I've been learning about Convolutional Neural Networks (CNNs) recently and wanted to give them a go on a challenge I have attempted with a Deep Neural Network in the past. I got 0.94285 accuracy with a Deep Neural Network so I'm keen to do better with a CNN.

Let's begin by loading the libraries we will be using.

In [17]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
```

Load and data analysis

First things first, load the training dataset. Although this is an image recognition task, the data comes in a csv format rather than a directory of PNGs/JPGs. The easiest way to load this data is to use pandas. We'll do this and have a look at the first row.

In [18]:
```
data = pd.read_csv('/home/student/Desktop/train.csv')
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42000 entries, 0 to 41999
Columns: 785 entries, label to pixel783
dtypes: int64(785)
memory usage: 251.5 MB
None
```

So we can see that each row of data represents an image. Each image is made up 784 pixels (which is a 28x28 image). Each pixel is represented in the columns with a prefix of "pixel". These columns contain a number between 0 and 255 where 0 represents a white pixel, 255 a black pixel and the other numbers represent shades of grey.

We also have a column called "label" that represents the number in the image (the images class). This number will be anything between 0 and 9.

To test the model as it trains I will split the dataset into a training and validation dataset. Then I'll remove the labels from the datasets and put them into their own dataframes. I want to make a lot of the images available for training so taking some influence from other notebooks in this competition I'm going to go for a 90%-10% split.

In [19]:
```
train_data = data.head(37800)
val_data = data.tail(4200)

train_labels = train_data.pop('label')
val_labels = val_data.pop('label')
```

I'll be using Tensorflow to train an image classifier model. Tensorflow has it's own api for creating a data pipeline which makes it easier to feed the data into a Tensorflow model. To use this api the data needs to be loaded into a Tensorflow data object.

In [20]:
```
tf_train_data = tf.data.Dataset.from_tensor_slices((train_data.valu
tf_val_data = tf.data.Dataset.from_tensor_slices((val_data.values,
```

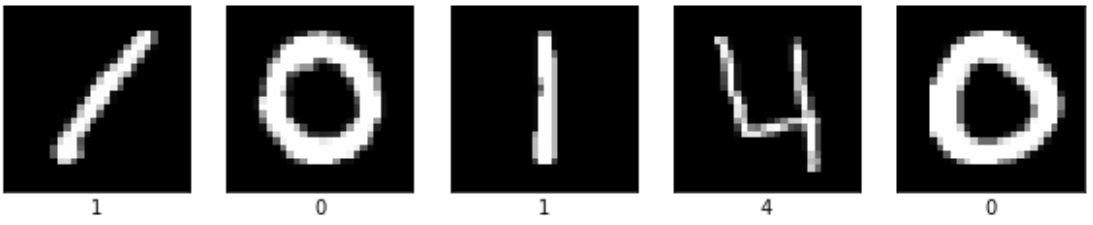
```
print(tf_train_data)
print(tf_val_data)
```

```
2022-09-12 10:44:02.750633: W tensorflow/stream_executor/platform
/default/dso_loader.cc:64] Could not load dynamic library 'libcud
a.so.1'; dlerror: libcuda.so.1: cannot open shared object file: N
o such file or directory
2022-09-12 10:44:02.750660: W tensorflow/stream_executor/cuda/cud
a_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-09-12 10:44:02.750679: I tensorflow/stream_executor/cuda/cud
a_diagnostics.cc:156] kernel driver does not appear to be running
on this host (student-OptiPlex-3050): /proc/driver/nvidia/version
does not exist
2022-09-12 10:44:02.793119: I tensorflow/core/platform/cpu_featur
e_guard.cc:193] This TensorFlow binary is optimized with oneAPI D
eep Neural Network Library (oneDNN) to use the following CPU inst
ructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the a
ppropriate compiler flags.

<TensorSliceDataset element_spec=(TensorSpec(shape=(784,), dtype=
tf.int64, name=None), TensorSpec(shape=(), dtype=tf.int64, name=N
one))>
<TensorSliceDataset element_spec=(TensorSpec(shape=(784,), dtype=
tf.int64, name=None), TensorSpec(shape=(), dtype=tf.int64, name=N
one))>

2022-09-12 10:44:02.920065: W tensorflow/core/framework/cpu_alloc
ator_impl.cc:82] Allocation of 237081600 exceeds 10% of free syst
em memory.
```

Before going on to form the data pipeline, I'll have a look at some of the images in the dataset and visualise them with their labels. As pixels for the images have been flattened into a 2D array, I have used a reshape function to put the images into a 28x28 shape that is more human friendly.

In [21]:
```
plt.figure(figsize=(10,10))
i = 0

for image, label in tf_train_data.take(5):
    plt.subplot(1,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    plt.imshow(image.numpy().reshape((28, 28)), cmap='gray')
    plt.xlabel(label.numpy())

    i = i + 1
```

```
2022-09-12 10:44:07.523909: W tensorflow/core/framework/cpu_alloc
ator_impl.cc:82] Allocation of 237081600 exceeds 10% of free syst
em memory.
```



Data pipeline

Now that the data is loaded and a quick piece of analysis has been done I am ready to form the pipeline that will feed the images into the model. The first step is to perform some small transformations to the images. I'll do the following to each image:

1. **Reshape:** The images are currently formatted as a flat list of pixels. CNNs need the data to be reshaped into a 3D matrix which in this case is a 28x28 matrix. I also need to specify the channels. As these images are greyscale I only need the one. The shape of the data then is (28, 28, 1).
2. **Scale:** Models seem to perform better when numeric features are put onto a 0-1 scale. As all the features are numeric and on a 0-255 scale it is easy to put the data onto this scale by simply dividing the features by 255.

In [23]:
```
def preprocess_image(image, label):
    image = tf.reshape(image, [28, 28, 1])
    image = tf.cast(image, tf.float32) / 255.

    return image, label

tf_train_data = tf_train_data.map(
    preprocess_image,
    num_parallel_calls=tf.data.experimental.AUTOTUNE
)

tf_val_data = tf_val_data.map(
    preprocess_image,
    num_parallel_calls=tf.data.experimental.AUTOTUNE
)

print(tf_train_data)
print(tf_val_data)
```

```
<ParallelMapDataset element_spec=(TensorSpec(shape=(28, 28, 1), d
type=tf.float32, name=None), TensorSpec(shape=(), dtype=tf.int64,
name=None))>
<ParallelMapDataset element_spec=(TensorSpec(shape=(28, 28, 1), d
type=tf.float32, name=None), TensorSpec(shape=(), dtype=tf.int64,
name=None))>
```

Now the data is preprocessed there are few configurations that can be made to the pipeline. Here's a list of those configurations:

1. **Shuffle:** To make sure the model doesn't pick up anything from the order of the rows in the dataset the top 100 rows of data will be shuffled per training step.
2. **Batch:** This is quite a large dataset in terms of both rows and columns. To ensure the processor doesn't get overloaded the data will be fed in 32 images at a time.
3. **Prefetch:** The speed up the training the pipeline can be set to fetch the next batch of data while the current batch of data is in training. When the current step ends there is no time wasted loading the next batch as it is ready to go.

In [24]:
```
def pipeline(tf_data):
    tf_data = tf_data.shuffle(100)
    tf_data = tf_data.batch(32)
```

```
    tf_data = tf_data.prefetch(tf.data.experimental.AUTOTUNE)

    return tf_data

tf_train_data = pipeline(tf_train_data)
tf_val_data = pipeline(tf_val_data)

print(tf_train_data)
print(tf_val_data)
```

```
<PrefetchDataset element_spec=(TensorSpec(shape=(None, 28, 28,
1), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype
=tf.int64, name=None))>
<PrefetchDataset element_spec=(TensorSpec(shape=(None, 28, 28,
1), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype
=tf.int64, name=None))>
```

Train model

The data is now ready to be fed into a model. I've chosen to use a CNN here and will be using an architecture based on the LeNet-5 architecture. I won't go into how CNNs work or why they are good for image classification tasks in this notebook. If you would like to learn more about them though I learnt about them on this course (https://www.coursera.org/learn/convolutional-neural-networks?specialization=deep-learning) by Andrew Ng and I can't recommend it highly enough.

Although I learnt about the LeNet-5 CNN architecture on the course and not by reading the paper it only seems right to quote the paper that LeNet-5 comes from: *"LeCun et al., 1998, Gradient-based learning applied to document recognition"*

So without further ado I'll use Tensorflow to define the model. As you will see the model is much the same as the LeNet-5 model except for a few changes:

1. All average pooling layers have been replaced with max pooling layers
2. The input shape is slightly smaller than the 32x32 shape in the paper. The first convolutional layer uses "same" padding to ensure the data is in the same shape as it is in the paper by the time it hits the first pooling layer.

In [25]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(6, (5, 5), activation='relu', padding='s
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(16, (5, 5), activation='relu', padding='
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(120, activation='relu'),
    tf.keras.layers.Dense(84, activation='relu'),

    tf.keras.layers.Dense(10, activation='softmax'),
])
```

Now I'll compile the model and get a print out describing it.

In [26]:
```
optimiser = tf.keras.optimizers.Adam(learning_rate=0.001)
```

```
model.compile(
    optimizer=optimiser,
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2D  (None, 14, 14, 6)        0
 )

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 16)         0
 2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

Tensorflow has a piece of functionality called callbacks that enable us to monitor the models metrcis during training and take some action as a result. Using this functionality I will implement the following:

1. **Learning Rate Decay:** If the models loss does not reduce for 2 consecutive epochs, reduce the learning rate. This should stop the model stepping over the optimum too many times.
2. **Early Stopping:** If the models loss does not reduce for 5 consecutive epochs then it's clear the model is not learning anything anymore. Save some computation and stop training.

Note: the verbose parameter simply states whether a message should be printed in the console (1) or not (0) when a callback is made.

In [27]:
```
callbacks = [
    tf.keras.callbacks.ReduceLROnPlateau(monitor='loss', patience=2
    tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5, ve
]
```

With the model compiled and the data sitting ready in the pipeline it is time to train the model. The validation dataset has been included here so that the model validates its accuracy after each step by making predictions against the validation dataset.

Note: Each epoch takes around 22 seconds to complete so this may take a few minutes

to run depending on how many epochs this is run for.

In [28]:
```
train_log = model.fit(
    tf_train_data,
    validation_data=tf_val_data,
    epochs=30,
    callbacks=callbacks
)
```

```
Epoch 1/30

2022-09-12 10:45:11.689213: W tensorflow/core/framework/cpu_alloc
ator_impl.cc:82] Allocation of 237081600 exceeds 10% of free syst
em memory.
```

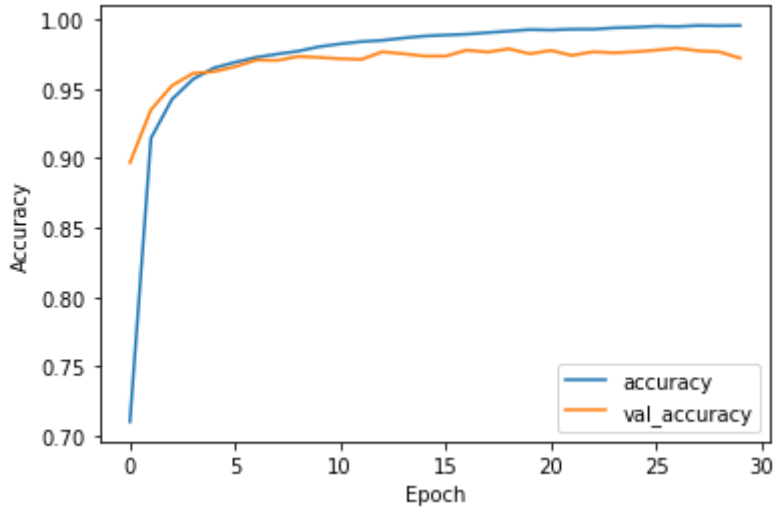
```
1182/1182 [==============================] - 10s 7ms/step - loss:
0.8473 - accuracy: 0.7104 - val_loss: 0.3455 - val_accuracy: 0.89
67 - lr: 0.0010
Epoch 2/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.2736 - accuracy: 0.9143 - val_loss: 0.2117 - val_accuracy: 0.93
48 - lr: 0.0010
Epoch 3/30
1182/1182 [==============================] - 8s 7ms/step - loss:
0.1790 - accuracy: 0.9422 - val_loss: 0.1554 - val_accuracy: 0.95
21 - lr: 0.0010
Epoch 4/30
1182/1182 [==============================] - 8s 7ms/step - loss:
0.1354 - accuracy: 0.9567 - val_loss: 0.1265 - val_accuracy: 0.96
10 - lr: 0.0010
Epoch 5/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.1109 - accuracy: 0.9649 - val_loss: 0.1185 - val_accuracy: 0.96
21 - lr: 0.0010
Epoch 6/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0980 - accuracy: 0.9689 - val_loss: 0.1060 - val_accuracy: 0.96
57 - lr: 0.0010
Epoch 7/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0865 - accuracy: 0.9724 - val_loss: 0.0936 - val_accuracy: 0.97
05 - lr: 0.0010
Epoch 8/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0785 - accuracy: 0.9748 - val_loss: 0.0942 - val_accuracy: 0.97
02 - lr: 0.0010
Epoch 9/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0701 - accuracy: 0.9769 - val_loss: 0.0846 - val_accuracy: 0.97
31 - lr: 0.0010
Epoch 10/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0627 - accuracy: 0.9801 - val_loss: 0.0800 - val_accuracy: 0.97
24 - lr: 0.0010
Epoch 11/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0559 - accuracy: 0.9821 - val_loss: 0.0873 - val_accuracy: 0.97
14 - lr: 0.0010
Epoch 12/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0516 - accuracy: 0.9837 - val_loss: 0.0849 - val_accuracy: 0.97
10 - lr: 0.0010
Epoch 13/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0467 - accuracy: 0.9847 - val_loss: 0.0683 - val_accuracy: 0.97
64 - lr: 0.0010
Epoch 14/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0422 - accuracy: 0.9863 - val_loss: 0.0726 - val_accuracy: 0.97
50 - lr: 0.0010
Epoch 15/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0388 - accuracy: 0.9878 - val_loss: 0.0798 - val_accuracy: 0.97
33 - lr: 0.0010
Epoch 16/30
```

```
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0361 - accuracy: 0.9884 - val_loss: 0.0800 - val_accuracy: 0.97
33 - lr: 0.0010
Epoch 17/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0337 - accuracy: 0.9891 - val_loss: 0.0704 - val_accuracy: 0.97
76 - lr: 0.0010
Epoch 18/30
1182/1182 [==============================] - 9s 7ms/step - loss:
0.0298 - accuracy: 0.9902 - val_loss: 0.0783 - val_accuracy: 0.97
62 - lr: 0.0010
Epoch 19/30
1182/1182 [==============================] - 9s 7ms/step - loss:
```

With the model trained I'll plot the accuracy achieved per epoch per dataset on a chart so that training progress is clearer.

In [29]:
```
plt.plot(train_log.history['accuracy'], label='accuracy')
plt.plot(train_log.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

print('Training accuracy: %f' % train_log.history['accuracy'][-1])
print('Validation accuracy: %f' % train_log.history['val_accuracy'][
```

```
Training accuracy: 0.995291
Validation accuracy: 0.971905
```



Inference

The final step is to use this trained model to classify the images in the test dataset. The first step is to load the training dataset and put it into the Tensorflow data format.

In [31]:
```
test_data = pd.read_csv('/home/student/Desktop/test.csv')
tf_test_data = tf.data.Dataset.from_tensor_slices(([test_data.to_nu
```

```
2022-09-12 10:50:54.781668: W tensorflow/core/framework/cpu_alloc
ator_impl.cc:82] Allocation of 175616000 exceeds 10% of free syst
em memory.
```

Then use the model to classify each image. The model will not just output one class but

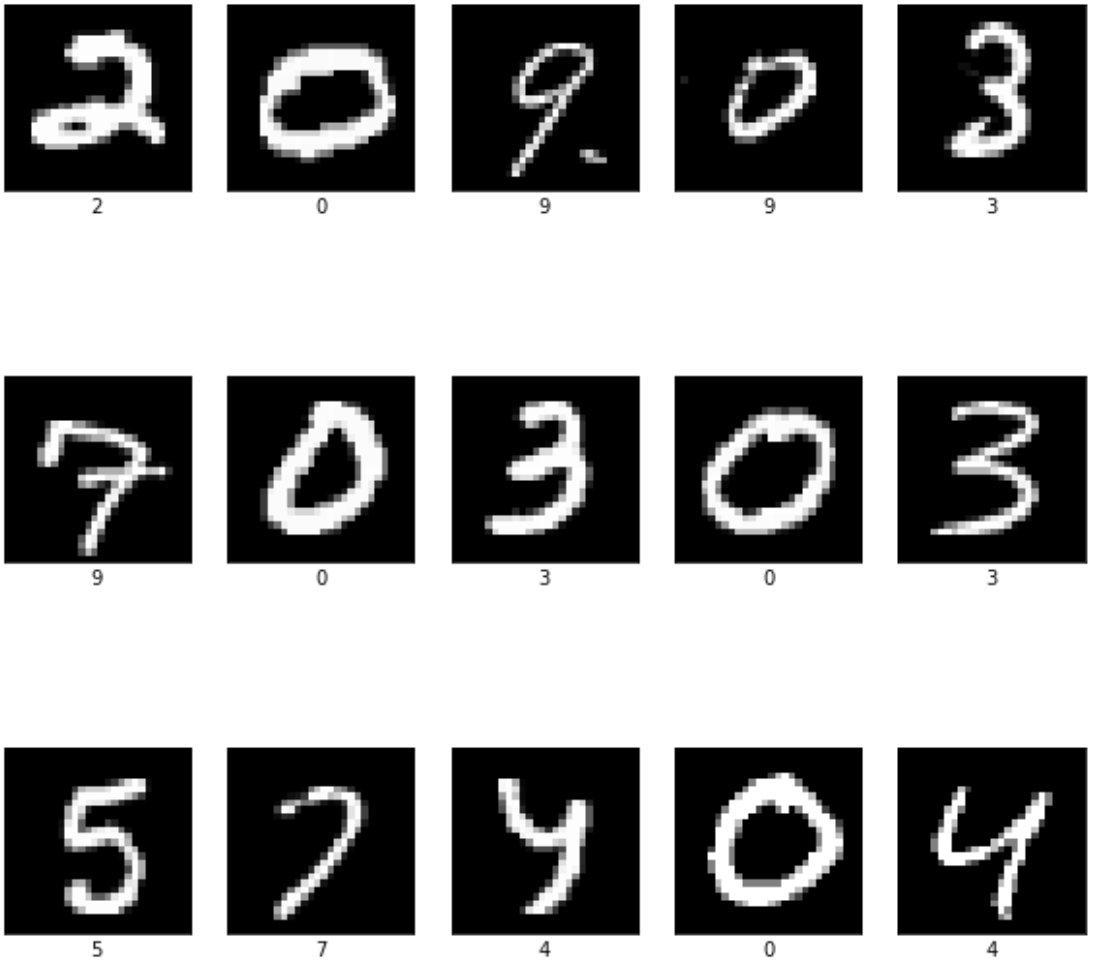
will instead output a probability for each possible class. Numpys argmax is thus used to find the class with the highest probability and this class is used for the image.

In [32]:
```
predictions = model.predict(tf_test_data)
predictions = np.argmax(predictions, axis=1)
```

```
2022-09-12 10:51:12.978704: W tensorflow/core/framework/cpu_alloc
ator_impl.cc:82] Allocation of 175616000 exceeds 10% of free syst
em memory.

1/1 [==============================] - 1s 1s/step
```

To see if the model is performing as expected I will check visualisations of the first fifteen images against the classifications produced by the model.

In [33]:
```
plt.figure(figsize=(10,10))

for i, row in test_data.head(15).iterrows():
    plt.subplot(3,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    plt.imshow(row.values.reshape((28, 28)), cmap='gray')
    plt.xlabel(predictions[i])
```



While it is unfeasible to eyeball all 28000 test images and predicted classes, checking the first fifteen and seeing 98% accuracy on the validation dataset gives me some confidence that the model is performing well. The last thing to do then is to put the classifications into

a submission dataframe and write it to a csv file

In [34]:
```
predictions_df = pd.DataFrame(data={'Label': predictions}, index=pd
predictions_df.index = predictions_df.index.rename('ImageId')

predictions_df.to_csv('submission_file.csv')
```

Final note

I'm still relatively new to data science and I'm keen to learn. Any comments, ideas and points to improve on would be much appreciated.

In [ ]:

In [ ]: