Capstone Project

May 5, 2021

1 Capstone Project

1.1 Image classifier for the SVHN dataset

1.1.1 Instructions

In this notebook, you will create a neural network that classifies real-world images digits. You will use concepts from throughout this course in building, training, testing, validating and saving your Tensorflow classifier model.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

1.1.2 How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (File -> Download as -> PDF via LaTeX). You should then submit this pdf for review.

1.1.3 Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

```
In [221]: import tensorflow as tf
          from scipy.io import loadmat
```

For the capstone project, you will use the SVHN dataset. This is an image dataset of 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.

Your 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 [222]:
```
# Run this cell to load the dataset

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.

1.2 1. Inspect and preprocess the dataset

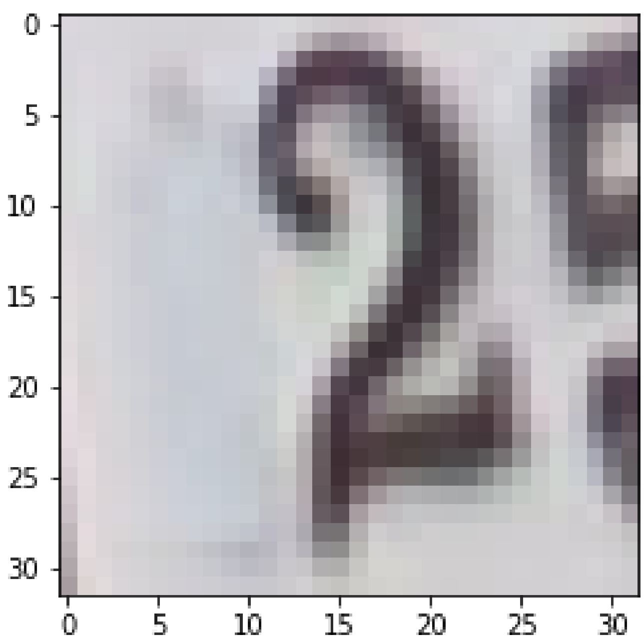
- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking the average across all colour channels for each pixel. *Hint: retain the channel dimension, which will now have size 1.*
- Select a random sample of the grayscale images and corresponding labels from the dataset (at least 10), and display them in a figure.

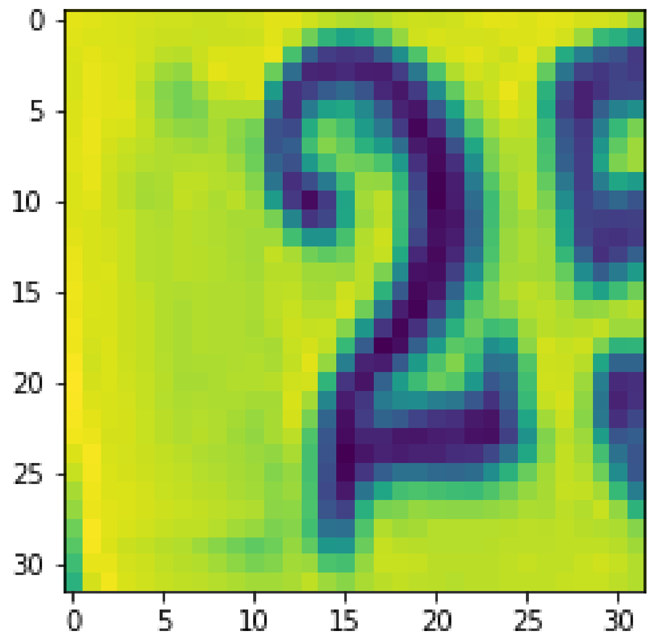
In [413]:
```
import numpy as np

X_train_original = train['X']
```

```
X_test_original = test['X']
y_train = train['y']
y_test = test['y']

# reshape array so the number of samples is the first dimension
###### X_test = X.reshape((X_test_original.shape[3],32,32,3)) # couldn't get this to
X_train_3_channels = np.moveaxis(X_train_original,-1,0)
X_test_3_channels = np.moveaxis(X_test_original,-1,0)
X_train = np.average(X_train_3_channels, axis=3).reshape(73257, 32, 32,1)/255
X_test = np.average(X_test_3_channels, axis=3).reshape(26032, 32, 32,1)/255

y_train[y_train==10] = 0
y_test[y_test==10] = 0
```

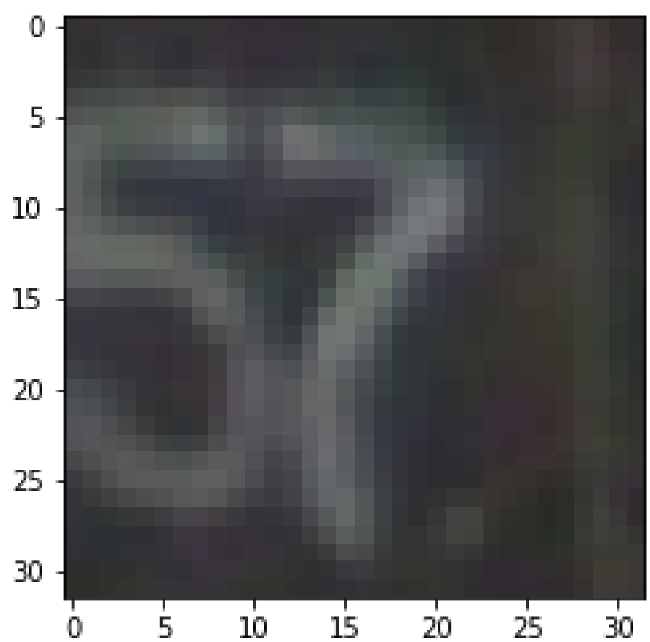
In [420]:
```
import matplotlib.pyplot as plt
import random
%matplotlib inline

for i in range(10):
    n = random.randrange(0,X_train_3_channels.shape[0])
    print(f'Image#(below): {n}\nlabel:{y_train[n]}')
    plt.imshow(X_train_3_channels[n])
    plt.show()
    plt.imshow(X_train[n,:,:,0])
    plt.show()
    print('=========================================')
```

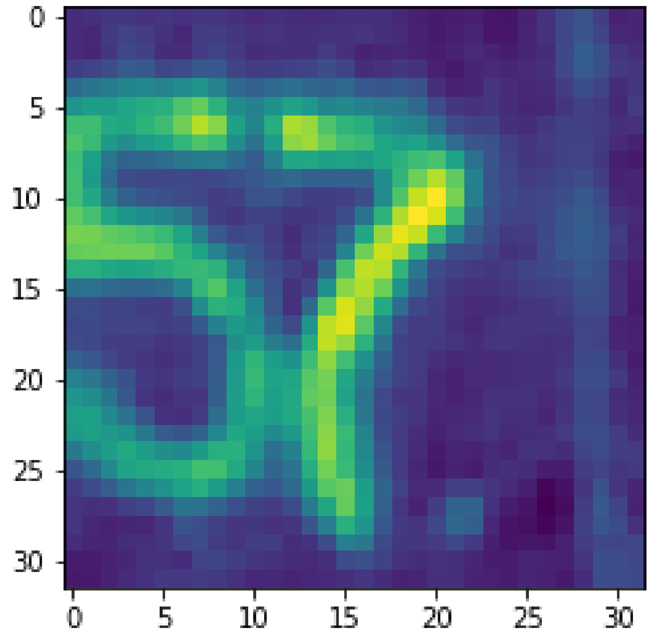
```
Image#(below): 64132
label:[2]
```



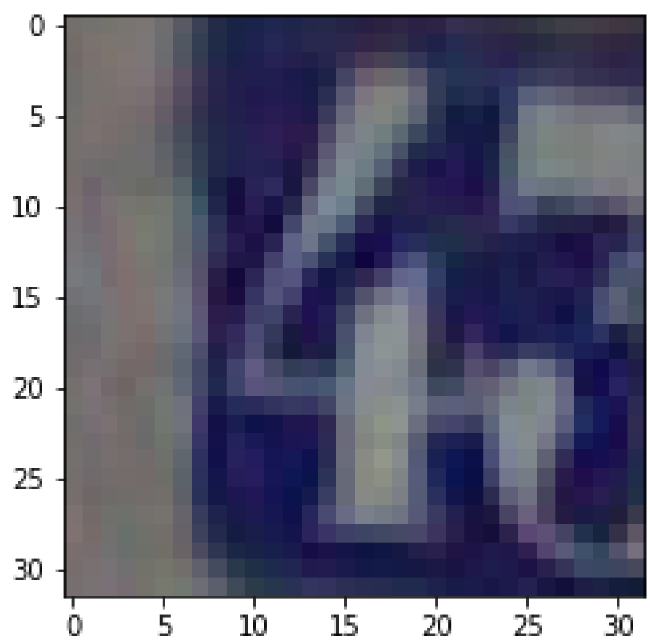


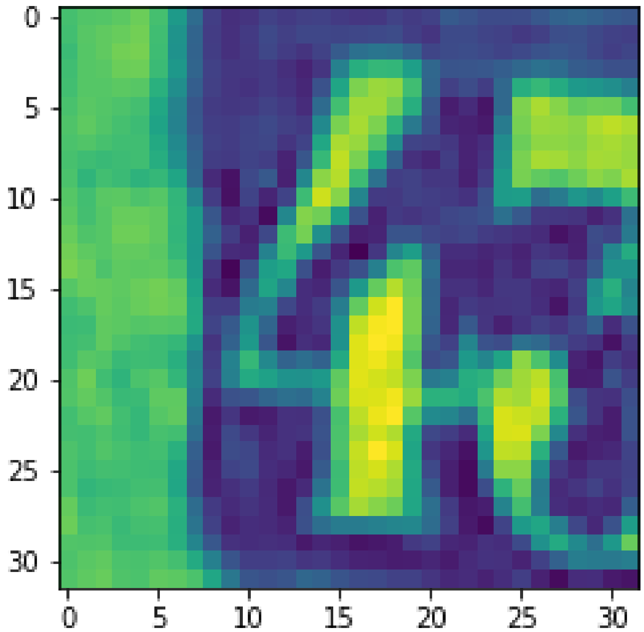
```
==========================================
Image#(below): 68885
label:[7]
```



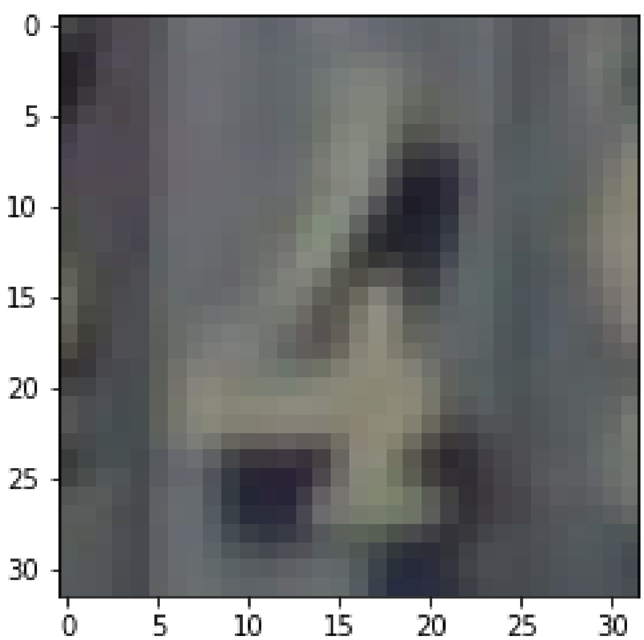


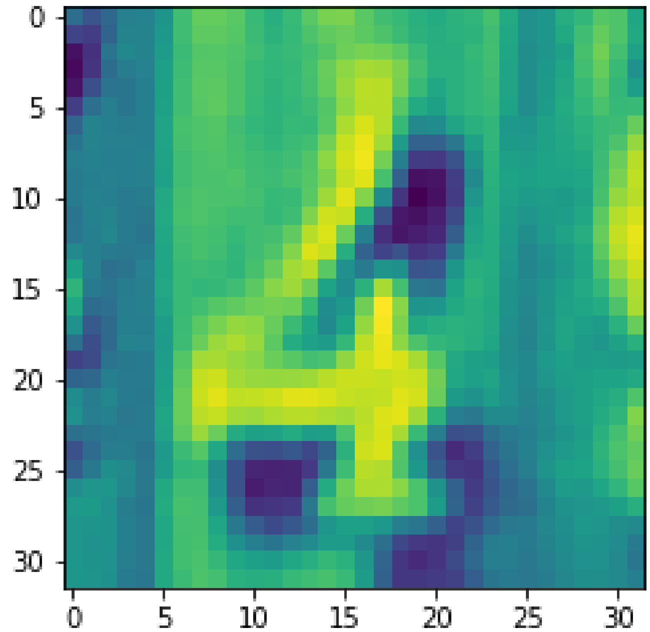

```
==========================================
Image#(below): 13428
label:[4]
```



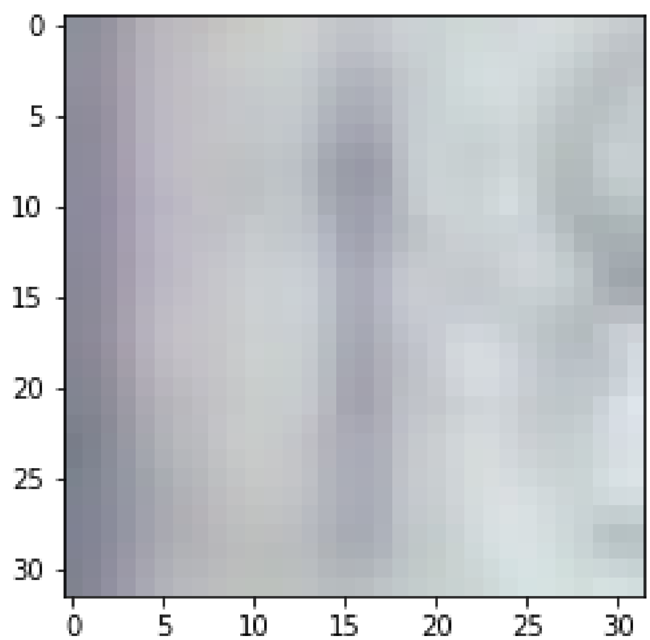


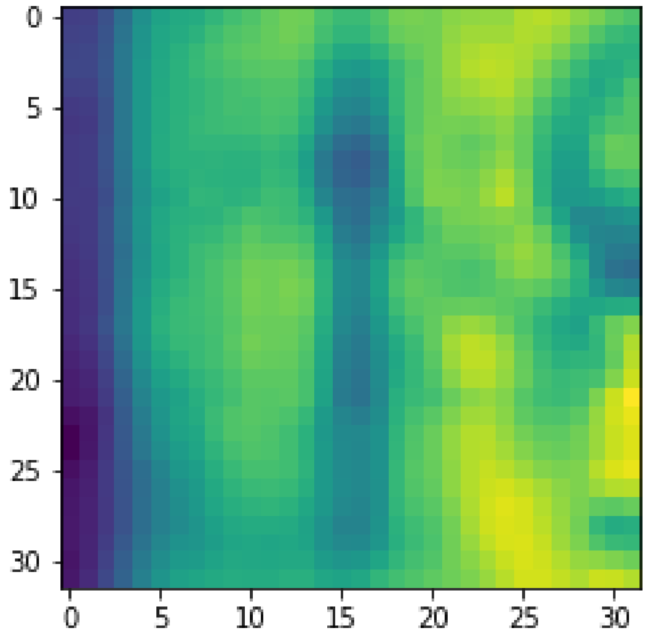

```
==========================================
Image#(below): 70388
label:[4]
```



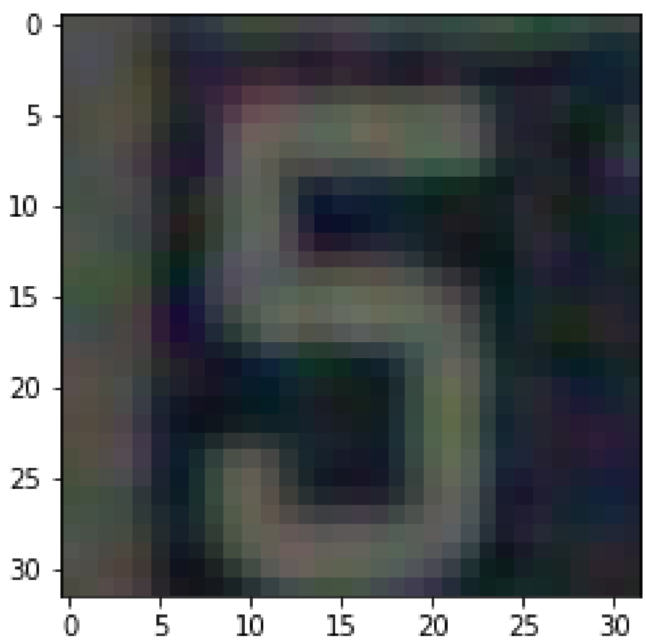


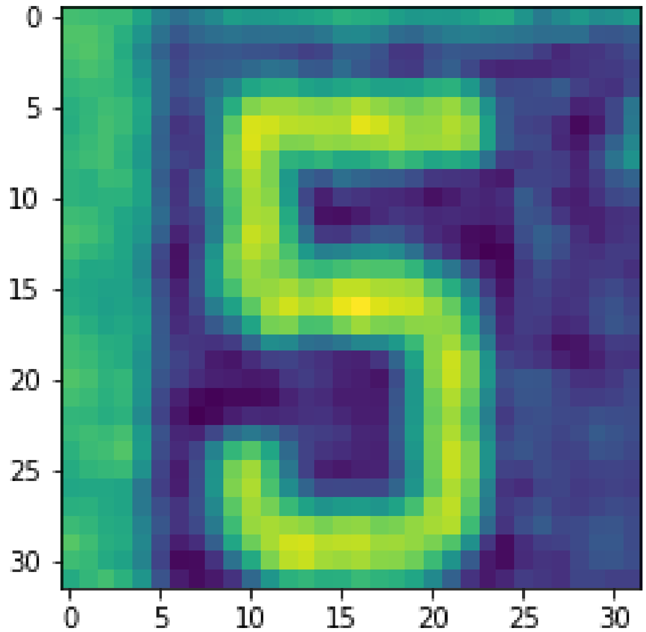

```
==========================================
Image#(below): 38695
label:[1]
```



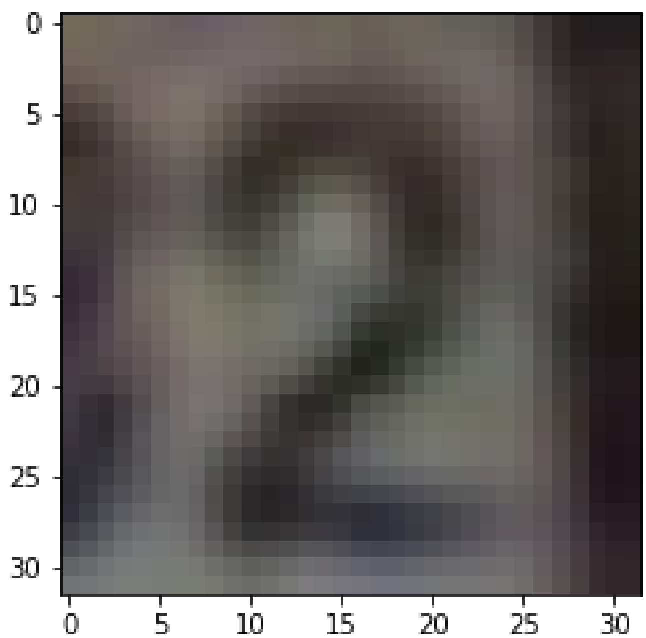


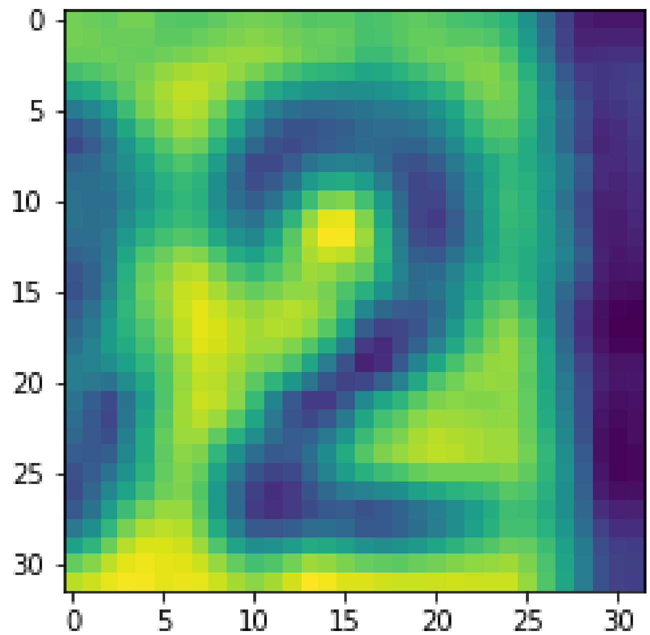

```
==========================================
Image#(below): 51877
label:[5]
```



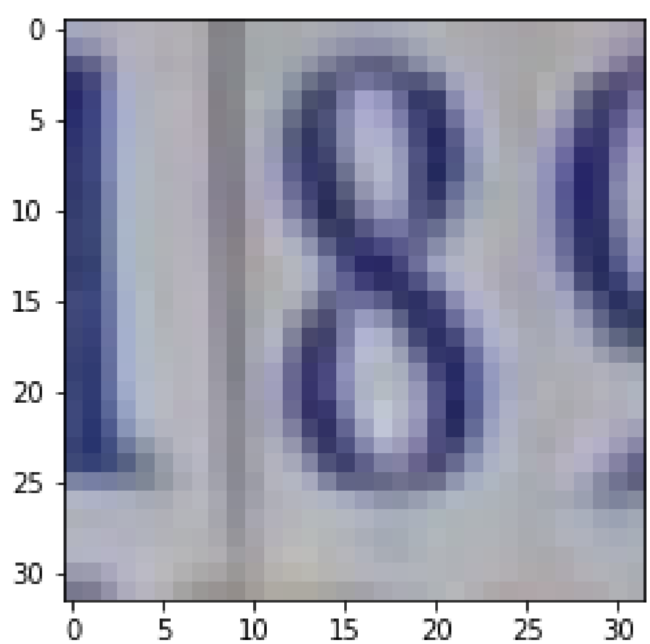


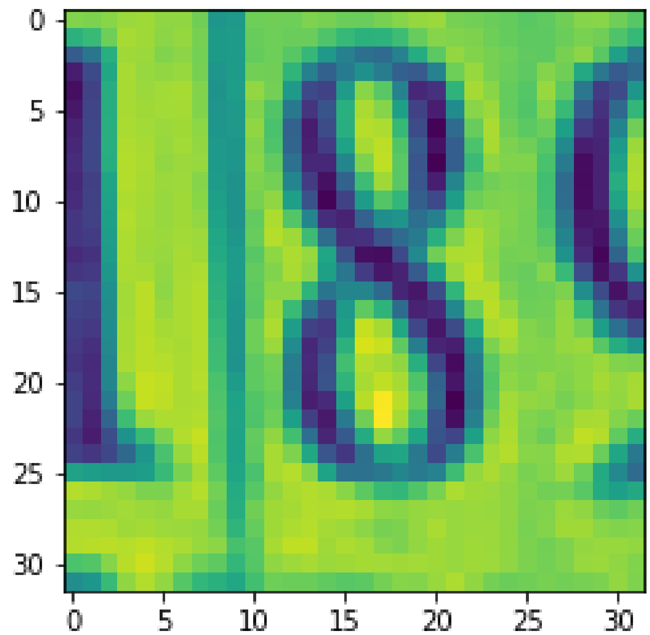
```
=========================================
Image#(below): 26931
label:[2]
```



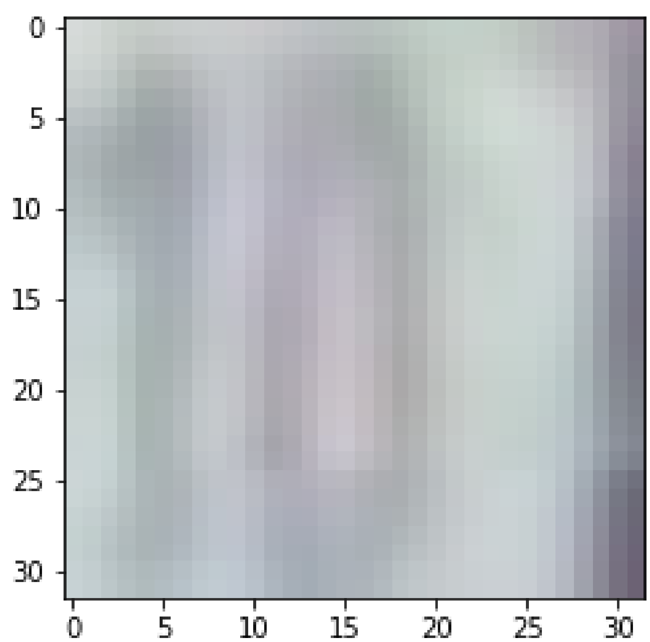


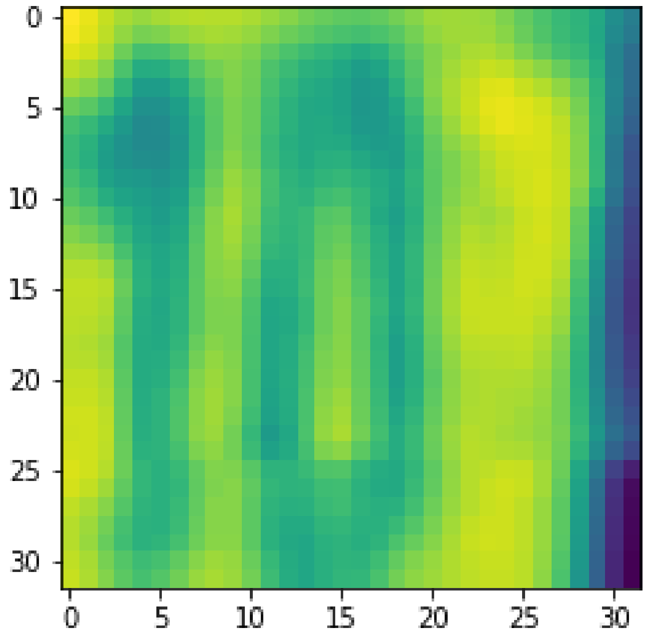
```
==========================================
Image#(below): 69078
label:[8]
```



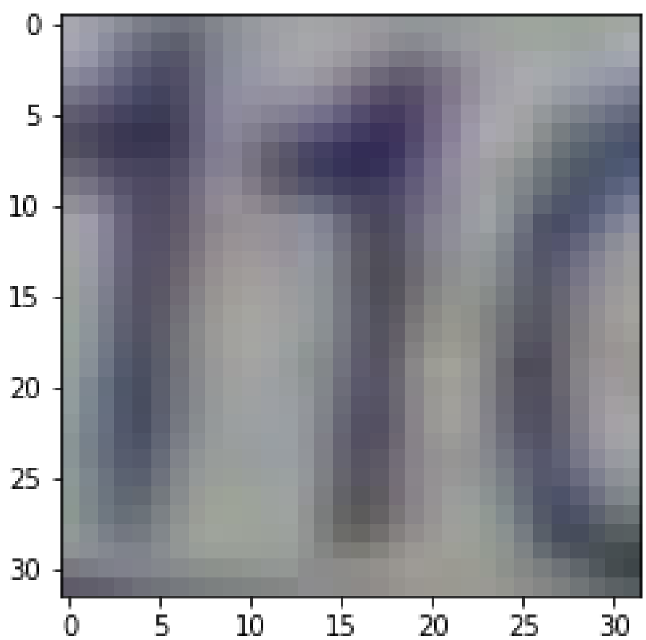


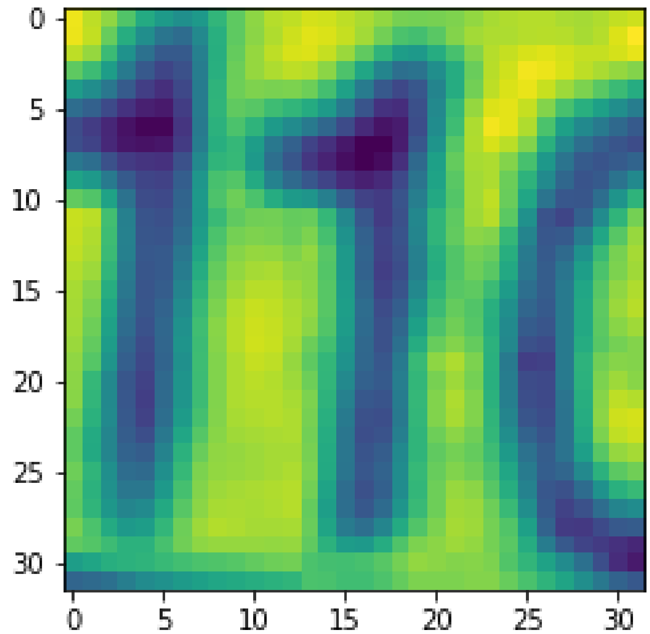
```
==========================================
Image#(below): 41264
label:[0]
```






```
==========================================
Image#(below): 11011
label:[1]
```





==

Another option for showing image stored in an array

```
from tensorflow.keras.preprocessing.image import load_img
from PIL import Image

img1 = Image.fromarray(X_train[345])
img1
```

1.3 2. MLP neural network classifier

- Build an MLP classifier model using the Sequential API. Your model should use only Flatten and Dense layers, with the final layer having a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different MLP architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 4 or 5 layers.*
- Print out the model summary (using the summary() method)
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- As a guide, you should aim to achieve a final categorical cross entropy training loss of less than 1.0 (the validation loss might be higher).

- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

In [422]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

In [429]:
```
model = Sequential([
    Flatten(input_shape=(32,32,1)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

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

In [430]:
```
model.summary()
```

```
Model: "sequential_62"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_58 (Flatten)         (None, 1024)              0
_________________________________________________________________
dense_157 (Dense)            (None, 128)               131200
_________________________________________________________________
dense_158 (Dense)            (None, 64)                8256
_________________________________________________________________
dense_159 (Dense)            (None, 32)                2080
_________________________________________________________________
dropout_44 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_160 (Dense)            (None, 10)                330
=================================================================
Total params: 141,866
Trainable params: 141,866
Non-trainable params: 0
_________________________________________________________________
```

In [431]:
```
#callbacks
early_stopping = EarlyStopping(monitor='loss', patience=3)
```

```
best_checkpoint = ModelCheckpoint('checkpoints_best_only/checkpoint', save_weights_on
                                  monitor='val_accuracy', mode='max')
```

```
In [432]: history = model.fit(
              X_train,
              y_train,
              epochs=30,
              validation_data=(X_test,y_test),
              callbacks=[early_stopping,best_checkpoint],
              verbose=2
          )
```

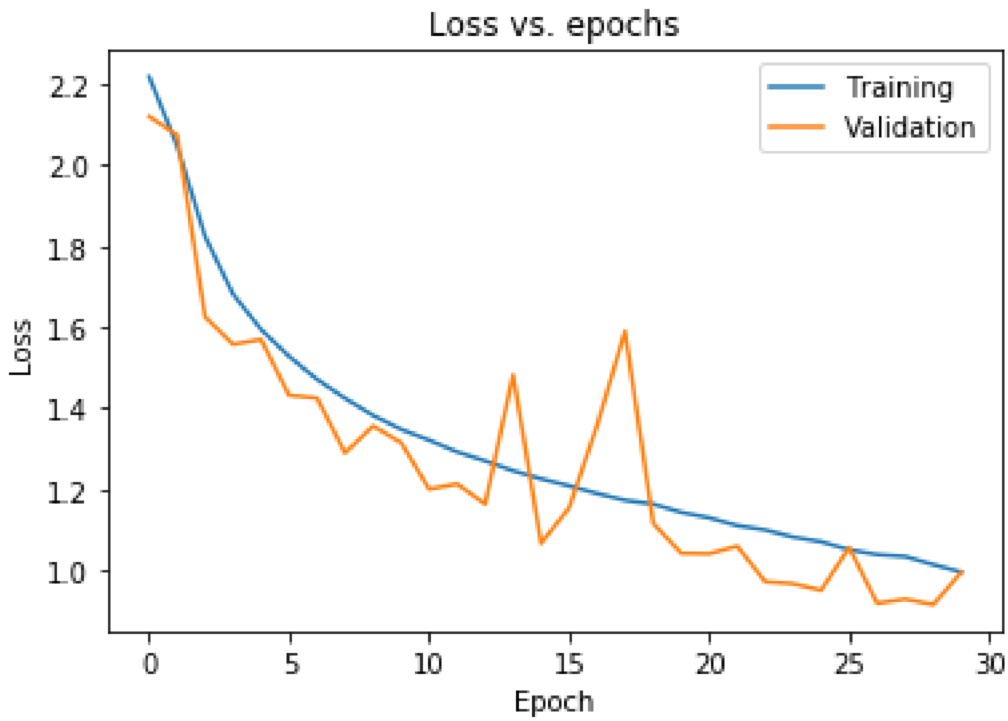
```
Train on 73257 samples, validate on 26032 samples
Epoch 1/30
73257/73257 - 26s - loss: 2.2181 - accuracy: 0.1907 - val_loss: 2.1198 - val_accuracy: 0.2287
Epoch 2/30
73257/73257 - 23s - loss: 2.0476 - accuracy: 0.2731 - val_loss: 2.0741 - val_accuracy: 0.3038
Epoch 3/30
73257/73257 - 23s - loss: 1.8237 - accuracy: 0.3588 - val_loss: 1.6255 - val_accuracy: 0.4491
Epoch 4/30
73257/73257 - 24s - loss: 1.6817 - accuracy: 0.4118 - val_loss: 1.5575 - val_accuracy: 0.4690
Epoch 5/30
73257/73257 - 25s - loss: 1.5945 - accuracy: 0.4491 - val_loss: 1.5698 - val_accuracy: 0.4780
Epoch 6/30
73257/73257 - 26s - loss: 1.5281 - accuracy: 0.4781 - val_loss: 1.4323 - val_accuracy: 0.5250
Epoch 7/30
73257/73257 - 26s - loss: 1.4717 - accuracy: 0.5000 - val_loss: 1.4256 - val_accuracy: 0.5423
Epoch 8/30
73257/73257 - 26s - loss: 1.4248 - accuracy: 0.5190 - val_loss: 1.2898 - val_accuracy: 0.5884
Epoch 9/30
73257/73257 - 25s - loss: 1.3826 - accuracy: 0.5343 - val_loss: 1.3564 - val_accuracy: 0.5764
Epoch 10/30
73257/73257 - 26s - loss: 1.3481 - accuracy: 0.5449 - val_loss: 1.3150 - val_accuracy: 0.5755
Epoch 11/30
73257/73257 - 26s - loss: 1.3214 - accuracy: 0.5577 - val_loss: 1.2009 - val_accuracy: 0.6198
Epoch 12/30
73257/73257 - 26s - loss: 1.2924 - accuracy: 0.5657 - val_loss: 1.2137 - val_accuracy: 0.6100
Epoch 13/30
73257/73257 - 26s - loss: 1.2702 - accuracy: 0.5728 - val_loss: 1.1641 - val_accuracy: 0.6288
Epoch 14/30
73257/73257 - 26s - loss: 1.2463 - accuracy: 0.5823 - val_loss: 1.4829 - val_accuracy: 0.5017
Epoch 15/30
73257/73257 - 27s - loss: 1.2263 - accuracy: 0.5898 - val_loss: 1.0674 - val_accuracy: 0.6656
Epoch 16/30
73257/73257 - 27s - loss: 1.2092 - accuracy: 0.5962 - val_loss: 1.1546 - val_accuracy: 0.6386
Epoch 17/30
73257/73257 - 25s - loss: 1.1901 - accuracy: 0.6034 - val_loss: 1.3578 - val_accuracy: 0.5470
Epoch 18/30
```

```
73257/73257 - 25s - loss: 1.1733 - accuracy: 0.6097 - val_loss: 1.5904 - val_accuracy: 0.5157
Epoch 19/30
73257/73257 - 25s - loss: 1.1640 - accuracy: 0.6155 - val_loss: 1.1164 - val_accuracy: 0.6498
Epoch 20/30
73257/73257 - 25s - loss: 1.1435 - accuracy: 0.6210 - val_loss: 1.0425 - val_accuracy: 0.6722
Epoch 21/30
73257/73257 - 25s - loss: 1.1301 - accuracy: 0.6258 - val_loss: 1.0414 - val_accuracy: 0.6755
Epoch 22/30
73257/73257 - 24s - loss: 1.1110 - accuracy: 0.6352 - val_loss: 1.0601 - val_accuracy: 0.6614
Epoch 23/30
73257/73257 - 23s - loss: 1.1001 - accuracy: 0.6398 - val_loss: 0.9726 - val_accuracy: 0.6973
Epoch 24/30
73257/73257 - 23s - loss: 1.0825 - accuracy: 0.6467 - val_loss: 0.9677 - val_accuracy: 0.7048
Epoch 25/30
73257/73257 - 24s - loss: 1.0709 - accuracy: 0.6520 - val_loss: 0.9515 - val_accuracy: 0.7037
Epoch 26/30
73257/73257 - 23s - loss: 1.0511 - accuracy: 0.6586 - val_loss: 1.0566 - val_accuracy: 0.6598
Epoch 27/30
73257/73257 - 24s - loss: 1.0392 - accuracy: 0.6636 - val_loss: 0.9194 - val_accuracy: 0.7180
Epoch 28/30
73257/73257 - 23s - loss: 1.0344 - accuracy: 0.6669 - val_loss: 0.9298 - val_accuracy: 0.7062
Epoch 29/30
73257/73257 - 24s - loss: 1.0145 - accuracy: 0.6735 - val_loss: 0.9160 - val_accuracy: 0.7162
Epoch 30/30
73257/73257 - 23s - loss: 0.9962 - accuracy: 0.6788 - val_loss: 0.9959 - val_accuracy: 0.6891
```

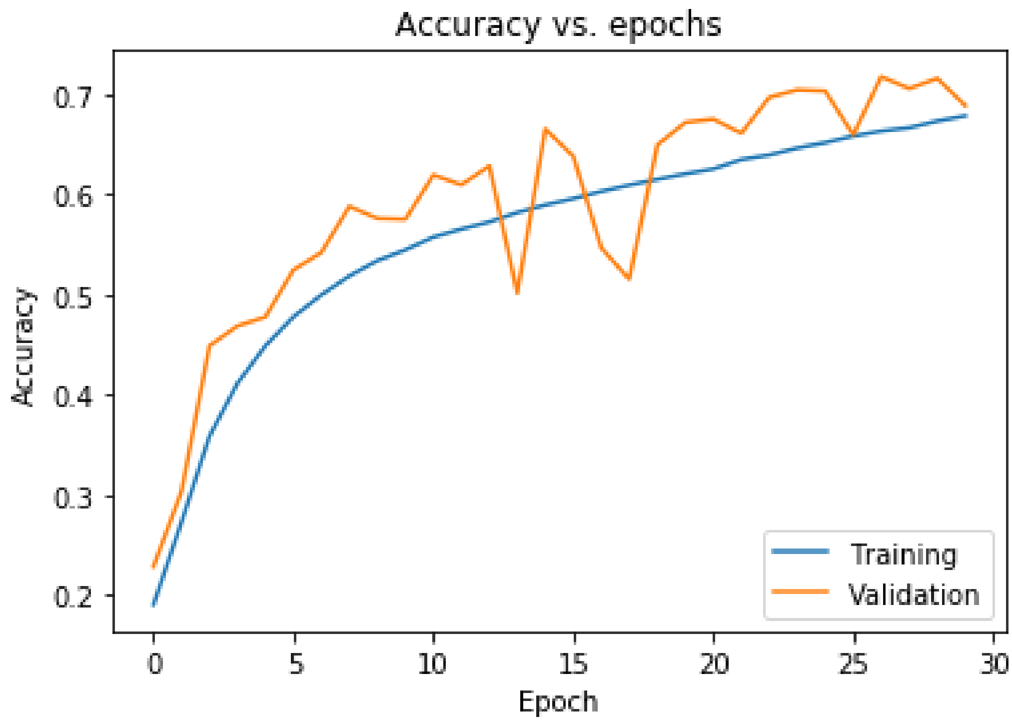
In [433]:
```
! ls checkpoints_best_only/
```

```
checkpoint  checkpoint.data-00000-of-00001  checkpoint.index
```

In [434]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```




```
In [435]: plt.plot(history.history['accuracy'])
          plt.plot(history.history['val_accuracy'])
          plt.title('Accuracy vs. epochs')
          plt.ylabel('Accuracy')
          plt.xlabel('Epoch')
          plt.legend(['Training', 'Validation'], loc='lower right')
          plt.show()
```



1.4 3. CNN neural network classifier

- Build a CNN classifier model using the Sequential API. Your model should use the Conv2D, MaxPool2D, BatchNormalization, Flatten, Dense and Dropout layers. The final layer should again have a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different CNN architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 2 or 3 convolutional layers and 2 fully connected layers.)*
- The CNN model should use fewer trainable parameters than your MLP model.
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- You should aim to beat the MLP model performance with fewer parameters!
- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

```
In [436]: from tensorflow.keras.layers import Conv2D, Dropout, BatchNormalization, MaxPooling2I
```

```
In [438]: model_cnn = Sequential([
              Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(32,32,1)),
              MaxPooling2D(pool_size=(3,3)),
              BatchNormalization(),
```

```
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

model_cnn.compile(
    optimizer='sgd',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

In [439]:
```
model_cnn.summary()
```

```
Model: "sequential_64"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_53 (Conv2D)           (None, 30, 30, 16)        160
_________________________________________________________________
max_pooling2d_44 (MaxPooling (None, 10, 10, 16)        0
_________________________________________________________________
batch_normalization_17 (Batc (None, 10, 10, 16)        64
_________________________________________________________________
flatten_60 (Flatten)         (None, 1600)              0
_________________________________________________________________
dense_163 (Dense)            (None, 64)                102464
_________________________________________________________________
dropout_46 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_164 (Dense)            (None, 10)                650
=================================================================
Total params: 103,338
Trainable params: 103,306
Non-trainable params: 32
_________________________________________________________________
```

In [440]:
```
#callbacks
early_stopping_cnn = EarlyStopping(monitor='loss', patience=3)
best_checkpoint_cnn = ModelCheckpoint('checkpoints_best_only_cnn/checkpoint', save_we
                           monitor='val_accuracy', mode='max')
```

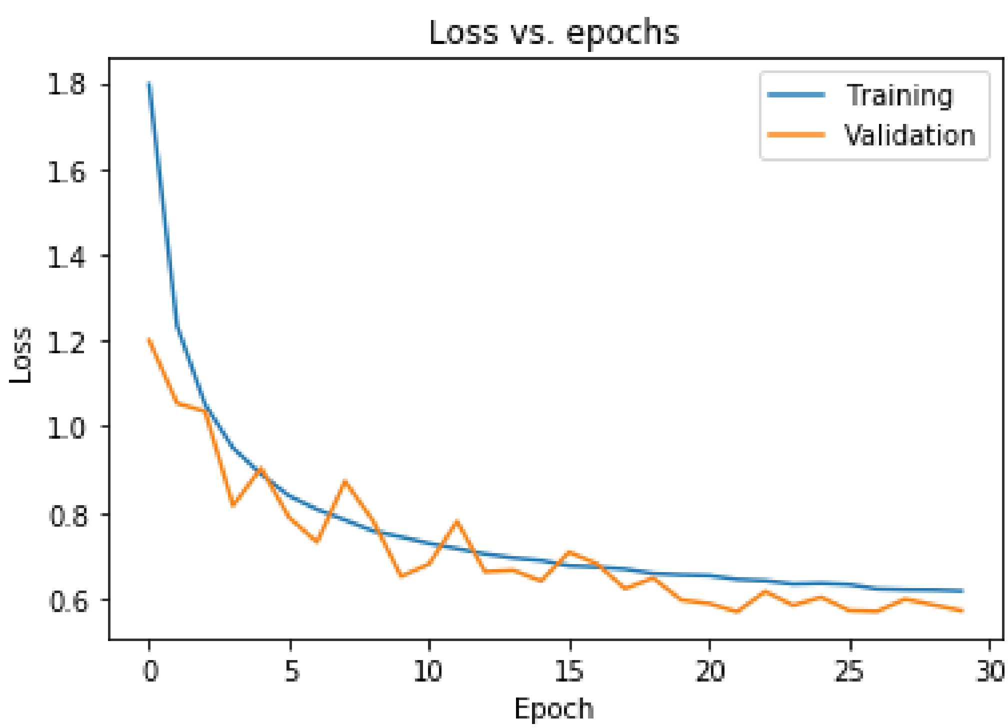
In [441]:
```
history_cnn = model_cnn.fit(
    X_train,
    y_train,
    epochs=30,
    validation_data=(X_test,y_test),
    callbacks=[early_stopping_cnn,best_checkpoint_cnn],
```

```
              verbose=2
          )
```

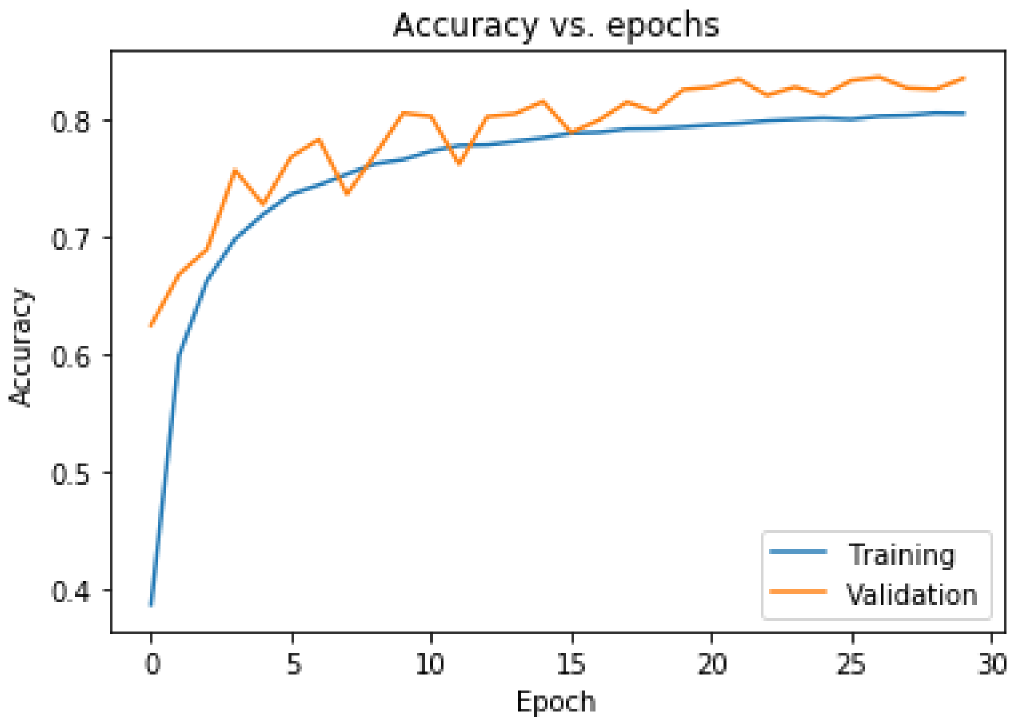
```
Train on 73257 samples, validate on 26032 samples
Epoch 1/30
73257/73257 - 142s - loss: 1.7958 - accuracy: 0.3871 - val_loss: 1.2007 - val_accuracy: 0.6248
Epoch 2/30
73257/73257 - 140s - loss: 1.2343 - accuracy: 0.5988 - val_loss: 1.0540 - val_accuracy: 0.6680
Epoch 3/30
73257/73257 - 141s - loss: 1.0511 - accuracy: 0.6627 - val_loss: 1.0350 - val_accuracy: 0.6890
Epoch 4/30
73257/73257 - 145s - loss: 0.9497 - accuracy: 0.6978 - val_loss: 0.8161 - val_accuracy: 0.7564
Epoch 5/30
73257/73257 - 147s - loss: 0.8894 - accuracy: 0.7187 - val_loss: 0.9036 - val_accuracy: 0.7273
Epoch 6/30
73257/73257 - 151s - loss: 0.8388 - accuracy: 0.7358 - val_loss: 0.7885 - val_accuracy: 0.7676
Epoch 7/30
73257/73257 - 143s - loss: 0.8072 - accuracy: 0.7438 - val_loss: 0.7309 - val_accuracy: 0.7827
Epoch 8/30
73257/73257 - 146s - loss: 0.7824 - accuracy: 0.7530 - val_loss: 0.8727 - val_accuracy: 0.7361
Epoch 9/30
73257/73257 - 143s - loss: 0.7571 - accuracy: 0.7618 - val_loss: 0.7808 - val_accuracy: 0.7696
Epoch 10/30
73257/73257 - 145s - loss: 0.7429 - accuracy: 0.7655 - val_loss: 0.6511 - val_accuracy: 0.8049
Epoch 11/30
73257/73257 - 137s - loss: 0.7279 - accuracy: 0.7726 - val_loss: 0.6802 - val_accuracy: 0.8021
Epoch 12/30
73257/73257 - 135s - loss: 0.7157 - accuracy: 0.7773 - val_loss: 0.7794 - val_accuracy: 0.7614
Epoch 13/30
73257/73257 - 136s - loss: 0.7033 - accuracy: 0.7779 - val_loss: 0.6621 - val_accuracy: 0.8019
Epoch 14/30
73257/73257 - 136s - loss: 0.6942 - accuracy: 0.7808 - val_loss: 0.6655 - val_accuracy: 0.8043
Epoch 15/30
73257/73257 - 136s - loss: 0.6882 - accuracy: 0.7839 - val_loss: 0.6405 - val_accuracy: 0.8148
Epoch 16/30
73257/73257 - 135s - loss: 0.6759 - accuracy: 0.7878 - val_loss: 0.7077 - val_accuracy: 0.7885
Epoch 17/30
73257/73257 - 136s - loss: 0.6729 - accuracy: 0.7886 - val_loss: 0.6803 - val_accuracy: 0.7991
Epoch 18/30
73257/73257 - 144s - loss: 0.6682 - accuracy: 0.7915 - val_loss: 0.6225 - val_accuracy: 0.8143
Epoch 19/30
73257/73257 - 144s - loss: 0.6578 - accuracy: 0.7918 - val_loss: 0.6473 - val_accuracy: 0.8060
Epoch 20/30
73257/73257 - 136s - loss: 0.6548 - accuracy: 0.7932 - val_loss: 0.5963 - val_accuracy: 0.8248
Epoch 21/30
73257/73257 - 135s - loss: 0.6530 - accuracy: 0.7950 - val_loss: 0.5880 - val_accuracy: 0.8271
Epoch 22/30
73257/73257 - 135s - loss: 0.6439 - accuracy: 0.7962 - val_loss: 0.5689 - val_accuracy: 0.8336
```

```
Epoch 23/30
73257/73257 - 134s - loss: 0.6410 - accuracy: 0.7984 - val_loss: 0.6160 - val_accuracy: 0.8200
Epoch 24/30
73257/73257 - 157s - loss: 0.6334 - accuracy: 0.7997 - val_loss: 0.5837 - val_accuracy: 0.8272
Epoch 25/30
73257/73257 - 158s - loss: 0.6351 - accuracy: 0.8008 - val_loss: 0.6027 - val_accuracy: 0.8201
Epoch 26/30
73257/73257 - 156s - loss: 0.6322 - accuracy: 0.7997 - val_loss: 0.5718 - val_accuracy: 0.8325
Epoch 27/30
73257/73257 - 148s - loss: 0.6218 - accuracy: 0.8023 - val_loss: 0.5697 - val_accuracy: 0.8355
Epoch 28/30
73257/73257 - 144s - loss: 0.6201 - accuracy: 0.8030 - val_loss: 0.5983 - val_accuracy: 0.8263
Epoch 29/30
73257/73257 - 132s - loss: 0.6195 - accuracy: 0.8050 - val_loss: 0.5848 - val_accuracy: 0.8251
Epoch 30/30
73257/73257 - 133s - loss: 0.6174 - accuracy: 0.8047 - val_loss: 0.5716 - val_accuracy: 0.8343
```

```
In [442]: plt.plot(history_cnn.history['loss'])
          plt.plot(history_cnn.history['val_loss'])
          plt.title('Loss vs. epochs')
          plt.ylabel('Loss')
          plt.xlabel('Epoch')
          plt.legend(['Training', 'Validation'], loc='upper right')
          plt.show()
```



```
In [443]: plt.plot(history_cnn.history['accuracy'])
          plt.plot(history_cnn.history['val_accuracy'])
          plt.title('Accuracy vs. epochs')
          plt.ylabel('Accuracy')
          plt.xlabel('Epoch')
          plt.legend(['Training', 'Validation'], loc='lower right')
          plt.show()
```



1.5 4. Get model predictions

- Load the best weights for the MLP and CNN models that you saved during the training run.
- Randomly select 5 images and corresponding labels from the test set and display the images with their labels.
- Alongside the image and label, show each model's predictive distribution as a bar chart, and the final model prediction given by the label with maximum probability.

```
In [445]: model2 = Sequential([
              Flatten(input_shape=(32,32,1)),
              Dense(128, activation='relu'),
              Dense(64, activation='relu'),
              Dense(32, activation='relu'),
              Dropout(0.5),
```

```
        Dense(10, activation='softmax')
    ])

    model2.load_weights('checkpoints_best_only/checkpoint')
```

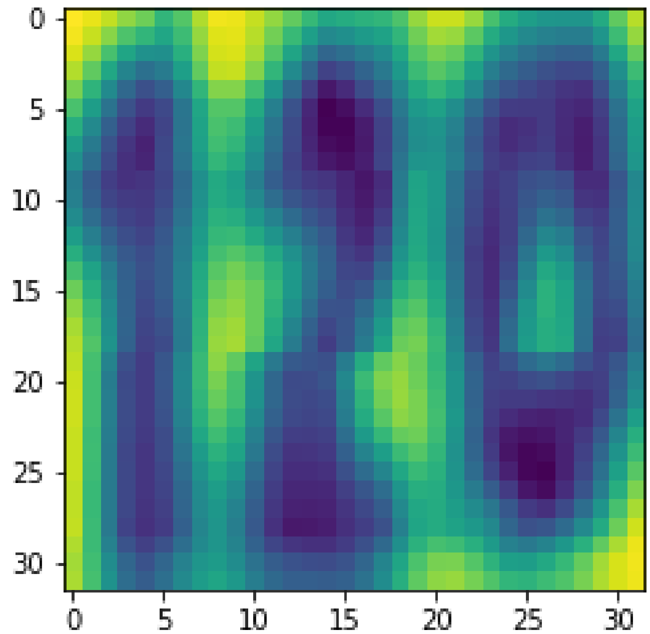
Out[445]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fb0f62a6978>

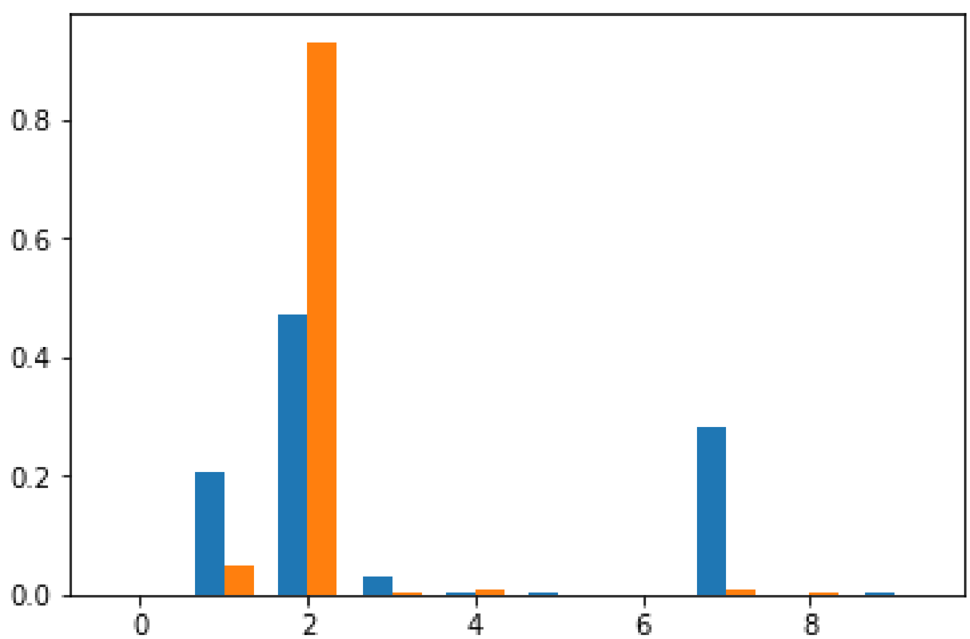
In [452]:
```
model_cnn2 = Sequential([
        Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(32,32,1)),
        MaxPooling2D(pool_size=(3,3)),
        BatchNormalization(),
        Flatten(),
        Dense(64, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])

    model_cnn2.load_weights('checkpoints_best_only_cnn/checkpoint')
```

Out[452]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fb0f94e97b8>

In [487]:
```
x = np.arange(10)  # the label locations
width = 0.35  # the width of the bars

for i in range(5):
    n = random.randrange(0,X_test.shape[0])
    preds_probs = model2.predict(X_test[n].reshape(1,32,32,1))
    preds_probs_cnn = model_cnn2.predict(X_test[n].reshape(1,32,32,1))
    print(f'Image#(below): {n}\nlabel:{y_test[n]}\nPLM Prediction:{np.argmax(preds_p
    plt.imshow(X_test[n,:,:,0])
    plt.show()

    #Bar Chart
    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width/2, preds_probs[0], width, label='PLM')
    rects2 = ax.bar(x + width/2, preds_probs_cnn[0], width, label='CNN')

    plt.show()
    print('=========================================')
```

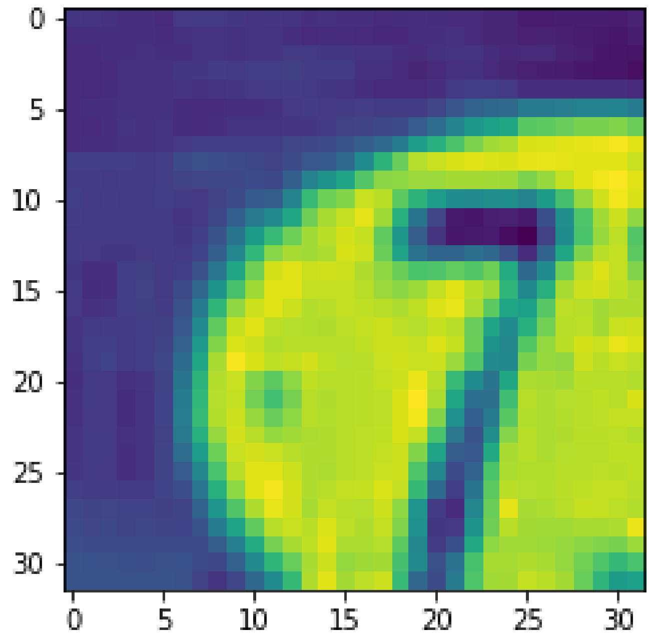
```
Image#(below): 2790
label:[2]
PLM Prediction:[2]
CNN Prediction:[2]
```

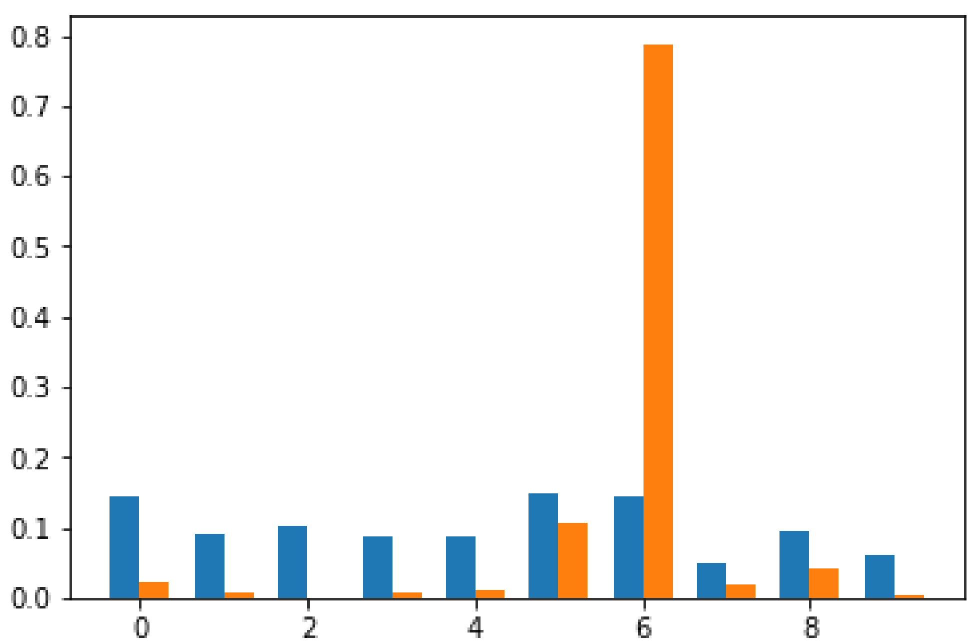




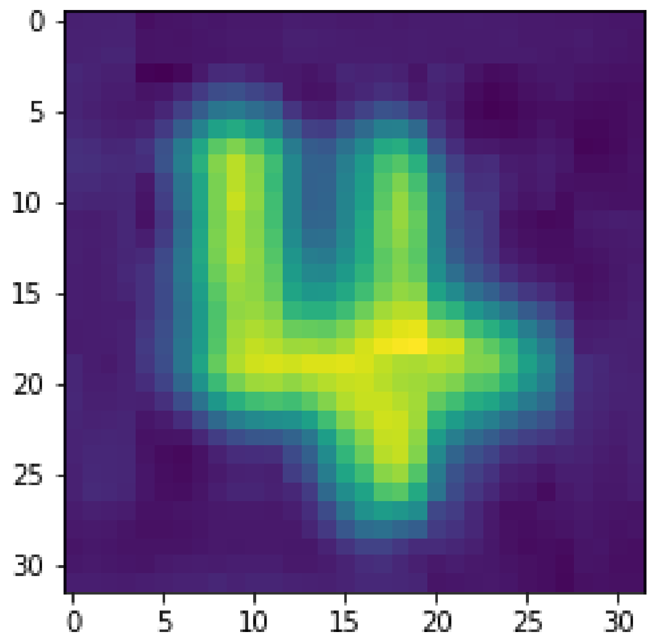

```
==========================================
Image#(below): 24596
label:[7]
```

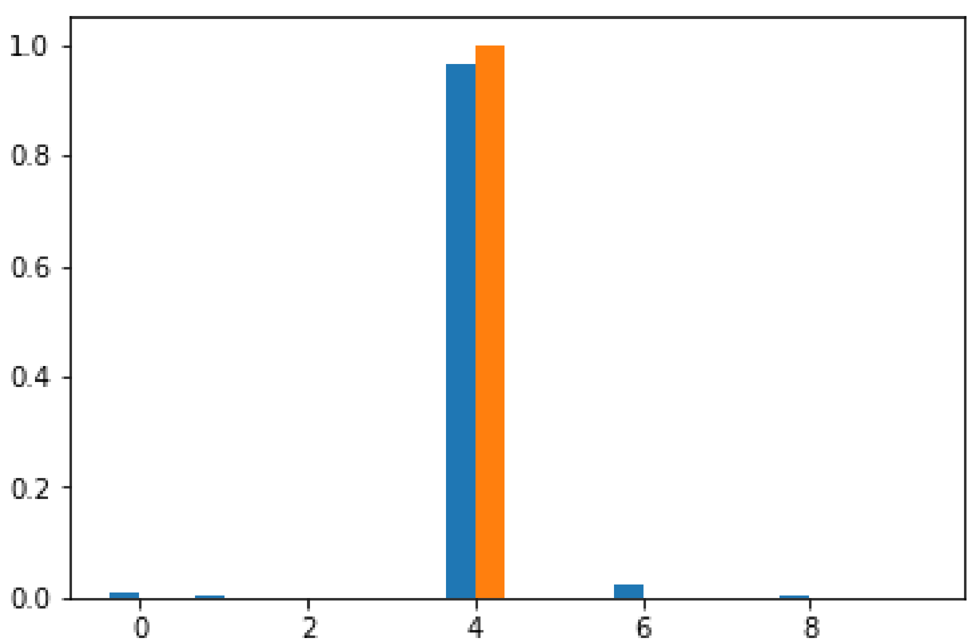
```
PLM Prediction:[5]
CNN Prediction:[6]
```



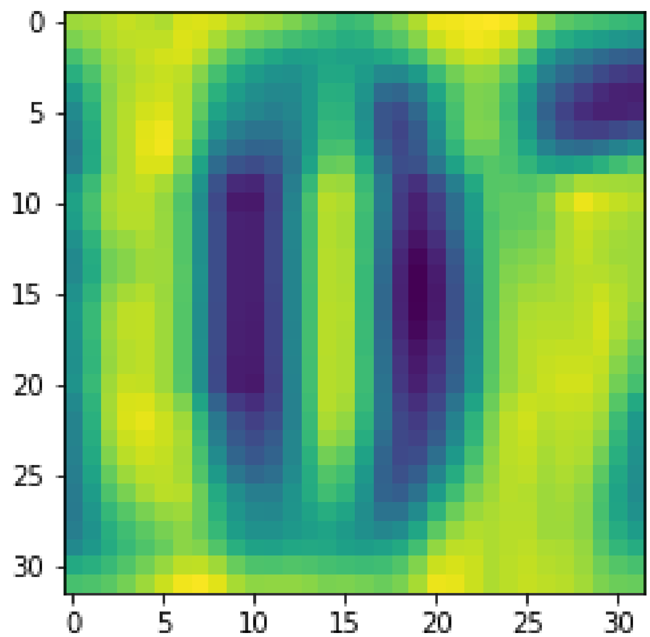


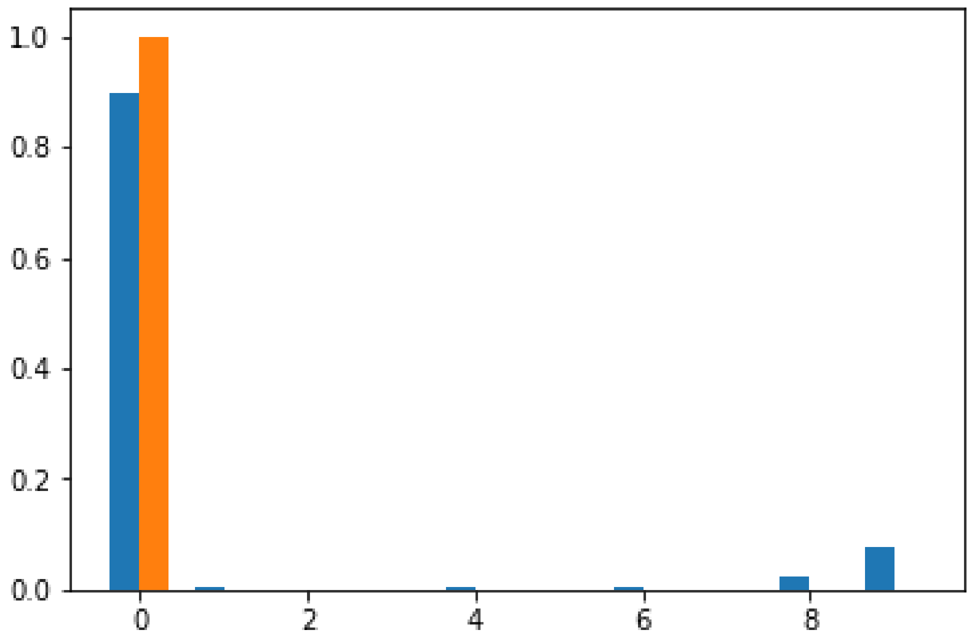
```
==========================================
Image#(below): 16869
label:[4]
PLM Prediction:[4]
CNN Prediction:[4]
```



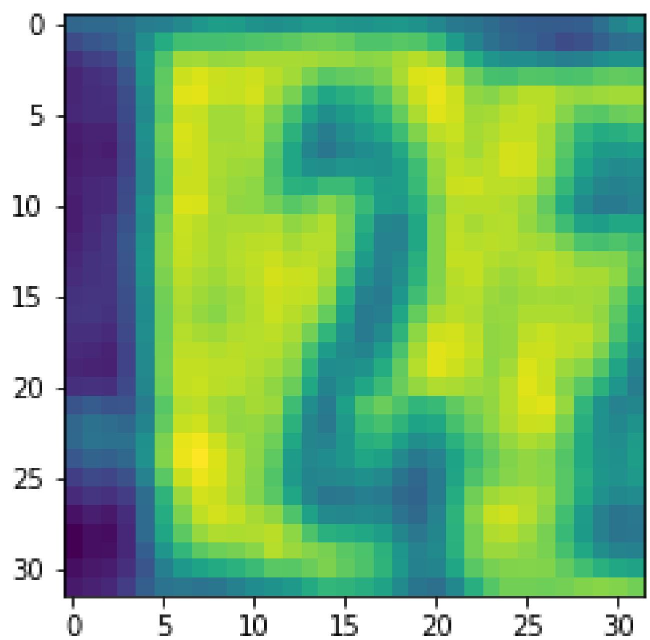


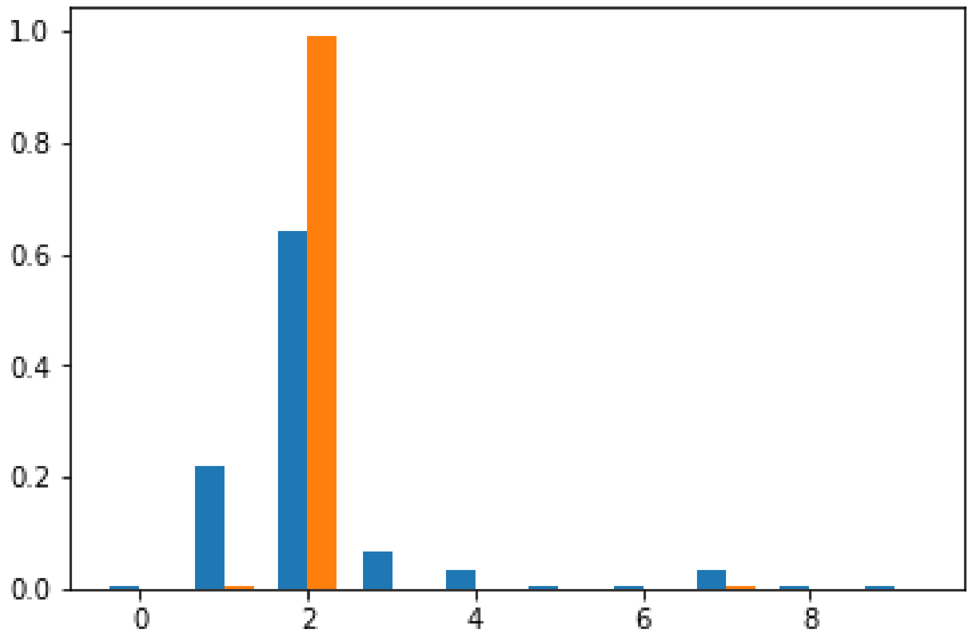
```
==========================================
Image#(below): 12675
label:[0]
PLM Prediction:[0]
CNN Prediction:[0]
```






```
=========================================
Image#(below): 18475
label:[2]
PLM Prediction:[2]
CNN Prediction:[2]
```





==

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```