

Capstone Project

Image classifier for the SVHN dataset

In this notebook, I create a neural network that classifies real-world images digits from the ["Street View House Numbers" dataset](#).

Let's begin!

First, I'll import the libraries and modules that I will need throughout the notebook.

```
In [ ]: import tensorflow as tf
        from scipy.io import loadmat
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Conv2D, Flatten, BatchNormalization, MaxPool2D, Dense
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import random
        from sklearn.preprocessing import OneHotEncoder
        from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

To give some background, this dataset features over 600,000 digit images in all, and is a harder dataset than MNIST as the numbers appear in the context of natural scene images. SVHN is obtained from house numbers in Google Street View images.

- Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning". NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

The goal is to develop an end-to-end workflow for building, training, validating, evaluating and saving a neural network that classifies a real-world image into one of ten classes.

```
In [ ]: # Load the dataset (Can be downloaded from the Link above)

train = loadmat('data/train_32x32.mat')
test = loadmat('data/test_32x32.mat')
```

Both `train` and `test` are dictionaries with keys `x` and `y` for the input images and labels respectively.

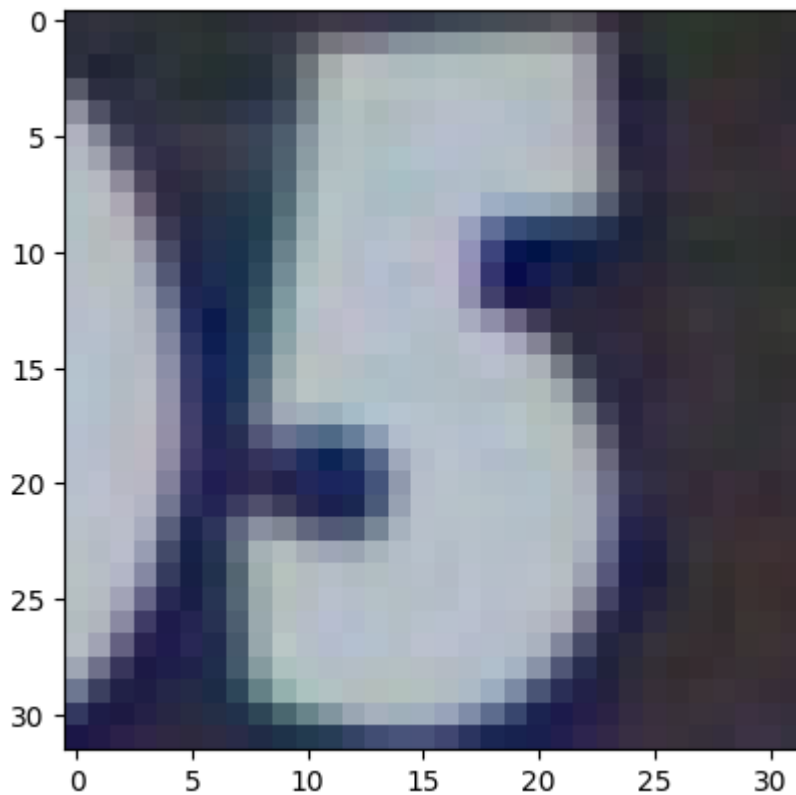
1. Inspect and preprocess the dataset

First, I extract the images from that dataset dictionaries, splitting them into separate variables. To demonstrate, I will sample 10 random images from the dataset along with their accompanying labels!

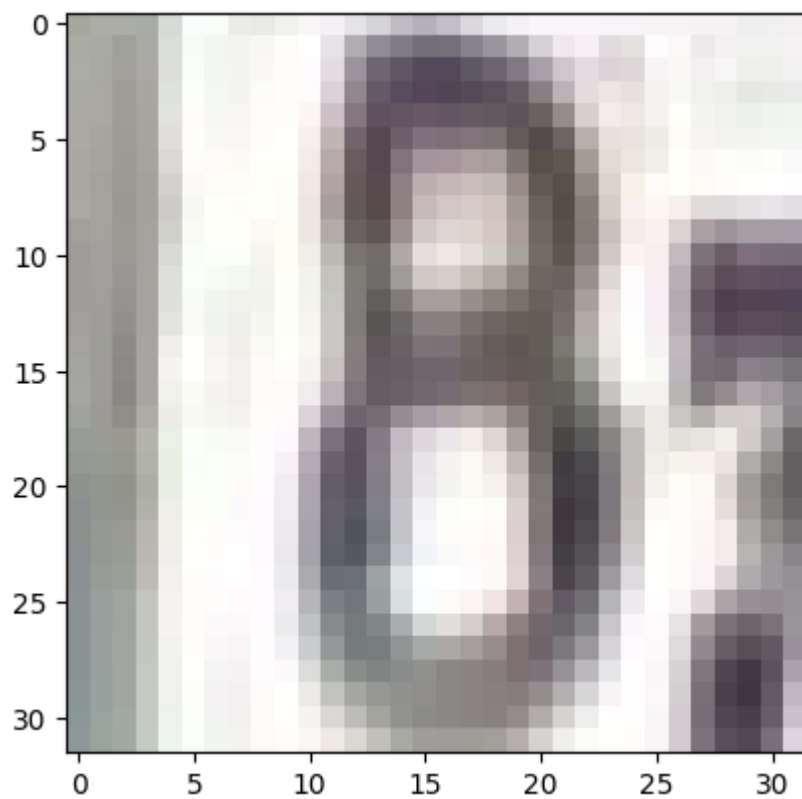
```
In [ ]: X_train = train['X']
X_test = test['X']
y_train = train['y']
y_test = test['y']

X_train = np.moveaxis(X_train, -1, 0)
X_test = np.moveaxis(X_test, -1, 0)
```

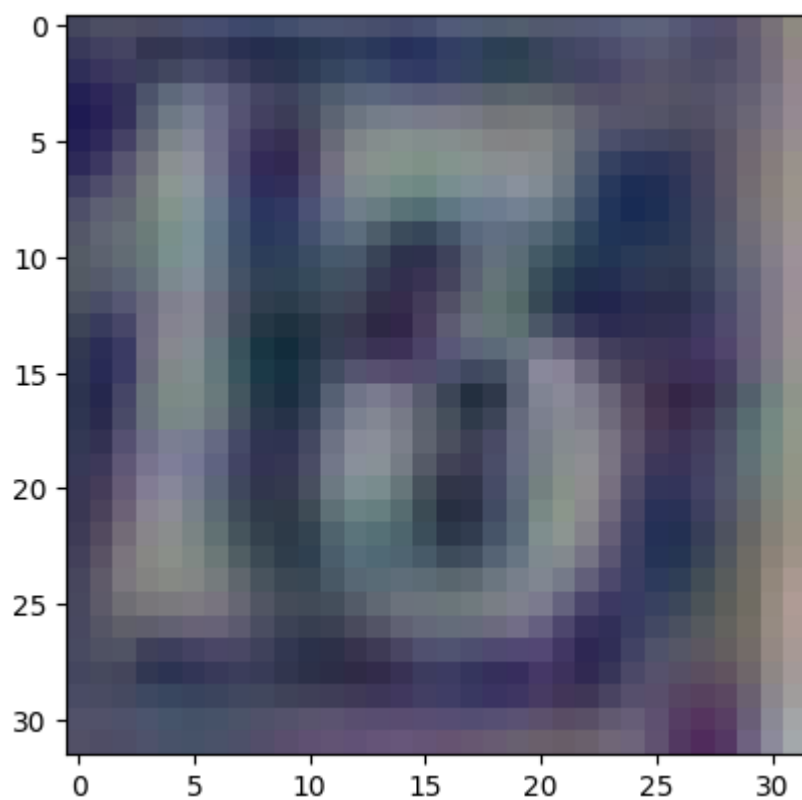
```
In [ ]: sample_number_list = random.sample(range(0,73257),10)
for i in sample_number_list:
    plt.imshow(X_train[i,:,:,:])
    plt.show()
    print('Number', y_train[i])
```



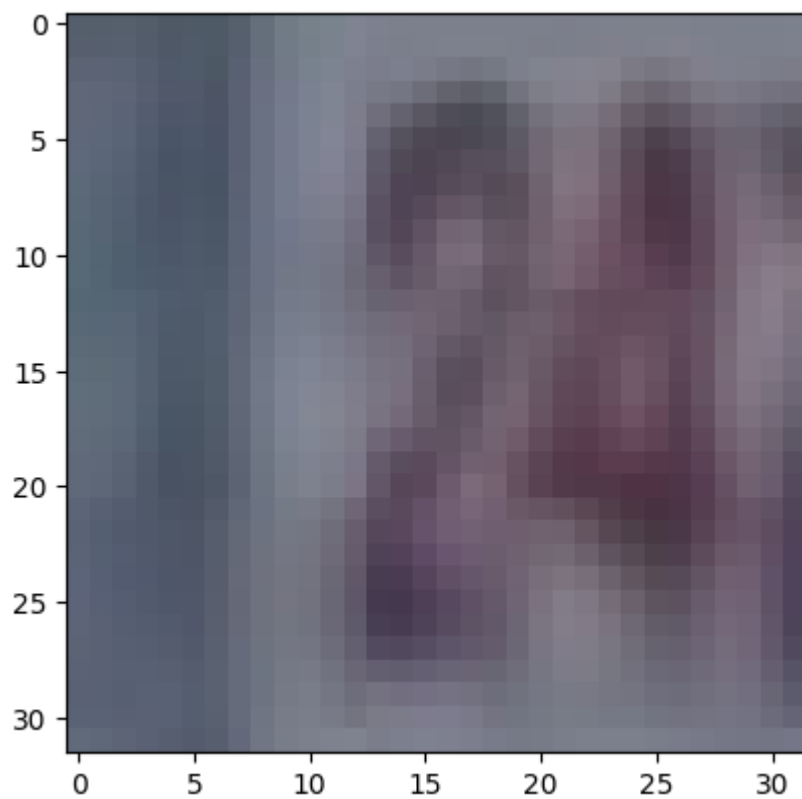
Number [5]



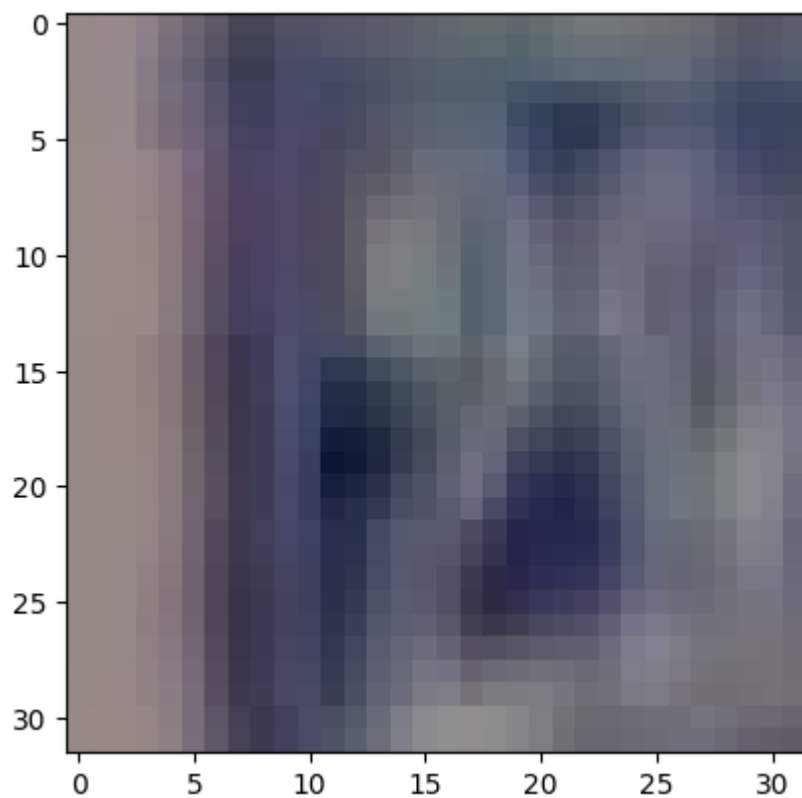
Number [8]



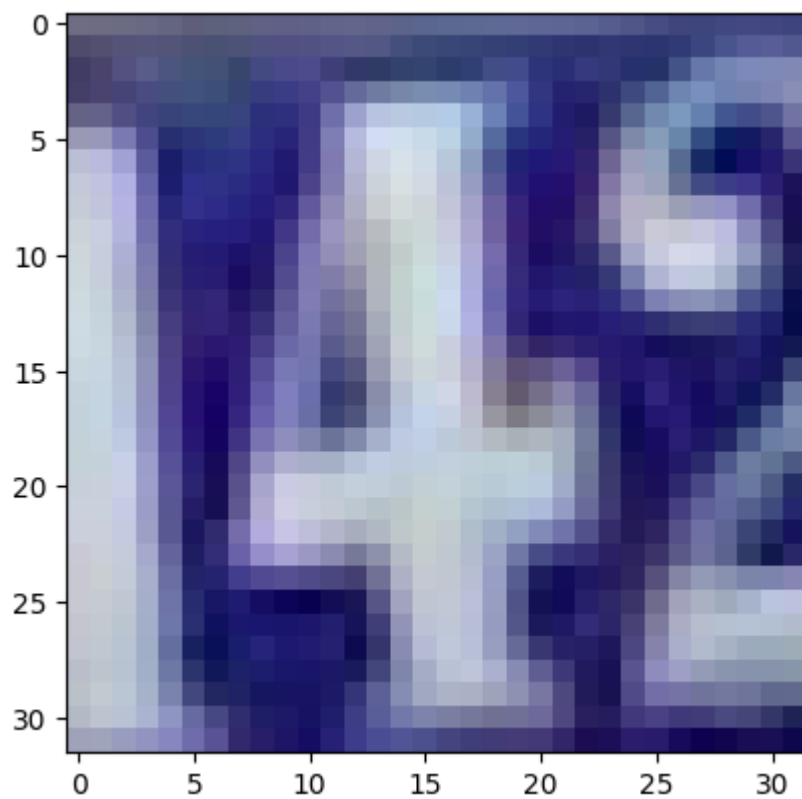
Number [3]



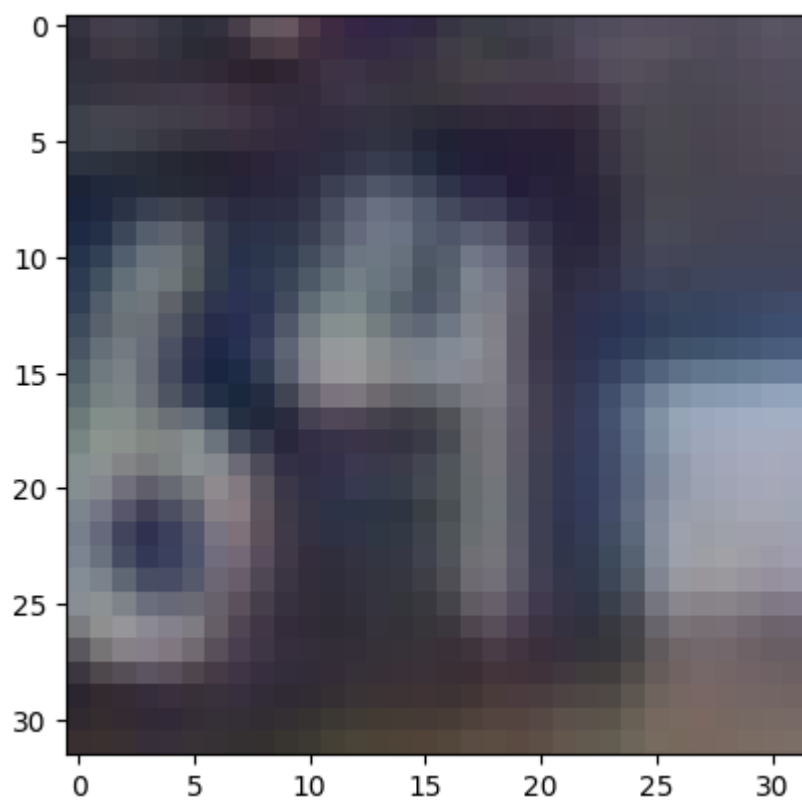
Number [2]



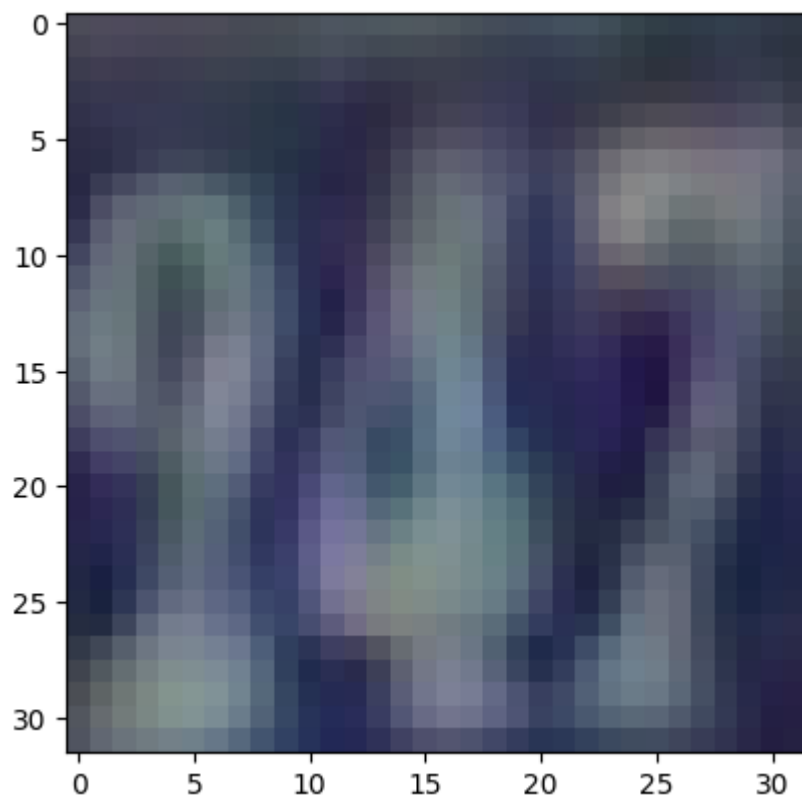
Number [2]



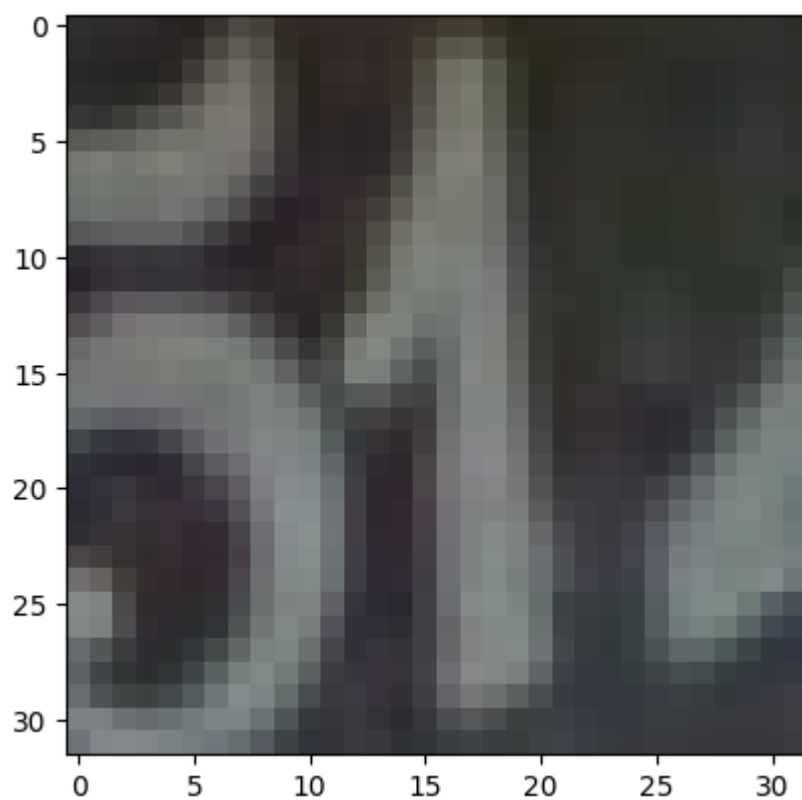
Number [4]



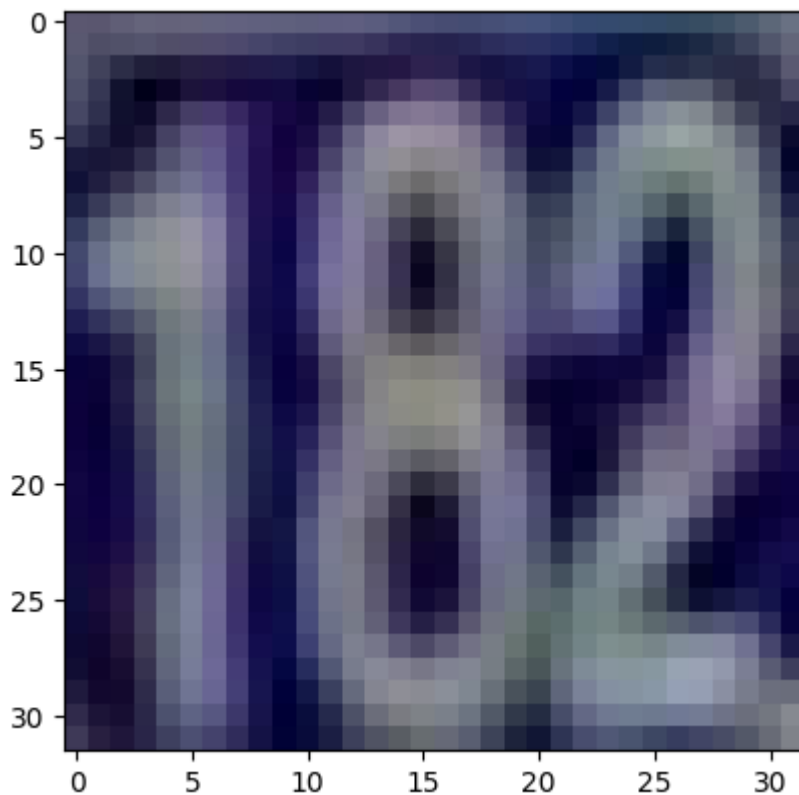
Number [4]



Number [4]



Number [1]

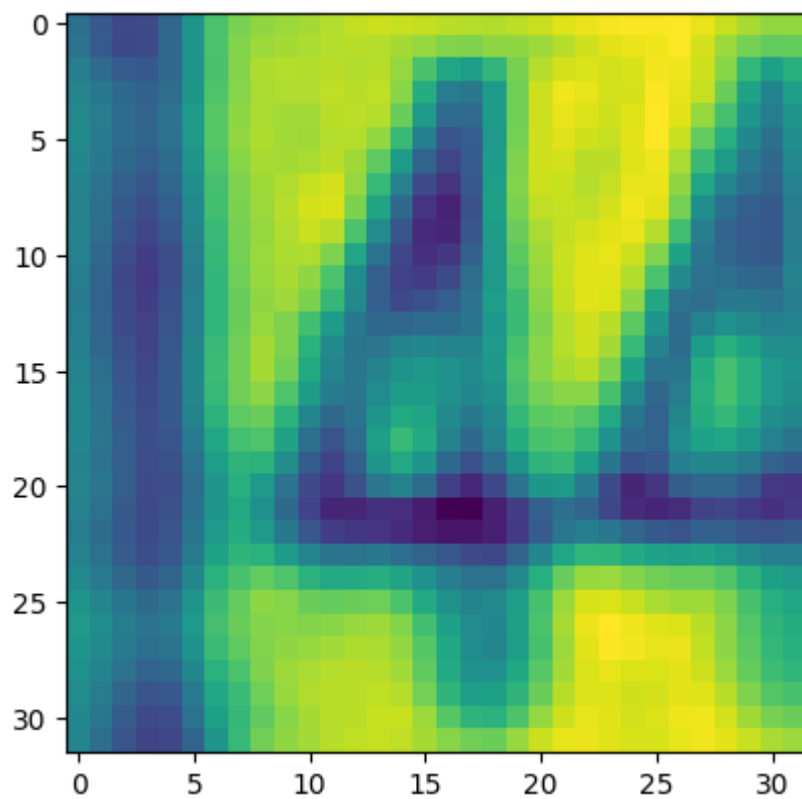


Number [8]

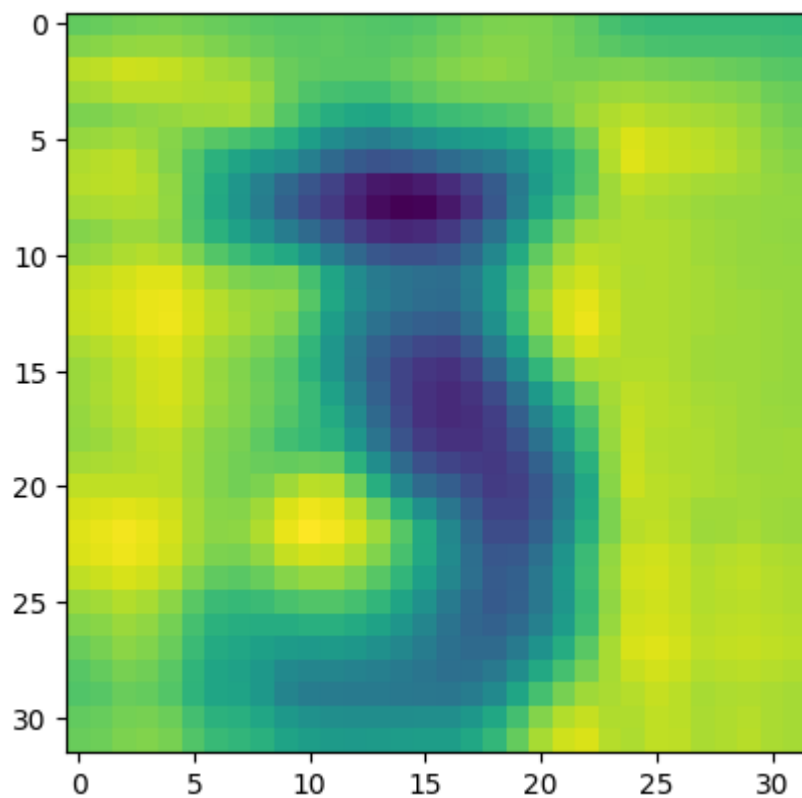
Here, I convert the images to greyscale using the `np.mean` function. This allows the image to be processed more easily, but is not necessarily required. Next, I normalize the array to values between 0 and 1 for easier processing as well.

```
In [ ]: X_train_grey = np.mean(X_train, 3).reshape(73257, 32, 32,1)/255
X_test_grey = np.mean(X_test, 3).reshape(26032, 32, 32,1)/255
X_train_plt = np.mean(X_train, 3)
```

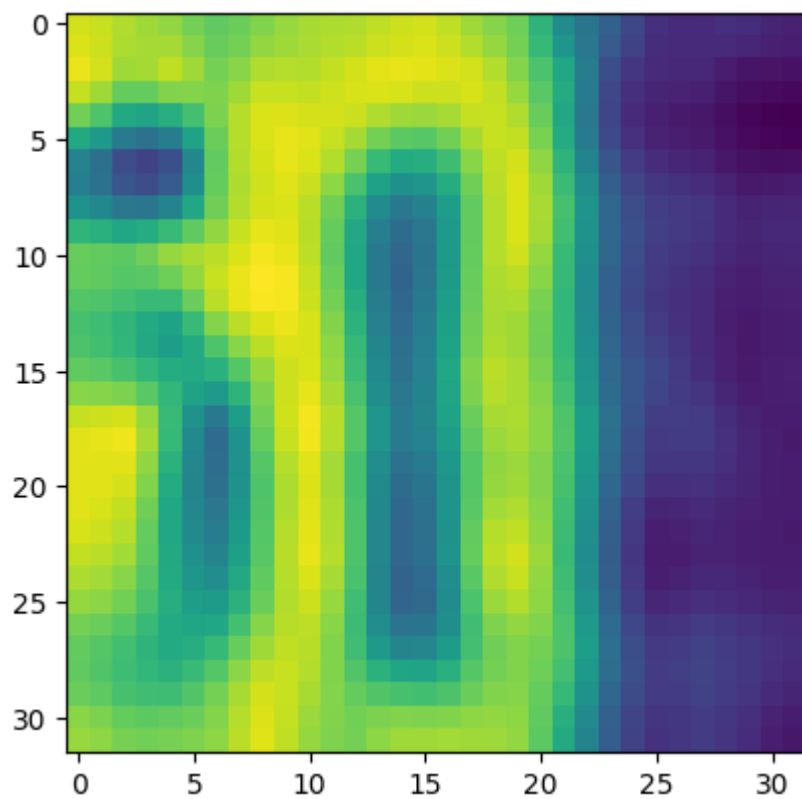
```
In [ ]: sample_number_list = random.sample(range(0,73257),10)
for i in sample_number_list:
    plt.imshow(X_train_grey[i,:,:,:0])
    plt.show()
    print('Number', y_train[i])
```



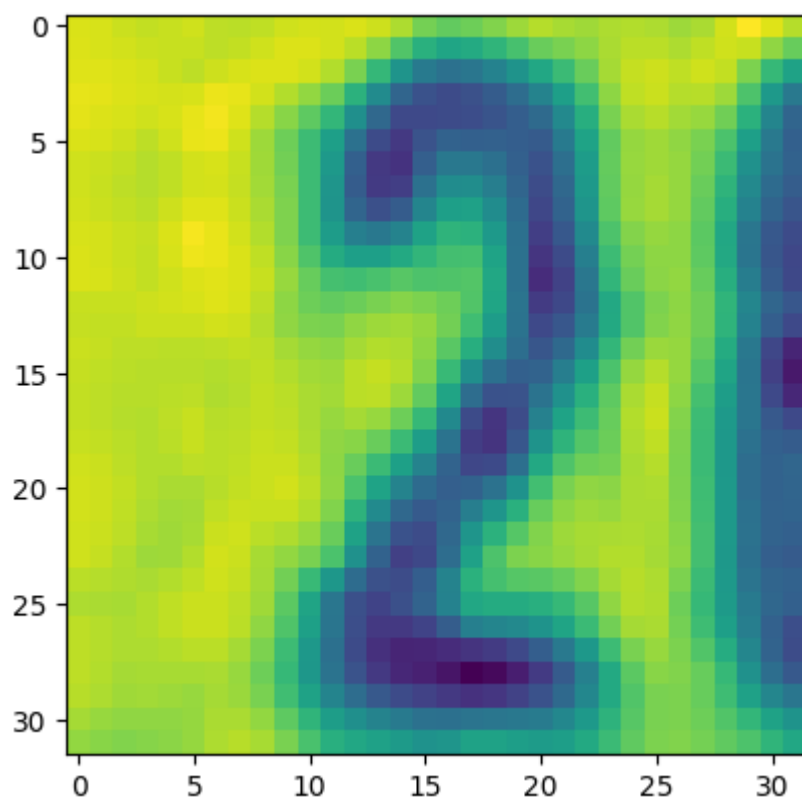
Number [4]



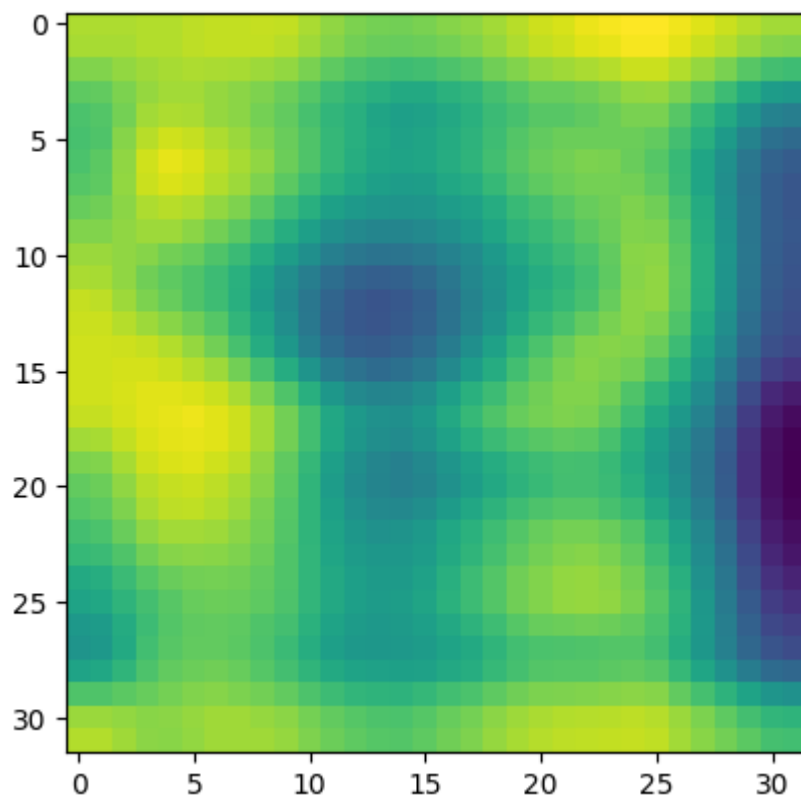
Number [3]



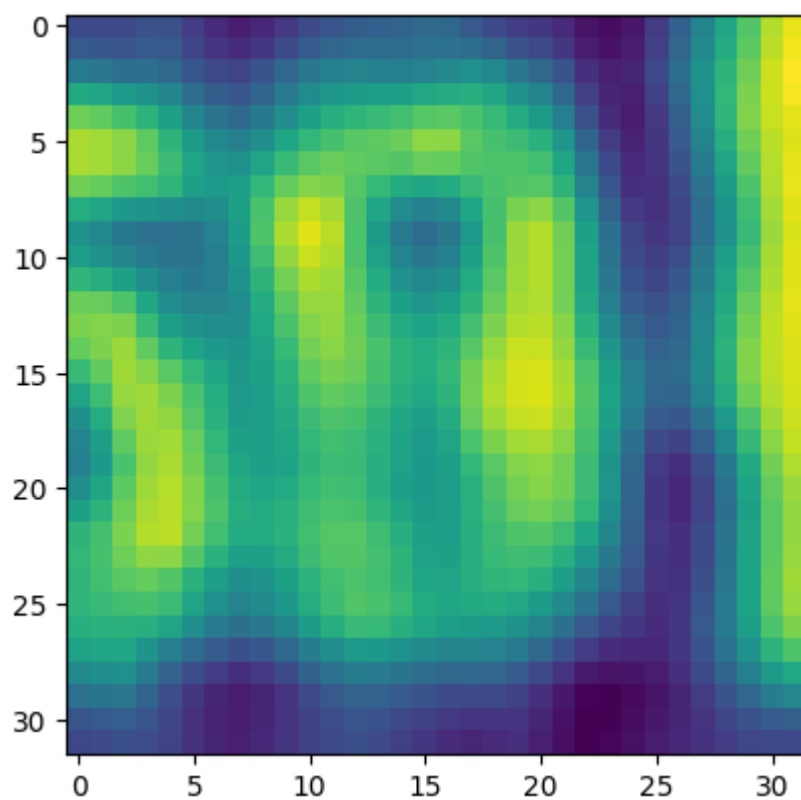
Number [1]



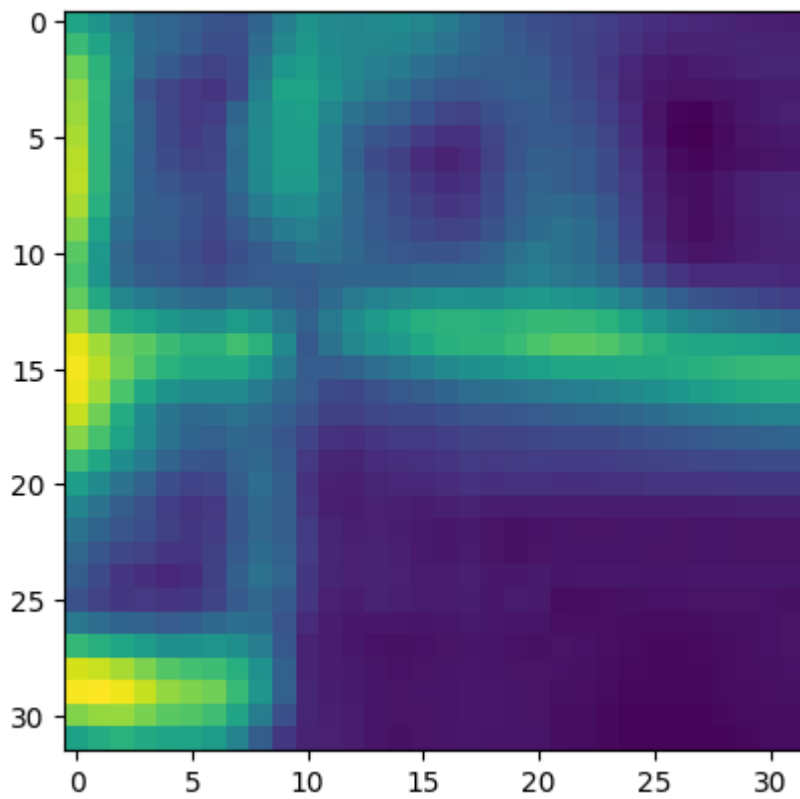
Number [2]



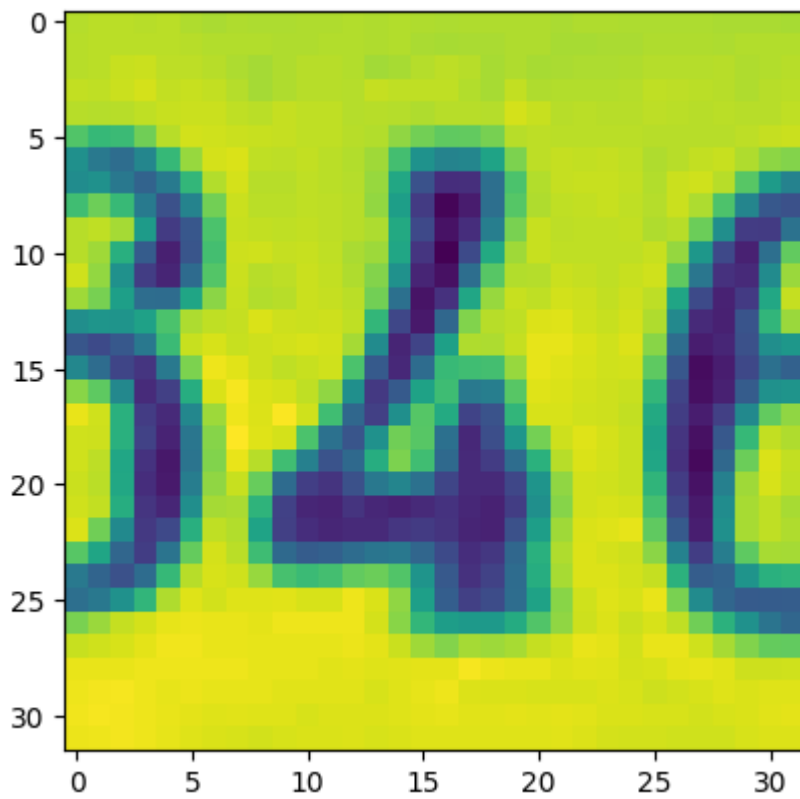
Number [6]



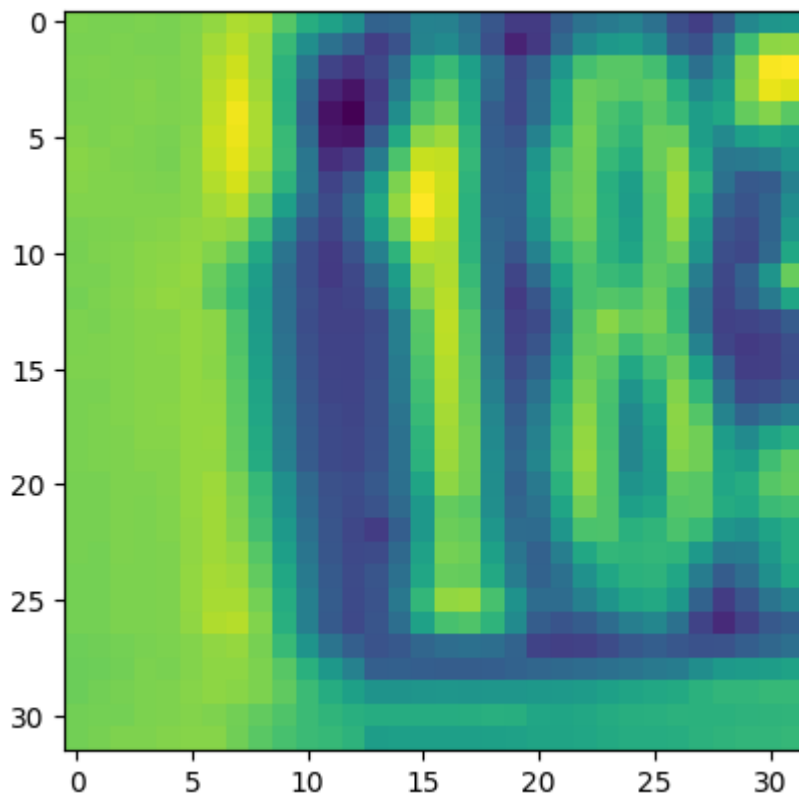
Number [10]



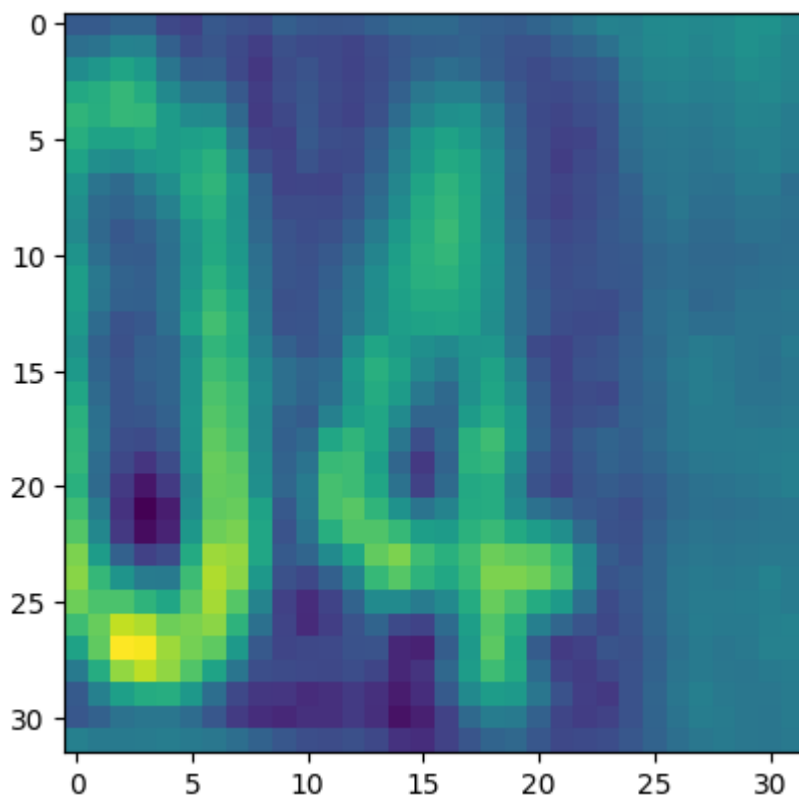
Number [4]



Number [4]



Number [1]



Number [4]

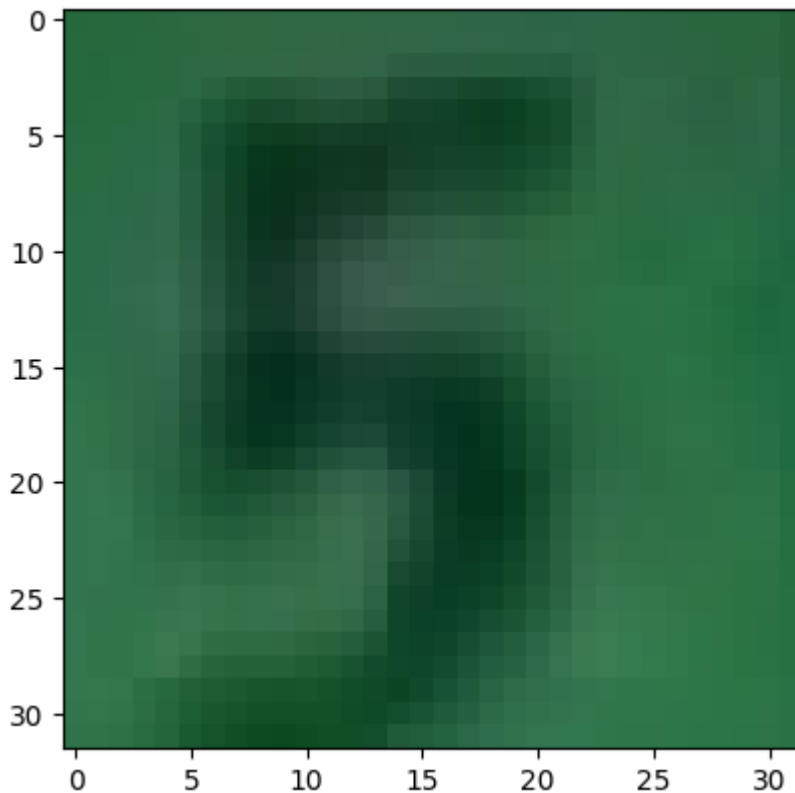
The OneHotEncoder function converts categorical features as noted in `y_train` and `y_test` to an array.

```
In [ ]: encoder=OneHotEncoder().fit(y_train)
        y_train_encoded = encoder.transform(y_train).toarray()
```

```
y_test_encoded = encoder.transform(y_test).toarray()
```

```
In [ ]: plt.imshow(X_test[0])
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x2a11a581000>
```



2. MLP neural network classifier

Now I can create my model! Throughout the next code blocks, I:

- Define basic callbacks: ModelCheckpoint and EarlyStopping
- Define my model - this basic one features 3 hidden dense layers and an output layer using the softmax activation function
- Compile my model using the Adam optimizer, calculating loss through categorical crossentropy and using accuracy as the metric for optimization
- Train my model in 20 epochs, with a batch size of 128

```
In [ ]: best_val_acc_path = 'best_val_acc/checkpoint'
best_val_acc = ModelCheckpoint(filepath = best_val_acc_path, save_weights_only=True, s
earlystopping = EarlyStopping(patience = 3, monitor='loss')

model_1 = Sequential([
    Flatten(input_shape=X_train[0].shape),
    Dense(512, activation = 'relu', ),
    Dense(128, activation = 'relu'),
    Dense(128, activation='relu'),
    Dense(10, activation = 'softmax')
])
```

```
model_1.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
flatten (Flatten)	(None, 3072)	0
dense (Dense)	(None, 512)	1573376
dense_1 (Dense)	(None, 128)	65664
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 10)	1290
=====		
Total params: 1,656,842		
Trainable params: 1,656,842		
Non-trainable params: 0		

```
In [ ]: model_1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy']
history_1 = model_1.fit(X_train, y_train_encoded, epochs=20, validation_data = (X_test
```

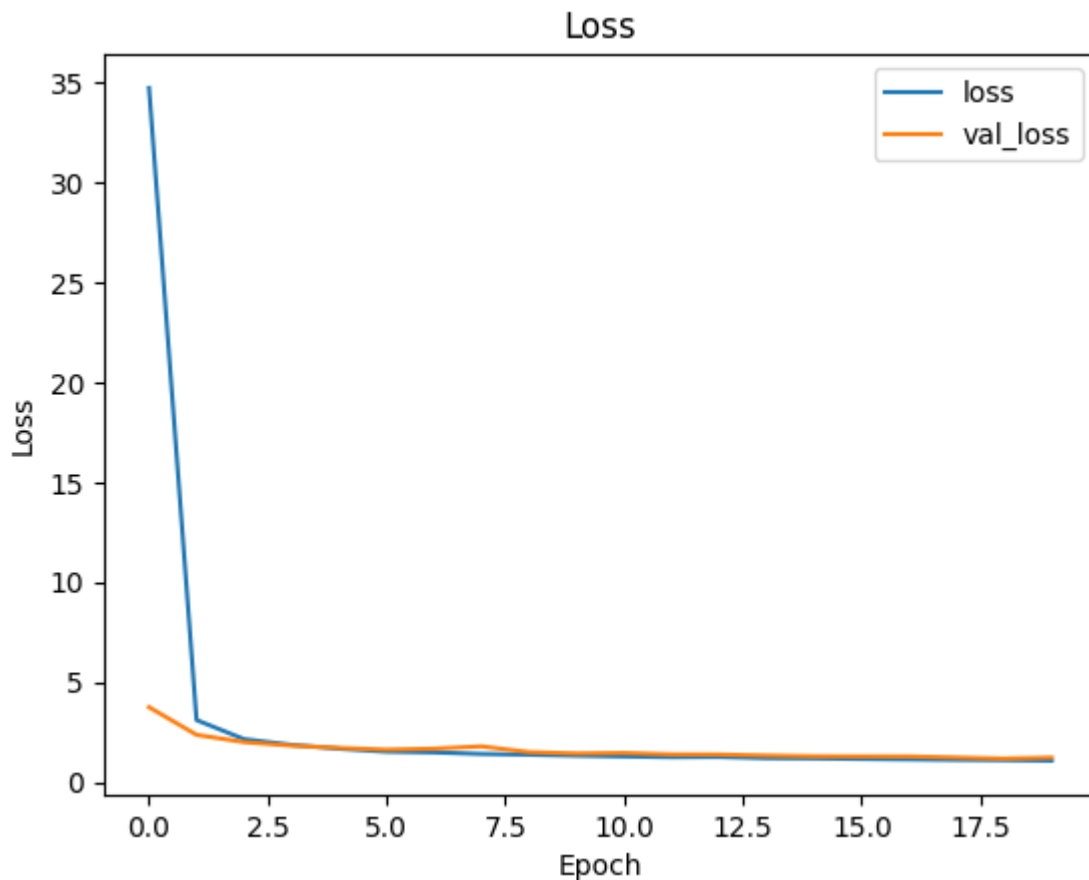
Epoch 1/20
573/573 [=====] - 8s 9ms/step - loss: 34.7351 - accuracy: 0.1283 - val_loss: 3.7419 - val_accuracy: 0.0934
Epoch 2/20
573/573 [=====] - 3s 6ms/step - loss: 3.0972 - accuracy: 0.1584 - val_loss: 2.3574 - val_accuracy: 0.2086
Epoch 3/20
573/573 [=====] - 3s 6ms/step - loss: 2.1423 - accuracy: 0.2657 - val_loss: 1.9906 - val_accuracy: 0.3607
Epoch 4/20
573/573 [=====] - 3s 6ms/step - loss: 1.8538 - accuracy: 0.3845 - val_loss: 1.8163 - val_accuracy: 0.4165
Epoch 5/20
573/573 [=====] - 3s 5ms/step - loss: 1.6627 - accuracy: 0.4550 - val_loss: 1.7018 - val_accuracy: 0.4537
Epoch 6/20
573/573 [=====] - 3s 5ms/step - loss: 1.5134 - accuracy: 0.5083 - val_loss: 1.6143 - val_accuracy: 0.5047
Epoch 7/20
573/573 [=====] - 2s 4ms/step - loss: 1.4765 - accuracy: 0.5264 - val_loss: 1.6633 - val_accuracy: 0.4974
Epoch 8/20
573/573 [=====] - 2s 4ms/step - loss: 1.4014 - accuracy: 0.5528 - val_loss: 1.7790 - val_accuracy: 0.4520
Epoch 9/20
573/573 [=====] - 3s 4ms/step - loss: 1.3640 - accuracy: 0.5645 - val_loss: 1.4923 - val_accuracy: 0.5414
Epoch 10/20
573/573 [=====] - 3s 5ms/step - loss: 1.3142 - accuracy: 0.5832 - val_loss: 1.4225 - val_accuracy: 0.5652
Epoch 11/20
573/573 [=====] - 3s 5ms/step - loss: 1.2746 - accuracy: 0.5989 - val_loss: 1.4477 - val_accuracy: 0.5579
Epoch 12/20
573/573 [=====] - 3s 4ms/step - loss: 1.2417 - accuracy: 0.6099 - val_loss: 1.3773 - val_accuracy: 0.5940
Epoch 13/20
573/573 [=====] - 2s 4ms/step - loss: 1.2517 - accuracy: 0.6062 - val_loss: 1.3695 - val_accuracy: 0.5892
Epoch 14/20
573/573 [=====] - 3s 5ms/step - loss: 1.1879 - accuracy: 0.6305 - val_loss: 1.3185 - val_accuracy: 0.5986
Epoch 15/20
573/573 [=====] - 3s 5ms/step - loss: 1.1757 - accuracy: 0.6334 - val_loss: 1.2816 - val_accuracy: 0.6166
Epoch 16/20
573/573 [=====] - 3s 5ms/step - loss: 1.1508 - accuracy: 0.6400 - val_loss: 1.2756 - val_accuracy: 0.6124
Epoch 17/20
573/573 [=====] - 3s 5ms/step - loss: 1.1294 - accuracy: 0.6484 - val_loss: 1.2750 - val_accuracy: 0.6202
Epoch 18/20
573/573 [=====] - 3s 5ms/step - loss: 1.1041 - accuracy: 0.6572 - val_loss: 1.2203 - val_accuracy: 0.6265
Epoch 19/20
573/573 [=====] - 4s 6ms/step - loss: 1.0886 - accuracy: 0.6623 - val_loss: 1.1667 - val_accuracy: 0.6503

Epoch 20/20

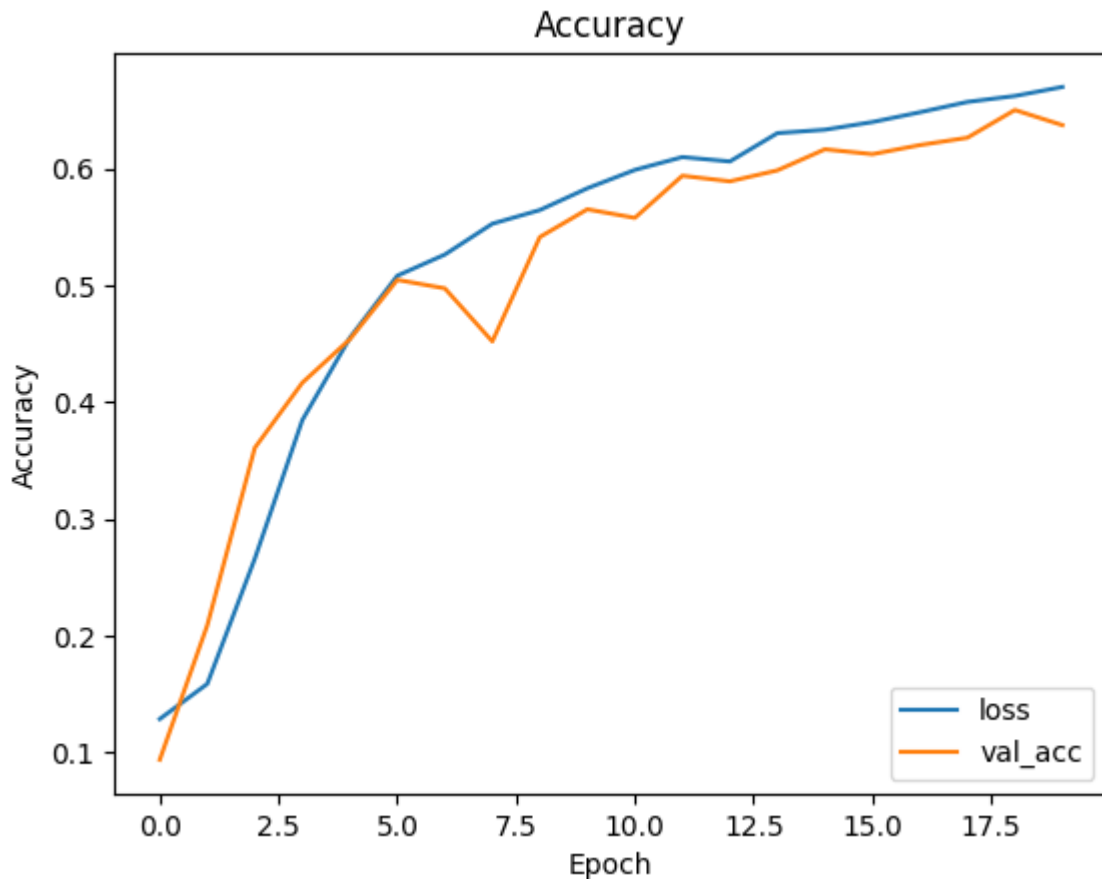
573/573 [=====] - 3s 6ms/step - loss: 1.0636 - accuracy: 0.6701 - val_loss: 1.2180 - val_accuracy: 0.6373

Now that my model is trained, let's compare it to the validation set. It is important to check for overfitting, as the model will hypothetically be used in real-world scenarios.

```
In [ ]: plt.plot(history_1.history['loss'])
plt.plot(history_1.history['val_loss'])
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



```
In [ ]: plt.plot(history_1.history['accuracy'])
plt.plot(history_1.history['val_accuracy'])
plt.legend(['loss', 'val_acc'], loc='lower right')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```

Overall, not too bad, but we can do better!

3. CNN neural network classifier

I previously used a multi-layer perceptron to make predictions, but realistically, convolutional neural networks (CNNs) should be used in scenarios such as this one. We will do the same thing as before, but use CNNs instead. Note the usage of Conv2D, Maxpool2D, BatchNormalization, and Dropout in this model. The latter two are used as a method to prevent overfitting, while the former two are key parts of CNNs.

```
In [ ]: best_val_acc_path_2='CNN/best'
best_val_acc=ModelCheckpoint(filepath = best_val_acc_path_2, save_best_only=True, monitor='val_acc',
earlystopping = EarlyStopping(monitor='loss', patience=5, verbose=1))

model_2 = Sequential([
    Conv2D(filters= 16, kernel_size= 3, activation='relu', input_shape=X_train[0].shape[1:]),
    MaxPool2D(pool_size= (3,3)),
    Conv2D(filters= 32, kernel_size= 3, padding='valid', activation='relu'),
    MaxPool2D(pool_size= (2,2), strides= 2),
    BatchNormalization(),
    Conv2D(filters= 32, kernel_size= 3, padding='valid', strides=2, activation='relu'),
    Dropout(0.5),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
```

```
])
```

```
model_2.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 30, 30, 16)	448
max_pooling2d (MaxPooling2D)	(None, 10, 10, 16)	0
conv2d_1 (Conv2D)	(None, 8, 8, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 4, 4, 32)	0
batch_normalization (Batch Normalization)	(None, 4, 4, 32)	128
conv2d_2 (Conv2D)	(None, 1, 1, 32)	9248
dropout (Dropout)	(None, 1, 1, 32)	0
flatten_1 (Flatten)	(None, 32)	0
dense_4 (Dense)	(None, 128)	4224
dropout_1 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1290
=====		
Total params: 19,978		
Trainable params: 19,914		
Non-trainable params: 64		

```
In [ ]: model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy']  
history_2 = model_2.fit(X_train, y_train_encoded, callbacks=[best_val_acc, earlystoppi
```

Epoch 1/20
573/573 [=====] - 8s 8ms/step - loss: 1.6032 - accuracy: 0.4
375 - val_loss: 0.9126 - val_accuracy: 0.7243

Epoch 2/20
573/573 [=====] - 4s 7ms/step - loss: 1.0272 - accuracy: 0.6
582 - val_loss: 0.7938 - val_accuracy: 0.7738

Epoch 3/20
573/573 [=====] - 4s 7ms/step - loss: 0.8944 - accuracy: 0.7
072 - val_loss: 0.8072 - val_accuracy: 0.7567

Epoch 4/20
573/573 [=====] - 4s 7ms/step - loss: 0.8338 - accuracy: 0.7
282 - val_loss: 0.6628 - val_accuracy: 0.8022

Epoch 5/20
573/573 [=====] - 4s 7ms/step - loss: 0.7929 - accuracy: 0.7
431 - val_loss: 0.6592 - val_accuracy: 0.8005

Epoch 6/20
573/573 [=====] - 4s 7ms/step - loss: 0.7650 - accuracy: 0.7
542 - val_loss: 0.6912 - val_accuracy: 0.8033

Epoch 7/20
573/573 [=====] - 4s 8ms/step - loss: 0.7392 - accuracy: 0.7
643 - val_loss: 0.7413 - val_accuracy: 0.7929

Epoch 8/20
573/573 [=====] - 4s 8ms/step - loss: 0.7284 - accuracy: 0.7
663 - val_loss: 0.6281 - val_accuracy: 0.8193

Epoch 9/20
573/573 [=====] - 4s 7ms/step - loss: 0.7181 - accuracy: 0.7
726 - val_loss: 0.6263 - val_accuracy: 0.8150

Epoch 10/20
573/573 [=====] - 4s 8ms/step - loss: 0.7006 - accuracy: 0.7
753 - val_loss: 0.6217 - val_accuracy: 0.8234

Epoch 11/20
573/573 [=====] - 4s 7ms/step - loss: 0.6966 - accuracy: 0.7
792 - val_loss: 0.6415 - val_accuracy: 0.8198

Epoch 12/20
573/573 [=====] - 4s 8ms/step - loss: 0.6844 - accuracy: 0.7
812 - val_loss: 0.6645 - val_accuracy: 0.8154

Epoch 13/20
573/573 [=====] - 4s 7ms/step - loss: 0.6808 - accuracy: 0.7
831 - val_loss: 0.6634 - val_accuracy: 0.8092

Epoch 14/20
573/573 [=====] - 4s 7ms/step - loss: 0.6686 - accuracy: 0.7
903 - val_loss: 0.7058 - val_accuracy: 0.8047

Epoch 15/20
573/573 [=====] - 4s 7ms/step - loss: 0.6701 - accuracy: 0.7
888 - val_loss: 0.6487 - val_accuracy: 0.8281

Epoch 16/20
573/573 [=====] - 4s 7ms/step - loss: 0.6646 - accuracy: 0.7
901 - val_loss: 0.6371 - val_accuracy: 0.8330

Epoch 17/20
573/573 [=====] - 4s 7ms/step - loss: 0.6585 - accuracy: 0.7
907 - val_loss: 0.6668 - val_accuracy: 0.8158

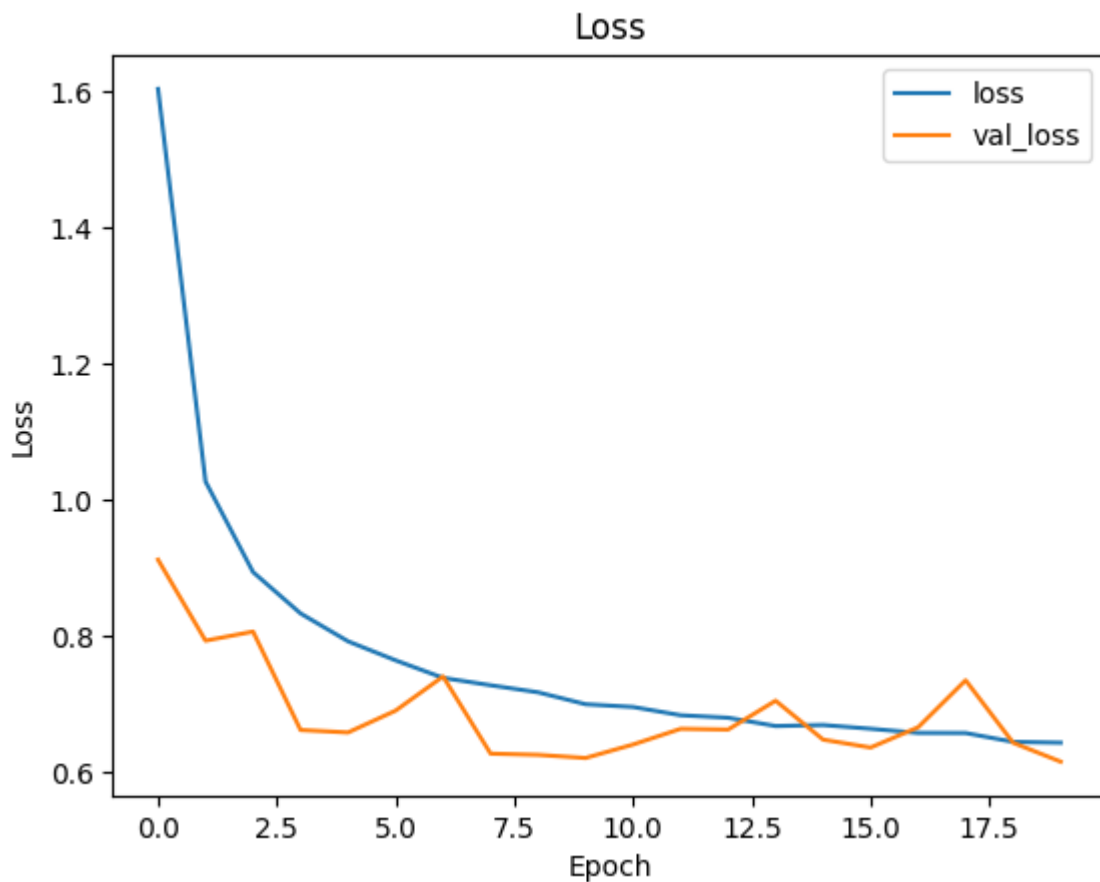
Epoch 18/20
573/573 [=====] - 4s 7ms/step - loss: 0.6583 - accuracy: 0.7
941 - val_loss: 0.7357 - val_accuracy: 0.7929

Epoch 19/20
573/573 [=====] - 4s 6ms/step - loss: 0.6454 - accuracy: 0.7
976 - val_loss: 0.6441 - val_accuracy: 0.8316

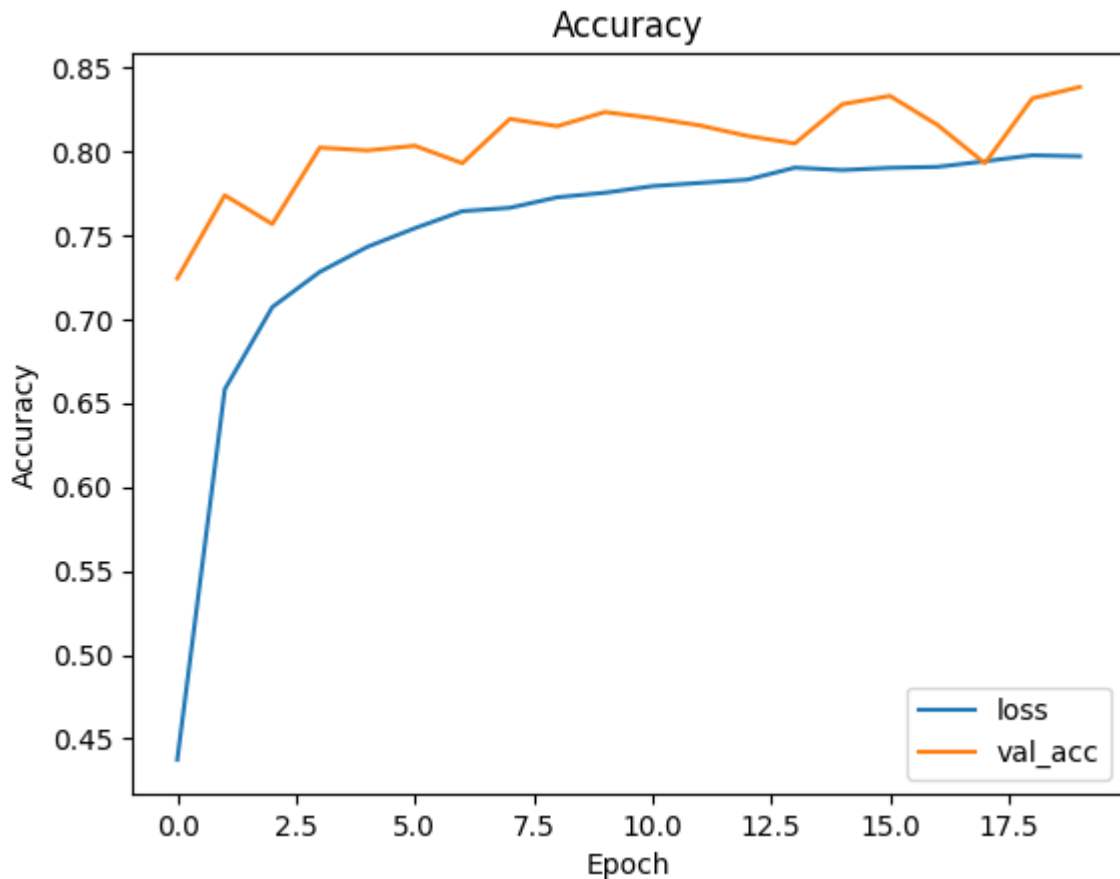
Epoch 20/20

573/573 [=====] - 4s 6ms/step - loss: 0.6442 - accuracy: 0.7970 - val_loss: 0.6163 - val_accuracy: 0.8384

```
In [ ]: plt.plot(history_2.history['loss'])
plt.plot(history_2.history['val_loss'])
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



```
In [ ]: plt.plot(history_2.history['accuracy'])
plt.plot(history_2.history['val_accuracy'])
plt.legend(['loss', 'val_acc'], loc='lower right')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```



Note that the CNN achieved higher accuracy while having significantly less parameters than the MLP.

4. Get model predictions

Let's see what the network is doing! In this section, I load the best weights in both models and pit them against 5 randomly chosen image samples. I then plot the model's predictions to see how confident it is in its response.

```
In [ ]: model_1.load_weights(best_val_acc_path)
```

```
Out[ ]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x2a27e5a64d0>
```

```
In [ ]: test_image = X_test.shape[0]

random1 = np.random.choice(test_image, 5)
random_test_image = X_test[random1, ...]
random_test_target = y_test[random1, ...]
prediction = model_1.predict(random_test_image)

graph, axes = plt.subplots(5, 2, figsize=(20, 15))
graph.subplots_adjust(hspace=0.4, wspace=-0.2)

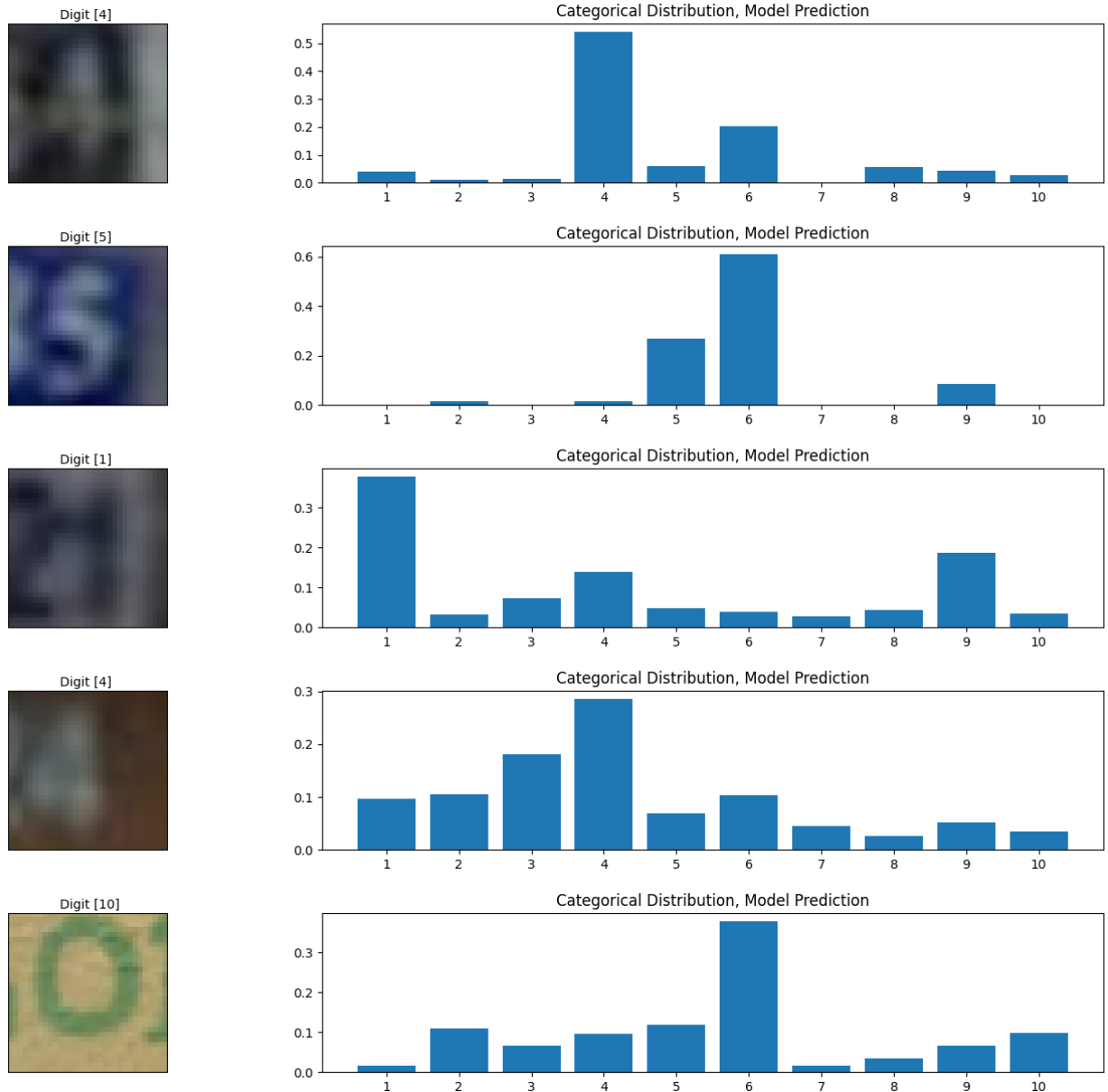
for n, (predict, image, label) in enumerate(zip(prediction, random_test_image, random_
    axes[n,0].imshow(np.squeeze(image))
    axes[n,0].get_xaxis().set_visible(False)
```

```

axes[n,0].get_yaxis().set_visible(False)
axes[n,0].text(10., -1.5, f'Digit {label}')
axes[n,1].bar(np.arange(1,11), predict)
axes[n,1].set_xticks(np.arange(1,11))
axes[n,1].set_title("Categorical Distribution, Model Prediction")
plt.show()

```

1/1 [=====] - 0s 93ms/step



```
In [ ]: model_2.load_weights(best_val_acc_path_2)
```

```
Out[ ]: <tensorflow.python.training.training.util.CheckpointLoadStatus at 0x2a11a74c0d0>
```

```

In [ ]: test_image = X_test.shape[0]

random1 = np.random.choice(test_image, 5)
random_test_image = X_test[random1, ...]
random_test_target = y_test[random1, ...]
prediction = model_2.predict(random_test_image)

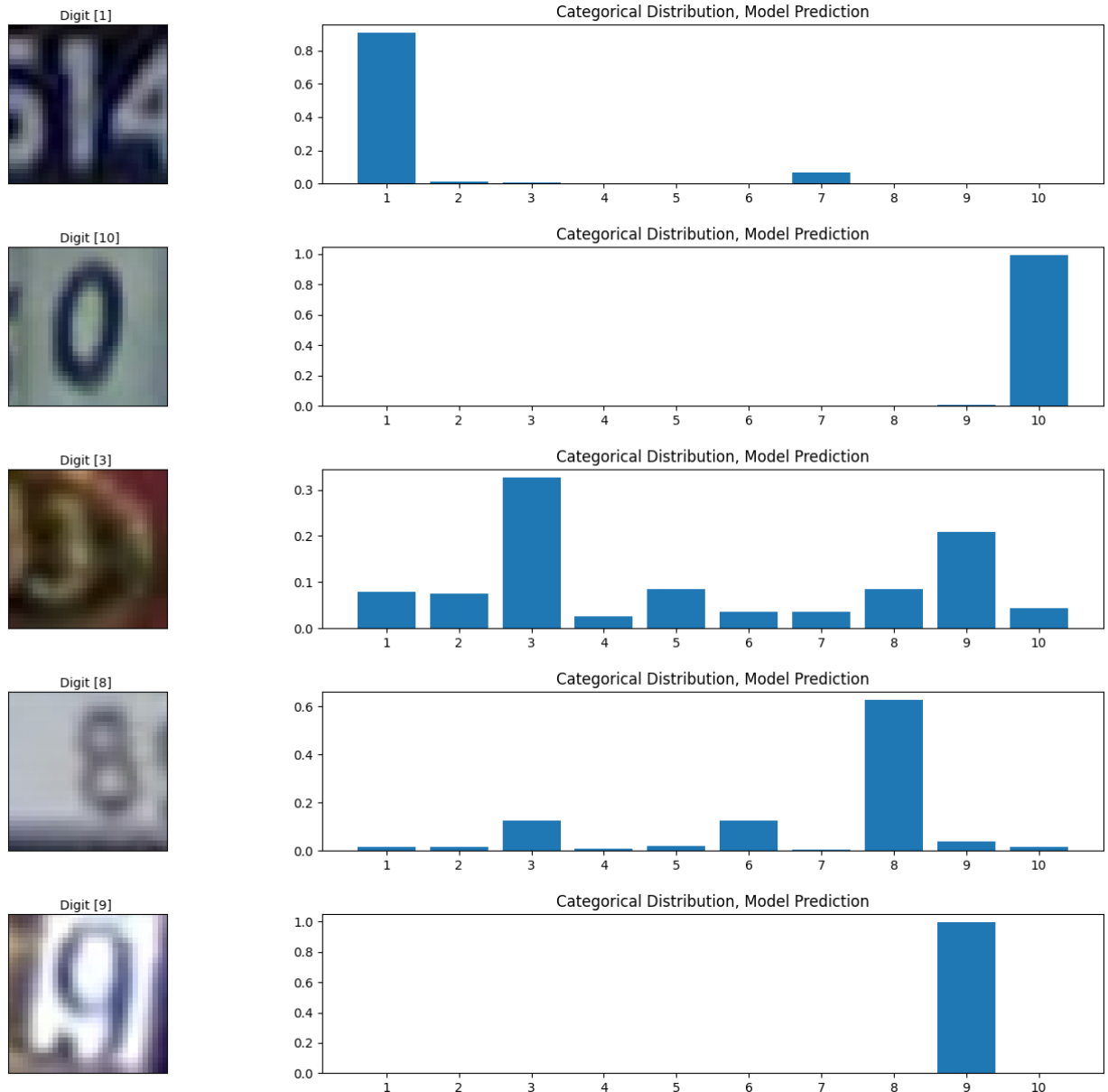
graph, axes = plt.subplots(5, 2, figsize=(20, 15))

```

```
graph.subplots_adjust(hspace=0.4, wspace=-0.2)

for n, (predict, image, label) in enumerate(zip(prediction, random_test_image, random_
axes[n,0].imshow(np.squeeze(image))
axes[n,0].get_xaxis().set_visible(False)
axes[n,0].get_yaxis().set_visible(False)
axes[n,0].text(10., -1.5, f'Digit {label}'))
axes[n,1].bar(np.arange(1,11), predict)
axes[n,1].set_xticks(np.arange(1,11))
axes[n,1].set_title("Categorical Distribution, Model Prediction")
plt.show()
```

1/1 [=====] - 0s 155ms/step



Note the higher confidence of the CNN as apposed to the MLP!

Through this notebook, I have compared MLPs and CNNs against each other under the SVHN dataset. It can be concluded that CNNs are much stronger than MLPs when it comes to computer vision, which is why they are the most popular method for static image analysis.