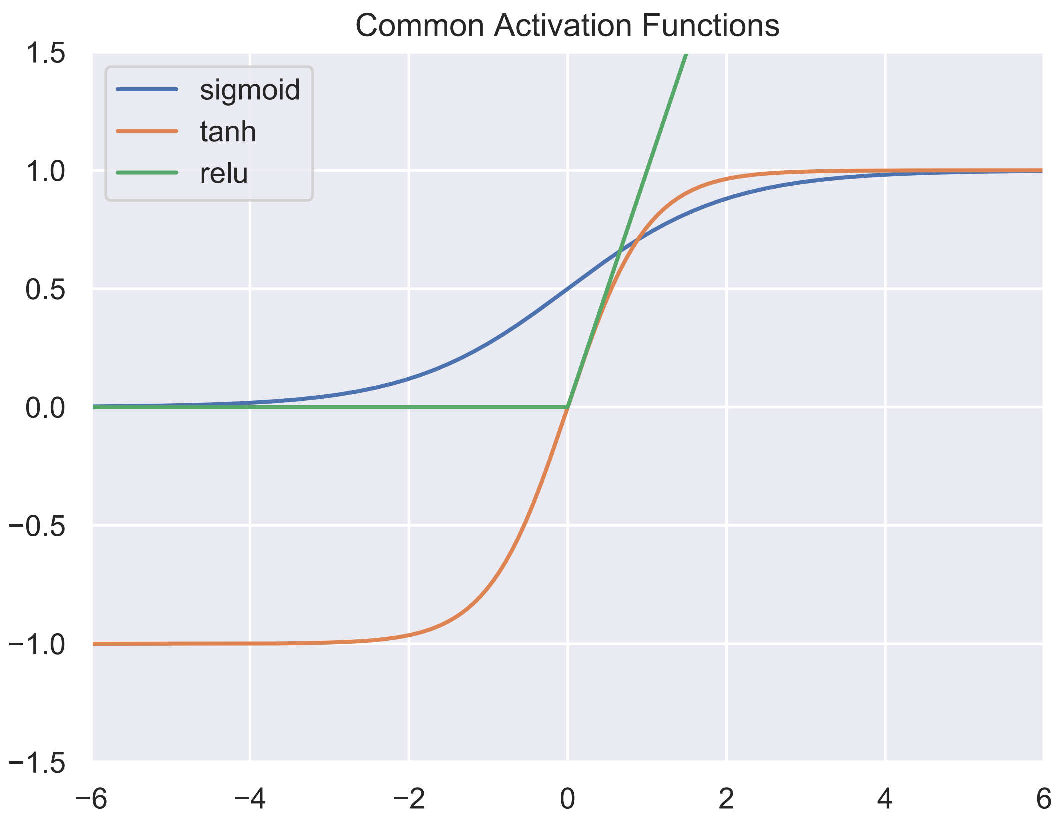
We want is values to be random but vary in a *reasonable range* across the layers. In other words, *we want the variance of the output of the layers to stay stable across the layers*. To address this issue **Xavier Glorot** and Yoshua Bengio proposed a normalised initialisation method in their paper Understanding the difficulty of training deep feedforward neural networks; from section 4.2, "we suggest the following initialization procedure to approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network":

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

where U is the uniform distribution and n_j is the number inputs of the layer and n_{j+1} is the number of outputs.

Activation Functions

We want to learn beyond just linear functions. So $Wx + b$ on it's own doesn't allow us to learn **non-linear** functions. *Therefore, to introduce non-linearity into the network we apply non-linear activation functions.*



Let's look at the most common activation functions, this is non-exhaustive, there are variants of these and other activation functions but these are the most commonly used ones:

- **linear (identity)** is the same as having no activation function. It is called linear because it exposes the linear transformation that often precedes the activation, $y = W^T x$ directly.
- **sigmoid** compresses the output to the range between 0 and 1 with 0 corresponding to 0.5. It is also called the *logistic function* coming from logistic regression which views the activation

function as a soft version of the threshold function we saw earlier. Instead of suddenly jumping from 0 to 1 at a given threshold, it varies smoothly.

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

- **tanh** adjusts the sigmoid such that input 0 now corresponds to output 0 and thus it ranges between -1 and 1. It is a scaled version of sigmoid.

$$tanh(x) = \frac{2}{1 + (e^{-2x})} - 1$$

- **ReLU** stands rectified linear unit. This activation is the most commonly used one for feed-forward networks since it preserves the desirable properties of a linear function while introducing non-linearity. It is a **piece-wise linear** function which means it is composed of linear functions but overall it is a non-linear function itself.

$$relu(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

- **softmax** can be thought as the n-dimensional version of sigmoid, it compresses the sum of the output vector to be 1.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}; z \in \mathcal{R}^n$$

So what to use when? Well, in the hidden layers we often use ReLU to get the computational benefits and stability of the linear functions while having non-linearity. Can't we just use the linear activation everywhere? **No!** That would just make things linear again:

$$y = W^1(W^2x) = Ux; U = W^1W^2$$

If we don't use non-linear activations the entire network becomes linear as well! So ReLU is our closest friend. We can also use `tanh` or `sigmoid` in the hidden layers which works well for shallow classification networks but in general you'll see by default it is a good idea to start with ReLU when using feed-forward networks and then experiment.

The output layer activation is special since it will work in tandem with the loss function. So the final layer activation will almost always depend on what we are trying to achieve with the network, is it binary classification, multi-class or just regression?

Loss Functions

The **loss function** is the function we are trying to minimise such that when we do so we learn the relationship between the given inputs and the desired outputs. It is *crucial* to select / design the correct loss function in order to be able to not only learn but also learn something meaningful. Instead of just enumerating the loss functions there are, Keras loss functions, *we will take a practical, problem first approach here.*

Regression

When the task is to predict a continuous variable such as the velocity of a car, angle of a robot, house price etc we have a regression problem. What we will be doing is often called *non-linear regression* because the function we are trying to learn will be non-linear.

When we have such a problem we use the **squared error** loss function:

$$L(y^{(i)}, a^{(i)}) = (a^{(i)} - y^{(i)})^2$$

which intuitively is 0 when the network output is same as the desired output.

When there are multiple outputs, we take the mean which yields the **mean-squared error** that is often used as the loss function in deep learning libraries. The $a^{(i)}$ is our network output for the corresponding desired target, the final layer activation if you will.

$$\frac{1}{d}\sum_{j}^{d}(a_j^{(i)} - y_j^{(i)})^2$$

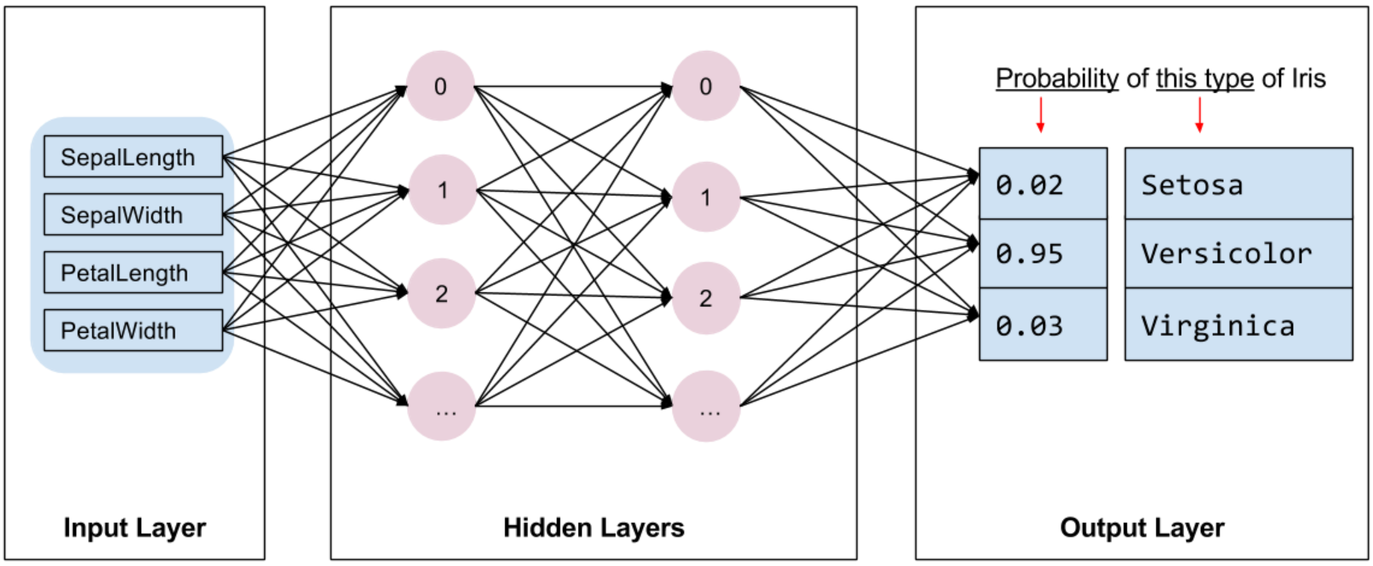
```
# From Keras losses.py source code
def mean_squared_error(y_true, y_pred):
  return K.mean(K.square(y_pred - y_true), axis=-1)
# Notice it uses vectorised functions rather than a for loop
```

What is the activation of the last layer? Since the output is not bounded, we often use **linear activation when regressing** with the squared error loss function. It allows to produce a continuous linear output which maps well to the target domain of house prices for example.

Classification

Perhaps the more common type of problems we encounter are classifying things which entails that our output is **categorical** (discrete). Given some input we want it to be labelled with one category, for example happy or sad.

- **Binary classification** refers to the situation when we have two classes
- **multi-class classification** is when we have more than two. We *assume* every input belongs to one class and one class only.
- When we want to predict multiple classes / labels we have **multi-label classification** for example a molecule could be odourless and flammable at the same time.



Type	Layer Activation	Desired Output	Loss
binary	sigmoid	0 or 1	binary cross entropy
multi-class	softmax	one-hot	categorical cross entropy
multi-label	sigmoid	0s and 1s	binary cross entropy

We can interpret the sigmoid output as a probability of that input belonging to a class $p(y|x)$, i.e. the probability of the output given input. **We regard our network output to be probability distribution over classes.**

As such we would like to *maximise* the likelihood of the network assigning the corrects labels to all inputs in our dataset:

$$\prod_i^N p(y^{(i)}|x^{(i)};\theta)$$

assuming that the examples are independent and identically distributed (i.i.d.) such that $p(A \wedge B) = p(A)p(B)$.

We get a product over all all input output pairs. In this case of **binary classification** this will spell out as:

$$\prod_i^N (a^{(i)})^{y^{(i)}} (1 - a^{(i)})^{1-y^{(i)}}$$

which is to say in the binary case the probability of one class is 1 minus the other; the $a^{(i)}$ is our network output for the corresponding input. If we output the same label as the desired one for every pair we get 1, if we output the complete opposite then we get 0.

We take the logarithm:

$$\sum_i^N y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$$

since maximising the logarithm is the same as maximising the original product. Let's take the inner equation defined on a single pair:

$$L(y^{(i)}, a^{(i)}) = -(y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}))$$

is the **binary cross entropy** loss function which is also referred to as **negative log likelihood** based on the equation. Notice the extra minus in front, this is because when optimising we often minimise the loss.

This can be extended to the multi-class situation:

$$L(y^{(i)}, a^{(i)}) = -\sum_k y_k^{(i)} \log(a_k^{(i)})$$

is the **categorical cross entropy** loss function where k is the number of classes we have. In this situation we use **softmax** activation function in the final layer to get $a^{(i)}$ which yields a probability distribution. For example, we can have $[0.2, 0.6, 0.2]$ with our desired output $[0, 1, 0]$ (one-hot encoded). Minimising this loss function will incrementally push our output towards $[0.0001, 0.9999, 0.0001]$ the desired output.

To get an intuition as to why these are called cross entropy losses and possibly explore (not within the scope of this module) the information theory background of these loss functions have a look at the cross entropy formula:

$$H(p,q) = -\sum_x p(x)\log(q(x))$$

where p and q are probability distributions. Looking familiar? When we consider our network output and the desired outputs to be probability distributions, we end up with this cross entropy formula which stems from the KL divergence.

> **Pro Tip:** Since we cannot represent numbers to an arbitrary precision, for example we have 32 bits for floating point numbers, we can get close to 1 such as 0.999987945 but perhaps not get exactly 1 because the updates get really really small. Most libraries compensate for numerical instabilities as numbers get really small, for example when we approach desired targets and the loss becomes infinitesimally small.

Can we not use just use squared error again with sigmoid activation? Yes, we can. Nothing will stop you from trying it out, in fact it will work to some extent. But, using a log based loss provides better convergence properties. Recall that the sigmoid has e^z and the cross entropy loss has log, these cancel out and have a better defined slope / gradient allowing the network to converge better.

Here are some examples:

Problem	Type	Activation	Loss
future stock prices	regression	linear	mse
stock prices up or down	binary	sigmoid	binary crossentropy

MORE TO COME :)

Training

When we say we are training, we mean we are adjusting / modifying parameters to *fit* the desired data. Cool. So all we need is **a way to adjust the parameters such that the network gets better at predicting our data.**

> **Pro Tip:** When someone says they are training their neural network, it means they are wondering around aimlessly while the computer desperately tries to find the best parameters for their network. No one manually adjusts parameters by hand unless you are Jeff Dean who gets the best parameters at random initialisation.

Back Propagation

How do we know whether we should increase or decrease a weight? We need to understand how the weight affects our loss function. For this, we use the **derivative** of the loss function with respect to the weight. We'll assume you are familiar with taking derivatives and partial derivatives for this module. Intuitively, the derivative of a function with respect to a parameter $\frac{dy}{dx}$ tells us how y changes when x does. If you are unfamiliar with the concept, let's consider this basic case:

$$z = x^2 + 4x - 3y + 10$$
$$\frac{\partial z}{\partial x} = 2x + 4$$
$$\frac{\partial z}{\partial y} = -3$$

$$\frac{\partial Loss}{\partial W^{(L)}} = \frac{\partial Loss}{A^{(L)}} \cdot \frac{\partial A^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial W^{(L)}}$$

What just happened? We applied the **chain rule** of derivation to find the partial derivative of the loss function with respect to the weights since $A = f(Z); z = WX$. We are taking the partial derivative of the loss with respect to *every weight* in the layer, because we want to adjust all of them.

Let's look at the hidden layer before the output layer:

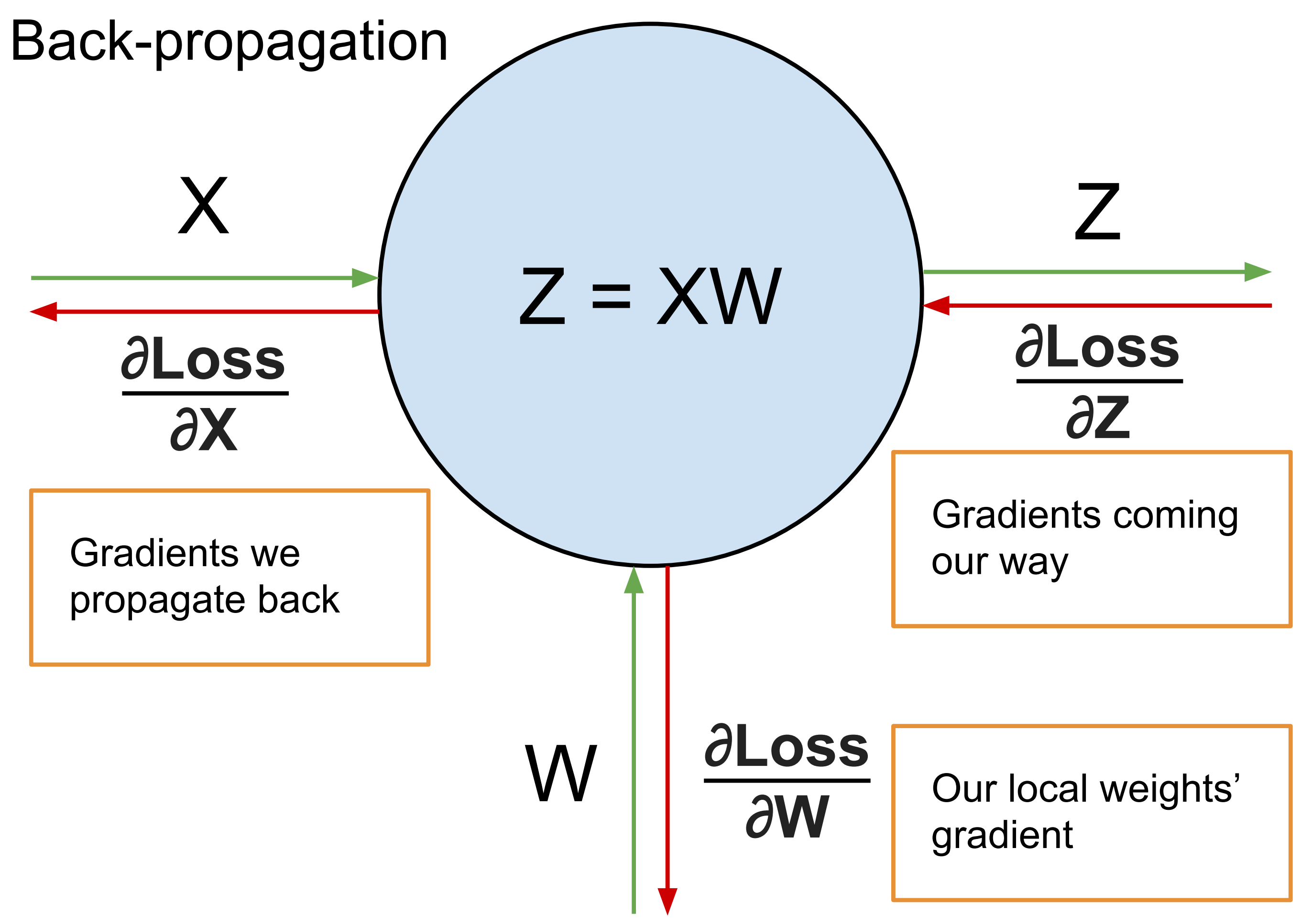
$$\frac{\partial Loss}{\partial W^{(L-1)}} = \frac{\partial Loss}{A^{(L)}} \cdot \frac{\partial A^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial A^{(L-1)}} \cdot \frac{\partial Z^{(L)}}{\partial A^{(L-1)}} \cdot \frac{\partial A^{(L-1)}}{\partial Z^{(L-1)}} \cdot \frac{\partial Z^{(L-1)}}{\partial W^{(L-1)}}$$

It seems like a pattern is emerging here, as if we are almost following the forward computation backwards starting at the output and going backwards to the weights. This is the **back-propagation** algorithm, we propagate the gradients backwards through the network layers or in general back through the computation graph.

If we extract the pattern out it intuitively reads:

$$\frac{\partial Loss}{\partial W^{(l)}} = \text{gradient w.r.t. my output} \times \text{gradient w.r.t. my weights}$$

Why? When you *wiggle* (change the value of) $W^{(L-1)}$, it will affect $Z^{(L-1)}$ which will affect $A^{(L-1)}$ which is the input to the next layer which affects $Z^{(L)}$ and so on. This occurs simply because that is how we *wired* things together, one layer connected to another.



Now we will do the full derivation of a linear layer $Z = XW + B$ where $X \in \mathcal{R}^{N \times D}$, $W \in \mathcal{R}^{D \times M}$ and $B \in \mathcal{R}^{N \times M}$ such that $Z \in \mathcal{R}^{N \times M}$. The bias B is the stacked version of $b \in \mathcal{R}^{1 \times M}$ to match the shapes. What is going on, why are we multiplying XW and not the other way around. **In practice, we never compute single data point at a time, we calculate everything in vector form.** So to compensate for that and keep things clear, we change the multiplication to be on the right XW so the shapes match nicely. Consider $N = D = 2; M = 3$, let's write things out clearly:

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

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

$$B = \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

$$Z = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

We only require that $\frac{\partial Loss}{\partial Z}$ is given to us. That gradient is passed by the upper layer which is using our outputs as their input. That is the gradient that comes our way, we have multiple outputs so that is also a matrix! Not to panic (yet):

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

tells us the gradient with respect to this layer's outputs.

We are interested in the gradients with respect to W, b and X. We need $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$ because we are going to be using them to *learn*, and we need $\frac{\partial L}{\partial X}$ to able to pass backwards (we can't be selfish) to any other layers before us, if any. By the chain rule we know:

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

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

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

In more detail:

$$\frac{\partial Loss}{\partial X} = \begin{bmatrix} \frac{\partial Loss}{\partial x_{1,1}} & \frac{\partial Loss}{\partial x_{1,2}} \\ \frac{\partial Loss}{\partial x_{2,1}} & \frac{\partial Loss}{\partial x_{2,2}} \end{bmatrix}$$

Since the loss is scalar value, notice how the derivative $\frac{\partial Loss}{\partial X}$ has the same shape as X, we have one value per input. This identical shape property holds for other partial derivatives too since they are all partials of the loss function which is a scalar.

Let's take just *one* element and investigate:

$$\frac{\partial Z}{\partial x_{1,1}} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ 0 & 0 & 0 \end{bmatrix}$$

$$\frac{\partial Z}{\partial x_{2,1}} = \begin{bmatrix} 0 & 0 & 0 \\ w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix}$$

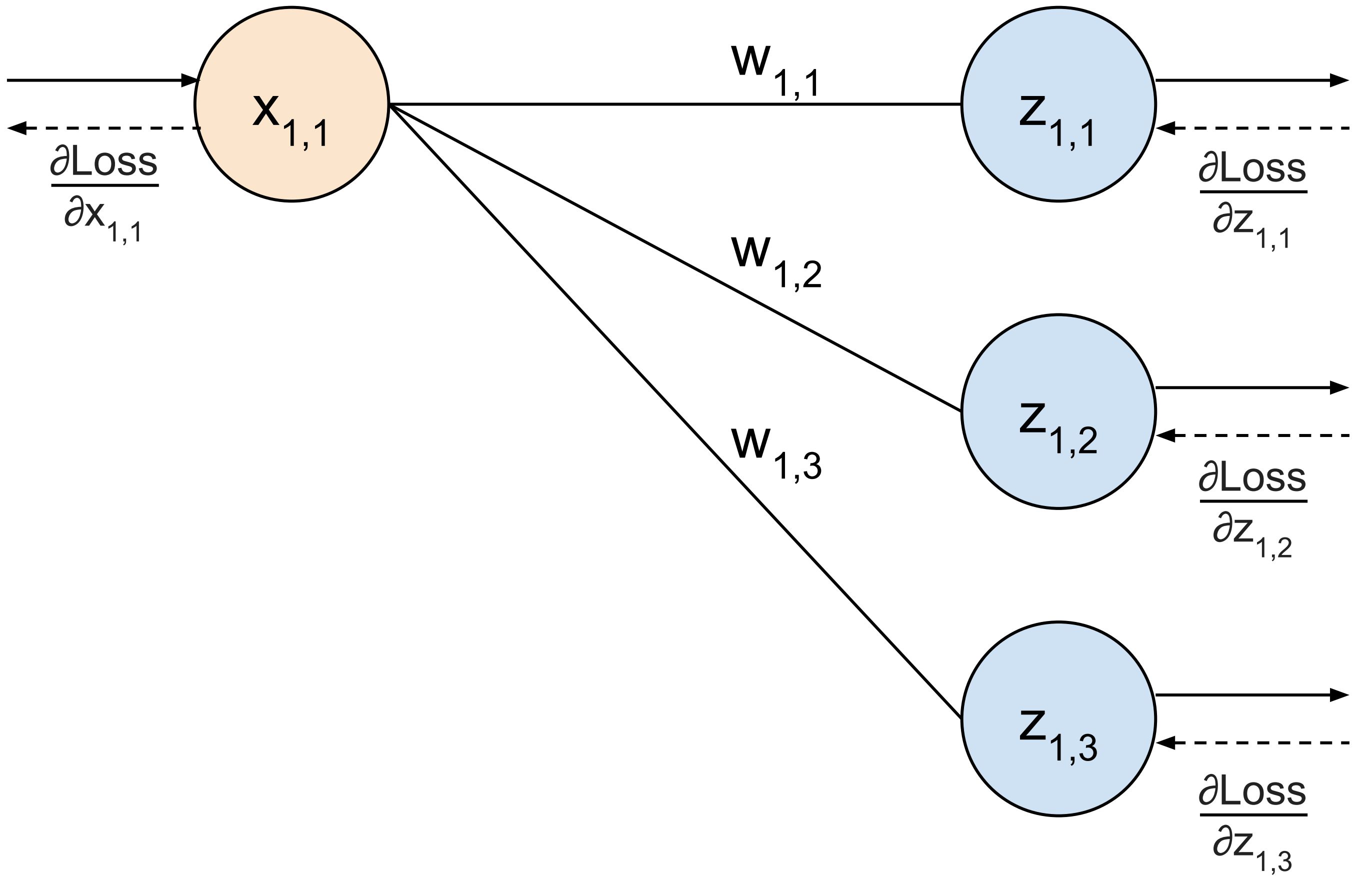
This can be obtained by looking at Z finding where $x_{1,1}$ or $x_{2,1}$ occurs and taking the individual derivatives.

Let's plug it into the chain rule as well for just one element:

$$\frac{\partial Loss}{\partial x_{1,1}} = \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} & \frac{\partial Loss}{\partial z_{1,3}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} & \frac{\partial Loss}{\partial z_{2,3}} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ 0 & 0 & 0 \end{bmatrix}$$

Note the general dot-product here, we are just multiplying every row with every other row and summing up, which gives us:

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



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

Intuitively, how $x_{1,1}$ affects the loss is determined by the weights it multiplies with and then whatever is using that output upstream. Imagine the paths from $x_{1,1}$ that lead to the loss function.

Now we can repeat this for every element of X and we get:

$$\frac{\partial Loss}{\partial X} = \begin{bmatrix} \frac{\partial L}{\partial z_{1,1}} w_{1,1} + \frac{\partial L}{\partial z_{1,2}} w_{1,2} + \frac{\partial L}{\partial z_{1,3}} w_{1,3} & \frac{\partial L}{\partial z_{1,1}} w_{2,1} + \frac{\partial L}{\partial z_{1,2}} w_{2,2} + \frac{\partial L}{\partial z_{1,3}} w_{2,3} \\ \frac{\partial L}{\partial z_{2,1}} w_{1,1} + \frac{\partial L}{\partial z_{2,2}} w_{1,2} + \frac{\partial L}{\partial z_{2,3}} w_{1,3} & \frac{\partial L}{\partial z_{2,1}} w_{2,1} + \frac{\partial L}{\partial z_{2,2}} w_{2,2} + \frac{\partial L}{\partial z_{2,3}} w_{2,3} \end{bmatrix}$$

Extracting out the gradients that were passed to us, we get:

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

$$\frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Z} W^T$$

Phew, that wasn't all that bad. You'll see people replacing intermediate results with different symbols such as δ to encapsulate some steps or use the ∇ operator from vector calculus, we just went all guns blazing and wrote the thing out instead of making a soup Greek letters.

In a similar fashion, we can look at a single weight:

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

which captures how $w_{1,1}$ affects the output and when we perform the same analysis on every weight, we get this beauty:

$$\frac{\partial Loss}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial z_{1,1}} x_{1,1} + \frac{\partial L}{\partial z_{2,1}} x_{2,1} & \frac{\partial L}{\partial z_{1,2}} x_{1,1} + \frac{\partial L}{\partial z_{2,2}} x_{2,1} & \frac{\partial L}{\partial z_{1,3}} x_{1,1} + \frac{\partial L}{\partial z_{2,3}} x_{2,1} \\ \frac{\partial L}{\partial z_{1,1}} x_{1,2} + \frac{\partial L}{\partial z_{2,1}} x_{2,2} & \frac{\partial L}{\partial z_{1,2}} x_{1,2} + \frac{\partial L}{\partial z_{2,2}} x_{2,2} & \frac{\partial L}{\partial z_{1,1}} x_{1,2} + \frac{\partial L}{\partial z_{2,1}} x_{2,2} \end{bmatrix}$$

We again extract out the gradients that were given to us:

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

$$\frac{\partial Loss}{\partial W} = X^T \frac{\partial Loss}{\partial Z}$$

Similarly when we look at the bias we obtain:

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

but the individual steps of doing so is left for you as an exercise. Above, $\mathbf{1}$ is a column vector of ones.

Once the derivation is done, all the deep learning libraries nowadays compute the gradients automatically using these derivatives. For example, the AutoGrad package of PyTorch is dedicated for this. Let's look at an example using Chainer:

```
# For a more concrete example consider
# this snippet using the Chainer library
import chainer
x_data = np.array([5], dtype=np.float32)
x = chainer.Variable(x_data)
# this is the forward computation
y = x**2 - 2*x + 1
print(y) # variable([16.])
# compute and backpropagate gradients
y.backward()
# gradients propagated to x
x.grad == [8.]
# --- what is y.grad ? ---
```

What about an activation function? Suppose we again have $Z = f(X)$ where $X \in \mathcal{R}^{N \times D}$ and $Z \in \mathcal{R}^{N \times D}$. Note we are applying the function to every element:

$$Z = f(X) = \begin{bmatrix} f(x_{1,1}) & f(x_{1,2}) \\ f(x_{2,1}) & f(x_{2,2}) \end{bmatrix}$$

This time we only need the derivative with respect to X, and using the chain rule we know:

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

Applying the same one element at a time analysis, we look at $x_{1,1}$:

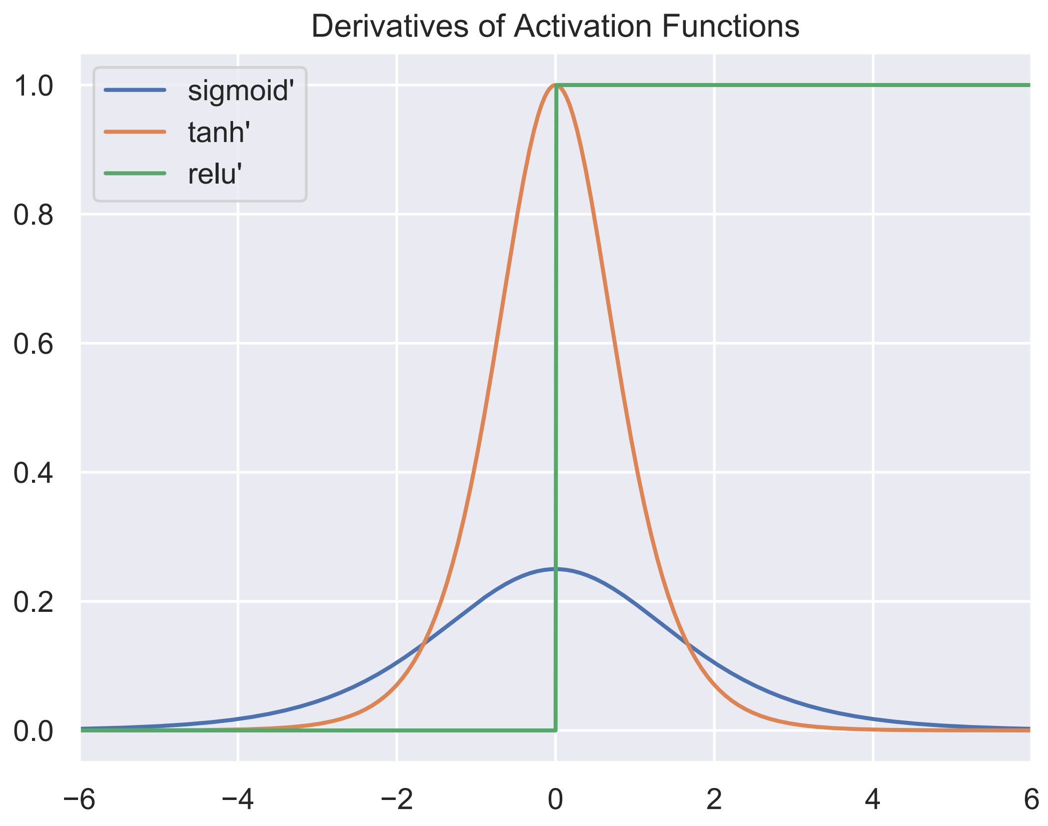
$$
\begin{aligned}
\frac{\partial Loss}{\partial x_{1,1}} &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} \end{bmatrix} \cdot \begin{bmatrix} f'(x_{1,1}) & 0 \\ 0 & 0 \end{bmatrix} \\
&= \frac{\partial Loss}{\partial z_{1,1}} f'(x_{1,1})
\end{aligned}
$$

where f' is the derivative of our activation function. From an intuitive point of view, *our input only affects one output and that depends on how much the activation function affects the output*.

Generalising to matrix form again:

$$
\begin{aligned}
\frac{\partial Loss}{\partial X} &= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} f'(x_{1,1}) & \frac{\partial Loss}{\partial z_{1,2}} f'(x_{1,2}) \\ \frac{\partial Loss}{\partial z_{2,1}} f'(x_{2,1}) & \frac{\partial Loss}{\partial z_{2,2}} f'(x_{2,2}) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial Loss}{\partial z_{1,1}} & \frac{\partial Loss}{\partial z_{1,2}} \\ \frac{\partial Loss}{\partial z_{2,1}} & \frac{\partial Loss}{\partial z_{2,2}} \end{bmatrix} \circ \begin{bmatrix} f'(x_{1,1}) & f'(x_{1,2}) \\ f'(x_{2,1}) & f'(x_{2,2}) \end{bmatrix} \\
&= \frac{\partial Loss}{\partial Z} \circ f'(X)
\end{aligned}
$$

where ○ is **element-wise** multiplication (also known as Hadamard product). Just like the linear layer, this holds for every activation function. *One equation to rule them all*.



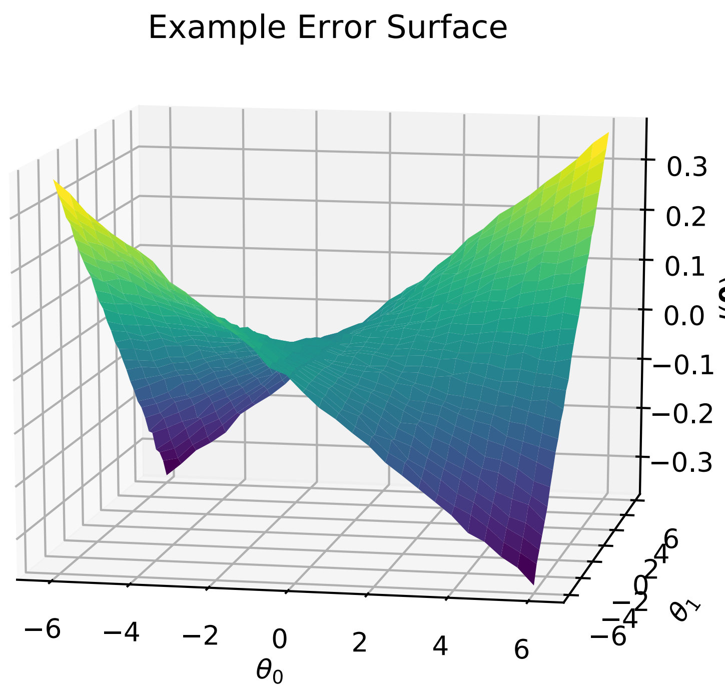
Here are derivatives of some activation functions:

- **linear:** $f'(x) = 1$ for any given x.
- **sigmoid (logistic):** $f'(x) = f(x)(1 - f(x))$ really neat no? The derivative can be defined using it self.
- **tanh:** $f'(x) = 1 - f^2(x)$ again we can define the derivative using the function itself here.
- **ReLU:** $f'(x) = 1$ if $x > 0$ otherwise it is just 0 as well. Looking at the function gives it away.

This is also an appropriate time to discuss why **ReLU** is preferred. Look at the gradients of **sigmoid** and **tanh** they strictly range between 0 and 1. You multiply these as you back-propagate which *causes the gradients to vanish*. This is the **vanishing gradient problem** and it makes training deep neural networks very difficult.

Gradient Descent

We just iteratively descend in the direction of the gradient slowly minimising the loss which is referred to as **gradient descent**. Gradient descent is a general optimisation technique and is not in any way special to neural networks. You can optimise linear models using their gradients as well for example, or anything you can compute the gradient of.



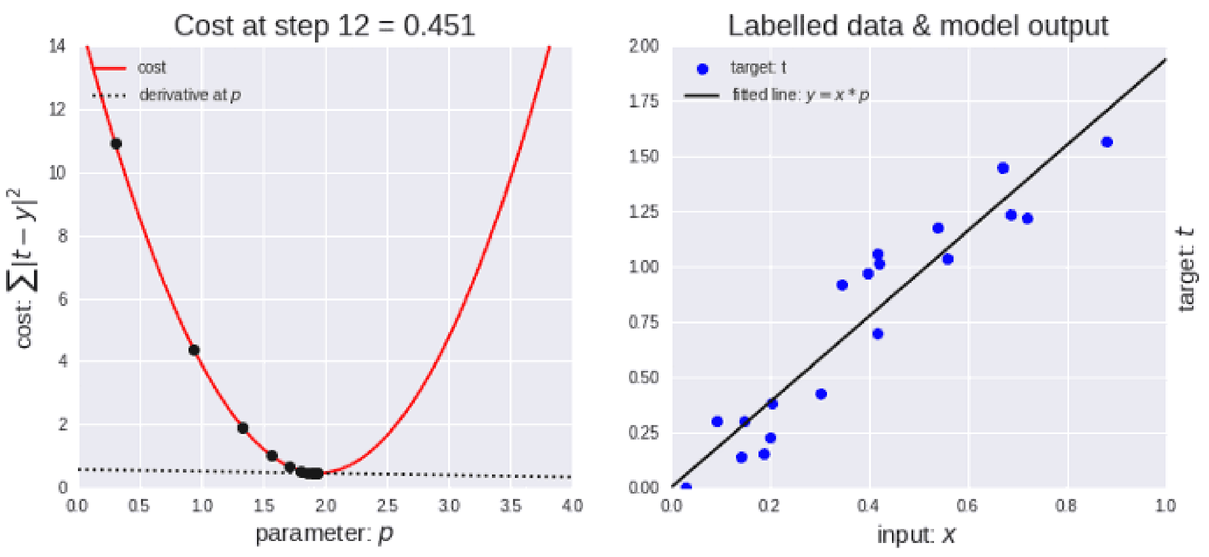
We start with random weights / parameters θ then we compute the gradients with respect to some data and loss function, also called an **objective function** denoted $J(\theta)$, adjust weights in that direction and repeat until we reach some minimum. A non-linear function can have multiple **local minimum** with respect to the objective function. There is *no guarantee we will converge to the global minimum* when just walk in the direction of the gradient.

> **Pro Tip:** You see a village in the distance and start walking in that direction. Although you will get closer to the village that doesn't mean you will arrive at it, you could get stuck at some hill or cliff.

So what do we actually do? We update the weights using the gradient:

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

where α is the **learning rate** that tells us how much of a leap we should take in that direction. At each update, we slowly adjust the weights in the direction that minimises the loss that is given by the gradient we computed. We often set $\alpha = 0.01$ or 0.001, essentially something small but it depends on the problem.



An important observation is that **we assumed that the gradient could be computed for every parameter**. This is very important, we need our functions, network and loss to be **differentiable** in order to use this method. What if they are not? There are other algorithms such as genetic algorithms that do not use the gradient to optimise the network parameters.

Ideally we would like to compute the gradient using the entire training data, but that is computationally expensive and slow if for example we have thousands of data points. Instead **we estimate the true gradient using small random batches**:

- **Stochastic Gradient Descent (SGD):** in it's vanilla form takes one random data point and immediately updates the weights using that gradient.
- **Batch Gradient Descent:** uses the entire data set and then updates the weights.
- **Mini-batch Gradient Descent:** takes small batches such as 32 data points at a time and then updates the weights.

But in practice we just refer to **mini-batch** as **batch** and **mini-batch gradient descent** as **stochastic gradient descent**. This is the ugly truth:

```
# Keras library
network.compile('sgd', 'mse') # mse is mean squared error here
network.fit(X, Y, batch_size=32, epochs=40, shuffle=True)
```

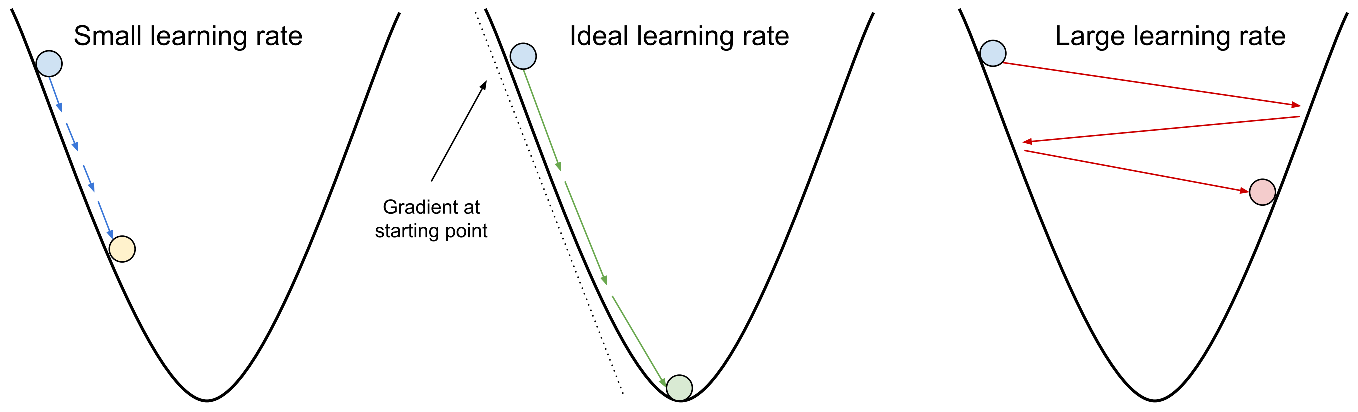
where `batch_size` is actually the *mini-batch* size, the first dimension of the input $X \in \mathcal{R}^{N \times D}$.

Let's break down what is going on here:

1. We take training data of size N each with D features, a matrix $N \times D$ if you will.
2. We shuffle it so the network doesn't see the same data points grouped together, this helps with convergence since different combinations of data points can give different useful gradient estimations.
3. We chop it into batches of size B, so we get a batch of shape $B \times D$. How many batches are there? N/B many.
4. We compute the forward pass to collect the network output *one batch at a time*.
5. We compute the derivative of the loss with respect to the network outputs.

6. We then back-propagate the gradients to compute the derivative of the loss with respect to *every* parameter in the network.
7. We update the weights using the given learning rate.
8. We repeat 4 to 7 for every batch.
9. Once we do this for every batch, we finished an **epoch**, a full run over our training data. Now we repeat from 2 to 8 until we have done the required number of epochs or reached a convergence criteria.

Anything that is not learned and is set by the user is a **hyper-parameter**. It is up to the user to *find*, *guess* or *tune* until this entire training process gives desirable outputs.

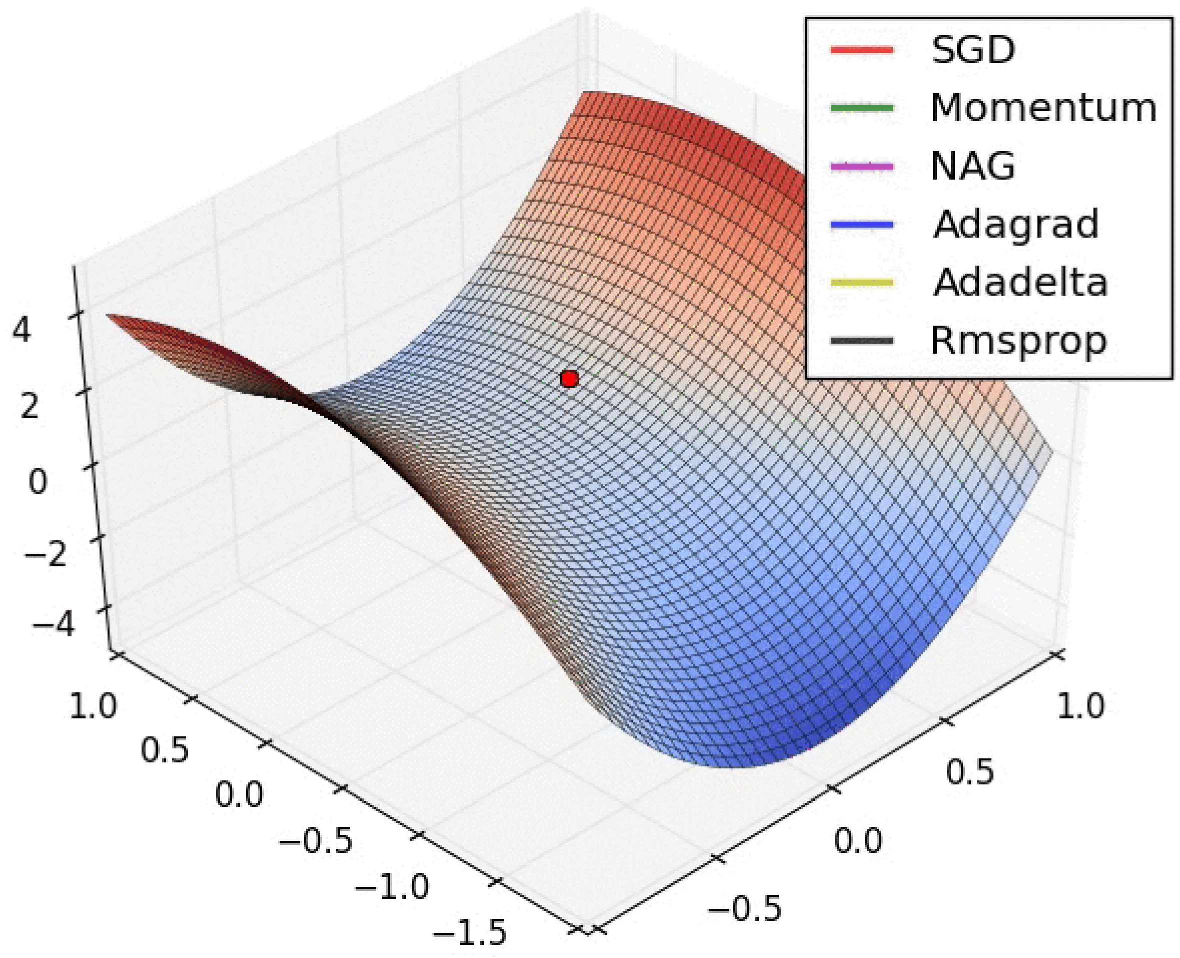


Although currently the **learning rate** is fixed, there are many extensions to gradient descent that make it adaptive such that the training becomes more efficient. The simplest of them is **learning rate decay** which reduces the learning rate by a factor each epoch:

$$\alpha \leftarrow \alpha d$$

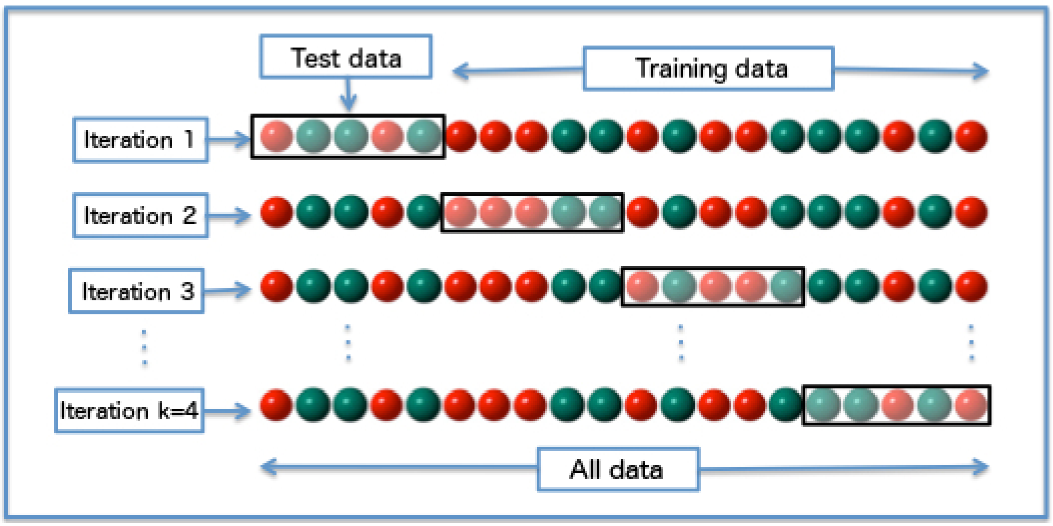
where $d \in [0, 1]$. The idea is we reduce the rate as we approach the minimal loss, so take smaller steps to not overshoot.

More sophisticated methods such as having a one learning rate per parameter also exist and they span their own family of gradient descent algorithms such as RMSProp, AdaDelta and Adam which are not within the scope of this module.



Evaluating

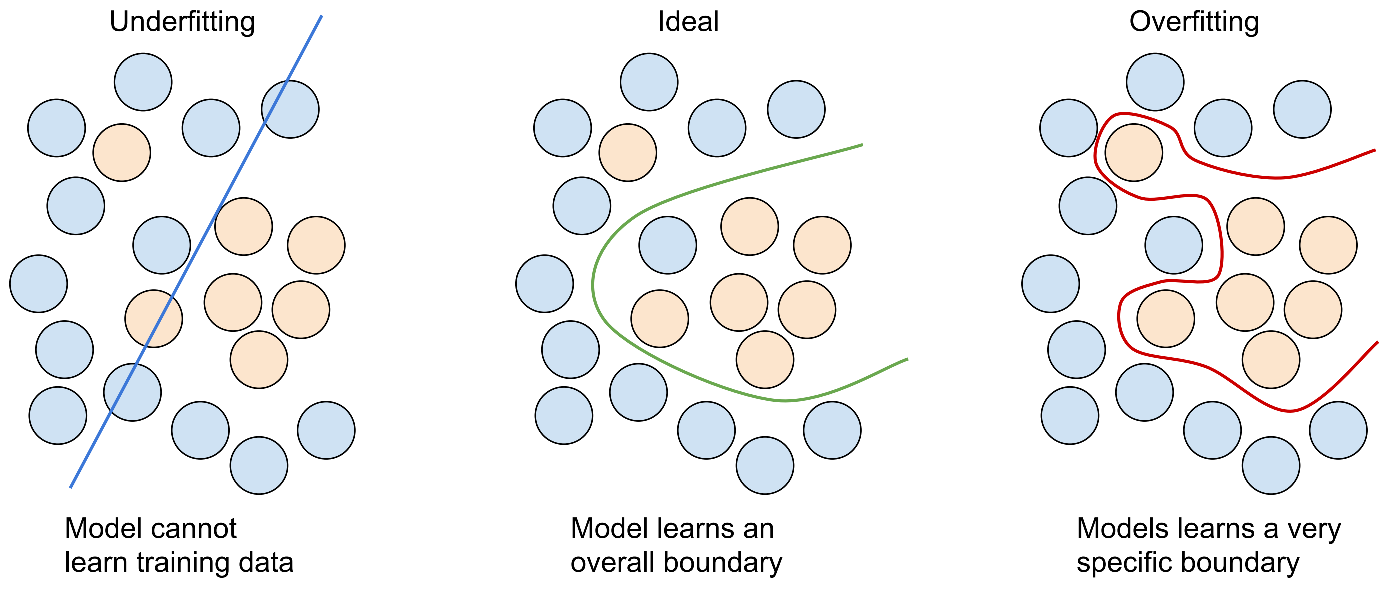
So you are training your neural networks and everything is fine, the loss is going down, the model seems to be learning something. **Is it????** To ensure that our networks, models are learning something beyond just the **training data** which is what we use to train with, we need **test or validation data** that are separate from training. **The model, network never ever trains on the test or validation data points!**

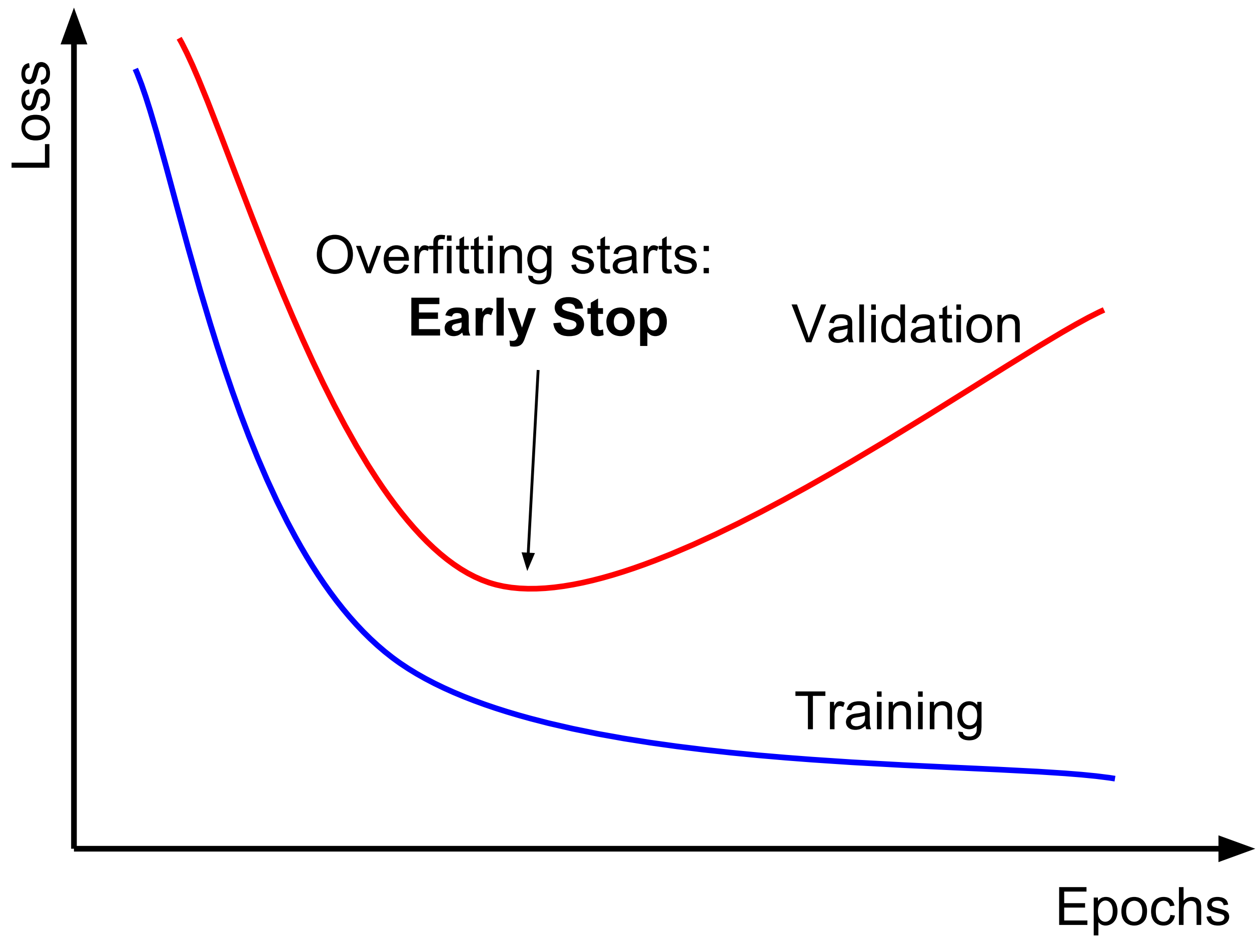


Where on Earth do we get validation or test data? We partition the overall data set. For example if we have 10000 data points, we could do a 0.1, 0.1 test validation split to get 1000 test points, 1000 validation points and 8000 training points. Keras has the validation split option built-in, `model.fit(X, Y, validation_split=0.1, ...)` for example will split the overall data X and Y into two partitions of ratios 0.1 and 0.9 for validation and train respectively. *Some data sets come with pre-set train, validation and test splits.* We also often use **k-fold cross validation** which just means we take k many different random train-validation-test splits / folds (preferably non-overlapping) to reduce bias.

The more important question is **why** do we need this separation. Neural networks can learn very complex non-linear functions, which means they are in danger of learning patterns specific to or even memorising the training data. Imagine we have 100 data points and a neural network with 100 units in the hidden layer. Every unit can learn which input it is and we can predict based on that, no actual learning that we are interested in is done.

Over-fitting


Underfitting
Ideal
Overfitting
Model cannot learn training data
Model learns an overall boundary
Models learns a very specific boundary



One method for avoiding over-fitting is to simply stop early. Very cleverly, this is called **early stopping**. Keras implements this in `keras.callbacks.EarlyStopping` and it is a very simple but very useful tool.

> **Pro Tip:** If you come up with a very useful algorithm that is very simple like early stopping, you are allowed to name it early stopping. For example, a door stopper is a very very useful thing and it is aptly named a door stopper.

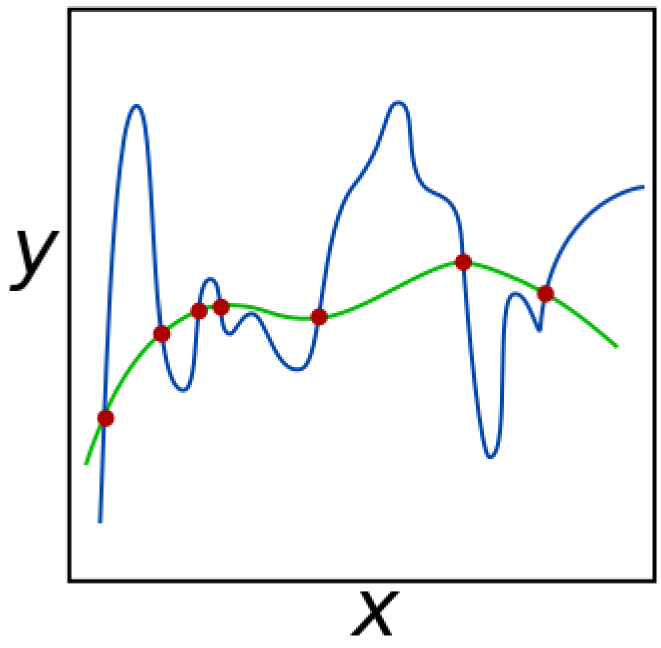
There is a correlation between the network **capacity** and over-fitting. The capacity refers to the number of layers and units those layers have. More layers, more units means the network has more parameters and therefore a bigger capacity to learn more complex functions. Here is some advice on what to do:

- if the network is not learning on the training data, i.e. the loss is not decreasing at all and it's under-fitting, then it might not have enough capacity to learn. We can try increasing the number of units and/or the number of hidden layers.
- If the network is over-fitting, that is the validation is increasing while the training loss is decreasing, it has too much capacity and we can look at either reducing it by lowering the number units, layers or apply regularisation.

Best solution to any over-fitting problem is to GET MORE DATA. But this is rarely the case in practice... We don't have infinite number of cat images (thank goodness).

Regularisation

Regularisation is way to penalise the model in some way to stop it from over-fitting. Another way of looking at it is we are reducing the effective capacity of the network for example by penalising how large the weights could be.



One way the model can over-fit is if it's weights are allowed to grow out of control. **Why?** If there is a very useful feature that describes only the training data, the network will latch onto the information to learn the function, but the validation data might not have that. So we end up with the blue line in the image, it aggressively finds a way to learn the training data.

$$J(\theta) = Loss(Y, A) + \lambda \sum_w w^2$$

$$w \leftarrow w - \alpha(\frac{\partial Loss}{\partial w} + 2\lambda w)$$

The **L2 regularisation** adds the squared weight w^2 to the objective function along side the loss. So if the weight gets larger, so does the loss, in effect to reduce the loss the network needs to keep the weights small as well. The λ tells how much we want to regularise, we usually set it to 0.01. The effect is seen on the weight update rule, now the update is proportional to the weight itself; thus large weights shrink proportionally faster.

$$J(\theta) = Loss(Y, A) + \lambda \sum_{w} |w|$$

$$w \leftarrow w - \alpha(\frac{\partial Loss}{\partial w} + \lambda \text{sign}(w))$$

The **L1 regularisation** just adds the absolute value of the weight itself to the objective function. The update rule now considers a fixed movement towards 0. If the weight is negative it is always nudged upwards, if it is positive it is always nudged downwards by a fixed λ amount towards 0. Both L1 and L2 are sometimes are referred to as **weight decay** which stems from the fact that they both decay the weights towards 0.

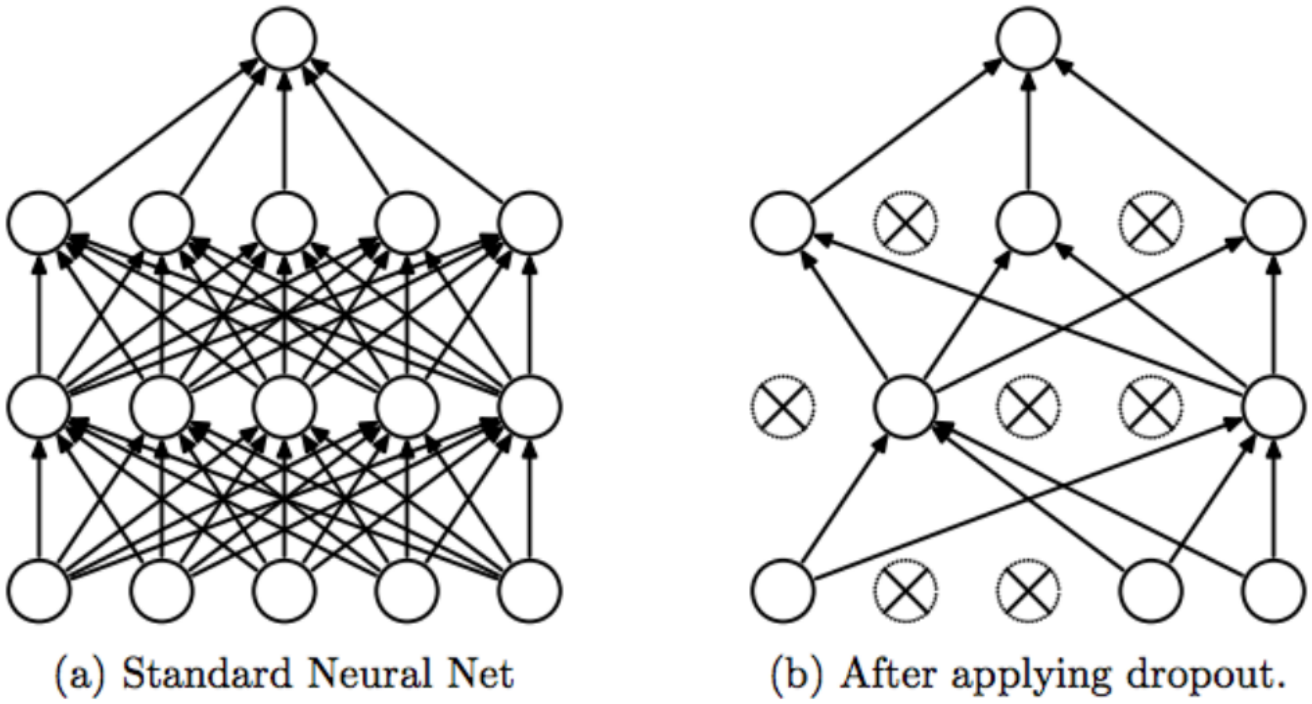
```
# From Keras library source code, I didn't write this, it is the library itself
class L1L2(Regularizer):
"""Regularizer for L1 and L2 regularization.
# Arguments
l1: Float; L1 regularization factor.
l2: Float; L2 regularization factor.
"""
def __init__(self, l1=0., l2=0.):
  self.l1 = K.cast_to_floatx(l1)
  self.l2 = K.cast_to_floatx(l2)
def __call__(self, x):
  regularization = 0.
  if self.l1:
    regularization += K.sum(self.l1 * K.abs(x))
  if self.l2:

    regularization += K.sum(self.l2 * K.square(x))
return regularization
# This gets added to the overall objective function
```

So why does this help with over-fitting? Recognise that both methods push weights towards 0, a weight of 0 means no connection. No connection means less capacity so in effect we are simplifying the network which removes its ability to learn very specific patterns of the training set.

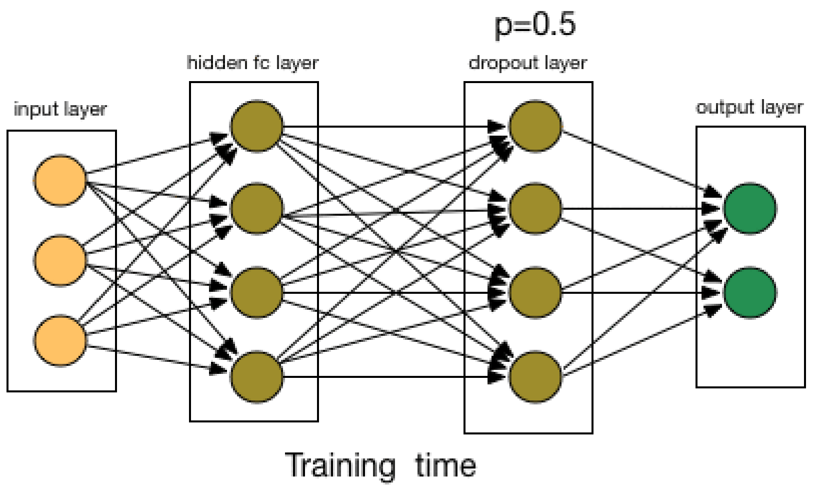
One common feature mentioned for the L1 regulariser is that it produces sparse weights. Intuitively it pushes most of the weights to 0, so only the most useful features will need to have non-zero weights for the network to learn / predict the patterns in the data. This **sparsity** leads to what is called **feature selection**, the training causes the layer on which L1 regularisation is applied to select few inputs to produce an output in order to keep the weights small. L2 on the other hand ensures smoothness, so it encourages a combination of inputs / features to be used since weights are not forced towards 0 when they are already small.

Dropout

So if pushing the weights towards 0 essentially cuts the connection and reduces the capacity of the network, why don't we just remove them entirely and not have fully-connected layers? Well, that is exactly what **dropout** does: it randomly sets outputs of layers to 0 essentially turning off neurons. It is very simple, don't believe me? Here is the title of the paper that published the idea -> Dropout: A Simple Way to Prevent Neural Networks from Overfitting:



With probability p we set the output of the neuron to 0. The name stems from the fact that we drop certain neurons by setting their output to 0. **It is as if we are training smaller networks inside a larger one** which reduces inter-dependency between neurons across layers since it might be dropped next round. Dropout is only applied at *training time*. At test or prediction time the entire network is active and no neuron is dropped but the weights or the activations might be scaled in order not to saturate neurons with all connections active.

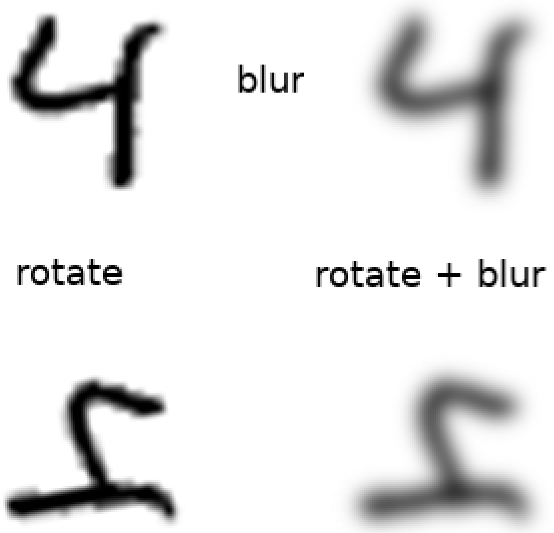


Data Pre-Processing

One common approach is to do **data augmentation** which enhances our existing data. For example if we have some numerical data we can introduce some noise to it:

$$X' = X + \mathcal{N}(\mu, \sigma^2)$$

Why does adding noise help? Because if there are any specific features / points the network learns, we now wiggle them around using some Gaussian noise so they are not static anymore. To look at it another way, we are generating artificial data based on the original data.



Other common ways of data augmentation include:

- Blurring, flipping images: a cat is still cat if it is slightly blurred or flipped horizontally.
- Replacing words with synonyms in sentences. Replacing happy with joyous still keeps the sentence positive for a sentiment classification. It might change how positive so we need to be careful what is acceptable data augmentation.

Perhaps a more common technique is **data normalisation** in which the input and potentially the output data is normalised. For example we can map the largest value to b and smallest value to a:

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

is called **feature scaling** because we apply a fix scaling on the data. We often reduce the range to $[0, 1]$ or $[-1, 1]$.

Similarly we can normalise using the mean and standard deviation of the data:

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

which makes the data have mean 0 and standard deviation 1. This normalisation is often applied when we know the input is normally distributed, such as height of people. It is also referred to as **z-normalisation, standard score or z-score normalisation** since we are mapping our data to the unit normal distribution $\mathcal{N}(0, 1)$.

Why does normalisation help?

$$\frac{\partial Loss}{\partial W} = X^T \frac{\partial Loss}{\partial Z}$$

Notice how the X, our actual input appears in the equation if this was the first hidden layer in network. So **updating weights which uses this gradient relies on the magnitude of the input**.

> **Pro Tip:** Just because someone says this is all the material you need to know for the exam doesn't mean this is *all* there is to know. You should explore beyond the boundaries of any given knowledge.