Deep Learning

Week 2: Multilayer perceptrons

Contents

Introduction

In the last week of the module we reviewed some important concepts in machine learning, including generalisation, validation, dataset splits, overfitting/underfitting and regularisation. We also took a look at the most fundamental building blocks and operations in Keras. You saw how the low level objects Tensors and Variables are including in these models to encapsulate mutable parameters and computational operations.

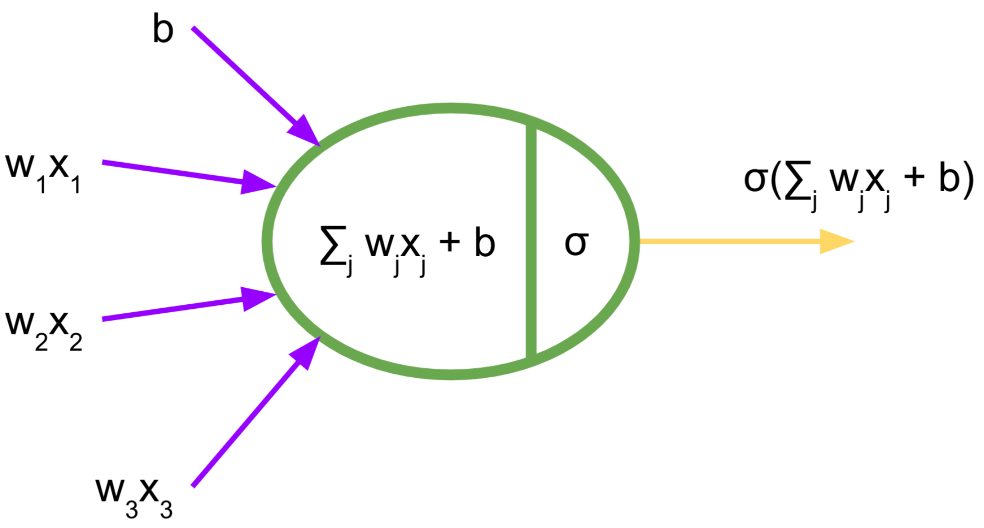
This week, we will introduce the concept of the artifical neuron, and define the simplest of deep learning architectures; the multilayer perceptron (MLP) or feedforward network. We will review the stochastic gradient descent algorithm that is commonly used to train neural network models.

We will see how to build and train MLP models in Keras, using the Sequential API. These MLP models can be constructed from Dense layer objects, which inherit from the base Layer class in Keras. We will also see how we can write our own custom Layer objects by subclassing the base Layer class.

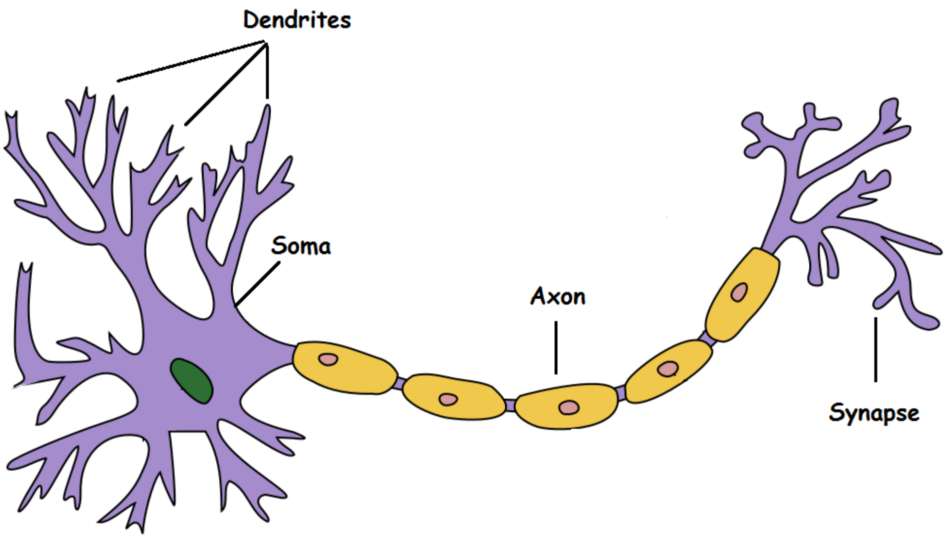
Mathematical neuron

Early research into neural networks focused on models of learning in the brain, and used mathematical (or artificial) neurons as fundamental building blocks. These are simple models of neurons in the brain, that receive a set of inputs, which are weighted and summed before being passed through an activation function (or transfer function):

$$y_k = \sigma\left(\sum_j w_{kj} x_j + b_k\right) \tag{4}$$



Sketch of a mathematical neuron.



Sketch of a biological neuron. Source: Wikipedia.

In the above, the inputs to the neuron are denoted by x_j, the weights w_{kj}, bias b_k and activation function σ. The weights and bias are parameters that need to be tuned for the given task. The first artificial neuron was developed by McCulloch and Pitts, which used a simple threshold activation function (step function) only on binary inputs, and produce a binary output. Later, Rosenblatt developed the **perceptron**, which also used a step function threshold for binary classification (but with more general weights and inputs), and importantly also introduced a learning algorithm for the weights. The perceptron learning algorithm is guaranteed to converge for linearly separable data. However, the limitations of linear models was largely responsible for the decline in interest in neural networks until its revival in the 1980s.

McCulloch-Pitts neuron

As an example, we will use Tensors to implement the McCulloch-Pitts neuron for a simple logical function. The McCulloch-Pitts neuron operates on boolean inputs, and uses a threshold activation to produce a boolean output. The function can be written as

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i x_i \geq b \\ 0 & \text{if } \sum_i x_i < b \end{cases}$$

In [2]:
```python
import keras
from keras import ops
```

In [3]:
```python
# Define the AND function

def logical_and(x):
    return ops.cast(ops.greater_equal(ops.sum(x),
                                      ops.prod(ops.shape(x))), 'int32')
```

In [4]:
```python
# Test the AND function with a few examples

print(logical_and(ops.array([1, 1])))
print(logical_and(ops.array([1, 1, 0])))
print(logical_and(ops.ones((2, 3), dtype='int32')))
```

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
```

In [5]:
```python
# Define the OR function

def logical_or(x):
    return ops.cast(ops.greater_equal(ops.sum(x), 1), 'int32')
```

In [6]:
```python
# Test the OR function with a few examples

print(logical_or(ops.array([1, 0])))
print(logical_or(ops.zeros(3,)))
```

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(0, shape=(), dtype=int32)
```

Exercise. Define the function for the NOR operation below (all inputs must be zero) for inputs `x`. *Hint: use the* `keras.ops.logical_not` *function.*

In [7]:
```python
# Define the NOR function

def logical_nor(x):
    pass
```

In [8]:
```python
# Test the NOR function with a few examples

print(logical_nor(ops.array([1, 0])))  # False
print(logical_nor(ops.array([0, 0])))  # True
print(logical_nor(ops.array([0, 0, 0])))  # True
print(logical_nor(ops.array([1, 0, 1])))  # False
```

```
None
None
None
None
```

The perceptron

The perceptron is also a linear binary classifier, but with more flexible weights. It can be written as the following function

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{if } \sum_i w_i x_i + b < 0 \end{cases}$$

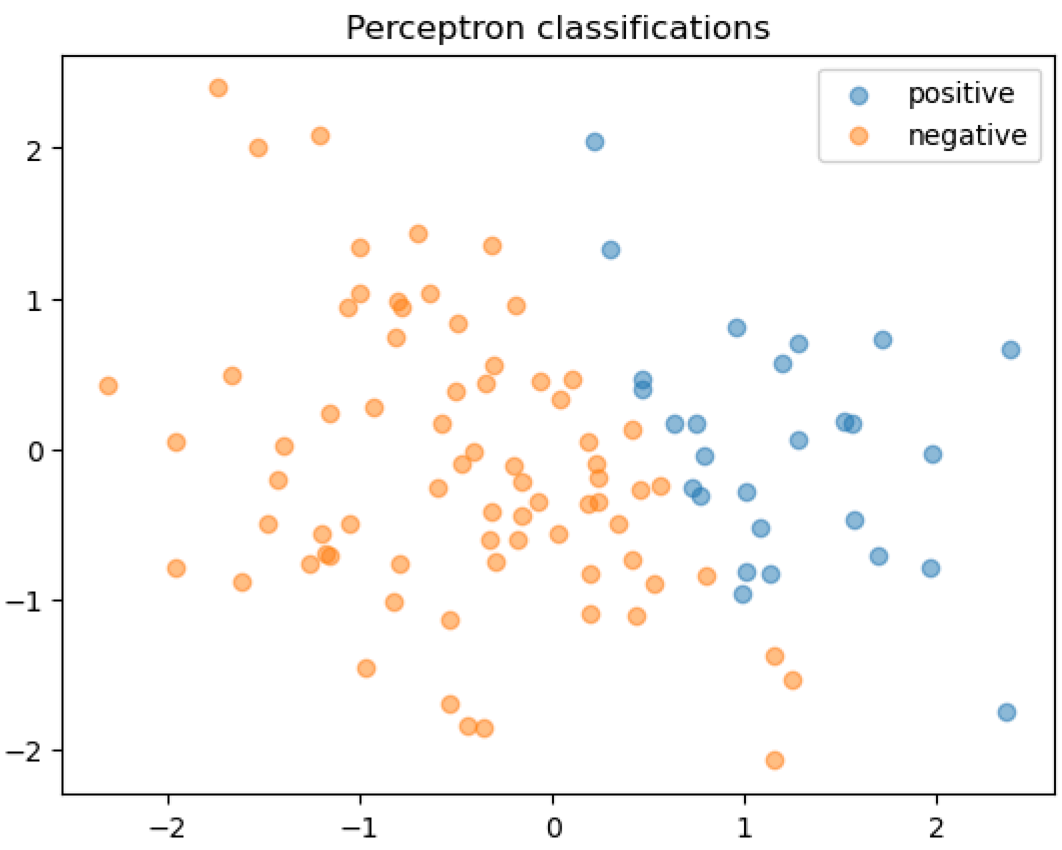
As an example, we will use Tensors to implement the perceptron classifier.

In [9]:
```python
# Implement the weights and bias as Tensors

weights = ops.array([1., 0.5])
bias = ops.array(-0.5)
```

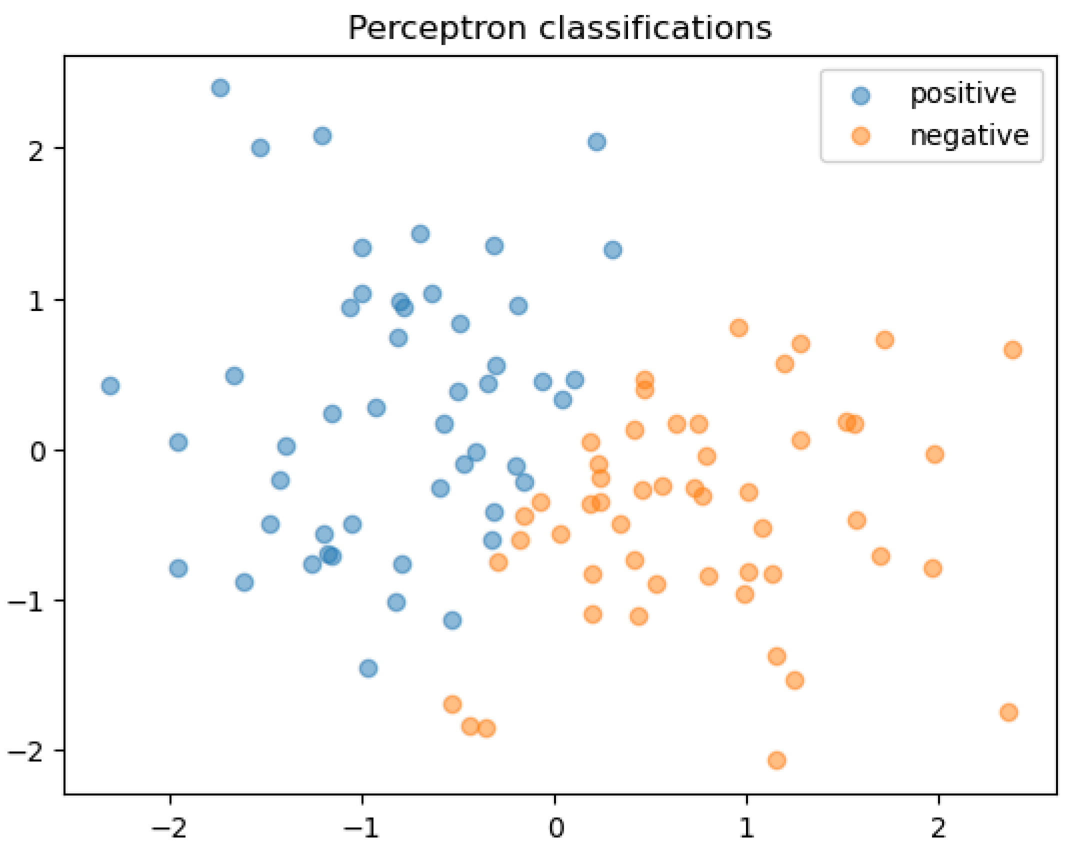
In [10]:
```python
# Define the perceptron classifier

def perceptron(x):
    return ops.greater_equal(ops.tensordot(x, weights, axes=1) + bias, 0.
```

In [11]:
```python
# Create a random set of test points

x = keras.random.normal((100, 2))
```

In [12]:
```python
# Plot the points coloured by class prediction

import matplotlib.pyplot as plt

preds = perceptron(x)
positive_class = x[preds]
negative_class = x[~preds]
plt.scatter(ops.convert_to_numpy(positive_class[:, 0]),
            ops.convert_to_numpy(positive_class[:, 1]),
            alpha=0.5, label='positive')
plt.scatter(ops.convert_to_numpy(negative_class[:, 0]),
            ops.convert_to_numpy(negative_class[:, 1]),
            alpha=0.5, label='negative')
plt.title("Perceptron classifications")
plt.legend()
plt.show()
```



In [13]:
```python
# Update the weights and bias and re-plot

weights -= ops.array([1.5, 0.3])
bias += ops.array(0.5)

preds = perceptron(x)
positive_class = x[preds]
negative_class = x[~preds]
plt.scatter(ops.convert_to_numpy(positive_class[:, 0]),
            ops.convert_to_numpy(positive_class[:, 1]),
            alpha=0.5, label='positive')
plt.scatter(ops.convert_to_numpy(negative_class[:, 0]),
            ops.convert_to_numpy(negative_class[:, 1]),
            alpha=0.5, label='negative')
plt.title("Perceptron classifications")
plt.legend()
plt.show()
```



Exercise. Can you find weights and bias values to implement the NOT gate for $x \in \{0, 1\}$ and the XOR gate for $x \in \{0, 1\}^2$? If yes, what are the values? If no, why not?

Stochastic gradient descent

The second wave of interest in neural networks in the 80s was driven in large part by the connectionist movement (see e.g. Rumelhart et al (1986a)), which focused on the concept of intelligent behaviour arising out of many simple computations composed together, with knowledge being distributed across many units. Smooth activation functions were increasingly studied, as they allowed gradient-based methods such as stochastic gradient descent (SGD, Robbins & Monro 1951) to be used in the optimisation of model parameters.

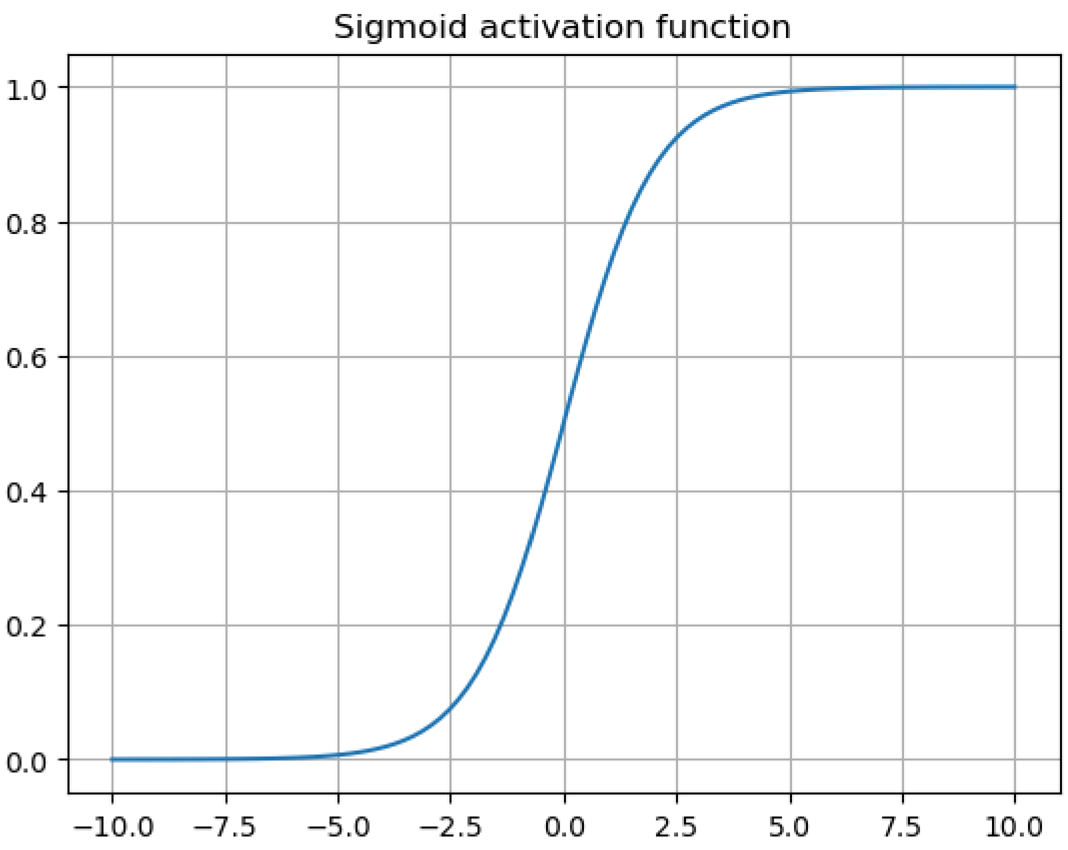
Activation functions

A typical example of a smooth activation function is the logistic sigmoid:

In [14]:
```
import keras
from keras import ops
```

In [15]:
```
# Plot the sigmoid function using the Keras implementation

import matplotlib.pyplot as plt
import numpy as np

x = ops.linspace(-10, 10, 100)
```

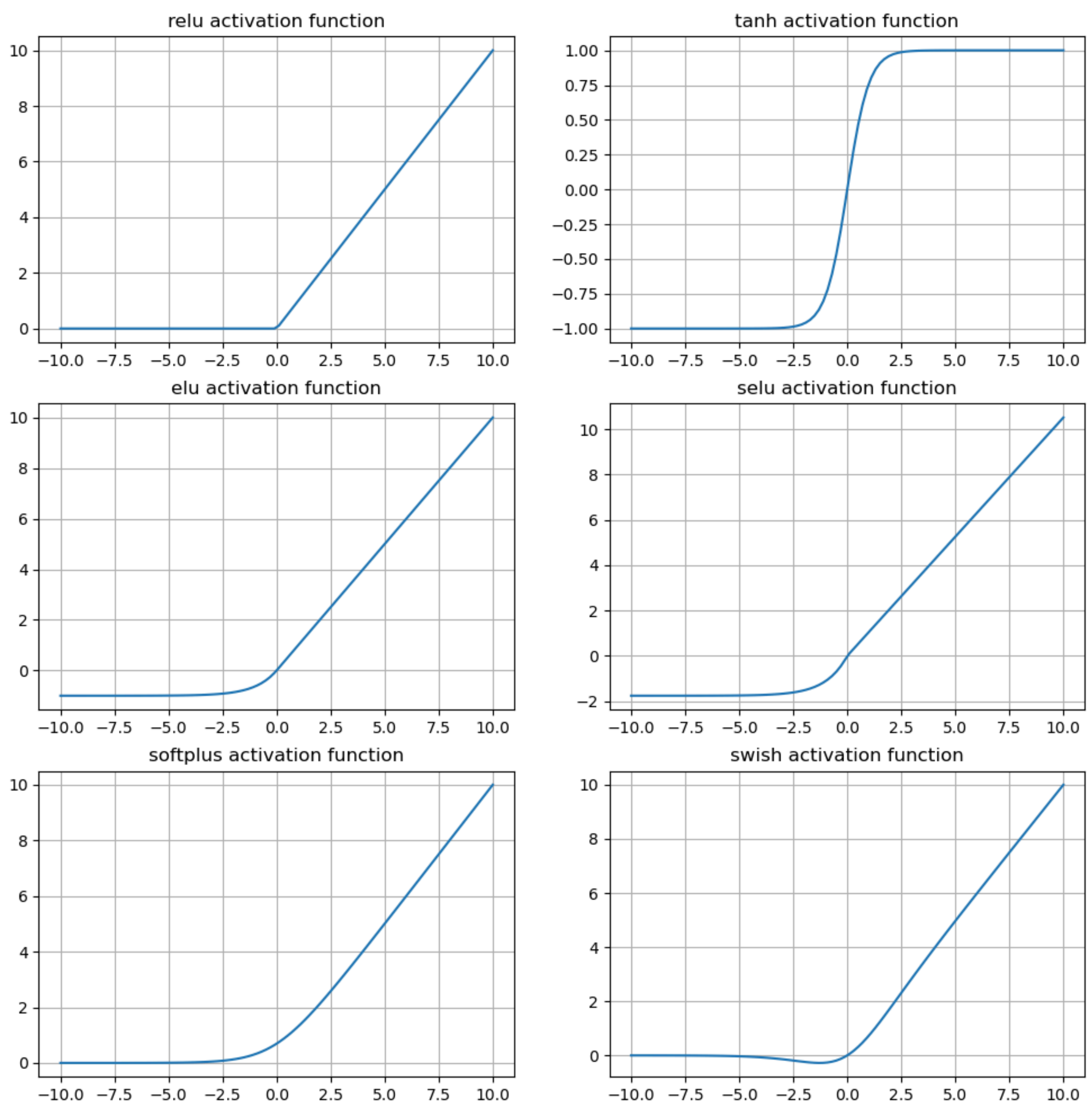
```
y = keras.activations.sigmoid(x)
x, y = ops.convert_to_numpy(x), ops.convert_to_numpy(y)
plt.grid()
plt.plot(x, y)
plt.title("Sigmoid activation function")
plt.show()
```



Note that linear regression and logistic regression can both be viewed as artificial neuron models, with linear (or no) activation function and sigmoid activation function respectively.

Other activation functions that are commonly used in deep learning models are the ReLU (rectified linear unit), tanh, ELU (exponential linear unit, Clevert et al 2016), SELU (scaled exponential linear unit, Klambauer et al 2017), softplus, swish (Ramachandran et al 2018).

In [16]:
```
# Plot example activation functions

x = ops.linspace(-10, 10, 100)
fig = plt.figure(figsize=(12, 12))
for i, activation in enumerate(
    ['relu', 'tanh', 'elu', 'selu', 'softplus', 'swish']):
    row = (i % 2) + 1
    col = (i // 2) + 1
    fig.add_subplot(3, 2, i + 1)
    plt.grid()
    plt.title("{} activation function".format(activation))
    plt.plot(ops.convert_to_numpy(x),
             ops.convert_to_numpy(
                 getattr(keras.activations, activation)(x)))
plt.show()
```



You can see a complete list of available activation functions in `keras.activations` here.

Gradient descent

Suppose we have constructed our neural network model, which we represent as the function $f_\theta : \mathbb{R}^D \mapsto Y$, where Y is the target space (e.g. $\mathbb{R}$ or $[0,1]$). Suppose also that we have defined a suitable loss function

$$L(\theta; \mathcal{D}_{train}) := \frac{1}{|\mathcal{D}_{train}|} \sum_{x_i, y_i \in \mathcal{D}_{train}} l(y_i, f_\theta(x_i)),$$

where $l(y_i, f_\theta(x_i))$ is the per-example loss. Then the gradient $\nabla_\theta L(\theta_0; \mathcal{D}_{train})$ evaluated at θ_0 defines the direction of steepest ascent in parameter space at the point θ.

The gradient descent algorithm takes an initial guess for the parameters θ_0 and updates the parameter values according to the rule

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{D}_{train}), \qquad t \in \mathbb{N}_0$$

where $\eta_t > 0$ is a small learning rate which may depend on t. For a suitably chosen η_t, the iterates $L(\theta_t; \mathcal{D}_{train})$, $t \in \mathbb{N}_0$ converge to a local minimum.

Stochastic gradient descent

Note that computing $\nabla_\theta L(\theta; \mathcal{D}_{train})$ as above requires computing the gradients of the per-example loss for every element in the training set. For large datasets (and large models) this can be prohibitively expensive.

Stochastic gradient descent provides a cheaper estimate of the full gradient, by computing the gradient on a minibatch of data points, instead of the full dataset. In particular, we evaluate the gradient

$$L(\theta; \mathcal{D}_m) = \frac{1}{M} \sum_{x_i, y_i \in \mathcal{D}_m} l(y_i, f_\theta(x_i)),$$

where $\mathcal{D}_m$ is a randomly sampled minibatch of training data points, $M = |\mathcal{D}_m|$ is the size of the minibatch (typically much smaller than $|\mathcal{D}_{train}|$). We then use the gradient $\nabla_\theta L(\theta_t; \mathcal{D}_m)$ to update the parameters

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{D}_m), \qquad t \in \mathbb{N}_0$$

This update provides a stochastic approximation to the true gradient which is far more efficient to compute, and provides a huge speed up in the training process for large datasets.

Multilayer perceptrons

The simplest type of deep learning model is the **multilayer perceptron**, also known as a **feedforward network**. This type of neural network can be viewed as an architecture consisting of layers of mathematical neurons, linked together in a directed acyclic graph.

MLP with single hidden layer

A key property of deep learning models is the fact that they are *compositional* instead of *additive*. Where as linear regression models (or logistic regression, kernel regression) increase complexity by adding extra basis functions ϕ_i in the expansion

$$f(\mathbf{x}) = \sum_i w_i \phi_i(\mathbf{x}),$$

deep learning models increase complexity by composing multiple simple functions φ_k together:

$$f(\mathbf{x}) = \varphi_L(\varphi_{L-1}(\ldots \varphi_2(\varphi_1(\mathbf{x}))\ldots)).$$

The functions φ_k are defined to be affine transformations followed by an element-wise activation function. An example is the MLP with a single hidden layer:

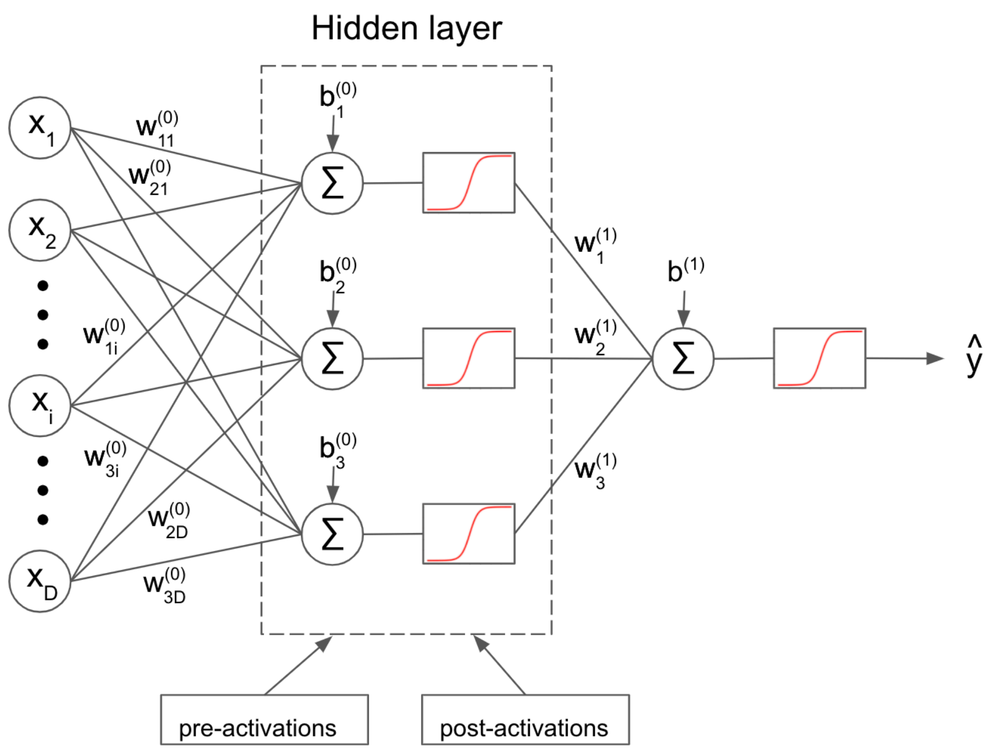
$$h_j^{(1)} = \sigma\left(\sum_{i=1}^{D} w_{ji}^{(0)} x_i + b_j^{(0)}\right), \qquad j = 1, \ldots, n_h, \tag{5}$$

$$\hat{y} = \sigma_{out}\left(\sum_{i=1}^{n_h} w_i^{(1)} h_i^{(1)} + b^{(1)}\right). \tag{6}$$

In the above, $\mathbf{x} \in \mathbb{R}^D$ is an example input, $n_h \in \mathbb{N}$ is the number of hidden units in the network, $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ are activation functions, $w_{ji}^{(0)} \in \mathbb{R}$ and $w_{ji}^{(1)} \in \mathbb{R}$ are weights, and $b_j^{(0)} \in \mathbb{R}$ and $b^{(1)} \in \mathbb{R}$ are biases.

Following (5) and (6) we also define the **pre-activations** $a_j^{(1)} := \sum_{i=1}^{D} w_{ji}^{(0)} x_i + b_j^{(0)}$. Correspondingly, the $h_j^{(1)}$ are referred to as the **post-activations** (or frequently, just **activations**).

This construction can be summarised in the following diagram.



Multilayer perceptron with a single hidden layer consisting of three neurons.

We will usually write equations (5) and (6) in the more concise form:

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right), \tag{7}$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(1)}\mathbf{h}^{(1)} + b^{(1)}\right), \tag{8}$$

where $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{W}^{(0)} \in \mathbb{R}^{n_h \times D}$, $\mathbf{b}^{(0)} \in \mathbb{R}^{n_h}$, $\mathbf{h}^{(1)} \in \mathbb{R}^{n_h}$, $\mathbf{w}^{(1)} \in \mathbb{R}^{1 \times n_h}$, $b^{(1)} \in \mathbb{R}$ and we overload notation with the activation functions $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ by applying them element-wise in the above.

This hidden layer is a type of neural network layer that is often referred to as a **dense** or **fully connected** layer.

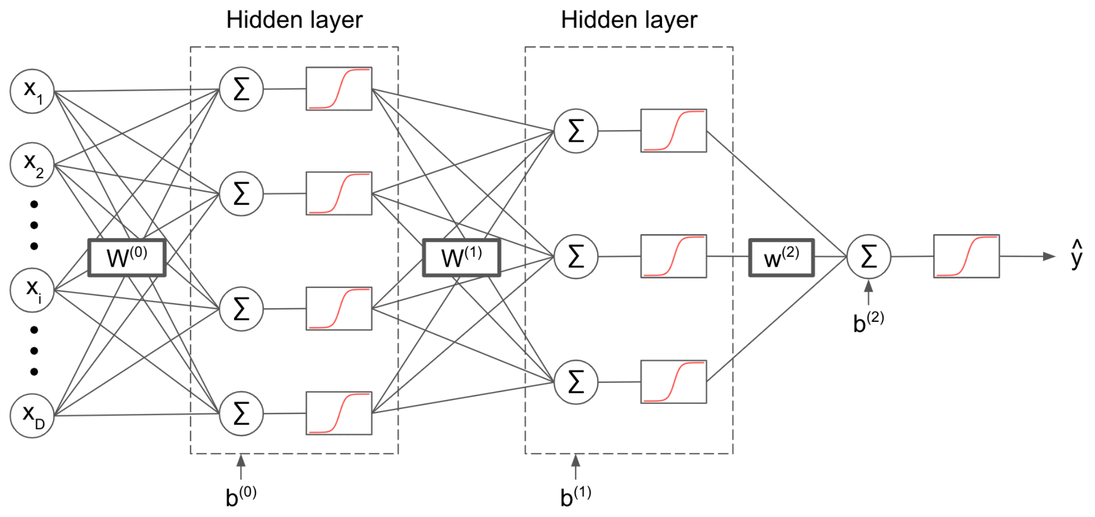
MLP with multiple hidden layers

More generally, for an MLP with L hidden layers, we have

$$\begin{aligned} \mathbf{h}^{(0)} &:= \mathbf{x}, && (9)\\ \mathbf{h}^{(k)} &= \sigma\left(\mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}\right), \qquad k = 1, \ldots, L, && (10)\\ \hat{y} &= \sigma_{out}\left(\mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}\right), && (11) \end{aligned}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1}\times n_k}$, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_{k+1}}$, $\mathbf{h}^{(k)} \in \mathbb{R}^{n_k}$, and we have set $n_0 := D$, and n_k is the number of units in the k-th hidden layer.

The following diagram shows the MLP architecture with two hidden layers.



Multilayer perceptron with a two hidden layers.

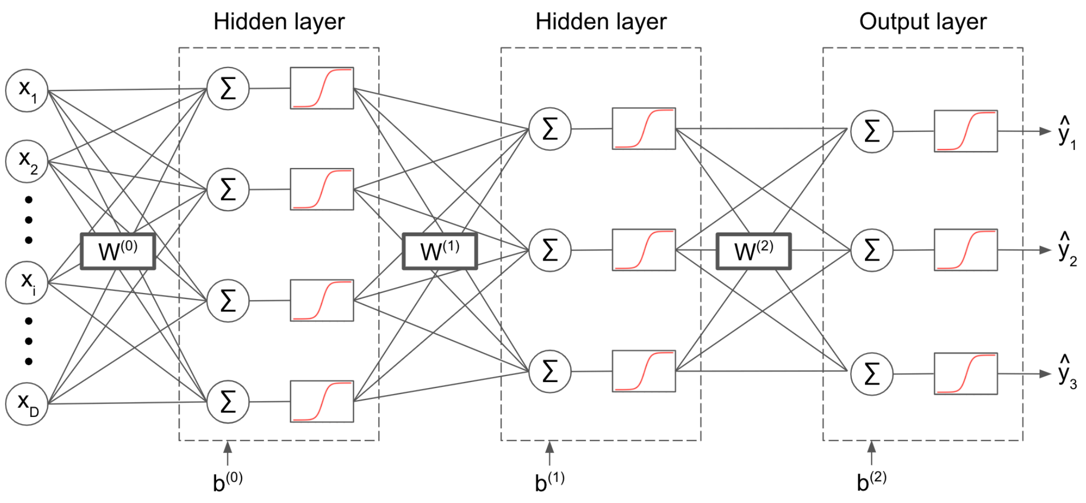
The hidden layers inside a deep network can be viewed as *learned feature extractors*. The weights of the network learn to encode the data in such a way as to represent progressively more complex or abstract features of the data that are useful for solving the problem task at hand. This hierarchy of representations is a core property of the expressive power of deep learning models (Rumelhart et al 1986b).

Output layers

One of the strengths of deep learning models is their applicability to a wide range of dataset types and problem tasks. In equation (11) we have considered a single unit output y, which is produced by passing the pre-activation $a^{(L+1)} := \mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}$ through the activation function σ_{out}.

Note how linear regression and logistic regression can both be viewed as a neural network without a hidden layer. In this case, if σ_{out} is be the identity (or linear) activation, then we are left with a simple linear regression model. Likewise, if σ_{out} is the sigmoid function, then we have the logistic regression model.

The architecture can also be easily modified to output multiple target variables $\hat{\mathbf{y}}$ by replacing (11) with $\hat{\mathbf{y}} = \sigma_{out}\left(\mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}\right)$:



Multilayer perceptron with multiple outputs.

Moreover, the activation functions in the output layer can be chosen according to the requirements of the target variables. For example, if the network should output an estimate for a standard deviation parameter, then we will want to constrain the output to be positive. This can be achieved by passing the pre-activation through a softplus or exponential activation function, for example. It is common for a sigmoid activation to be used where the output should be interpreted as a probability (as in logistic regression). More generally, for target variables that should be constrained to an interval, then a sigmoid or tanh activation can be used followed by a suitable rescaling. Different activation functions could be applied to different units in the output layer, if appropriate.

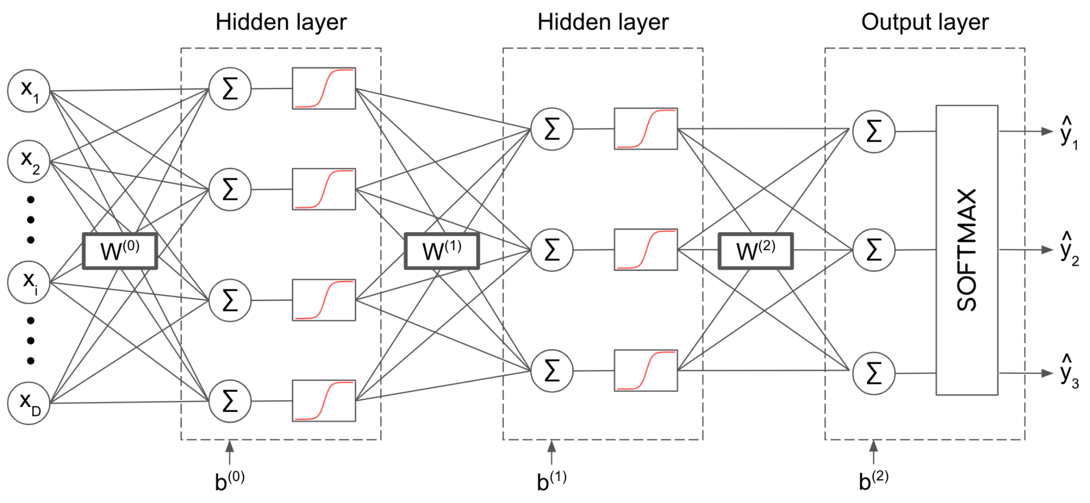
Another common output layer is the **softmax**, which is used for multiclass classification models. The softmax layer outputs a normalised array, which can be interpreted as a probability vector specifying a categorical distribution. For pre-activations

$$\mathbf{a}^{(L+1)} := \mathbf{W}^{(L)}\mathbf{h}^{(L)} + \mathbf{b}^{(L)}$$

with $\mathbf{W}^{(L)} \in \mathbb{R}^{C\times n_L}$, $\mathbf{b}^{(L)} \in \mathbb{R}^C$ where C is the number of classes, the softmax function is given by

$$\hat{\mathbf{y}}_j := \mathrm{softmax}(\mathbf{a}^{(L+1)})_j = \frac{\exp(a_j)}{\sum_i \exp(a_i)}.$$

Note that the softmax function operates on all pre-activations in the output layer, in contrast to the usual element-wise application of most activation functions.



Multilayer perceptron with a softmax output layer.

The Sequential class

There are multiple ways to build and apply deep learning models in Keras, from high-level, quick and easy-to-use APIs, to low-level operations. In this section you will walk through using the Keras API for quickly building, training, evaluating and predicting from deep learning models. In particular, you will see how to use the `Sequential` class to implement MLP models.

In [17]:
```
import keras
from keras import ops
```

The `Dense` layer

We will see how to build MLP models using the `Dense` layer class from Keras. This class implements the layer transformation $\mathbf{h}^{(k+1)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{h}^{(k)} + \mathbf{b}^{(k)}\right)$.

In [18]:
```
# Create a Dense layer

from keras.layers import Dense

dense_layer = Dense(4, activation='sigmoid')
```

In [19]:
```
# Inspect the layer parameters

dense_layer.variables
```

Out[19]:
```
[]
```

The `Dense` layer above does not yet have any weights because it has not yet been built. A Keras layer is typically built only when it is first called on an input, and only then are the weights created. This mechanism means that to create a `Dense` layer you only need to specify the number of output units, and not the number of input units.

We will build the layer (and create the weights) by calling it on a dummy input. Keras models and layers are designed to process batches of data at once, and always expect inputs to have a batch dimension in the first axis. For example, a batch of 16 inputs, each of which is a length 4 vector, should have a shape `[16, 4]`.

In [20]:
```
# Call the dense layer on an input to create the weights

x = keras.random.uniform((2, 3))
y = dense_layer(x)
y
```

Out[20]:
```
<tf.Tensor: shape=(2, 4), dtype=float32, numpy=
array([[0.54627895, 0.39618725, 0.39288798, 0.42004484],
       [0.5433938 , 0.33856195, 0.36310792, 0.35392445]], dtype=float32)
>
```

In [21]:
```
# Inspect the layer parameters

dense_layer.variables
```

Out[21]:
```
[<Variable path=dense/kernel, shape=(3, 4), dtype=float32, value=[[-0.54
54782  -0.3512184   0.4467895  -0.91423696]
  [ 0.2217362  -0.8057442  -0.8703502  -0.3159535 ]
  [ 0.82019556  0.90672517  0.38929796 -0.08354867]]>,
 <Variable path=dense/bias, shape=(4,), dtype=float32, value=[0. 0. 0.
0.]>]
```

The base `Layer` class

The `Dense` class inherits from the base `Layer` class, which is core to the Keras API. `Layer` objects encapsulate a state (the layer parameters) and the computation that is carried out by the layer.

An example implementation of an affine transformation layer is shown below as a subclass of `Layer`.

In [22]:
```
# Create an Affine layer class

from keras.layers import Layer

class Affine(Layer):

    def __init__(self, units, input_dim, name='affine'):
        super().__init__(name=name)
        self.w = self.add_weight(
            shape=(input_dim, units),
            initializer="random_normal",
            trainable=True,
            name="kernel"
        )
        self.b = self.add_weight(
            shape=(units,),
            initializer='zeros',
            trainable=True,
            name='bias'
        )
        self.counter = self.add_weight(
            shape=(),
            initializer='zeros',
            trainable=False,
            dtype='int',
            name='counter'
        )

    def call(self, inputs):
        self.counter.assign_add(1)
        # Recall inputs has shape (batch_size, input_dim)
        return ops.matmul(inputs, self.w) + self.b
```

Layer variables should be added as above using the `add_weight` method to ensure they are properly tracked by the layer. Variables with `trainable` set to `False` will not be modified during training. By default it is set to `True`.

In [23]:
```
# Create an instance of the Affine layer class

affine = Affine(3, 2)
affine.weights
```

Out[23]:
```
[<Variable path=affine/kernel, shape=(2, 3), dtype=float32, value=[[-0.0
5398561  0.01093747 -0.01071178]
  [ 0.01399361 -0.02729112  0.04422139]]>,
 <Variable path=affine/bias, shape=(3,), dtype=float32, value=[0. 0. 0.]
>,
 <Variable path=affine/counter, shape=(), dtype=int64, value=0>]
```

In [24]:
```
# Call the affine layer on an input

affine(ops.ones([1, 2]))
```

Out[24]:
```
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=array([[-0.039992  , -0.0
1635365,  0.03350962]], dtype=float32)>
```

In [25]:
```
# Layer objects also have a trainable attribute

affine.trainable
```

Out[25]:
```
True
```

In [26]:
```
# trainable_variables attribute

affine.trainable_variables
```

Out[26]:
```
[<Variable path=affine/kernel, shape=(2, 3), dtype=float32, value=[[-0.0
5398561  0.01093747 -0.01071178]
  [ 0.01399361 -0.02729112  0.04422139]]>,
 <Variable path=affine/bias, shape=(3,), dtype=float32, value=[0. 0. 0.]
>]
```

In [27]:
```
# variables attribute

affine.variables
```

Out[27]:
```
[<Variable path=affine/kernel, shape=(2, 3), dtype=float32, value=[[-0.0
5398561  0.01093747 -0.01071178]
  [ 0.01399361 -0.02729112  0.04422139]]>,
 <Variable path=affine/bias, shape=(3,), dtype=float32, value=[0. 0. 0.]
>,
 <Variable path=affine/counter, shape=(), dtype=int64, value=1>]
```

In [28]:
```python
# Inspect counter

affine.counter.value
```

Out[28]:
```
<tf.Variable 'affine/counter:0' shape=() dtype=int64, numpy=1>
```

Note that for our `Affine` layer class we need to supply both input and output dimensions, but for the `Dense` layer class the input dimension is inferred when the layer is first called.

It is good practice to defer creation of the weights using the `build` method:

In [29]:
```python
# Create an Affine layer class that defers creation of the weights

class NewAffine(Layer):

    def __init__(self, units, name='affine'):
        super().__init__(name=name)
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
            name="kernel"
        )
        self.b = self.add_weight(
            shape=(self.units,),
            initializer='zeros',
            trainable=True,
            name='bias'
        )

    def call(self, inputs):
        # Recall inputs has shape (batch_size, input_dim)
        return ops.matmul(inputs, self.w) + self.b
```

In [30]:
```python
# Create an instance of the Affine layer class

new_affine = NewAffine(3)
```

In [31]:
```python
# The trainable weights are not created yet

new_affine.weights
```

Out[31]:
```
[]
```

In [32]:
```python
# Call the affine layer on an input
```

```
new_affine(ops.ones([1, 2]))
```

Out[32]:
```
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=array([[0.10020073, 0.086
34429, 0.05016537]], dtype=float32)>
```

In [33]:
```
# Inspect the weights

new_affine.weights
```

Out[33]:
```
[<Variable path=affine/kernel, shape=(2, 3), dtype=float32, value=[[ 0.0
5164189  0.03492694 -0.04321422]
  [ 0.04855883  0.05141735  0.09337959]]>,
 <Variable path=affine/bias, shape=(3,), dtype=float32, value=[0. 0. 0.]
>]
```

Other layers you will find in the `keras.layers` module subclass the base `Layer` class in a similar way.

Exercise. In the cells below, write and test a subclassed layer that reflects each input $\mathbf{x} \in \mathbb{R}^d$ in the hyperplane $\{\mathbf{v} \mid \mathbf{n} \cdot \mathbf{v} = 0\}$, where $\mathbf{n} \in \mathbb{R}^d$ is a trainable Variable. Your layer should randomly initialise $\mathbf{n}$ in the `build` method.

In [ ]:
```
# Create a reflection layer

class Reflection(Layer):
    pass
```

In [ ]:
```
# Create an instance of the Reflection layer
```

In [ ]:
```
# Inspect the layer weights - they should only be initialised once the la
```

To test your layer, create a batch of dummy inputs `x` and pass them through your layer to obtain the Tensor `y`. Print out the shape of `y`. Is it what you'd expect? Does your layer reflect the inputs in the hyperplane $\{\mathbf{v} \mid \mathbf{n} \cdot \mathbf{v} = 0\}$ as intended?

In [ ]:
```
# Test the layer on some inputs
```

MLP models

To construct an MLP model, we stack multiple `Dense` layers together by passing them in a list to the `Sequential` API:

In [34]:
```
# Build an MLP model

from keras.models import Sequential
from keras.layers import Dense

mlp = Sequential([
    Dense(4, activation='relu'),
    Dense(4, activation='relu'),
    Dense(3)
])
```

The default value for the `activation` keyword argument is `None`, in which case no activation (linear activation) is applied.

In [35]:
```
# Call the model on an input to create the weights

x = keras.random.normal((2, 6))
y = mlp(x)
y
```

Out[35]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 0.17716321, -0.3059078 , -0.5900371 ],
       [-0.60204136, -0.08998922,  0.19682413]], dtype=float32)>
```

It is worth knowing that the `Sequential` class itself inherits from the `Layer` class, so all the same properties and methods are also available for `Sequential` models.

In [36]:
```
# Inspect the model parameters

mlp.weights
```

Out[36]:
```
[<Variable path=sequential/dense_1/kernel, shape=(6, 4), dtype=float32,
value=[[-0.12999868  0.70716286 -0.17657733 -0.3910728 ]
  [-0.30152196 -0.6241319   0.4182731  -0.4979077 ]
  [ 0.15677887  0.65172136  0.6211966  -0.25529164]
  [ 0.5616859   0.31261396 -0.37253332  0.04135406]
  [-0.23161459  0.5337554   0.48886657  0.48423374]
  [-0.7272962  -0.681459   -0.25453097 -0.6159008 ]]>,
 <Variable path=sequential/dense_1/bias, shape=(4,), dtype=float32, valu
e=[0. 0. 0. 0.]>,
 <Variable path=sequential/dense_2/kernel, shape=(4, 4), dtype=float32,
value=[[-0.688107   -0.23176974 -0.5693469  -0.29667807]
  [-0.16445673 -0.39674157 -0.7614669   0.822509  ]
  [ 0.7637927  -0.83295304  0.5320682   0.3813545 ]
  [ 0.8452869  -0.20229     0.49174124  0.4496041 ]]>,
 <Variable path=sequential/dense_2/bias, shape=(4,), dtype=float32, valu
e=[0. 0. 0. 0.]>,
 <Variable path=sequential/dense_3/kernel, shape=(4, 3), dtype=float32,
value=[[ 0.89017355 -0.80865407 -0.46270233]
  [-0.74235    -0.8153304   0.87501097]
  [-0.5986936   0.7879641  -0.27506608]
  [-0.66966355  0.04602784  0.24557114]]>,
 <Variable path=sequential/dense_3/bias, shape=(3,), dtype=float32, valu
e=[0. 0. 0.]>]
```

Model layers can be retrieved using the `layers` attribute of a Keras model.

In [37]:
```
# Inspect the model layers

print(mlp.layers)
print(mlp.layers[1])
print(mlp.layers[1].kernel)
```

```
[<Dense name=dense_1, built=True>, <Dense name=dense_2, built=True>, <Dens
e name=dense_3, built=True>]
<Dense name=dense_2, built=True>
<Variable path=sequential/dense_2/kernel, shape=(4, 4), dtype=float32, val
ue=[[-0.688107   -0.23176974 -0.5693469  -0.29667807]
 [-0.16445673 -0.39674157 -0.7614669   0.822509  ]
 [ 0.7637927  -0.83295304  0.5320682   0.3813545 ]
 [ 0.8452869  -0.20229     0.49174124  0.4496041 ]]>
```

In [38]:
```
# Print the model summary

mlp.summary()
```

Model: "sequential"

Layer (type)	**Output Shape**	
dense_1 (Dense)	(2, 4)	
dense_2 (Dense)	(2, 4)	
dense_3 (Dense)	(2, 3)	

Total params: 63 (252.00 B)
Trainable params: 63 (252.00 B)
Non-trainable params: 0 (0.00 B)

`Sequential` models (and layers) also have `trainable_weights` and `non_trainable_weights` properties, which are displayed in the model summary.

MNIST MLP

In this section, we will run through a complete example of building, training, evaluating and predicting from an MLP model trained on the MNIST dataset of handwritten digits.

The MNIST dataset can be downloaded using the Keras API.

In [39]:
```
# Load the MNIST dataset

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

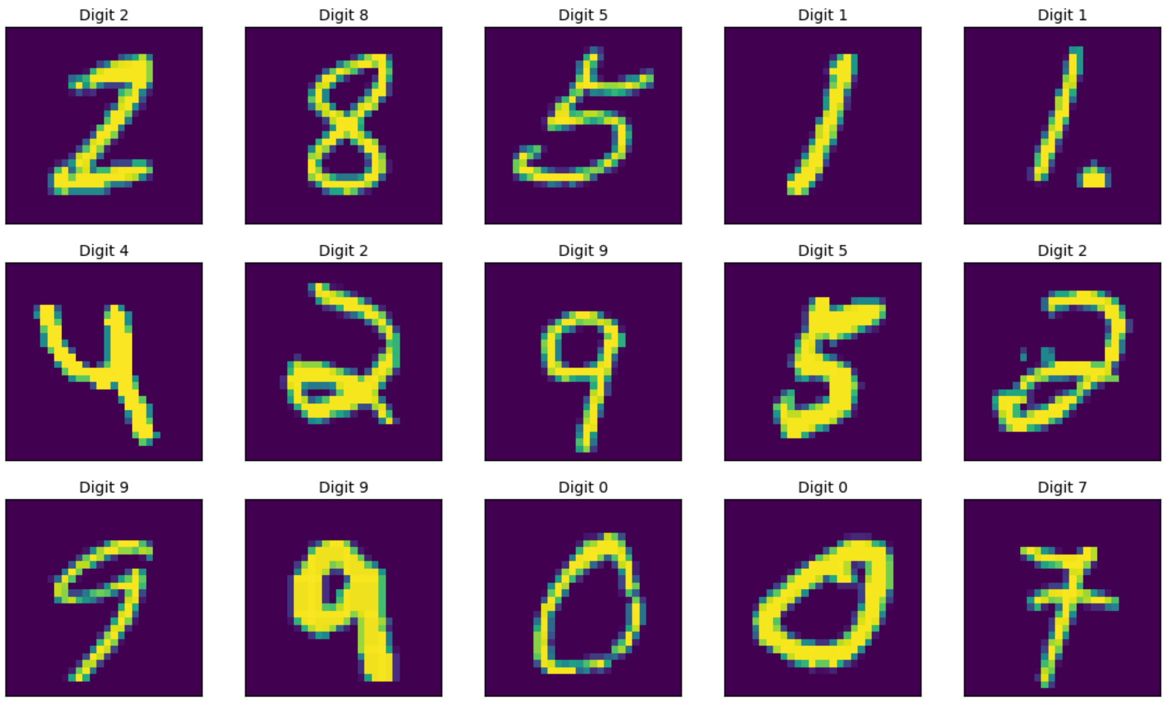
Keras datasets are downloaded to and stored locally at `~/.keras/datasets`. Keras will first check in this location to load the dataset, and download it only if it is not available locally.

Several datasets are available to load using the Keras API, see the docs.

In [40]:
```
# Inspect the data shapes

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

In [41]:
```
# View a few training data examples

import numpy as np
import matplotlib.pyplot as plt

n_rows, n_cols = 3, 5
random_inx = np.random.choice(
    x_train.shape[0], n_rows * n_cols, replace=False)
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
fig.subplots_adjust(hspace=0.2, wspace=0.1)

for n, i in enumerate(random_inx):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(x_train[i])
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
    axes[row, col].text(10., -1.5, f'Digit {y_train[i]}')
plt.show()
```



Multidimensional inputs (i.e., with rank >= 2) can also be processed by an MLP network by simply unrolling, or flattening the dimensions. This can be done easily using the `Flatten` layer.

The cell below also shows how the we can use an `Input` layer to specify the shape of the model inputs (minus the batch size), so that the model will be built straight away (weights created).

In [42]:
```
# Create an MNIST classifier model

from keras.layers import Flatten, Input
```

```
mnist_model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Dense(64, activation='tanh'),
    Dense(64, activation='tanh'),
    Dense(10, activation='softmax')
])
mnist_model.summary()
```

```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━
┃ Layer (type)                       ┃ Output Shape             ┃       
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━
│ flatten (Flatten)                  │ (None, 784)              │       
├────────────────────────────────────┼──────────────────────────┼────────
│ dense_4 (Dense)                    │ (None, 64)               │       
├────────────────────────────────────┼──────────────────────────┼────────
│ dense_5 (Dense)                    │ (None, 64)               │       
├────────────────────────────────────┼──────────────────────────┼────────
│ dense_6 (Dense)                    │ (None, 10)               │       
└────────────────────────────────────┴──────────────────────────┴────────

 Total params: 55,050 (215.04 KB)
 Trainable params: 55,050 (215.04 KB)
 Non-trainable params: 0 (0.00 B)
```

To train the model, we need to specify a loss function to minimise, and an optimisation algorithm. The average negative log-likelihood on the training set is given by the categorical cross entropy

$$L(\theta) = -\frac{1}{|\mathcal{D}_{train}|} \sum_{x_i \in \mathcal{D}_{train}} \sum_{j=1}^{10} \tilde{y}_{ij} \ln f_\theta(x_i)_j,$$

where f_θ is the neural network function (with parameters θ) that outputs a length 10 probability vector $f_\theta(x_i) \in \mathbb{R}^{10}$ for an input example image $x_i \in \mathbb{R}^{28\times 28}$, and $\tilde{y}_{ij}$ is 1 if the correct label for example i is j, and 0 otherwise.

As our labels `y_train` and `y_test` are in sparse form, we use the `sparse_categorical_crossentropy` loss function. We also will use the stochastic gradient descent (SGD) optimiser.

In [43]:
```
# Compile the model

mnist_model.compile(loss='sparse_categorical_crossentropy',
                    optimizer='sgd',metrics=['accuracy'])
```

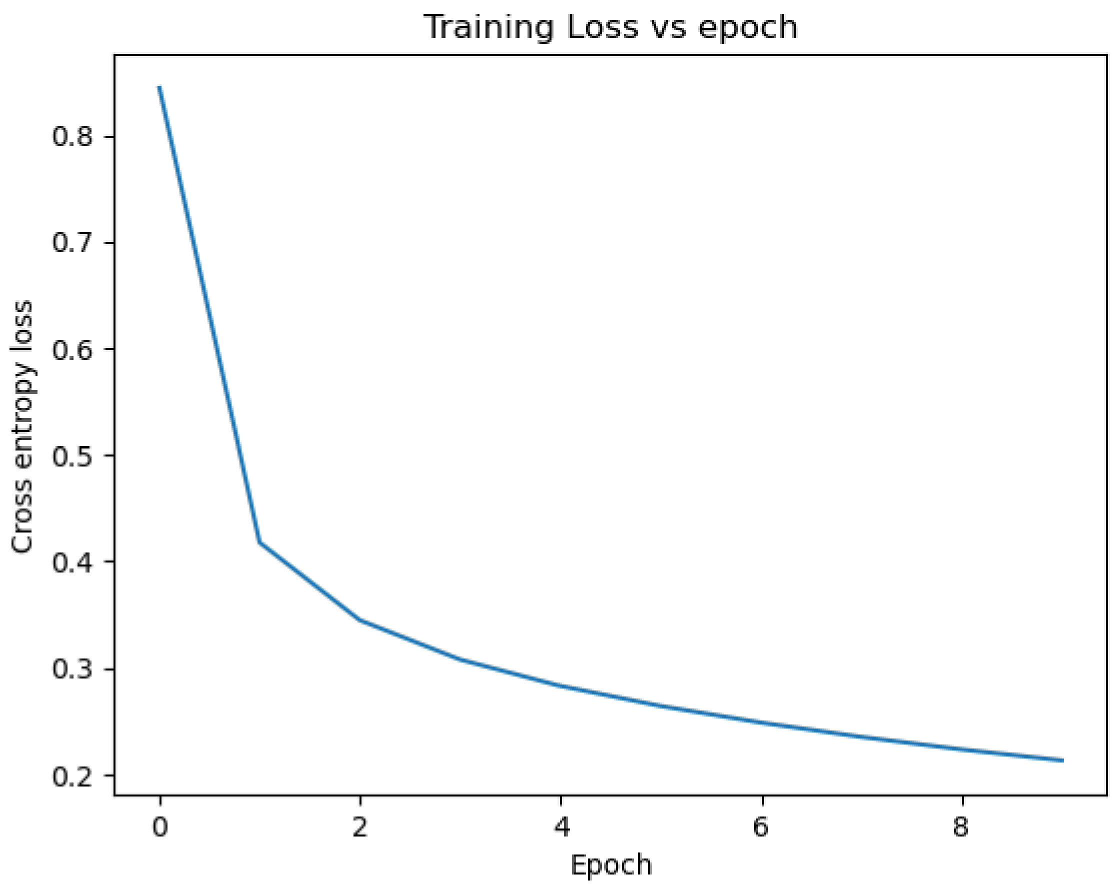
The image data is filled with integer pixel values from 0 to 255. To facilitate the training, we rescale the values to the interval $[0, 1]$.

In [44]:
```
# Rescale the image data

x_train = x_train / 255.
x_test = x_test / 255.
```

```
In [45]: # Train the model

         history = mnist_model.fit(x_train, y_train, epochs=10, batch_size=64)
```

```
Epoch 1/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.6690 - loss: 1.2562
Epoch 2/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8841 - loss: 0.4446
Epoch 3/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9040 - loss: 0.3561
Epoch 4/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9135 - loss: 0.3134
Epoch 5/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9184 - loss: 0.2897
Epoch 6/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9229 - loss: 0.2714
Epoch 7/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9287 - loss: 0.2504
Epoch 8/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9335 - loss: 0.2347
Epoch 9/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9350 - loss: 0.2231
Epoch 10/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9381 - loss: 0.2172
```

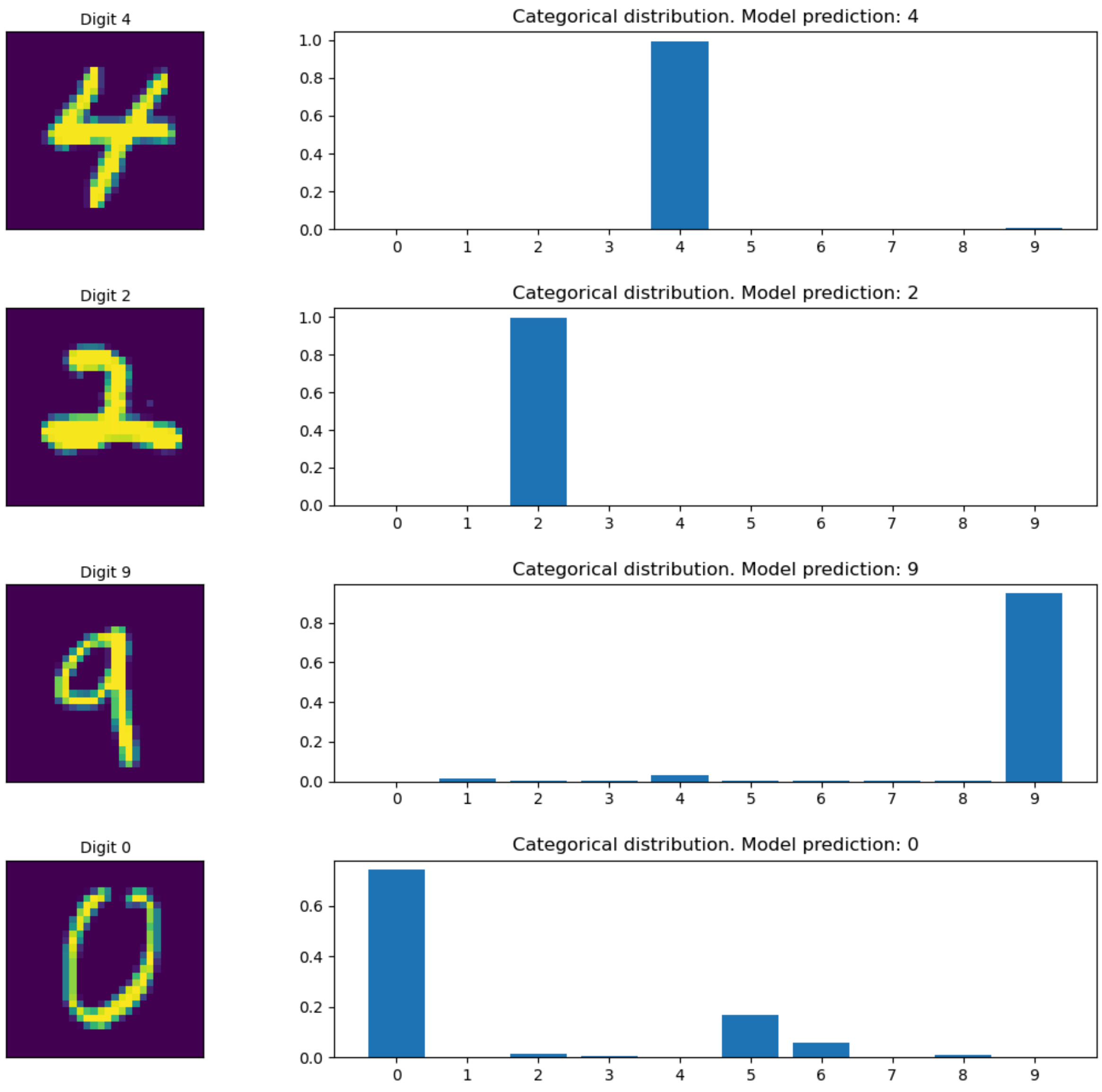
```
In [46]: # Plot the learning curve

         plt.plot(history.history['loss'])
         plt.xlabel("Epoch")
         plt.ylabel("Cross entropy loss")
         plt.title("Training Loss vs epoch")
         plt.show()
```



```
In [47]: # Evaluate the model on the test set

         mnist_model.evaluate(x_test, y_test)

         313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9297 - loss: 0.2370

Out[47]: [0.2062128186225891, 0.9412999749183655]
```

```
In [48]: # Get predictions from model

         preds = mnist_model.predict(x_test)
         preds.shape

         313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 864us/step

Out[48]: (10000, 10)
```

```
In [49]: # Plot some predicted categorical distributions

         num_test_images = x_test.shape[0]

         random_inx = np.random.choice(num_test_images, 4)
         random_preds = preds[random_inx, ...]
         random_test_images = x_test[random_inx, ...]
         random_test_labels = y_test[random_inx, ...]

         fig, axes = plt.subplots(4, 2, figsize=(16, 12))
         fig.subplots_adjust(hspace=0.4, wspace=-0.2)

         for i, (prediction, image, label) in enumerate(zip(
             random_preds, random_test_images, random_test_labels)):
             axes[i, 0].imshow(np.squeeze(image))
             axes[i, 0].get_xaxis().set_visible(False)
             axes[i, 0].get_yaxis().set_visible(False)
             axes[i, 0].text(10., -1.5, f'Digit {label}')
             axes[i, 1].bar(np.arange(len(prediction)), prediction)
             axes[i, 1].set_xticks(np.arange(len(prediction)))
             title = "Categorical distribution"
             title += f"Model prediction: {np.argmax(prediction)}"
             axes[i, 1].set_title(title)
         plt.show()
```



Exercise. The MNIST dataset is an easy dataset, and the above model is far from optimal. Try experimenting with longer training times and/or model architecture changes to see if you can improve on the performance.

References

- Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016), "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", in *4th International Conference on Learning Representations, {ICLR} 2016", San Juan, Puerto Rico, May 2-4, 2016.
- Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017), "Self-Normalizing Neural Networks", *Neural Information Processing Systems (NIPS)*, 971-980.
- McCulloch, W. & Pitts, W. (1943), "A Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, **5**, 127-147.
- Ramachandran, P., Zoph, B. & Le, Q. V. (2018) "Searching for Activation Functions", arXiv preprint, abs/1710.05941.
- Robbins, H. and Monro, S. (1951), "A stochastic approximation method", *The annals of mathematical statistics*, 400–407.

- Rosenblatt, F. (1958), "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", Psychological Review, 65-386.
- Rosenblatt, F. (1961), "Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms", Defense Technical Information Center.
- Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (1986a), "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", MIT Press, Cambridge.
- Rumelhart, D. E., Hinton, G., & Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.