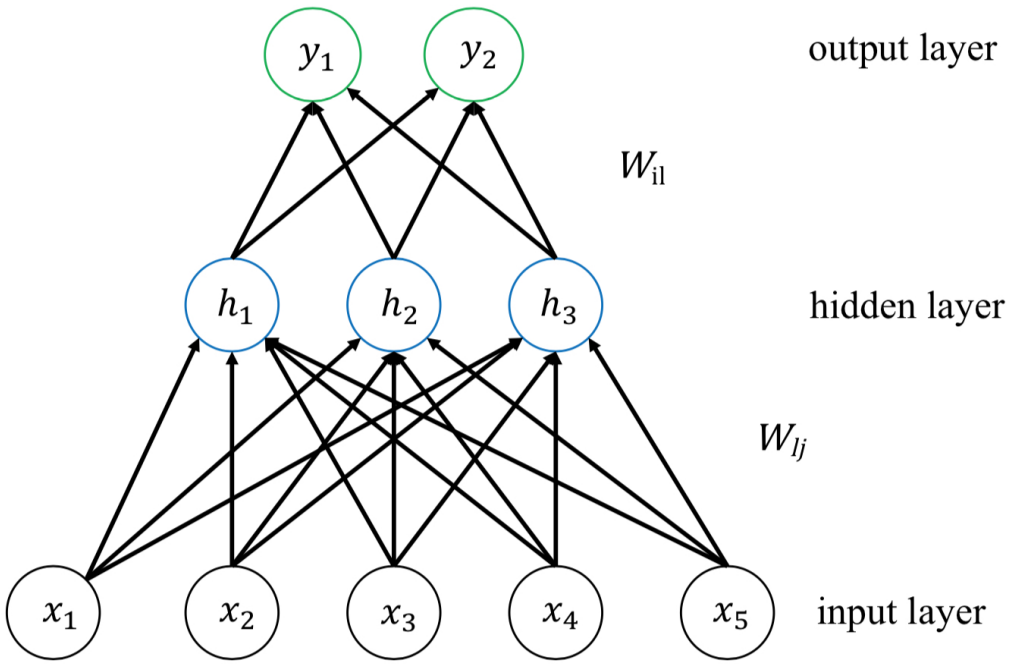
# Introduction to Artificial Neural Network Theory and Applications

ACLS ZHAW Life Sciences

Chapter 3: Multilayer Neural Networks

# Feedforward Neural Networks



## Feedforward Neural Networks

Neuron-like processing units use

$$y_i = \sigma(\sum_j w_{ij} \cdot x_j + b_i)$$

where the $x_j$ are the inputs, the $w_{ij}$ are the weights, $b$ is the bias, $y$ is the output and $\sigma$ is an activation function.

- FNN are built up by a combination of nodes with activation functions.

## Feedforward Neural Networks

Example of a fully connected network with one hidden layer:

- Input nodes: $x_j$
  Nodes in the hidden layer $l$: $h_l^{(\text{layer})}$
  Output nodes: $y_i$
- Fully connected network: each node receives connections from all the nodes in the previous layer.

The network computes

$$h_l^{(1)} = \sigma^{(1)}(\sum_j w_{lj}^{(1)} x_j + b_l^{(1)})$$

$$y_i = \sigma^{(2)}(\sum_l w_{il}^{(2)} h_l + b_i^{(2)}).$$

## Feedforward Neural Networks

Compact formulation

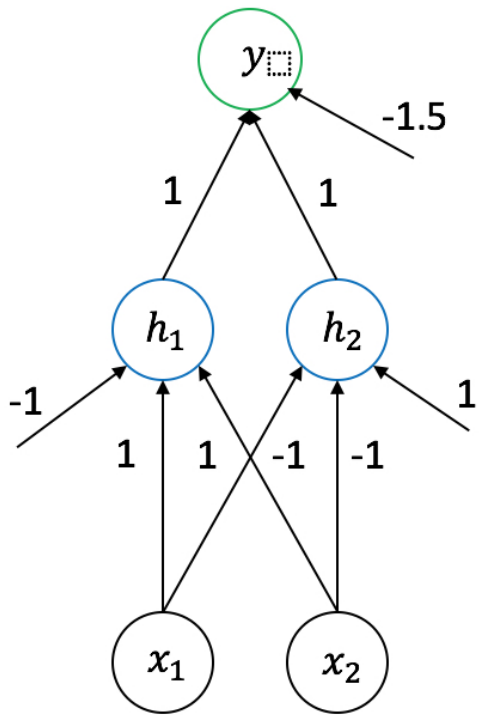
$$\mathbf{h}^{(1)} = \sigma^{(1)}(\mathbf{W}^{(1)} \cdot \mathbf{x})$$
$$\mathbf{y} = \sigma^{(2)}(\mathbf{W}^{(2)} \cdot \mathbf{h})$$

Combination of all the training examples into a single matrix $X$:
all predictions can be computed using a single matrix multiplication

$$\mathbf{H}^{(1)} = \sigma^{(1)}(\mathbf{W}^{(1)T}\mathbf{X})$$
$$\mathbf{Y} = \sigma^{(2)}(\mathbf{W}^{(2)T}\mathbf{H})$$

## Feedforward Neural Networks

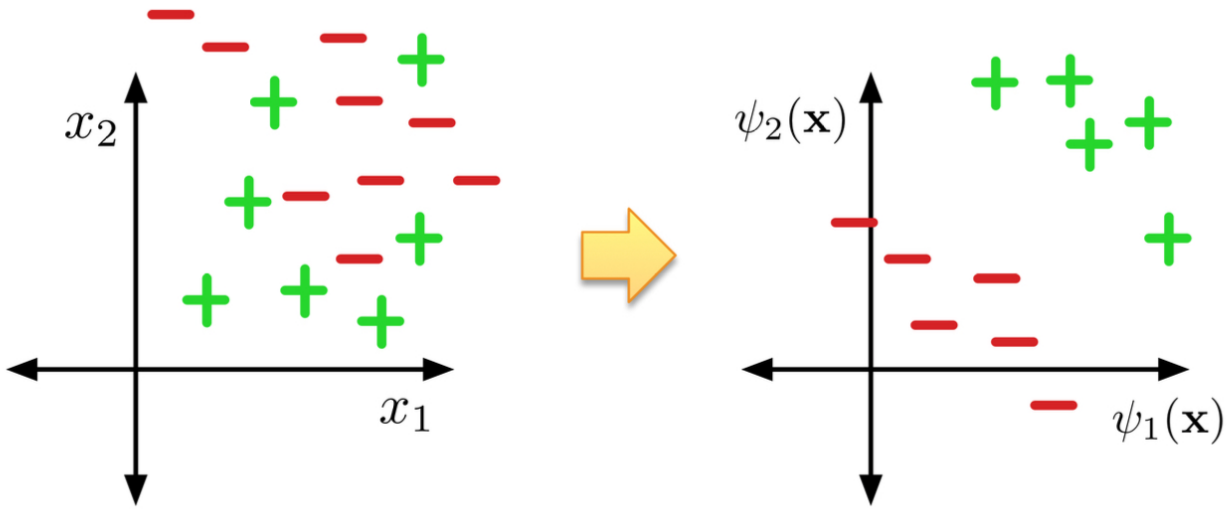
FNN or MLP are more powerful than the perceptron and its variants.
For example, the XOR function can be computed with a single hidden layer with two nodes:




**Task** Find the weights and biases for a MLP that implements the XOR function. Implement the MLP in Python.

## Feedforward Neural Networks

FNN can be thought of as a way of learning nonlinear feature mappings:
Hidden layer: feature map
Output layer: linear classifier using those features.




The hope is that it is possible to learn a feature representation where the data becomes linearly separable.

## Feedforward Neural Networks

FNN or MLP are much more powerful than the perceptron versions introduced in the previous chapter. But how powerful?

- If activation function is linear: networks with many layers are no more powerful than shallow ones.
  Reason: With linear activation function $\sigma(x) = x$, the network's function can be expanded as $y = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)}\cdot\cdot\mathbf{W}^{(1)}\mathbf{x}$ which can be treated as a single linear layer with weights given by $\mathbf{W} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)}\cdot\cdot\mathbf{W}^{(1)}$.
- With nonlinear activation functions: FNN with just one hidden layer can represent any function.
  This is called the *universal approximation theorem*.

## Feedforward Neural Networks

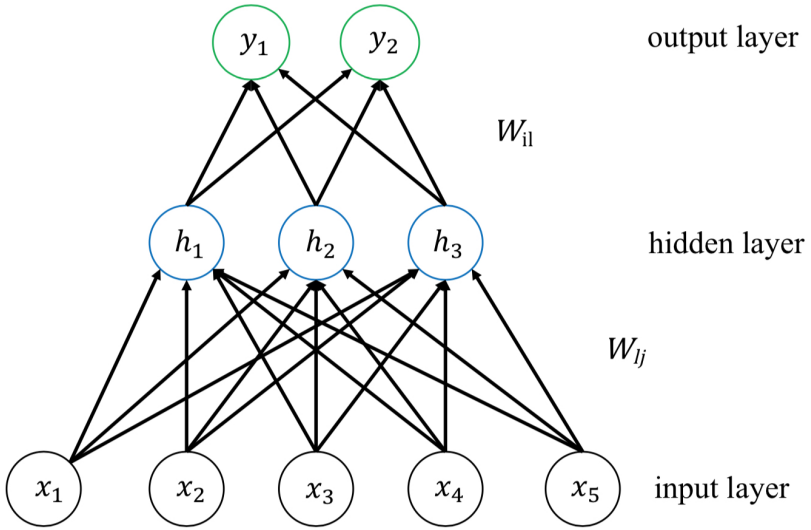
With universal approximation theorem: why using deep FNN with many hidden layers?

1. deep networks allow for a more compact representation whereas the FNN with one hidden layer may need an exponentially large layer,
2. compactness is computationally more efficient,
3. although any function can be represented by an FNN with one hidden layer this does not mean that the FNN can learn well the function and that it generalises well to unseen examples.

FNN with few hidden layers are called *shallow* neural networks, the FNN with only one hidden layer are also called *vanilla* networks

# Backpropagation

- How can we train FNNs? This is achieved with the famous *backpropagation algorithm*.
- We explain the algorithm by the example of a FNN with one hidden layer, but the method generalises to FNN with any number of hidden layers.



## Backpropagation

Let us assume the cost function

$$E(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{M}(t_i - y_i)^2.$$

With equation $y_i = \sigma(\sum_l w_{il}^{(2)} h_l)$ one gets

$$E(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{M}(t_i - \sigma(\sum_l w_{il}^{(2)} h_l))^2.$$

and with equation $h_l^{(1)} = \sigma^{(1)}(\sum_j w_{lj}^{(1)} x_j + b_l^{(1)})$ one has

$$E(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{M}(t_i - \sigma^{(2)}(\sum_l w_{il}^{(2)} \sigma^{(1)}(\sum_j w_{lj}^{(1)} x_j + b_l^{(1)})))^2, \tag{1}$$

We then use the gradient descent algorithm. That is, we change the weights proportional to the gradient of $E$. But now we have two sets of weights, $w_{lj}^{(1)}$ and $w_{il}^{(2)}$ and the changes are $\Delta w_{lj}^{(1)} = -\eta \frac{\partial E}{\partial w_{lj}^{(1)}}$ and $\Delta w_{il}^{(2)} = -\eta \frac{\partial E}{\partial w_{il}^{(2)}}$.

## Backpropagation

Let us thus compute $\frac{\partial E}{\partial w_{il}^{(2)}}$:

$$\frac{\partial E}{\partial w_{il}^{(2)}} = \frac{\partial}{\partial w_{il}^{(2)}} \frac{1}{2} \sum_i (t_i - y_i)^2 = \frac{1}{2} \frac{\partial}{\partial w_{il}^{(2)}} (t_i - y_i)^2 \tag{2}$$

$$= (t_i - y_i) \frac{\partial}{\partial w_{il}^{(2)}} (t_i - y_i) \tag{3}$$

$$= (t_i - y_i) \frac{\partial}{\partial w_{il}^{(2)}} (t_i - \sigma(\sum_l w_{il}^{(2)} h_l)) \tag{4}$$

$$= -(t_i - y_i) \sigma'(\sum_l w_{il}^{(2)} h_l) h_l. \tag{5}$$

We thus have for $\Delta w_{il}^{(2)} = -\eta \frac{\partial E}{\partial w_{il}^{(2)}}$ with $\delta_i = (t_i - y_i)\sigma'(\sum_l w_{il}^{(2)} h_l)$

$$\Delta w_{il}^{(2)} = \eta \delta_i h_l. \tag{6}$$

## Backpropagation

For $\frac{\partial E}{\partial w_{lj}^{(1)}}$ we get:

$$\frac{\partial E}{\partial w_{lj}^{(1)}} = \frac{\partial E}{\partial h_l}\frac{\partial h_l}{\partial w_{lj}^{(1)}} \tag{7}$$

$$= -\sum_i (t_i - y_i)\sigma'(\sum_l w_{il}^{(2)} h_l) w_{il}^{(2)} \sigma'(\sum_j w_{lj}^{(1)} x_j) x_j \tag{8}$$

$$= -\sum_i \delta_i w_{il}^{(2)} \sigma'(\sum_j w_{lj}^{(1)} x_j) x_j \tag{9}$$

$$\tag{10}$$

We thus have for $\Delta w_{lj}^{(1)} = -\eta \frac{\partial E}{\partial w_{lj}^{(1)}}$ with $\delta_l = \sigma'(\sum_j w_{lj}^{(1)} x_j) \sum_i \delta_i w_{il}^{(2)}$

$$\Delta w_{lj}^{(1)} = \eta \delta_l x_j . \tag{11}$$

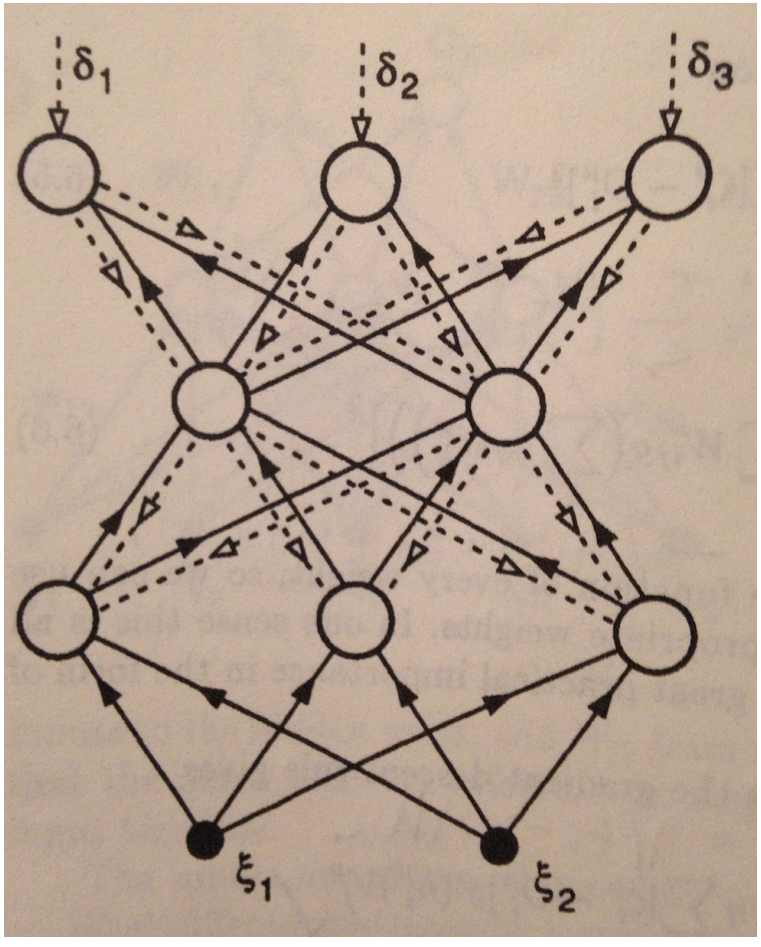
## Backpropagation

- Comparing expressions (6) and (11) one sees that one gets the same expression save with different $\delta$'s.
- In general, with an arbitrary number of layers, the backpropagation algorithm yields

$$\Delta w_{pq} = \eta \delta_{out} v_{in},$$

  where *out* and *in* refer to the two ends $p$ and $q$ of the connection concerned and $v$ stands for the appropriate input-output activation from a hidden node or an input node.
- A FNN propagates a signal, the input, forward to generate an output and thus an error. By equation (6) and (11) the errors are propagated backwards and used to change the weights; hence the name "backpropagation".
- The algorithm works for any differentiable activation function and thus differentiable cost function.

# Backpropagation



## Backpropagation

For the sigmoid activation function, $\sigma(z) = \frac{1}{1+e^{-z}}$, for example, one gets

$$\delta_i = y_i(1 - y_i)(t_i - y_i). \tag{12}$$

and

$$\delta_l = h_l(1 - h_l)(\sum_i \delta_i w_{il}^{(2)}). \tag{13}$$

With these functions the backpropagation algorithm of an FNN with one hidden layer and sigmoid activation functions can be implemented.

## Backpropagation

**Task** Implement a sigmoid MLP with one hidden layer that learns the XOR function.

## Activation Functions

So far, we encountered the following activation functions

- the threshold activation function $y = \sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$, used for the classical perceptron. It can be used for binary classification task. The activation function is not differentiable and hence cannot be used for gradient descent learning.
- the linear activation function $y = \sigma(z) = z$ which can be used for regression problems, when used in the output layer.
- the sigmoid activation function $y = \sigma(z) = \frac{1}{1+e^{-z}}$. It outputs a value between 0 and 1 and can thus indicate in the output layer a probability score for binary classification tasks. It can also be used for multiclass classification but its values do not sum to 1.

## Activation Functions

Further activation functions are

- For multiclass classification the so-called softmax function, i.e., $y = \sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$, is usually used in the output layer as it outputs values between 0 and 1 which sum to 1. Hence its output can be interpreted as a probability distribution.
- The tanh function, that is, $y = \sigma(z) = \tanh z$ which is also a sigmoid function that outputs values in the range $-1$ to 1 is often used as activation function for the hidden nodes.
- Mostly however the so-called ReLU (rectified linear unit) activation, i.e., $y = \sigma(z) = \begin{cases} 0 & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$ is used as activation function for the hidden layers as it has shown better and faster training results than other activation functions.

## Cost Functions

Besides the activation functions also a loss or error function needs to be chosen.

- For binary and multiclass classification problems where the output of your network is a probability or probability distribution the *cross entropy* loss function is usually used. Cross entropy is a quantity from information theory that measures the distance between probability distributions or, in the context of ANN, between the ground-truth distribution and the predictions, that is, $E = -\frac{1}{N}\sum_i y_i \log(\hat{y}_i)$, where $\mathbf{y}$ is the ground-truth or target and $\hat{\mathbf{y}}$ are the outputs or predictions of the network.
- For regression tasks the mean squared error (MSE) loss function, i.e., $E = \frac{1}{N}\sum_i (t_i - y_i)^2$ or the root mean squared error (RMSE) loss function, i.e., $E = \sqrt{\frac{1}{N}\sum_i (t_i - y_i)^2}$, or the mean absolute error (MAE) loss function, i.e., $E = \frac{1}{N}\sum_i |t_i - y_i|$, is used.

Note that besides these standard tasks the cost functions are usually custom as they need to be tailored to the particular task or problem.

## Optimizers

- Standard SGD or mini-batch or batch gradient descent. In SGD, the weight update is

$$w = w - \eta \nabla E_w(\mathbf{x}) \tag{14}$$

  with $\eta$ the learning rate.
- SGD with momentum: there is an addition to the classic SGD algorithm,

$$w = w - \eta \nabla E_w(\mathbf{x}) + \alpha \triangle w \tag{15}$$

  with an additional parameter $\alpha$ usually set around 0.9.
  Often SGD with momentum works better and faster than SGD. It may be thought of as using a moving average of gradients accelerating SGD in the relevant direction.

## Optimizers

- RMSprop: a variant of SGD with adaptive learning rate

$$w = w - \frac{\eta}{\sqrt{v(w,t)}} \nabla E_w(\mathbf{x}) \tag{16}$$

  with $v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_w(\mathbf{x}))^2$, $\gamma$ an additional parameter usually set around 0.9.

- Adaptive Moment Estimation (Adam): another method that computes adaptive learning rates.
  It can be viewed as a combination of RMSprop and SGD with momentum.

## Regularization

- Central problem in neural network learning: make an algorithm perform well not just on the training data, but also on new, unseen data input.
- *Regularization*: strategies to reduce the test error.
- Practical success of an algorithm often hinges on the regularization techniques.
- Developing more effective regularization: major research effort in the field.

## Underfitting & Overfitting

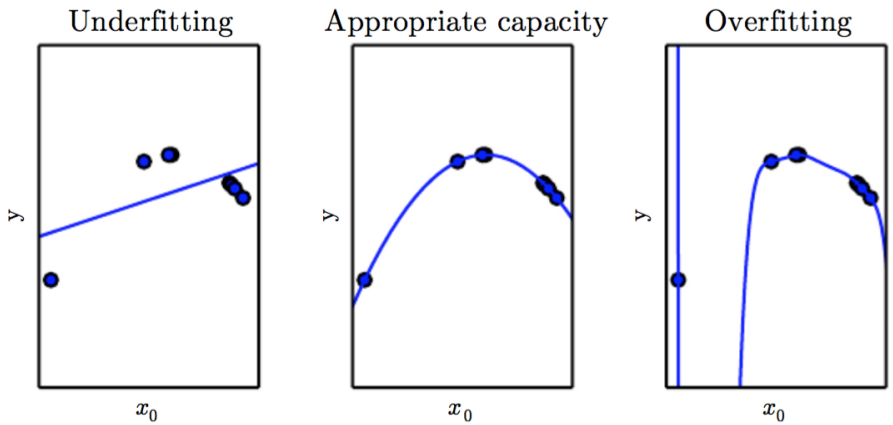
- Performance of algorithm:
  1. Make the training error small.
  2. Make the gap between training and test error small.

  These two factors correspond to two central challenges in neural network learning: *underfitting* and *overfitting*.
- Underfitting: model is not able to obtain a sufficiently low error value on the training set.
- Overfitting: gap between the training error and test error is too large.

# Model Capacity

One way to control model is more likely to overfit or underfit: by its *capacity*.

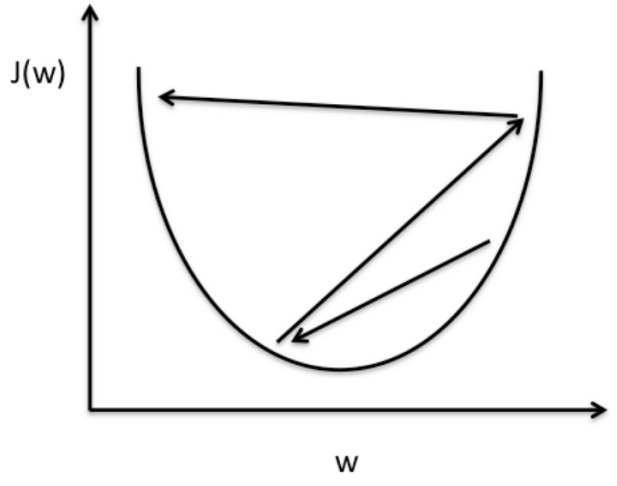
Model's capacity: ability to fit a wide variety of functions.
Models with low capacity: struggle to fit the training set.
Models with high capacity: can overfit by adapting to properties of the training set too much.



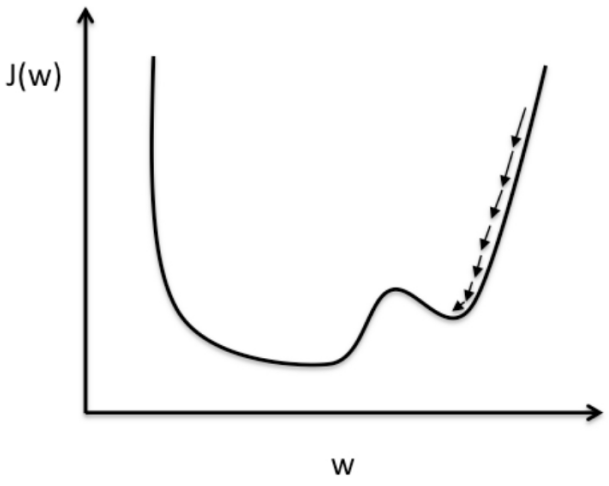

A linear, quadratic and degree 9 function is compared attempting to fit a problem where the true underlying function is quadratic.

## Regularization Techniques

1. **Model complexity**
2. **Learning rate**
   Most models, such as the linear variant of the perceptron, have a learning rate, i.e. a hyperparameter that controls how much we are adjusting the weights in a learning step.
   In practice, it often requires some experimentation to find a good learning rate. A good start is by plotting the cost function for different learning rates.




Large learning rate: Overshooting.




Small learning rate: Many iterations until convergence and trapping in local minima.

3. **Weight regularisation** With ANN, often multiple sets of weight values (multiple models) for some training data and a network architecture, can explain the data. However, simpler models are less likely to overfit than complex ones.
Fight overfitting: limit the complexity of a network by forcing its weights to take only small values.
This is called weight regularisation and is achieved by adding costs associated with large weights to the loss function of the network

$$E_{\mathbf{W}}(\mathbf{x}) = E_{\mathbf{W}}(\mathbf{x}) + \alpha g(\mathbf{W}). \tag{17}$$

Two main variations:
$L^1$ regularisation - The cost added is proportional to the absolute value of the weight coefficients, that is, $g(\mathbf{W}) = ||w||_1 = \sum_i |w_i|$.
$L^2$ regularisation - The cost added is proportional to the square of the value of the weight coefficients, that is, $g(\mathbf{W}) = ||w||_2 = \sum_i w_i^2$.
$L^2$ regularisation is sometimes also called weight decay in the context of ANN.

4. **Early stopping**
   Often one can observe that training error decreases steadily over time, but validation set error begins to rise again.
   Better validation set error: parameter setting at the point in time with the lowest validation set error.
   One of the most commonly used form of regularisation in neural networks learning as it can both be effectiv and simple to implement.

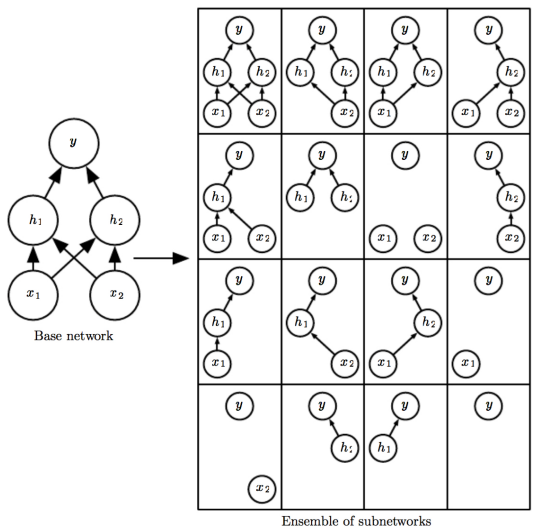
5. **Feature scaling**
   Feature scaling: input is rescaled.
   Often it is not only easier to find an appropriate learning rate if the features are on the same scale, but it also often leads to faster convergence and can prevent the weights from becoming too small (numerical stability).
   A common way of feature scaling is standardisation

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i} \tag{18}$$

   where $\mu_i$ is the sample mean of the feature $x_i$ and $\sigma_i$ is the standard deviation, respectively. After standardisation, the features will have unit variance and are centered around mean zero.

## Regularization Techniques

5. **Dropout** Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. Specifically, dropout trains an ensemble consisting of all subnetworks that can be formed by removing non-output nodes from an underlying base network.

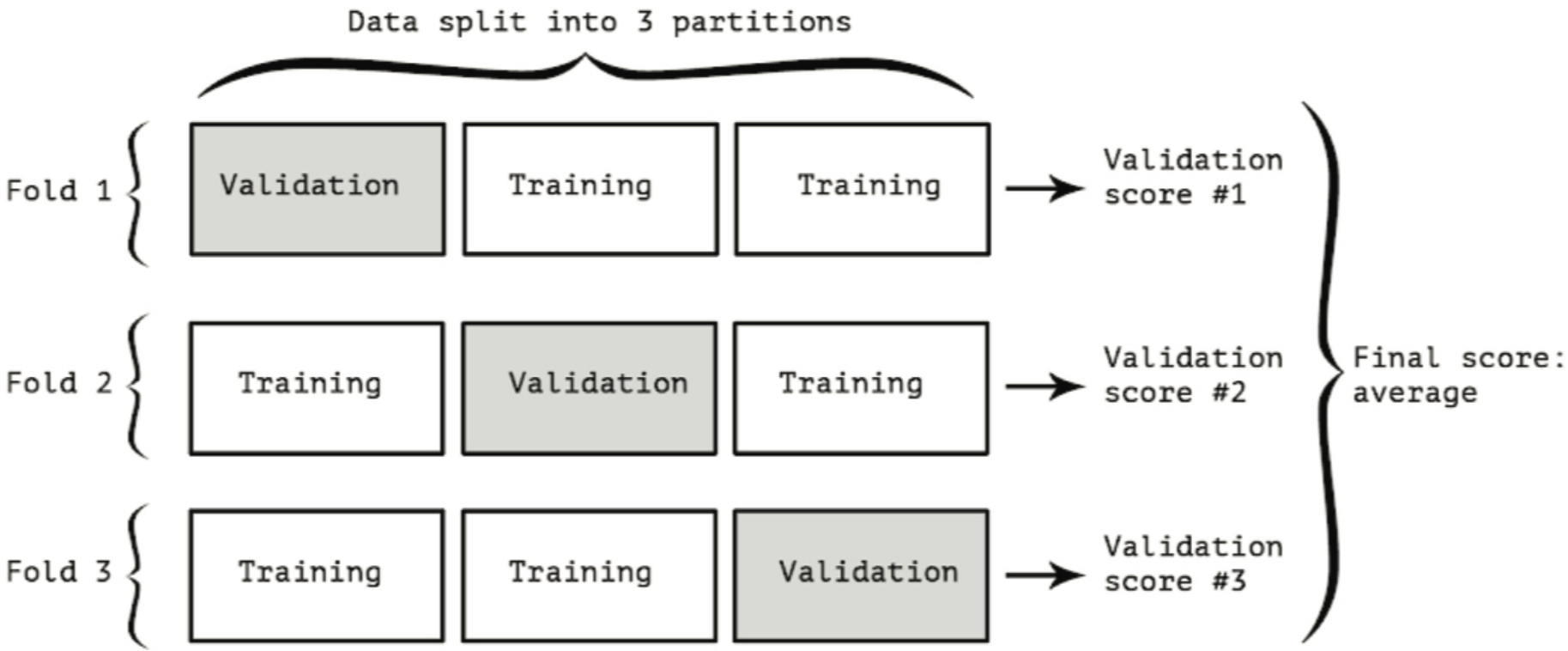



One can think of dropout as a way to achieve that each hidden node performs well regardless of which other hidden nodes are in the model. Dropout thus regularizes each hidden node to be not merely a good feature but a feature that is good in many contexts.

## Regularization Techniques

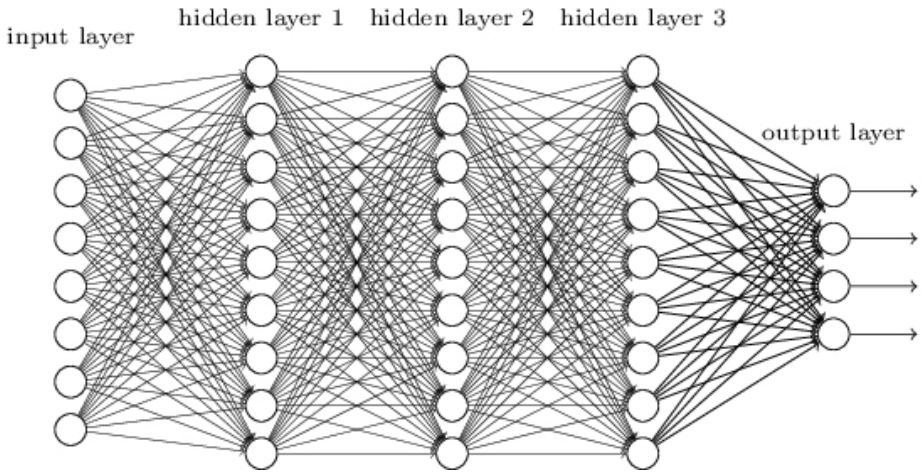
6. **Stochastic gradient descent**
   Stochastic gradient descent can also have an impact on the learning algorithm's performance.
7. **Data set augmentation**
   If possible, one of the best way to make a neural network model generalise better is to train it on more data.

# K-Fold Validation



# What is Deep Learning?

- Deep learning network (DNN): artificial neural network (ANN) with many hidden layers.

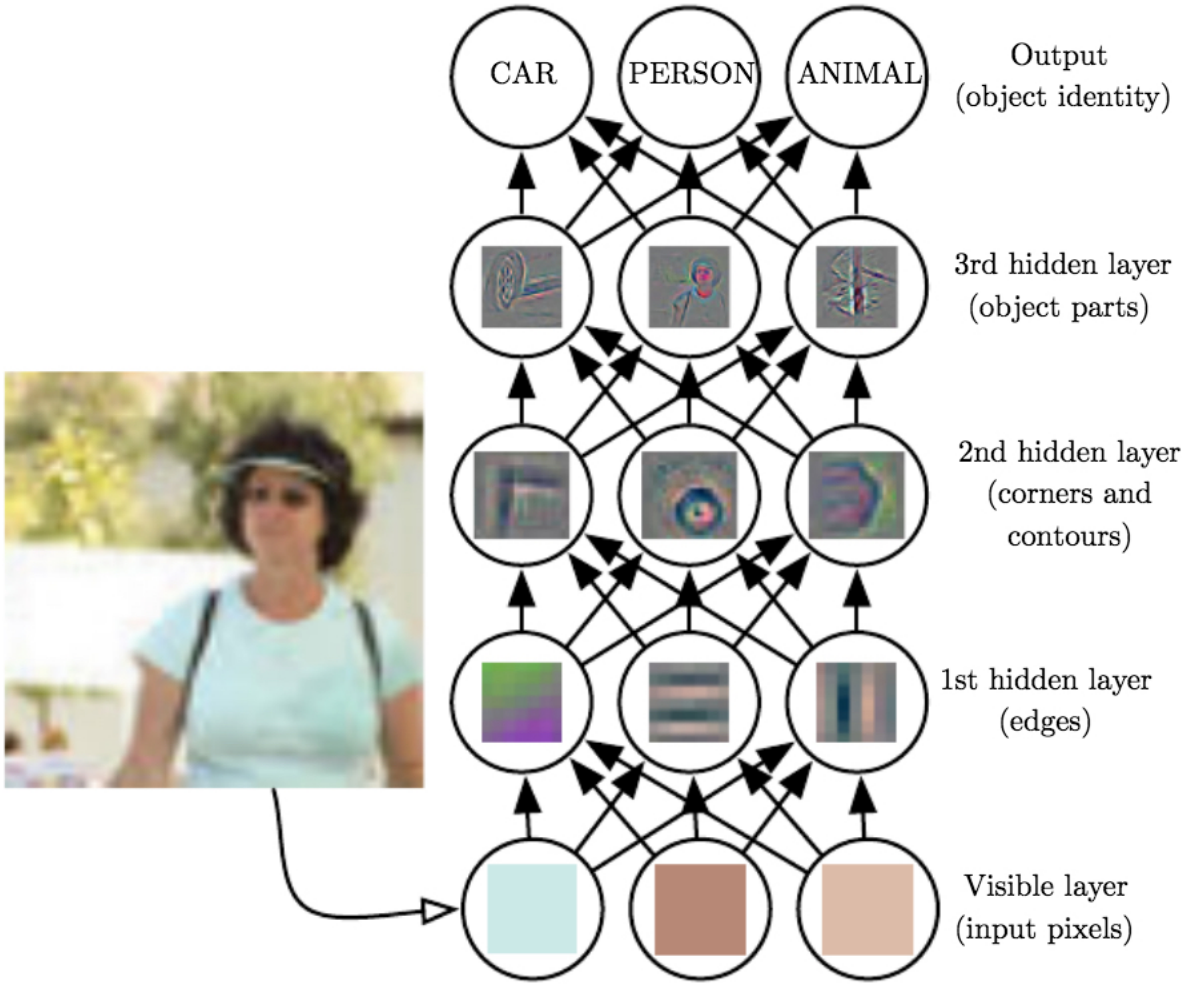



- The depth of DNN enables the network to learn more complicated, multistep programs.
  Several layers in a network offer greater power because later processes can refer back to the earlier processes in the network where the layers do not necessarily encode the input variations directly.

# What is Deep Learning?

- Another view of deep learning: learning *representations*.
- Conventional machine-learning techniques:
  requires careful engineering and expertise to design a feature extractor that transforms the raw data (e.g. the pixel values of an image) into a suitable internal representation or feature vector from which e.g. a classifier could detect or classify patterns in the input.
- **Deep learning: network is fed with raw data and discovers automatically the representations or features needed for a task.**
- Deep learning allows for multiple levels of representation:
  by composing simple modules that each transform the representation at one level, starting with the raw input, into a representation at a higher, slightly more abstract level.

# What is Deep Learning?



# What is Deep Learning?

- The basic techniques behind deep learning have been around for many years.
- New frameworks, data, computing power and specialised techniques and network architectures: breakthroughs with many problems.
- Deep convolutional networks/specific recurrent networks: breakthroughs in processing images, video, speech, audio and text.
- Very active research field, both in academics as well as industry.

# Toolboxing

# Toolboxing

We will use Tensorflow 2.

## Tensorflow 2 Exercises

**Tasks**

1. Install Tensorflow 2 and get it running.
2. Go through the MNIST classification problem.
3. Do the iris classification problem with Tensorflow 2.

# Tensorflow 2 Exercises

**Task**

1. Predicting house prices: Predict the median price of homes in a given Boston suburb in the 1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.
   Boston housing dataset: provided by Tensorflow 2
2. Experiment with:
   Number of hidden layers
   Number of hidden units
   Try different loss functions.
   Try different activation functions.
   Try different regularization techniques.
   Preprocess input data.
3. Competition: the winner takes it all!