Neural Networks

Madiba Hudson-Quansah

Contents

Chapter 1

Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model that is inspired by the way biological neural networks in the human brain work. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example and are thus capable of solving unsupervised as well as supervised learning tasks.

1.1 Biological Neurons

Artificial Neural Networks being inspired by biological neural networks are composed of neurons, thus to understand the decision designs of ANNs we must first understand the biological neuron. Neurons are made up of three parts, the *soma* or the cell body, the *axon* and *dendrites* A neuron is a specialized cell responsible for the creation and transportation of short electrical impulses called *action potentials* / AP / signals, used to communicate with other neurons with the overarching goal of transmitting information throughout the body. Through this vast network of neurons, the brain is able to perform highly complex computations. Research shows that neurons in the brain are organized in consecutive layers with each layer performing a specific task. This is the basis of the design of ANNs.

1.2 Logical Computation with Neurons

The Artificial Neuron is designed on this framework of the biological neuron, with a simple ANN proposed by McCulloch and Pitts, having one or more binary inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active, allowing for simple computation of logical functions, such as propositional logic. For example a simple neuron designed to perform the logical AND operation:

Let the output neuron be C and take two input neuron A and B. As the neuron C always requires two input signals to give an output signal we can then require the two input signals to only give one output signal if activated. This would in turn require both neurons A and B to be activated to give an output signal for neuron C, thus performing the logical AND operation.

1.3 The Perceptron

A simple ANN, based on a artificial neuron called a *threshold logic unit* (TLU) / *linear threshold unit* (LTU). It inputs and outputs numbers and each input connection is associated with a weight. The TLU first computes a linear function of its inputs

$$z = w_1 x_1 + \ldots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$$

Where the model parameters are the input weights $\mathbf{w}$ and b is the bias term

It then applies a **step function** to the result

$$h_w(\mathbf{x}) = \text{step}(z)$$

A common step function used in perceptrons is the *Heavyside step function*, and the *sign function* defined:

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

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

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a fully **connected layer** or a **dense layer**, and the inputs constitute the **input layer**. As the final layer of TLU's produce the output this layer is called the **output layer**.

Using linear algebra the outputs of nodes in the output layers, for several instances at once, i.e.:

$$h_{W,b}(X) = \phi(XW + b)$$

Where:

- W is a matrix containing all the connection weights, with one row per unit and one column per node.
- X is a matrix containing the input features, with noe row per instance and one colum per feature, where and instance is a data point.
- b is the bias vector that contains all the bias terms, one entry per neuron, where whe entires in b are broadcasted across the rows of XW.
- ϕ is the **activation function**, but when the artificial neurons are TLUs this is termed as the step function

The perceptron training algorithm is largely inspired by Hebb's rule, which suggests what when neurons on either side of a synapse or in the case of artificial neurons connection, fire together the connection between them is strengthened. This results in the strengthening of connections that help reduce the error in the output layer.

The perceptron is fed one training instance at a time and for each instance it makes predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is as follows:

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

Where:

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

The decision boundary of each output neuron is linear, so perceptrons are incapable of learning complex patterns (just like logistic regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the **perceptron convergence theorem**.

[1]:
```
import sys

!{sys.executable} -m pip install pydot
```

```
Requirement already satisfied: pydot in
/home/mads/.pyenv/versions/ai/lib/python3.12/site-packages (2.0.0)
Requirement already satisfied: pyparsing≥3 in
/home/mads/.pyenv/versions/ai/lib/python3.12/site-packages (from pydot) (3.1.2)
```

[2]:
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris.target == 0

per = Perceptron(random_state=42)
per.fit(X, y)

XNew = [[2, 0.5], [3, 1]]
per.predict(XNew)
```

[2]:
```
array([ True, False])
```

1.4 Multilayer Perceptron (MLP) and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called *hidden layers* and one output layer, also comprised of TLUs. Layers close the input layers are called the *lower layers* and those close to the output layers are called *upper layers*.

When signals only flow from the input layer across, hidden layers and finally stopping at the output layers, i.e one direction, the resulting architecture is called a **Feedforward Neural Network (FNN)**.

When an ANN contains a deep stack of hidden layers, it is called a **Deep Neural Network (DNN)**. A deep stack is when the number of hidden layers is large, typically more than 10.

Using the *reverse-mode automatic differentiation / reverse-mode autodiff* algorithm the gradient of a ANN's error in relation to the parameters it's given can be computed. Using this gradient it is possible to perform gradient descent to find the optimal weights and biases for the ANN, i.e. the parameters that minimize the error. This combination of both the reverse-mode autodiff and gradient descent is called **Backpropagation**.

Going into detail the Backpropagation algorithm is as follows:

- A mini-batch (for example 32 instances) of the training data is used at a time, and the algorithm goes through the full training data several times. Each time through the full training data is called an **epoch**.
- Each mini-batch enters the network through the input layer, then the output of the all the neurons in the first hidden layer is computed for each instance in the mini-batch. The result is then passed to the subsequent hidden layer and computed so on until the output layer is reached. This is called a **forward pass**, where the model makes predictions but all the intermediate results between hidden layers are stored for use in the backwards pass.
- The algorithm then calculates the model's output error, using a loss function that compares the result to the desired output of the network for the given parameters and returns the measure of the error.
- Then it computes how much each output bias and each connection to the output layer contributed to the error, by analytically applying the chain rule.
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backwards until it reaches the input layer. This is called the **backward / reverse pass**.

- Finally the algorithm performs a gradient descent step to tweak all the connections in the network, using the error contributions computed earlier. This is called the **gradient descent step**.

In order for Backpropagation to work properly it needs a gradient to work with, thus the step functions normally used in single layer perceptrons are replaced with differentiable functions, such as the **logistic / sigmoid function**, **hyperbolic tangent function** and the **rectified linear unit function (ReLU)**, defined below:

$$\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$$

$$\tanh(z) = 2\sigma(2z) - 1$$

$$\text{ReLU}(z) = \max(0, z)$$

1.5 Regression MLPs

Hyperparameter	Typical Value
# hidden layers	Depends on the problem typically 1 - 5
# neurons per hidden layer	Depends on the problem typically 10 - 100
# output neurons	1 per prediction dimension
hidden activation	ReLu
output activation	None, or ReLU/ softplus if positive outputs or sigmoid / tanhn if bounded
Loss function	MSE, or huber if outliers

[3]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import root_mean_squared_error
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()
XTrainFull, XTest, yTrainFull, yTest = train_test_split(
    housing.data, housing.target, random_state=42
)
XTrain, XValid, yTrain, yValid = train_test_split(
    XTrainFull, yTrainFull, random_state=42
)

mlpReg = MLPRegressor(
    hidden_layer_sizes=[50, 50, 50], random_state=42, activation="relu"
)
mlpRegPipe = make_pipeline(StandardScaler(), mlpReg)
mlpRegPipe.fit(XTrain, yTrain)
```

[3]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('mlpregressor',
                 MLPRegressor(hidden_layer_sizes=[50, 50, 50],
                              random_state=42))])
```

[4]:
```python
preds = mlpRegPipe.predict(XValid)
rmse = root_mean_squared_error(yValid, preds)
rmse
```

[4]:
```
0.5053326657968522
```

1.6 Classification MLPs

For binary classification the output neuron has to have a bounded activation function like sigmoid so the output will be a number between 0 and 1, which can be interpreted as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

For multilabel binary classification the output layer should have one output neuron per class, and each neuron should have a sigmoid activation function, where the first neuron predicts the probability of the first class, the second neuron the probability of the second class and so on.

If each instance can only belong to one category out of all the possible categories, then the output layer should have one neuron per class and use the **softmax** activation function for the whole output layer. This ensures that the sum of all the estimated class probabilities for each instance is equal to one, and that the estimated class probabilities are all non-negative.

Hyperparameter	Binary Classification	Multilabel Binary Classification	Multiclass Classification
# hidden layers	Typically 1 - 5 layers, depending on the task	Typically 1 - 5 layers, depending on the task	Typically 1 - 5 layers, depending on the task
# output neurons	1	1 per binary label	1 per class
Output activation	Sigmoid	Sigmoid	Softmax
Loss function	Cross-entropy	Cross-entropy	Cross-entropy

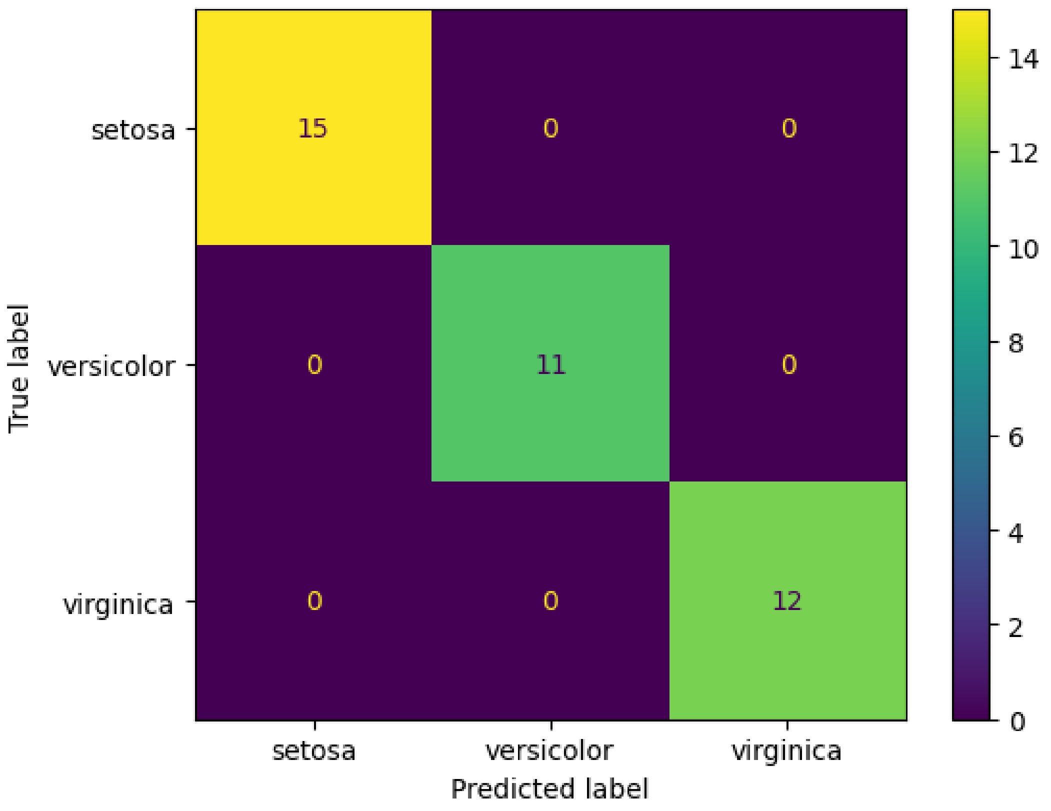
[5]:
```python
from sklearn.datasets import load_iris
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, ConfusionMatrixDisplay

iris = load_iris()
XTrainFull, XTest, yTrainFull, yTest = train_test_split(
    iris.data, iris.target, random_state=42
)
XTrain, XValid, yTrain, yValid = train_test_split(
    XTrainFull, yTrainFull, random_state=42
)

mlpCls = MLPClassifier(hidden_layer_sizes=[10], max_iter=10000,␣
  ↪random_state=42)
mlpClsPipe = make_pipeline(StandardScaler(), mlpCls)
mlpClsPipe.fit(XTrain, yTrain)
```

[5]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('mlpclassifier',
                 MLPClassifier(hidden_layer_sizes=[10], max_iter=10000,
                               random_state=42))])
```

```
[6]: preds = mlpClsPipe.predict(XTest)
     print(classification_report(yTest, preds))
     ConfusionMatrixDisplay.from_predictions(yTest, preds, display_labels=iris.
       ↪target_names)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       1.00      1.00      1.00        11
           2       1.00      1.00      1.00        12

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38
```

```
[6]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
     0x7e520c2cf560>
```



1.7 Implementing MLPs in Keras

1.7.1 Building an Image Classifier using the Sequential API

[7]:
```
import tensorflow as tf

fmnist = tf.keras.datasets.fashion_mnist.load_data()
(XTrainFull, yTrainFull), (XTest, yTest) = fmnist

cutoff = -5000
XTrain, yTrain = XTrainFull[:cutoff], yTrainFull[:cutoff]
XValid, yValid = XTrainFull[cutoff:], yTrainFull[cutoff:]
XTrain.shape
```

```
2024-07-13 23:17:06.657046: I tensorflow/core/util/port.cc:113] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-07-13 23:17:06.664028: I external/local_tsl/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2024-07-13 23:17:06.745207: I external/local_tsl/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2024-07-13 23:17:06.881027: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:479] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-07-13 23:17:06.986610: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:10575] Unable to
register cuDNN factory: Attempting to register factory for plugin cuDNN when one
has already been registered
2024-07-13 23:17:06.987620: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1442] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-07-13 23:17:07.182248: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2024-07-13 23:17:09.164982: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

[7]:
```
(55000, 28, 28)
```

[8]:
```
XTrain, XValid, XTest = XTrain / 255.0, XValid / 255.0, XTest / 255.0
classNames = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]


def name(x):
    return classNames[x]


name(yTrain[0])
```

[8]: 'Ankle boot'

[9]:
```
# Set random seed for reproducible results
tf.keras.utils.set_random_seed(42)

# Create a Sequential model a single stack of layers connected sequentially
cls = tf.keras.Sequential()

# The first layer, an Input layer, added to the model with shape
# 28x28, needed to determine the shape of the weight matrix of te
# first hidden layer
cls.add(tf.keras.layers.Input(shape=[28, 28]))

# A Flatten layer to convert the input images into 1D arrays
cls.add(tf.keras.layers.Flatten())

# A Dense layer with 300 neurons, using the ReLU activation function
# Each Dense layer manages it's own weight matrix containing the connection␣
 ↪weights
# between neurons and their inputs
cls.add(tf.keras.layers.Dense(300, activation="relu"))

# A second Dense layer with 100 neurons, also using the ReLU activation␣
 ↪function
cls.add(tf.keras.layers.Dense(100, activation="relu"))

# A final Dense layer as the output layer with 10 neurons, one per category
# And using the softmax activation function as classes are exclusive
cls.add(tf.keras.layers.Dense(10, activation="softmax"))
cls.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ flatten (Flatten)               │ (None, 784)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 300)            │       235,500 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
```

```
│ dense_1 (Dense)                  │ (None, 100)               │        30,100 │
├──────────────────────────────────┼───────────────────────────┼───────────────┤
│ dense_2 (Dense)                  │ (None, 10)                │         1,010 │
└──────────────────────────────────┴───────────────────────────┴───────────────┘

 Total params: 266,610 (1.02 MB)

 Trainable params: 266,610 (1.02 MB)

 Non-trainable params: 0 (0.00 B)
```

These steps can be condensed as seen below with the first `Input` layer dropped and the `input_shape` specified in the first layer

[10]:
```
cls = tf.keras.Sequential(
    [
        tf.keras.layers.Input(shape=[28, 28]),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(300, activation="leaky_relu"),
        tf.keras.layers.Dense(100, activation="leaky_relu"),
        tf.keras.layers.Dense(10, activation="softmax"),
    ]
)
cls.summary()
```

```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape              ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ flatten_1 (Flatten)              │ (None, 784)               │             0 │
├──────────────────────────────────┼───────────────────────────┼───────────────┤
│ dense_3 (Dense)                  │ (None, 300)               │       235,500 │
├──────────────────────────────────┼───────────────────────────┼───────────────┤
│ dense_4 (Dense)                  │ (None, 100)               │        30,100 │
├──────────────────────────────────┼───────────────────────────┼───────────────┤
│ dense_5 (Dense)                  │ (None, 10)                │         1,010 │
└──────────────────────────────────┴───────────────────────────┴───────────────┘

 Total params: 266,610 (1.02 MB)

 Trainable params: 266,610 (1.02 MB)

 Non-trainable params: 0 (0.00 B)
```

[11]:
```
display(cls.layers)
hid1 = cls.layers[1]
```

```
w, b = hid1.get_weights()
b
```

```
[<Flatten name=flatten_1, built=True>,
 <Dense name=dense_3, built=True>,
 <Dense name=dense_4, built=True>,
 <Dense name=dense_5, built=True>]
```

```
[11]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
              0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

1.7.1.1 Compiling the Model

```
[12]: # Loss - Specifies the loss function to use when minimizing the value of the␣
        ↪loss function
      # Optimize - Specifies the optimization algorithm to use in reducing the loss␣
        ↪function
      # Metrics - Specifies other metrics to monitor during training and testing
      cls.compile(
          loss="sparse_categorical_crossentropy",
          optimizer=tf.keras.optimizers.SGD(),
          metrics=["accuracy"],
      )
```

- We use `sparse_categorical_crossentropy` loss as we have sparse labels, for each instance there is just a target class index from 0 to 9, and the classes are exclusive. If we instead had one target probabilty per class for each instance for example when using One Hot Encoding, we would use `categorical_crossentropy` loss. When doing binary classification or multilabel binary classification we would use the `binary_crossentropy` loss
- The optimizer `sgd` stands for *Stochastic Gradient Descent*, and means the model will be trained using this gradient descent variant. The `sgd` optimizer will perform the backpropagation algorithm. When using the `sdg` optimizer it is important to set a learning rate, this can be done by passing `tf.keras.optimizers.SGD(learning_rate=x)` where `x` is the learning rate

1.7.1.2 Training and Evaluating the Model

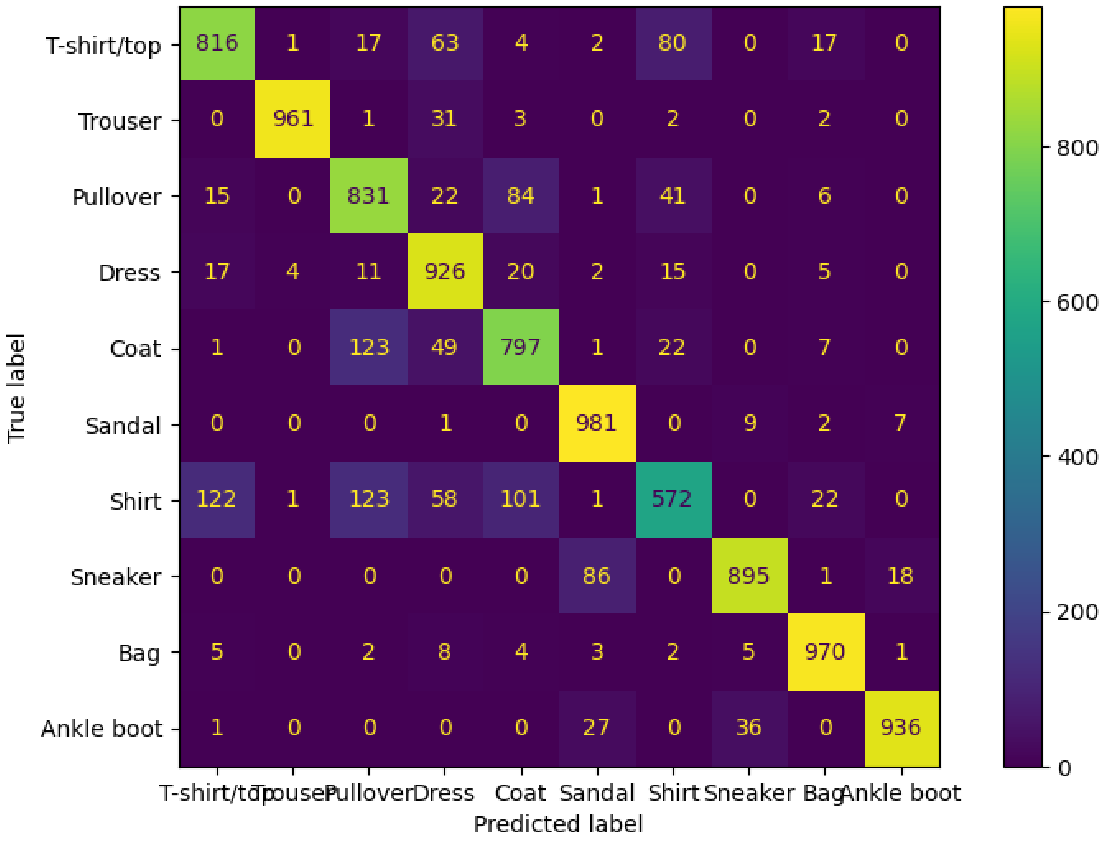
[13]:
```
hist = cls.fit(XTrain, yTrain, epochs=30, validation_data=(XValid, yValid))
```

```
Epoch 1/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step -
accuracy: 0.6862 - loss: 0.9684 - val_accuracy: 0.8304 - val_loss: 0.5021
Epoch 2/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8262 - loss: 0.5073 - val_accuracy: 0.8400 - val_loss: 0.4540
Epoch 3/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step -
accuracy: 0.8420 - loss: 0.4570 - val_accuracy: 0.8466 - val_loss: 0.4333
Epoch 4/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step -
accuracy: 0.8505 - loss: 0.4295 - val_accuracy: 0.8470 - val_loss: 0.4207
Epoch 5/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8570 - loss: 0.4097 - val_accuracy: 0.8506 - val_loss: 0.4106
Epoch 6/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8618 - loss: 0.3941 - val_accuracy: 0.8528 - val_loss: 0.4020
Epoch 7/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8670 - loss: 0.3810 - val_accuracy: 0.8552 - val_loss: 0.3963
Epoch 8/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8704 - loss: 0.3696 - val_accuracy: 0.8564 - val_loss: 0.3912
Epoch 9/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8732 - loss: 0.3596 - val_accuracy: 0.8582 - val_loss: 0.3856
Epoch 10/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step -
accuracy: 0.8758 - loss: 0.3507 - val_accuracy: 0.8602 - val_loss: 0.3817
Epoch 11/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8784 - loss: 0.3426 - val_accuracy: 0.8634 - val_loss: 0.3768
Epoch 12/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step -
accuracy: 0.8806 - loss: 0.3350 - val_accuracy: 0.8644 - val_loss: 0.3739
Epoch 13/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8829 - loss: 0.3280 - val_accuracy: 0.8654 - val_loss: 0.3708
Epoch 14/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8845 - loss: 0.3213 - val_accuracy: 0.8658 - val_loss: 0.3687
Epoch 15/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8863 - loss: 0.3152 - val_accuracy: 0.8668 - val_loss: 0.3667
Epoch 16/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8884 - loss: 0.3093 - val_accuracy: 0.8666 - val_loss: 0.3629
Epoch 17/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step -
```

```
accuracy: 0.8900 - loss: 0.3037 - val_accuracy: 0.8674 - val_loss: 0.3609
Epoch 18/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8918 - loss: 0.2984 - val_accuracy: 0.8678 - val_loss: 0.3598
Epoch 19/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8938 - loss: 0.2934 - val_accuracy: 0.8686 - val_loss: 0.3590
Epoch 20/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8958 - loss: 0.2885 - val_accuracy: 0.8686 - val_loss: 0.3580
Epoch 21/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.8979 - loss: 0.2838 - val_accuracy: 0.8700 - val_loss: 0.3593
Epoch 22/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9001 - loss: 0.2792 - val_accuracy: 0.8694 - val_loss: 0.3567
Epoch 23/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step -
accuracy: 0.9014 - loss: 0.2748 - val_accuracy: 0.8704 - val_loss: 0.3590
Epoch 24/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9030 - loss: 0.2707 - val_accuracy: 0.8722 - val_loss: 0.3579
Epoch 25/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9044 - loss: 0.2666 - val_accuracy: 0.8722 - val_loss: 0.3559
Epoch 26/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9056 - loss: 0.2627 - val_accuracy: 0.8712 - val_loss: 0.3577
Epoch 27/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9070 - loss: 0.2589 - val_accuracy: 0.8710 - val_loss: 0.3560
Epoch 28/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9086 - loss: 0.2552 - val_accuracy: 0.8714 - val_loss: 0.3571
Epoch 29/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9099 - loss: 0.2516 - val_accuracy: 0.8722 - val_loss: 0.3569
Epoch 30/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9115 - loss: 0.2480 - val_accuracy: 0.8730 - val_loss: 0.3553
```

[14]:
```python
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, ConfusionMatrixDisplay


def classStats(actuals, preds, classNames):
    print(classification_report(actuals, preds))
    fig, ax = plt.subplots(figsize=(10, 6))
    ConfusionMatrixDisplay.from_predictions(
        actuals, preds, display_labels=classNames, ax=ax
    )


preds = cls.predict(XTest)
preds = preds.argmax(axis=1)

classStats(yTest, preds, classNames)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
              precision    recall  f1-score   support

           0       0.84      0.82      0.83      1000
           1       0.99      0.96      0.98      1000
           2       0.75      0.83      0.79      1000
           3       0.80      0.93      0.86      1000
           4       0.79      0.80      0.79      1000
           5       0.89      0.98      0.93      1000
           6       0.78      0.57      0.66      1000
           7       0.95      0.90      0.92      1000
           8       0.94      0.97      0.95      1000
           9       0.97      0.94      0.95      1000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```



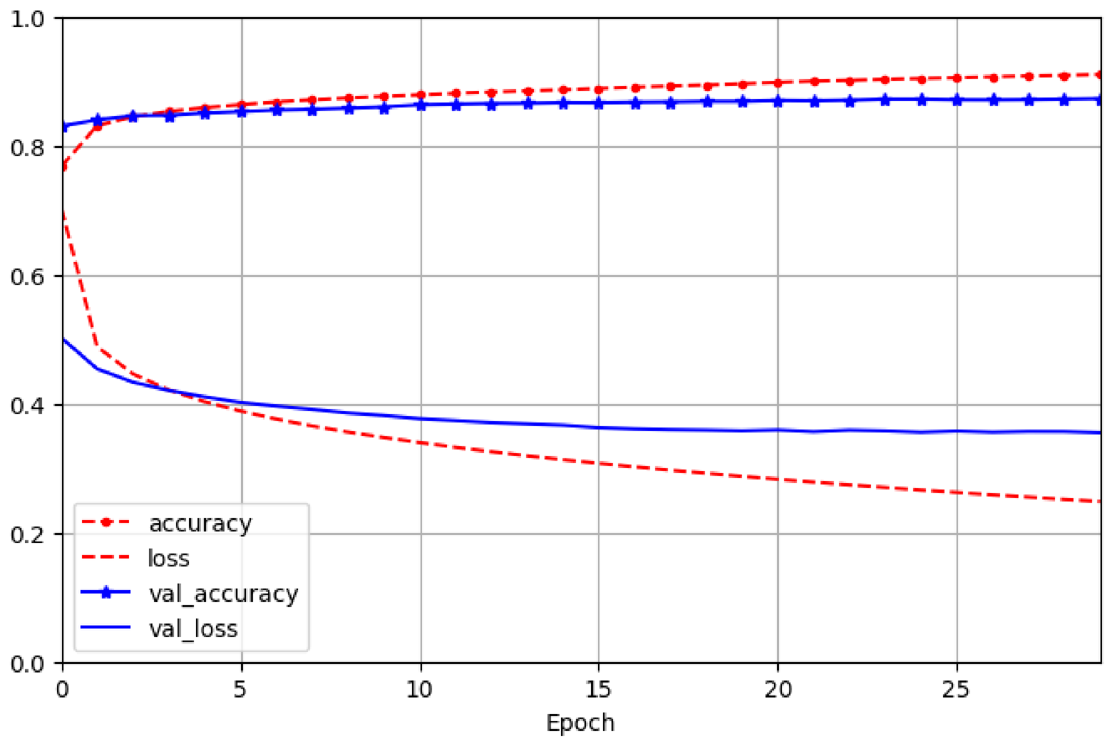
True label \ Predicted label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	816	1	17	63	4	2	80	0	17	0
Trouser	0	961	1	31	3	0	2	0	2	0
Pullover	15	0	831	22	84	1	41	0	6	0
Dress	17	4	11	926	20	2	15	0	5	0
Coat	1	0	123	49	797	1	22	0	7	0
Sandal	0	0	0	1	0	981	0	9	2	7
Shirt	122	1	123	58	101	1	572	0	22	0
Sneaker	0	0	0	0	0	86	0	895	1	18
Bag	5	0	2	8	4	3	2	5	970	1
Ankle boot	1	0	0	0	0	27	0	36	0	936

If the training data was very skewed, with over/under-representation of some classes you can set the `class_weight` argument in the `fit` method to give a larger weight to underpresented classes and a smaller weight to overrepresented classes. This is used when computing loss during training, so the model gives more importance to underrepresented classes. If per instance weights are needed the `sample_weight` argument can be used. When both `class_weight` and `sample_weight` are used the weights are multiplied. For example if you wanted to give more weight to more recent data you could use `sample_weight` and if you wanted to give more weight to underrepresented classes you could use `class_weight`.

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`) and a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set. You can use the `history.history` dictionary to plot the learning curves.

[15]:
```
import pandas as pd

pd.DataFrame(hist.history).plot(
    figsize=(8, 5),
    xlim=[0, 29],
    ylim=[0, 1],
    grid=True,
    xlabel="Epoch",
    style=["r--.", "r--", "b-*", "b-"],
)
```

[15]:
```
<Axes: xlabel='Epoch'>
```

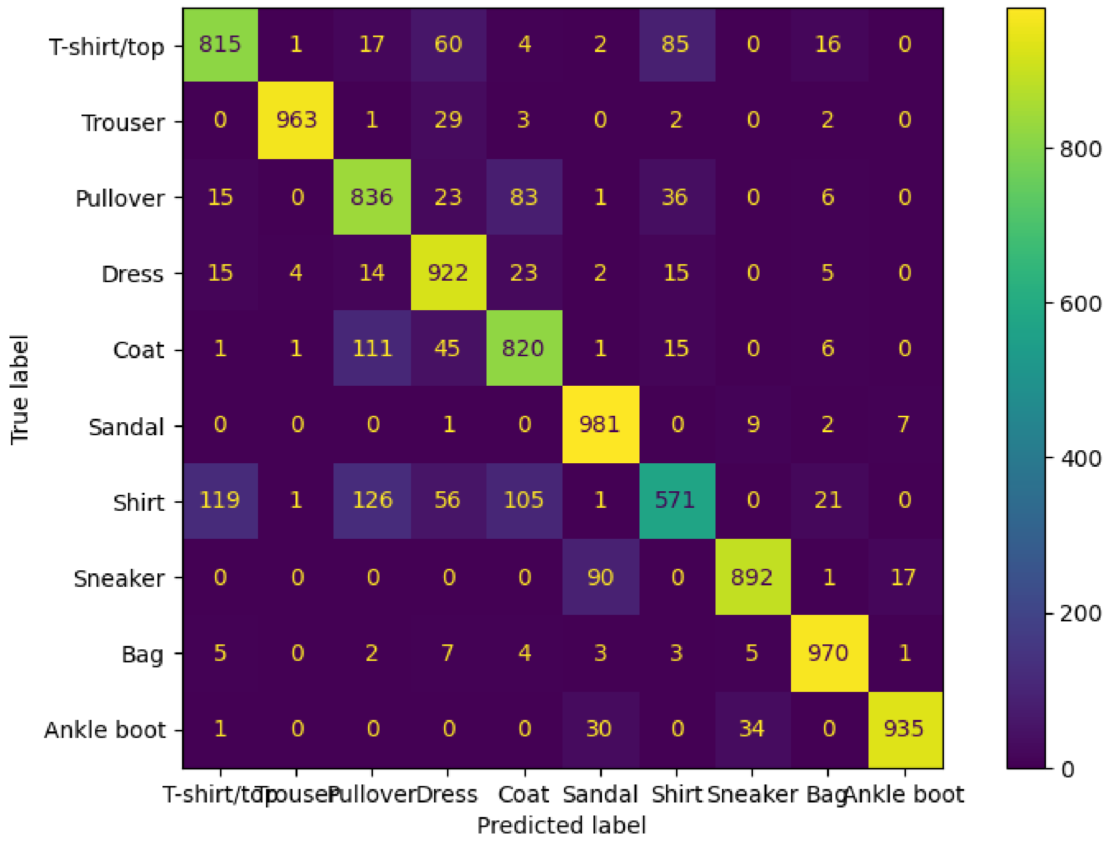


[16]:
```
hist = cls.fit(XTrain, yTrain, epochs=5, validation_data=(XValid, yValid))
```

```
Epoch 1/5
 430/1719 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step -
accuracy: 0.9149 - loss: 0.2405

1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9135 - loss: 0.2446 - val_accuracy: 0.8724 - val_loss: 0.3561
Epoch 2/5
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9147 - loss: 0.2414 - val_accuracy: 0.8734 - val_loss: 0.3570
Epoch 3/5
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step -
accuracy: 0.9158 - loss: 0.2381 - val_accuracy: 0.8736 - val_loss: 0.3571
Epoch 4/5
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step -
accuracy: 0.9167 - loss: 0.2350 - val_accuracy: 0.8730 - val_loss: 0.3584
Epoch 5/5
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step -
accuracy: 0.9178 - loss: 0.2319 - val_accuracy: 0.8734 - val_loss: 0.3590
```

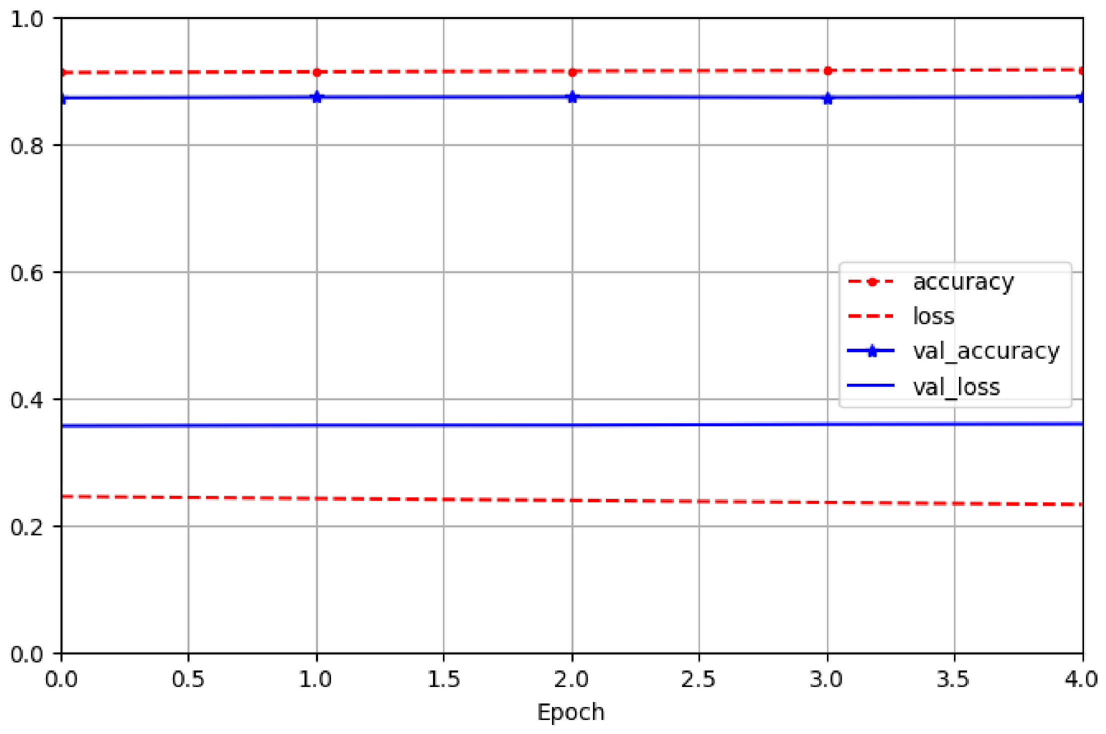
[17]:
```
preds = cls.predict(XTest)
preds = preds.argmax(axis=1)

display(cls.evaluate(XTest, yTest))
classStats(yTest, preds, classNames)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 884us/step -
accuracy: 0.8727 - loss: 0.3713

[0.3727381229400635, 0.8705000281333923]
              precision    recall  f1-score   support

           0       0.84      0.81      0.83      1000
           1       0.99      0.96      0.98      1000
           2       0.76      0.84      0.79      1000
           3       0.81      0.92      0.86      1000
           4       0.79      0.82      0.80      1000
           5       0.88      0.98      0.93      1000
           6       0.79      0.57      0.66      1000
           7       0.95      0.89      0.92      1000
           8       0.94      0.97      0.96      1000
           9       0.97      0.94      0.95      1000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```



[18]:
```
pd.DataFrame(hist.history).plot(
    figsize=(8, 5),
    xlim=[0, 4],
    ylim=[0, 1],
    grid=True,
    xlabel="Epoch",
    style=["r--.", "r--", "b-*", "b-"],
)
```

[18]:
```
<Axes: xlabel='Epoch'>
```



1.7.2 Building a Regression MLP using the Sequential API

[19]:
```
tf.keras.utils.set_random_seed(42)

housing = fetch_california_housing()
XTrainFull, XTest, yTrainFull, yTest = train_test_split(
    housing.data, housing.target, random_state=42
)
XTrain, XValid, yTrain, yValid = train_test_split(
    XTrainFull, yTrainFull, random_state=42
)
```

[20]:
```
norm = tf.keras.layers.Normalization()

reg = tf.keras.Sequential(
    [
        tf.keras.layers.Input(shape=XTrain.shape[1:]),
        norm,
        tf.keras.layers.Dense(50, activation="relu"),
        tf.keras.layers.Dense(50, activation="relu"),
        tf.keras.layers.Dense(50, activation="relu"),
        tf.keras.layers.Dense(1),
    ]
)
```

[21]:
```
opt = tf.keras.optimizers.Adam(learning_rate=1e-3)

reg.compile(loss="mse", optimizer=opt, metrics=["RootMeanSquaredError"])
```

[22]:
```
reg.summary()
```

Model: "sequential_2"

Layer (type)	**Output Shape**	**Param #**
normalization (Normalization)	(None, 8)	17
dense_6 (Dense)	(None, 50)	450
dense_7 (Dense)	(None, 50)	2,550
dense_8 (Dense)	(None, 50)	2,550
dense_9 (Dense)	(None, 1)	51

Total params: 5,618 (21.95 KB)

Trainable params: 5,601 (21.88 KB)

Non-trainable params: 17 (72.00 B)

[23]:
```
norm.adapt(XTrain)
hist = reg.fit(XTrain, yTrain, epochs=20, validation_data=(XValid, yValid))
```

```
Epoch 1/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step -
RootMeanSquaredError: 1.2143 - loss: 1.5762 - val_RootMeanSquaredError: 0.6527 -
val_loss: 0.4261
Epoch 2/20
  1/363 ━━━━━━━━━━━━━━━━━━━━ 10s 28ms/step -
RootMeanSquaredError: 0.8502 - loss: 0.7228

363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step -
RootMeanSquaredError: 0.6372 - loss: 0.4066 - val_RootMeanSquaredError: 0.5933 -
val_loss: 0.3521
Epoch 3/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.6074 - loss: 0.3692 - val_RootMeanSquaredError: 0.6970 -
val_loss: 0.4858
Epoch 4/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5922 - loss: 0.3509 - val_RootMeanSquaredError: 0.7863 -
val_loss: 0.6183
```

```
Epoch 5/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 964us/step -
RootMeanSquaredError: 0.5808 - loss: 0.3375 - val_RootMeanSquaredError: 1.1434 -
val_loss: 1.3074
Epoch 6/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5719 - loss: 0.3272 - val_RootMeanSquaredError: 1.3506 -
val_loss: 1.8242
Epoch 7/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step -
RootMeanSquaredError: 0.5658 - loss: 0.3202 - val_RootMeanSquaredError: 0.7185 -
val_loss: 0.5162
Epoch 8/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 974us/step -
RootMeanSquaredError: 0.5564 - loss: 0.3096 - val_RootMeanSquaredError: 0.8469 -
val_loss: 0.7172
Epoch 9/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 958us/step -
RootMeanSquaredError: 0.5513 - loss: 0.3040 - val_RootMeanSquaredError: 0.6418 -
val_loss: 0.4119
Epoch 10/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 980us/step -
RootMeanSquaredError: 0.5463 - loss: 0.2985 - val_RootMeanSquaredError: 0.6756 -
val_loss: 0.4564
Epoch 11/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5426 - loss: 0.2944 - val_RootMeanSquaredError: 0.7295 -
val_loss: 0.5321
Epoch 12/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 973us/step -
RootMeanSquaredError: 0.5396 - loss: 0.2912 - val_RootMeanSquaredError: 1.0084 -
val_loss: 1.0169
Epoch 13/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 961us/step -
RootMeanSquaredError: 0.5377 - loss: 0.2891 - val_RootMeanSquaredError: 0.6520 -
val_loss: 0.4251
Epoch 14/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 970us/step -
RootMeanSquaredError: 0.5341 - loss: 0.2853 - val_RootMeanSquaredError: 1.0415 -
val_loss: 1.0847
Epoch 15/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 945us/step -
RootMeanSquaredError: 0.5318 - loss: 0.2828 - val_RootMeanSquaredError: 0.7899 -
val_loss: 0.6239
Epoch 16/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 957us/step -
RootMeanSquaredError: 0.5298 - loss: 0.2807 - val_RootMeanSquaredError: 1.1863 -
val_loss: 1.4073
Epoch 17/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 990us/step -
RootMeanSquaredError: 0.5282 - loss: 0.2790 - val_RootMeanSquaredError: 0.5334 -
val_loss: 0.2845
Epoch 18/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
```

```
RootMeanSquaredError: 0.5241 - loss: 0.2748 - val_RootMeanSquaredError: 0.5660 -
val_loss: 0.3204
Epoch 19/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step -
RootMeanSquaredError: 0.5215 - loss: 0.2720 - val_RootMeanSquaredError: 0.5181 -
val_loss: 0.2684
Epoch 20/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 979us/step -
RootMeanSquaredError: 0.5194 - loss: 0.2698 - val_RootMeanSquaredError: 0.5378 -
val_loss: 0.2893
```

[24]:
```
mse, rmse = reg.evaluate(XTest, yTest)
```

```
162/162 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5235 - loss: 0.2742
```

[25]:
```
XNew = XTest[:3]
preds = reg.predict(XNew)
preds.round(2)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 92ms/step
```

[25]:
```
array([[0.47],
       [1.2 ],
       [4.8 ]], dtype=float32)
```

1.7.3 Building Complex Models using the Functional API

An example of a non-sequential neural network is a *Wide & Deep* neural network, where all or part of the inputs are connected directly to the output layer. This makes it possible for the network to learn both deep patterns (using the deep path) and simple rules (through the short path). In contrast a regular MLP forces all the data to flow through the deep path, thus simple patterns in the data may end up being distorted by this sequence of transformations.

[26]:
```
from functools import partial
from tensorflow.keras.layers import Normalization, Dense, Concatenate, Input

hidLay = partial(Dense, units=30, activation="relu")

normLayer = Normalization()
hidLay1 = hidLay()
hidLay2 = hidLay()
concatLayer = Concatenate()
outputLayer = Dense(1)

input_ = Input(shape=XTrain.shape[1:])
normed = normLayer(input_)
hid1 = hidLay1(normed)
hid2 = hidLay2(hid1)
concat = concatLayer([normed, hid2])
output = outputLayer(concat)

regWide = tf.keras.Model(inputs=[input_], outputs=[output])
regWide.summary()
```

```
Model: "functional_7"

┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)        ┃ Output Shape      ┃    Param # ┃ Connected to      ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ input_layer_3       │ (None, 8)         │          0 │ -                 │
│ (InputLayer)        │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ normalization_1     │ (None, 8)         │         17 │ input_layer_3[0]… │
│ (Normalization)     │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_10 (Dense)    │ (None, 30)        │        270 │ normalization_1[… │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_11 (Dense)    │ (None, 30)        │        930 │ dense_10[0][0]    │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ concatenate         │ (None, 38)        │          0 │ normalization_1[… │
│ (Concatenate)       │                   │            │ dense_11[0][0]    │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_12 (Dense)    │ (None, 1)         │         39 │ concatenate[0][0] │
└─────────────────────┴───────────────────┴────────────┴───────────────────┘

 Total params: 1,256 (4.91 KB)

 Trainable params: 1,239 (4.84 KB)

 Non-trainable params: 17 (72.00 B)
```

- First we create five layers: a `Normalization` layer to standardize inputs, two `Dense` layers with 30 neurons each using the ReLU activation function, a `Concatenate` layer, and one more `Dense` layer with a single output layer without any activation function.
- Next we create an `Input` object, specifying the shape and dtype of the inputs. This is used as a function, passing it the inputs.
- Then we pass the `Normalization` layer just like a function, passing the `Input` layer. The passing of layers as arguments to other layers only serves to connect them together.
- We pass `normed` to `hidLay1`, which outputs `hid1` and we pass `hid1` to `hidLay2`, which outputs `hid2`.
- So far all the connections have been sequential, but then we use `concatLayer` to concatenate the input and second hidden layer's output
- Then pass `concat` to the `outputLayer` which finally gives us `output`.
- Finally we create a Keras model, specifying which inputs and outputs to use.

[27]:
```
opt = tf.keras.optimizers.Adam(learning_rate=1e-3)
regWide.compile(loss="mse", optimizer=opt, metrics=["RootMeanSquaredError"])
```

[28]:
```
normLayer.adapt(XTrain)
```

```
hist = regWide.fit(XTrain, yTrain, epochs=20, validation_data=(XValid,␣
  ↪yValid))
```

```
Epoch 1/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step -
RootMeanSquaredError: 1.4678 - loss: 2.2793 - val_RootMeanSquaredError: 2.9178 -
val_loss: 8.5133
Epoch 2/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 844us/step -
RootMeanSquaredError: 0.7139 - loss: 0.5107 - val_RootMeanSquaredError: 1.7047 -
val_loss: 2.9059
Epoch 3/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.6398 - loss: 0.4096 - val_RootMeanSquaredError: 1.0166 -
val_loss: 1.0334
Epoch 4/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 902us/step -
RootMeanSquaredError: 0.6173 - loss: 0.3812 - val_RootMeanSquaredError: 0.6176 -
val_loss: 0.3815
Epoch 5/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 952us/step -
RootMeanSquaredError: 0.6050 - loss: 0.3662 - val_RootMeanSquaredError: 0.5938 -
val_loss: 0.3526
Epoch 6/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 876us/step -
RootMeanSquaredError: 0.5967 - loss: 0.3562 - val_RootMeanSquaredError: 0.5811 -
val_loss: 0.3377
Epoch 7/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 887us/step -
RootMeanSquaredError: 0.5902 - loss: 0.3485 - val_RootMeanSquaredError: 0.5746 -
val_loss: 0.3301
Epoch 8/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 857us/step -
RootMeanSquaredError: 0.5838 - loss: 0.3409 - val_RootMeanSquaredError: 0.5665 -
val_loss: 0.3209
Epoch 9/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 861us/step -
RootMeanSquaredError: 0.5788 - loss: 0.3351 - val_RootMeanSquaredError: 0.5610 -
val_loss: 0.3147
Epoch 10/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5745 - loss: 0.3302 - val_RootMeanSquaredError: 0.5579 -
val_loss: 0.3113
Epoch 11/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 901us/step -
RootMeanSquaredError: 0.5712 - loss: 0.3264 - val_RootMeanSquaredError: 0.5627 -
val_loss: 0.3166
Epoch 12/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 891us/step -
RootMeanSquaredError: 0.5684 - loss: 0.3232 - val_RootMeanSquaredError: 0.5619 -
val_loss: 0.3158
Epoch 13/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 859us/step -
RootMeanSquaredError: 0.5652 - loss: 0.3196 - val_RootMeanSquaredError: 0.6016 -
```

```
val_loss: 0.3620
Epoch 14/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 864us/step -
RootMeanSquaredError: 0.5624 - loss: 0.3164 - val_RootMeanSquaredError: 0.5646 -
val_loss: 0.3187
Epoch 15/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5596 - loss: 0.3133 - val_RootMeanSquaredError: 0.7858 -
val_loss: 0.6175
Epoch 16/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 958us/step -
RootMeanSquaredError: 0.5574 - loss: 0.3108 - val_RootMeanSquaredError: 0.6477 -
val_loss: 0.4196
Epoch 17/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 939us/step -
RootMeanSquaredError: 0.5553 - loss: 0.3084 - val_RootMeanSquaredError: 1.1061 -
val_loss: 1.2234
Epoch 18/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 901us/step -
RootMeanSquaredError: 0.5533 - loss: 0.3063 - val_RootMeanSquaredError: 0.6701 -
val_loss: 0.4490
Epoch 19/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 916us/step -
RootMeanSquaredError: 0.5516 - loss: 0.3044 - val_RootMeanSquaredError: 1.7290 -
val_loss: 2.9895
Epoch 20/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 900us/step -
RootMeanSquaredError: 0.5539 - loss: 0.3069 - val_RootMeanSquaredError: 1.9701 -
val_loss: 3.8811
```

[29]:
```
regWide.evaluate(XTest, yTest)
```

```
162/162 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step -
RootMeanSquaredError: 0.5631 - loss: 0.3174
```

[29]:
```
[0.31699109077453613, 0.5630196332931519]
```

In passing different subsets of features along different paths in the network multiple inputs could be used.

[30]:
```
inputWide = Input(shape=[5], name="inputWide")
inputDeep = Input(shape=[6], name="inputDeep")

normLayerWide = Normalization()
normLayerDeep = Normalization()

normWide = normLayerWide(inputWide)
normDeep = normLayerDeep(inputDeep)

hid1 = hidLay()(normDeep)
hid2 = hidLay()(hid1)

concat = tf.keras.layers.concatenate([normWide, hid2])

output = Dense(1)(concat)
```

```
regWide = tf.keras.Model(inputs=[inputWide, inputDeep], outputs=[output])
regWide.summary()
```

```
Model: "functional_9"

┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)        ┃ Output Shape      ┃    Param # ┃ Connected to      ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ inputDeep           │ (None, 6)         │          0 │ -                 │
│ (InputLayer)        │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ normalization_3     │ (None, 6)         │         13 │ inputDeep[0][0]   │
│ (Normalization)     │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ inputWide           │ (None, 5)         │          0 │ -                 │
│ (InputLayer)        │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_13 (Dense)    │ (None, 30)        │        210 │ normalization_3[… │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ normalization_2     │ (None, 5)         │         11 │ inputWide[0][0]   │
│ (Normalization)     │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_14 (Dense)    │ (None, 30)        │        930 │ dense_13[0][0]    │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ concatenate_1       │ (None, 35)        │          0 │ normalization_2[… │
│ (Concatenate)       │                   │            │ dense_14[0][0]    │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_15 (Dense)    │ (None, 1)         │         36 │ concatenate_1[0]… │
└─────────────────────┴───────────────────┴────────────┴───────────────────┘

 Total params: 1,200 (4.70 KB)

 Trainable params: 1,176 (4.59 KB)

 Non-trainable params: 24 (104.00 B)
```

- Each `Dense` layer is created and called on the same line as is common practice. However this isn't done with the `Normalization` layers as a reference to the layer is needed for adaptation.
- Used `tf.keras.layers.concatenate` to concatenate the inputs and the output of the second hidden layer.
- We specified two inputs when creating the model as there are two inputs.

[31]:
```
opt = tf.keras.optimizers.Adam(learning_rate=1e-3)
regWide.compile(loss="mse", optimizer=opt, metrics=["RootMeanSquaredError"])
```

```
[32]: XTrainWide, XTrainDeep = XTrain[:, :5], XTrain[:, 2:]
      XValidWide, XValidDeep = XValid[:, :5], XValid[:, 2:]
      XTestWide, XTestDeep = XTest[:, :5], XTest[:, 2:]
      XNewWide, XNewDeep = XTestWide[:3], XTestDeep[:3]

      normLayerWide.adapt(XTrainWide)
      normLayerDeep.adapt(XTrainDeep)

      hist = regWide.fit(
          (XTrainWide, XTrainDeep),
          yTrain,
          validation_data=((XValidWide, XValidDeep), yValid),
          epochs=20,
      )
```

```
Epoch 1/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step -
RootMeanSquaredError: 1.7786 - loss: 3.2631 - val_RootMeanSquaredError: 0.9088 -
val_loss: 0.8260
Epoch 2/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 917us/step -
RootMeanSquaredError: 0.8502 - loss: 0.7241 - val_RootMeanSquaredError: 0.9169 -
val_loss: 0.8408
Epoch 3/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 940us/step -
RootMeanSquaredError: 0.7640 - loss: 0.5841 - val_RootMeanSquaredError: 0.8662 -
val_loss: 0.7503
Epoch 4/20
297/363 ━━━━━━━━━━━━━━━━━━━━ 0s 679us/step -
RootMeanSquaredError: 0.7173 - loss: 0.5149

363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step -
RootMeanSquaredError: 0.7125 - loss: 0.5081 - val_RootMeanSquaredError: 0.9910 -
val_loss: 0.9821
Epoch 5/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.6711 - loss: 0.4506 - val_RootMeanSquaredError: 0.9600 -
val_loss: 0.9216
Epoch 6/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.6402 - loss: 0.4101 - val_RootMeanSquaredError: 0.8299 -
val_loss: 0.6888
Epoch 7/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 938us/step -
RootMeanSquaredError: 0.6181 - loss: 0.3821 - val_RootMeanSquaredError: 1.3156 -
val_loss: 1.7309
Epoch 8/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 954us/step -
RootMeanSquaredError: 0.6084 - loss: 0.3702 - val_RootMeanSquaredError: 1.4480 -
val_loss: 2.0968
Epoch 9/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.6010 - loss: 0.3612 - val_RootMeanSquaredError: 1.2405 -
val_loss: 1.5388
```

```
Epoch 10/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 929us/step -
RootMeanSquaredError: 0.5949 - loss: 0.3541 - val_RootMeanSquaredError: 1.3897 -
val_loss: 1.9313
Epoch 11/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 926us/step -
RootMeanSquaredError: 0.5920 - loss: 0.3506 - val_RootMeanSquaredError: 1.3386 -
val_loss: 1.7919
Epoch 12/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 939us/step -
RootMeanSquaredError: 0.5907 - loss: 0.3490 - val_RootMeanSquaredError: 1.2488 -
val_loss: 1.5596
Epoch 13/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 940us/step -
RootMeanSquaredError: 0.5863 - loss: 0.3438 - val_RootMeanSquaredError: 0.9414 -
val_loss: 0.8862
Epoch 14/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 936us/step -
RootMeanSquaredError: 0.5836 - loss: 0.3407 - val_RootMeanSquaredError: 0.9483 -
val_loss: 0.8992
Epoch 15/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 896us/step -
RootMeanSquaredError: 0.5806 - loss: 0.3372 - val_RootMeanSquaredError: 0.9576 -
val_loss: 0.9170
Epoch 16/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 899us/step -
RootMeanSquaredError: 0.5799 - loss: 0.3364 - val_RootMeanSquaredError: 1.0987 -
val_loss: 1.2072
Epoch 17/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 922us/step -
RootMeanSquaredError: 0.5794 - loss: 0.3357 - val_RootMeanSquaredError: 1.0939 -
val_loss: 1.1966
Epoch 18/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 974us/step -
RootMeanSquaredError: 0.5778 - loss: 0.3339 - val_RootMeanSquaredError: 1.2056 -
val_loss: 1.4534
Epoch 19/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 965us/step -
RootMeanSquaredError: 0.5776 - loss: 0.3337 - val_RootMeanSquaredError: 1.1802 -
val_loss: 1.3928
Epoch 20/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5764 - loss: 0.3323 - val_RootMeanSquaredError: 1.2530 -
val_loss: 1.5700
```

```
[33]: mseTest = regWide.evaluate((XTestWide, XTestDeep), yTest)
```

```
162/162 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step -
RootMeanSquaredError: 0.5803 - loss: 0.3369
```

```
[34]: preds = regWide.predict((XNewWide, XNewDeep))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
```

```
[ ]:
```

Chapter 2

Deep Computer Vision and Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are designed based on the way the human visual cortex perceives images with the neurons in the visual cortex being sensitive to small sub-regions of the visual field, called *receptive fields*. This is the basis of the design of CNNs, where the neurons in the first layer are connected to the pixels in the images in their receptive fields, and the neurons in the next layer are connected to the neurons in the previous layer, and so on

2.1 Convolutional Layers

Neurons in the first Convolutional layer are not connected to every pixel in the input image but instead only to pixels in their receptive fields. In turn each neuron in the second convolutional layer is connected only to neurons located within in a small rectangle in the first layer. This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer, and so on. The layers of a CNN are arranged in 2 dimensions in rows and in columns unlike the multi-layered perceptrons which are arranged in a single dimension.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field. In order for a layer to have the same height and weight as the previous layer it is common to add zeros around the inputs, called *zero padding*. It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, thereby greatly reducing the model's computational complexity.

The horizontal or vertical step size from one receptive field to the next is called the *stride*. By increasing the stride the output layer will be smaller than the input layer, but the receptive field will be larger.

2.1.1 Filters

A neuron's weights can be represented as a small image the size of the receptive field. These weights can be termed as *filters* / *convolutional kernels* / *kernels*. These weights can be used to change what data the neuron actually takes in with entires in the filter being the parameters that the backpropagation algorithm can tweak. For example for a neuron with a receptive field of size 7x7, a filter could be a 7x7 matrix with all entries being 0 except for the 4th column which would be 1s. This filter would cause the neuron to detect everything in its receptive field except for the central vertical line, making it only sensitive to vertical lines.

Now if all the neurons in a layer use this vertical line filter (and the same bias term) an image passed to this layer would enhance the vertical lines in the image, while the horizontal lines would be blurred. Thus a layer full of neuron using the same filter outputs a **feature map**, which highlights the areas in an image that activate the filter the most.

During training the convolutional layer learns the most useful features fo the task at hand, which above layers can then use to detect more complex patterns.

Employing the filter of a feature map the output of a neuron in a convolutional layer can be calculated as follows:

$$z_{i,j} = b + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i',j'} \times w_{u,v} \text{ with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

With:

- $z_{i,j}$ being the output of the neuron located in row i and column j in the convolutional layer
- b being the bias term for the neuron
- s_h and s_w being the vertical and horizontal strides
- f_h and f_w being the height and width of the receptive field
- $x_{i',j'}$ being the input of the neuron located in row i' and column j' in the previous layer
- $w_{u,v}$ being the connection weight between any neuron in the previous layer and the neuron located in row u and column $v in the convolutional layer

2.1.2 Stacking Feature Maps

Convolution layers can have multiple filters and outputs one feature map per feature. CLs have one on neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters, kernel and bias term. Neurons in different feature maps have different parameters, meaning the previous CL in the network neurons' receptive fields extend across the depth of the next CL, simultaneously applying all the filters to its input making it able to detect multiple features anywhere in its inputs.

Input layers are also composed of layers on per colour channel, usually coloured pictures have Red, Green and Blue layers. Specifically a neuron in the ith row, jth column and k feature map of a CL l is connected to the output neurons of the $l-1$ CL in the in the rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps in later $l-1$. Note that within a CL, all the neuron in the same row i and column j but in different feature maps are connected to the outputs of the same neurons in the previous layer. This leads to the output of a neuron in a feature map having the following equation:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \text{ with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

With

- $z_{i,j,k}$ being the output of the neuron located in row i, column j in feature map k of the convolutional layer l
- s_h and s_h being the vertical and horizontal stride and f_w and f_h being the width and height of the receptive field
- $f_{n'}$ being the number of feature maps in the previous layer ($l-1$)
- $x_{i',j',k'}$ being the output if the neuron located in layer $l-1$, row i', column j' and feature map k' or channel k' if the previous layer is the input layer.
- b_k being the bias term for the feature map k in layer l, which basically determines the brightness of feature map k
- $w_{u,v,k',k}$ being the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v, relative to the neuron's receptive field, and feature map k'

[35]:
```python
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
images.shape
```

[35]:
```
TensorShape([2, 70, 120, 3])
```

[36]:
```python
convLayer1 = tf.keras.layers.Conv2D(filters=32, kernel_size=7, padding="same")
fmaps = convLayer1(images)
fmaps.shape
```

[36]:
```
TensorShape([2, 70, 120, 32])
```

[37]:
```python
kernels, biases = convLayer1.get_weights()
kernels.shape
```

[37]:
```
(7, 7, 3, 32)
```

[38]:
```python
biases.shape
```

[38]:
```
(32,)
```

2.1.3 Activation functions

As the convolutional layers perform linear operations, and thus can only learn linear transformations of the input data, it is necessary to introduce non-linearity to the layers to allow the network to learn complex patterns. This is done by applying an activation function to the output of each neuron in the convolutional layer. The most common activation function used in CNNs is the **Rectified Linear Unit (ReLU)**, already defined above. This function is usually applied to the hidden convolutional layers of the network, but not to the output layer.

Activation functions are applied on the whole layer at once, meaning that the same activation function is applied to all the neurons in the layer. This is done to ensure that the network can learn to detect different patterns in different parts of the image. Therefore the activation function augments the output of the neuron in the feature map k of the layer l as follows:

$$h_{i,j,k} = \text{ReLU}(z_{i,j,k})$$

And in the case of a convolutional layer with a single feature map:

$$h_{i,j} = \text{ReLU}(z_{i,j})$$

[39]:
```python
convLayer1 = tf.keras.layers.Conv2D(
    filters=32, kernel_size=7, padding="same", activation="relu"
)
fmaps = convLayer1(images)
fmaps.shape
```

[39]:
```
TensorShape([2, 70, 120, 32])
```

[40]:
```python
kernels, biases = convLayer1.get_weights()
kernels.shape
```

```
[40]: (7, 7, 3, 32)
```

2.2 Pooling Layers

The goal of pooling layers is to subsample / shrink the input image in order to reduce the computational load, memory usage and the number of parameters. Just like convolutional layers each neuron in a pooling layer is only connected to a limited number of neurons in the previous layer in a receptive field, however a pooling layer has no weights, all it does is aggregate the inputs using an aggregation function such as the mean or the max. A pooling layer employing the max aggregation function is called a **max pooling layer**.

Other than reducing computations, memory usage, and the number of parameters, max pooling layers introduce some level of ***invariance*** to small translations. This means that if a small feature is slightly moved in the input image, the output of the max pooling layer may not change. This can be useful for detecting patterns in images, as the location of the pattern in the image will not matter. By inserting a max pooling layer after every convolutional layer, the network can learn to be invariant to small translations which can be useful for classification tasks.

Max Pooling is however very destructive, as it discards all the information except for the maximum value. This can be a problem for tasks that require precise localization. To address this issue, a common strategy is to add a convolutional layer with the same number of filters as the max pooling layer just before the max pooling layer. This way the network can learn to preserve the information that will be most useful to the max pooling layer.

Pooling can also be performed along the depth dimension instead of spatial dimensions which will allow the CNN to be invariant to various features, such as rotation and scaling.

```
[41]: maxPool = tf.keras.layers.MaxPool2D(pool_size=2)
```

```
[42]: class DepthPool(tf.keras.layers.Layer):
          def __init__(self, pool_size=2, **kwargs):
              super().__init__(**kwargs)
              self.pool_size = pool_size

          def call(self, inputs):
              shape = tf.shape(inputs)
              groups = shape[-1] // self.pool_size
              newShape = tf.concat([shape[:-1], [groups, self.pool_size]], axis=0)
              return tf.reduce_max(tf.reshape(inputs, newShape), axis=-1)
```

Another type of pooling layer used commonly is the *global average pooling layer*. This layer computes the mean of each entire feature map, like an average pooling layer using a pooling size of the entire feature map. This means the layer outputs a single number per feature map and per instance. Although this is very destructive it can be useful just before the output layer.

Keras does not have a Depth wise Max Pooling layer, but it can be implemented reshaping its inputs to split the channels into groups of the desired, size then using `tf.reduce_max()` to compute the max of each group. This implementation assumes that the stride is equal to the pooling size.

```
[43]: globAvgPool = tf.keras.layers.GlobalAveragePooling2D()
      globAvgPool(images)
```

```
[43]: <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
      array([[0.64338624, 0.5971759 , 0.5824972 ],
             [0.76306933, 0.2601113 , 0.10849128]], dtype=float32)>
```

2.3 CNN Architectures

A typical CNN architecture consists of a few convolutional layers, each generally followed by a ReLU layer, then a pooling layer, then another few convolutional layers again followed by a ReLU, then another pooling layer, and so on. The image gets smaller and smaller as it goes through the network, but also getting deeper and deeper due to the increasing number of filters in each convolutional layer. At the top of the stack of convolutional layers a regular feedforward neural network is added, composed of a few fully connected layers (+ReLU), and the final layer outputs the prediction, potentially using a softmax activation function if the network is used for classification to output estimated class probabilities.

[44]:
```
from functools import partial

conv2DLayer = partial(
    tf.keras.layers.Conv2D,
    kernel_size=3,
    padding="same",
    activation="relu",
    kernel_initializer="he_normal",
)
ffLayer = partial(
    tf.keras.layers.Dense, activation="relu", kernel_initializer="he_normal"
)

cnnCls = tf.keras.Sequential(
    [
        tf.keras.layers.Input(shape=[28, 28, 1]),
        tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
        conv2DLayer(filters=64, kernel_size=7),
        tf.keras.layers.MaxPool2D(),
        conv2DLayer(filters=128),
        conv2DLayer(filters=128),
        tf.keras.layers.MaxPool2D(),
        conv2DLayer(filters=256),
        conv2DLayer(filters=256),
        tf.keras.layers.MaxPool2D(),
        tf.keras.layers.Flatten(),
        ffLayer(units=128),
        tf.keras.layers.Dropout(0.5),
        ffLayer(units=64),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=10, activation="softmax"),
    ]
)

cnnCls.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 28, 28, 1)	0

```
│ conv2d_2 (Conv2D)               │ (None, 28, 28, 64)     │        3,200 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 14, 14, 64)     │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ conv2d_3 (Conv2D)               │ (None, 14, 14, 128)    │       73,856 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ conv2d_4 (Conv2D)               │ (None, 14, 14, 128)    │      147,584 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 7, 7, 128)      │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ conv2d_5 (Conv2D)               │ (None, 7, 7, 256)      │      295,168 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ conv2d_6 (Conv2D)               │ (None, 7, 7, 256)      │      590,080 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ max_pooling2d_3 (MaxPooling2D)  │ (None, 3, 3, 256)      │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ flatten_2 (Flatten)             │ (None, 2304)           │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ dense_16 (Dense)                │ (None, 128)            │      295,040 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ dropout (Dropout)               │ (None, 128)            │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ dense_17 (Dense)                │ (None, 64)             │        8,256 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ dropout_1 (Dropout)             │ (None, 64)             │            0 │
├─────────────────────────────────┼────────────────────────┼──────────────┤
│ dense_18 (Dense)                │ (None, 10)             │          650 │
└─────────────────────────────────┴────────────────────────┴──────────────┘

 Total params: 1,413,834 (5.39 MB)

 Trainable params: 1,413,834 (5.39 MB)

 Non-trainable params: 0 (0.00 B)
```

- First we use the `partial()` function to defined a template Convolutional Layer, with a small kernel size of 3x3, a stride of 1, the `same` padding, and the ReLU activation function. This template can be used to create any number of layers in the CNN.
- Next we create the CNN with the first `Input` layer taking in images in array representation of size $28 \times 28 \times 1$ as the images in the Fashion MNIST dataset are of size 28×28 and are in greyscale, with a `Reshape` layer to ensure the images are of this size.
- The first CL is of size 64 with fairly large filters (7×7) which creates a 64 feature maps each of size 28×28.
- Then we add a max pooling layer with a default pool size of 2, so it downscales the feature maps by a factor of 2, resulting in 64 feature maps of size 14×14.
- We repeat the same structure twice, two convolutional layers followed by a max pooling layer, each adding

on more feature maps, deepening the network. For larger images this structure could be repeated using the number of repetitions hyperparameter. The number of filters double as we climb up the CNN toward the output layer (64, 128, 256), as the number of low-level features is usually much lower than the number of high-level features. It is common to double the number of filters after each pooling layer as the pooling layer reduces the size of the feature maps by 2 by default, so we can afford to double the number of filters in the next layer without increasing the computational load.

- Next is the fully connected network, composed of two dense hidden layers and a dense output layer. Since it is a classification task with 10 classes the output layer has 10 layers and uses the softmax activation function to determine the class probabilities of each class. We must flatten the inputs just before the first dense layer as it expects a 1D array of features for each instance. Two dropout layers with a dropout rate of 50% each have also been added to reduce overfitting.

[45]:
```
cnnCls.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.05),
    metrics=["accuracy"],
)
cnnCls.summary()
```

```
Model: "sequential_3"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ reshape (Reshape)               │ (None, 28, 28, 1)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)               │ (None, 28, 28, 64)     │         3,200 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 14, 14, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_3 (Conv2D)               │ (None, 14, 14, 128)    │        73,856 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_4 (Conv2D)               │ (None, 14, 14, 128)    │       147,584 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 7, 7, 128)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_5 (Conv2D)               │ (None, 7, 7, 256)      │       295,168 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_6 (Conv2D)               │ (None, 7, 7, 256)      │       590,080 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_3 (MaxPooling2D)  │ (None, 3, 3, 256)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_2 (Flatten)             │ (None, 2304)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_16 (Dense)                │ (None, 128)            │       295,040 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_17 (Dense)                │ (None, 64)             │         8,256 │
```

```
┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━━━┫
┃ dropout_1 (Dropout)              │ (None, 64)             │             0 ┃
┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━━━┫
┃ dense_18 (Dense)                 │ (None, 10)             │           650 ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━┛
```

```
 Total params: 1,413,834 (5.39 MB)

 Trainable params: 1,413,834 (5.39 MB)

 Non-trainable params: 0 (0.00 B)
```

[46]:
```
(XTrainFull, yTrainFull), (XTest, yTest) = fmnist

cutoff = -5000
XTrain, yTrain = XTrainFull[:cutoff], yTrainFull[:cutoff]
XValid, yValid = XTrainFull[cutoff:], yTrainFull[cutoff:]
XTrain, XValid, XTest = XTrain / 255.0, XValid / 255.0, XTest / 255.0
```

[47]:
```
hist = cnnCls.fit(XTrain, yTrain, epochs=7, validation_data=(XValid, yValid))
```

```
Epoch 1/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 172s 99ms/step
- accuracy: 0.5581 - loss: 1.2113 - val_accuracy: 0.8534 - val_loss: 0.4111
Epoch 2/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 161s 94ms/step
- accuracy: 0.8238 - loss: 0.5049 - val_accuracy: 0.8718 - val_loss: 0.3413
Epoch 3/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 169s 98ms/step
- accuracy: 0.8605 - loss: 0.4066 - val_accuracy: 0.8874 - val_loss: 0.3043
Epoch 4/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 166s 97ms/step
- accuracy: 0.8781 - loss: 0.3537 - val_accuracy: 0.8954 - val_loss: 0.2863
Epoch 5/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 165s 96ms/step
- accuracy: 0.8951 - loss: 0.3128 - val_accuracy: 0.9036 - val_loss: 0.2639
Epoch 6/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 163s 95ms/step
- accuracy: 0.9034 - loss: 0.2843 - val_accuracy: 0.9080 - val_loss: 0.2556
Epoch 7/7
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 167s 97ms/step
- accuracy: 0.9125 - loss: 0.2618 - val_accuracy: 0.9050 - val_loss: 0.2645
```

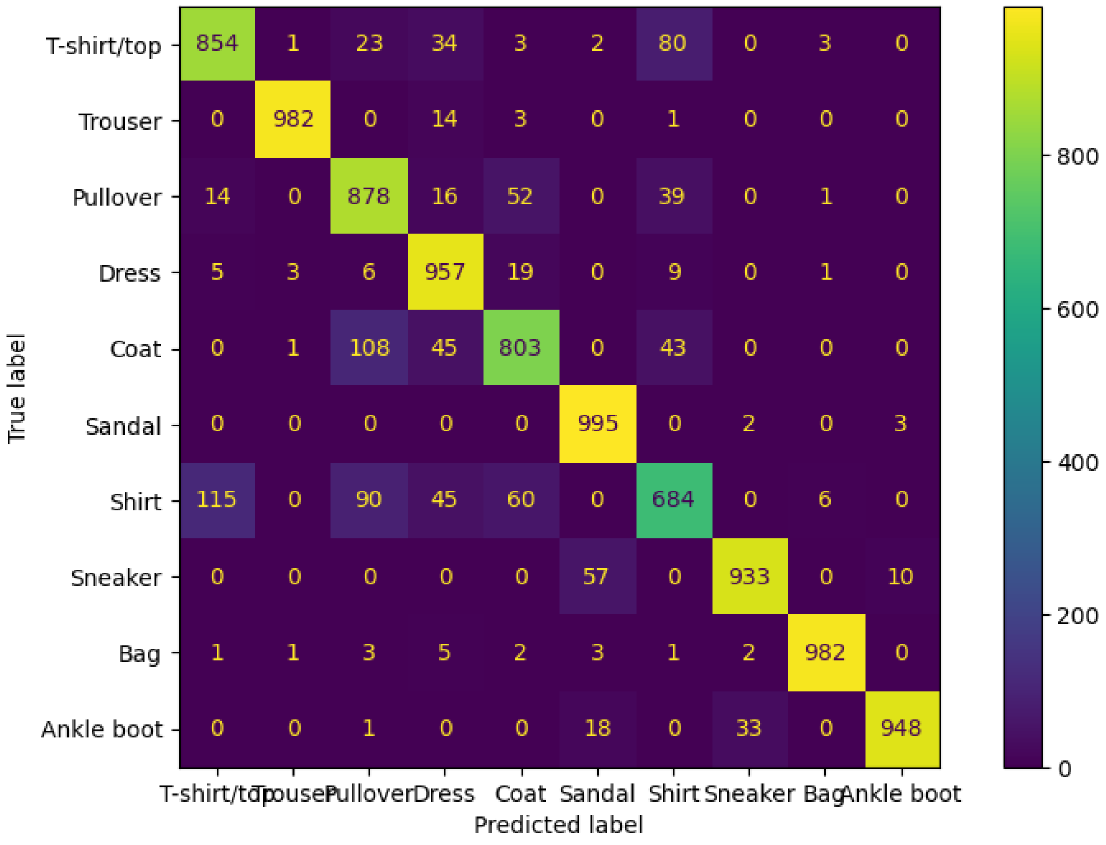
[48]:
```
preds = cnnCls.predict(XTest)
preds = preds.argmax(axis=1)
cnnCls.evaluate(XTest, yTest)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 32ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 10s 32ms/step -
accuracy: 0.9029 - loss: 0.3004
```

```
[48]: [0.29452764987945557, 0.9016000032424927]
```

```
[49]: classStats(yTest, preds, classNames)
```

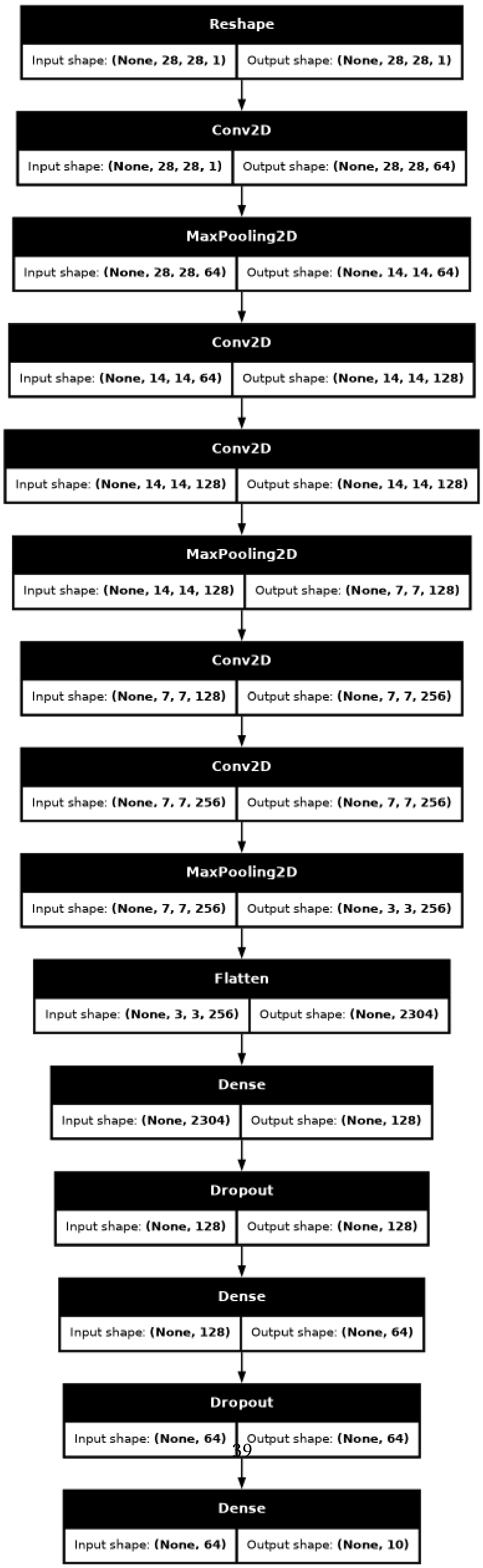
```
              precision    recall  f1-score   support

           0       0.86      0.85      0.86      1000
           1       0.99      0.98      0.99      1000
           2       0.79      0.88      0.83      1000
           3       0.86      0.96      0.90      1000
           4       0.85      0.80      0.83      1000
           5       0.93      0.99      0.96      1000
           6       0.80      0.68      0.74      1000
           7       0.96      0.93      0.95      1000
           8       0.99      0.98      0.99      1000
           9       0.99      0.95      0.97      1000

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000
```



	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	854	1	23	34	3	2	80	0	3	0
Trouser	0	982	0	14	3	0	1	0	0	0
Pullover	14	0	878	16	52	0	39	0	1	0
Dress	5	3	6	957	19	0	9	0	1	0
Coat	0	1	108	45	803	0	43	0	0	0
Sandal	0	0	0	0	0	995	0	2	0	3
Shirt	115	0	90	45	60	0	684	0	6	0
Sneaker	0	0	0	0	0	57	0	933	0	10
Bag	1	1	3	5	2	3	1	2	982	0
Ankle boot	0	0	1	0	0	18	0	33	0	948

```
[50]: tf.keras.utils.plot_model(cnnCls, show_shapes=True, dpi=50)
```

```
[50]:
```


Reshape
Input shape: (None, 28, 28, 1)
Output shape: (None, 28, 28, 1)
Conv2D
Input shape: (None, 28, 28, 1)
Output shape: (None, 28, 28, 64)
MaxPooling2D
Input shape: (None, 28, 28, 64)
Output shape: (None, 14, 14, 64)
Conv2D
Input shape: (None, 14, 14, 64)
Output shape: (None, 14, 14, 128)
Conv2D
Input shape: (None, 14, 14, 128)
Output shape: (None, 14, 14, 128)
MaxPooling2D
Input shape: (None, 14, 14, 128)
Output shape: (None, 7, 7, 128)
Conv2D
Input shape: (None, 7, 7, 128)
Output shape: (None, 7, 7, 256)
Conv2D
Input shape: (None, 7, 7, 256)
Output shape: (None, 7, 7, 256)
MaxPooling2D
Input shape: (None, 7, 7, 256)
Output shape: (None, 3, 3, 256)
Flatten
Input shape: (None, 3, 3, 256)
Output shape: (None, 2304)
Dense
Input shape: (None, 2304)
Output shape: (None, 128)
Dropout
Input shape: (None, 128)
Output shape: (None, 128)
Dense
Input shape: (None, 128)
Output shape: (None, 64)
Dropout
Input shape: (None, 64)
Output shape: (None, 64)
Dense
Input shape: (None, 64)
Output shape: (None, 10)

[ ]:

```
```