Current Students

Copyright and UWA unit content

Is it OK to download and share course material such as lectures, unit outlines, exam papers, articles and ebooks?

UWA is committed to providing easy access to learning material and many of your lectures are available for online access via the Lecture Capture System (LCS), accessible through the LMS. Your unit coordinator may make their lecture recordings available to download if they wish. You are allowed to access recorded lectures in the format they are supplied on the LCS – so if they are not made available to download, you must not use any software or devices to attempt to download them.

All recorded lectures and other course material, such as presentation slides, lecture and tutorial handouts, unit outlines and exam papers, are protected under the Copyright Act and remain the property of the University. You are not allowed to share these materials outside of the LMS – for example, by uploading them to study resource file sharing websites or emailing them to friends at other universities. Distributing course material outside of the LMS is a breach of the University Policy on Academic Conduct and students found to be sharing material on these sites will be penalised. University data, emails and software are also protected by copyright and should not be accessed, copied or destroyed without the permission of the copyright owner.

Other material accessible from the LMS or via the Library, such as ebooks and journal articles, are made available to you under licensing agreements that allow you to access them for personal educational use, but not to share with others.

Can I share my login details?

No! Pheme is your key to accessing a number of UWA's online services, including LMS, studentConnect, UWA email, your Library account, and Unifi. These services hold copyright material as well as your personal information, including your unit marks, enrolment information and contact details, so it is important that you do not share the access credentials with anyone else.

https://www.student.uwa.edu.au/learning/resources/ace/respect-intellectual-property/copyright-and-uwa-unit-content

CITS5508 Machine Learning

Débora Corrêa (Unit Coordinator and Lecturer)
UWA

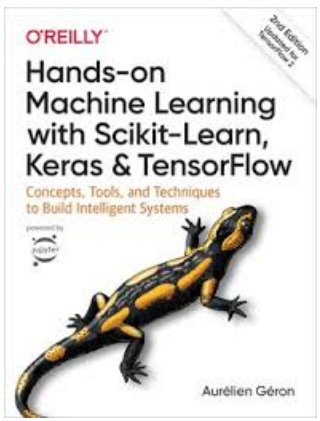
2024

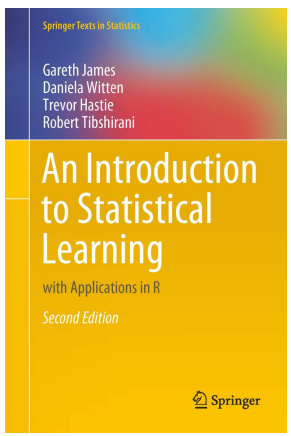
Today

Introduction to Neural Networks

Hands-on Machine Learning with Scikit-Learn & TensorFlow

An Introduction to Statistical Learning





Introduction to Artificial Neural Networks

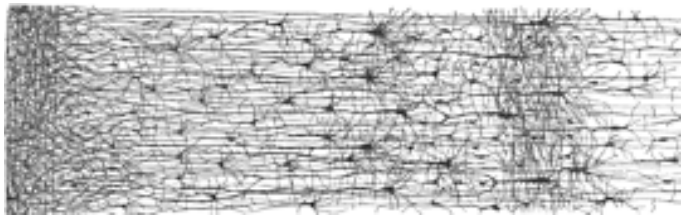
Here are the main topics we will cover:

- Perceptrons and *threshold logic unit* (TLU)
- Training of a Perceptron and *Hebb's rule*
- Perceptrons and the XOR classification problem
- Multi-Layer Perceptrons (MLP) and backpropagation
- Commonly used activation functions
- Fine-tuning neural network hyperparameters
- Convolutional Neural Networks
- Recurrent Neural Networks

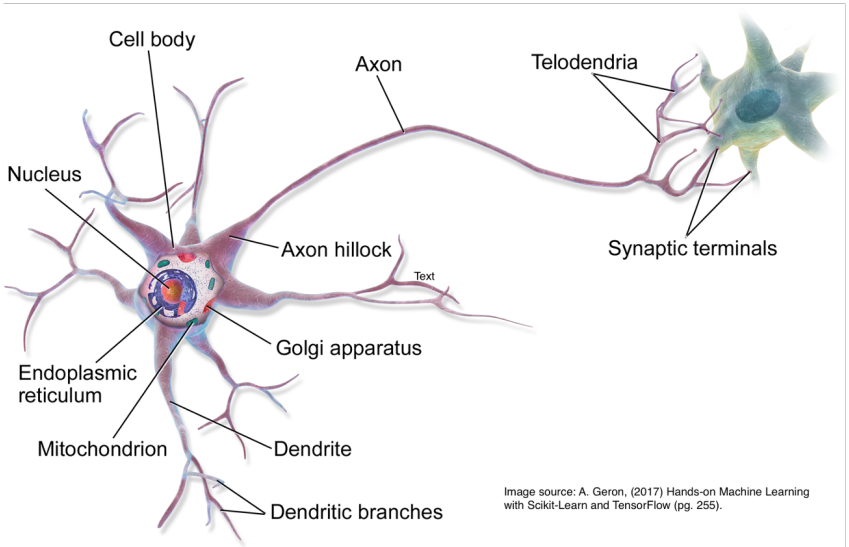
ANNs: Neuro-Inspired Computation

Artificial neural networks (ANNs): a family of models with a gross inspiration from the brain.

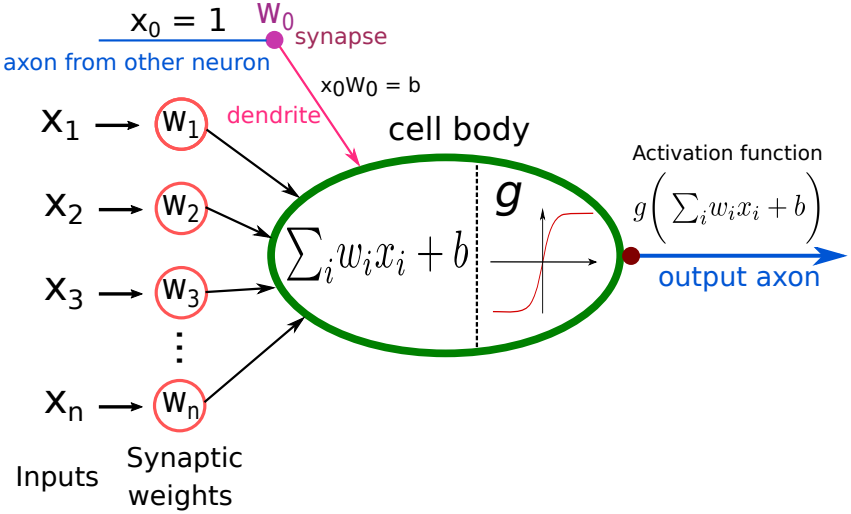
Good on unstructured data.



Biological Neuron



Artificial Neuron

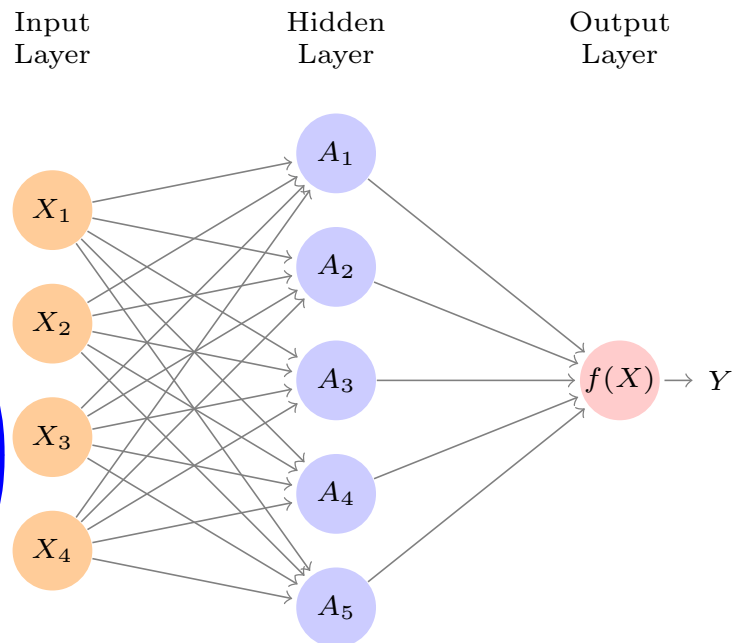


Overall Structure of a Neural Network

Input: Vector of n variables $X = (X_1, X_2, ..., X_n)$.
Output: Predicted response Y from a nonlinear function $f(X)$.

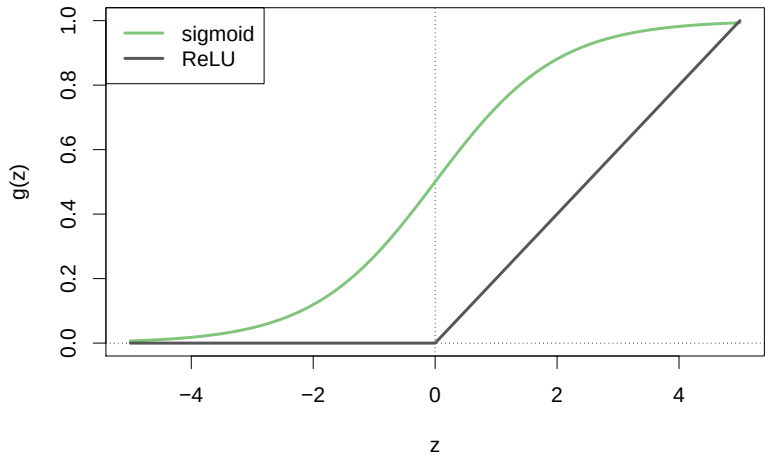
$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X)$$

$$= \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{n} w_{kj} X_j\right)$$



$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{n} w_{kj} X_j)$: activations in the `hidden layer`.
$g(z)$: activation function. Popular choices: `sigmoid`, `ReLU`.

Overall Structure of a Neural Network



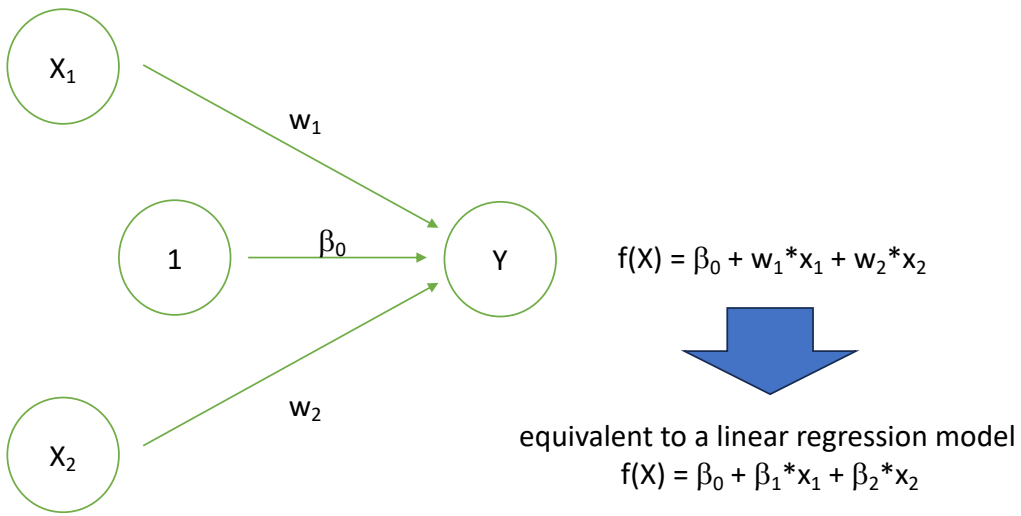
$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{n} w_{kj} X_j)$: activations in the `hidden layer`.
$g(z)$: activation function. Popular choices: `sigmoid`, `ReLU`.

Activation functions in the hidden layers are like derived features: nonlinear transformations of linear combinations of the features.

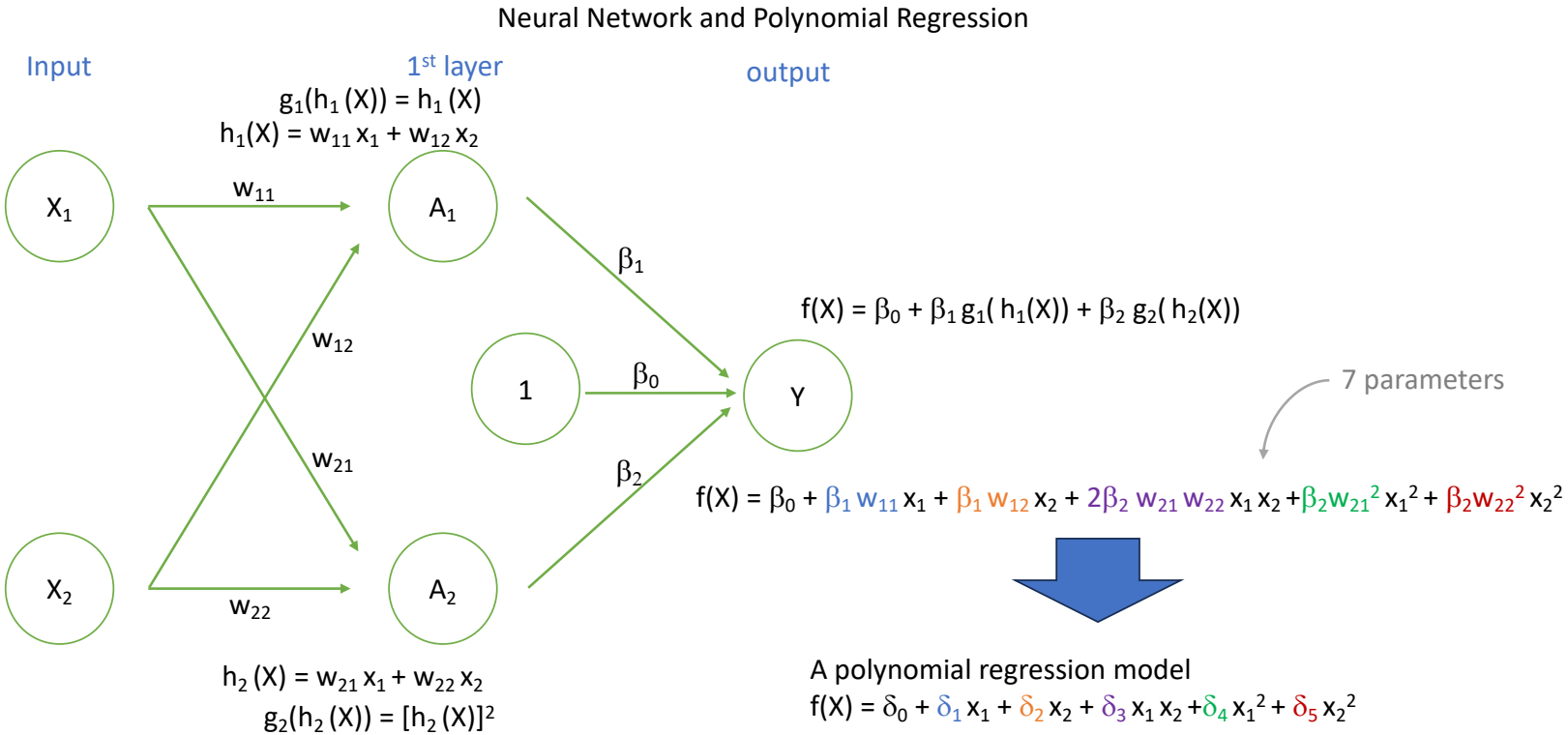
Fit the model: minimise $\sum_{i=1}^{m} (y_i - f(x_i))^2$ (e.g. for regression).

Neural Networks and Linear Regression

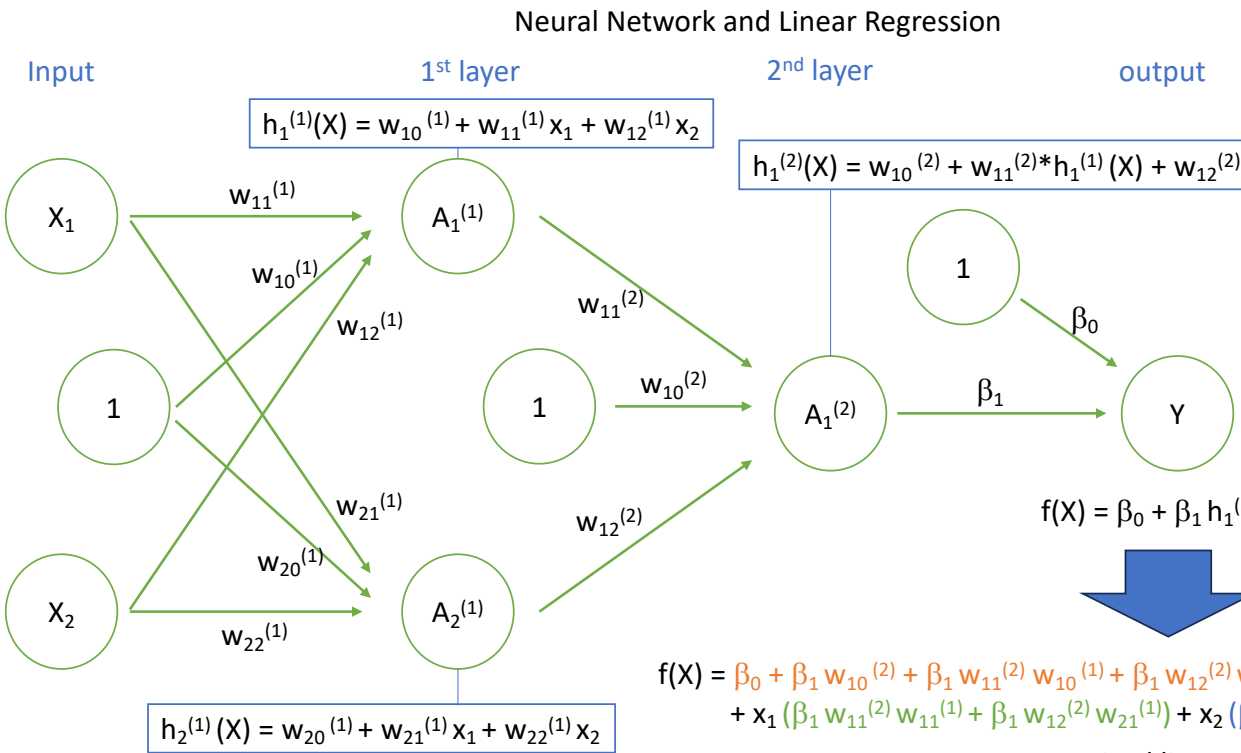
Neural Network and Linear Regression



Neural Networks and Linear Regression



Neural Networks and Linear Regression



$f(X) = \beta_0 + \beta_1 h_1^{(2)}(X)$

11 parameters

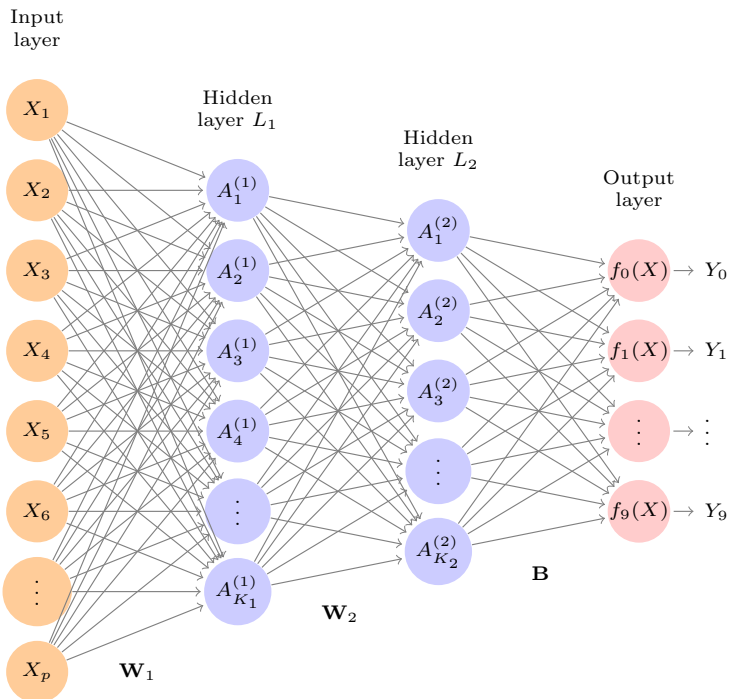
$$f(X) = \beta_0 + \beta_1 w_{10}^{(2)} + \beta_1 w_{11}^{(2)} w_{10}^{(1)} + \beta_1 w_{12}^{(2)} w_{20}^{(1)} + x_1 (\beta_1 w_{11}^{(2)} w_{11}^{(1)} + \beta_1 w_{12}^{(2)} w_{21}^{(1)}) + x_2 (\beta_1 w_{11}^{(2)} w_{12}^{(1)} + \beta_1 w_{12}^{(2)} w_{22}^{(1)})$$

A superparameterised linear regression model

$f(X) = \delta_0 + x_1 \delta_1 + x_2 \delta_1$

Considering
$g(h_i^{(j)}(X)) = h_i^{(j)}(X)$

Example - NMIST dataset



Handwritten digits as 28 x 28 grayscale images.

Features are the 784-pixel grayscale values $\in$ (0,255).

Labels are the digit class 0-9.

Goal: Fit a classifier to predict the image class.

Two-layer network: the first hidden layer has 256 units, the second hidden layer has 128 units, and the output layer has 10 units.

Adding the intercepts, there are **235,146** parameters.

Example - NMIST dataset

The first hidden layer has activations

$$A_k^{(1)} = h_k^{(1)}(X)$$
$$= g\left(w_{k0}^{(1)} + \sum_{j=1}^{n} w_{kj}^{(1)} X_j\right), \text{for } k = 1, ..., K_1.$$

The second hidden layer receives as input the activations $A_k^{(1)}$ of the first hidden layer and computes new activations:

$$A_l^{(2)} = h_l^{(2)}(X)$$
$$= g\left(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lk}^{(2)} A_k^{(1)}\right), \text{for } l = 1, ..., K_2.$$

$\mathbf{W_1}$ represents the entire matrix of weights from the input layer to the first hidden layer. It has $785 \times 256 = 200,960$ elements.

$\mathbf{W_2}$ is the weight matrix from the first hidden layer that feeds the second hidden layer. It has $257 \times 128 = 32,896$ elements.

Example - NMIST dataset

The output layer will contain 10 linear combinations of activations from the second layer:

$$Z_o = \beta_{o0} + \sum_{l=1}^{K_2} \beta_{ol} h_l^{(2)}(X)$$

$$= \beta_{o0} + \sum_{l=1}^{K_2} \beta_{ol} A_l^{(2)}, \text{for } o = 0, 1, ..., 9.$$

B contains $129 \times 10 = 1,290$ elements/weights.

Output activation function encodes the softmax activation function:

$$f_o(X) = \Pr(Y = o|X) = \frac{e^{Z_o}}{\sum_{l=0}^{9} e^{Z_l}}, \text{for } o = 0, 1, ..., 9.$$

Fit the model by minimising the negative multinomial log-likelihood (or cross-entropy):

$$-\sum_{i=1}^{m} \sum_{o=0}^{9} y_{io} log(f_o(x_i)).$$

The Perceptron

The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.

It is based on a slightly different artificial neuron called a *threshold logic unit* (TLU), where each input connection is associated with a weight.

The TLU computes a weighted sum of its inputs:
$z = w_1 x_1 + \cdots + w_n x_n = \mathbf{x}^T \mathbf{w}$
then applies a *step function* to that sum and outputs the result:
$h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{x}^T \mathbf{w})$.

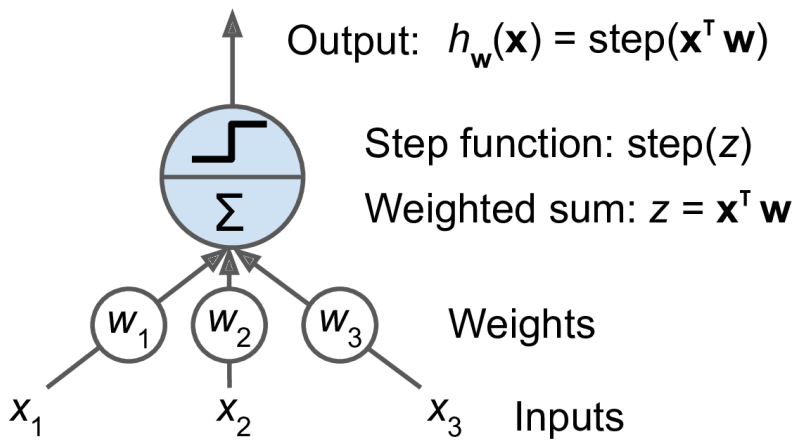


Figure 10-4.
Threshold logic unit

The Perception (cont.)

Step Functions

The most common step function used in Perceptrons is the *Heaviside step function*:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Sometimes the *sign function* is used instead:

$$\text{sign}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron (cont.)

A single TLU can be used for simple linear binary classification: It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).

A Perceptron is simply composed of a single layer of TLUs, with

- special passthrough neurons called *input neurons*, which just output whatever input they are fed; and
- an extra bias feature ($x_0 = 1$), which is typically represented using a special type of neuron called a *bias neuron* (it outputs 1 all the time).

Perceptron: An example

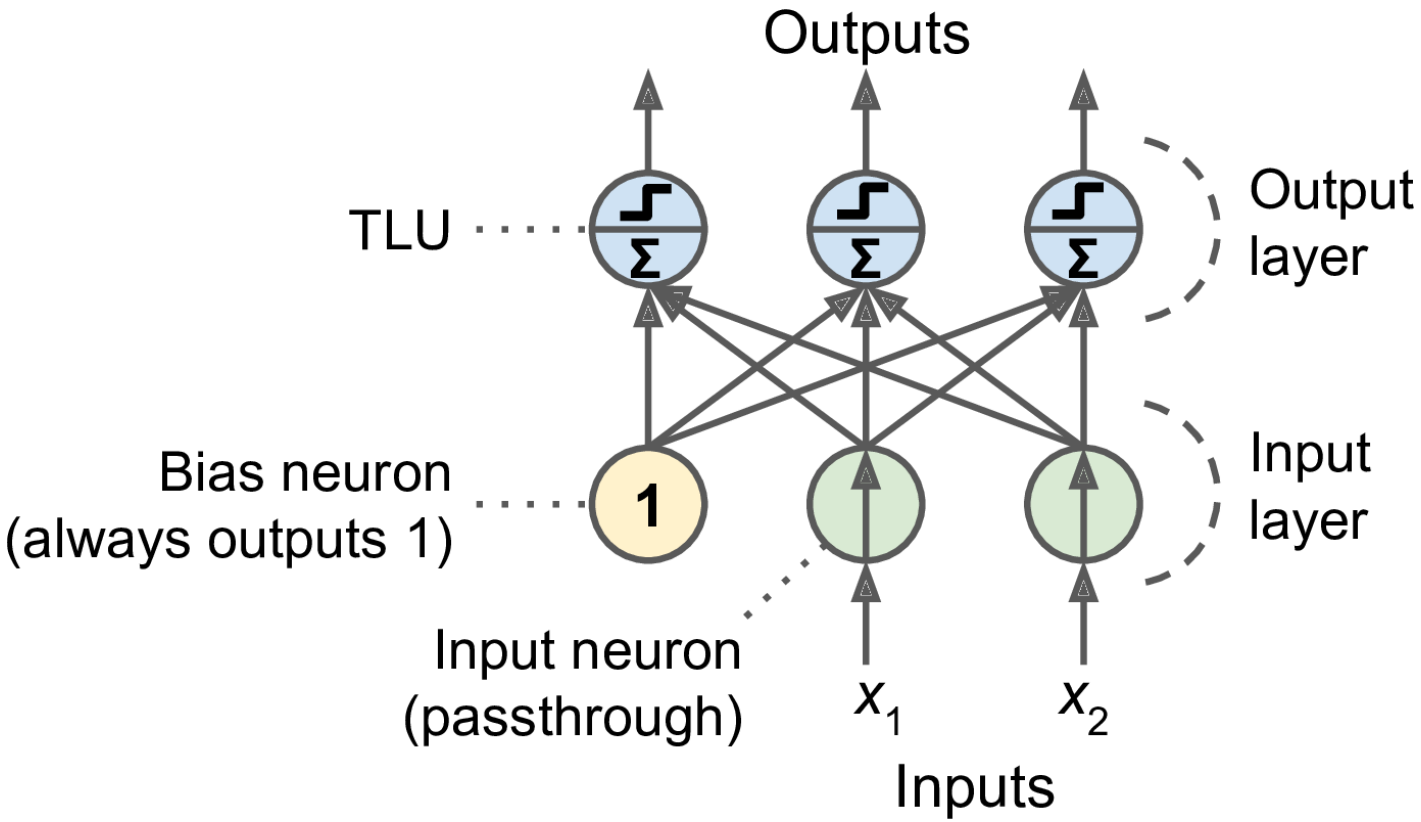
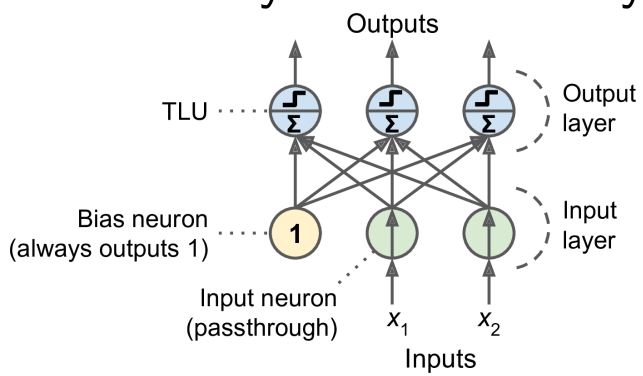


Figure 10-5. A Perceptron with two input, one bias, and three output neurons.

Perceptron: Computing the outputs

We can efficiently compute the outputs of a fully connected layer:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \phi(\mathbf{x}^\top \mathbf{W} + \mathbf{b}^\top)$$



where

- $\mathbf{x}$ represents the input feature
- $\mathbf{W}$ represents the unknown weight matrix which contains all the connection weights except for the ones from the bias neuron
- $\mathbf{b}$ is the bias vector containing all the connection weights between the bias neuron and the output neurons.
- ϕ is called the *activation function*.

How is a Perceptron trained?

The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb's rule* or *Hebbian learning* (name after Donald Hebb) which can be summarized as "cells that fire together, wire together".

How is a Perceptron trained?

The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb's rule* or *Hebbian learning* (name after Donald Hebb) which can be summarized as "cells that fire together, wire together".

More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

How is a Perceptron trained? (cont.)

Perceptron learning rule (weight update):

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

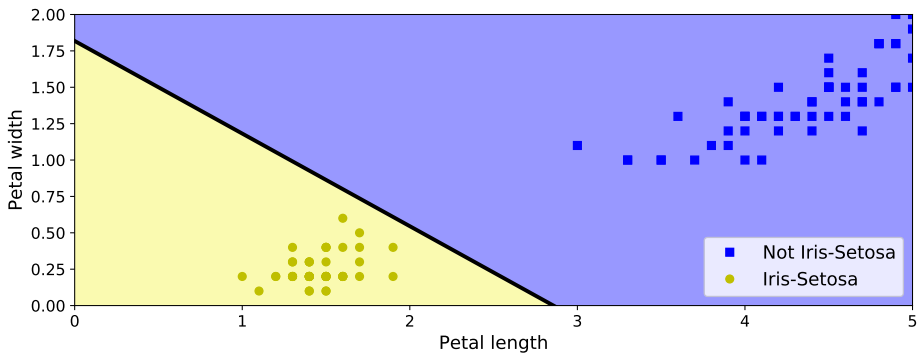
where

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

Implementing Perceptron in Scikit-Learn

Scikit-Learn provides a `Perceptron` class that implements a single TLU network:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris()
X = iris.data[:, (2, 3)]                    # petal length, petal width
y = (iris.target == 0).astype(np.int)       # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

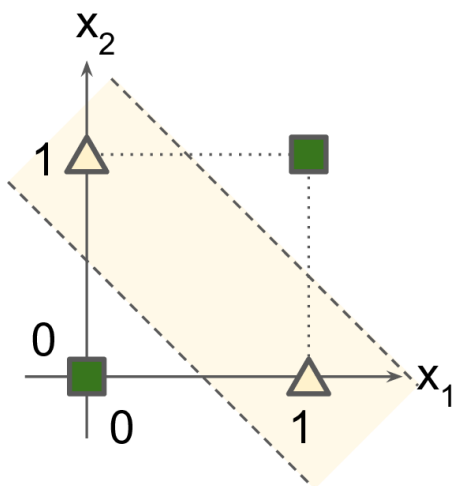


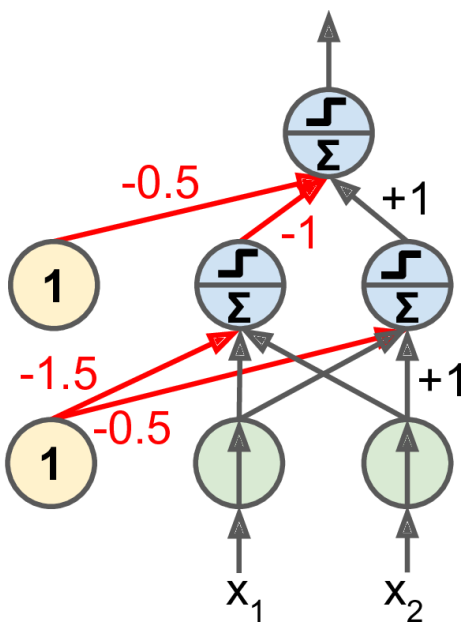
Decision boundary output by the code above.

Multi-Layer Perceptron

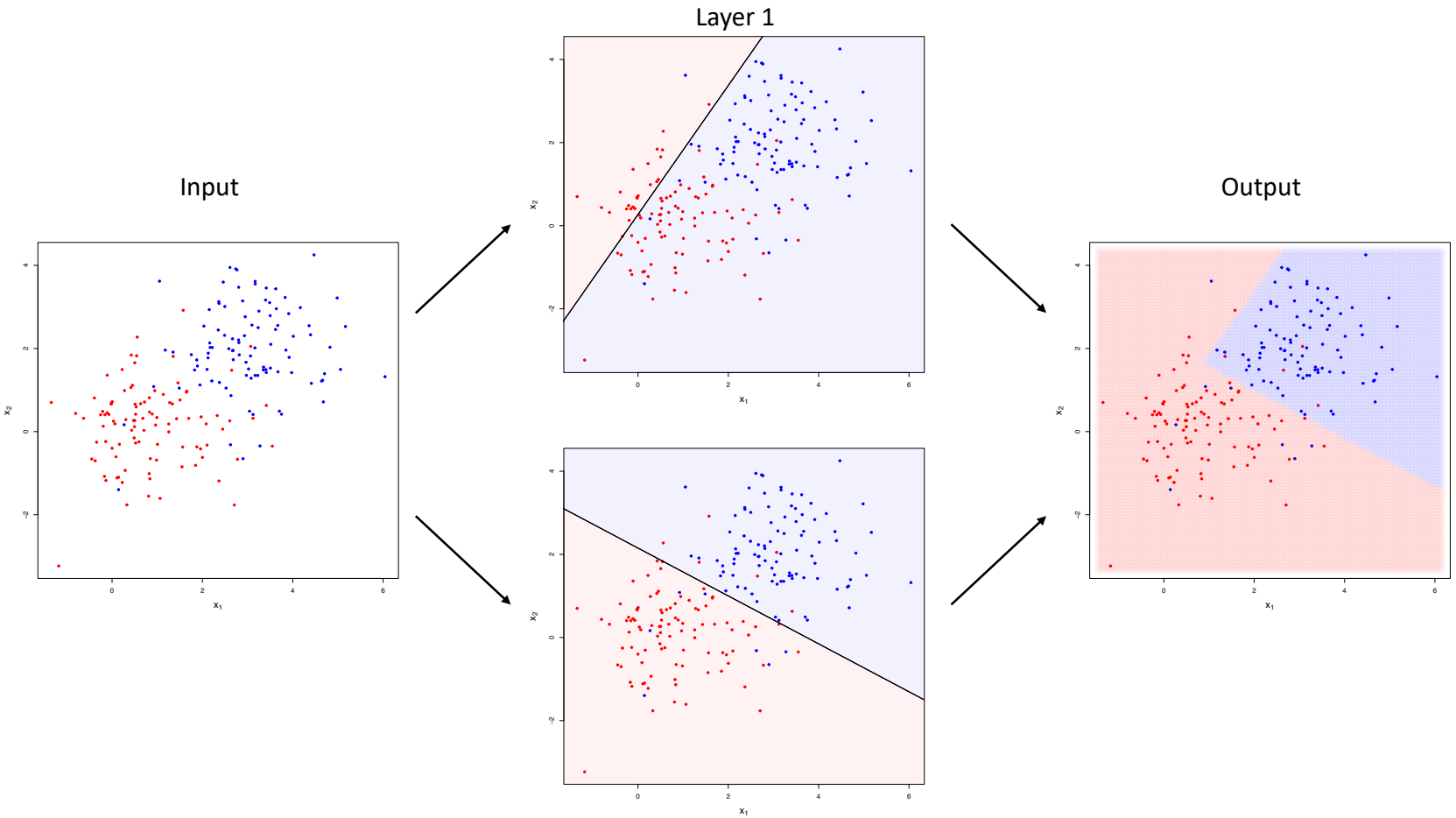
Some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multi-Layer Perceptron* (MLP).

In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP for each combination of inputs: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

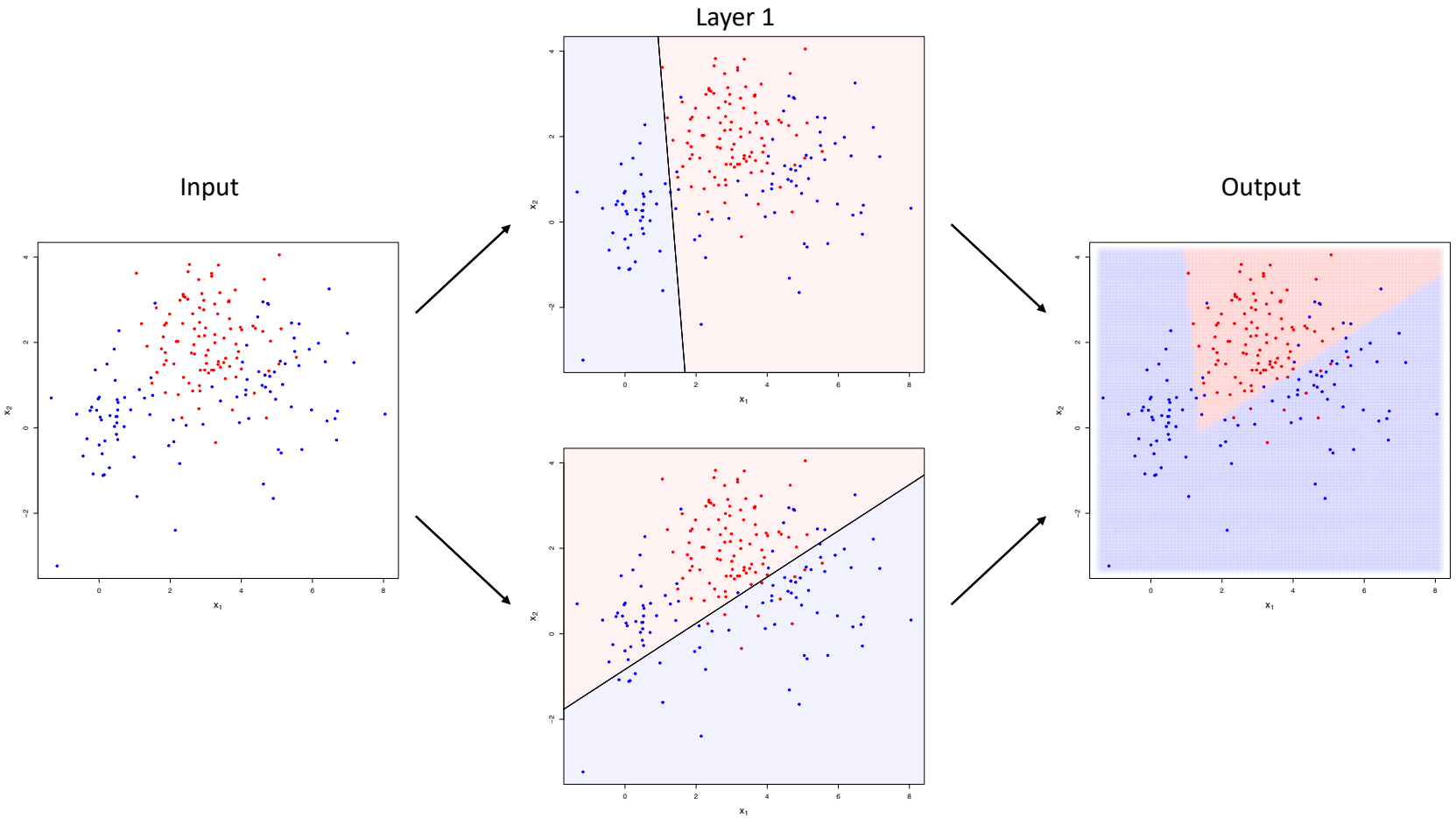




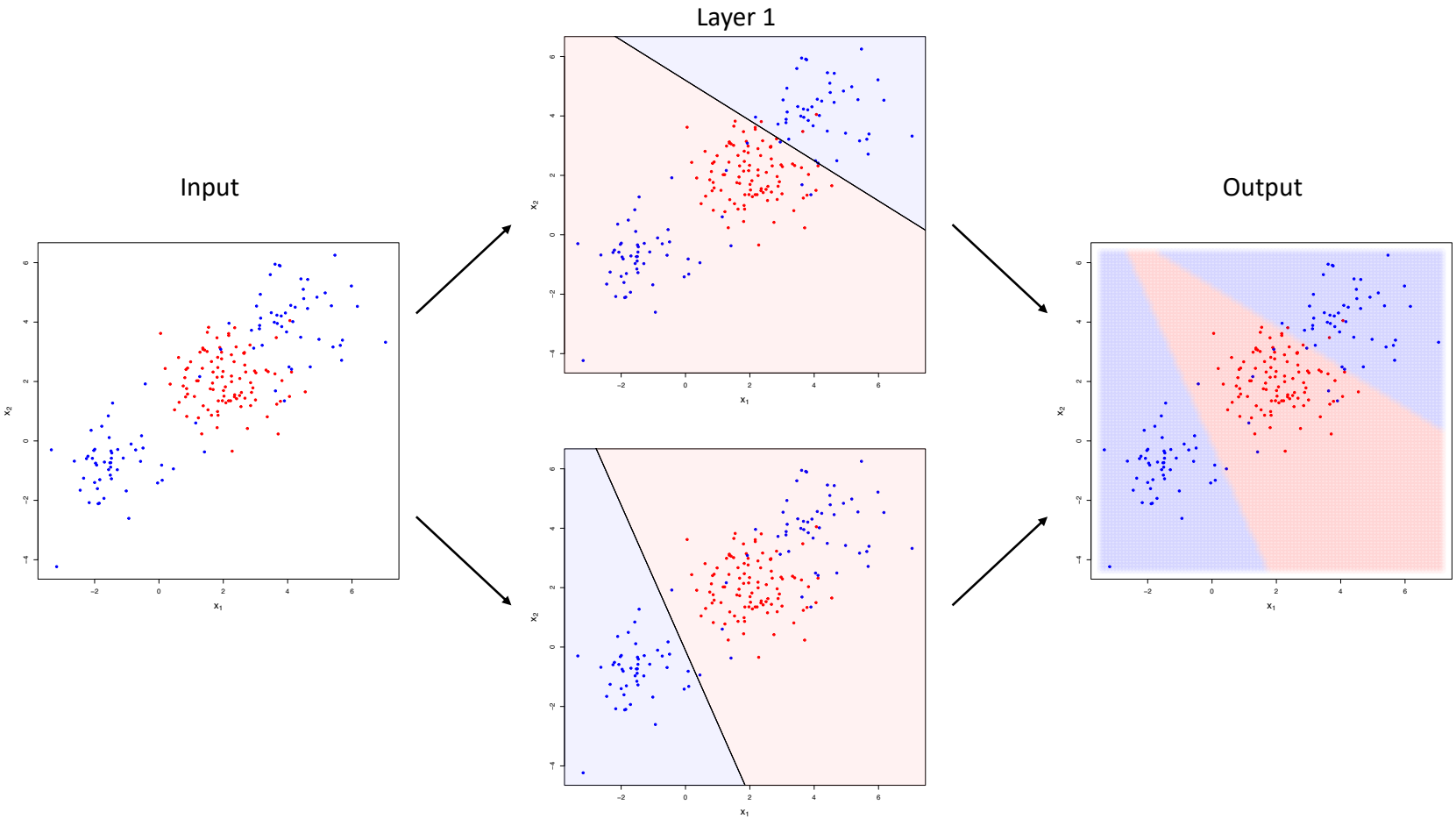
Multi-Layer Perceptron - Example



Multi-Layer Perceptron - Example



Multi-Layer Perceptron - Example



Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer*.

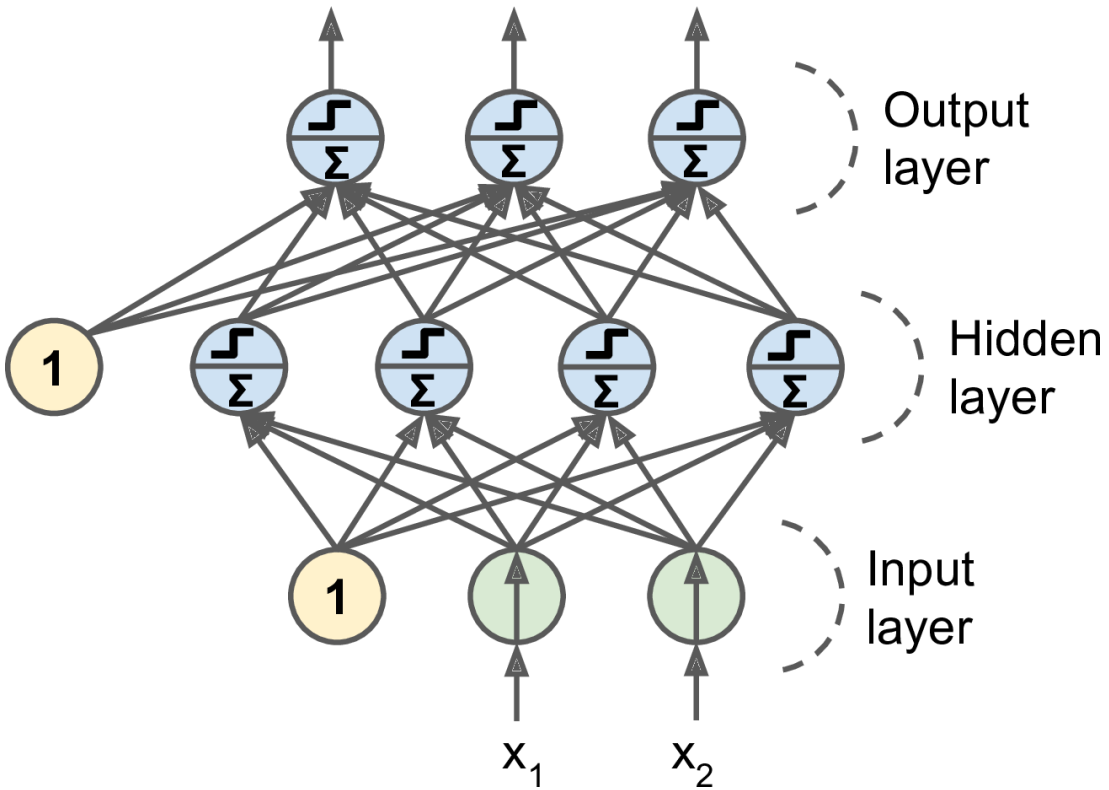


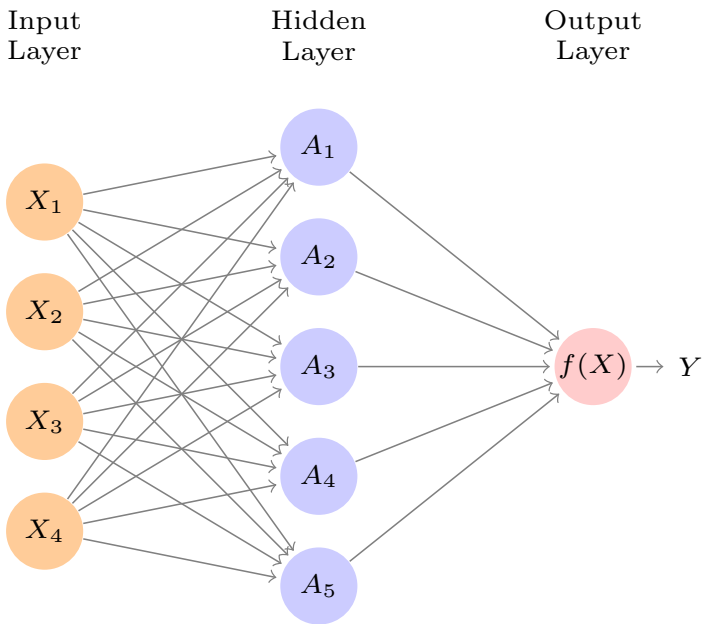
Figure 10-7. Multi-Layer Perceptron.

Multi-Layer Perceptron and Backpropagation (cont.)

Backpropagation training algorithm: Gradient Descent using the chain rule for differentiation.
For each training instance, the algorithm

- feeds it to the network and computes the output of every neuron in each layer (this is the *forward pass*),
- measures the network's output error (i.e., the difference between the desired output and the actual output of the network),
- goes through each layer in reverse order to measure the error contribution from each connection (this is the *reverse pass*) until the input layer is reached (typically applying the chain rule), and
- finally slightly tweaks the connection weights to reduce the error (this is the *Gradient Descent step*).

Multi-Layer Perceptron and Backpropagation (cont.)

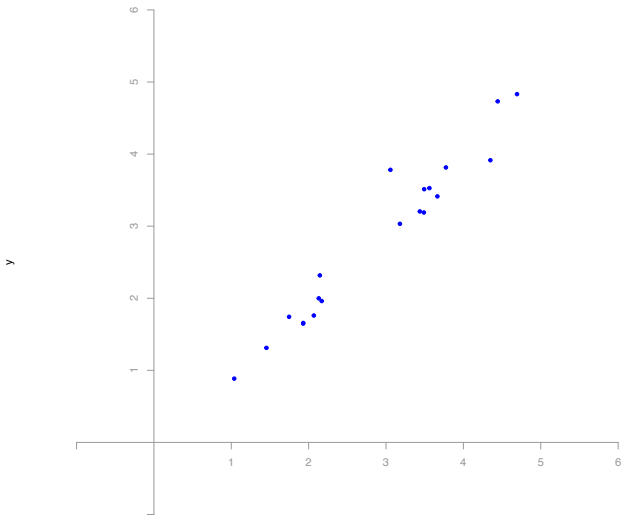


$$\underset{\{w_k\}_1^K,\beta}{\text{minimise}} \frac{1}{2}\sum_{i=1}^{m}(y_i - f(x_i))^2, \text{where}$$

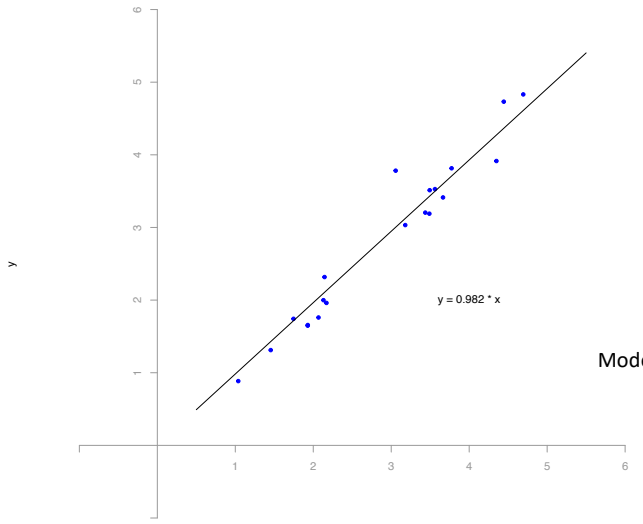
$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{n} w_{kj}x_{ij}\right).$$

Illustrating the Behaviour of the Loss Function in NNs

The data

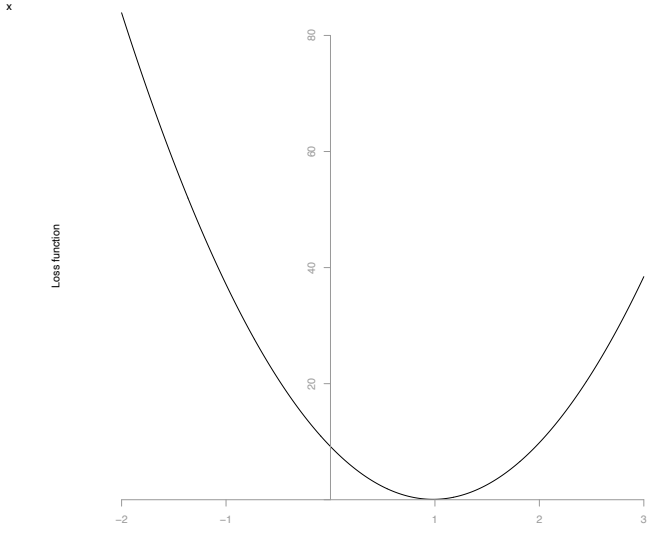


Linear regression



Model: $y = \beta_1 * x$

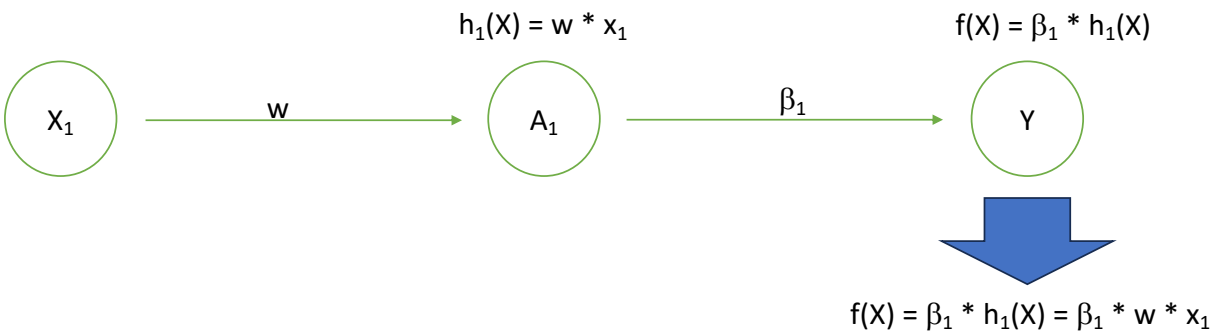
Loss function



β_1 ← There is only one parameter

Illustrating the Behaviour of the Loss Function in NNs

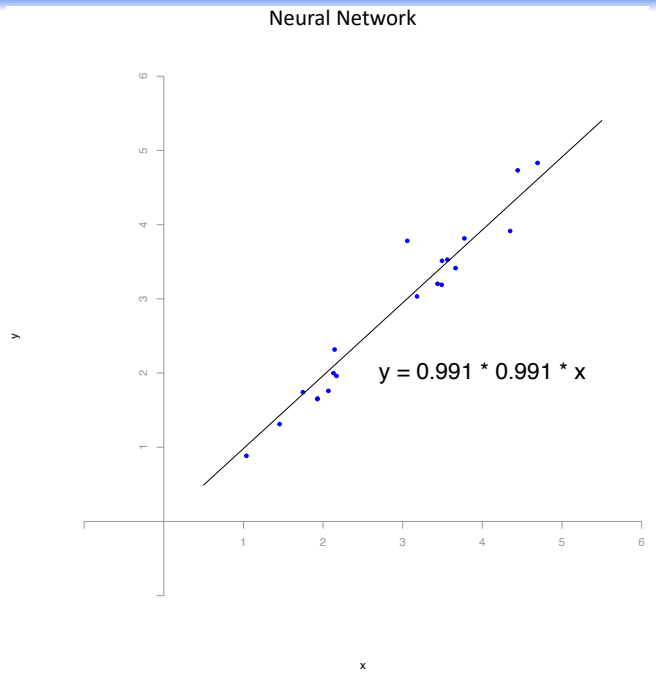
Neural Network

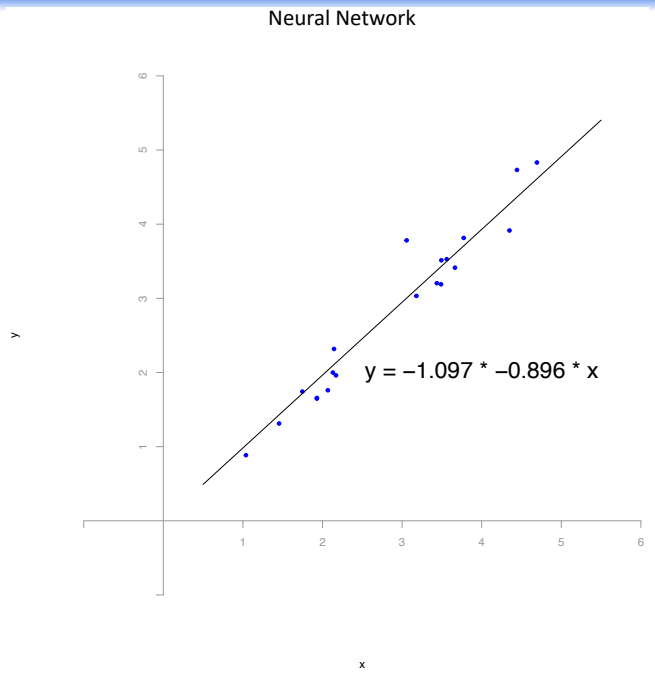


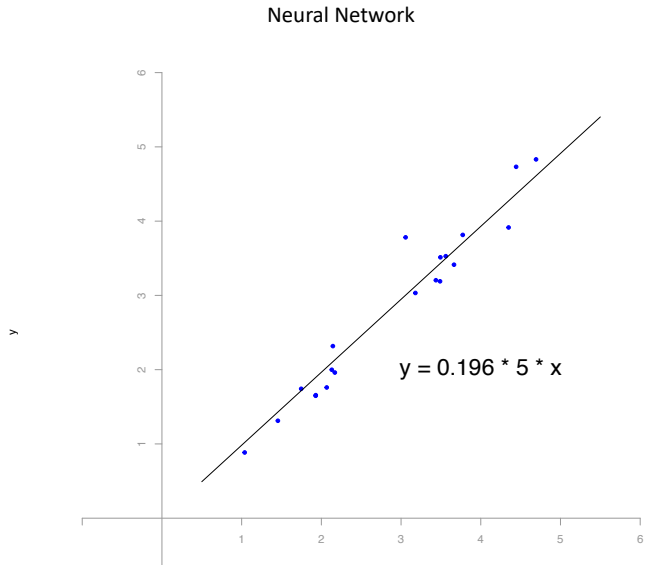
	Linear Regression	This Neural Network
Model:	$y = \beta_1 * x$	$y = \beta_1 * w * x$
No. param.:	1	2

Loss: MSE for both models.

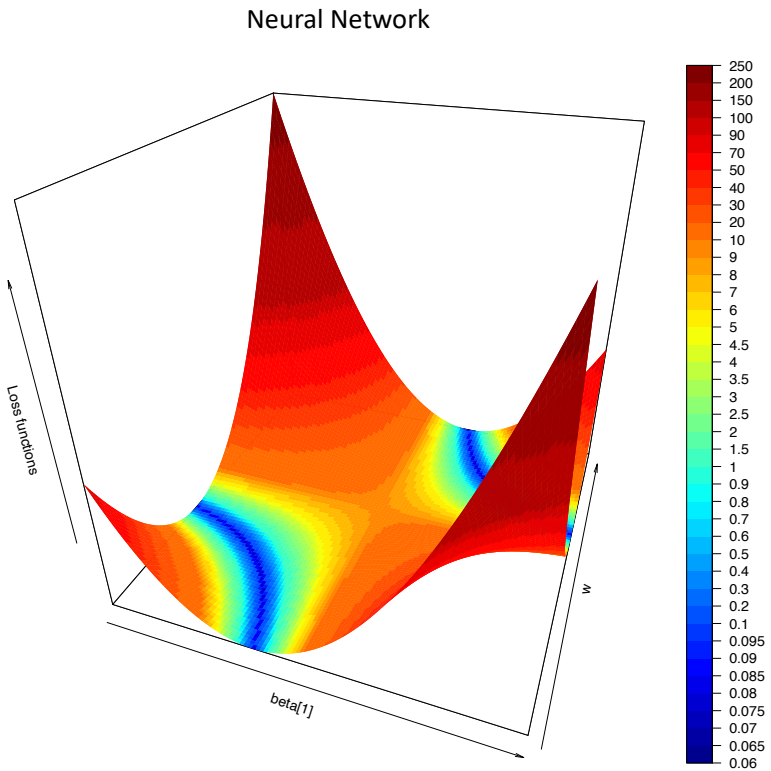
Illustrating the Behaviour of the Loss Function in NNs





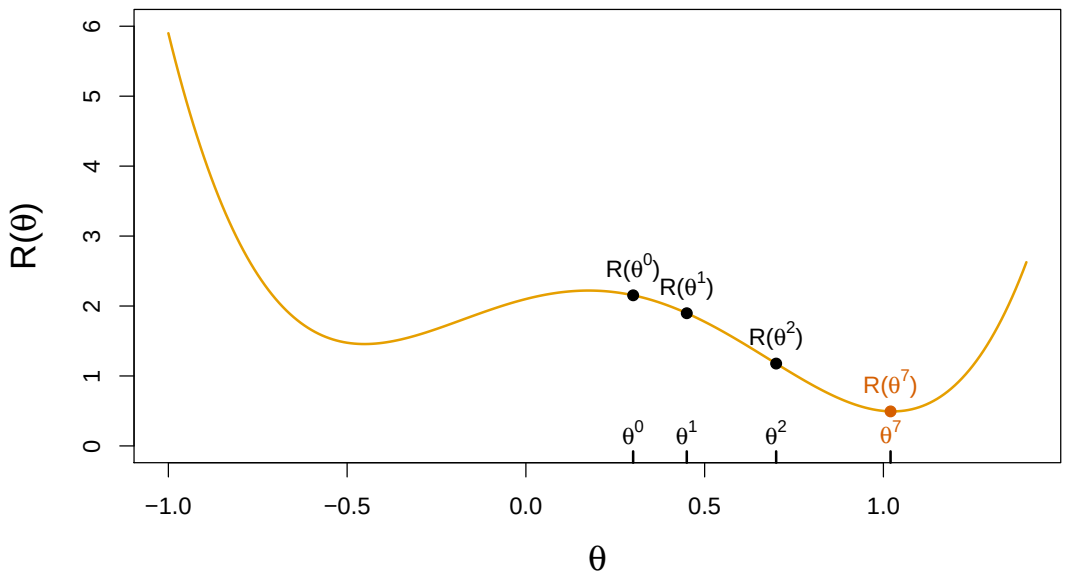


Illustrating the Behaviour of the Loss Function in NNs



Multi-Layer Perceptron and Backpropagation (back to the example)

Let $R(\theta) = \frac{1}{2}\sum_{i=1}^{m}(y_i - f_\theta(x_i))^2$ with $\theta = (\{w_k\}_1^K, \beta)$.



1. Randomly initialise θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective $R(\theta)$ stops decreasing:
   a. Compute a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective: $R(\theta^{t+1}) < R(\theta^t)$.
   b. Increment t, that is, $t \leftarrow t + 1$.

Multi-Layer Perceptron and Backpropagation (cont.)

To find the vector δ that points downhill, we compute the gradient vector, that is, the partial derivatives at the current guess θ^t.

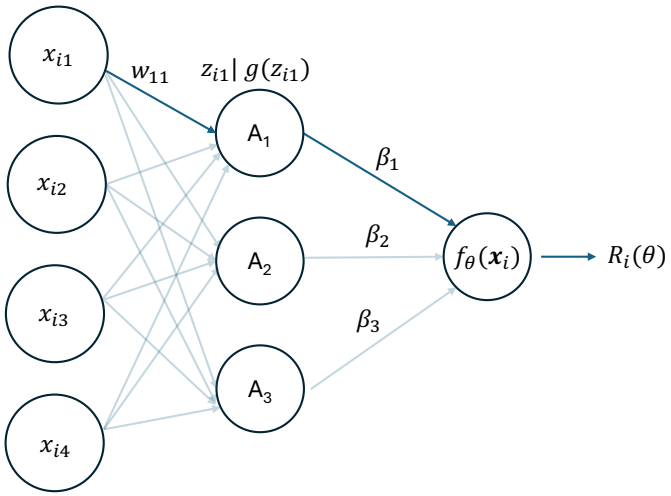
$$\nabla R(\theta^t) = \left. \frac{\partial R(\theta)}{\partial \theta} \right|_{\theta=\theta^t}$$

Then, update:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \, \nabla R(\theta^t),$$

where η is the learning rate.

Multi-Layer Perceptron and Backpropagation (cont.)



$$\nabla R(\theta) = \begin{pmatrix} \frac{\partial R_i(\theta)}{\partial \beta_1} \\ \frac{\partial R_i(\theta)}{\partial \beta_2} \\ \frac{\partial R_i(\theta)}{\partial \beta_3} \\ \frac{\partial R_i(\theta)}{\partial w_{11}} \\ \frac{\partial R_i(\theta)}{\partial w_{12}} \\ \frac{\partial R_i(\theta)}{\partial w_{13}} \\ \frac{\partial R_i(\theta)}{\partial w_{21}} \\ \frac{\partial R_i(\theta)}{\partial w_{22}} \\ \vdots \\ \frac{\partial R_i(\theta)}{\partial w_{43}} \end{pmatrix}$$

$R(\theta) = \sum_{i=1}^{m} R_i(\theta)$: sum of gradients.
$R_i(\theta) = \frac{1}{2}(y_i - f_\theta(x_i))^2$
$f_\theta(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g(z_{ik})$
$z_{ik} = w_{k0} + \sum_{j=1}^{n} w_{kj} x_{ij}$.

Then,

$$\frac{\partial R_i(\theta)}{\partial \beta_1} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \frac{\partial f_\theta(x_i)}{\partial \beta_1}$$

$$\frac{\partial R_i(\theta)}{\partial w_{11}} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \frac{\partial f_\theta(x_i)}{\partial g(z_{i1})} \frac{\partial g(z_{i1})}{\partial z_{i1}} \frac{\partial z_{i1}}{\partial w_{11}}$$

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

In order for the backpropagation training algorithm to work, Rumelhart et al made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function has no gradient to work with, while the logistic function has a well-defined nonzero derivative everywhere.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

In order for the backpropagation training algorithm to work, Rumelhart et al made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function has no gradient to work with, while the logistic function has a well-defined nonzero derivative everywhere.

Such a function which defines the output of a neuron (or node) given an input or set of inputs is known as the *activation function*.

(The derivative is: $\sigma'(z) = (1 + \exp(-z))^{-2} \exp(-z) = \frac{\exp(-z)}{(1+\exp(-z))^2} = \frac{1+\exp(-z)-1}{(1+\exp(-z))^2} = \frac{1}{1+\exp(-z)} - \frac{1}{(1+\exp(-z))^2} = \sigma(z) - \sigma^2(z) = \sigma(z)(1 - \sigma(z))$)

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

Apart from the logistic function, two other popular activation functions are:

- *The hyperbolic tangent function*: $\tanh(z) = 2\sigma(2z) - 1$. This function has a similar S shape as the logistic function, is continuous and differentiable, but its values range from -1 to $+1$.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

Apart from the logistic function, two other popular activation functions are:

- *The hyperbolic tangent function*: $\tanh(z) = 2\sigma(2z) - 1$. This function has a similar S shape as the logistic function, is continuous and differentiable, but its values range from -1 to $+1$.
- *The ReLU function*: $\text{ReLU}(z) = \max(0, z)$. This function is continuous but, unfortunately, not differentiable everywhere. However, in practice, it works very well and has the advantage of being fast to compute. Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent.

Activation functions

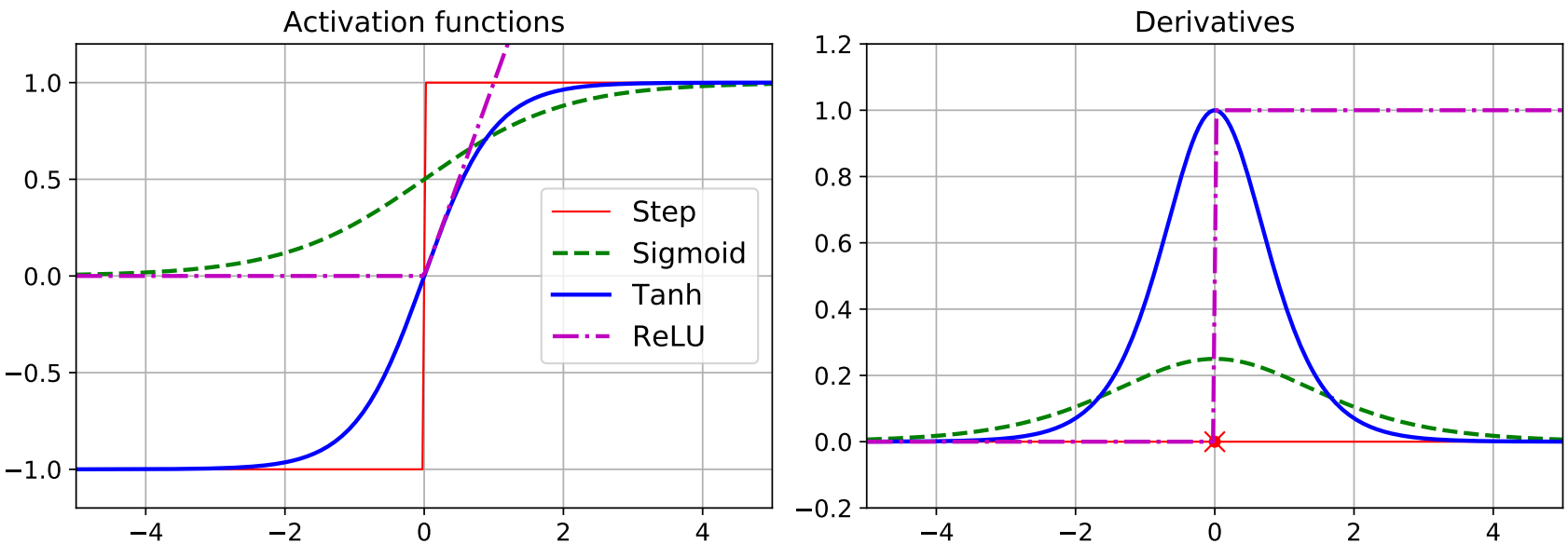


Figure 10-8. Activation functions and their derivatives

Fine-Tuning Neural Network Hyperparameters

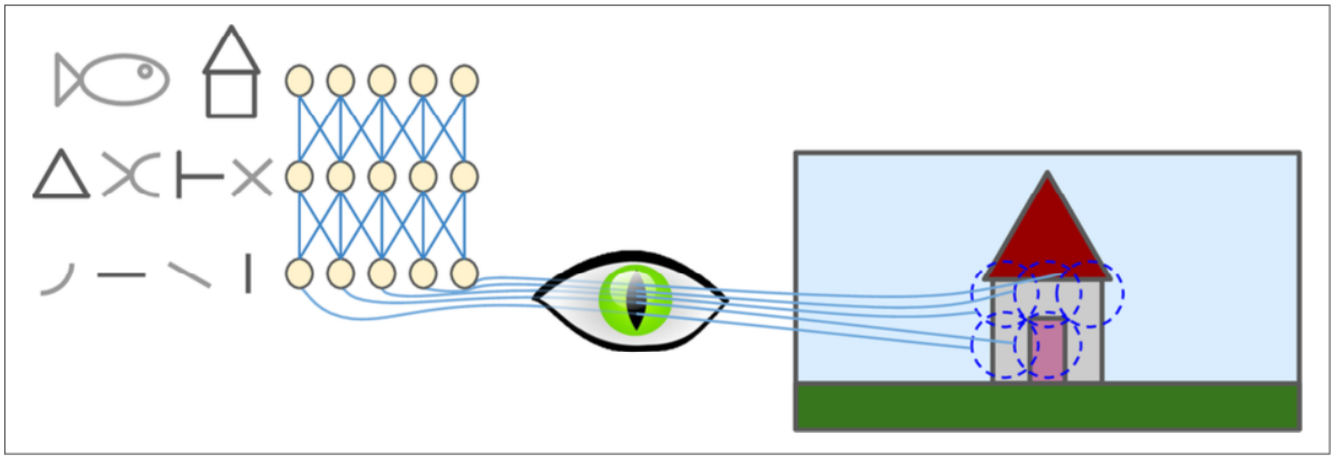
The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. We can use `GridSearchCV` or `RandomizedSearchCV` to explore the hyperparameter space.

There are also many Python libraries you can use to optimise hyperparameters: Keras Tuner, Scikit-Optimize (skopt), Spearmint, etc.
Examples of hyperparameters:

- Number of Hidden Layers;
- Number of Neurons per Hidden Layer;
- Weights Initialisation;
- Activation Functions;
- Optimisers;
- Regularisation Strategies.

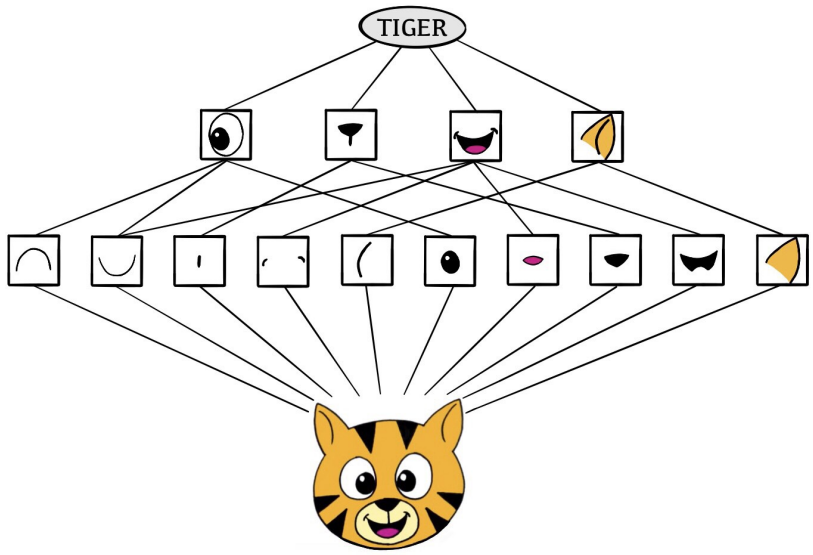
Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex and the concept of local receptive field: neurons react only to visual stimuli located in a limited region of the visual field.



Popular for image recognition since the 1980s.

Convolutional Neural Networks (CNNs)



Idea: build up an image in a hierarchical fashion.

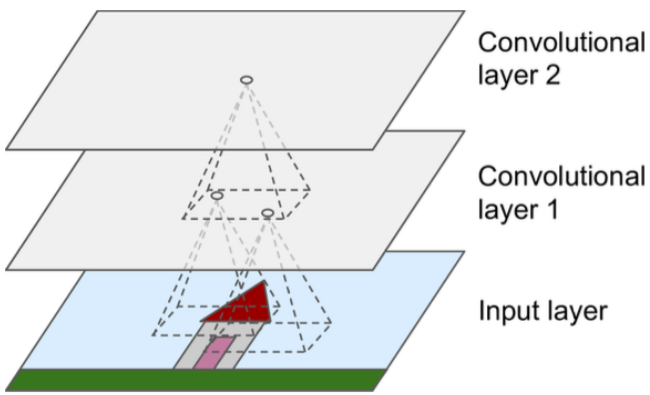
Edges and shapes are recognised and combined to form more complex shapes.

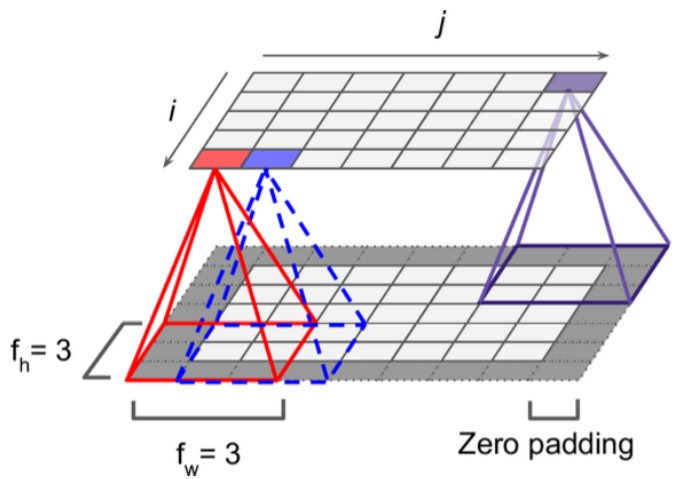
Hierarchical construction via a combination of two types of layers: convolutional and pooling.

Convolutional Layers

$$\text{Input Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \qquad \text{Convolutional Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

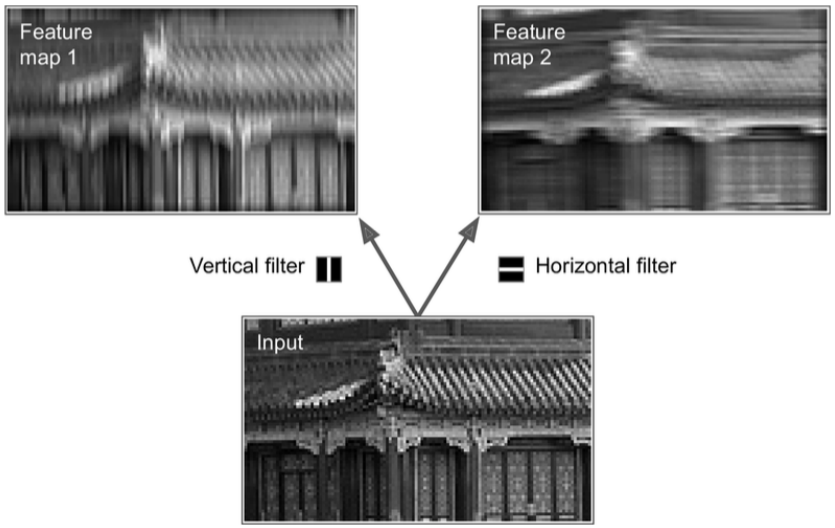
$$\text{Convoluted Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$





The weights in the filters are learned during training.

Convolutional Layers



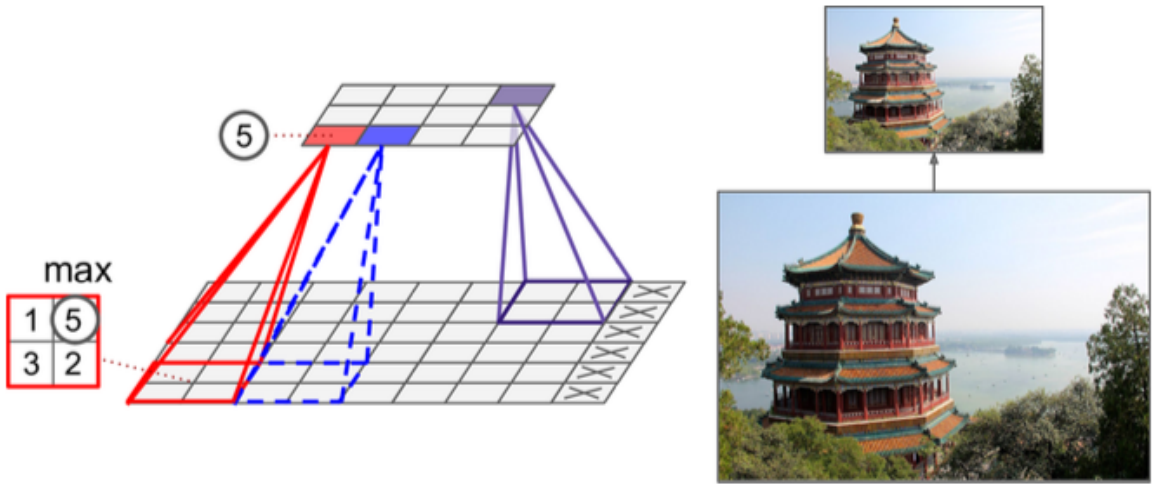
The idea of the convolution with a filter is to find common patterns that occur in different parts of the image.

A layer of neurons using the same filter outputs a feature map, which highlights the areas in an image that activate the filter the most.

The convolutional layer will learn the most useful filters for its task. Layers above will learn to combine them into more complex patterns.

A convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter.

Pooling Layers



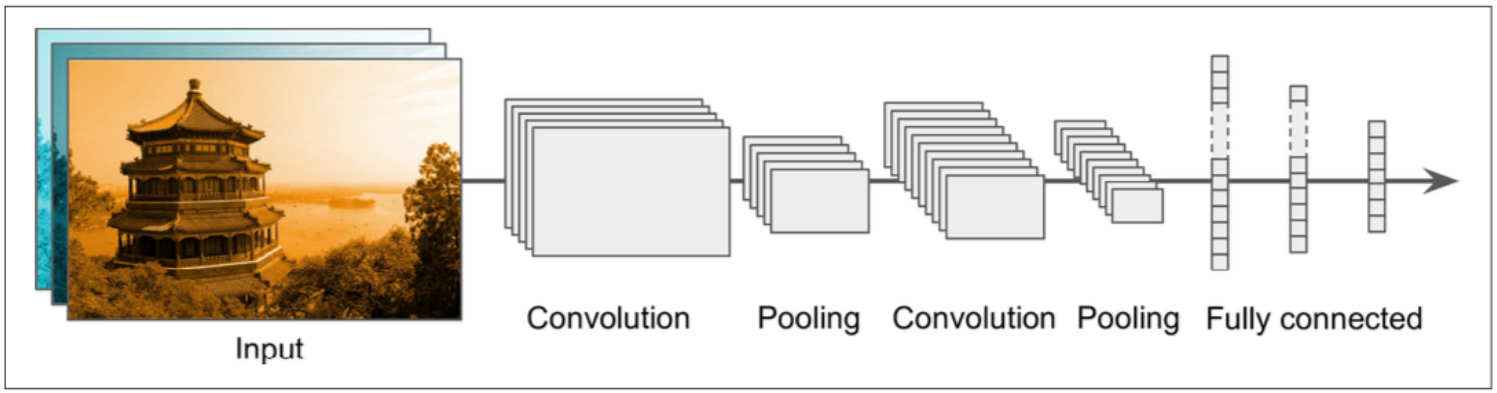
A pooling neuron has no weights: it aggregates the inputs using an aggregation function such as the max or mean.

This sharpens the feature identification.

Reduces computation, memory usage, and the number of parameters.

Introduces some level of locational invariance.

Typical CNN Architecture



Popular architectures: LeNet-5, AlexNet, GoogleNet, VGGNet, ResNet, Xception, SENet.

Transfer Learning: Training a network that has been pre-trained on a similar problem (reuse the lower layers of a pre-trained model).

Recurrent Neural Networks (RNNs)

ANNs architectures that take into account the temporal context of the data.

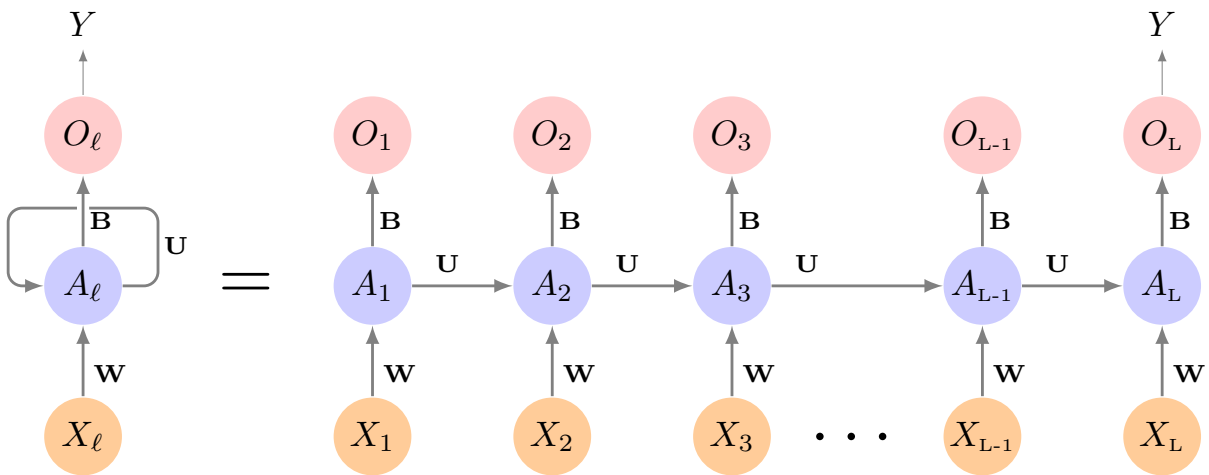
- Sentences, documents: relative position of the words are meaningful.
- Time-series data (e.g. weather, financial, sensors, etc.).
- Speech, music.

RNNs are built to account for the sequential nature of the data and build a memory of the past.

The feature vector for each example is a sequence of vectors $X = X_1, X_2, \ldots, X_L$.

The target Y can be a single variable, one-hot vector for multiclass, or a sequence (e.g. the same document in a different language).

Recurrent Neural Networks (RNNs)



The hidden layer is a sequence of vectors A_l that receives as input X_l **and** A_{l-1}. A_l produces an output O_l.

Shared weights **W**, **U** and **B** at each step in the sequence.

The A_l represents an evolving model for the response that is updated as each element X_l is processed.

Recurrent Neural Networks (RNNs)

Advances RNNs: Long Short-Term Memory Neural Network (LSTM), Gated Recurrent Units (GRU), combinations with CNNs.

Examples of applications:

- Language translation;
- Video activity recognition;
- Sentiment analysis;
- Speech recognition;
- Anomaly detection;
- Time-series prediction and time-series classification.

Software

Software available for neural networks and deep learning: **Tensorflow** from Google and **PyTorch** from Facebook. Both are Python packages.

Tensorflow and **Keras** API in Python. Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks.
https://keras.io/.

For next week

Work through Assignment 3 and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

We will have two guest speakers from industry in the next class.

Interactive Google platform for playing with ANNs:
playground.tensorflow.org

Have a good week. :)