Training Neural Networks with Keras

Welcome to the first practical session of the course! In this session, we will learn how to train neural networks with Keras. We will start with a simple example of a feedforward neural network for classification and then we will study the impact of the initialization of the weights on the convergence of the training algorithm.

Keras is a high-level neural network API, built on top of TensorFlow 2.0. It provides a user-friendly interface to build, train and deploy deep learning models. Keras is designed to be modular, fast and easy to use.

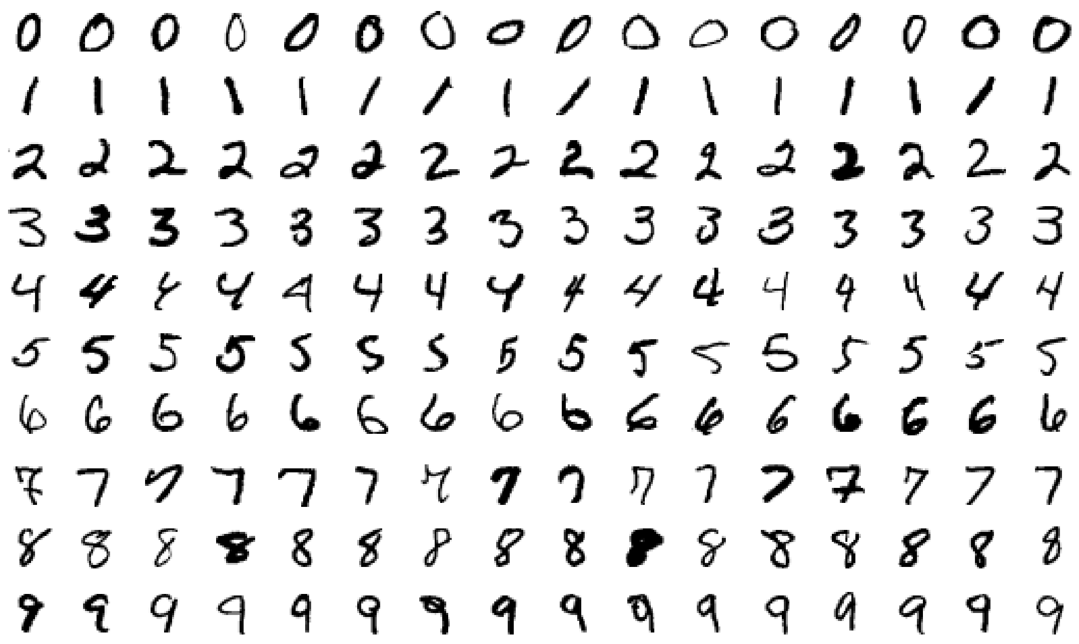
Throughout this course, we will focus on using Keras and TensorFlow for building and training neural networks. However, there are other popular deep learning frameworks such as PyTorch, MXNet, CNTK, etc. that you can also use to build and train neural networks.

In order to use our code on Google Colab, we will need to ensure that any required packages are installed. We will use the following packages in this session:

- `tensorflow`: an open-source library for numerical computation and large-scale machine learning.
- `matplotlib`: a plotting library for the Python programming language and its numerical mathematics extension NumPy.
- `numpy`: a library for scientific computing in Python.
- `scikit-learn`: a machine learning library for the Python programming language.
- `pandas`: a library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Today, we will be working with the famous MNIST dataset. MNIST (Modified National Institute of Standards and Technology) is a database of low resolution images of handwritten digits. The history here is interesting - the dataset was originally created in the 1980s, when researchers from the aforementioned institute collected samples from American Census Bureau employees and high school students. The dataset was then modified in the 1990s (hence the M in MNIST), and has since become a popular benchmark for machine learning algorithms.

The dataset contains images, each of which is a 28x28 grayscale image of a handwritten digit. The goal is to classify each image into one of the 10 possible classes (0-9).



The Scikit-Learn library provides a convenient function to download and load the MNIST dataset. The following cell will download the dataset. Then we will take a look at the shape of the data.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_digits

digits = load_digits()
```

In [2]:
```
digits.images.shape
```

Out[2]:
```
(1797, 8, 8)
```

This means that we have 1797 images, each of which is a 8x8 image. For basic image processing, we will need to flatten the images into a 1D array. In this case, Scikit-Learn has already provided the data in this format too:

In [3]:
```
digits.data.shape
```

Out[3]:
```
(1797, 64)
```

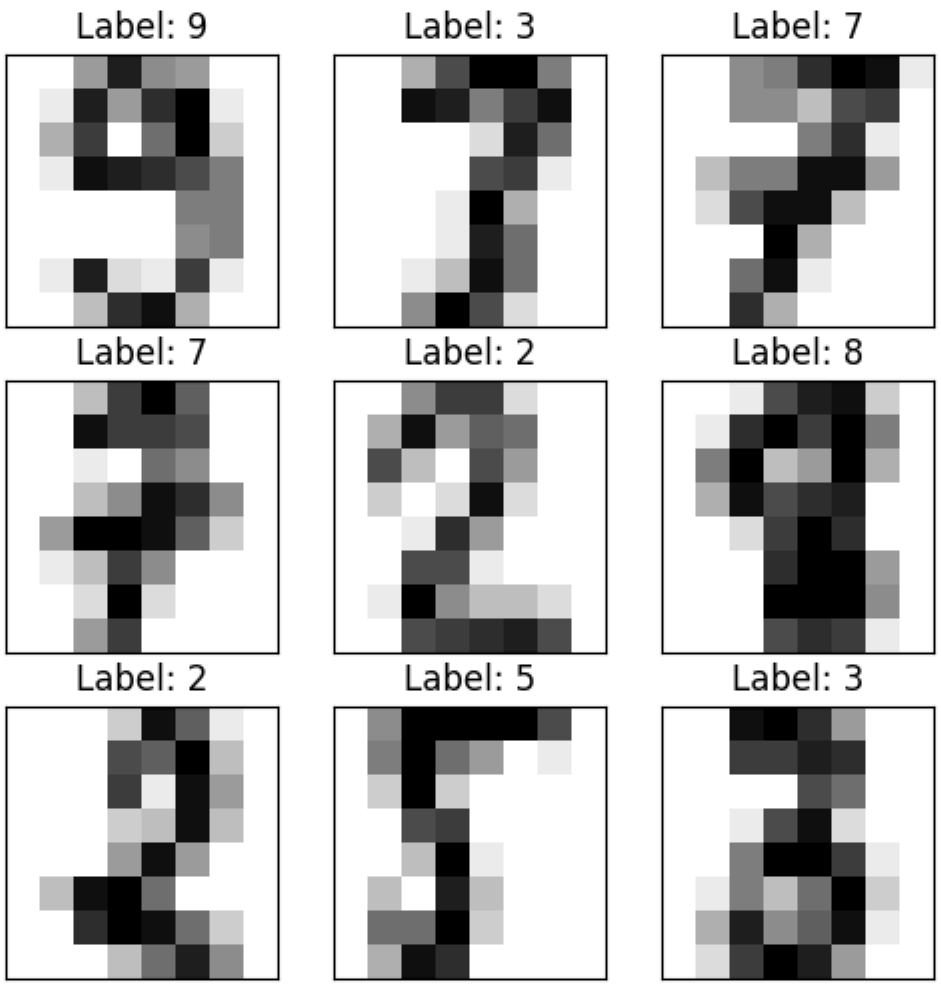
For each image, we also have the corresponding label (or target, or class) in `digits.target`:

In [4]:
```
digits.target.shape
```

Out[4]:
```
(1797,)
```

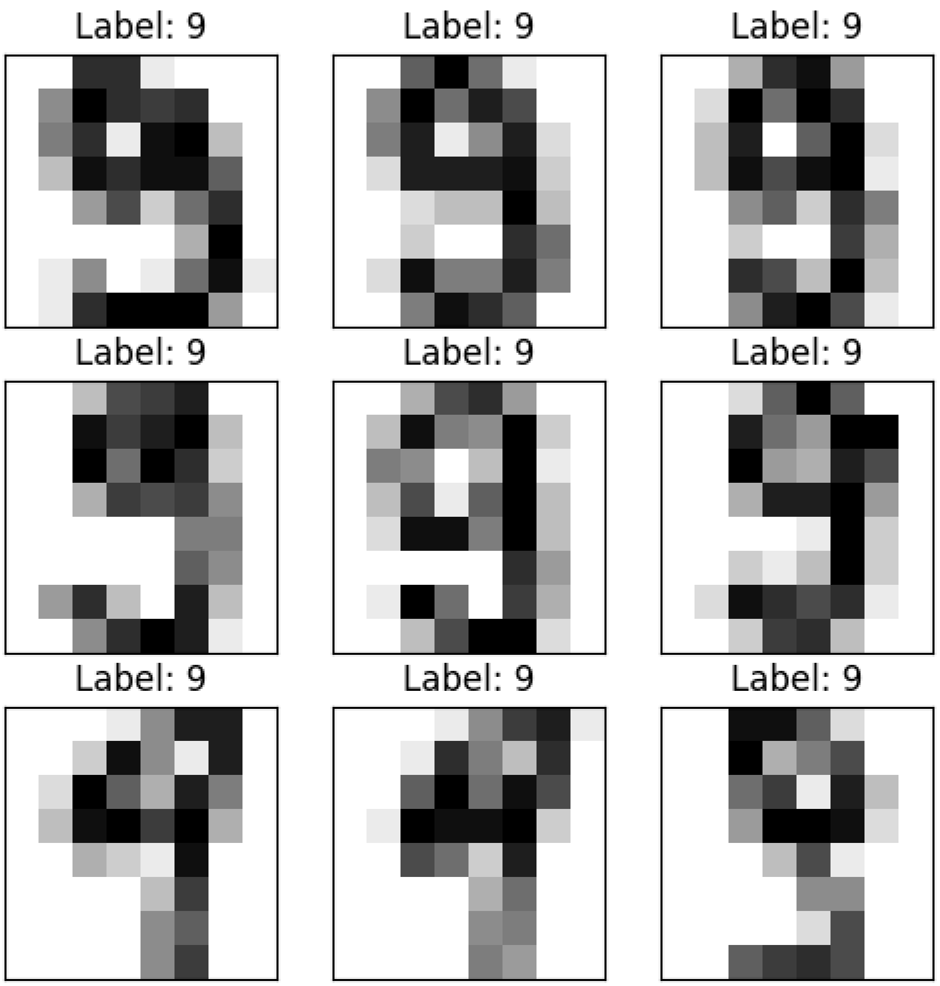
We can take a look at some random images from the dataset. The following cell will select 9 random images and plot them in a 3x3 grid (meaning that you can rerun the cell to see different images).

In [6]:
```python
# Selecting 9 random indices
random_indices = np.random.choice(len(digits.images), 9, replace=False)

# Creating a 3x3 grid plot
fig, axes = plt.subplots(3, 3, figsize=(6, 6))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[random_indices[i]], cmap=plt.cm.gray_r, interpol
    ax.set_title(f"Label: {digits.target[random_indices[i]]}")

    # Removing axis labels
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```



As you can see, these images are very low resolution. This is because they were originally scanned from paper forms, and then scaled down to 8x8 pixels. This is a common problem in machine learning - the quality of the data is often a limiting factor in the performance of the model. In this case, the low resolution of the images makes it difficult to distinguish between some digits, even for humans. For example, the following images are all labelled as 9, but they look very different:

```
In [7]: # Selecting 9 random indices of images labelled as 9
        random_indices = np.random.choice(np.where(digits.target == 9)[0], 9, replac

        # Creating a 3x3 grid plot
        fig, axes = plt.subplots(3, 3, figsize=(6, 6))

        for i, ax in enumerate(axes.flat):
            ax.imshow(digits.images[random_indices[i]], cmap=plt.cm.gray_r, interpol
            ax.set_title(f"Label: {digits.target[random_indices[i]]}")

            # Removing axis labels
            ax.set_xticks([])
            ax.set_yticks([])

        plt.show()
```



While we are plotting the samples as images, remember that our model is only going to see a 1D array of numbers.

Train / Test Split

In order to understand how well our model performs on *new* data, we need to split our dataset into a training set and a test set. The training set will be used to train the model, and the test set will be used to evaluate the performance of the model.

Let's keep some held-out data to be able to measure the generalization performance of our model.

In [8]:
```
from sklearn.model_selection import train_test_split


X_train, X_test, y_train, y_test = train_test_split(
    digits.data,
    digits.target,
    test_size=0.2, # 20% of the data is used for testing
    random_state=42 # Providing a value here means getting the same "random'
)
```

Let's confirm that the data has been split correctly:

In [9]:
```
print(f'X_train shape: {X_train.shape}')
print(f'y_train shape: {y_train.shape}')
print(f'X_test shape: {X_test.shape}')
print(f'y_test shape: {y_test.shape}')
```

```
X_train shape: (1437, 64)
y_train shape: (1437,)
X_test shape: (360, 64)
y_test shape: (360,)
```

This is what we expected to see. It's always good to check as you go, to make sure that you haven't made a mistake somewhere - this is something that working in a notebook like this makes it easy to do.

Preprocessing of the Target Data

The labels that we have are integers between 0 and 9. However, we want to train a neural network to classify the images into one of 10 classes. It can be a little counter-intuitive because we are dealing with numbers, but our classes are not ordinal.

What do we mean by that? Let's imagine we were trying to predict the height of a building (separated into classes) from images. If a given building was actually 10m tall, and our model predicted 9m, we would consider that to be a better prediction than if it predicted 1m. This is because the classes are ordinal - there is meaning in the difference between the classes.

In our case, even though we are dealing with numbers, the classes are not ordinal. If a given image is actually a 9, and our model predicts 8, we would consider that to be just as bad as if it predicted 1. This is because the classes are not ordered, and the difference between the classes is not meaningful.

Because of this, we need to convert our labels from an integer value into a one-hot encoded vector. This means that each label will be represented as a vector of length 10, with a 1 in the position corresponding to the class, and 0s everywhere else. For example, the label 9 would be represented as `[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]`. This is a common way of representing categorical data in machine learning. By doing this, we ensure that our model is taught the correct relationship between the classes.

In [10]:
```python
from tensorflow.keras.utils import to_categorical

print(f'Before one-hot encoding: {y_train[0]}')
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
print(f'After one-hot encoding: {y_train[0]}')
```

```
2025-07-02 19:37:30.490060: I tensorflow/core/util/port.cc:153] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-07-02 19:37:30.916452: I tensorflow/core/platform/cpu_feature_guard.cc:
210] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_
VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compi
ler flags.
Before one-hot encoding: 6
After one-hot encoding: [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

Feed Forward Neural Networks with Keras

Now that we have prepared our data, it's time to build a simple neural network! In this section, we will use the Keras API to build a simple feed forward neural network. We will then train the model on the MNIST dataset, and evaluate its performance on the test set.

In most modern deep learning frameworks, the process of building a model can be broken down into a few steps:

- Define the model architecture: this is where we define the layers of the model, and how they are connected to each other.

- Compile the model: this is where we define the loss function, the optimizer, and the metrics that we want to use to evaluate the model.
- Train the model: this is where we train the model on the training data.

Let's start with defining the model architecture. There are two ways to do this in Keras - the Sequential API and the Functional API. The Sequential API is the simplest way to build a model, and is suitable for most use cases. The Functional API is more flexible, and allows you to build more complex models. We will start with the Sequential API, and then we will look at the Functional API later in the course.

Our simple neural network will be "fully-connected". This means that each neuron in a given layer is connected to every neuron in the next layer. This is also known as a "dense" layer. We will use the `Dense` class from Keras to define our layers.

In [11]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

# Input layer
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL

# Hidden layer
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation

# Output layer
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.summary()
```

```
/home/labber/miniconda3/envs/dsi_participant/lib/python3.9/site-packages/ker
as/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`i
nput_dim` argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Par
dense (Dense)	(None, 64)	4
dense_1 (Dense)	(None, 64)	4
dense_2 (Dense)	(None, 10)	

Total params: 8,970 (35.04 KB)

Trainable params: 8,970 (35.04 KB)

Non-trainable params: 0 (0.00 B)

Congratulations! You have just built your first neural network with Keras. As we can confirm from the `model.summary()` output, our model has 3 layers. The first layer has 64 neurons, the second layer has 64 neurons, and the output layer has 10 neurons. The output layer uses the softmax activation function, which is commonly used for multi-class classification problems. The other layers use the ReLU activation function, which is commonly used for hidden layers in neural networks.

Next, we need to compile the model. This is where we define the loss function, the optimizer, and the metrics that we want to use to evaluate the model. We will use the `compile` method of the model to do this.

In [12]:
```
model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer='sgd', # Optimizer
    metrics=['accuracy'] # Metrics to evaluate the model
)
```

Because we are predicting which class a sample belongs to, we will use the `categorical_crossentropy` function. This loss function is commonly used for multi-class classification problems.

For our optimizer, we are using the standard stochastic gradient descent (SGD) algorithm. This is a simple optimizer that works well for many problems. We will look at more advanced optimizers later in the course.

Finally, we are using the `accuracy` metric to evaluate the model. This is a common metric for classification problems, and it is simply the fraction of samples that are correctly classified. This is an easier metric for us to understand, but it's not quite as useful for actually training the model (for example, it doesn't tell us how "confident" the model is in its predictions).

Now that we have (a) defined the model architecture and (b) compiled the model, we are ready to train the model. We will use the `fit` method of the model to do this.

In [13]:
```
model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)
```

```
Epoch 1/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - accuracy: 0.3543 - loss: 3.1230 -
val_accuracy: 0.8056 - val_loss: 0.6428
Epoch 2/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8475 - loss: 0.4969 -
val_accuracy: 0.8646 - val_loss: 0.4300
Epoch 3/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8928 - loss: 0.3678 -
val_accuracy: 0.8958 - val_loss: 0.3420
Epoch 4/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.9367 - loss: 0.2535 -
val_accuracy: 0.9201 - val_loss: 0.3121
Epoch 5/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9620 - loss: 0.1757 -
val_accuracy: 0.9132 - val_loss: 0.2677
```

```
Out[13]: <keras.src.callbacks.history.History at 0x77355418cc40>
```

We have now trained our model! We can see that the model has been trained for 5 epochs, and the loss and accuracy have been printed for each epoch. We can also see that the model has been evaluated on the validation data at the end of each epoch. This is useful for us to see how the model is performing on data that it hasn't seen during training.

Once the model is trained, it's time to evaluate the model on the test set. We can use the `evaluate` method of the model to do this. If you were building a model for a real-world application, this is the very last thing you would do, and the result here would be the figure you'd report in your paper or presentation.

```
In [14]: loss, accuracy = model.evaluate(X_test, y_test)

         print(f'Loss:     {loss:.2f}')
         print(f'Accuracy: {accuracy*100:.2f}%')
```

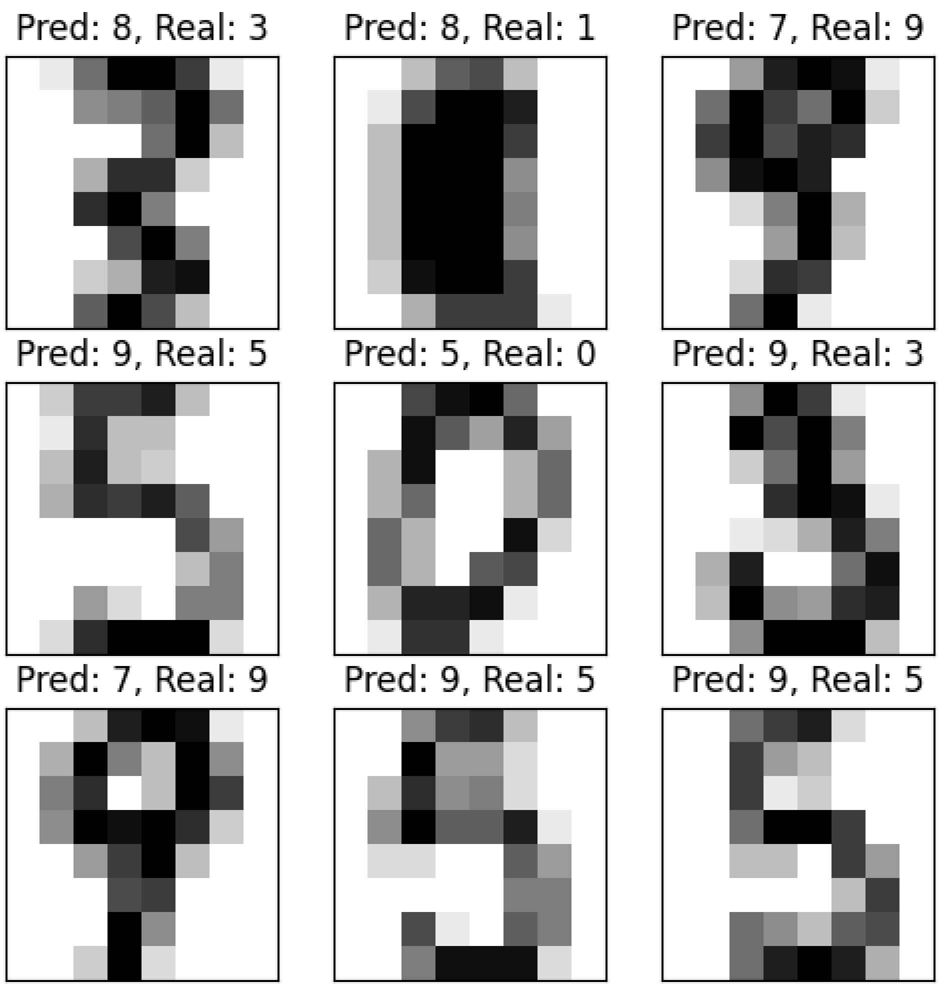
```
12/12 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9411 - loss: 0.1847
Loss:     0.22
Accuracy: 93.06%
```

Hopefully you have achieved an accuracy of around 95%. This is pretty good, but we can do better! In the next section, we will look at how we can improve the performance of our model by using a more advanced optimizer. But before we get there, let's do one other thing - let's look at the predictions that our model is making on the test set. When you are building a model, it's often useful to have a look at some of the examples your model is getting wrong. Sometimes this can reveal problems with the data, or it can give you ideas for how to improve your model.

```
In [15]: # Get the predictions for the test data
         predictions = model.predict(X_test)

         # Get the index of the largest probability (i.e. the predicted class)
         predicted_classes = np.argmax(predictions, axis=1)
```

```
true_classes = np.argmax(y_test, axis=1)
misclassified_indices = np.where(predicted_classes != true_classes)[0]

# Get the misclassified samples themselves
misclassified_samples = X_test[misclassified_indices]
misclassified_labels = np.argmax(y_test[misclassified_indices], axis=1)

# Pick 9 random misclassified samples
random_indices = np.random.choice(len(misclassified_indices), 9, replace=Fal

fig, axes = plt.subplots(3, 3, figsize=(6, 6))
for i, ax in enumerate(axes.flat):
    ax.imshow(misclassified_samples[random_indices[i]].reshape(8, 8), cmap=p
    ax.set_title(f"Pred: {predicted_classes[misclassified_indices[random_inc

    # Removing axis labels
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
```



What do you think? Would you have made the same mistakes as the model? Determining whether the mistakes are "understandable" is a rough way of seeing if you could improve the model further, or if this is the best you can do with the data you have.

b) Exercises: Impact of the Optimizer

In this section, you will play around with the optimizer and see how it affects the performance of the model. We will start with the standard SGD optimizer, and then we will look at more advanced optimizers.

1. Try decreasing the learning rate of the SGD optimizer by a factor of 10, or 100. What do you observe?
2. Try increasing the learning rate of the SGD optimizer. What happens?
3. The SGD optimizer has a momentum parameter. In a nutshell, this parameter controls how much the gradient from the previous step affects the current step. Try enabling momentum in the SGD optimizer with a value of 0.9. What happens?

Notes:

The keras API documentation is available at:

https://www.tensorflow.org/api_docs/python/tf/keras

It is also possible to learn more about the parameters of a class by using the question mark: type and evaluate:

```
optimizers.SGD?
```
in a jupyter notebook cell.

It is also possible to type the beginning of a function call / constructor and type "shift-tab" after the opening paren:

```
optimizers.SGD(<shift-tab>
```

In [ ]:
```
# 1. Decreasing the learning rate
from tensorflow.keras.optimizers import SGD

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

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer= SGD(learning_rate = 0.0001), # Optimizer (default learning ra
    metrics=['accuracy'] # Metrics to evaluate the model
)
```

```
model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)

loss, accuracy = model.evaluate(X_test, y_test)

print(f'Loss:     {loss:.2f}')
print(f'Accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.0482 - loss: 6.5278 -
val_accuracy: 0.0590 - val_loss: 5.5437
Epoch 2/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.0861 - loss: 4.8985 -
val_accuracy: 0.1076 - val_loss: 4.7051
Epoch 3/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.1235 - loss: 4.3277 - v
al_accuracy: 0.1389 - val_loss: 4.0946
Epoch 4/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.1288 - loss: 3.8533 - v
al_accuracy: 0.1667 - val_loss: 3.5815
Epoch 5/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.1548 - loss: 3.2558 - v
al_accuracy: 0.1806 - val_loss: 3.1616
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1583 - loss: 3.1910
Loss:     3.20
Accuracy: 16.39%
```

In [27]:
```
# 2. Increasing the learning rate

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer= SGD(learning_rate = 0.2), # Optimizer #default learning rate
    metrics=['accuracy'] # Metrics to evaluate the model
)

model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)

loss, accuracy = model.evaluate(X_test, y_test)
```

```
print(f'Loss:     {loss:.2f}')
print(f'Accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.1244 - loss: 37.3898 -
val_accuracy: 0.1076 - val_loss: 2.3032
Epoch 2/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.0955 - loss: 2.3030 - v
al_accuracy: 0.1076 - val_loss: 2.3062
Epoch 3/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.0758 - loss: 2.3086 -
val_accuracy: 0.1076 - val_loss: 2.3067
Epoch 4/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.1195 - loss: 2.3047 -
val_accuracy: 0.0833 - val_loss: 2.3098
Epoch 5/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.1004 - loss: 2.3012 -
val_accuracy: 0.0833 - val_loss: 2.3097
12/12 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.1054 - loss: 2.3114
Loss:     2.31
Accuracy: 9.72%
```

In [ ]:

```
# 3. SGD with momentum

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer= SGD(learning_rate = 0.01 momentum = ), # Optimizer #default l
    metrics=['accuracy'] # Metrics to evaluate the model
)

model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)

loss, accuracy = model.evaluate(X_test, y_test)

print(f'Loss:     {loss:.2f}')
print(f'Accuracy: {accuracy*100:.2f}%')
```

Next, let's try a more advanced optimizer. Adam is likely the most popular optimizer for deep learning. It is an adaptive learning rate optimizer, which means that it automatically adjusts the learning rate based on how the training is going. This can be very useful, as it means that we don't need to manually tune the learning rate. Let's see how it performs on our model.

1. Replace the SGD optimizer by the Adam optimizer from keras and run it with the default parameters.
2. Add another hidden layer with ReLU activation and 64 neurons. Does it improve the model performance?

In [ ]:
```python
# Adam optimizer
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer= SGD(learning_rate = 0.2), # Optimizer #default learning rate
    metrics=['accuracy'] # Metrics to evaluate the model
)

model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)

loss, accuracy = model.evaluate(X_test, y_test)

print(f'Loss:     {loss:.2f}')
print(f'Accuracy: {accuracy*100:.2f}%')
```

In [28]:
```python
# Extra hidden layer

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(64,))) # 64 neurons, ReL
model.add(Dense(64, activation='relu')) # 64 neurons, ReLU activation
model.add(Dense(64, activation='relu')) # extra hidden layer with 64 neurons
model.add(Dense(10, activation='softmax')) # 10 neurons, softmax activation

model.compile(
    loss='categorical_crossentropy', # Loss function
    optimizer= SGD(learning_rate = 0.2), # Optimizer #default learning rate
    metrics=['accuracy'] # Metrics to evaluate the model
)

model.fit(
    X_train, # Training data
    y_train, # Training labels
    epochs=5, # Number of epochs
    batch_size=32, # Number of samples per batch
    validation_split=0.2 # Use 20% of the data for validation
)
```

```
loss, accuracy = model.evaluate(X_test, y_test)

print(f'Loss:     {loss:.2f}')
print(f'Accuracy: {accuracy*100:.2f}%')
```

```
/home/labber/miniconda3/envs/dsi_participant/lib/python3.9/site-packages/ker
as/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`i
nput_dim` argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 6s 66ms/step - accuracy: 0.0924 - loss: nan - val
_accuracy: 0.1215 - val_loss: nan
Epoch 2/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.0822 - loss: nan - val
_accuracy: 0.1215 - val_loss: nan
Epoch 3/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.0889 - loss: nan - val
_accuracy: 0.1215 - val_loss: nan
Epoch 4/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 39ms/step - accuracy: 0.0865 - loss: nan - val
_accuracy: 0.1215 - val_loss: nan
Epoch 5/5
36/36 ━━━━━━━━━━━━━━━━━━━━ 2s 35ms/step - accuracy: 0.1026 - loss: nan - val
_accuracy: 0.1215 - val_loss: nan
12/12 ━━━━━━━━━━━━━━━━━━━━ 2s 45ms/step - accuracy: 0.0877 - loss: nan
Loss:     nan
Accuracy: 9.17%
```

Exercises: Forward Pass and Generalization

Let's look in more detail at how the model makes predictions on the test set. We will walk through each step of making predictions, examining exactly what's going on.

To start, we will apply our model to the test set, and look at what we get as output:

In [29]:
```
predictions_tf = model(X_test)
predictions_tf[:5]
```

Out[29]:
```
<tf.Tensor: shape=(5, 10), dtype=float32, numpy=
array([[nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan, nan, nan, nan, nan]], dtype=float32)>
```

In [30]:
```
type(predictions_tf), predictions_tf.shape
```

Out[30]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([360, 10]))
```

The raw output of the model is a tensor of shape `(360, 10)`. This means that we have 360 samples, and for each sample we have 10 values. Each of these values represents the probability that the sample belongs to a given class. This means that we have 10 probabilities for each sample, and the sum of these probabilities is 1. We can confirm this by summing the probabilities for each sample:

In [31]:
```
import tensorflow as tf

tf.reduce_sum(predictions_tf, axis=1)[:5]
```

Out[31]:
```
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([nan, nan, nan, nan, na
n], dtype=float32)>
```

...okay, there might be a small rounding error here and there. This is to do with how floating point numbers are represented in computers, and it's not something we need to worry about for now.

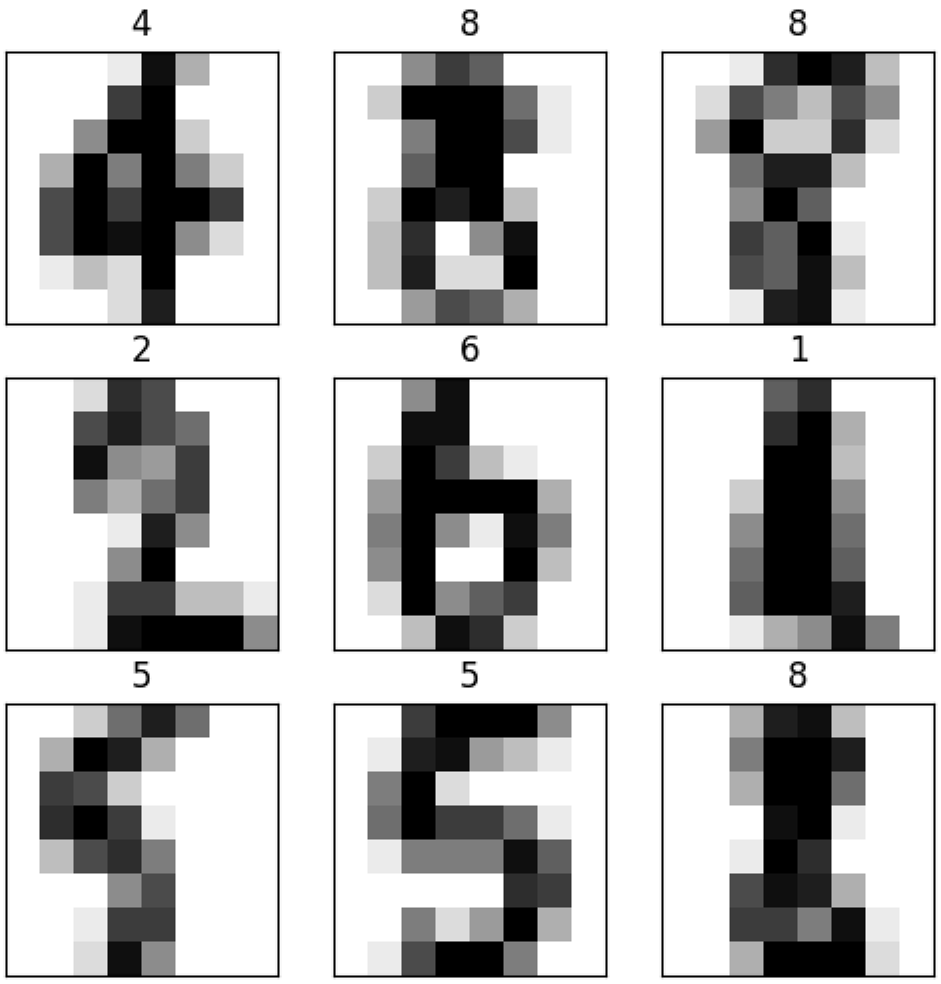
We can also extract the label with the highest probability using the tensorflow API:

In [32]:
```
predicted_labels_tf = tf.argmax(predictions_tf, axis=1)
predicted_labels_tf[:5]
```

Out[32]:
```
<tf.Tensor: shape=(5,), dtype=int64, numpy=array([0, 0, 0, 0, 0])>
```

One helpful aspect of this approach is that we don't just get the prediction, but also a sense of how confident the model is in its prediction. To see this in practice, let's take a look at some of the predictions the model is highly confident about (i.e. a lot of the probability mass is on one class):

In [33]:
```
# Get the values corresponding to the predicted labels for each sample
predicted_values_tf = tf.reduce_max(predictions_tf, axis=1)

# Get the indices of the samples with the highest predicted values
most_confident_indices_tf = tf.argsort(predicted_values_tf, direction='DESCE

# Get the 9 most confident samples
most_confident_samples_tf = X_test[most_confident_indices_tf]

# Get the true labels for the 9 most confident samples
most_confident_labels_tf = np.argmax(y_test[most_confident_indices_tf], axis

# Plot the 9 most confident samples
fig, axes = plt.subplots(3, 3, figsize=(6, 6))

for i, ax in enumerate(axes.flat):
    ax.imshow(most_confident_samples_tf[i].reshape(8, 8), cmap=plt.cm.gray_r
    ax.set_title(f"{most_confident_labels_tf[i]}")
    # Removing axis labels
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```



Impact of Initialization

Let's study the impact of a bad initialization when training a deep feed forward network.

By default, Keras dense layers use the "Glorot Uniform" initialization strategy to initialize the weight matrices:

- each weight coefficient is randomly sampled from [-scale, scale]
- scale is proportional to $\frac{1}{\sqrt{n_{in} + n_{out}}}$

This strategy is known to work well to initialize deep neural networks with "tanh" or "relu" activation functions and then trained with standard SGD.

To assess the impact of initialization let us plug an alternative init scheme into a 2 hidden layers networks with "tanh" activations. For the sake of the example let's use normal distributed weights with a manually adjustable scale (standard deviation) and see the impact the scale value:

In [41]:
```
from tensorflow.keras import initializers
from tensorflow.keras import optimizers

input_dim = 64
hidden_dim = 64
output_dim = 10

normal_init = initializers.TruncatedNormal(stddev=0.01, seed=42)

model = Sequential()
model.add(Dense(hidden_dim, input_dim=input_dim, activation="tanh",
                kernel_initializer=normal_init))
model.add(Dense(hidden_dim, activation="tanh",
                kernel_initializer=normal_init))
model.add(Dense(output_dim, activation="softmax",
                kernel_initializer=normal_init))

model.compile(optimizer=optimizers.SGD(learning_rate=0.1),
              loss='categorical_crossentropy', metrics=['accuracy'])
```

In [42]:
```
model.layers
```

Out[42]:
```
[<Dense name=dense_46, built=True>,
 <Dense name=dense_47, built=True>,
 <Dense name=dense_48, built=True>]
```

Let's have a look at the parameters of the first layer after initialization but before any training has happened:

In [43]:
```
model.layers[0].weights
```

Out[43]:
```
[<Variable path=sequential_15/dense_46/kernel, shape=(64, 64), dtype=float3
2, value=[[ 0.00015817 -0.01590087  0.00103594 ...  0.00962818  0.00624957
    0.00994726]
  [ 0.0081879   0.00756818 -0.00668142 ...  0.01084459 -0.00317478
   -0.00549116]
  [-0.00086618 -0.00287623  0.00391693 ...  0.00064558 -0.00420471
    0.00174566]
  ...
  [-0.0029006  -0.0091218   0.00804327 ... -0.01407086  0.00952832
   -0.01348555]
  [ 0.00375078  0.00967842  0.00098119 ... -0.00413454  0.01695471
    0.00025196]
  [ 0.00459809  0.01223094 -0.00213172 ...  0.01246831 -0.00714749
   -0.00868595]]>,
 <Variable path=sequential_15/dense_46/bias, shape=(64,), dtype=float32, va
lue=[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]>]
```

```
In [37]: w = model.layers[0].weights[0].numpy()
         w
```

```
Out[37]: array([[ 0.00015817, -0.01590087,  0.00103594, ...,  0.00962818,
                  0.00624957,  0.00994726],
                [ 0.0081879 ,  0.00756818, -0.00668142, ...,  0.01084459,
                 -0.00317478, -0.00549116],
                [-0.00086618, -0.00287623,  0.00391693, ...,  0.00064558,
                 -0.00420471,  0.00174566],
                ...,
                [-0.0029006 , -0.0091218 ,  0.00804327, ..., -0.01407086,
                  0.00952832, -0.01348555],
                [ 0.00375078,  0.00967842,  0.00098119, ..., -0.00413454,
                  0.01695471,  0.00025196],
                [ 0.00459809,  0.01223094, -0.00213172, ...,  0.01246831,
                 -0.00714749, -0.00868595]], dtype=float32)
```

```
In [38]: w.std()
```

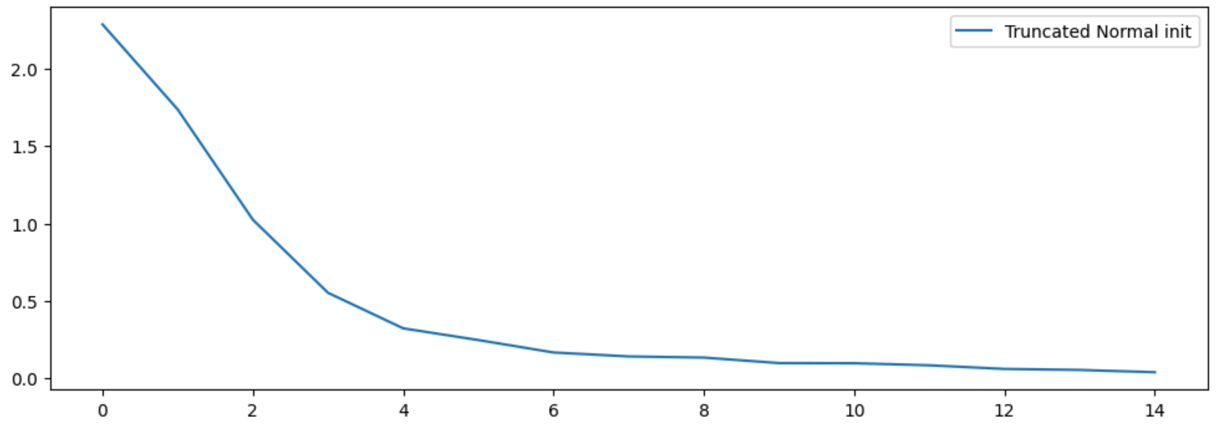
```
Out[38]: 0.008835949
```

```
In [39]: b = model.layers[0].weights[1].numpy()
         b
```

```
Out[39]: 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.], dtype=float32)
```

```
In [40]: history = model.fit(X_train, y_train, epochs=15, batch_size=32)

         plt.figure(figsize=(12, 4))
         plt.plot(history.history['loss'], label="Truncated Normal init")
         plt.legend();
```

```
Epoch 1/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.1629 - loss: 2.2973
Epoch 2/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.4229 - loss: 1.9542
Epoch 3/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.6573 - loss: 1.1518
Epoch 4/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.8381 - loss: 0.6327
Epoch 5/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9239 - loss: 0.3309
Epoch 6/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9329 - loss: 0.2693
Epoch 7/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9670 - loss: 0.1566
Epoch 8/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9724 - loss: 0.1220
Epoch 9/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9677 - loss: 0.1315
Epoch 10/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9728 - loss: 0.0963
Epoch 11/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.9809 - loss: 0.0912
Epoch 12/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9839 - loss: 0.0751
Epoch 13/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9896 - loss: 0.0542
Epoch 14/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9867 - loss: 0.0515
Epoch 15/15
45/45 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9952 - loss: 0.0380
```



Once the model has been fit, the weights have been updated and notably the biases are no longer 0:

In [ ]:
```
model.layers[0].weights
```

Questions:

- Try the following initialization schemes and see whether the SGD algorithm can successfully train the network or not:

    - a very small e.g. `stddev=1e-3`
    - a larger scale e.g. `stddev=1` or `10`
    - initialize all weights to 0 (constant initialization)
- What do you observe? Can you find an explanation for those outcomes?

- Are more advanced solvers such as SGD with momentum or Adam able to deal better with such bad initializations?

In [ ]:
```
# Your code here
```