CARTE-Enbridge Bootcamp

Lab 2-1

Introduction to TensorFlow

In this lab, we will go over the basics of using TensorFlow to build and train a neural network. TensorFlow is one of the two most popular deep learning frameworks (the other being PyTorch). It is developed by Google and is used in many of their products.

Using the sub-module Keras, we will build a simple neural network to classify images of handwritten digits from the MNIST dataset.

First, we will import TensorFlow and check the version:

In [1]:

```python
# Import TensorFlow
import tensorflow as tf
import tensorflow.keras as keras
from sklearn.metrics import classification_report, ConfusionMatrixDisplay

print("Using TensorFlow version", tf.__version__)

# Use GPU, if available
device_name = tf.test.gpu_device_name()
if device_name != "/device:GPU:0":
    print("GPU device not found")
else:
    print(f"Found GPU at: {device_name}")
```

```
2023-11-07 17:12:49.543210: I tensorflow/core/util/port.cc:110] 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`.
2023-11-07 17:12:49.808585: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in 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 compiler flags.
Using TensorFlow version 2.13.1
Found GPU at: /device:GPU:0
```

```
2023-11-07 17:12:51.747167: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.881429: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.881588: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.935949: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.936692: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.936785: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:51.936850: I tensorflow/core/common_runtime/gpu/gpu_device.
cc:1639] Created device /device:GPU:0 with 7826 MB memory:  -> device: 0, na
me: NVIDIA GeForce RTX 3060, pci bus id: 0000:02:00.0, compute capability:
8.6
```

Now we will download the MNIST dataset. Keras provides a convenient function for this. The dataset is already split into training and test sets. We will use the training set to train the model and the test set to evaluate the model's performance on unseen data.

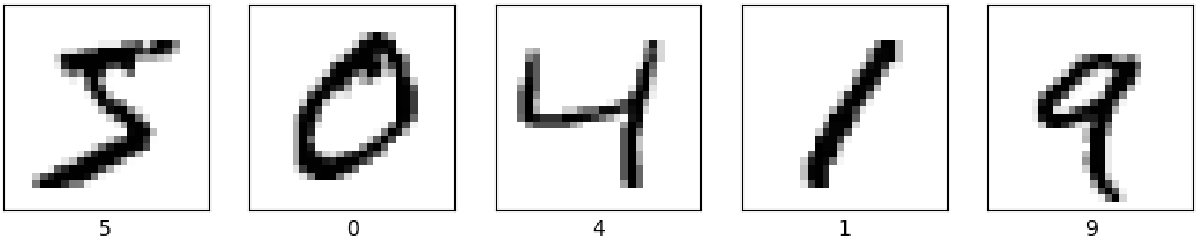
In [2]:
```
# Download and prepare the MNIST dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Before we move forward, it's always helpful to visualize our data to get a sense of what we're working with. Let's display a few of the images from the training set along with their corresponding labels:

In [3]:
```
# Visualize the first five images from the training dataset
import matplotlib.pyplot as plt

%matplotlib inline
plt.figure(figsize=(10, 10))
```

```python
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(y_train[i])
plt.show()
```



Let's establish some basic information about our dataset, while we're at it:

In [4]:
```python
input_shape = x_train[0].shape
num_classes = len(set(y_train))
print("There are", num_classes, "classes in our dataset")
print("The shape of each image is", input_shape)
```

```
There are 10 classes in our dataset
The shape of each image is (28, 28)
```

When working with neural networks, it is important to normalize the data so that the values all fall between 0 and 1. This is done by dividing each value by the maximum value in the dataset, which is 255 in the case of the MNIST dataset.

We will also one-hot encode the labels. One-hot encoding is a process where we replace each label with a vector of length equal to the number of possible classes. It's called 'one-hot' because only one of the values in the vector is 1 (aka 'hot'), and the rest are 0.

Part of the reason we do this here is that even though the labels are numbers, they are not ordinal. That is, the fact that the label for a 3 is greater than the label for a 2 does not mean that a 3 is more similar to a 2 than it is to a 4. In fact, the labels are categorical, not numerical. One-hot encoding allows us to treat the labels as categorical, which is important for the loss function we will use later on.

This means that we will convert the labels from a single number to a vector whose length is equal to the number of possible classes. The vector will be all 0s except for the index corresponding to the label, which will be 1. For example, if the label is 3, the one-hot encoded label will be [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

In [5]:
```python
# Normalize the data so that the values all fall between 0 and 1
x_train = x_train / x_train.max()
```

```
x_test = x_test / x_test.max()
```

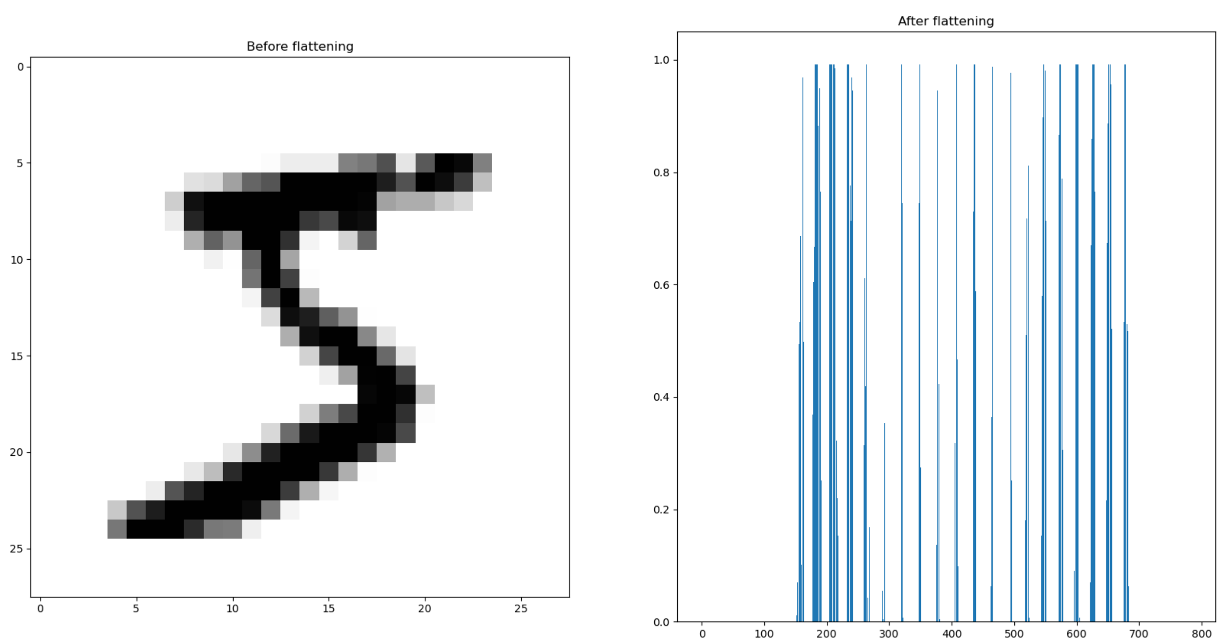
In [6]:
```
# One-hot encode the labels
print(f"Before one-hot encoding: {y_train[0]}")
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
print(f"After one-hot encoding:\n {y_train[0]}")
```

```
Before one-hot encoding: 5
After one-hot encoding:
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

The last thing we will do before getting to building our actual model is reshape our input. The images in the MNIST dataset are 28x28 pixels, but we need to flatten them into a single vector of length 784 in order to feed them into our neural network. Later on in the lab, we'll look at a neural network that can accept a 2D image, but for now we will keep it simple. We will do this using the reshape() function.

In [7]:
```
# Flatten the input images
x_train_flattened = x_train.reshape(60000, 28 * 28)
x_test_flattened = x_test.reshape(10000, 28 * 28)

fig, ax = plt.subplots(1, 2, figsize=(20, 10))
ax[0].imshow(x_train[0], cmap=plt.cm.binary)
ax[0].set_title("Before flattening")
ax[1].bar(range(28 * 28), x_train_flattened[0])
ax[1].set_title("After flattening")
plt.show()
```



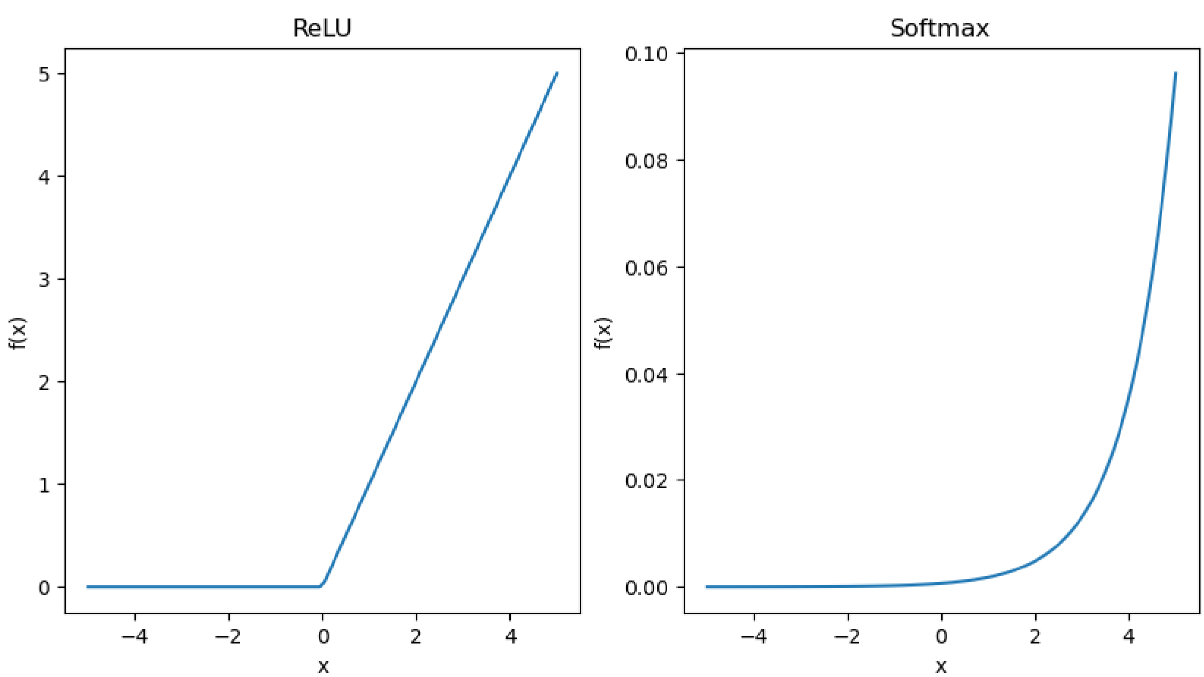
Okay, now we're ready to build our model! The Keras Sequential API makes this very easy. We just define one layer of our neural network at a time, starting with the input layer. The first layer we add must specify the input shape, which is (28*28,) in our case.

We will also be using two activation functions we haven't seen before - inside the network, we will use ReLU, and at the end we will use Softmax. ReLU stands for Rectified Linear Unit, and it is defined as f(x) = max(0, x). It is a very simple function, but it is very effective in neural networks.

Softmax is a function that takes a vector of numbers and outputs a vector of the same length, where each value is between 0 and 1 and the sum of the values is 1. It is often used as the activation function for the output layer of a neural network, because it allows us to interpret the output as a probability distribution over the possible classes.

In [8]:
```
import numpy as np

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
# Plot the ReLU function
x = np.linspace(-5, 5, 100)
ax[0].plot(x, np.maximum(0, x))
ax[0].set_title("ReLU")
ax[0].set_xlabel("x")
ax[0].set_ylabel("f(x)")
# Plot the softmax function
ax[1].plot(x, np.exp(x) / np.sum(np.exp(x)))
ax[1].set_title("Softmax")
ax[1].set_xlabel("x")
ax[1].set_ylabel("f(x)")
plt.show()
```



In [9]:
```
# First, define the input to the neural network
input = keras.layers.Input(shape=(28 * 28,)) # Aka (784,)
```

In [10]:
```
# Next, define the first hidden layer
hidden1 = keras.layers.Dense(128, activation="relu")(input) # 128 neurons, R
```

```
2023-11-07 17:12:53.265289: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.265490: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.265573: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.265773: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.265863: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.265939: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.266060: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.266179: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-07 17:12:53.266232: I tensorflow/core/common_runtime/gpu/gpu_device.
cc:1639] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 78
26 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id: 000
0:02:00.0, compute capability: 8.6
```

In [11]:
```
# Next, define the second hidden layer
hidden2 = keras.layers.Dense(64, activation="relu")(hidden1) # 64 neurons, R
```

In [12]:
```
# Finally, define the output layer
output = keras.layers.Dense(10, activation="softmax")(hidden2) # 10 neurons,
```

In [13]:
```
# Now we can define the model, specifying the input and output layers
model = keras.models.Model(inputs=input, outputs=output)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0         
                                                                 
 dense (Dense)               (None, 128)               100480    
                                                                 
 dense_1 (Dense)             (None, 64)                8256      
                                                                 
 dense_2 (Dense)             (None, 10)                650       
                                                                 
=================================================================
Total params: 109386 (427.29 KB)
Trainable params: 109386 (427.29 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [14]:
```
# Visualize the model
keras.utils.plot_model(model, show_shapes=True)
```

```
You must install pydot (`pip install pydot`) and install graphviz (see instr
uctions at https://graphviz.gitlab.io/download/) for plot_model to work.
```

It really is that easy! Now we just need to compile the model, specifying the optimizer, loss function, and metrics we want to use. We will use the standard stochastic gradient descent optimizer, the categorical cross-entropy loss function, and the accuracy metric. We use categorical cross-entropy because we have more than two classes. If we had only two classes, we would use binary cross-entropy.

In [15]:
```
optimizer = "sgd" # Stochastic gradient descent - the foundation of most neu
loss = "categorical_crossentropy" # The loss function we will use to train t
metrics = ["accuracy"] # The metric we will use to evaluate the model - accu
```

In [16]:
```
# Compile the model
model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=metrics
)
```

Now we can train the model. We just need to specify the training data, the number of epochs, and the batch size. The batch size is the number of training examples that are fed into the model at once. The number of epochs is the number of times the model will see the entire training set.

In [17]:
```python
# Train the model
model.fit(x_train_flattened, y_train, epochs=5, batch_size=32, validation_sp
```

```
Epoch 1/5
2023-11-07 17:12:54.260584: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:606] TensorFloat-32 will be used for the matrix multiplication.
This will only be logged once.
2023-11-07 17:12:54.354385: I tensorflow/compiler/xla/service/service.cc:16
8] XLA service 0x7f1580067ec0 initialized for platform CUDA (this does not g
uarantee that XLA will be used). Devices:
2023-11-07 17:12:54.354412: I tensorflow/compiler/xla/service/service.cc:17
6]   StreamExecutor device (0): NVIDIA GeForce RTX 3060, Compute Capability
8.6
2023-11-07 17:12:54.380663: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_dnn.cc:432] Loaded cuDNN version 8800
2023-11-07 17:12:54.452319: I tensorflow/tsl/platform/default/subprocess.cc:
304] Start cannot spawn child process: No such file or directory
  44/1688 [..............................] - ETA: 1s - loss: 2.2153 - accura
cy: 0.1996
2023-11-07 17:12:54.515422: I ./tensorflow/compiler/jit/device_compiler.h:18
6] Compiled cluster using XLA!  This line is logged at most once for the lif
etime of the process.
1688/1688 [==============================] - 3s 1ms/step - loss: 0.6861 - ac
curacy: 0.8175 - val_loss: 0.2895 - val_accuracy: 0.9180
Epoch 2/5
1688/1688 [==============================] - 2s 1ms/step - loss: 0.3132 - ac
curacy: 0.9103 - val_loss: 0.2281 - val_accuracy: 0.9327
Epoch 3/5
1688/1688 [==============================] - 2s 1ms/step - loss: 0.2599 - ac
curacy: 0.9252 - val_loss: 0.1960 - val_accuracy: 0.9440
Epoch 4/5
1688/1688 [==============================] - 2s 1ms/step - loss: 0.2246 - ac
curacy: 0.9354 - val_loss: 0.1719 - val_accuracy: 0.9522
Epoch 5/5
1688/1688 [==============================] - 2s 1ms/step - loss: 0.1976 - ac
curacy: 0.9417 - val_loss: 0.1578 - val_accuracy: 0.9610
```

Out[17]:
```
<keras.src.callbacks.History at 0x7f16a01aaa10>
```

At the end of each epoch, the model will evaluate its performance on the validation set - this is a held out part of our training data that allows us to monitor how well the model is training. It gives us an idea of how well the model will generalize to unseen data. But we still need to evaluate the model on the test set to get a final measure of its performance:

In [18]:
```python
# Evaluate the model on the test set
loss, accuracy = model.evaluate(x_test_flattened, y_test)
print(f"Test loss: {loss:.2f}")
print(f"Test accuracy: {accuracy*100:.2f}%")
```

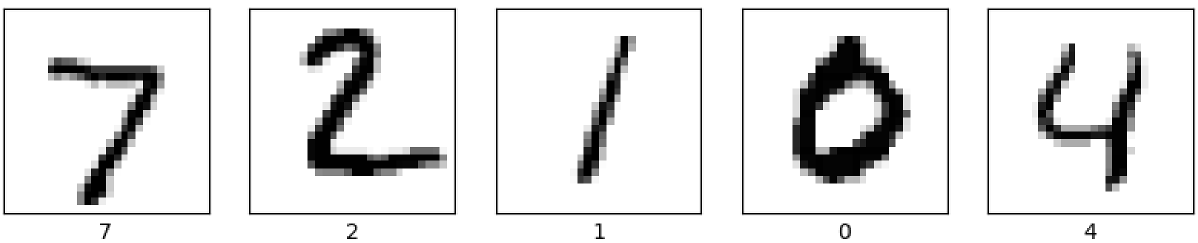
```
313/313 [==============================] - 0s 847us/step - loss: 0.1829 - ac
curacy: 0.9477
Test loss: 0.18
Test accuracy: 94.77%
```

That is very good! Let's take a look at some of the predictions the model made:

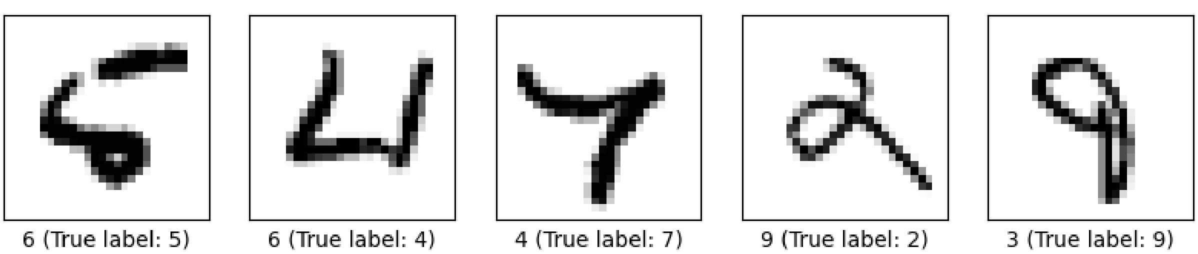
In [19]:
```python
# Visualize the first 5 correct predictions
import numpy as np

predictions = model.predict(x_test_flattened)
correct_indices = np.nonzero(
    np.argmax(predictions, axis=1) == np.argmax(y_test, axis=1)
)[0]
plt.figure(figsize=(10, 10))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test[correct_indices[i]].reshape(28, 28), cmap=plt.cm.binar
    plt.xlabel(np.argmax(predictions[correct_indices[i]]))
plt.show()
```

```
313/313 [==============================] - 0s 567us/step
```



In [20]:
```python
# Visualize the first 5 incorrect predictions
incorrect_indices = np.nonzero(
    np.argmax(predictions, axis=1) != np.argmax(y_test, axis=1)
)[0]
plt.figure(figsize=(10, 10))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test[incorrect_indices[i]].reshape(28, 28), cmap=plt.cm.bir
    plt.xlabel(f'{np.argmax(predictions[incorrect_indices[i]])} (True label:
plt.show()
```

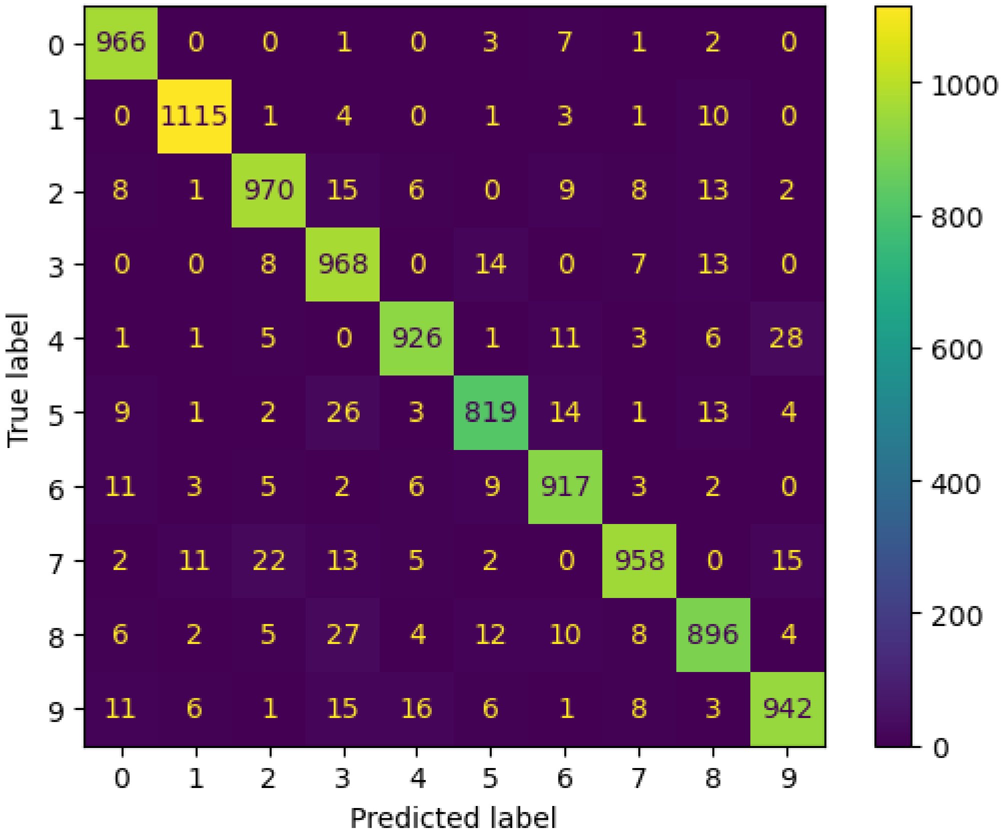


As above, it can be helpful to look at some of the incorrect predictions to get a sense of what the model is getting wrong. Sometimes, there might be something surprising that we can learn from, but here it seems like the model is just getting stuck on some of the more unusually written digits.

We can also use Sklearn's ConfusionMatrixDisplay to visualize the confusion matrix for our model:

In [21]:
```
# Plot the confusion matrix
ConfusionMatrixDisplay.from_predictions(np.argmax(y_test, axis=1), np.argmax
```

Out[21]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f16a03
fc150>
```



Convolutional Neural Networks

Now we will build a convolutional neural network to classify the same images. Convolutional neural networks are a special type of neural network that are designed to work with images. With convolutional layers, a "filter" is learned, which is a small matrix of weights. The filter is applied to each part of the image, and the output is a new image that is a "filtered" version of the original image. The filters are learned during training, and the goal is for the filters to learn to detect certain features of the image, such as edges or corners.

In this way, CNNs can accept 2D images as input, rather than a flattened vector. This allows us to preserve the spatial information of the image, which is important for image classification.

In [22]:
```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Build the model

input = keras.layers.Input(
    shape=(28, 28, 1)
)  # Instead of a flattened vector, we specify the shape of the input to be

conv1 = Conv2D(32, (3, 3), activation="relu")(
    input
)  # Our first convolutional layer - uses a 3x3 filter of weights to produce

conv2 = Conv2D(64, (3, 3), activation="relu")(
    conv1
)  # Our second convolutional layer - uses a 3x3 filter of weights to produc

flatten = Flatten()(
    conv2
)  # Flatten the output of the convolutional layers into a vector, so we can

hidden1 = Dense(128, activation="relu")(
    flatten
)  # Our first hidden layer - takes in the flattened output of the convoluti

output = Dense(10, activation="softmax")(
    hidden1
)  # Our output layer - takes in the output of the first hidden layer, and c
```

Like before, we just need to compile the model, specifying the optimizer, loss function, and metrics we want to use. We will use the Adam optimizer, the categorical cross-entropy loss function, and the accuracy metric.

In [23]:
```
model = keras.models.Model(inputs=input, outputs=output)
model.summary()

# Visualize the model
keras.utils.plot_model(model, show_shapes=True)
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 28, 28, 1)]       0         
                                                                 
 conv2d (Conv2D)             (None, 26, 26, 32)        320       
                                                                 
 conv2d_1 (Conv2D)           (None, 24, 24, 64)        18496     
                                                                 
 flatten (Flatten)           (None, 36864)             0         
                                                                 
 dense_3 (Dense)             (None, 128)               4718720   
                                                                 
 dense_4 (Dense)             (None, 10)                1290      
                                                                 
=================================================================
Total params: 4738826 (18.08 MB)
Trainable params: 4738826 (18.08 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
You must install pydot (`pip install pydot`) and install graphviz (see instr
uctions at https://graphviz.gitlab.io/download/) for plot_model to work.
```

In [24]:
```python
# Compile the model - this time we will use adam, a more advanced optimizer
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["a
```

In [25]:
```python
# Train the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.1)
```

```
Epoch 1/5
2023-11-07 17:13:06.765958: I tensorflow/tsl/platform/default/subprocess.cc:
304] Start cannot spawn child process: No such file or directory
2023-11-07 17:13:06.872708: I tensorflow/compiler/mlir/tensorflow/utils/dump
_mlir_util.cc:255] disabling MLIR crash reproducer, set env var `MLIR_CRASH_
REPRODUCER_DIRECTORY` to enable.
1688/1688 [==============================] - 6s 3ms/step - loss: 0.1141 - ac
curacy: 0.9651 - val_loss: 0.0580 - val_accuracy: 0.9850
Epoch 2/5
1688/1688 [==============================] - 4s 3ms/step - loss: 0.0347 - ac
curacy: 0.9889 - val_loss: 0.0459 - val_accuracy: 0.9882
Epoch 3/5
1688/1688 [==============================] - 4s 3ms/step - loss: 0.0206 - ac
curacy: 0.9933 - val_loss: 0.0604 - val_accuracy: 0.9848
Epoch 4/5
1688/1688 [==============================] - 4s 3ms/step - loss: 0.0123 - ac
curacy: 0.9960 - val_loss: 0.0469 - val_accuracy: 0.9900
Epoch 5/5
1688/1688 [==============================] - 4s 3ms/step - loss: 0.0118 - ac
curacy: 0.9958 - val_loss: 0.0498 - val_accuracy: 0.9878
```

Out[25]:
```
<keras.src.callbacks.History at 0x7f16a05dfb10>
```

In [26]:
```python
# Evaluate the model on the test set
loss, accuracy = model.evaluate(x_test, y_test)
```

```
print(f"Test loss: {loss:.2f}")
print(f"Test accuracy: {accuracy*100:.2f}%")
```

```
313/313 [==============================] - 0s 1ms/step - loss: 0.0500 - accu
racy: 0.9872
Test loss: 0.05
Test accuracy: 98.72%
```

Between swapping out stochastic gradient descent for Adam, and using convolutional layers instead of dense layers, we were able to improve our accuracy by 1.5%!

Next, we are going to introduce a couple of 'Callbacks'. These are functions that are called during training. We will use two callbacks: EarlyStopping and ModelCheckpoint.

EarlyStopping will stop the training process if the validation loss stops improving. This means that we don't have to specify the number of epochs ahead of time - the model will train until it stops improving, and then stop automatically.

ModelCheckpoint will save the best model (the one with the lowest validation loss) to a file. This is useful because we can then load the best model and evaluate it on the test set.

In [27]:
```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

early_stopping = EarlyStopping(
    monitor="val_loss", patience=5
)  # Stop training if the validation loss hasn't improved in 5 epochs

model_checkpoint = ModelCheckpoint(
    "best_model", monitor="val_loss", save_best_only=True
)  # Save the model with the lowest validation loss to a folder called 'best
```

This part is the same as before, but we want a new model so we'll run it again:

In [28]:
```
input = keras.layers.Input(shape=(28, 28, 1))
conv1 = Conv2D(32, (3, 3), activation="relu")(input)
pool1 = MaxPooling2D((2, 2))(
    conv1
)  # Add a pooling layer to reduce the size of the output of the convolution
conv2 = Conv2D(64, (3, 3), activation="relu")(pool1)
pool2 = MaxPooling2D((2, 2))(conv2)  # Add another pooling layer
flatten = Flatten()(pool2)
hidden1 = Dense(128, activation="relu")(flatten)
output = Dense(10, activation="softmax")(hidden1)
model = keras.models.Model(inputs=input, outputs=output)
```

And now we can train:

In [29]:
```
# Compile the model
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["a
```

```python
# Train the model with callbacks
history = model.fit(
    x_train,
    y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.1,
    callbacks=[early_stopping, model_checkpoint],
)
```

```
Epoch 1/50
1688/1688 [==============================] - ETA: 0s - loss: 0.1367 - accura
cy: 0.9577INFO:tensorflow:Assets written to: best_model/assets
INFO:tensorflow:Assets written to: best_model/assets
1688/1688 [==============================] - 4s 2ms/step - loss: 0.1367 - ac
curacy: 0.9577 - val_loss: 0.0458 - val_accuracy: 0.9867
Epoch 2/50
1688/1688 [==============================] - ETA: 0s - loss: 0.0431 - accura
cy: 0.9866INFO:tensorflow:Assets written to: best_model/assets
INFO:tensorflow:Assets written to: best_model/assets
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0431 - ac
curacy: 0.9866 - val_loss: 0.0379 - val_accuracy: 0.9883
Epoch 3/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0293 - ac
curacy: 0.9908 - val_loss: 0.0401 - val_accuracy: 0.9890
Epoch 4/50
1683/1688 [============================>.] - ETA: 0s - loss: 0.0203 - accura
cy: 0.9934INFO:tensorflow:Assets written to: best_model/assets
INFO:tensorflow:Assets written to: best_model/assets
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0204 - ac
curacy: 0.9934 - val_loss: 0.0349 - val_accuracy: 0.9898
Epoch 5/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0157 - ac
curacy: 0.9946 - val_loss: 0.0403 - val_accuracy: 0.9907
Epoch 6/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0123 - ac
curacy: 0.9959 - val_loss: 0.0457 - val_accuracy: 0.9897
Epoch 7/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0104 - ac
curacy: 0.9966 - val_loss: 0.0403 - val_accuracy: 0.9913
Epoch 8/50
1654/1688 [============================>.] - ETA: 0s - loss: 0.0087 - accura
cy: 0.9971INFO:tensorflow:Assets written to: best_model/assets
INFO:tensorflow:Assets written to: best_model/assets
```

```
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0087 - ac
curacy: 0.9970 - val_loss: 0.0337 - val_accuracy: 0.9930
Epoch 9/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0079 - ac
curacy: 0.9973 - val_loss: 0.0365 - val_accuracy: 0.9915
Epoch 10/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0054 - ac
curacy: 0.9984 - val_loss: 0.0541 - val_accuracy: 0.9902
Epoch 11/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0054 - ac
curacy: 0.9981 - val_loss: 0.0560 - val_accuracy: 0.9917
Epoch 12/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0059 - ac
curacy: 0.9980 - val_loss: 0.0398 - val_accuracy: 0.9912
Epoch 13/50
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0035 - ac
curacy: 0.9989 - val_loss: 0.0421 - val_accuracy: 0.9913
```

In [30]:

```
# Load the best model
model = keras.models.load_model("best_model")
```

In [31]:

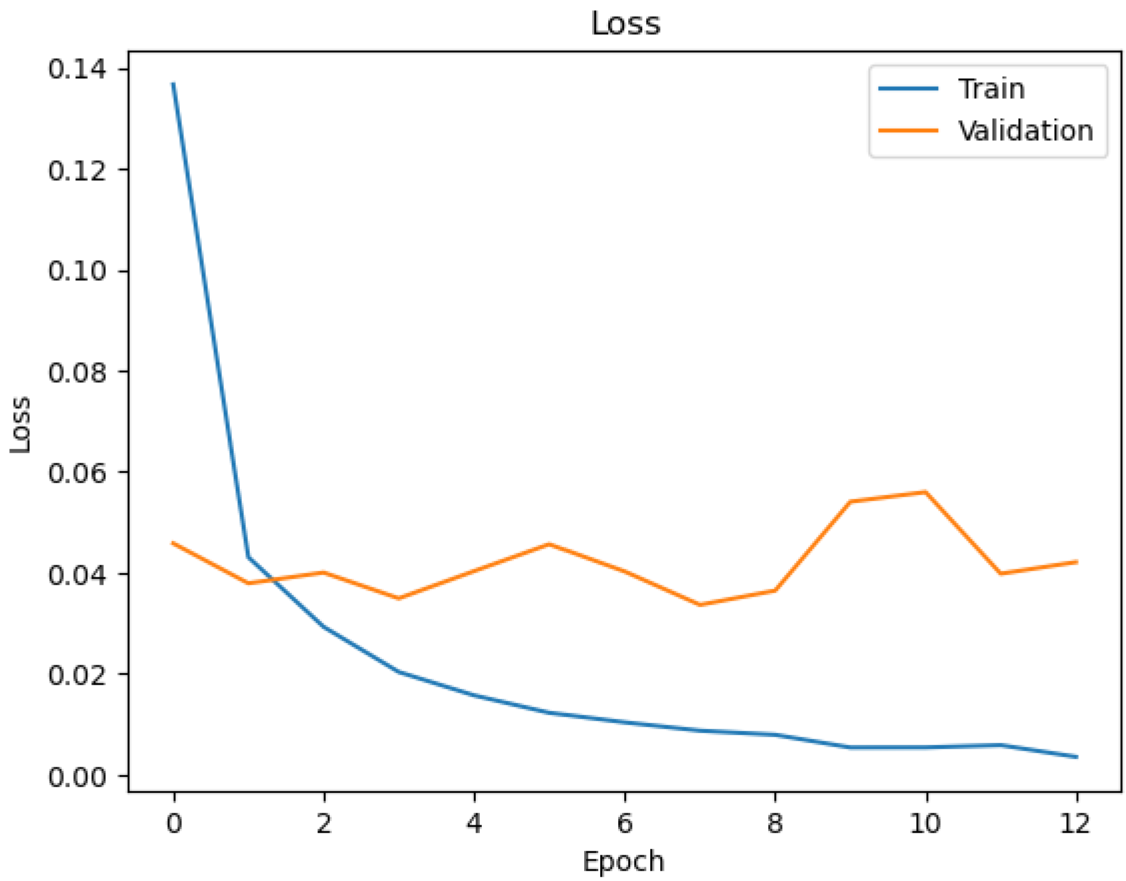
```
# Evaluate the model on the test set
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test loss: {loss:.2f}")
print(f"Test accuracy: {accuracy*100:.2f}%")
```

```
313/313 [==============================] - 0s 986us/step - loss: 0.0345 - ac
curacy: 0.9912
Test loss: 0.03
Test accuracy: 99.12%
```

When we train a model, we can also access the history of the training process. This includes the loss and accuracy on the training and validation sets at each epoch. We can use this to plot the training and validation loss over time, which can help us diagnose problems with our model.

In [32]:

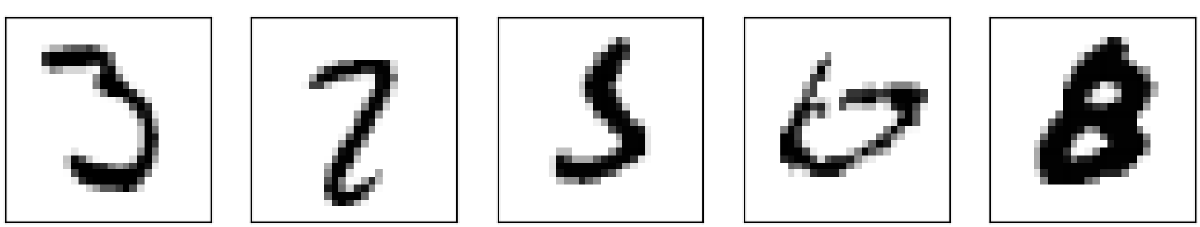
```
# Plot the training and validation loss
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title("Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(["Train", "Validation"])
plt.show()
```



Let's take a look at some of the incorrect predictions for this model:

In [33]:
```
# Visualize the first 5 incorrect predictions
predictions = model.predict(x_test)
incorrect_indices = np.nonzero(
    np.argmax(predictions, axis=1) != np.argmax(y_test, axis=1)
)[0]
plt.figure(figsize=(10, 10))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test[incorrect_indices[i]], cmap=plt.cm.binary)
    plt.xlabel(
        f"{np.argmax(predictions[incorrect_indices[i]])} (True label: {np.ar
    )
plt.show()
```

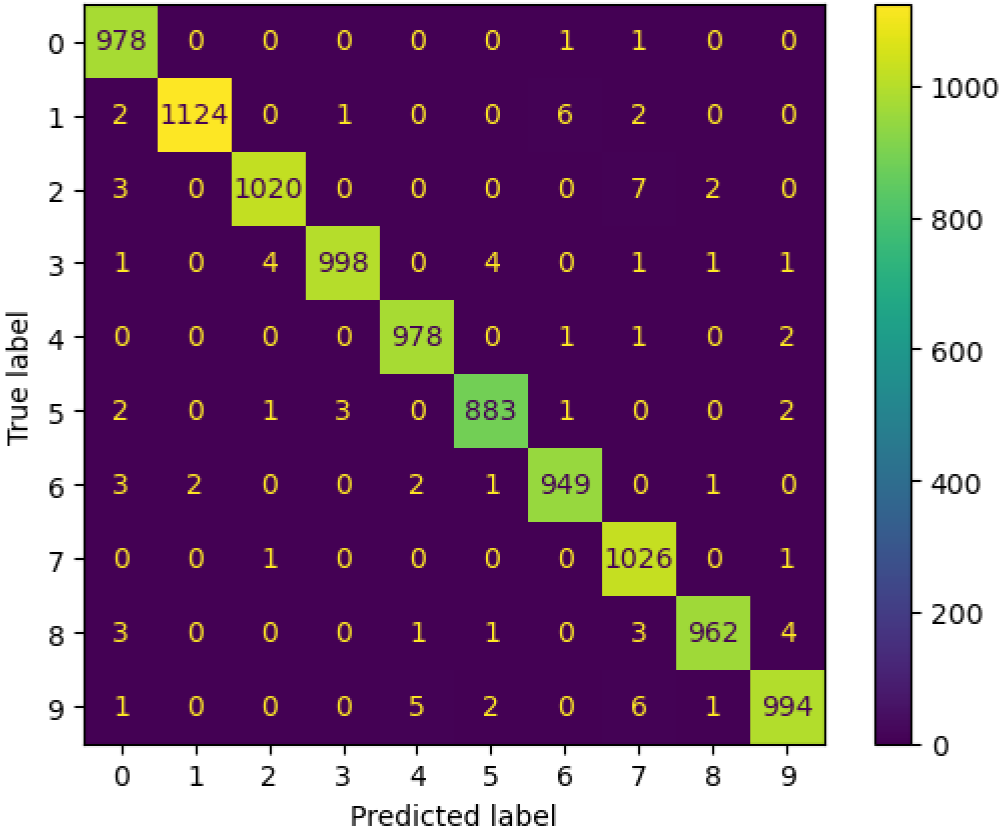
```
313/313 [==============================] - 0s 700us/step
```



2 (True label: 3) 7 (True label: 2) 3 (True label: 5) 0 (True label: 6) 0 (True label: 8)

In [34]:
```
# Plot the confusion matrix
ConfusionMatrixDisplay.from_predictions(np.argmax(y_test, axis=1), np.argmax
```

Out[34]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f16783
df790>
```



Your Turn

Now it's your turn to build a neural network! We will use the CIFAR-10 dataset, which consists of 60,000 32x32 color images in 10 classes. We will build a neural network to classify these images.

First, we will download the dataset:

In [35]:
```
from tensorflow.keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
class_names = [
    "airplane",
    "automobile",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
```

```
    "truck",
]

# Preview the first 5 images
plt.figure(figsize=(10, 10))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i])
    plt.xlabel(class_names[y_train[i][0]])
plt.show()
```

In [36]:
```
# Normalize the data
x_train = x_train / x_train.max()
x_test = x_test / x_test.max()
```

In [37]:
```
# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

Using the example code above, build a model that can classify these images. You can copy directly one of the model definitions from above, but if you feel like it, you can also extend the network further. You can get more information about available layers and other options in the Keras documentation.

Note that there are two key differences between our MNIST images of digits, and these CIFAR images:

- The CIFAR images are in color, so they have 3 channels instead of 1.
- The CIFAR images are 32x32 pixels, instead of 28x28.

This means that instead of each image having a shape of (28,28,1), indicating a 28x28 image with 1 channel, each image will have a shape of (32,32,3), indicating a 32x32 image with 3 channels. You will need to change the input shape of your model accordingly.

In [38]:
```
input = keras.layers.Input(shape=(32, 32, 3))
conv1 = Conv2D(32, (3, 3), activation="relu")(input)
pool1 = MaxPooling2D((2, 2))(conv1)
conv2 = Conv2D(64, (3, 3), activation="relu")(pool1)
pool2 = MaxPooling2D((2, 2))(conv2)
```

```python
flatten = Flatten()(pool2)
hidden1 = Dense(128, activation="relu")(flatten)
output = Dense(10, activation="softmax")(hidden1)
model = keras.models.Model(inputs=input, outputs=output)
model.summary()
```

```
Model: "model_3"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_4 (Conv2D)           (None, 30, 30, 32)        896

 max_pooling2d_2 (MaxPoolin  (None, 15, 15, 32)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_3 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 flatten_2 (Flatten)         (None, 2304)              0

 dense_7 (Dense)             (None, 128)               295040

 dense_8 (Dense)             (None, 10)                1290

=================================================================
Total params: 315722 (1.20 MB)
Trainable params: 315722 (1.20 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [39]:
```python
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["a
```

In [40]:
```python
# Train the model with callbacks
history = model.fit(
    x_train,
    y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.1,
    callbacks=[early_stopping, model_checkpoint],
)
```

```
Epoch 1/50
1407/1407 [==============================] - 4s 2ms/step - loss: 1.4366 - ac
curacy: 0.4874 - val_loss: 1.1729 - val_accuracy: 0.5838
Epoch 2/50
1407/1407 [==============================] - 2s 2ms/step - loss: 1.0835 - ac
curacy: 0.6198 - val_loss: 1.0134 - val_accuracy: 0.6482
Epoch 3/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.9492 - ac
curacy: 0.6698 - val_loss: 0.9511 - val_accuracy: 0.6766
Epoch 4/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.8484 - ac
curacy: 0.7036 - val_loss: 0.8780 - val_accuracy: 0.6988
Epoch 5/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.7721 - ac
curacy: 0.7293 - val_loss: 0.8735 - val_accuracy: 0.7066
Epoch 6/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.6997 - ac
curacy: 0.7546 - val_loss: 0.9389 - val_accuracy: 0.6930
Epoch 7/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.6348 - ac
curacy: 0.7753 - val_loss: 0.8996 - val_accuracy: 0.7052
Epoch 8/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.5726 - ac
curacy: 0.7984 - val_loss: 0.9125 - val_accuracy: 0.7116
Epoch 9/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.5151 - ac
curacy: 0.8180 - val_loss: 0.8976 - val_accuracy: 0.7152
Epoch 10/50
1407/1407 [==============================] - 2s 2ms/step - loss: 0.4639 - ac
curacy: 0.8361 - val_loss: 0.9810 - val_accuracy: 0.7056
```

In [41]:

```python
# Code to evaluate model performance
predictions = model.predict(x_test)
predictions = np.argmax(predictions, axis=1)
y_test_max = np.argmax(y_test, axis=1)

print(classification_report(y_test_max, predictions, target_names=class_name
```

```
313/313 [==============================] - 0s 873us/step
              precision    recall  f1-score   support

    airplane       0.66      0.79      0.72      1000
  automobile       0.82      0.81      0.82      1000
        bird       0.46      0.71      0.56      1000
         cat       0.54      0.48      0.51      1000
        deer       0.70      0.60      0.65      1000
         dog       0.62      0.54      0.58      1000
        frog       0.81      0.72      0.76      1000
       horse       0.79      0.72      0.75      1000
        ship       0.81      0.81      0.81      1000
       truck       0.86      0.70      0.77      1000

    accuracy                           0.69     10000
   macro avg       0.71      0.69      0.69     10000
weighted avg       0.71      0.69      0.69     10000
```

In [42]:
```
# Plot the confusion matrix
ConfusionMatrixDisplay.from_predictions(y_test_max, predictions, display_lab
```

Out[42]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f16785
bdd50>
```