Deep Learning

Week 3: Loss functions and backpropagation

Contents

Introduction

In the last week of the module we took a first look at the prototypical deep learning architecture, which is the multilayer perceptron. You also trained your first deep learning models in Keras, using the Sequential API, and learned the core methods `compile`, `fit`, `evaluate` and `predict`. You saw how the low level Tensor objects are included in these models to encapsulate mutable parameters and computational operations.

In this week of the course, we will look closer at the issue of training neural networks. We will begin by walking through some commonly used loss functions and the principle of maximum likelihood, and study the important backpropagation algorithm, with a focus on its application to MLP models. We will also look at two commonly used techniques used in deep learning models, which are dropout and batch normalization.

We will also learn how to implement all of these techniques in TensorFlow and PyTorch, and see how gradients can easily be computed using automatic differentiation tools in both of these frameworks. One of the major advantages of using deep learning frameworks like TensorFlow or PyTorch is the ability to automatically compute gradients of any differentiable operation. For most use cases, Keras will handle the gradient computation and parameter updates behind the scenes, but it will be useful later to understand how automatic differentiation is performed at a lower level, particularly when implementing custom training algorithms.

Loss functions

The loss function is the performance measure which is used to train neural networks. In order to enable gradient-based training, this loss function should be differentiable with respect to the model parameters.

Given a task and some data, we need a way of formulating a suitable loss function to test our model. The principle that's often used to do this is maximum likelihood.

The likelihood function

A probability density function (or probability mass function) $P(y \mid \theta)$ is usually viewed as a function of a sample y, with the parameter(s) θ fixed. In contrast, the likelihood function $\mathcal{L}(\theta \mid y)$ considers y to be fixed and is viewed as a function of θ:

$$\mathcal{L}(\theta \mid y) := P(y \mid \theta).$$

For independent and identically distributed (i.i.d.) samples $y_1, \ldots, y_N$, the probability density/mass function (and likelihood function) decomposes as the product

$$\mathcal{L}(\theta \mid y_1, \ldots, y_N) = P(y_1, \ldots, y_N \mid \theta) = \prod_{i=1}^{N} P(y_i \mid \theta).$$

An example is the Bernoulli distribution with parameter θ. This is the distribution of a random variable that takes value 1 with probability θ and 0 with probability $1 - \theta$. Let $P(y \mid \theta)$ be the probability that the event y occurs given parameter θ. Then

$$\begin{aligned} \mathcal{L}(\theta \mid y) = P(y \mid \theta) &= \begin{cases} 1 - \theta & \text{if } y = 0 \\ \theta & \text{if } y = 1 \end{cases} && (1) \\ &= (1 - \theta)^{1-y} \theta^{y} \quad y \in \{0, 1\} && (2) \end{aligned}$$

If we assume samples $y_1, \ldots, y_N$ are i.i.d., we also have

$$\mathcal{L}(\theta \mid y_1, \ldots, y_N) = \prod_{i=1}^{N} (1 - \theta)^{1-y_i} \theta^{y_i}.$$

Another example is the Normal distribution. This distribution has two parameters: a mean μ and a standard deviation σ, so then $\theta = (\mu, \sigma)$. The probability density function is:

$$\mathcal{L}(\theta \mid y) = P(y|\theta) = P(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{1}{2\sigma^2}(y - \mu)^2\Big).$$

For a sequence of i.i.d. observations $y_1, \ldots, y_N$, the likelihood is

$$\mathcal{L}(\mu, \sigma \mid y_1, \ldots, y_N) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{1}{2\sigma^2}(y_i - \mu)^2\Big).$$

Maximum likelihood estimation

The likelihood function is commonly used in statistical inference when we are trying to fit a distribution to some data. Suppose we have observed data $y_1, \ldots, y_N$, assumed to be from some distribution with unknown parameter θ, which we want to estimate. The likelihood is

$$\mathcal{L}(\theta \mid y_1, \ldots, y_N).$$

The *maximum likelihood estimate* θ_{MLE} of the parameter θ is then the value that maximises the likelihood $\mathcal{L}(\theta \mid y_1, \ldots, y_N)$.

Recall that, for i.i.d. observations, the likelihood becomes a product:

$$\mathcal{L}(\theta \mid y_1, \ldots, y_N) = \prod_{i=1}^{N} P(y_i|\theta).$$

Furthermore, since the $\log$ function is a strictly concave function, maximising the likelihood is equivalent to maximising the log-likelihood $\log \mathcal{L}(\theta \mid y_1, \ldots, y_N)$. This changes the product into a sum:

$$\begin{aligned}
\theta_{\text{MLE}} &= \arg\max_{\theta} \mathcal{L}(\theta \mid y_1, \ldots, y_N) && (3)\\
&= \arg\max_{\theta} \log \mathcal{L}(\theta \mid y_1, \ldots, y_N) && (4)\\
&= \arg\max_{\theta} \log \prod_{i=1}^{N} \mathcal{L}(\theta \mid y_i) && (5)\\
&= \arg\max_{\theta} \sum_{i=1}^{N} \log \mathcal{L}(\theta \mid y_i). && (6)
\end{aligned}$$

Conventionally, we usually choose to instead equivalently *minimise* the *negative log-likelihood*:

$$\theta_{\text{MLE}} = \arg\min_{\theta} \left(-\sum_{i=1}^{N} \log \mathcal{L}(\theta \mid y_i) \right).$$

Training neural networks

We can use the principle of maximum likelihood to formulate loss functions that are used to train neural networks. Given some training data $\mathcal{D} := (x_i, y_i)_{i=1}^{N}$ of inputs x_i and outputs y_i, we search for the weights of the neural network that minimise the negative log-likelihood of the data.

For example, suppose we would like to train a neural network binary classifier. Then we have $y_i \in \{0, 1\}$. Denote the neural network model as $f_\theta : \mathbb{R}^D \mapsto [0, 1]$, where θ are the parameters of the network. We can constrain the range of the network by using a sigmoid activation function on a single output neuron. Then we can interpret the output of the network as the probability that a given input x_i has output $y_i = 1$:

$$f_\theta(x_i) = P_\theta(y_i = 1 \mid x_i).$$

Therefore the network output is parameterising a Bernoulli distribution. The negative log-likelihood can be written

$$
\begin{aligned}
-\log \mathcal{L}(\theta \mid \mathcal{D}) &= -\sum_{i=1}^{N} \log \mathcal{L}(\theta \mid y_i, x_i) && (7)\\
&= -\sum_{i=1}^{N} \log \Big\{ (1 - f_\theta(x_i))^{(1-y_i)} f_\theta(x_i)^{y_i} \Big\} && (8)\\
&= -\sum_{i=1}^{N} \left\{ (1-y_i) \log(1 - f_\theta(x_i)) + y_i \log f_\theta(x_i) \right\}, && (9)
\end{aligned}
$$

and we have derived the binary cross entropy loss function to be minimised.

As another example, suppose we would like to train a network as a regression function, so that $y_i \in \mathbb{R}$. Again denote the neural network model as $f_\theta : \mathbb{R}^D \mapsto \mathbb{R}$, where θ are the parameters of the network. This time, we can use no activation function on the output neuron of the network.

We can additionally define our model such that the network $f_\theta(x_i)$ parameterises the mean of a normal distribution, so that

$$
y_i \sim N(f_\theta(x_i), \sigma^2)
$$

for some fixed $\sigma^2 > 0$. In this case, the negative log-likelihood can be written

$$
\begin{aligned}
-\log \mathcal{L}(\theta \mid \mathcal{D}) &= -\sum_{i=1}^{N} \log \mathcal{L}(\theta \mid y_i, x_i) && (10)\\
&= -\sum_{i=1}^{N} \log \left\{ \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma^2} (f_\theta(x_i) - y_i)^2 \right) \right\} && (11)\\
&= \sum_{i=1}^{N} \frac{1}{2\sigma^2} (f_\theta(x_i) - y_i)^2 + \log(\sigma\sqrt{2\pi}). && (12)
\end{aligned}
$$

The term $\log(\sigma\sqrt{2\pi})$ does not depend on the parameters θ that are to be optimised, and so can be ignored for the purpose of finding θ_{MLE}. Then we have derived the mean squared error loss, up to a scaling factor (which also does not affect θ_{MLE} as the minimizer).

Note that importantly, these loss functions are differentiable with respect to the parameters θ, so long as the network function f_θ itself is differentiable. This is important to allow the use of gradient-based optimizers.

Error backpropagation

Gradient-based neural network optimisation can be seen as iterating over the following two main steps:

1. Computation of the (stochastic) gradient of the loss function with respect to the model parameters
2. Use of the computed gradient to update the parameters

We have already seen the parameter update rule according to stochastic gradient descent for a neural network model $f_\theta : \mathbb{R}^D \mapsto Y$, where Y is the target space (e.g. $\mathbb{R}^{n_{L+1}}$ or $[0,1]^{n_{L+1}}$):

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

In the above equation, the minibatch loss $L(\theta_t; \mathcal{D}_m)$ is calculated on a randomly drawn sample of data points from the training set,

$$L(\theta_t; \mathcal{D}_m) = \frac{1}{M} \sum_{x_i, y_i \in \mathcal{D}_m} l(y_i, f_{\theta_t}(x_i)). \tag{2}$$

where $\mathcal{D}_m$ is the randomly sampled minibatch, $M = |\mathcal{D}_m| \ll |\mathcal{D}_{train}|$ is the size of the minibatch, and we will denote $L_i : Y \times Y \mapsto \mathbb{R}$, given by $L_i := l(y_i, f_\theta(x_i))$, as the per-example loss.

The update (1) requires computation of the term $\nabla_t L(\theta_t; \mathcal{D}_m)$ (from here we drop the $\mathcal{D}_m$ in this expresesion for brevity); that is, the gradient of the loss function with respect to all of the model parameters, evaluated at the current parameter settings θ_t.

The computation of the derivatives is done through applying the chain rule of differentiation. The algorithm for computing these derivatives in an efficient manner is known as **backpropagation**, and was popularised for use in neural network optimisation in Rumelhart et al 1986b and Rumelhart et al 1986c, although the technique dates back earlier, see e.g. Werbos which includes Paul Werbos' 1974 dissertation.

In this section, we will derive the important backpropagation algorithm for finding the loss function derivatives for a multilayer perceptron.

First recall the layer transformations in the MLP:

$$\mathbf{h}^{(0)} := \mathbf{x}, \tag{3}$$

$$\mathbf{h}^{(k)} = \sigma\left(\mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}\right), \qquad k = 1, \ldots, L, \tag{4}$$

$$\hat{\mathbf{y}} = \sigma_{out}\left(\mathbf{w}^{(L)}\mathbf{h}^{(L)} + b^{(L)}\right), \tag{5}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_{k+1}}$, $\mathbf{h}^{(k)} \in \mathbb{R}^{n_k}$, $\hat{\mathbf{y}} \in Y$, $\sigma, \sigma_{out} : \mathbb{R} \mapsto \mathbb{R}$ are activation functions that are applied element-wise, $n_0 := D$, and n_k is the number of units in the k-th hidden layer.

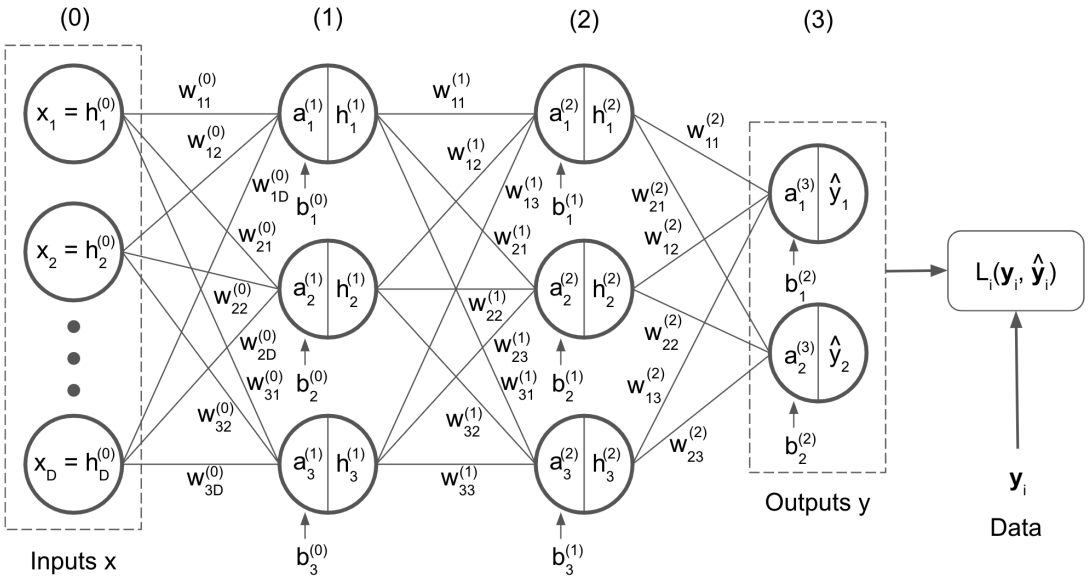
Also recall that we define the **pre-activations**

$$\mathbf{a}^{(k)} = \mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)} \tag{6}$$

and **post-activations**

$$\mathbf{h}^{(k)} = \sigma(\mathbf{a}^{(k)}). \tag{7}$$

We will consider the gradient of the loss computed on a single data example $\nabla_t L_i(\theta_t)$, given the sum (2). We first compute the **forward pass** $(3) - -(5)$ and store the preactivations $\mathbf{a}^{(k)}$ and post-activations $\mathbf{h}^{(k)}$.



Pre-activations, post-activations, weights and biases in the forward pass

Consider the derivative of L_i with respect to $w_{pq}^{(k)}$ and $b_p^{(k)}$. We have:

$$\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \frac{\partial L_i}{\partial a_p^{(k+1)}} \frac{\partial a_p^{(k+1)}}{\partial w_{pq}^{(k)}} \tag{13}$$

$$= \frac{\partial L_i}{\partial a_p^{(k+1)}} h_q^{(k)}, \tag{14}$$

where the second line follows from (6). Similarly,

$$\frac{\partial L_i}{\partial b_p^{(k)}} = \frac{\partial L_i}{\partial a_p^{(k+1)}} \frac{\partial a_p^{(k+1)}}{\partial b_p^{(k)}} \tag{15}$$

$$= \frac{\partial L_i}{\partial a_p^{(k+1)}}. \tag{16}$$

We introduce the notation $\delta_p^{(k)} := \frac{\partial L_i}{\partial a_p^{(k)}}$, called the **error**. We then write

$$\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \delta_p^{(k+1)} h_q^{(k)} \tag{8}$$

$$\frac{\partial L_i}{\partial b_p^{(k)}} = \delta_p^{(k+1)}. \tag{9}$$

We therefore need to compute the quantity $\delta_p^{(k+1)}$ for each hidden and output unit in the network. Again using the chain rule, we have

$$\delta_p^{(k)} \equiv \frac{\partial L_i}{\partial a_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \frac{\partial L_i}{\partial a_j^{(k+1)}} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{17}$$

$$= \sum_{j=1}^{n_{k+1}} \delta_j^{(k+1)} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{10}$$

Combining (6) and (7) we see that

$$a_j^{(k+1)} = \sum_{l=1}^{n_k} w_{jl}^{(k)} \sigma(a_l^{(k)}) + b_p^{(k)} \tag{11}$$

$$\frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} = w_{jp}^{(k)} \sigma'(a_p^{(k)}) \tag{18}$$

where σ' is the derivative of the activation function. So from the above equation and (10) we have

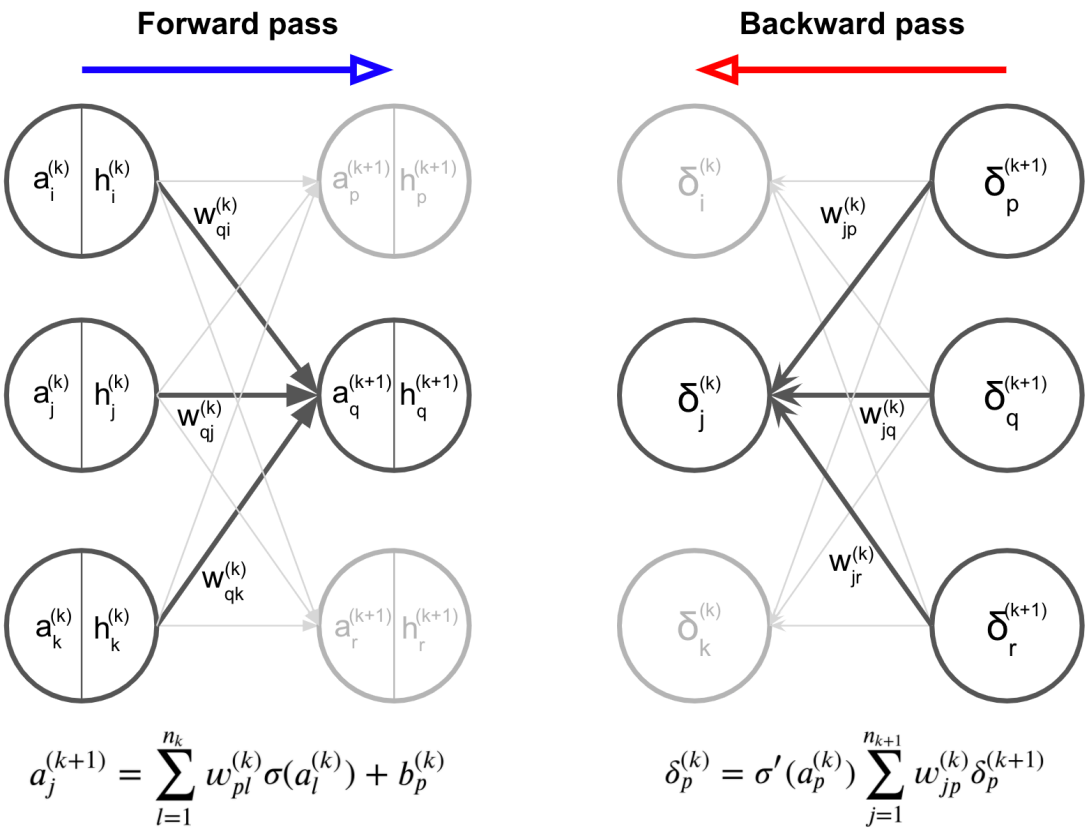
$$\delta_p^{(k)} \equiv \frac{\partial L_i}{\partial a_p^{(k)}} = \sum_{j=1}^{n_{k+1}} \delta_p^{(k+1)} \frac{\partial a_j^{(k+1)}}{\partial a_p^{(k)}} \tag{19}$$

$$= \sigma'(a_p^{(k)}) \sum_{j=1}^{n_{k+1}} w_{jp}^{(k)} \delta_j^{(k+1)} \tag{12}$$

Equation (12) is analogous to (11), and describes the backpropagation of errors through the network. We can write it in the more concise form (analogous to (7)):

$$\delta^{(k)} = \sigma'(\mathbf{a}^{(k)})(\mathbf{W}^{(k)})^T \delta^{(k+1)},$$

where $\sigma'(\mathbf{a}^{(k)}) = \mathrm{diag}([\sigma'(a_p^{(k)})]_{p=1}^{n_k})$.



Forward and backward passes

Now we can summarise the backpropagation algorithm as follows:

> 1. Propagate the signal forwards by passing an input vector x_i through the network and computing all pre-activations and post-activations using $\mathbf{a}^{(k)} = \mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}$
> 2. Evaluate $\delta^{(L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(L+1)}}$ for the output neurons
> 3. Backpropagate the errors to compute $\delta^{(k)}$ for each hidden unit using $\delta^{(k)} = \sigma'(\mathbf{a}^{(k)})(\mathbf{W}^{(k)})^T \delta^{(k+1)}$
> 4. Obtain the derivatives of L_i with respect to the weights and biases using $\frac{\partial L_i}{\partial w_{pq}^{(k)}} = \delta_p^{(k+1)} h_q^{(k)}, \quad \frac{\partial L_i}{\partial b_p^{(k)}} = \delta_p^{(k+1)}$

The backpropagation algorithm can easily be extended to apply to any directed acyclic graph, but we have presented it here in the case of MLPs for simplicity.

Automatic differentiation

One of the major advantages of using deep learning frameworks like TensorFlow or PyTorch is the ability to automatically compute gradients of any differentiable operation. In both frameworks, the process consists of recording the operations that take place in the forward pass, and then in the backward pass the gradients are computed by traversing this sequence of operations in reverse.

When using the Keras `model.fit` API, Keras applies the backpropagation equations automatically to compute gradients, and then uses the optimiser algorithm selected to update the parameters.

In this section, we will see how lower-level tools in TensorFlow and PyTorch can be leveraged to compute gradients of differentiable expressions.

TensorFlow

In [2]:
```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'

import keras
import tensorflow as tf
```

In TensorFlow, operations that you want to take gradients with respect to need to be defined inside a `tf.GradientTape` context. This context can be thought of as setting a tape recording all operations that take place within it.

In [3]:
```
# Define a simple operation and take the gradient

x = tf.constant(3.)

with tf.GradientTape() as tape:
    tape.watch(x)
    y = 4 * tf.square(x) - x
dydx = tape.gradient(y, x)
dydx
```

Out[3]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=23.0>
```

The `tape.watch` line tells TensorFlow to track all operations that apply to the Tensor `x`. A new Tensor `y` is defined as a sequence of operations that include `x`, and so the gradient of `y` with respect to `x` can be computed with `tape.gradient`.

In [4]:
```
# Take multiple derivatives

x = tf.constant([[1.5, 2.], [0., 1.1]])
y = tf.constant([-1., 0.5])

with tf.GradientTape() as tape:
    tape.watch([x, y])
    a = tf.constant([0.3, 2.5])
    z = tf.reduce_mean(x**2, axis=0) * a + y
tape.gradient(z, [x, y])
```

Out[4]:
```
[<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
 array([[0.45000002, 5.        ],
        [0.        , 2.75      ]], dtype=float32)>,
 <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 1.], dtype=floa
t32)>]
```

In [5]:
```
# Gradients can also be computed with respect to intermediate variables

x = tf.random.normal((2, 3))
```

```
with tf.GradientTape() as tape:
    tape.watch(x)
    y = tf.math.sin(x)
    a = tf.random.normal((3, 2))
    z = tf.reduce_mean(tf.linalg.matmul(y, a))
dzdx, dzdy = tape.gradient(z, [x, y])
print(dzdx)
print(dzdy)
```

```
tf.Tensor(
[[-0.01780516 -0.02976127  0.13821083]
 [-0.05971102  0.03874915  0.20262638]], shape=(2, 3), dtype=float32)
tf.Tensor(
[[-0.06604232 -0.23019192  0.28587097]
 [-0.06604232 -0.23019192  0.28587097]], shape=(2, 3), dtype=float32)
```

In [6]:
```
# Gradients can be taken once by default

x = tf.random.normal((2, 3))

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = tf.math.sin(x)
    a = tf.random.normal((3, 2))
    z = tf.reduce_mean(tf.matmul(y, a))
dzdx = tape.gradient(z, x)
dzdy = tape.gradient(z, y)
del tape
print(dzdx)
print(dzdy)
```

```
tf.Tensor(
[[ 0.04144448 -0.44516852  0.649199  ]
 [ 0.18133853 -0.42557085  0.4072824 ]], shape=(2, 3), dtype=float32)
tf.Tensor(
[[ 0.20343632 -0.44544768  0.68451613]
 [ 0.20343632 -0.44544768  0.68451613]], shape=(2, 3), dtype=float32)
```

Note that TensorFlow `Variable` objects are always automatically tracked and so there is no need to call `tape.watch` on them.

In [7]:
```
# Variable objects are tracked automatically

x = tf.Variable(x)

with tf.GradientTape() as tape:
    y = tf.math.sin(x)
    a = tf.random.normal((3, 2))
    z = tf.reduce_mean(tf.matmul(y, a))
tape.gradient(z, x)
```

Out[7]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[-0.05744037,  0.5515819 , -0.06639831],
       [-0.25132784,  0.5272996 , -0.04165574]], dtype=float32)>
```

In [8]:
```
# Take gradients with respect to a layer operation

from keras.layers import Dense
```

```
x = tf.random.normal((1, 4))
dense_layer = Dense(1, activation='elu')

with tf.GradientTape() as tape:
    y = dense_layer(x)
grads = tape.gradient(y, dense_layer.trainable_variables)
print(grads)
```

```
[<tf.Tensor: shape=(4, 1), dtype=float32, numpy=
array([[ 0.07576472],
       [-0.4370297 ],
       [-0.48305142],
       [ 0.4101626 ]], dtype=float32)>, <tf.Tensor: shape=(1,), dtype=floa
t32, numpy=array([0.9912168], dtype=float32)>]
```

In [9]:
```
# Take gradients with respect to a model operation

from keras.models import Sequential

model = Sequential([
    Dense(3, activation='sigmoid'),
    Dense(1, activation='linear')
])

x = tf.random.uniform((2, 2))
y = tf.random.uniform((2, 1))

with tf.GradientTape() as tape:
    loss = tf.keras.losses.mse(y, model(x))
grads = tape.gradient(loss, model.trainable_weights)
grads
```

Out[9]:
```
[<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
 array([[ 0.0259908 , -0.02663141,  0.01809538],
        [ 0.02290116, -0.02338981,  0.01604233]], dtype=float32)>,
 <tf.Tensor: shape=(3,), dtype=float32, numpy=array([ 0.05743254, -0.058
73006,  0.04013848], dtype=float32)>,
 <tf.Tensor: shape=(3, 1), dtype=float32, numpy=
 array([[0.15758316],
        [0.10420758],
        [0.09531184]], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.2445302], dtype=f
loat32)>]
```

For more information on automatic differentiation in TensorFlow, see the guide here.

Exercise. Use `tf.GradientTape` to make a plot of the function $\frac{dy}{dx} : [-5, 5] \mapsto \mathbb{R}$, where $y = \sin(x^2) - \frac{x^2}{4}$.

PyTorch

In [1]:
```
import os
os.environ['KERAS_BACKEND'] = 'torch'

import keras
import torch
```

When defining the computational graph, PyTorch will record all operations that apply to Tensors that are created with the `requires_grad` property set to `True`. `requires_grad` is contagious. This means whenever we use a Tensor with `requires_grad=True` in an operation, the resulting Tensor will also have `requires_grad` set to `True`. By default, newly created Tensors will have `requires_grad` set to `False`, unless they are created as a `torch.nn.Parameter`. (NB: an existing Tensor which has `requires_grad=False` can have this property set to `True` using the `requires_grad_()` method.)

Each Tensor has an attribute called `grad_fn`, this defines the equation necessary to calculate the gradient. If `requires_grad` is set to `False`, `grad_fn` would be `None`.

The `backward` function computes the gradient of the current Tensor with respect to the graph leaves, and stores the result in the respective Tensors' `.grad` attribute. What is a graph leaf? Leaf nodes are the values from which the computation begins. When a Tensor is created and `requires_grad` is set to True, that will be a leaf node. Whenever a computation is executed on it, the resulting Tensor will not be a leaf anymore. Typically, in our deep learning models, model parameters are leaf nodes. To check if a Tensor is a leaf, simply check its `is_leaf` attribute.

In [2]:
```
# Create leaf and non-leaf Tensors

a = torch.rand(10, requires_grad=True)
print(a.is_leaf)

b = torch.rand(10, requires_grad=True) + 4
print(b.is_leaf)
```

```
True
False
```

In [3]:
```
# Define a simple operation and take the gradient

x = torch.tensor(3., requires_grad=True)
print(x.grad)  # Initially set to None

y = 4 * torch.square(x) - x
print(y.requires_grad)  # True, requires_grad is contagious
```

```
None
True
```

In [4]:
```
# We now compute the gradients using backward

dydx = y.backward()
print(x.grad)
```

```
tensor(23.)
```

`backward()` can be called with no arguments only with scalar Tensors. If `backward` is called on a non-scalar Tensor `x` then you need to pass in a Tensor `gradient` of the same shape as `x` - you can think of the Tensor `gradient` as defining the gradient of a scalar valued function with respect to `x`. The resulting stored gradients `.grad` in

the leaf Tensors are then the gradients of the scalar valued function with respect to those leaf Tensors.

In [5]:
```
# Call backward on a non-scalar Tensor

x = torch.tensor([[1.5, 2.], [0., 1.1]], requires_grad=True)
y = torch.tensor([-1., 0.5], requires_grad=False)
y.requires_grad_()
# Can also manually set requires_grad after the Tensor is created

a = torch.tensor([0.3, 2.5])
z = torch.mean(x**2, dim=0) * a + y

z.backward(torch.FloatTensor([1., 1.]))
print(x.grad)
print(y.grad)
```

```
tensor([[0.4500, 5.0000],
        [0.0000, 2.7500]])
tensor([1., 1.])
```

As mentioned before, PyTorch only stores gradients in leaf Tensors. To force it to store the gradient value, call `.retain_grad()` on the Tensor.

In [6]:
```
# Gradients can also be computed with respect to intermediate variables

x = torch.normal(mean=0, std=1, size=(2, 3), requires_grad=True)

y = torch.sin(x)
y.retain_grad()
a = torch.normal(mean=0, std=1, size=(3, 2))
z = torch.mean(torch.matmul(y, a))
z.backward()
print(x.grad)
print(y.grad)
```

```
tensor([[-0.3191, -0.5084,  0.4084],
        [-0.7288, -0.5416,  0.2770]])
tensor([[-0.7506, -0.7898,  0.5112],
        [-0.7506, -0.7898,  0.5112]])
```

In [7]:
```
# Take gradients with respect to a layer operation

from keras.layers import Dense

x = torch.normal(mean=0, std=1, size=(1, 4))
dense_layer = Dense(1, activation='elu')
y = dense_layer(x)
y.backward()

for param in dense_layer.trainable_variables:
    print(param.name, param.value.grad)
```

```
kernel tensor([[-1.5054],
        [-0.2548],
        [-0.9080],
        [-3.0942]], device='cuda:0')
bias tensor([1.], device='cuda:0')
```

One subtlety in the above is that since the `Dense` layer is a Keras object, the parameters of the layer are Keras objects that wrap the underlying torch parameters. The torch parameters can be accessed using the `value` attribute as above.

In [8]:
```
# Take gradients with respect to a model operation

from keras.models import Sequential
from keras import ops

model = Sequential([
    Dense(3, activation='sigmoid'),
    Dense(1, activation='linear')
])

x = torch.rand((2, 2))
y = torch.rand((2, 1))

loss = ops.mean(keras.losses.mean_squared_error(y, model(x)))
loss.backward()

for param in model.trainable_variables:
    print(param.name, param.value.grad)
```

```
kernel tensor([[-0.3107,  0.0475,  0.2592],
        [-0.1288,  0.0197,  0.1084]], device='cuda:0')
bias tensor([-0.4110,  0.0628,  0.3403], device='cuda:0')
kernel tensor([[-0.6590],
        [-0.6294],
        [-0.6608]], device='cuda:0')
bias tensor([-1.5805], device='cuda:0')
```

For more information on automatic differentiation in PyTorch, see the guide here.

Exercise. Use `backward()` to make a plot of the function $\frac{dy}{dx} : [-5, 5] \mapsto \mathbb{R}$, where $y = \sin(x^2) - \frac{x^2}{4}$.

Datasets

TensorFlow and PyTorch both have tools for working with Datasets.

In TensorFlow, the `tf.data` module can be used to load the training data into a `tf.data.Dataset` object, which is equipped with efficient data processing pipelines, and often improves overall performance. See here for a guide on the use of these Dataset objects.

In PyTorch, the `torch.utils.data` module can be used to store the training data into a `torch.utils.data.Dataset` object, and iterate through the dataset using a `torch.utils.data.DataLoader` object. This pipeline is further equipped with data processing functionality. See here for a guide on the use of Dataset and DataLoader objects.

One of the nice features of Keras is that either of these two data pipelines can be used to train models, regardless of the backend selected.

In this section, we will get a working knowledge of TensorFlow Datasets and PyTorch Datasets and DataLoaders. In both cases we will use the Fashion-MNIST dataset as a demonstration.

In [2]:
```
import keras
```

TensorFlow

In TensorFlow, we will use the `tf.data` module, and the `tf.data.Dataset` class.

In [3]:
```
import tensorflow as tf
```

In [4]:
```
# Load the Fashion-MNIST dataset

(x_train, y_train), _ = keras.datasets.fashion_mnist.load_data()
```

`from_tensor_slices`

If data is stored in NumPy arrays as above, then it is easy to create a `tf.data.Dataset` object using the `from_tensor_slices` method. If we pass in a tuple of arrays as below, then the Dataset will correspondingly return tuples of Tensors. The first dimension of the arrays needs to be the same, as this is assumed to index the elements in the dataset.

In [6]:
```
# Load the data into a tf.data.Dataset

tf_train_dataset = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train))
tf_train_dataset.element_spec
```

Out[6]:
```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

In [7]:
```
# Iterate over the Dataset object

for inputs, labels in tf_train_dataset.take(2):
    print(type(inputs))
    print(type(labels))
    print(inputs.shape)
    print(labels.shape)
```

```
<class 'tensorflow.python.framework.ops.EagerTensor'>
<class 'tensorflow.python.framework.ops.EagerTensor'>
(28, 28)
()
<class 'tensorflow.python.framework.ops.EagerTensor'>
<class 'tensorflow.python.framework.ops.EagerTensor'>
(28, 28)
()
```

`map` and `filter`

`Dataset` objects come with `map` and `filter` methods for data preprocessing on the fly. For example, we can normalise the pixel values to the range $[0, 1]$ with the `map` method:

In [8]:
```
# Normalise the pixel values

def normalise_pixels(image, label):
    return (tf.cast(image, tf.float32) / 255., label)

tf_train_dataset = tf_train_dataset.map(normalise_pixels)
tf_train_dataset.element_spec
```

Out[8]:
```
(TensorSpec(shape=(28, 28), dtype=tf.float32, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

We could also filter out data examples according to some criterion with the `filter` method. For example, if we wanted to exclude all data examples with label 9 from the training:

In [9]:
```
# Filter out all examples with label 9 (ankle boot)

tf_train_dataset = tf_train_dataset.filter(
    lambda x, y: tf.math.logical_not(tf.equal(y, 9)))
```

In [10]:
```
# Shuffle the dataset

tf_train_dataset = tf_train_dataset.shuffle(buffer_size=60000)
```

In [11]:
```
# Batch the dataset

tf_train_dataset = tf_train_dataset.batch(batch_size=64)
```

In [12]:
```
# Add a prefetch step

tf_train_dataset = tf_train_dataset.prefetch(tf.data.AUTOTUNE)
```

In [13]:
```
# Print the element_spec

tf_train_dataset.element_spec
```

Out[13]:
```
(TensorSpec(shape=(None, 28, 28), dtype=tf.float32, name=None),
 TensorSpec(shape=(None,), dtype=tf.uint8, name=None))
```

In [14]:
```
# Iterate over the batched and shuffled Dataset object

for inputs, labels in tf_train_dataset.take(2):
    print(inputs.shape)
    print(labels.shape)
    print(labels)
```

```
(64, 28, 28)
(64,)
tf.Tensor(
[7 2 5 1 2 4 2 4 7 1 4 8 6 8 1 6 2 6 5 1 8 0 3 1 6 5 7 6 4 3 0 2 8 0 7 0 8
 0 4 3 1 7 2 3 1 0 4 2 2 3 1 4 4 1 2 1 0 2 4 0 3 0 5 2], shape=(64,), dtyp
e=uint8)
(64, 28, 28)
(64,)
tf.Tensor(
[5 0 5 8 1 5 3 6 4 8 3 5 3 4 5 2 2 6 5 7 7 7 0 4 6 3 4 8 6 5 1 4 3 3 5 3 3
 1 0 5 1 2 3 5 6 1 5 1 4 3 5 1 0 0 3 3 4 3 4 6 6 5 7 6], shape=(64,), dtyp
e=uint8)
```

A typical pattern is to create a `Dataset` object as above, that returns a tuple of `(inputs, outputs)`. As we will see below, this `Dataset` object can then be used in the `model.fit` call instead of passing in input and output arrays. Keras detects that a `Dataset` object has been passed in, and expects it to return tuples of inputs and outputs as above.

PyTorch

In PyTorch, we will use the `torch.utils.data` module, and the `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` classes.

In [15]:
```
import torch
```

In [16]:
```
# Load the Fashion-MNIST dataset

(x_train, y_train), _ = keras.datasets.fashion_mnist.load_data()
```

In [17]:
```
# Normalise the pixel values and change dtype

import numpy as np

x_train_normalised = (x_train / 255.).astype(np.float32)
```

`TensorDataset`

If data is stored in NumPy arrays as above, then it is easy to create a `torch.utils.data.Dataset` object using the `torch.utils.data.TensorDataset` class. If we pass in Tensors as below, then the Dataset will correspondingly return tuples of Tensors. The first dimension of the Tensors needs to be the same, as this is assumed to index the elements in the dataset.

In [18]:
```
# Load the data into a torch.utils.data.TensorDataset

pt_train_dataset = torch.utils.data.TensorDataset(
    torch.from_numpy(x_train_normalised),
    torch.from_numpy(np.array(y_train)))
```

Custom `Dataset`

For extra flexibility, it is possible to create custom `Dataset` classes that incorporate transformations. Your custom class should implement `__init__`, `__len__` and

`__getitem__` methods.

In [19]:
```
# Create a custom Dataset class

class CustomFashionMNISTDataset(torch.utils.data.Dataset):

    def __init__(self, inputs, targets, transform=None,
                 target_transform=None):
        self.inputs = inputs
        self.targets = targets
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, idx):
        image = self.inputs[idx]
        label = self.targets[idx]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

In [20]:
```
# Use the NumPy arrays from the Keras API in custom Dataset, including no

def normalise(image):
    return torch.Tensor((image / 255.))

pt_train_dataset = CustomFashionMNISTDataset(
    x_train, y_train, transform=normalise)
```

`torchvision` datasets

PyTorch also has its own API for loading common datasets such as Fashion-MNIST.

In [22]:
```
# Load the Fashion-MNIST dataset using torchvision

import torchvision

pt_train_dataset = torchvision.datasets.FashionMNIST(
    root="~/torchdata",
    train=True,
    download=True,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Lambda(lambda x: torch.squeeze(x))
    ])
)
```

The above will download and store the dataset at the location `~/torchdata` (if it is not already there). The `train=True` argument specifies to download the training split (a test split is also available with `train=False`). If no transform is specified, the images are returned as `PIL.Image.Image` objects, but we need them represented as Tensor objects. The transform `ToTensor` accomplishes this for us, including scaling

the pixel values to the interval $[0, 1]$. The final transform reshapes the images from `(1, 28, 28)` to `(28, 28)` (removes the channel axis).

In [23]:
```
# Iterate over the Dataset

for i, (inputs, labels) in enumerate(pt_train_dataset):
    if i == 2:
        break
    print(type(inputs))
    print(type(labels))
    print(inputs.shape)
    print(labels)
```

```
<class 'torch.Tensor'>
<class 'int'>
torch.Size([28, 28])
9
<class 'torch.Tensor'>
<class 'int'>
torch.Size([28, 28])
0
```

We could also filter out data examples according to some criterion with the `Subset` class. For example, if we wanted to exclude all data examples with label 9 from the training:

In [24]:
```
# Filter out all examples with label 9 (ankle boot)

from torch.utils.data import Subset

train_idx = np.where(pt_train_dataset.targets != 9)[0]
pt_train_dataset = Subset(pt_train_dataset, train_idx)
```

We typically do not iterate directly through the Dataset. Instead, a `DataLoader` handles this, including constructing minibatches, reshuffling the data at every epoch, and speeding up performance.

In [25]:
```
# Create Dataloader

pt_train_loader = torch.utils.data.DataLoader(
    pt_train_dataset, batch_size=64, shuffle=True, drop_last=True)
```

The `DataLoader` is an iterable, and we can pull minibatches from it as follows:

In [26]:
```
# Test the DataLoader

x, y = next(iter(pt_train_loader))
print(x.shape)
print(y)
```

```
torch.Size([64, 28, 28])
tensor([6, 2, 4, 7, 8, 1, 4, 7, 3, 1, 7, 4, 6, 4, 3, 8, 6, 3, 2, 4, 1, 7,
4, 0,
        6, 7, 2, 5, 7, 7, 2, 2, 0, 7, 2, 6, 8, 4, 3, 6, 5, 6, 5, 2, 5, 7,
6, 5,
        0, 0, 2, 7, 5, 7, 7, 7, 6, 6, 4, 1, 8, 3, 1, 5])
```

A typical pattern is to create a `DataLoader` object as above, that returns a tuple of `(inputs, outputs)`. As we will see below, this `DataLoader` object can then be used in the `model.fit` call instead of passing in input and output arrays. Keras detects that a `DataLoader` object has been passed in, and expects it to return tuples of inputs and outputs as above.

Train an MNIST classifier

Below we show that a Keras model can be trained on either TensorFlow `Dataset` objects or PyTorch `DataLoader` objects, regardless of which backend is selected. Try running the following trainings with different backends.

In [27]:
```
# Check the Keras backend

keras.config.backend()
```

Out[27]:
```
'tensorflow'
```

In [28]:
```
# Define the classifier model

from keras.models import Sequential
from keras.layers import Flatten, Dense, Input

def get_model():
    model = Sequential([
        Input(shape=(28, 28)),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(64, activation='relu'),
        Dense(10)
    ], name='fashion_mnist_classifier')
    return model
```

In [32]:
```
# Define the loss and optimiser

loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
rmsprop = keras.optimizers.RMSprop(learning_rate=0.005)
```

First we create a model and train on the TensorFlow `Dataset`

In [33]:
```
# Train on the TensorFlow Dataset

tf_fashion_mnist_model = get_model()
tf_fashion_mnist_model.compile(loss=loss_fn, optimizer=rmsprop)
tf_fashion_mnist_model.fit(tf_train_dataset, epochs=2)
```

```
Epoch 1/2
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.7461
Epoch 2/2
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.4266
```

Out[33]:
```
<keras.src.callbacks.history.History at 0x72a3275bb310>
```

Now to train again using the PyTorch `DataLoader`, we create a new model instance and also a new optimizer object, since the existing optimizer is tied to the model above.

In [34]:
```
# Redefine the optimiser

rmsprop = keras.optimizers.RMSprop(learning_rate=0.005)
```

In [35]:
```
# Train on the PyTorch Dataset

pt_fashion_mnist_model = get_model()
pt_fashion_mnist_model.compile(optimizer=rmsprop, loss=loss_fn)
pt_fashion_mnist_model.fit(pt_train_loader, epochs=2)
```

```
Epoch 1/2
843/843 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.7450
Epoch 2/2
843/843 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - loss: 0.4317
```

Out[35]:
```
<keras.src.callbacks.history.History at 0x72a324df4350>
```

Dropout

Dropout was introduced by Srivastava et al in 2014 as a regularisation technique for neural networks, that also has the effect of modifying the behaviour of neurons within a network.

The following is taken from the paper abstract:

> Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.

The method of dropout is to randomly 'zero out' neurons (or equivalently, weight connections) in the network during training according to a Bernoulli mask whose values are independently sampled at every iteration.

Suppose $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$ is a weight matrix mapping neurons in layer k to layer $k+1$:

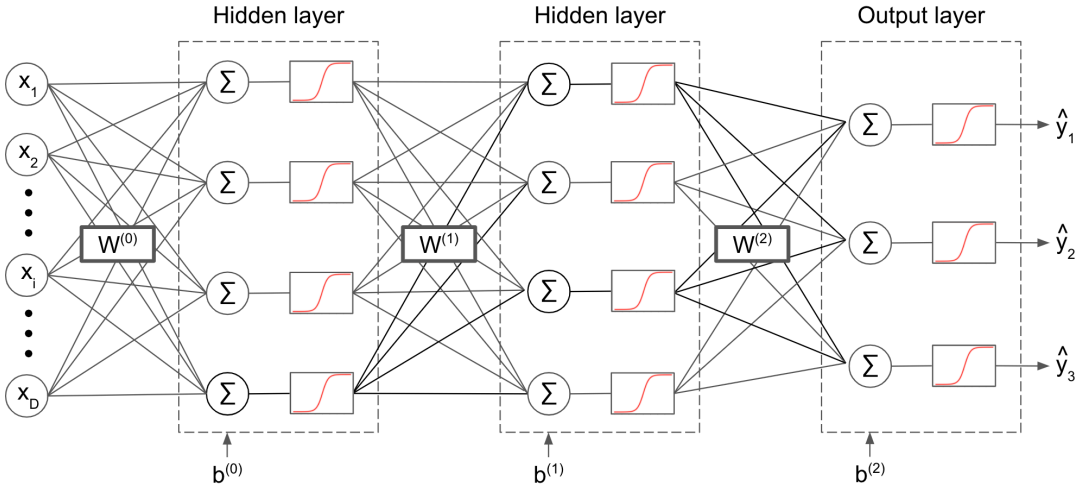
$$\mathbf{h}^{(k+1)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{h}^{(k)} + \mathbf{b}^{(k)}\right)$$

We can view dropout as randomly replacing each column of $\mathbf{W}^{(k)}$ with zeros with probability p_k. We can write this as applying a Bernoulli mask:

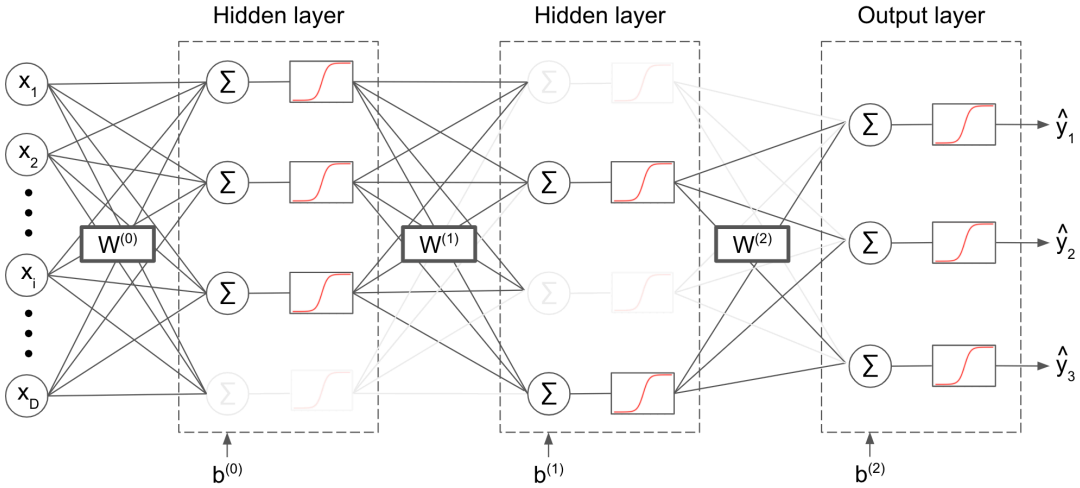
$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} \cdot \text{diag}([\mathbf{z}_{k,j}]_{j=1}^{n_{k-1}}) \tag{20}$$

$$\mathbf{z}_{k,j} \sim \text{Bernoulli}(p_k), \qquad k = 1, \ldots, L, \tag{21}$$

with $\mathbf{W}^{(k)} \in \mathbb{R}^{n_{k+1} \times n_k}$. The following diagrams illustrate the effect of dropout on a neural network.



Neural network without dropout



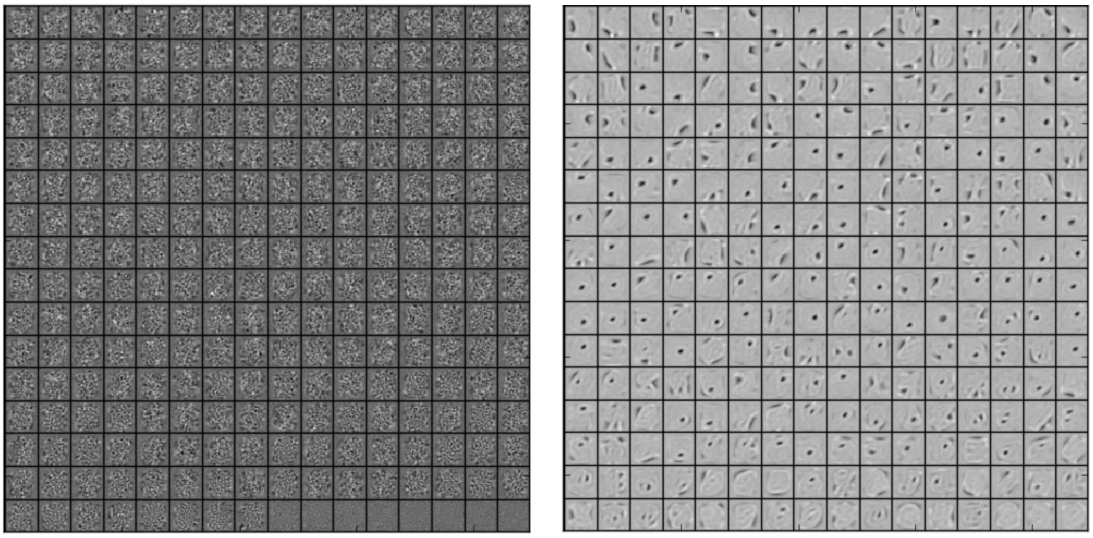
Neural network with dropout

By randomly dropping out neurons in the network, one obvious effect is that the capacity of the model is reduced, and so there is a regularisation effect. Each randomly sampled Bernoulli mask defines a new 'sub-network' that is smaller than the original.

In addition, a key motivation of dropout is that it prevents neurons from co-adapting too much. Any neuron in the network is no longer able to depend on any other specific neurons being present, and so each neuron learns features that are more robust, and generalise better.

In the figure below (taken from the original paper) we see features that are learned on the MNIST dataset for a model trained without dropout (left) and one trained with dropout (right). We see that the dropout model learns features that are much less noisy and more meaningful (it is detecting edges, textures, spots etc.) and help the model to

generalise better. The non-dropout model's features suggest a large degree of co-adaptation, where the neurons depend on the specific combination of features in order to make good predictions on the training data.



(a) Without dropout (b) Dropout with $p = 0.5$.

Learned features in a neural network trained without dropout (left) and with dropout (right). From Srivastava et al 2014

Typically, dropout is applied only in the training phase. When making predictions, all weight connections $\mathbf{W}^{(k)}$ are restored, but rescaled by a factor of $1 - p_k$ to take account for the fact that fewer connections were present at training.

However, Gal & Ghahramani describe a Bayesian interpretation of dropout, and proposed that dropout is also applied at test time in order to obtain a Bayesian predictive distribution.

In Keras, dropout is available to include in your models as another layer:

In [37]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout

dropout_model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(784,)),
    Dropout(rate=0.7),
    Dense(64, activation='sigmoid'),
    Dropout(rate=0.7),
    Dense(10, activation='softmax')
])
dropout_model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	
dense_15 (Dense)	(None, 64)	
dropout_2 (Dropout)	(None, 64)	
dense_16 (Dense)	(None, 64)	
dropout_3 (Dropout)	(None, 64)	
dense_17 (Dense)	(None, 10)	

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

Batch normalisation

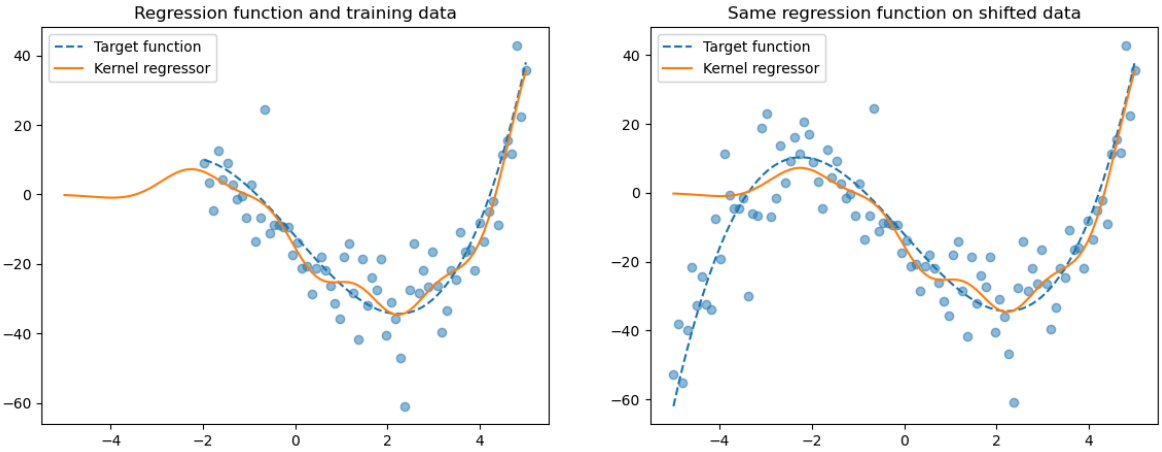
Batch normalisation (Ioffe & Szegedy 2015) is a widely used method in deep learning. It is used to normalise the distribution of internal activation values in the network, and greatly helps to stabilise learning especially in deep networks.

The core issue is that of **covariate shift**, which is the change in distribution of input variables to a machine learning model. This can happen in datasets where there is some change in conditions in subsequent data collections, for example over time or location. Whilst the underlying target function might not have changed, the distribution of the input variables does change which means the model could perform poorly in the changed conditions.

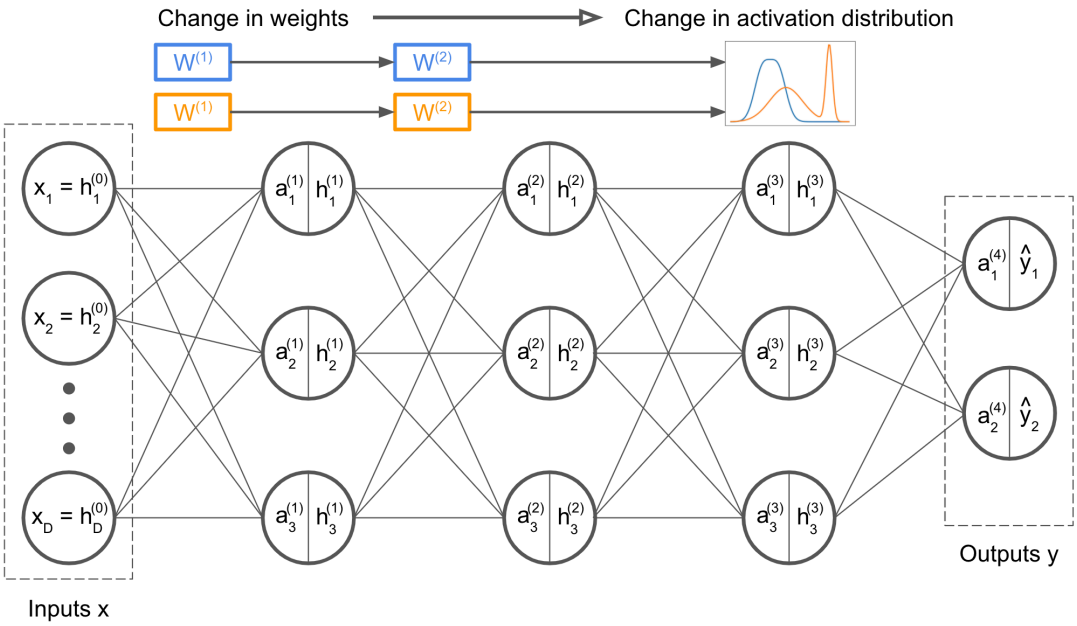
The following shows a simple example of a regression function that fails to generalise to new data points whose distribution has shifted from the training data, even though the underlying target function is the same.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.kernel_ridge import KernelRidge

def target(x):
    return x**3 - 15 * x - 12

n_samples = 100
x_train_all = np.linspace(-5, 5, n_samples)[..., np.newaxis]
x_train_sub = x_train_all[30:]
y_train_all = target(x_train_all) + 10 * np.random.randn(n_samples, 1)
y_train_sub = y_train_all[30:]

kernel_regressor_sub = KernelRidge(alpha=1e-2, kernel='rbf', gamma=0.5)
kernel_regressor_sub.fit(x_train_sub, y_train_sub)
mse1 = np.mean(
    (kernel_regressor_sub.predict(x_train_sub) - y_train_sub)**2)

kernel_regressor_all = KernelRidge(alpha=1e-2, kernel='rbf', gamma=0.5)
```

```
kernel_regressor_all.fit(x_train_all, y_train_all)

fig = plt.figure(figsize=(14, 5))

fig.add_subplot(1, 2, 1)
plt.plot(x_train_sub, target(x_train_sub), '--')
plt.plot(x_train_all, kernel_regressor_sub.predict(x_train_all))
plt.scatter(x_train_sub, y_train_sub, alpha=0.5)
plt.title("Regression function and training data")
plt.legend(['Target function', 'Kernel regressor'])

fig.add_subplot(1, 2, 2)
plt.plot(x_train_all, target(x_train_all), '--')
plt.plot(x_train_all, kernel_regressor_sub.predict(x_train_all))
plt.scatter(x_train_all, y_train_all, alpha=0.5)
plt.title("Same regression function on shifted data")
plt.legend(['Target function', 'Kernel regressor'])

plt.show()
```



The same phenomenon can occur during the course of training deep learning models on large datasets, where stochastic minibatches are used in the optimisation procedure. Furthermore, since deep learning models can be viewed as hierarchical feature extractors, we can encounter problems of **internal covariate shift**, where the activation values in hidden layers also undergo changes of distribution due to changes in parameter values and activations in earlier layers.



Changes in weights and activations earlier in the network cause internal covariate shift in activations in later layers

Batch normalisation reduces the internal covariate shift by normalising the mean and variance of the activation values in a layer. Intuitively speaking, this means that although layer inputs will change over the course of training, they won't change so much that learning becomes very slow or unstable. Batch normalisation also has a slight regularisation effect on the network.

For a layer with n_k-dimensional input $\mathbf{h}^{(k)} = (h_1^{(k)}, \ldots, h_{n_k}^{(k)})$, we normalise each input feature

$$\hat{h}_j^{(k)} = \frac{h_j^{(k)} - \mathbb{E}[h_j^{(k)}]}{\sqrt{\mathrm{Var}[h_j^{(k)}]}}.$$

In order to maintain full expressive power of the network, we make sure the final transformation can represent the identity:

$$z_j^{(k)} = \gamma_j^{(k)} \hat{h}_j^{(k)} + \beta_j^{(k)},$$

where $\gamma_j^{(k)}$ and $\beta_j^{(k)}$ are learned parameters. Note that setting $\gamma_j^{(k)} = \sqrt{\mathrm{Var}[h_j^{(k)}]}$ and $\beta_j^{(k)} = \mathbb{E}[h_j^{(k)}]$ recovers the original activations. However, now the model can control the mean and variance of activations within the hidden layer $\mathbf{h}^{(k)}$ by tuning the parameters $\gamma_j^{(k)}$ and $\beta_j^{(k)}$. $\mathbf{z}^{(k)}$ then becomes the new input to the next layer in the network.

Statistics $\mathbb{E}[h_j^{(k)}]$ and $\mathrm{Var}[h_j^{(k)}]$ are estimated over each minibatch $\mathcal{D}_m$:

$$\mu_{jm}^{(k)} = \frac{1}{M}\sum_{i=1}^{M} h_{ij}^{(k)} \tag{22}$$

$$\left(\sigma_{jm}^{(k)}\right)^2 = \frac{1}{M}\sum_{i=1}^{M}(h_{ij}^{(k)} - \mu_{jm}^{(k)})^2 \tag{23}$$

$$\hat{h}_j^{(k)} = \frac{h_j^{(k)} - \mu_{jm}^{(k)}}{\sqrt{\left(\sigma_{jm}^{(k)}\right)^2 + \epsilon}} \tag{24}$$

$$z_j^{(k)} = \gamma_j^{(k)}\hat{h}_j^{(k)} + \beta_j^{(k)} =: BN_{\gamma^{(k)},\beta^{(k)}}\left(h_j^{(k)}\right) \tag{25}$$

where $M = |\mathcal{D}_m|$, and $h_{ij}^{(k)}$ is the activation value for the j-th neuron for input $x_i \in \mathcal{D}_m$ in layer k.

At training time, the estimates $\mu_{jm}^{(k)}$ and $\sigma_{jm}^{(k)}$ are computed on the minibatch $\mathcal{D}_m$. In practical implementations, a running average of these estimates over the training run is used at test time.

The batch normalisation calculation is fully differentiable, and so gradients can be backpropagated through this calculation as normal.

The batch normalisation operation is implemented as a layer in Keras. In the following experiment we will recreate the first example from the original paper on the MNIST dataset.

In [2]:
```
import keras
```

In [3]:
```
# Load the MNIST dataset

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

In [5]:
```
# Create TensorFlow Dataset objects

import tensorflow as tf

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))

train_dataset.element_spec
```

Out[5]:
```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

You could also replace the above with a PyTorch `DataLoader`.

In [6]:
```
# Normalise pixel values in the Datasets

def normalise_pixels(image, label):
    return (tf.cast(image, tf.float32) / 255., label)

train_dataset = train_dataset.map(normalise_pixels)
test_dataset = test_dataset.map(normalise_pixels)
```

In [7]:
```
# Shuffle and batch the dataset

train_dataset = train_dataset.shuffle(1000).batch(60)
test_dataset = test_dataset.batch(60)
```

We now define an MLP classifier model for the MNIST dataset. The following demonstrates how to build a model using the functional API.

In [12]:
```
# Build the classifier model using the functional API

from keras.models import Model
from keras.layers import Flatten, Dense, Input
from keras.initializers import RandomNormal

inputs = Input(shape=(28, 28))
h = Flatten()(inputs)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
outputs = Dense(10, activation='softmax')(h)

no_bn_model = Model(inputs=inputs, outputs=outputs)
```

In [13]:
```
# Print the model summary

no_bn_model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	P
input_layer_1 (InputLayer)	(None, 28, 28)	
flatten_1 (Flatten)	(None, 784)	
dense_4 (Dense)	(None, 100)	
dense_5 (Dense)	(None, 100)	
dense_6 (Dense)	(None, 100)	
dense_7 (Dense)	(None, 10)	

Total params: 99,710 (389.49 KB)

Trainable params: 99,710 (389.49 KB)

Non-trainable params: 0 (0.00 B)

In [14]:
```
# Define a loss function, optimiser and metric

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

In [15]:
```
# Fit the model

no_bn_history = no_bn_model.fit(train_dataset, epochs=50,
                                validation_data=test_dataset)
```

```
Epoch 1/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.1082 - loss: 2.32
74 - val_accuracy: 0.1028 - val_loss: 2.3011
Epoch 2/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.1099 - loss: 2.30
16 - val_accuracy: 0.1028 - val_loss: 2.3003
Epoch 3/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - accuracy: 0.1112 - loss: 2.30
08 - val_accuracy: 0.1028 - val_loss: 2.2998
Epoch 4/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.1143 - loss: 2.30
01 - val_accuracy: 0.1028 - val_loss: 2.2989
Epoch 5/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1151 - loss: 2.29
94 - val_accuracy: 0.1028 - val_loss: 2.2982
Epoch 6/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1167 - loss: 2.29
84 - val_accuracy: 0.1028 - val_loss: 2.2974
Epoch 7/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1175 - loss: 2.29
74 - val_accuracy: 0.1028 - val_loss: 2.2960
Epoch 8/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1175 - loss: 2.29
62 - val_accuracy: 0.1028 - val_loss: 2.2945
Epoch 9/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1216 - loss: 2.29
48 - val_accuracy: 0.1148 - val_loss: 2.2924
Epoch 10/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1195 - loss: 2.29
27 - val_accuracy: 0.1253 - val_loss: 2.2900
Epoch 11/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1278 - loss: 2.29
00 - val_accuracy: 0.1029 - val_loss: 2.2867
Epoch 12/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1432 - loss: 2.28
63 - val_accuracy: 0.1254 - val_loss: 2.2813
Epoch 13/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1533 - loss: 2.28
06 - val_accuracy: 0.1615 - val_loss: 2.2737
Epoch 14/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.1883 - loss: 2.27
15 - val_accuracy: 0.3477 - val_loss: 2.2589
Epoch 15/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.2319 - loss: 2.25
49 - val_accuracy: 0.2699 - val_loss: 2.2312
Epoch 16/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.2825 - loss: 2.22
09 - val_accuracy: 0.4083 - val_loss: 2.1652
Epoch 17/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.3097 - loss: 2.14
05 - val_accuracy: 0.3726 - val_loss: 2.0220
Epoch 18/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.3424 - loss: 1.98
46 - val_accuracy: 0.3863 - val_loss: 1.8478
Epoch 19/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.3859 - loss: 1.82
30 - val_accuracy: 0.4158 - val_loss: 1.7292
Epoch 20/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4182 - loss: 1.71
16 - val_accuracy: 0.4322 - val_loss: 1.6368
```

```
Epoch 21/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4397 - loss: 1.61
44 - val_accuracy: 0.4530 - val_loss: 1.5328
Epoch 22/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4667 - loss: 1.50
30 - val_accuracy: 0.4858 - val_loss: 1.4169
Epoch 23/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5007 - loss: 1.38
84 - val_accuracy: 0.5259 - val_loss: 1.3135
Epoch 24/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5444 - loss: 1.28
96 - val_accuracy: 0.5742 - val_loss: 1.2197
Epoch 25/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.5885 - loss: 1.19
88 - val_accuracy: 0.6244 - val_loss: 1.1304
Epoch 26/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.6295 - loss: 1.11
05 - val_accuracy: 0.6536 - val_loss: 1.0407
Epoch 27/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.6625 - loss: 1.02
38 - val_accuracy: 0.6939 - val_loss: 0.9553
Epoch 28/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.6897 - loss: 0.94
29 - val_accuracy: 0.7143 - val_loss: 0.8839
Epoch 29/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7125 - loss: 0.87
74 - val_accuracy: 0.7396 - val_loss: 0.8278
Epoch 30/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7302 - loss: 0.82
69 - val_accuracy: 0.7511 - val_loss: 0.7862
Epoch 31/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7452 - loss: 0.78
78 - val_accuracy: 0.7643 - val_loss: 0.7516
Epoch 32/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7580 - loss: 0.75
63 - val_accuracy: 0.7747 - val_loss: 0.7233
Epoch 33/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7701 - loss: 0.72
79 - val_accuracy: 0.7820 - val_loss: 0.6982
Epoch 34/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7788 - loss: 0.70
49 - val_accuracy: 0.7943 - val_loss: 0.6746
Epoch 35/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.7899 - loss: 0.68
27 - val_accuracy: 0.8043 - val_loss: 0.6522
Epoch 36/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8005 - loss: 0.66
06 - val_accuracy: 0.8135 - val_loss: 0.6312
Epoch 37/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8087 - loss: 0.64
11 - val_accuracy: 0.8227 - val_loss: 0.6110
Epoch 38/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8179 - loss: 0.62
17 - val_accuracy: 0.8297 - val_loss: 0.5920
Epoch 39/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8244 - loss: 0.60
39 - val_accuracy: 0.8350 - val_loss: 0.5738
Epoch 40/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8336 - loss: 0.58
61 - val_accuracy: 0.8419 - val_loss: 0.5570
```

```
Epoch 41/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8394 - loss: 0.56
87 - val_accuracy: 0.8475 - val_loss: 0.5414
Epoch 42/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8452 - loss: 0.55
42 - val_accuracy: 0.8535 - val_loss: 0.5263
Epoch 43/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8510 - loss: 0.53
88 - val_accuracy: 0.8587 - val_loss: 0.5126
Epoch 44/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8551 - loss: 0.52
57 - val_accuracy: 0.8625 - val_loss: 0.4994
Epoch 45/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8584 - loss: 0.51
25 - val_accuracy: 0.8674 - val_loss: 0.4873
Epoch 46/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8633 - loss: 0.50
07 - val_accuracy: 0.8710 - val_loss: 0.4764
Epoch 47/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8676 - loss: 0.49
01 - val_accuracy: 0.8740 - val_loss: 0.4652
Epoch 48/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8716 - loss: 0.47
78 - val_accuracy: 0.8763 - val_loss: 0.4552
Epoch 49/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8740 - loss: 0.46
79 - val_accuracy: 0.8807 - val_loss: 0.4452
Epoch 50/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8765 - loss: 0.45
71 - val_accuracy: 0.8829 - val_loss: 0.4361
```

In [16]:
```
# Re-build the model with batch normalisation

from keras.layers import BatchNormalization

inputs = Input(shape=(28, 28))
h = Flatten()(inputs)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
h = BatchNormalization()(h)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
h = BatchNormalization()(h)
h = Dense(100, activation='sigmoid',
          kernel_initializer=RandomNormal())(h)
h = BatchNormalization()(h)
outputs = Dense(10, activation='softmax')(h)

bn_model = Model(inputs=inputs, outputs=outputs)
```

In [17]:
```
# Compile the model

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

In [18]:
```
# Fit the model

bn_history = bn_model.fit(train_dataset, epochs=50,
                          validation_data=test_dataset)
```

```
Epoch 1/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8237 - loss: 0.59
90 - val_accuracy: 0.9050 - val_loss: 0.3258
Epoch 2/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9017 - loss: 0.34
34 - val_accuracy: 0.9145 - val_loss: 0.2962
Epoch 3/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9101 - loss: 0.31
73 - val_accuracy: 0.9172 - val_loss: 0.2827
Epoch 4/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9150 - loss: 0.29
92 - val_accuracy: 0.9220 - val_loss: 0.2693
Epoch 5/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9191 - loss: 0.28
15 - val_accuracy: 0.9262 - val_loss: 0.2556
Epoch 6/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9254 - loss: 0.26
12 - val_accuracy: 0.9312 - val_loss: 0.2366
Epoch 7/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9300 - loss: 0.24
46 - val_accuracy: 0.9356 - val_loss: 0.2239
Epoch 8/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9350 - loss: 0.22
70 - val_accuracy: 0.9373 - val_loss: 0.2083
Epoch 9/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9398 - loss: 0.21
14 - val_accuracy: 0.9420 - val_loss: 0.1967
Epoch 10/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9440 - loss: 0.19
44 - val_accuracy: 0.9457 - val_loss: 0.1822
Epoch 11/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9478 - loss: 0.18
05 - val_accuracy: 0.9508 - val_loss: 0.1662
Epoch 12/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9503 - loss: 0.16
97 - val_accuracy: 0.9522 - val_loss: 0.1605
Epoch 13/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9536 - loss: 0.15
87 - val_accuracy: 0.9528 - val_loss: 0.1566
Epoch 14/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9569 - loss: 0.14
74 - val_accuracy: 0.9560 - val_loss: 0.1439
Epoch 15/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9594 - loss: 0.13
87 - val_accuracy: 0.9574 - val_loss: 0.1358
Epoch 16/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9609 - loss: 0.13
19 - val_accuracy: 0.9580 - val_loss: 0.1363
Epoch 17/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9642 - loss: 0.12
37 - val_accuracy: 0.9622 - val_loss: 0.1248
Epoch 18/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9651 - loss: 0.11
75 - val_accuracy: 0.9656 - val_loss: 0.1203
Epoch 19/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9670 - loss: 0.11
10 - val_accuracy: 0.9647 - val_loss: 0.1180
Epoch 20/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9685 - loss: 0.10
61 - val_accuracy: 0.9669 - val_loss: 0.1129
```

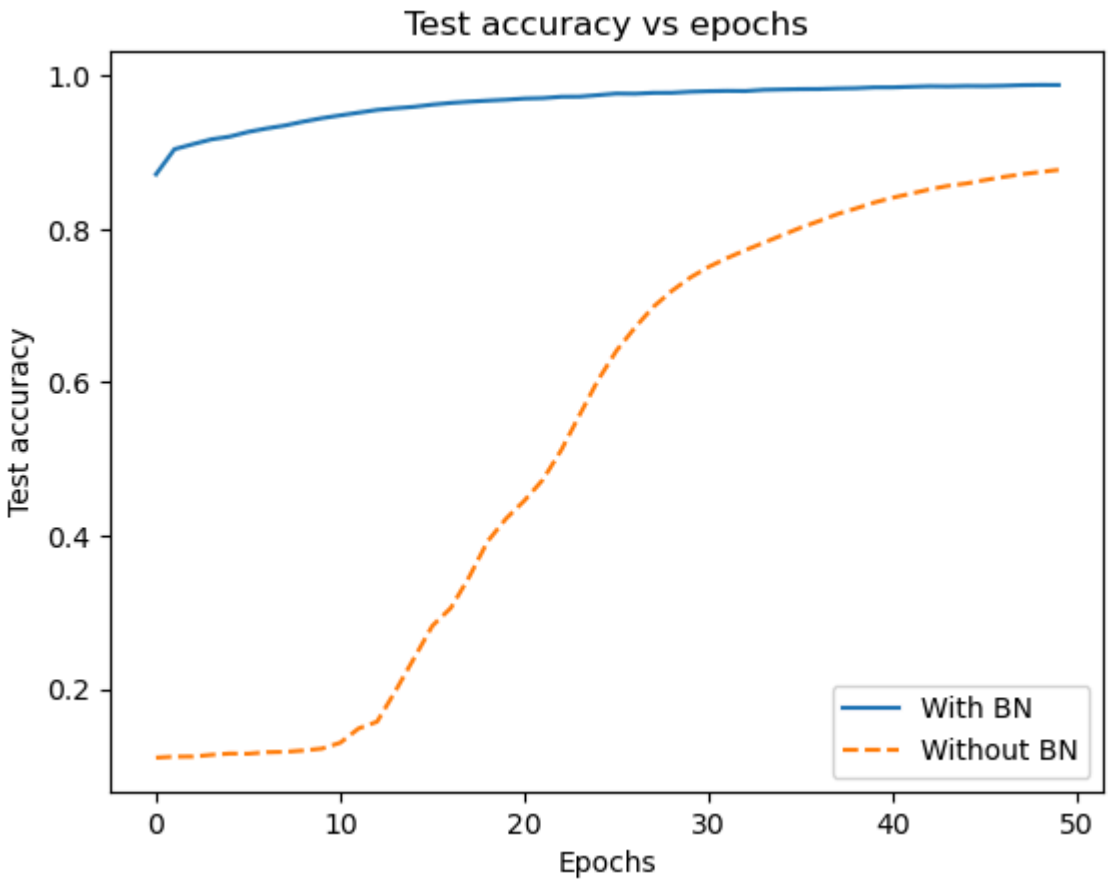
```
Epoch 21/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9705 - loss: 0.10
09 - val_accuracy: 0.9674 - val_loss: 0.1112
Epoch 22/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9708 - loss: 0.09
51 - val_accuracy: 0.9671 - val_loss: 0.1105
Epoch 23/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9723 - loss: 0.09
17 - val_accuracy: 0.9695 - val_loss: 0.1067
Epoch 24/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9736 - loss: 0.08
67 - val_accuracy: 0.9688 - val_loss: 0.1030
Epoch 25/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9754 - loss: 0.08
29 - val_accuracy: 0.9691 - val_loss: 0.1021
Epoch 26/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9765 - loss: 0.08
02 - val_accuracy: 0.9699 - val_loss: 0.1010
Epoch 27/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9764 - loss: 0.07
65 - val_accuracy: 0.9698 - val_loss: 0.0993
Epoch 28/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9769 - loss: 0.07
54 - val_accuracy: 0.9708 - val_loss: 0.0965
Epoch 29/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9773 - loss: 0.07
35 - val_accuracy: 0.9709 - val_loss: 0.0972
Epoch 30/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9793 - loss: 0.06
78 - val_accuracy: 0.9716 - val_loss: 0.0935
Epoch 31/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9789 - loss: 0.07
05 - val_accuracy: 0.9715 - val_loss: 0.0936
Epoch 32/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9803 - loss: 0.06
36 - val_accuracy: 0.9726 - val_loss: 0.0916
Epoch 33/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9794 - loss: 0.06
32 - val_accuracy: 0.9725 - val_loss: 0.0918
Epoch 34/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9812 - loss: 0.05
95 - val_accuracy: 0.9730 - val_loss: 0.0900
Epoch 35/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9812 - loss: 0.05
81 - val_accuracy: 0.9733 - val_loss: 0.0892
Epoch 36/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9820 - loss: 0.05
67 - val_accuracy: 0.9729 - val_loss: 0.0885
Epoch 37/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9836 - loss: 0.05
38 - val_accuracy: 0.9737 - val_loss: 0.0887
Epoch 38/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9823 - loss: 0.05
32 - val_accuracy: 0.9753 - val_loss: 0.0843
Epoch 39/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9826 - loss: 0.05
24 - val_accuracy: 0.9738 - val_loss: 0.0878
Epoch 40/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9846 - loss: 0.04
83 - val_accuracy: 0.9746 - val_loss: 0.0846
```

```
Epoch 41/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9844 - loss: 0.04
75 - val_accuracy: 0.9744 - val_loss: 0.0886
Epoch 42/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9847 - loss: 0.04
73 - val_accuracy: 0.9752 - val_loss: 0.0844
Epoch 43/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9854 - loss: 0.04
55 - val_accuracy: 0.9746 - val_loss: 0.0850
Epoch 44/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9852 - loss: 0.04
41 - val_accuracy: 0.9734 - val_loss: 0.0882
Epoch 45/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9866 - loss: 0.04
09 - val_accuracy: 0.9744 - val_loss: 0.0841
Epoch 46/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9861 - loss: 0.04
13 - val_accuracy: 0.9766 - val_loss: 0.0820
Epoch 47/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9870 - loss: 0.04
02 - val_accuracy: 0.9761 - val_loss: 0.0841
Epoch 48/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9874 - loss: 0.03
93 - val_accuracy: 0.9741 - val_loss: 0.0867
Epoch 49/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9875 - loss: 0.03
77 - val_accuracy: 0.9751 - val_loss: 0.0853
Epoch 50/50
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9874 - loss: 0.03
87 - val_accuracy: 0.9751 - val_loss: 0.0829
```

We will compare the progress of the test accuracy in both models.

In [19]:

```python
# Plot the test accuracy

import matplotlib.pyplot as plt

plt.plot(bn_history.history['accuracy'])
plt.plot(no_bn_history.history['accuracy'], '--')
plt.legend(['With BN', 'Without BN'])
plt.ylabel("Test accuracy")
plt.xlabel("Epochs")
plt.title("Test accuracy vs epochs")
plt.show()
```



We see clearly in the above plot that the batch normalisation layers help the model to train faster, and to a higher accuracy.

Batch normalisation reduces internal covariate shift, particularly early on in training. The distribution is more stable, making learning easier.

References

- Gal, Y. & Ghahramani, Z. (2016), "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning", Proceedings of The 33rd International Conference on Machine Learning, **48**, 1050-1059.
- Ioffe, S. & Szegedy, C., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", in *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, **37**, 448–456.
- Rumelhart, D. E., Hinton, G., and Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.
- Rumelhart, D. E., Hinton, G., and Williams, R. (1986c), "Learning Internal Representations by Error Propagation", in Rumelhart, D. E.; McClelland, J. L. (eds.), Parallel Distributed Processing : Explorations in the Microstructure of Cognition. Volume 1: Foundations, Cambridge, MIT Press.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research, **15**, 1929-1958.

- Werbos, P. J. (1994), "The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting", New York:, John Wiley & Sons.