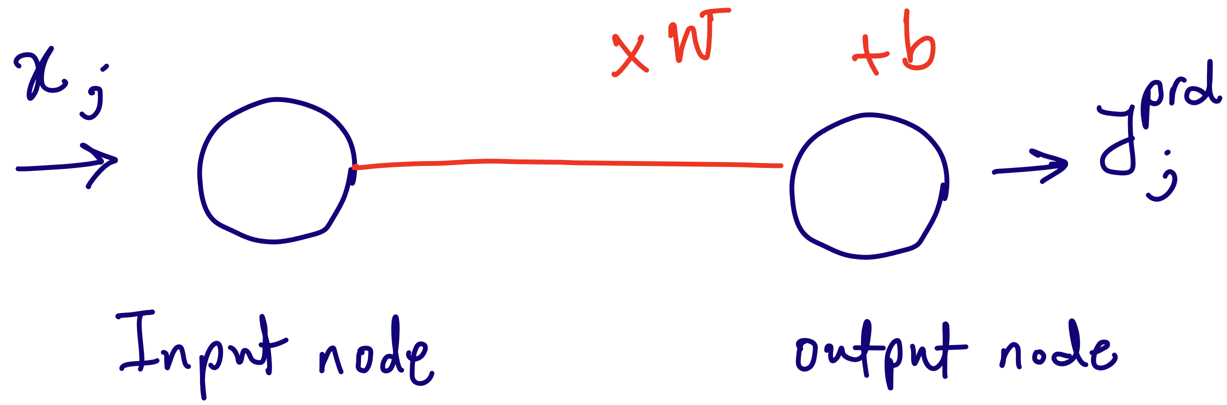
An Introduction to Neural Networks

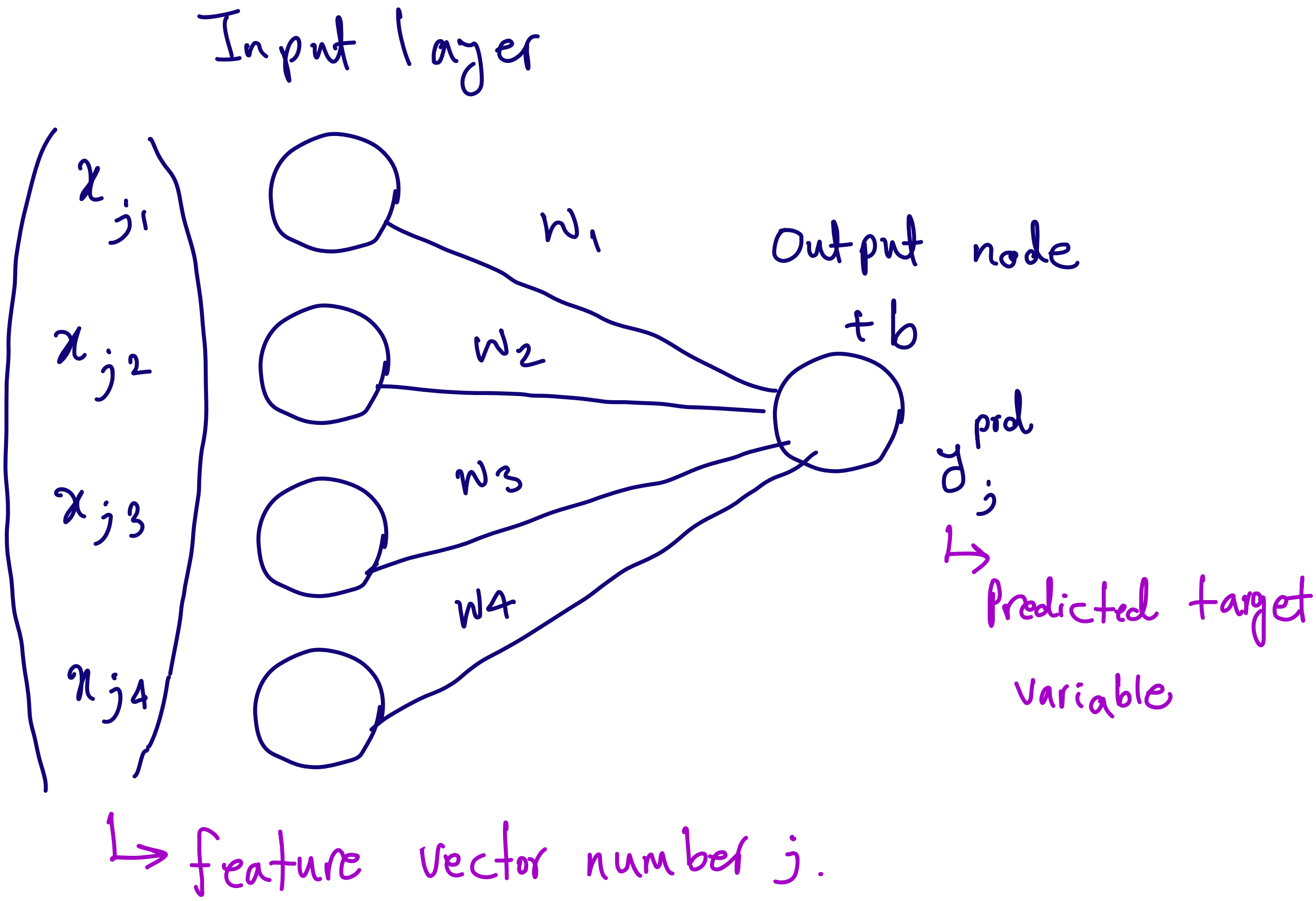
The simplest forms of a neural network



$$y_j^{prd} = \underbrace{w}_{\text{weight}} x_j + \underbrace{b}_{\text{bias}}$$

w and b are determined in the "training" phase.

The simplest forms of a neural network



$$y_j^{prd} = w_1 x_{j1} + w_2 x_{j2} + w_3 x_{j3} + w_4 x_{j4} + b$$

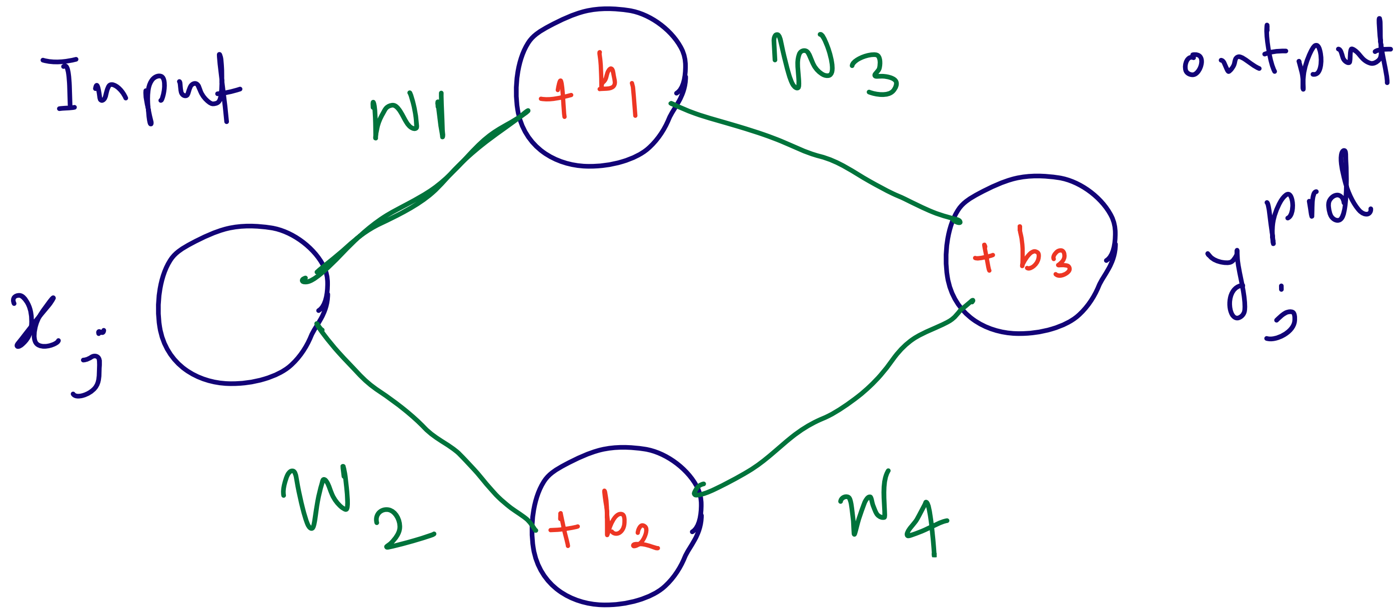
[Again, $w_1, \dots, w_4, b$ are determined during the training]

Hidden layers and activation functions (non-linear structures). A simple example

Hidden layer

$\Phi \rightarrow$ Activation function

$$\Phi(w_1 x_j + b_1)$$

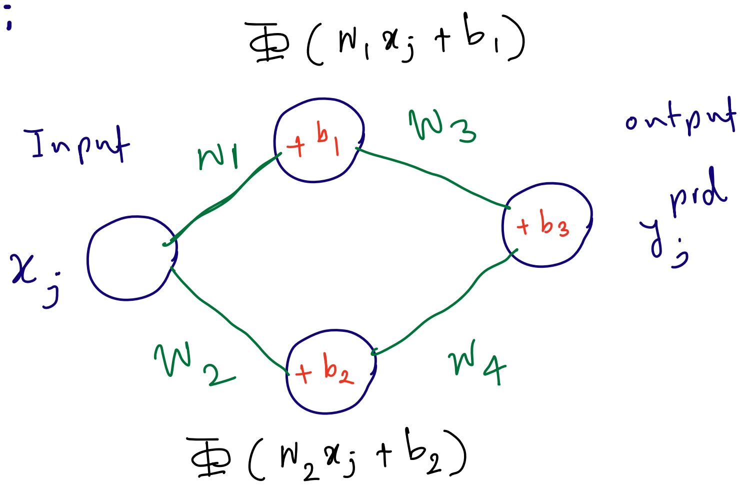


$$\Phi(w_2 x_j + b_2)$$

$$y_j^{prd} = w_3 \, \Phi(w_1 x_j + b_1) + w_4 \, \Phi(w_2 x_j + b_2) + b_3$$

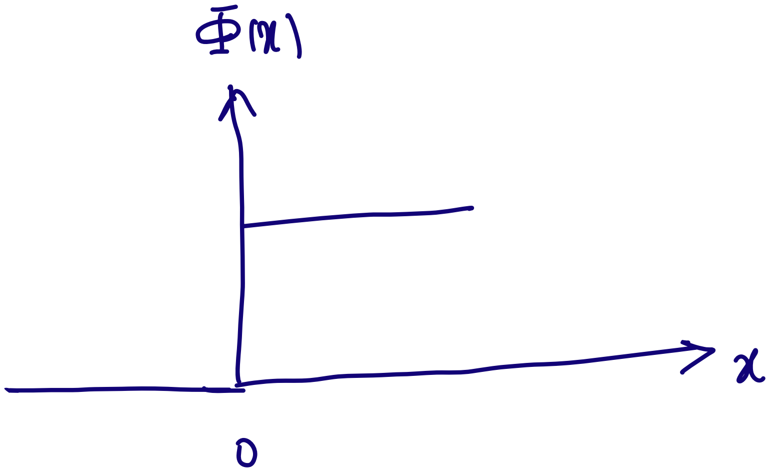
There are various types of activation functions.

Here are some examples:



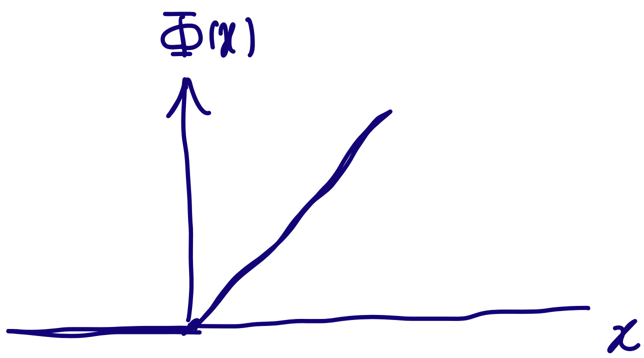
Threshold:

$$\Phi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



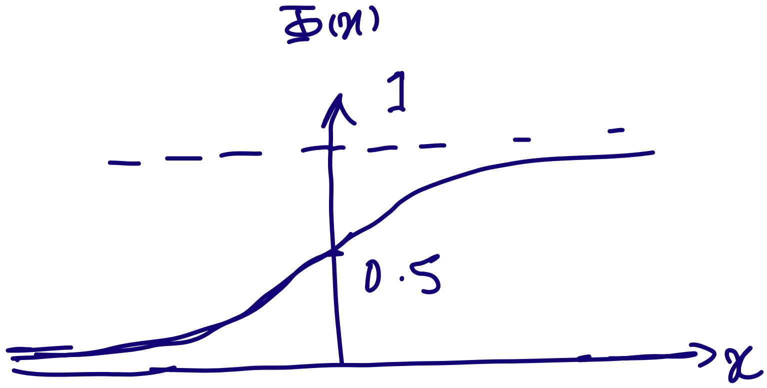
Rectified (ReLU):
Linear Unit

$$\Phi(x) = \max(0, x)$$



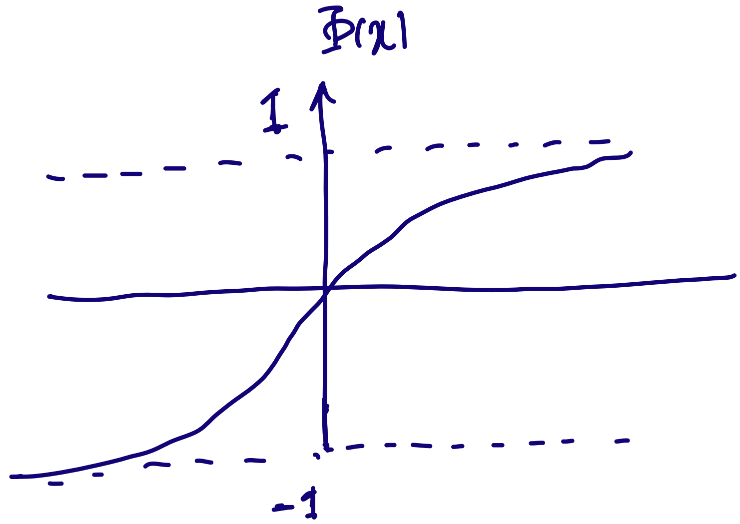
Sigmoid:

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



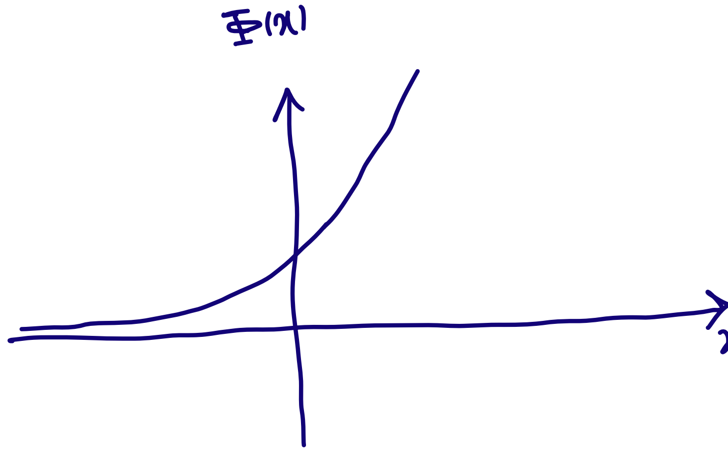
Tanh

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



Softplus

$$\Phi(x) = \ln\left[1 + e^{x}\right]$$



Backpropagation

Let's assume that we want to solve the following problem: For each dark matter halo with known mass (M_{vir}), maximum circular velocity (V_{max}) and spin (λ), we want to predict how many subhalos it hosts.

$\Rightarrow$ features for halo j: $x_{j1} = M_{vir,j}$, $x_{j2} = V_{max,j}$, $x_{j3} = \lambda_j$

Target for halo j: $y_j = N_{sub,j}$

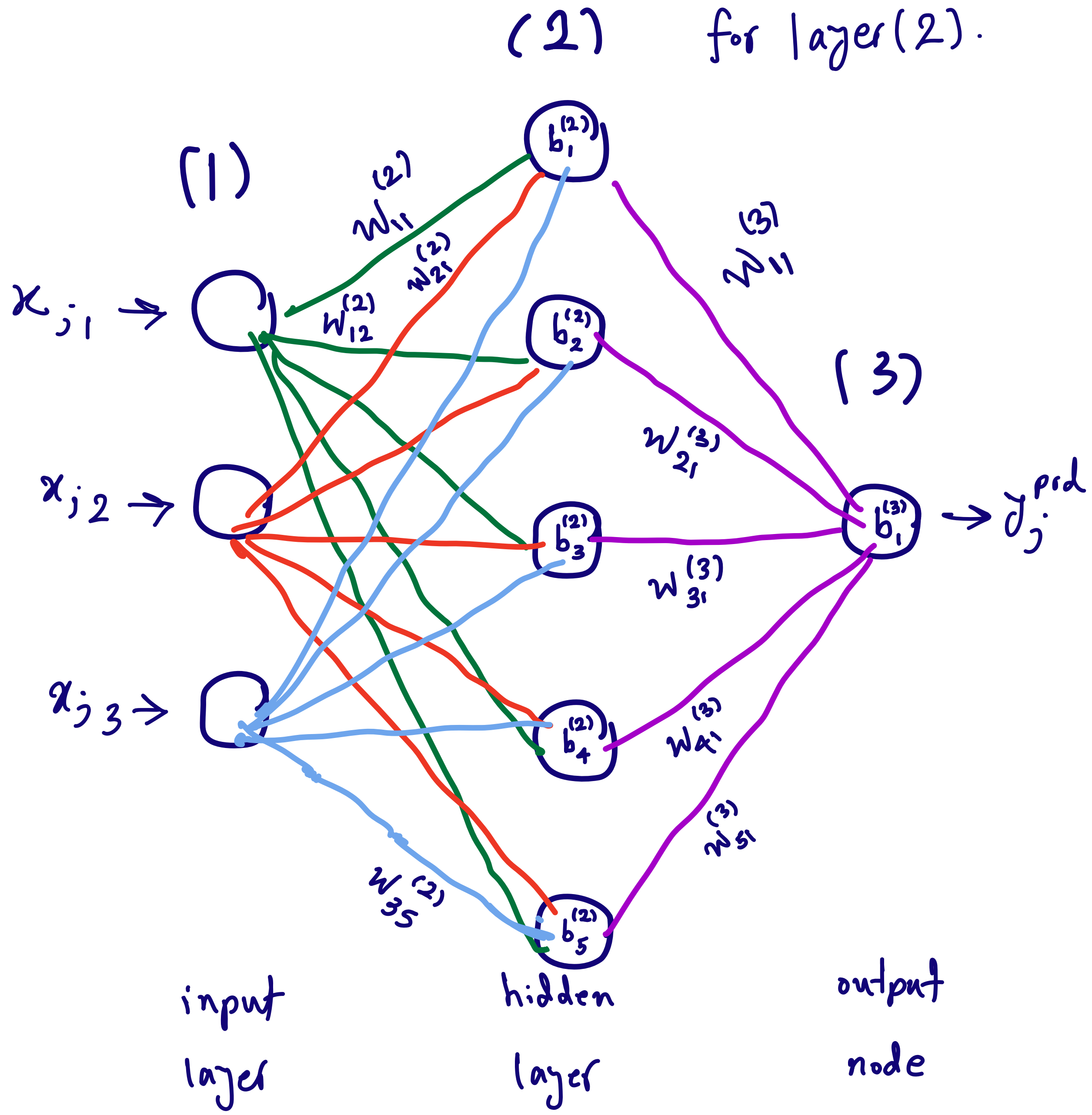
Feature matrix: $\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} \end{pmatrix}$ Each row corresponds to one halo

Target vector: $\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$

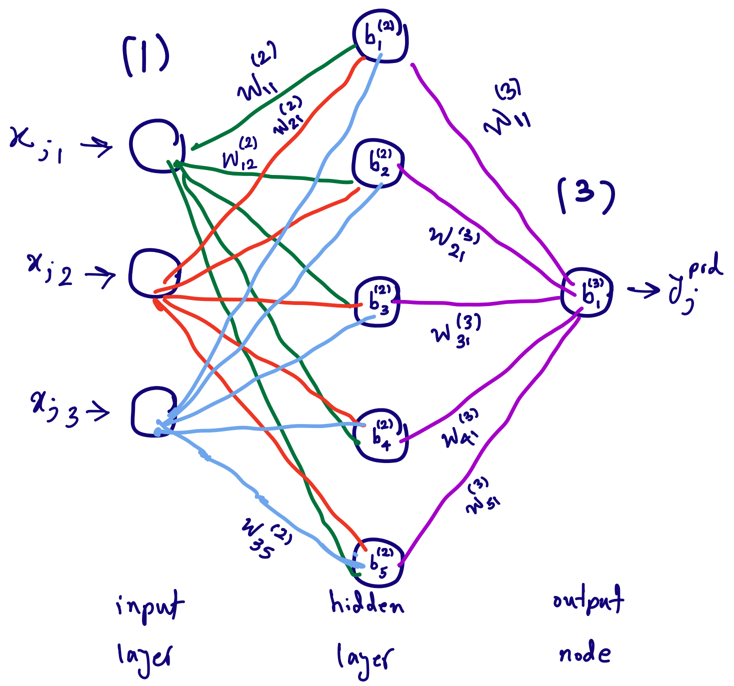
Now we can split the data into train and test sets (e.g. 80% train, 20% test). We use the training set to train a neural network (NN).

Let's consider a simple NN with just one hidden layer:

$\Phi^{(2)} \to$ Activation function for layer (2).



$\Phi^{(2)} \rightarrow$ Activation function for layer (2).



predicted target values

actual target values

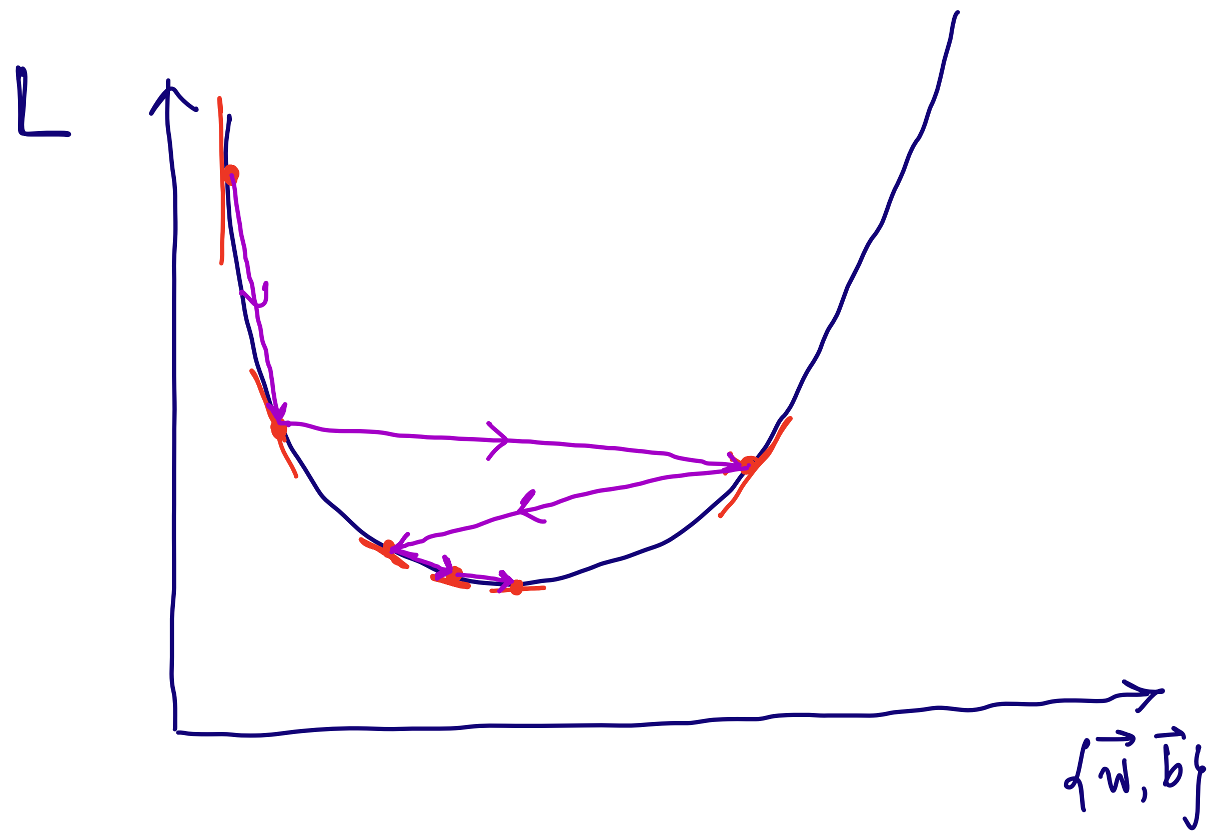
Loss function: (mean squared error)

$$L = \frac{1}{N_{tr}} \sum_{i=1}^{N_{tr}} \left(y_i^{prd} - y_i^{act}\right)^2$$

$\hookrightarrow$ Size of the train set

The aim is to minimize the loss function. One approach is using the Gradient Descent (GD) method.

Gradient Descent (GD)

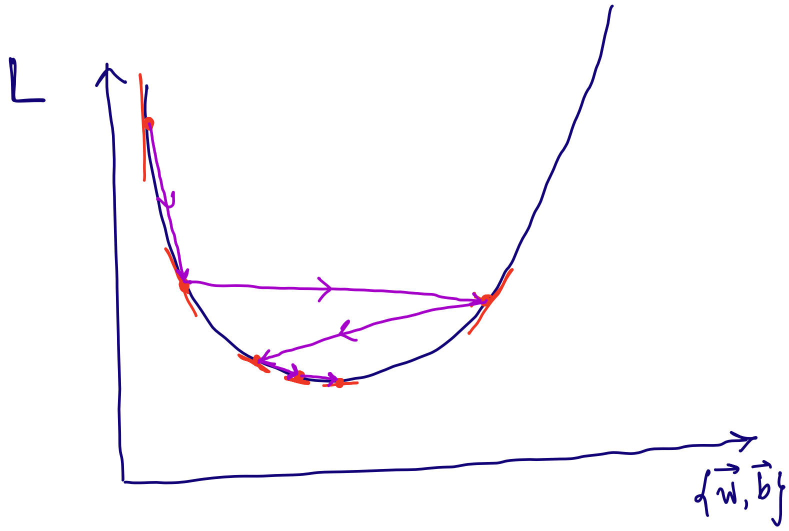


1) Initialize the parameters (weights and biases)
(Can be done by random sampling from, e.g., a standard Gaussian distribution)

2) Find the gradient of loss function with respect to the parameters (weights and biases).

In our example:

$$\vec{\nabla} L = \left(\frac{\partial L}{\partial w_{11}^{(2)}}, \frac{\partial L}{\partial w_{12}^{(2)}}, \ldots, \frac{\partial L}{\partial b_1^{(2)}}, \frac{\partial L}{\partial b_2^{(2)}}, \ldots, \frac{\partial L}{\partial b_1^{(3)}} \right)$$



3) For a given parameter θ (θ can be any of the weights or biases),

update the parameter via: $$\theta_{new} = \theta_{old} - \overbrace{\eta \left.\frac{\partial L}{\partial \theta}\right|_{\{\vec{w}_{old}, \vec{b}_{old}\}}}^{\text{step size}}$$

$\eta \rightarrow$ learning rate. Can be a fixed value (e.g. 0.001) or can be adaptive.

* All parameters are updated simultaneously. *

4) Repeat step 3 until convergence (e.g. when step size becomes too small or when a maximum allowed number of steps is reached). In other words, we stop when loss function is small and does not change meaningfully with additional steps (we monitor the values of loss function after each update)

These updating processes "train" the neural network.
This process is referred to as backpropagation.

Stochastic Gradient Descent (SGD)

Gradient descent becomes very slow when the network is large and/or data sets are large.

To solve this, people can use Stochastic Gradient Descent (SGD): Instead of using the entire training set for updating the parameters at each step, one can construct a loss function assuming that there is one feature vector and its target variable, then randomly select a feature vector and its target variable and perform parameter updating accordingly. Then repeat this process with new, randomly selected, feature vectors and their target variables (one at each step).

* A middle ground between GD and SGD is mini batching (Commonly used):

Instead of just one feature vector and its target variable at each step, one can divide the training set into smaller batches and, at each step, perform the parameter updating based on the data points in one of the batches (one batch at each step). People often use this.

- - - - - - - - - - - - - - -

Terminology: In any of the optimization methods, every time the entire data set (from the training set) is passed through the network for optimization is called an epoch. People usually use several epochs for training.

Some additional points

* For converging faster, reducing oscillations and avoiding getting stuck in local minima of the loss function, pepole add "momentum" to the optimization process:

updating process

$$\underbrace{\theta_{t+1}}_{\text{updated parameter at step } t+1} = \underbrace{\theta_t}_{\text{parameter at step } t} + \underbrace{v_t}_{\text{"velocity" at step } t \text{ (has the memory of past steps)}}$$

$$v_t = -\underbrace{\eta}_{\text{learning rate}} \frac{\partial L}{\partial \theta}(t) + \underbrace{\beta}_{\text{momentum coefficient (e.g. 0.9)}} v_{t-1}$$

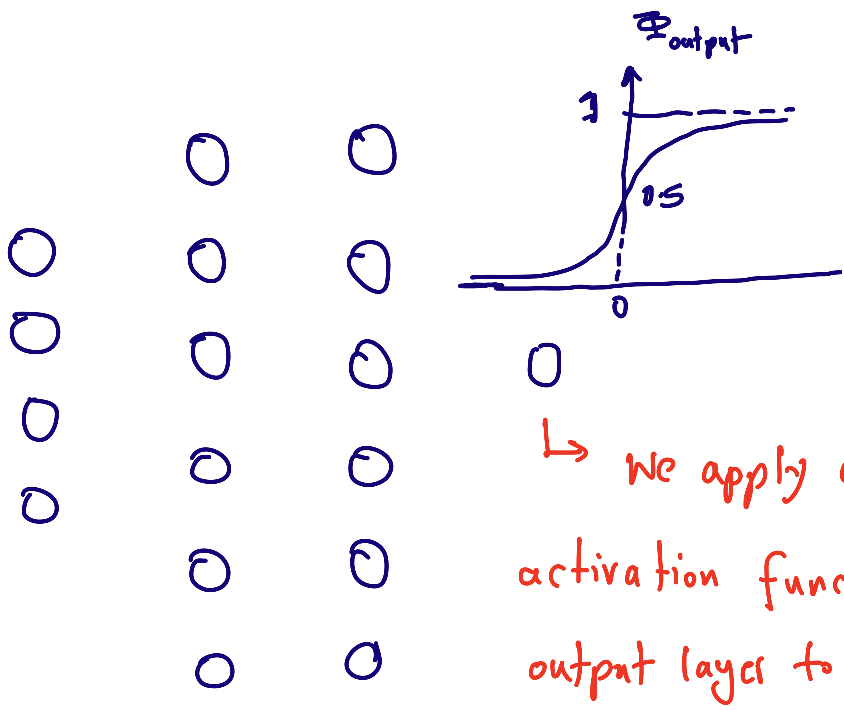
* People also use adaptive learning rates (η)
  methods: AdaGrad, RMSprop

** A widely used optimization strategy: <u>Adam</u>

It Combines momentum and adaptive learning rates in the optimization process.

Regarding classification problems

If it is a binary classification problem (0 or 1)



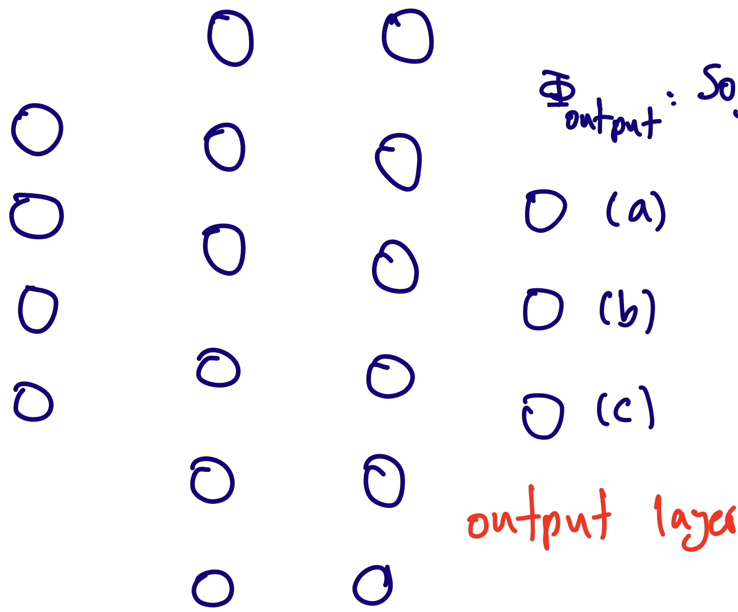
↳ We apply a Sigmoid activation function to the output layer to obtain a number between 0 and 1.

✱ : The optimization in this case is done via a different loss function. It is usually done by "cross-entropy" loss functions.

If the output layer had more than one node (e.g. when classifying diffrent types/classes of something), we use softmax activation function:

For example:

Φ_{output}: Softmax (defined below)



type 1: $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

type 2: $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$

type 3: $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

$$\Phi_{output}(x_a) = \frac{e^{x_a}}{e^{x_a} + e^{x_b} + e^{x_c}}$$

$$\Phi_{output}(x_b) = \frac{e^{x_b}}{e^{x_a} + e^{x_b} + e^{x_c}}$$

$$\Phi_{output}(x_c) = \frac{e^{x_c}}{e^{x_a} + e^{x_b} + e^{x_c}}$$

$$\boxed{\sum \Phi_{output} = 1}$$

★ The output layer can have any number of nodes and the procedure is similar.

★: In both binary and multiple tye cases, one can have an approximate probabilistic inference of the output

For instance: If output in our three-type example is $\begin{pmatrix} 0.85 \\ 0.1 \\ 0.05 \end{pmatrix} \rightarrow$ It can be approximately interpreted

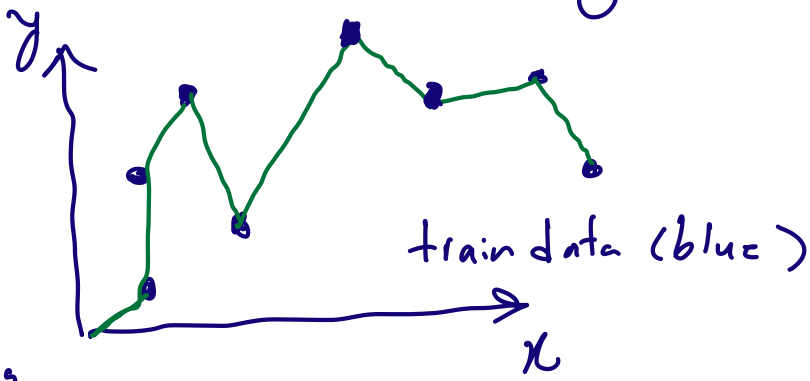
as: 85% type 1 $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

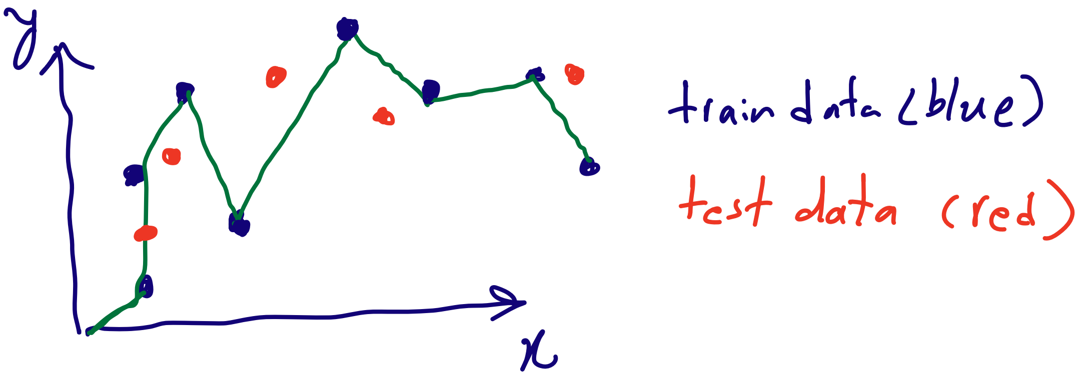
10% type 2 $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$

5% type 3 $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

How to avoid overfitting

1D example of overfitting





1) Split the data into train - evaluation - test sets (e.g. a 70% - 10% - 20% split) and check if after each epoch the loss values from training and evaluation sets are consistent. If, for example, the loss function is decreasing on the train data but increasing on the evaluation data, that can be a sign of over-fitting.

2) perform an N-fold cross-validation.

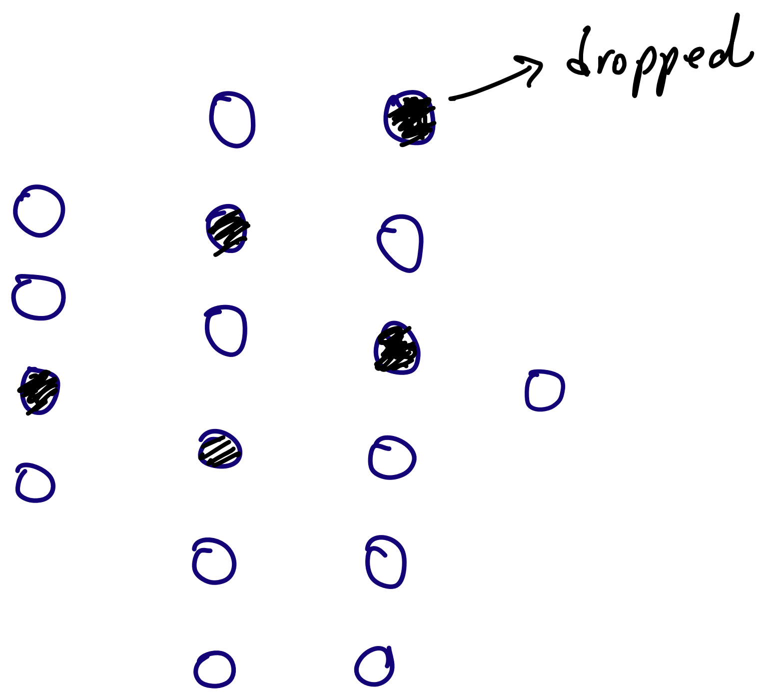
3) Use regularization tools for neural networks.

A famous example is dropout regularization.

It randomly turns off (drops) some neurons in the specified layer(s) each time it learns from the data.

This causes the model not to "memorize" the training data and not "rely" too much on any single neuron.

For example, in one instance



One point regarding input data:

In many situations, our input features need to be scaled, especially if their values differ by large amounts.

In our example:

$$\begin{array}{ccc}
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
\vdots & \vdots & \vdots \\
x_{n1} & x_{n2} & x_{n3}
\end{array}$$

One can use standard scaling, min-max scaling, etc.

In our example:

$$\begin{array}{ccc}
\hline
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
\vdots & \vdots & \vdots \\
x_{n1} & x_{n2} & x_{n3} \\
\hline
\end{array}$$

Standard scaling: Each column will have zero mean and unit standard deviation

$$x \to \frac{x - \mu}{\sigma}$$

Min-max scaling: Values in each column are between 0 and 1:

$$x \to \frac{x - x_{min}}{x_{max} - x_{min}}$$