ADVANCED MACHINE LEARNING
FINAL PROJECT

TOPIC – TensorFlow Digit classification

Prepared by

Manaswini Purumandla

Email – mpuruman@kent.edu

Submitted to

Murali Shanker

Email: mshanker@kent.edu

SUMMARY

In this project, I would like to elaborate on TensorFlow digit classification. The purpose of this project is to identify and classify handwritten digits. MNIST ("Modified National Institute of Standards and Technology") is the de facto "hello world" dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine-learning techniques emerge, MNIST remains a reliable resource for researchers and learners.
The goal is to correctly identify digits from a dataset of tens of thousands of handwritten images by using different techniques like the Sequential model using TensorFlow Keras. Image augmentation, Conv2D, Maxpool2D, Activation Function, Dropouts, Optimizer and Flatten etc. The measuring metric is accuracy.

PROBLEM

Everyone has a unique way of writing skills and how does a machine classify what exactly it is? The machine is trained to read the digits and classify the digits into labels. The goal is to correctly identify digits from a dataset of thousands of handwritten images.

DATA COLLECTION

The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel values of the associated image.

Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between

0 and 27, inclusive. Then pixelx is located on row i and column j of a 28 x 28 matrix, (indexing by zero).

Libraries

Here I have mentioned different libraries used.

NumPy - NumPy is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions. It is very useful for fundamental scientific computations in Machine Learning

Pandas - Pandas are one of the tools in Machine Learning which is used for data cleaning and analysis

Matplotlib - Matplotlib is one of the plotting libraries in python which is however widely in use for machine learning applications with its numerical mathematics extension- NumPy to create static, animated, and interactive visualizations.

TensorFlow - TensorFlow is an open-source library developed by Google primarily for deep learning applications

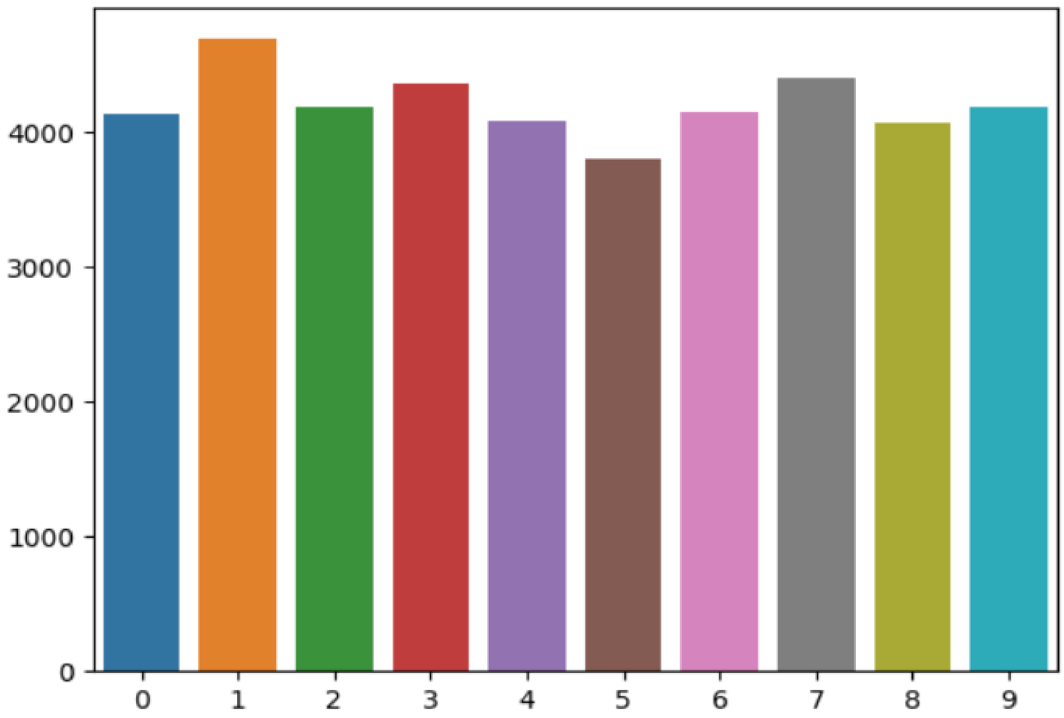
TensorFlow Keras - TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while Keras is a high-level neural network library that runs on top of TensorFlow

Seaborn - Seaborn is a library that uses Matplotlib underneath plot graphs. It will be used to visualize random distributions.

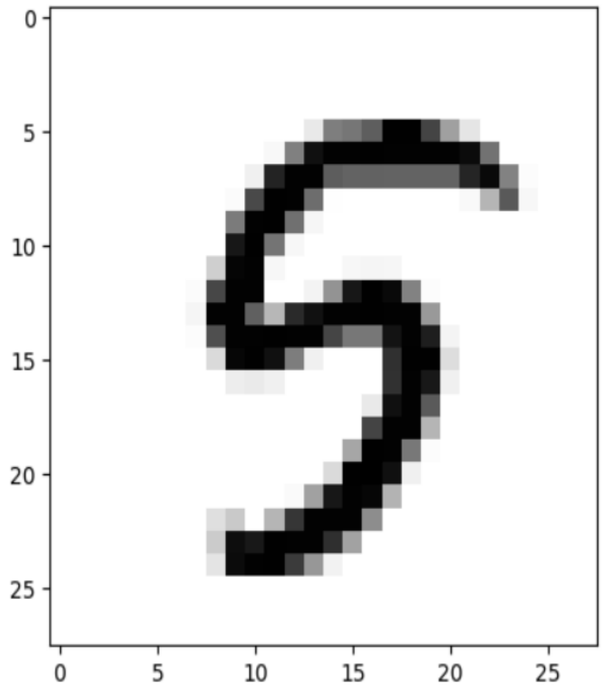
Data preparation and pre-processing

After installing the libraries, we load the data and divide the data into test and train files. Then we check for missing data. We proceed future if there is no missing data.

Data and target class visualizations



As you can see, there is a fairly even class distribution.



Here is an example of one of the digits. It is a 28 x 28 black and white image.

Creating training and validation sets

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.1, random_state=15)
print(X_train.shape, X_val.shape, y_train.shape, y_val.shape)
```

```
(37800, 784) (4200, 784) (37800,) (4200,)
```

Pre-processing pipelines

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

```
from sklearn.base import BaseEstimator, TransformerMixin
```

```
class ReshapeFunc(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass
    def fit(self, X, y=None):
        return self
    def transform(self, X, y=None):
        X = X.reshape((-1,28,28,1))
        return X
```

```
features_pipeline = Pipeline(steps=[
    ('Normalize', MinMaxScaler()),
    ('Reshape', ReshapeFunc())
])
```

Feature pipeline. Data is scaled between 0 and 1 and then reshaped into input format.

```
X_train = features_pipeline.fit_transform(X_train)
```

```
from sklearn.preprocessing import OneHotEncoder
target_pipeline = Pipeline(steps=[
    ('OneHot', OneHotEncoder())
])
```

####. One hot encoding is used, as we will be using a softmax activation function in the output node.*

```
y_train = target_pipeline.fit_transform(y_train.values.reshape(-1,1))
```

```
y_train = y_train.toarray()
```

```
print(X_train.shape, y_train.shape)
```

```
(37800, 28, 28, 1) (37800, 10)
```

```
X_val = features_pipeline.fit_transform(X_val)
```

```
y_val = target_pipeline.fit_transform(y_val.values.reshape(-1, 1))
```

```
y_val = y_val.toarray()
```

```
print(X_val.shape, y_val.shape)
```

```
(4200, 28, 28, 1) (4200, 10)
```

```
X_test = features_pipeline.fit_transform(df_test)
```

```python
# Precision (using keras backend)
def precision_metric(y_true, y_pred):
    threshold = 0.5  # Training threshold 0.5
    y_pred_y = K.cast(K.greater(K.clip(y_pred, 0, 1), threshold), K.floatx())

    true_positives = K.sum(K.clip(y_true * y_pred, 0, 1))
    false_negatives = K.sum(K.clip(y_true * (1-y_pred), 0, 1))
    false_positives = K.sum(K.clip((1-y_true) * y_pred, 0, 1))
    true_negatives = K.sum(K.clip((1 - y_true) * (1-y_pred), 0, 1))

    precision = true_positives / (true_positives + false_positives + K.epsilon())
    return precision

# Recall (using keras backend)
def recall_metric(y_true, y_pred):
    threshold = 0.5 #Training threshold 0.5
    y_pred = K.cast(K.greater(K.clip(y_pred, 0, 1), threshold), K.floatx())

    true_positives = K.sum(K.clip(y_true * y_pred, 0, 1))
    false_negatives = K.sum(K.clip(y_true * (1-y_pred), 0, 1))
    false_positives = K.sum(K.clip((1-y_true) * y_pred, 0, 1))
    true_negatives = K.sum(K.clip((1 - y_true) * (1-y_pred), 0, 1))

    recall = true_positives / (true_positives + false_negatives + K.epsilon())
    return recall

# F1-score (using keras backend)
def f1_metric(y_true, y_pred):
    precision = precision_metric(y_true, y_pred)
    recall = recall_metric(y_true, y_pred)
    f1 = 2 * ((precision * recall) / (recall+precision+K.epsilon()))
    return f1
```

```python
def build_model():
    inp = keras.Input(shape=(28,28,1))
    x = keras.layers.Conv2D(filters=32, kernel_size=(5,5), strides=(1,1),padding='SAME',
                            activation='relu')(inp)
    x = keras.layers.MaxPool2D(pool_size=(2,2))(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Conv2D(filters=64, kernel_size=(5,5), padding='SAME', activation='relu')(x)
    x = keras.layers.MaxPool2D(pool_size=(2,2))(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Flatten()(x)
    x = keras.layers.Dense(256, activation='relu')(x)
    x = keras.layers.Dropout(0.5)(x)
    output = keras.layers.Dense(10, activation='softmax')(x)

    model = keras.Model(inputs=inp, outputs=output)

    model.compile(loss=keras.losses.CategoricalCrossentropy(), optimizer=keras.optimizers.Adam(learning_rate=0.0001), r
    return model, inp, output
```

The model is built and compiled using categorical crossentropy and adam optimizer.

TECHNIQUES

Sequential model using TensorFlow keras - A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. Schematically, the following Sequential model: Define the Sequential model with 3 layers. model = keras.

Image Data Generator for image augmentation, which is a technique of applying different transformations to original images results in multiple transformed copies of the same image.

- Rescale: rescale the image
- Rotation range: value in degrees (0–180) within which to randomly rotate pictures
- Width shift, height shift: ranges within which to randomly translate pictures vertically or horizontally
- Shear range: randomly applying shearing transformations
- Zoom range: randomly zooming inside pictures
- Horizontal flip: randomly flipping half of the images horizontally
- Fill mode: used for filling in newly created pixels after rotation, width, and height shift.

CONV 2D: Applies a 2D convolution over an input signal composed of several input planes. where ⋆ is the valid 2D cross-correlation operator, N is the batch size, C denotes the number of channels, H is the height of input planes in pixels, and W is the s width in pixels.

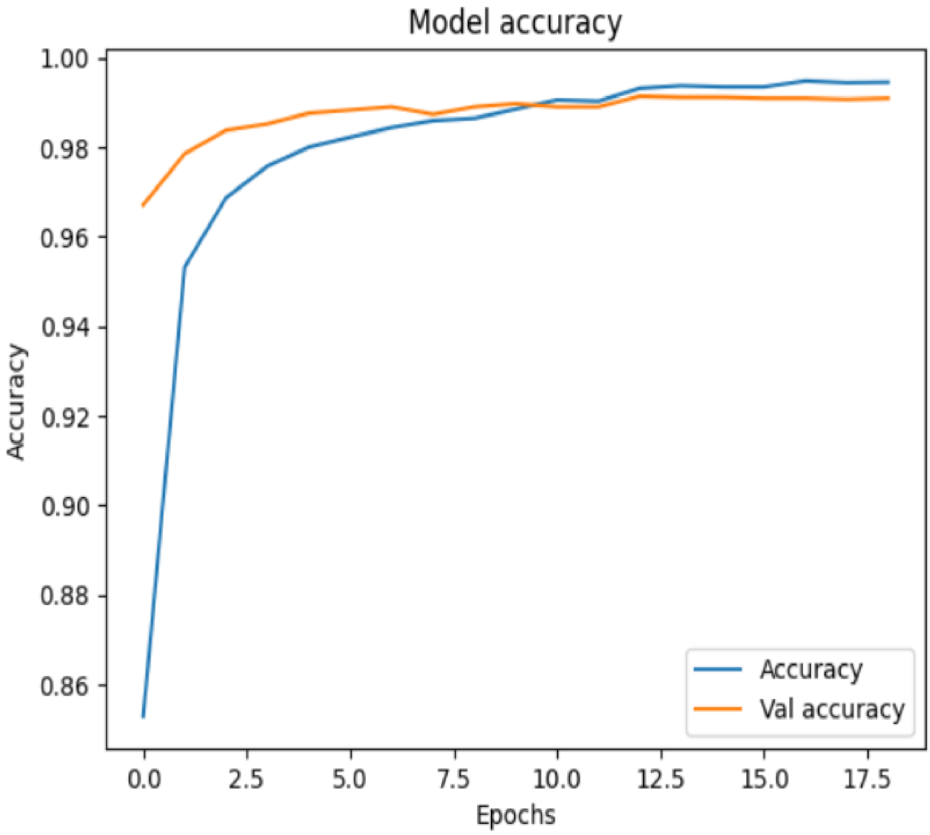
MaxPooling2D - Downsample the input along its spatial dimensions (height and width) by taking the maximum value over an input window (of size defined by pool size) for each channel of the input.

Dropout - The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network. All the forward and backward connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network.

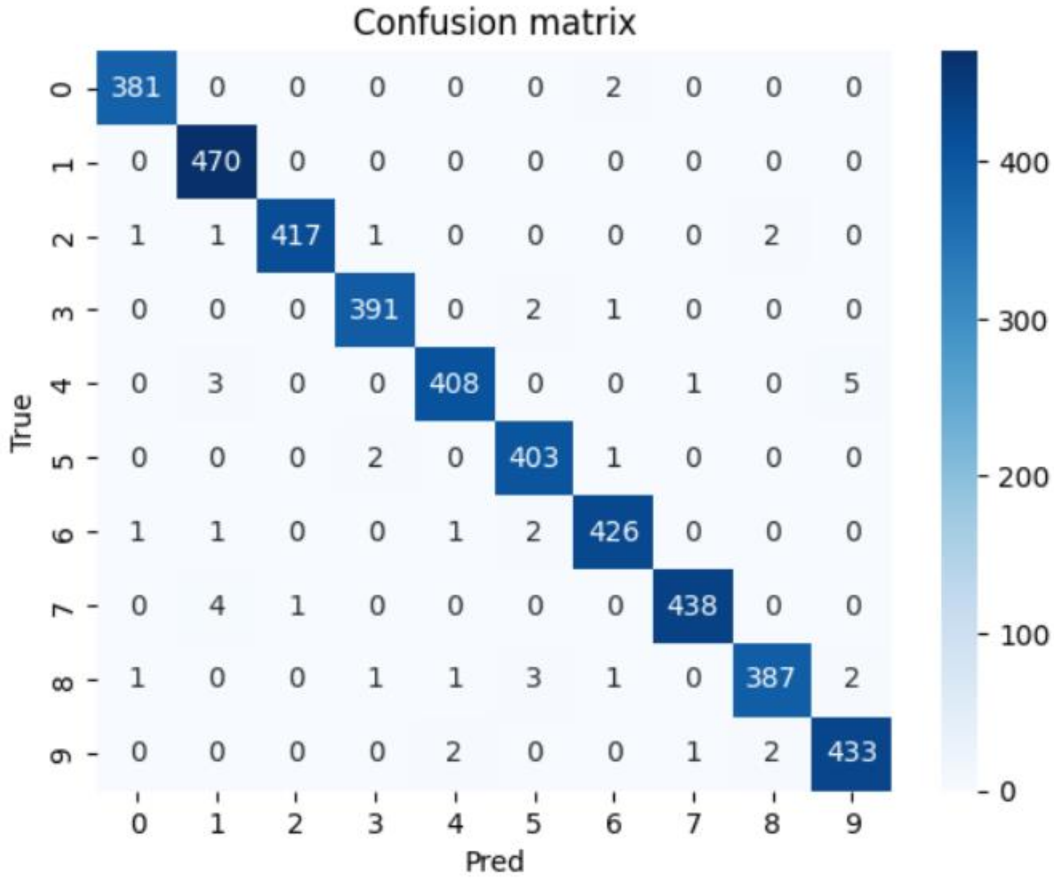
Flatten: Flattens the input. Does not affect the batch size.

Conclusions

The model is built and compiled using categorical cross entropy and adam optimizer.



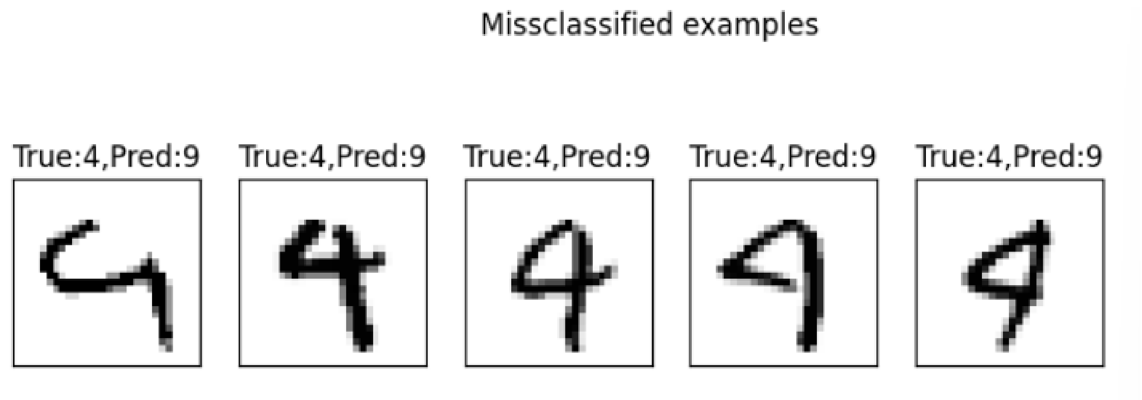
CONFUSION MATRIX




```
from sklearn.metrics import accuracy_score
accuracy_score(y_val_true, y_val_pred)
```

```
0.9890476190476191
```

The confusion matrix looks really good as well, with lots of predictions along the diagonal which is what we want to see. We got an accuracy of 0.9890476191. This is not bad for our first model! The model has done a pretty good job at classifying each class and is obtaining a high accuracy score. Now we should look at the examples the model is misclassifying. The class with the most confusion is between classes 5 and 6, let's look at the incorrectly predicted examples.



Looking at the misclassifications, it’s understandable why the model was unable to classify these examples correctly. Some of the examples are quite ambiguous, even a human labeler would probably be unable to clearly label them with good confidence.

However, it does seem that there is room for improvement in some of the examples. With the use of data augmentation and hyper-parameter tuning, we should be able to further improve performance.

Before we try these additional techniques, let's take a quick look at the learned convolutional filters and feature maps, which should give us some insight into how the network is learning.¶

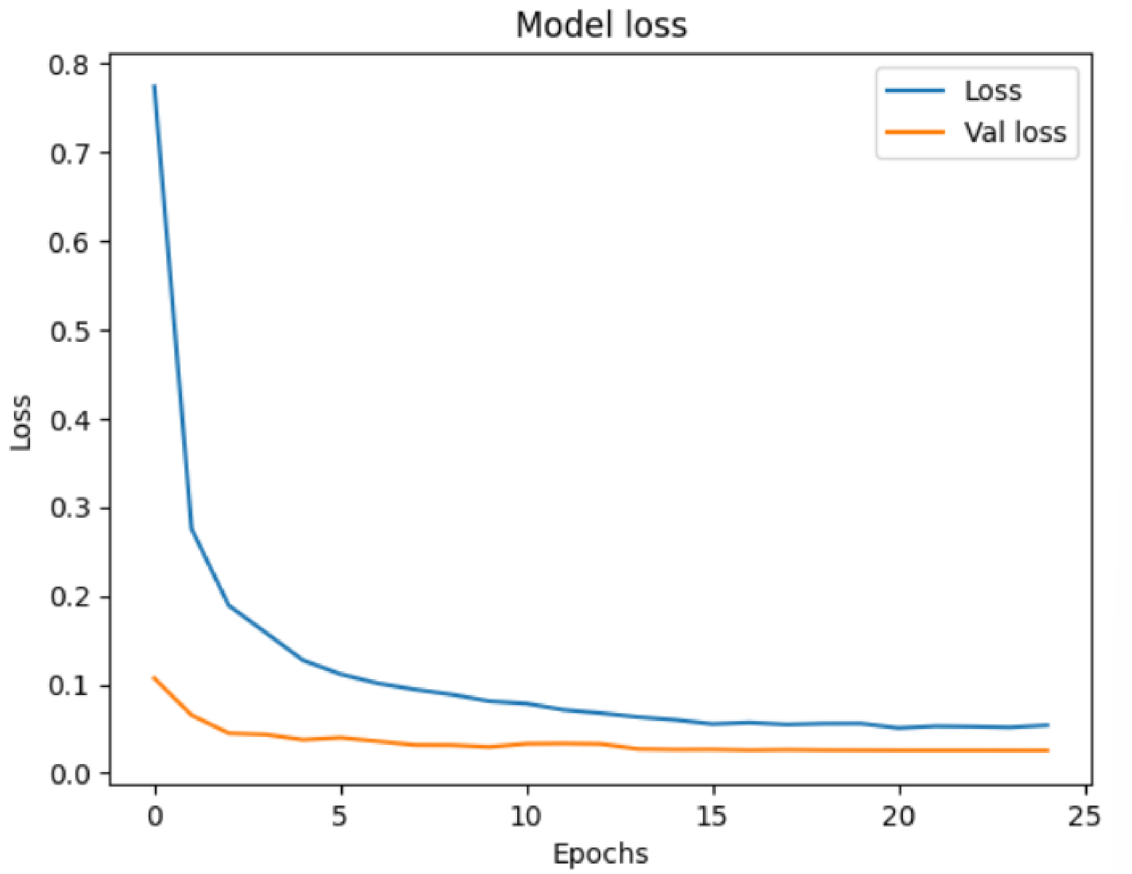
Then we took a test example. I ran it through the network to obtain and visualize the feature maps.

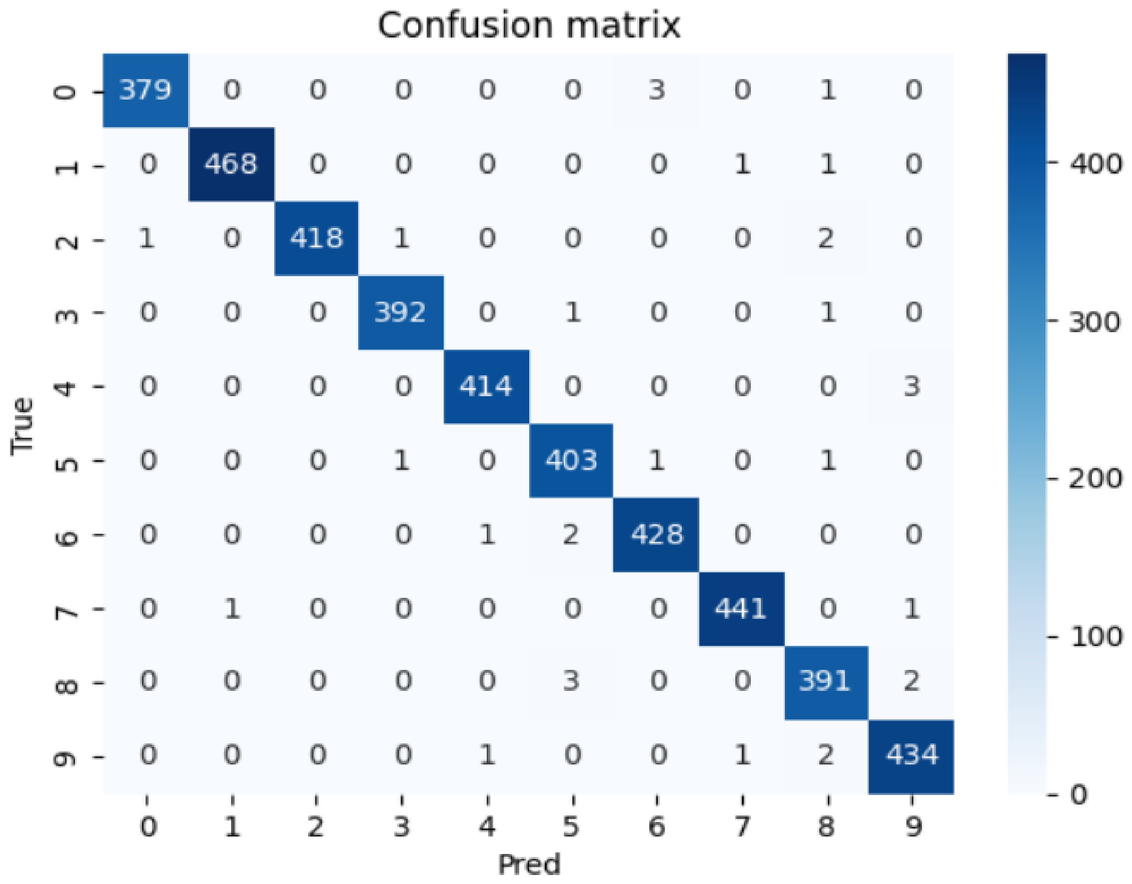
Contributions

To improve the accuracy more than 0.989% I used below technique.

.Data augmentation allows us to generate new images (artificial data) by slightly modifying the images in the training set by applying different transformations. In this notebook, we will use shifting, rotating, and zooming transformations to modify the data and generate new examples.

One of the benefits of data augmentation is it acts as a regularizer and helps to reduce overfitting when training a model. This is because, with more artificially generated images, the model is unable to overfit specific examples and is forced to generalize, thus the model becomes more robust. This generally leads to better overall performance.






```
accuracy_score(y_val_true, y_val_pred)
```

```
0.9923809523809524
```

We can see using data augmentation has improved our accuracy score by over 0.2%, which is quite a lot considering we are in the top 1% of accuracy score.

Hyper-parameter tuning

Now that we've implemented a data augmentation technique, we need to find the optimal hyper-parameter settings to maximize model performance. We will use the keras-tuner library, which is a hyperparameter optimization framework containing multiple tuning algorithms including Random Search, Hyper Band and Bayesian Optimization.

I tested both Random Search and Hyper Band and found Hyper Band to be much more successful so we will use that.

The accuracy score of our top model is significantly better than our previous model. This shows the importance of hyper-parameter tuning and was worth the time spent tuning.

APPENDIX

DIGIT RECOGNITION

In [1]:
```
#Libraries

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import tensorflow as tf
from tensorflow import keras
import seaborn as sns
```

Checking GPU is configured correctly for tensorflow.

In [107]:
```
tf.config.list_physical_devices()
```

Out[107]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
 PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

In [108]:
```
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

GPU is successfully configured. I will be using an Nvidia RTX 3080, which significantly increased training speeds by around 8x comparred to CPU.

In [451]:
```
file_dir = r'C:\Users\manaswini\Desktop\mnist'
```

Data preparation and pre-processing

Loading data

In [453]:
```
df_train = pd.read_csv(file_dir+r'\train.csv')
df_train
```

Out[453]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pix
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
41996	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
41997	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
41998	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
41999	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	

42000 rows × 785 columns

In [454]:
```
df_test = pd.read_csv(file_dir+r'\test.csv')
df_test
```

In [454]:
```
df_test = pd.read_csv(file_dir+r'\test.csv')
df_test
```

Out[454]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
27995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
27996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
27997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
27998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
27999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

Checking for missing values

In [112]:
```
df_train.isna().any().describe()
```

Out[112]:
```
count        785
unique         1
top        False
freq         785
dtype: object
```

In [113]:
```
df_test.isna().any().describe()
```

Out[113]:
```
count        784
unique         1
top        False
freq         784
dtype: object
```

In [114]:
```
df_train.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
```

In [115]:
```
X = df_train.iloc[:,1:]
y = df_train.iloc[:, 0]
print(X.shape, y.shape)
```

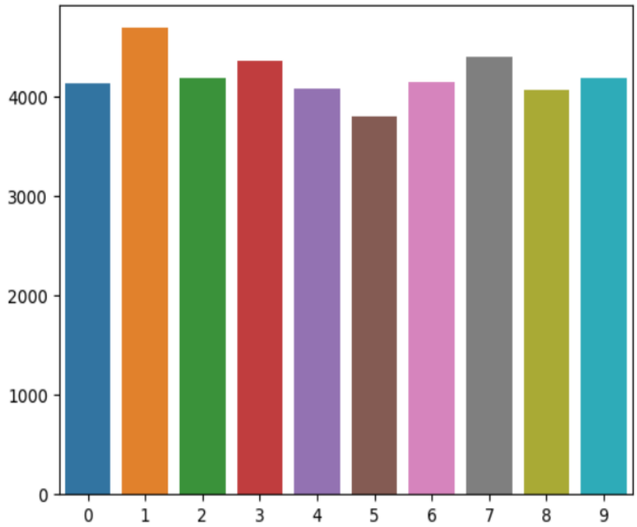
```
(42000, 784) (42000,)
```

No missing data, let's continue.

Data and target class visualizations

In [116]:
```
sns.barplot(x=y.value_counts().index, y=y.value_counts().values)
```

Out[116]: <AxesSubplot: >

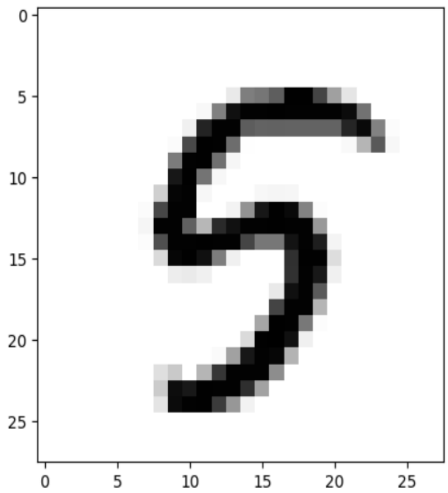


As you can see, there is a fairly even class distribution.

In [117]:
```
z = np.reshape(X.iloc[8].values, (28,28))
print(z.shape)
plt.imshow(z, cmap='Greys')
```

```
(28, 28)
```

Out[117]: <matplotlib.image.AxesImage at 0x1b69391a8b0>



Creating training and validation sets

In [13]:
```
from sklearn.model_selection import train_test_split
```

In [346]:
```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.1, random_state=15)
print(X_train.shape, X_val.shape, y_train.shape, y_val.shape)
```

```
(37800, 784) (4200, 784) (37800,) (4200,)
```

Pre-processing pipelines

In [347]:
```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

In [348]:
```
from sklearn.base import BaseEstimator, TransformerMixin
```

In [349]:
```
class ReshapeFunc(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass
    def fit(self, X, y=None):
        return self
    def transform(self, X, y=None):
        X = X.reshape((-1,28,28,1))
        return X
```

In [350]:
```
features_pipeline = Pipeline(steps=[
    ('Normalize', MinMaxScaler()),
    ('Reshape', ReshapeFunc())
])
```

Feature pipeline. Data is scaled between 0 and 1 and then reshaped into input format.

In [351]:
```
X_train = features_pipeline.fit_transform(X_train)
```

In [352]:
```
from sklearn.preprocessing import OneHotEncoder
target_pipeline = Pipeline(steps=[
    ('OneHot', OneHotEncoder())
])
```

####. One hot encoding is used, as we will be using a softmax activation function in the output node.*

In [353]:
```
y_train = target_pipeline.fit_transform(y_train.values.reshape(-1,1))
```

In [354]:
```
y_train = y_train.toarray()
```

In [355]:
```
print(X_train.shape, y_train.shape)
```

```
(37800, 28, 28, 1) (37800, 10)
```

In [356]:
```
X_val = features_pipeline.fit_transform(X_val)
```

In [357]:
```
y_val = target_pipeline.fit_transform(y_val.values.reshape(-1, 1))
```

In [358]:
```
y_val = y_val.toarray()
```

In [359]:
```
print(X_val.shape, y_val.shape)
```

```
(4200, 28, 28, 1) (4200, 10)
```

In [360]:
```
X_test = features_pipeline.fit_transform(df_test)
```

In [361]:
```
from keras import backend as K
```

In [362]:

```python
# Precision (using keras backend)
def precision_metric(y_true, y_pred):
    threshold = 0.5  # Training threshold 0.5
    y_pred_y = K.cast(K.greater(K.clip(y_pred, 0, 1), threshold), K.floatx())

    true_positives = K.sum(K.clip(y_true * y_pred, 0, 1))
    false_negatives = K.sum(K.clip(y_true * (1-y_pred), 0, 1))
    false_positives = K.sum(K.clip((1-y_true) * y_pred, 0, 1))
    true_negatives = K.sum(K.clip((1 - y_true) * (1-y_pred), 0, 1))

    precision = true_positives / (true_positives + false_positives + K.epsilon())
    return precision

# Recall (using keras backend)
def recall_metric(y_true, y_pred):
    threshold = 0.5 #Training threshold 0.5
    y_pred = K.cast(K.greater(K.clip(y_pred, 0, 1), threshold), K.floatx())

    true_positives = K.sum(K.clip(y_true * y_pred, 0, 1))
    false_negatives = K.sum(K.clip(y_true * (1-y_pred), 0, 1))
    false_positives = K.sum(K.clip((1-y_true) * y_pred, 0, 1))
    true_negatives = K.sum(K.clip((1 - y_true) * (1-y_pred), 0, 1))

    recall = true_positives / (true_positives + false_negatives + K.epsilon())
    return recall

# F1-score (using keras backend)
def f1_metric(y_true, y_pred):
    precision = precision_metric(y_true, y_pred)
    recall = recall_metric(y_true, y_pred)
    f1 = 2 * ((precision * recall) / (recall+precision+K.epsilon()))
    return f1
```

In [363]:

```python
def build_model():
    inp = keras.Input(shape=(28,28,1))
    x = keras.layers.Conv2D(filters=32, kernel_size=(5,5), strides=(1,1),padding='SAME',
                          activation='relu')(inp)
    x = keras.layers.MaxPool2D(pool_size=(2,2))(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Conv2D(filters=64, kernel_size=(5,5), padding='SAME', activation='relu')(x)
    x = keras.layers.MaxPool2D(pool_size=(2,2))(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Flatten()(x)
    x = keras.layers.Dense(256, activation='relu')(x)
    x = keras.layers.Dropout(0.5)(x)
    output = keras.layers.Dense(10, activation='softmax')(x)

    model = keras.Model(inputs=inp, outputs=output)

    model.compile(loss=keras.losses.CategoricalCrossentropy(), optimizer=keras.optimizers.Adam(learning_rate=0.0001), metrics
    return model, inp, output
```

The model is built and compiled using categorical crossentropy and adam optimizer.

In [364]:

```python
model, inp, out = build_model()
model.summary()
```

```
Model: "model_8"
```

```
Model: "model_8"

_________________________________________________________________

 Layer (type)                Output Shape              Param #

=================================================================

 input_8 (InputLayer)        [(None, 28, 28, 1)]       0


 conv2d_14 (Conv2D)          (None, 28, 28, 32)        832


 max_pooling2d_14 (MaxPoolin  (None, 14, 14, 32)       0

 g2D)


 batch_normalization_14 (Bat  (None, 14, 14, 32)       128

 chNormalization)


 dropout_21 (Dropout)        (None, 14, 14, 32)        0


 conv2d_15 (Conv2D)          (None, 14, 14, 64)        51264


max_pooling2d_15 (MaxPoolin  (None, 7, 7, 64)           0

g2D)


batch_normalization_15 (Bat  (None, 7, 7, 64)           256

chNormalization)


dropout_22 (Dropout)         (None, 7, 7, 64)           0


flatten_7 (Flatten)          (None, 3136)               0


dense_14 (Dense)             (None, 256)                803072


dropout_23 (Dropout)         (None, 256)                0


dense_15 (Dense)             (None, 10)                 2570


=================================================================
```

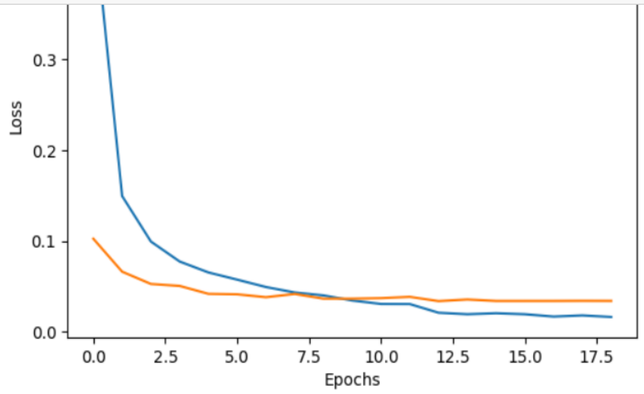
Training network

In [366]:
```python
batch_size=32
```

In [367]:
```python
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=40, batch_size=batch_size,
                    callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss',mode='min',patience=10,
                                                             min_delta=0.005, restore_best_weights=True),
                               keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', patience = 3)])
```

```
etric: 0.9910 - val_precision_metric: 0.9892 - lr: 1.0000e-05

Epoch 17/40

1182/1182 [==============================] - 9s 7ms/step - loss: 0.0168 - accuracy: 0.9949 - f1_metric: 0.9925 - recall_m
etric: 0.9945 - precision_metric: 0.9906 - val_loss: 0.0340 - val_accuracy: 0.9910 - val_f1_metric: 0.9901 - val_recall_m
etric: 0.9910 - val_precision_metric: 0.9891 - lr: 1.0000e-06

Epoch 18/40

1182/1182 [==============================] - 10s 8ms/step - loss: 0.0180 - accuracy: 0.9945 - f1_metric: 0.9920 - recall_
metric: 0.9938 - precision_metric: 0.9902 - val_loss: 0.0341 - val_accuracy: 0.9907 - val_f1_metric: 0.9899 - val_recall_
metric: 0.9908 - val_precision_metric: 0.9891 - lr: 1.0000e-06

Epoch 19/40

1182/1182 [==============================] - 9s 8ms/step - loss: 0.0163 - accuracy: 0.9946 - f1_metric: 0.9924 - recall_m
etric: 0.9942 - precision_metric: 0.9906 - val_loss: 0.0340 - val_accuracy: 0.9910 - val_f1_metric: 0.9901 - val_recall_m
etric: 0.9910 - val_precision_metric: 0.9891 - lr: 1.0000e-06
```

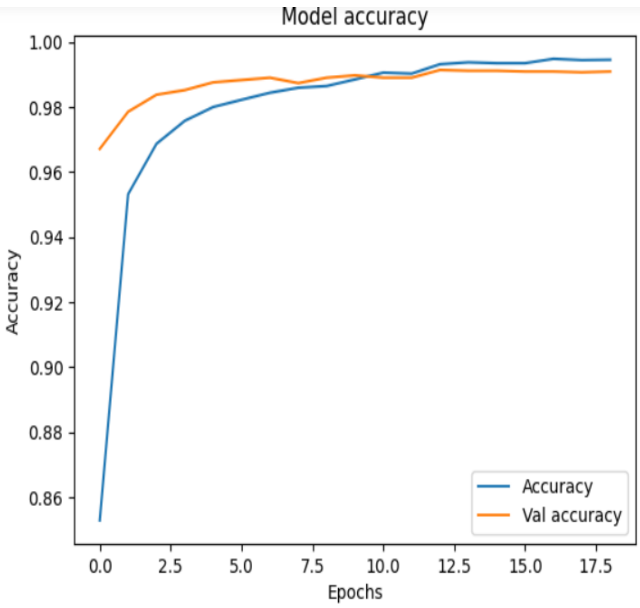
In [368]:
```python
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Loss','Val loss'])
```



In [369]:
```python
plt.plot(history.history['f1_metric'])
plt.plot(history.history['val_f1_metric'])
plt.title('Model F1')
plt.xlabel('Epochs')
plt.ylabel('F1')
plt.legend(['Training f1','Validation f1'])
```

In [370]:
```python
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Accuracy','Val accuracy'])
```

Out[370]: <matplotlib.legend.Legend at 0x1b690c2a640>



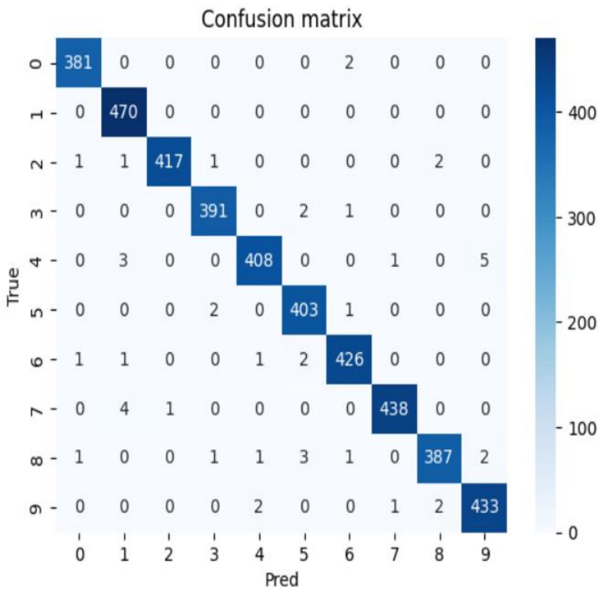
Looking at the loss plots, the network is converging well.

In [371]:
```
from sklearn.metrics import confusion_matrix
```

In [372]:
```
y_val_pred = np.argmax(model.predict(X_val), axis=1)
y_val_pred
```

In [373]:
```
cm = confusion_matrix(np.argmax(y_val,axis=1), y_val_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.title('Confusion matrix')
plt.xlabel('Pred')
plt.ylabel('True')
```

Out[373]: `Text(50.722222222222214, 0.5, 'True')`



True \ Pred	0	1	2	3	4	5	6	7	8	9
0	381	0	0	0	0	0	2	0	0	0
1	0	470	0	0	0	0	0	0	0	0
2	1	1	417	1	0	0	0	0	2	0
3	0	0	0	391	0	2	1	0	0	0
4	0	3	0	0	408	0	0	1	0	5
5	0	0	0	2	0	403	1	0	0	0
6	1	1	0	0	1	2	426	0	0	0
7	0	4	1	0	0	0	0	438	0	0
8	1	0	0	1	1	3	1	0	387	2
9	0	0	0	0	2	0	0	1	2	433

Confusion matrix looks really good as well, lots of predictions along the diagonal which is what we want to see.

In [374]:
```
y_val_true = np.argmax(y_val,axis=1)
y_val_true
```

Out[374]:
```
array([6, 2, 7, ..., 3, 1, 5], dtype=int64)
```

In [375]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_val_true, y_val_pred)
```

Out[375]:
```
0.9890476190476191
```

Ok, not bad for our first model! The model has done a pretty good job at classifying each class and is obtaining a high accuracy score. Now we should take a look at the examples the model is misclassifying.

In [376]:
```
y_test_pred = model.predict(X_test)
y_test_pred = np.argmax(y_test_pred,axis=1)
test_results = pd.DataFrame({'ImageID': np.arange(1,28001,1), 'Label': y_test_pred})
```

```
875/875 [==============================] - 1s 2ms/step
```

The class with the most confusions is between class 5 and 6, let's take a look at the incorrectly predicted examples.

In [377]:
```
cm_index = cm
np.fill_diagonal(cm_index,0)
cm_index = np.where(cm_index==cm_index.max())
if len(cm_index)>1:
    cm_index = [cm_index[0][0],cm_index[1][0]]
cm_index
```

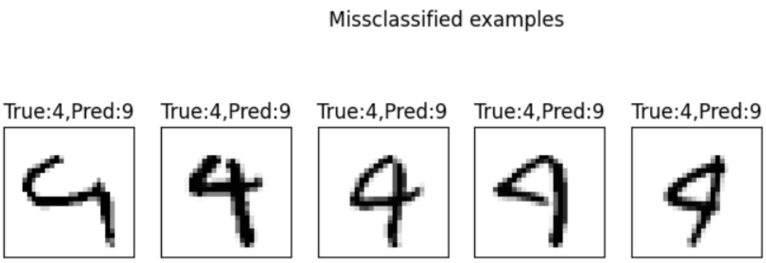
Out[377]:
```
[4, 9]
```

In [378]:
```
cm_index
```

Out[378]:
```
[4, 9]
```

In [379]:
```
nine_incorrect_examples = X_val[(y_val_true==cm_index[0]) & (y_val_pred==cm_index[1])]
nine_incorrect_examples = nine_incorrect_examples.reshape(-1,28,28)
```

In [380]:
```
filters_layer_1 = model.layers[1].get_weights()[0]
filters_layer_1 = np.squeeze(filters_layer_1).reshape(32,5,5)

fig = plt.figure(figsize=(10, 10))
fig.suptitle('Missclassified examples')
w = 10
h = 10
columns = 6
rows = 6
for i in range(columns*rows +1):
    if i==nine_incorrect_examples.shape[0]:
        break
    fig.add_subplot(rows, columns, i+1)
    plt.imshow(nine_incorrect_examples[i], cmap='Greys')
    plt.title('True:{},Pred:{}'.format(cm_index[0],cm_index[1]))
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Looking at the missclassifications, its understandable why the model was unable classify these examples correctly. Some of the examples are quite ambiguous, even a human labeller would probably be unable to clearly label them with good confidence.

However, it does seem that there is room for improvement for some of the examples. With the use of data augmentation and hyper-parameter tuning, we should be able to further improve performance.

Before we try these additional techniques, let's take a quick look at the learnt convolutional filters and feature maps, which should give us some insight on how the network is learning.

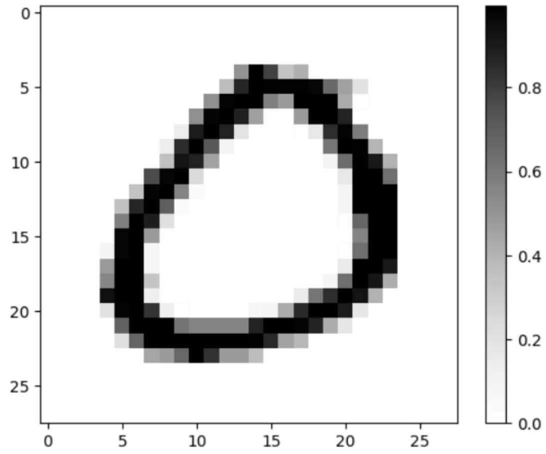
In [381]:
```
model.layers
```
Out[381]:
```
[<keras.engine.input_layer.InputLayer at 0x1b69113e670>,
 <keras.layers.convolutional.conv2d.Conv2D at 0x1b690410c10>,
 <keras.layers.pooling.max_pooling2d.MaxPooling2D at 0x1b6911228b0>,
 <keras.layers.normalization.batch_normalization.BatchNormalization at 0x1b690410f70>,
 <keras.layers.regularization.dropout.Dropout at 0x1b69449c0d0>,
 <keras.layers.convolutional.conv2d.Conv2D at 0x1b6905fc6a0>,
 <keras.layers.pooling.max_pooling2d.MaxPooling2D at 0x1b690306e20>,
 <keras.layers.normalization.batch_normalization.BatchNormalization at 0x1b691250b50>,
 <keras.layers.regularization.dropout.Dropout at 0x1b690410af0>,
 <keras.layers.reshaping.flatten.Flatten at 0x1b6905fc310>,
 <keras.layers.core.dense.Dense at 0x1b692bd4fa0>,
 <keras.layers.regularization.dropout.Dropout at 0x1b69061d520>,
 <keras.layers.core.dense.Dense at 0x1b6906219d0>]
```

In [382]:
```
model.layers[1].get_weights()[0].shape
```
Out[382]:
```
(5, 5, 1, 32)
```

In [384]:
```
for i in range(len(model.layers)):
    layer = model.layers[i]
    # check for convolutional layer
    if 'conv' not in layer.name:
        continue
    # summarize output shape
    print(i, layer.name, layer.output.shape)
```
```
1 conv2d_14 (None, 28, 28, 32)

5 conv2d_15 (None, 14, 14, 64)
```

In [385]:
```


successive_outputs = [layer.output for layer in model.layers[1:]]
fm_model = keras.Model(inputs=model.input, outputs=successive_outputs)

successive_outputs
```
Out[385]:
```
[<KerasTensor: shape=(None, 28, 28, 32) dtype=float32 (created by layer 'conv2d_14')>,
 <KerasTensor: shape=(None, 14, 14, 32) dtype=float32 (created by layer 'max_pooling2d_14')>,
 <KerasTensor: shape=(None, 14, 14, 32) dtype=float32 (created by layer 'batch_normalization_14')>,
 <KerasTensor: shape=(None, 14, 14, 32) dtype=float32 (created by layer 'dropout_21')>,
 <KerasTensor: shape=(None, 14, 14, 64) dtype=float32 (created by layer 'conv2d_15')>,
 <KerasTensor: shape=(None, 7, 7, 64) dtype=float32 (created by layer 'max_pooling2d_15')>,
 <KerasTensor: shape=(None, 7, 7, 64) dtype=float32 (created by layer 'batch_normalization_15')>,
 <KerasTensor: shape=(None, 7, 7, 64) dtype=float32 (created by layer 'dropout_22')>,
 <KerasTensor: shape=(None, 3136) dtype=float32 (created by layer 'flatten_7')>,
 <KerasTensor: shape=(None, 256) dtype=float32 (created by layer 'dense_14')>,
 <KerasTensor: shape=(None, 256) dtype=float32 (created by layer 'dropout_23')>,
 <KerasTensor: shape=(None, 10) dtype=float32 (created by layer 'dense_15')>]
```

In [386]:
```
test_example = X_train[[9]]
plt.imshow(test_example[0], cmap='Greys')
plt.colorbar()
plt.show()
successive_feature_maps = fm_model.predict(test_example)
len(successive_feature_maps) # 12 for 12 layers
```




```
1/1 [==============================] - 0s 81ms/step
```

Out[386]: 12

Now that we have a test example, lets run it through the network to obtain and visualize the feature-maps.

In [387]:

```
layer_names = [layer.name for layer in model.layers]
for layer_name, feature_map in zip(layer_names, successive_feature_maps):
    print(feature_map.shape)
    print(layer_name)
    if len(feature_map.shape) == 4:
        n_features = feature_map.shape[-1]  # number of features in the feature map
        size       = feature_map.shape[ 1]  # feature map shape (1, size, size, n_features)

        # We will tile our images in this matrix
        display_grid = np.zeros((size, size * n_features))

        # Postprocess the feature to be visually palatable
        for i in range(n_features):
            x  = feature_map[0, :, :, i]
            x -= x.mean()
            x /= x.std ()
            x *=  64
            x += 128
            x  = np.clip(x, 0, 255).astype('uint8')
            # Tile each filter into a horizontal grid
            display_grid[:, i * size : (i + 1) * size] = x

        # Display the grid
        scale = 20. / n_features
        plt.figure( figsize=(scale * n_features, scale) )
        plt.title(layer_name)
        plt.grid(False)
        plt.imshow(display_grid, aspect='auto', cmap='Greys' )
        plt.colorbar()

plt.show()
```

```
(1, 28, 28, 32)
```

input_8

(1, 14, 14, 32)

conv2d_14

(1, 14, 14, 32)

max_pooling2d_14

(1, 14, 14, 32)

batch_normalization_14

(1, 14, 14, 64)

dropout_21

(1, 7, 7, 64)

conv2d_15

(1, 7, 7, 64)

max_pooling2d_15

(1, 7, 7, 64)

batch_normalization_15

(1, 3136)

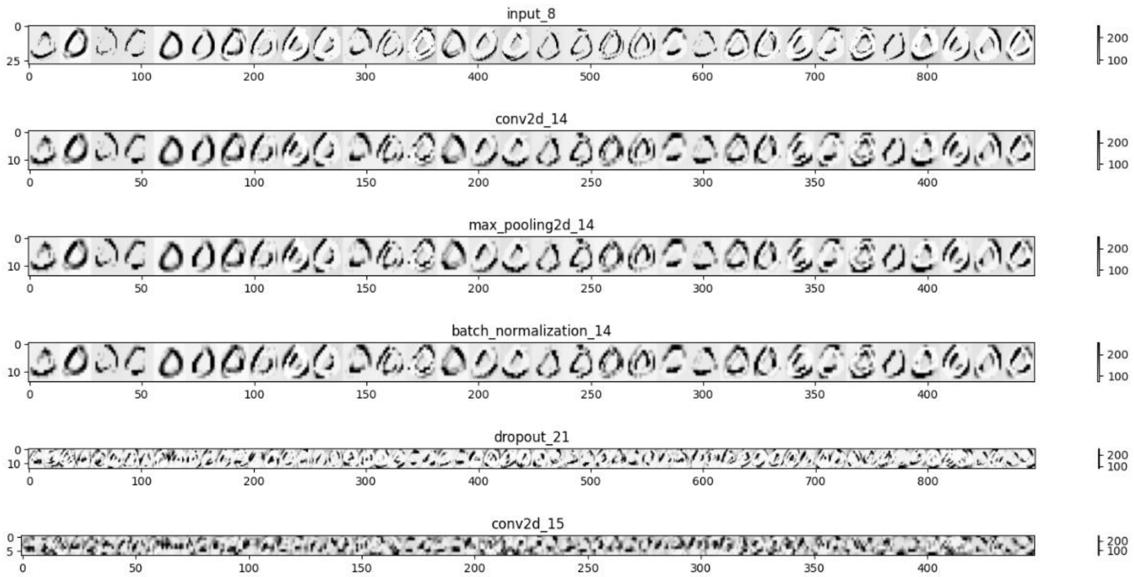
dropout_22

(1, 256)

dense_14

(1, 10)

dropout_23



Creating an ImageDataGenerator

In [407]:
```
from keras.preprocessing.image import ImageDataGenerator
```

In [408]:
```
datagen = ImageDataGenerator(
        rotation_range=10,
        width_shift_range=0.1,
        height_shift_range=0.1,
        zoom_range = 0.10,
)
```

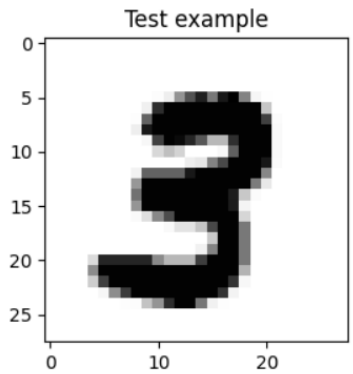
ImageDataGenerator is a brilliant class in keras, which allows us to augment images in real-time while our model is training. This means we can pass it as input to the model, and new augmented images will be generated in batches on the go.

Here is a good explation about the class in more depth: https://deepchecks.com/question/what-is-the-output-of-imagedatagenerator/

In [409]:
```
temp_example_X = X_train[1].reshape(-1,28,28,1)
temp_example_y = y_train[1].reshape(-1,10)
plt.figure(figsize=(3,3))
plt.title('Test example')
plt.imshow(temp_example_X.squeeze(), cmap='Greys')
```

Out[409]: <matplotlib.image.AxesImage at 0x1b6a05e0880>

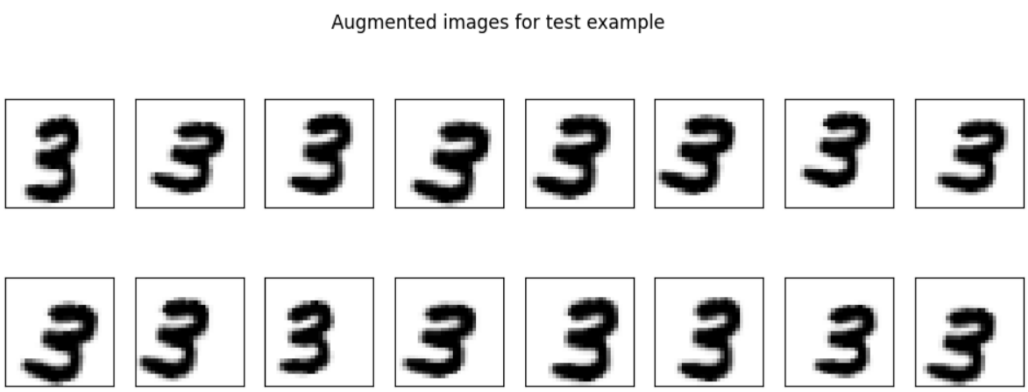
Out[409]: <matplotlib.image.AxesImage at 0x1b6a05e0880>

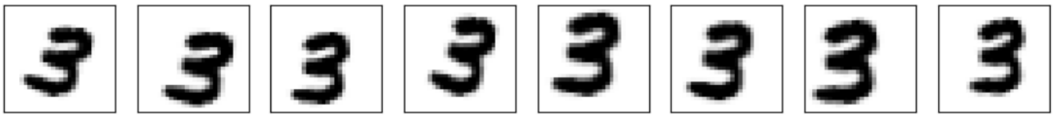


Lets select a test example for demonstration, and generate some new images using data augmentation.

Augmented images visualized

In [410]:
```
fig, axes = plt.subplots(3,8, figsize=(12,6))
fig.suptitle('Augmented images for test example')
for i in range(3):
    for j in range(8):
        augmented_example_X, augmented_example_y = datagen.flow(temp_example_X, temp_example_y, batch_size=1).next()
        axes[i,j].imshow(augmented_example_X.squeeze(), cmap='Greys')
        axes[i,j].set_xticks([])
        axes[i,j].set_yticks([])
```





Here are the artificial images generated. You can see how much variation can be created by slightly modifying just one image.

Training network on augmented data

Now we should try training the model again, except this time we will train the model using augmented data.

In [423]:
```
model2, _, _ = build_model()
```

In [424]:
```
train_generator = datagen.flow(X_train, y_train, batch_size=batch_size)
```

In [425]:
```
steps_per_epoch = train_generator.n // train_generator.batch_size
steps_per_epoch
```

Out[425]: 1181

In [426]:
```
print(train_generator.n, train_generator.batch_size)
```

```
37800 32
```

In [427]:
```
steps_per_epoch
```

Out[427]: 1181

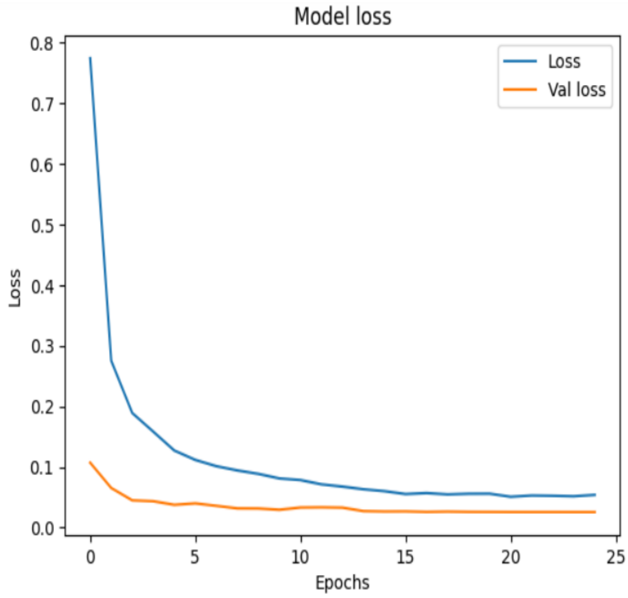
In [428]:
```
history2 = model2.fit(train_generator, validation_data=(X_val, y_val), epochs=40, steps_per_epoch=steps_per_epoch,
                       callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss',mode='min',patience=10,
                                                               min_delta=0.005, restore_best_weights=True),
                                  keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', patience = 3)])
```

```
metric: 0.9929 - val_precision_metric: 0.9896 - lr: 1.0000e-06

Epoch 23/40

1181/1181 [==============================] - 10s 8ms/step - loss: 0.0525 - accuracy: 0.9837 - f1_metric: 0.9784 - recall_
metric: 0.9823 - precision_metric: 0.9746 - val_loss: 0.0258 - val_accuracy: 0.9926 - val_f1_metric: 0.9910 - val_recall_
metric: 0.9924 - val_precision_metric: 0.9895 - lr: 1.0000e-06

Epoch 24/40

1181/1181 [==============================] - 11s 9ms/step - loss: 0.0517 - accuracy: 0.9844 - f1_metric: 0.9786 - recall_
metric: 0.9825 - precision_metric: 0.9747 - val_loss: 0.0257 - val_accuracy: 0.9929 - val_f1_metric: 0.9911 - val_recall_
metric: 0.9927 - val_precision_metric: 0.9896 - lr: 1.0000e-06

Epoch 25/40

1181/1181 [==============================] - 10s 9ms/step - loss: 0.0540 - accuracy: 0.9839 - f1_metric: 0.9784 - recall_
metric: 0.9821 - precision_metric: 0.9748 - val_loss: 0.0257 - val_accuracy: 0.9929 - val_f1_metric: 0.9910 - val_recall_
metric: 0.9924 - val_precision_metric: 0.9896 - lr: 1.0000e-07
```

Training the model on augmented data..

In [437]:
```
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Loss','Val loss'])
```

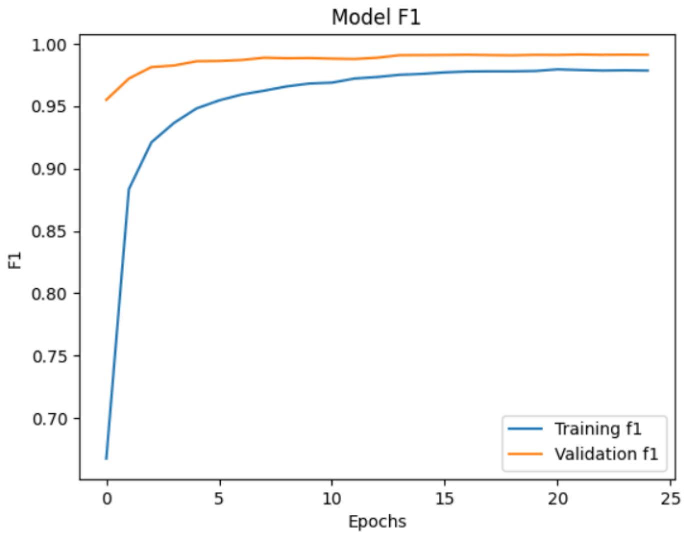
Out[437]: <matplotlib.legend.Legend at 0x1b695f82fa0>



In [438]:

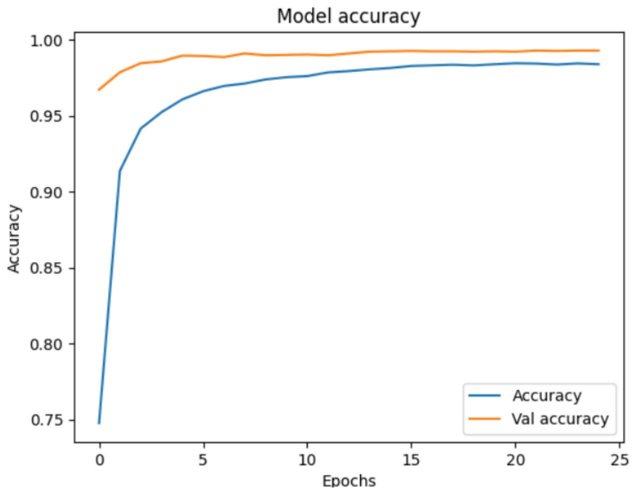
```
plt.plot(history2.history['f1_metric'])
plt.plot(history2.history['val_f1_metric'])
plt.title('Model F1')
plt.xlabel('Epochs')
plt.ylabel('F1')
plt.legend(['Training f1','Validation f1'])
```

Out[438]: <matplotlib.legend.Legend at 0x1b696c288b0>



```
In [439]: plt.plot(history2.history['accuracy'])
          plt.plot(history2.history['val_accuracy'])
          plt.title('Model accuracy')
          plt.xlabel('Epochs')
          plt.ylabel('Accuracy')
          plt.legend(['Accuracy','Val accuracy'])
```

```
Out[439]: <matplotlib.legend.Legend at 0x1b695c41850>
```



Validation results on augmented data

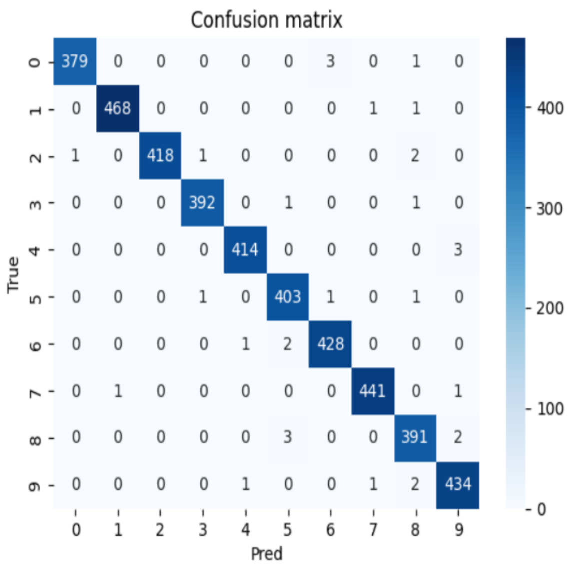
```
In [440]: y_val_pred = np.argmax(model2.predict(X_val), axis=1)
          y_val_pred
```

```
132/132 [==============================] - 0s 3ms/step
```

```
Out[440]: array([6, 2, 7, ..., 3, 1, 5], dtype=int64)
```

```
In [441]: sns.heatmap(confusion_matrix(np.argmax(y_val,axis=1), y_val_pred), annot=True, cmap='Blues', fmt='g')
          plt.title('Confusion matrix')
          plt.xlabel('Pred')
          plt.ylabel('True')
```

```
Out[441]: Text(50.722222222222214, 0.5, 'True')
```




```
In [442]: accuracy_score(y_val_true, y_val_pred)
```

```
Out[442]: 0.9923809523809524
```

```
In [443]: y_test_pred = model2.predict(X_test)
          y_test_pred = np.argmax(y_test_pred,axis=1)
          test_results = pd.DataFrame({'ImageID': np.arange(1,28001,1), 'Label': y_test_pred})

          875/875 [==============================] - 2s 2ms/step
```

Hyper-parameter tuning

Now that we've implemented a data augmentation technique, we need find the optimal hyper-parameter settings to maximize model performance. We will use the keras-tuner library, which is a hyperparameter optimization framework containing multiple tuning algorithms including RandomSearch, HyperBand and BayesianOptimization.

```
In [122]: import keras_tuner as kt
```

```
In [123]: def build_model_hp(hp):
              inp = keras.layers.Input(shape=[28,28,1])

              dropout = hp.Choice('conv_block_dropout', [0.125,0.25,0.375,0.5])
              conv_kernel_size = hp.Choice('conv_kernel_size', [5]) # Kernel size 5 is optimal after mutliple testing experiments

              n_layers = hp.Choice('n_conv_blocks', [2,3,4])

              filter_choice = hp.Choice('filter_combination_choice', [0,1,2,3])

              filter_combinations_2 = [[16,32],[32,64],[64,128],[128,256]]
              filter_combinations_3 = [[16,32,48],[16,32,64],[32,64,128],[64,128,256]]
              filter_combinations_4 = [[16,16,32,32],[32,32,64,64],[64,64,128,128],[128,128,256,256]]

              if n_layers==2:
                  filter_settings = filter_combinations_2[filter_choice]
              elif n_layers==3:
                  filter_settings = filter_combinations_3[filter_choice]
              elif n_layers==4:
                  filter_settings = filter_combinations_4[filter_choice]

              for i in range(n_layers):
                  if i == 0:
                      x = keras.layers.Conv2D(filters=filter_settings[i],
                                              kernel_size=conv_kernel_size,
                                              strides=1, padding='SAME',
                                              activation='relu')(inp)
                  else:
                      x = keras.layers.Conv2D(filters=filter_settings[i],
                                              kernel_size=conv_kernel_size,
                                              strides=1, padding='SAME',
                                              activation='relu')(x)
                  x = keras.layers.MaxPool2D(pool_size=2)(x)
                  x = keras.layers.BatchNormalization()(x)
                  x = keras.layers.Dropout(dropout)(x)

              x = keras.layers.Flatten()(x)

              n_fc_layers = hp.Choice('n_fc_layers', [1,2,3])

              fc_choice = hp.Choice('fc_units_combination_choice', [0,1])

              fc_combinations_1 = [[128],[256]]
              fc_combinations_2 = [[128,64],[256,128]]
              fc_combinations_3 = [[512,256,128],[256,128,64]]

              if n_fc_layers==1:
                  fc_units = fc_combinations_1[fc_choice]
              elif n_fc_layers==2:
                  fc_units = fc_combinations_2[fc_choice]
              elif n_fc_layers==3:
                  fc_units = fc_combinations_3[fc_choice]

              for j in range(n_fc_layers):
                  x = keras.layers.Dense(fc_units[j], activation='relu')(x)
                  x = keras.layers.Dropout(hp.Choice('fc_dropout', [0.125,0.25,0.5]))(x)

              out = keras.layers.Dense(10, activation='softmax')(x)

              model = keras.Model(inputs=inp, outputs=out)

              model.compile(loss=keras.losses.CategoricalCrossentropy(), optimizer=keras.optimizers.Adam(learning_rate=0.0001),
                            metrics=['accuracy', f1_metric, recall_metric, precision_metric])

              return model
```

In [125]:
```
tuner = kt.Hyperband(hypermodel=build_model_hp, objective='val_loss', max_epochs=50, executions_per_trial=2,
                        overwrite=False, project_name='hyperband_results')
```

```
INFO:tensorflow:Reloading Oracle from existing project .\hyperband_results\oracle.json

INFO:tensorflow:Reloading Tuner from .\hyperband_results\tuner0.json
```

I tested both RandomSearch and HyperBand, and found HyperBand to be much more successful so we will use that.

In [126]:
```
tuner.search_space_summary()
```

```
Search space summary

Default search space size: 7

conv_block_dropout (Choice)

{'default': 0.125, 'conditions': [], 'values': [0.125, 0.25, 0.375, 0.5], 'ordered': True}

conv_kernel_size (Choice)

{'default': 5, 'conditions': [], 'values': [5], 'ordered': True}

n_conv_blocks (Choice)

{'default': 2, 'conditions': [], 'values': [2, 3, 4], 'ordered': True}

filter_combination_choice (Choice)

{'default': 0, 'conditions': [], 'values': [0, 1, 2, 3], 'ordered': True}
```

In [133]:
```
tuner.search(train_generator, validation_data=(X_val, y_val), epochs=30, steps_per_epoch=steps_per_epoch,
             callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss',mode='min',patience=10,
                                                      min_delta=0.005, restore_best_weights=True),
                        keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', patience = 3)])
```

```
Trial 90 Complete [00h 10m 13s]

val_loss: 0.01791099552065134


Best val_loss So Far: 0.01662298757582903

Total elapsed time: 05h 29m 10s

INFO:tensorflow:Oracle triggered exit
```

Hyper-parameter results

In [234]:
```
top_model = tuner.get_best_models(1)[0]
top_model_hps = tuner.get_best_hyperparameters(1)[0]
print(top_model_hps.values)
top_model.summary()
```

```
 dropout_4 (Dropout)         (None, 256)               0


 dense_1 (Dense)             (None, 10)                2570


=================================================================

Total params: 755,914

Trainable params: 755,146

Non-trainable params: 768

_________________________________________________________________
```

In [235]:
```
y_val_true = np.argmax(y_val,axis=1)
y_val_pred = np.argmax(top_model.predict(X_val), axis=1)
accuracy_score(y_val_true, y_val_pred)
```

```
132/132 [==============================] - 1s 4ms/step
```

Out[235]: 0.9973809523809524

REFERENCES

https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filter-visualization-f75012a5a49c

https://www.kaggle.com/code/wonduk/explained-tensorflow-digit-classification