B. Testing Notebook

After training the model at train.ipynb we need to test its accuracy

1) Imports: We begin by importing our dependencies, for testing, we need tensorflow and tensorflow_datasets much like we did with training, but we also need numpy and seaborn

[1]:
```
import numpy as np
import seaborn as sns
from pprint import pprint
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_datasets as tfds
```

2) The dataset: We use the load method to load the mnist dataset, but this time, we only load the test split which contains 10000 images

[2]:
```
# load the mnist dataset
dstest, dsinfo = tfds.load(
    'mnist',
    split=['test'],  # only need the test set
    data_dir='../dataset/',
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

dstest = dstest[0] # Because tfds.load returns a list
```

```
2022-05-01 00:37:17.044709: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.072924: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.073686: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.074259: I tensorflow/core/platform/cpu_feature_guard.cc:
↪151]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2022-05-01 00:37:17.074565: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.074690: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.074789: I
```

```
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.499334: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.499480: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.499592: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:936] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-05-01 00:37:17.499686: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 4784 MB memory:  -> device:␣
↪0,
name: NVIDIA GeForce GTX 1060 6GB, pci bus id: 0000:01:00.0, compute␣
↪capability:
6.1
```

3) Preprocessing: We follow the same preprocessing we did while training the model, which is practically nothing that is necessary. However for performance reasons, we autotune and batch the test set

[3]:
```
batch_size = 128

# Evaluation pipleine
dstest = dstest.batch(batch_size)
dstest = dstest.cache()
dstest = dstest.prefetch(tf.data.AUTOTUNE)
```

4) The model: We load the model created in the train script

[4]:
```
model = tf.keras.models.load_model('./model.h5')
```

Printing the class names

[5]:
```
class_names = dsinfo.features['label'].names
print(class_names)
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Generating the model predictions

[6]:
```
model_probabilities = model.predict(dstest)
pprint(model_probabilities)
```

```
2022-05-01 00:37:30.189122: I tensorflow/stream_executor/cuda/cuda_dnn.cc:368]
Loaded cuDNN version 8303
```

```
array([[3.76803786e-13, 1.90661983e-12, 1.00000000e+00, ...,
        1.55628704e-11, 1.15425836e-10, 5.78510954e-11],
       [1.00000000e+00, 9.81507098e-10, 1.52358917e-10, ...,
        3.57904852e-11, 1.05996236e-11, 2.51395821e-10],
       [2.32067924e-11, 3.89482363e-10, 6.94404201e-10, ...,
        1.21059474e-09, 3.61068175e-09, 3.69479641e-10],
```

```
...,
[1.24935717e-12, 1.08420434e-12, 1.08029019e-09, ...,
 2.58623487e-11, 1.00000000e+00, 3.43887765e-12],
[1.00000000e+00, 2.40774800e-09, 1.48503154e-10, ...,
 3.69741703e-11, 6.14683976e-13, 1.21822230e-09],
[1.94027439e-09, 5.62938654e-11, 4.48329351e-12, ...,
 6.79747980e-10, 4.47745938e-11, 8.29751734e-09]], dtype=float32)
```

`model.predict` returns a list of lists, where each inner list contains the probabilities that image belongs to each class, to get the predicted labels we choose the class with the highest probability

[7]:
```
predictions = [np.argmax(x) for x in model_probabilities]
pprint(predictions[:10]) # printing the last 10 results
```

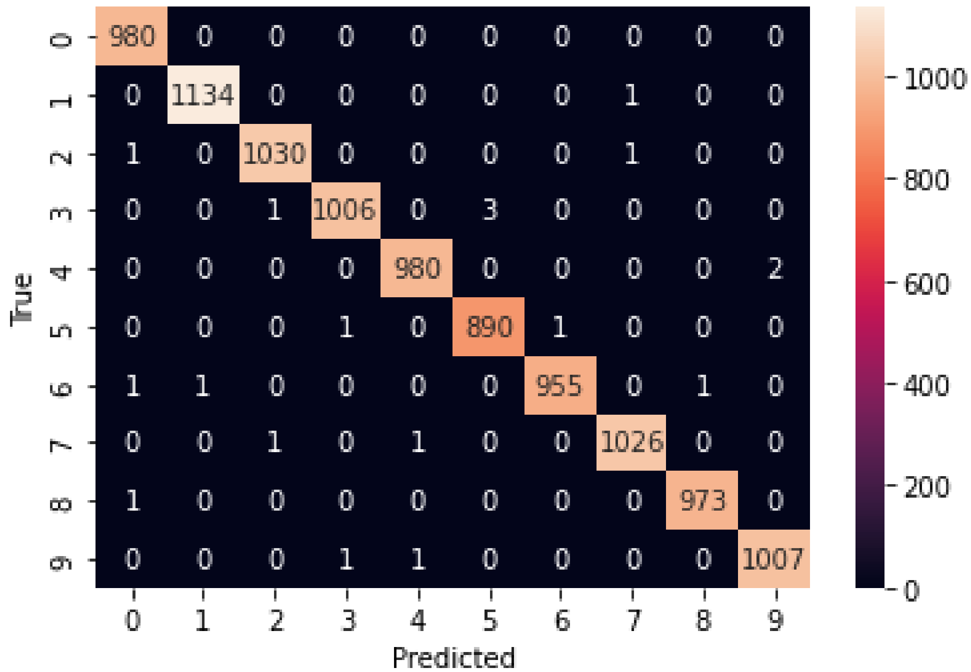
```
[2, 0, 4, 8, 7, 6, 0, 6, 3, 1]
```

5) Evaluation of the model: We get the actual true labels using this one line

[8]:
```
labels = np.concatenate([y for x, y in dstest], axis=0)
```

We then generate a confusion matrix and plot it

[9]:
```
confusion_matrix = tf.math.confusion_matrix(
    labels=labels,
    predictions=predictions,
)
sns.heatmap(confusion_matrix,
            annot=True,
            xticklabels=class_names,
            yticklabels=class_names,
            fmt='g')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('../paper/figs/confusion_matrix.svg', format='svg')
plt.show()
```



With the confusion matrix we can calculate the accurace as

$$\frac{\sum_{i=0}^{9} k_{ii}}{\sum_{i=0}^{9}\sum_{j=0}^{9} k_{ij}} \times 100$$

Where k_{xy} represent an element in the x^{th} row and the y^{th} column in the confusion matrix, in other words it is the sum of the elements in the diagonal of the confusion matrix, dividied by the total sum.

[10]:
```
diagonal_sum = 0
total_sum = 0

for i in range(len(class_names)):
    for j in range(len(class_names)):
        if(i == j):
            diagonal_sum += confusion_matrix[i][j]
        total_sum += confusion_matrix[i][j]

print("Diagonal sum: {}, Total sum: {}".format(diagonal_sum, total_sum)) #␣
↪total sum should be 10000 as the test split is 10000 images
accuracy = 100 * diagonal_sum/total_sum
print("Accuracy: {}".format(accuracy))
```

```
Diagonal sum: 9981, Total sum: 10000
Accuracy: 99.81
```

We can also calculate the per-digit accuracy for a digit i as

$$\frac{k_{ii}}{\sum_{j=0}^{9} k_{ij}} \times 100$$

(For example, for the digit 0, it was predicted correctly 980 times out of $970+0+0+0+0+0+0+0+0+0$) With that information, we can plot a bar chart to see how accurately our model predicts each digit with the following code

[12]:
```
fig = plt.figure(figsize=(15, 5))
ax = fig.add_axes([0, 0, 0.7, 1])
digits = list(range(0, 10))
digit_acc = []
for i in digits:
    row_sum = sum([confusion_matrix[i][j] for j in digits])
    digit_acc.append(100 * confusion_matrix[i][i]/row_sum)
plt.yticks(digits)
bars = ax.barh(digits, digit_acc)
ax.bar_label(bars)
ax.spines["top"].set_bounds(0,113)
ax.spines["bottom"].set_bounds(0,113)
ax.spines["right"].set_position(("outward", 60))
plt.rcParams.update({'font.size': 22}) # Increasing the text size
fig.tight_layout()
plt.savefig('../paper/figs/bar_plot.svg', format='svg', bbox_inches='tight')
plt.show()
```

```
/tmp/ipykernel_62979/3498705007.py:15: UserWarning: This figure includes Axes
that are not compatible with tight_layout, so results might be incorrect.
  fig.tight_layout()
```