Reyhane Shahrokhian 99521361

HomeWork4 of DeepLearning Course

Dr. DavoodAbadi

Question 1:

1-1:

- It is a hyperparameter tuning library for Keras. Hyperparameter tuning is the process of finding the best set of hyperparameters for a machine learning model to improve its performance.
  Keras Tuner provides a convenient way to search through different combinations of hyperparameters, such as learning rates, dropout rates, and the number of hidden layers, to optimize the performance of your neural network. It typically uses techniques like random search or Bayesian optimization to efficiently explore the hyperparameter space.
- Keras Tuner can be used to optimize hyperparameters for convolutional neural networks (CNNs) for image classification tasks. CNNs have several hyperparameters, such as the number of convolutional layers, filter sizes, number of filters, learning rates, and dropout rates. Finding the best combination of these hyperparameters is crucial for achieving optimal model performance.
  Keras Tuner allows you to define a search space for these hyperparameters. For example, you can specify the range of values for the number of filters, the choice of activation functions, and other relevant parameters. Keras Tuner will iteratively train and evaluate models with different configurations to find the best hyperparameters.

- The tuner is responsible for searching the hyperparameter space and finding the best set of hyperparameters for a given machine learning model. There are several tuners available in Keras Tuner, Such as Bayesian Optimization, Hyperband, and Random Search.

  Random Search Tuner: The Random Search tuner randomly samples hyperparameter combinations from a predefined search space. It explores various configurations without following a specific strategy.

  Bayesian Optimization Tuner: The Bayesian Optimization tuner uses a probabilistic model (usually Gaussian process) to model the surrogate function of the objective. It iteratively selects hyperparameter combinations based on the model's predictions and an acquisition function that balances exploration and exploitation.

  Hyperband Tuner: The Hyperband tuner is a resource allocation algorithm that runs multiple configurations in parallel. It allocates resources dynamically based on the performance of configurations, early stopping less promising ones, and focusing resources on the more promising ones.

  The suitability of a particular hyperparameter tuning algorithm depends on several factors, including the nature of the dataset, the complexity of the model, and the available computational resources. I would choose Random Search for the MNIST dataset, because MNIST is a relatively simple dataset and its images are grayscale and relatively small. Random Search is well-suited for situations where the optimal hyperparameters are not highly dependent on intricate relationships or specific configurations and also it is computationally efficient compared to more sophisticated algorithms like Bayesian optimization.

1-2:

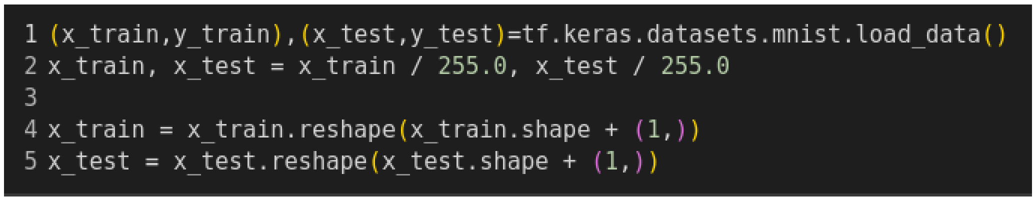
- The MNIST dataset contains 60,000 training images of handwritten digits from 0 to 9 and 10,000 images for testing. So, the MNIST dataset has 10 different classes. The handwritten digits images are represented as a 28×28 matrix where each cell contains grayscale pixel value.
- After defining the model function and the tuner, the optimal hyperparameters for the model are searched using the tuner.search(). Then the best model is retrieved and the best model is evaluated on the test dataset.
- Dropout and pooling are two techniques commonly used in neural network architectures to improve model performance and generalization.

  Dropout is a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to zero during training. This prevents complex co-adaptations on training data and promotes more robust generalization to unseen data.

  Pooling layers are used to reduce the spatial dimensions (width and height) of the input volume, which helps in reducing the computational complexity and controlling overfitting. Max pooling, in particular, retains the most important information from the feature maps.

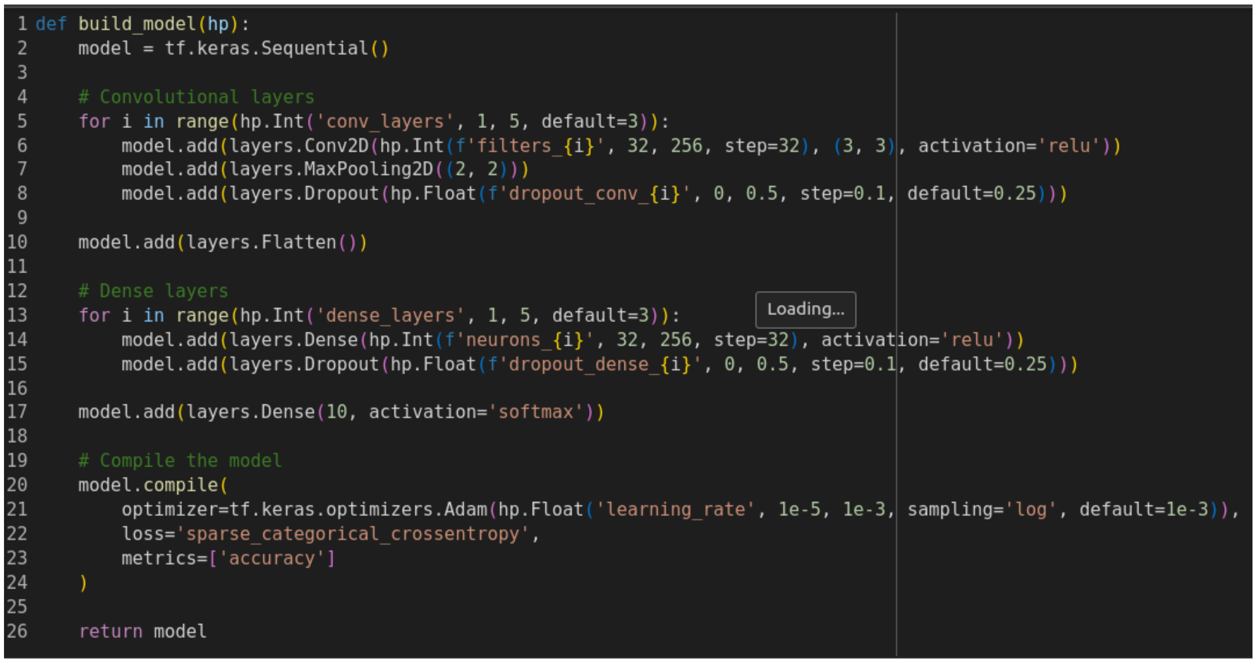
1-3:

- For implementing the model, first of all the dataset needed to be downloaded and input should be reshaped.



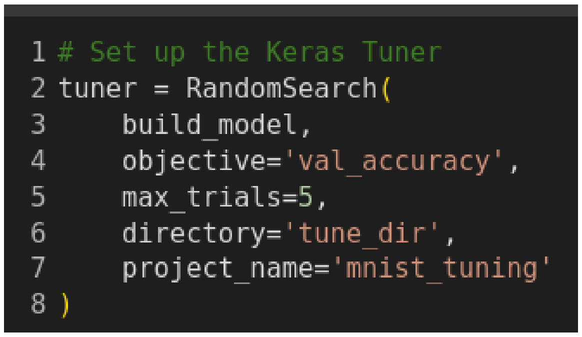
```
(x_train,y_train),(x_test,y_test)=tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.reshape(x_test.shape + (1,))
```

Then, the model should be defined with respect to the hyperparameter's tuning.

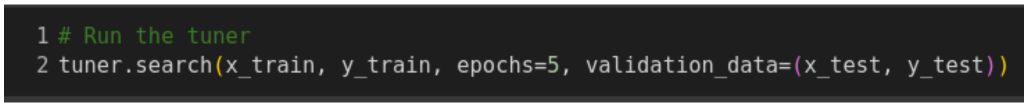


```python
def build_model(hp):
    model = tf.keras.Sequential()

    # Convolutional layers
    for i in range(hp.Int('conv_layers', 1, 5, default=3)):
        model.add(layers.Conv2D(hp.Int(f'filters_{i}', 32, 256, step=32), (3, 3), activation='relu'))
        model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Dropout(hp.Float(f'dropout_conv_{i}', 0, 0.5, step=0.1, default=0.25)))

    model.add(layers.Flatten())

    # Dense layers
    for i in range(hp.Int('dense_layers', 1, 5, default=3)):
        model.add(layers.Dense(hp.Int(f'neurons_{i}', 32, 256, step=32), activation='relu'))
        model.add(layers.Dropout(hp.Float(f'dropout_dense_{i}', 0, 0.5, step=0.1, default=0.25)))

    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(
        optimizer=tf.keras.optimizers.Adam(hp.Float('learning_rate', 1e-5, 1e-3, sampling='log', default=1e-3)),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model
```

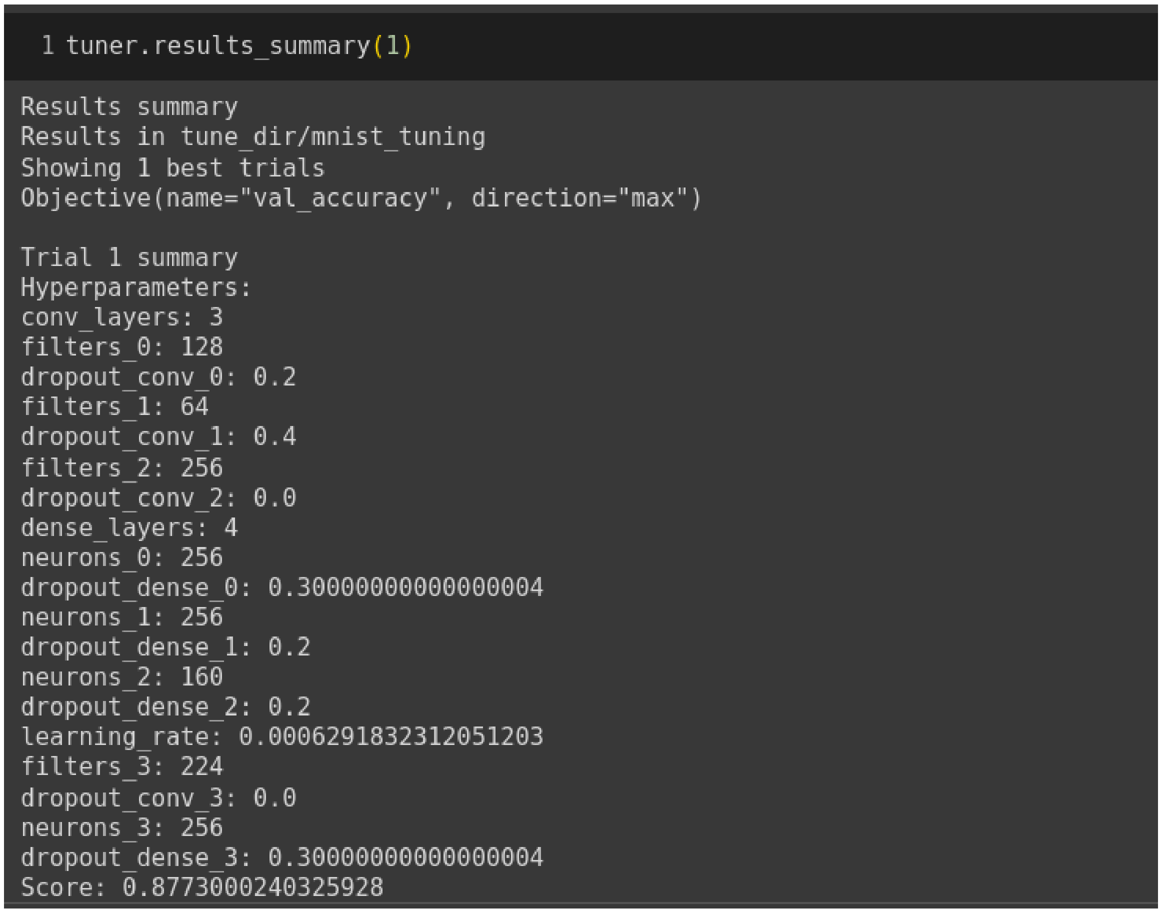
Now, the keras tuner has to be set and run to search for the best model.



```python
# Set up the Keras Tuner
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    directory='tune_dir',
    project_name='mnist_tuning'
)
```



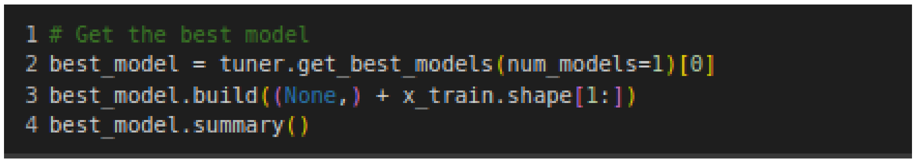
```python
# Run the tuner
tuner.search(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```



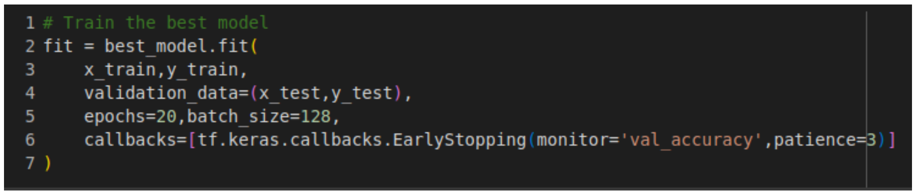
```
tuner.results_summary(1)
```

```
Results summary
Results in tune_dir/mnist_tuning
Showing 1 best trials
Objective(name="val_accuracy", direction="max")

Trial 1 summary
Hyperparameters:
conv_layers: 3
filters_0: 128
dropout_conv_0: 0.2
filters_1: 64
dropout_conv_1: 0.4
filters_2: 256
dropout_conv_2: 0.0
dense_layers: 4
neurons_0: 256
dropout_dense_0: 0.30000000000000004
neurons_1: 256
dropout_dense_1: 0.2
neurons_2: 160
dropout_dense_2: 0.2
learning_rate: 0.0006291832312051203
filters_3: 224
dropout_conv_3: 0.0
neurons_3: 256
dropout_dense_3: 0.30000000000000004
Score: 0.8773000240325928
```

At the end, the best model is trained with the dataset.

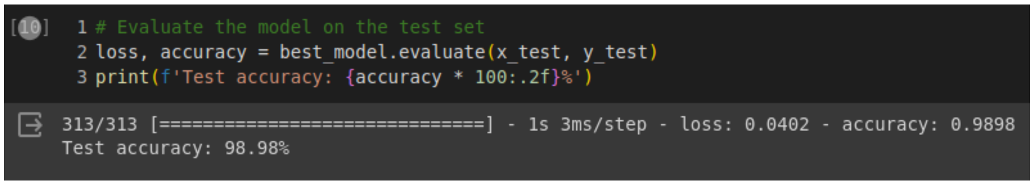


```
# Get the best model
best_model = tuner.get_best_models(num_models=1)[0]
best_model.build((None,) + x_train.shape[1:])
best_model.summary()
```



```
# Train the best model
fit = best_model.fit(
    x_train,y_train,
    validation_data=(x_test,y_test),
    epochs=20,batch_size=128,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',patience=3)]
)
```

And then the model is evaluated.



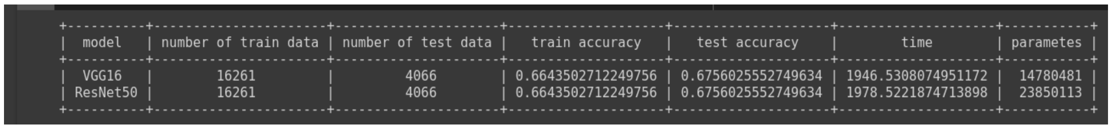
```
# Evaluate the model on the test set
loss, accuracy = best_model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy * 100:.2f}%')
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0402 - accuracy: 0.9898
Test accuracy: 98.98%
```

- The choice of (3, 3) is a common and widely used filter size in convolutional neural networks (CNNs). A 3x3 filter captures a small local receptive field that helps the network learn local features and patterns in the input images.
  Stacking multiple layers of 3x3 filters allows the network to learn hierarchical features. Each layer captures different levels of abstraction, and stacking them helps in building complex representations.
  Smaller filter sizes are computationally more efficient. A 3x3 filter requires fewer parameters to be learned compared to larger filter sizes, which helps in reducing the overall model complexity.
- Dropout and pooling are two techniques commonly used in neural network architectures to improve model performance and generalization.
  Dropout is a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to zero during training. This prevents complex co-adaptations on training data and promotes more robust generalization to unseen data.
  Pooling layers are used to reduce the spatial dimensions (width and height) of the input volume, which helps in reducing the computational complexity and controlling overfitting. Max pooling, in particular, retains the most important information from the feature maps.

Question 2:

As the TA mentioned the explanation is not needed and the code file is attached. And that's the conclusion part:



```
+----------+----------------------+---------------------+--------------------+--------------------+--------------------+-----------+
|  model   | number of train data | number of test data |   train accuracy   |   test accuracy    |        time        | parametes |
+----------+----------------------+---------------------+--------------------+--------------------+--------------------+-----------+
|  VGG16   |        16261         |        4066         | 0.6643502712249756 | 0.6756025552749634 | 1946.5308074951172 |  14780481 |
| ResNet50 |        16261         |        4066         | 0.6643502712249756 | 0.6756025552749634 | 1978.5221874713898 |  23850113 |
+----------+----------------------+---------------------+--------------------+--------------------+--------------------+-----------+
```

Question 3:

We feed the system with only one input at any given moment. This means that the inherent temporal sequencing requirement, where multiple pieces of data should be provided to an LSTM within a defined window, becomes problematic. The challenge lies in the fact that the model is designed to handle a single input per instance, making it difficult to effectively capture patterns and dependencies that extend across multiple time steps, which is typically crucial in scenarios involving temporal sequences.

Question 4:

4-1:

A Recurrent Neural Network (RNN) is a type of neural network designed for sequential data. It maintains a hidden state that is updated at each time step, allowing the network to capture dependencies and patterns in sequences. RNNs are particularly useful for tasks where the order and context of the input data matter, such as natural language processing, speech recognition, and time series analysis.

A Convolutional Neural Network (CNN) is a type of neural network designed for grid-like data, such as images and videos. It uses convolutional operations to automatically learn spatial hierarchies and local patterns in the input. CNNs are well-suited for tasks involving image classification, object detection, and image segmentation.

4-2:

- *Parameter count:*

  CNNs often have less parameters due to their weights sharing, while RNNs have more parameters because of using memory gates and having highly dependencies on the sequential nature of their computations.

- *Parallelization Capability:*

  Training RNNs can be computationally intensive and challenging to parallelize because the computation at each time step depends on the previous time step. But CNNs have a clear advantage due to the parallel nature of convolutional operations, making them more efficient for training on modern parallel processing hardware.

Question 5:

5-1:

We should know how to compute parameters and output_shapes:

Conv with padding=same:

$Parameters = filters \times (kernel\_size \times kerne\_size \times input\_channel\_size + 1)$

$Output\_width = Input\_width$

$Output\_channel = filters$

Dilated_Conv:

$Parameters = filters \times (kernel\_size \times kerne\_size \times input\_channel\_size + 1)$

$Output\_width = \frac{Input\_width + 2 \times padding - delation\_rate \times (kernel\_size - 1) - 1}{stride} + 1$

$Output\_channel = filters$

MaxPool:

$Parameters = 0$

$Output\_width = \frac{Input\_width - Pool\_size}{stride} + 1$

$Output\_channel$ = stay the same

If both stride and dilation consider(like pytorch):

Layers	**Output_shape**	**Parameters**
Conv(64,(3,3),stride=1,padding ='same')	$(256, 256, 64)$	$64 \times (3 \times 3 \times 3 + 1) = 1792$
Dilated-Conv(32,(5,5),stride=2, dilation rate=2,padding='valid')	$(124, 124, 32)$	$32 \times (5 \times 5 \times 64 + 1) = 51232$
Max-pool(size=(2,2),stride=2)	(62, 62, 32)	0
Conv(128,(3,3),stride=1,padding ='same')	(62, 62, 128)	$128 \times (3 \times 3 \times 32 + 1) = 36992$
Dilated-Conv(64,(5,5),stride=2, dilation rate=4,padding='valid')	$(24,\ 24,\ 64)$	$64 \times (5 \times 5 \times 128 + 1) = 204864$
Max-pool(size=(2,2),stride=2)	$(12, 12, 64)$	0
Conv(256,(3,3),stride=1,padding ='same')	$(12, 12, 256)$	$256 \times (3 \times 3 \times 64 + 1) = 147712$
Dilated-Conv(128,(5,5),stride=2 ,dilation rate=8,padding='valid')	$(-\ 9, -\ 9, 128)$	$128 \times (5 \times 5 \times 256 + 1) = 819328$
Max-pool(size=(2,2),stride=2)	$(-\ 4, -\ 4, 128)$	0

If stride doesn't consider with dilation(like tensorflow):

Layers	**Output_shape**	**Parameters**
Conv(64,(3,3),stride=1,padding ='same')	$(256, 256, 64)$	$64 \times (3 \times 3 \times 3 + 1) = 1792$
Dilated-Conv(32,(5,5),stride=2, dilation rate=2,padding='valid')	$(248, 248, 32)$	$32 \times (5 \times 5 \times 64 + 1) = 51232$
Max-pool(size=(2,2),stride=2)	(124, 124, 32)	0
Conv(128,(3,3),stride=1,padding ='same')	(124, 124, 128)	$128 \times (3 \times 3 \times 32 + 1) = 36992$
Dilated-Conv(64,(5,5),stride=2, dilation rate=4,padding='valid')	$(108,\ 108,\ 64)$	$64 \times (5 \times 5 \times 128 + 1) = 204864$
Max-pool(size=(2,2),stride=2)	$(54, 54, 64)$	0

Conv(256,(3,3),stride=1,padding='same')	$(54, 54, 256)$	$256 \times (3 \times 3 \times 64 + 1) = 147712$
Dilated-Conv(128,(5,5),stride=2,dilation rate=8,padding='valid')	$(22, 22, 128)$	$128 \times (5 \times 5 \times 256 + 1) = 819328$
Max-pool(size=(2,2),stride=2)	$(11, 11, 128)$	0

5-2:

In Conv2D the output dimension is:

$$Output = \frac{Input\_width - filter\_size + 2 \times padding}{stride} + 1$$

Now if we consider the stride =1 and output=input we have:

$$0 =- filter\_size + 2 \times padding + 1$$

So : $padding = \frac{filter\_size - 1}{2}$

Question 6:

6-1:

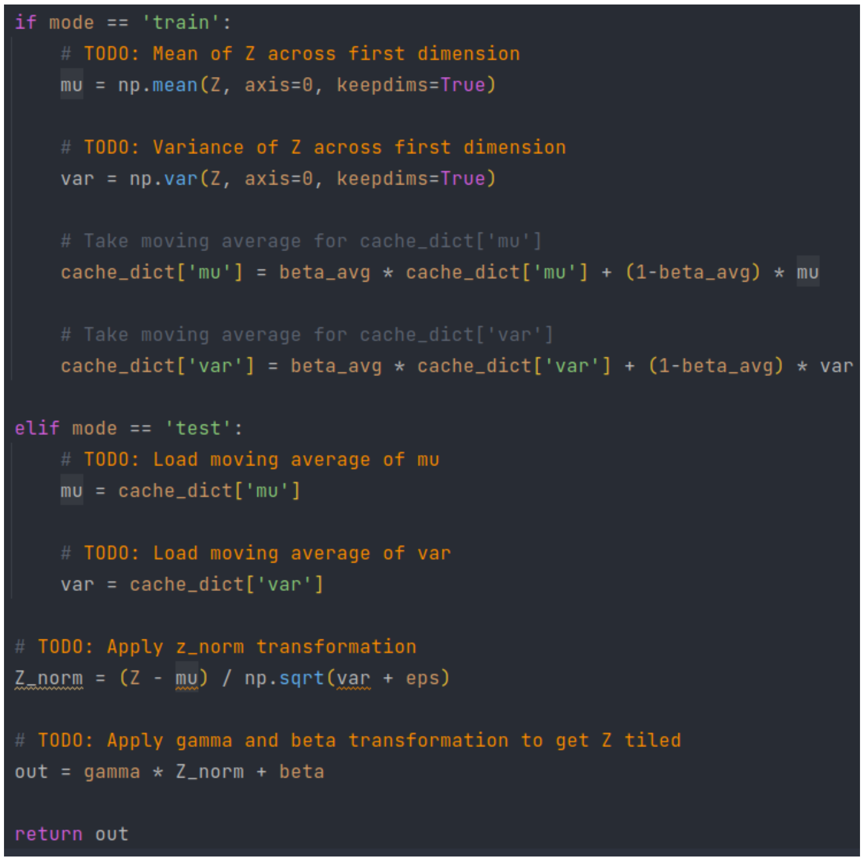
- False. There is no reason to assume that batch normalization makes processing a single batch faster. We can expect batch normalization to make each training iteration slower because of the extra processing required during the forward pass and the additional hyperparameters that require training during back-propagation. That being said, we expect the network to converge faster than it would without batch normalization, so training of the network is expected to be completed faster overall.
- True. Batch normalization normalizes the inputs of each layer(which is actually the output of the previous layer) within a mini-batch by subtracting the mean and dividing by the standard deviation.

- False. We are simply limiting the effect of our weight initialization and then normalizing the input to each layer, weakening the impact of the initial weights determined but not necessarily “allowing the network to initialize our weights to smaller values close to zero”.

With help of cs230exam_win20.

6-2:

The code is completed based on the comments.



```
if mode == 'train':
    # TODO: Mean of Z across first dimension
    mu = np.mean(Z, axis=0, keepdims=True)

    # TODO: Variance of Z across first dimension
    var = np.var(Z, axis=0, keepdims=True)

    # Take moving average for cache_dict['mu']
    cache_dict['mu'] = beta_avg * cache_dict['mu'] + (1-beta_avg) * mu

    # Take moving average for cache_dict['var']
    cache_dict['var'] = beta_avg * cache_dict['var'] + (1-beta_avg) * var

elif mode == 'test':
    # TODO: Load moving average of mu
    mu = cache_dict['mu']

    # TODO: Load moving average of var
    var = cache_dict['var']

# TODO: Apply z_norm transformation
Z_norm = (Z - mu) / np.sqrt(var + eps)

# TODO: Apply gamma and beta transformation to get Z tiled
out = gamma * Z_norm + beta

return out
```

This code implements the forward pass with batch normalization which contains two sections: train and test. It calculates the mean and variance of inputs.

6-3:

ϵ is a small constant added to the variance estimate to prevent division by zero. The addition of that ensures the denominator is never zero, and it stabilizes the normalization process.

6-4:

Batch Normalization relies on computing the mean and standard deviation of the input data within a mini-batch. With a batch size of 1, there is only one data point, making it difficult to compute reliable statistics for normalization. This can lead to inaccurate normalization and might not provide the expected benefits.

Batch normalization has a slight regularization effect because it adds noise to the training process by normalizing the inputs based on batch statistics. With only one sample per batch, this regularization effect is diminished.

With a batch size of 1, the model may become too specific to the training data and might not generalize well to unseen data.

6-5:

For a fully connected layer, the number of weights and biases are given by the formulas:

Number of weights = number of inputs×number of neurons in the layer

Number of biases= number of neurons in the layer

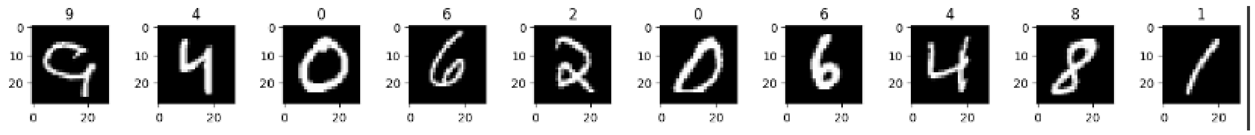
$\Rightarrow 20 \times 10 + 20 = 220$

Now, if batch normalization is added, each output of the fully connected layer will have two additional parameters (gamma and beta) associated with it.

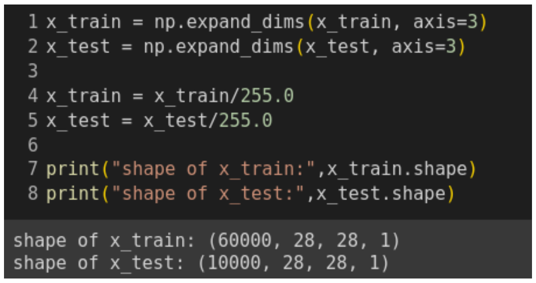
$\Rightarrow$Total trainable parameters with batch normalization=$220 + 2 \times 20 = 260$

Question 7:

First of all, we should follow the steps to load and shuffle the dataset. After that we show first 10 images of train data:

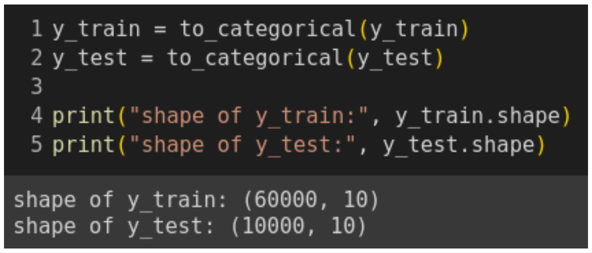


Now, we should add one dimension to x_train and x_test in order to use them with Conv2D. And we should also normalize the dimension of them. The labels should also change to one-hot.



```
x_train = np.expand_dims(x_train, axis=3)
x_test = np.expand_dims(x_test, axis=3)

x_train = x_train/255.0
x_test = x_test/255.0

print("shape of x_train:",x_train.shape)
print("shape of x_test:",x_test.shape)
```

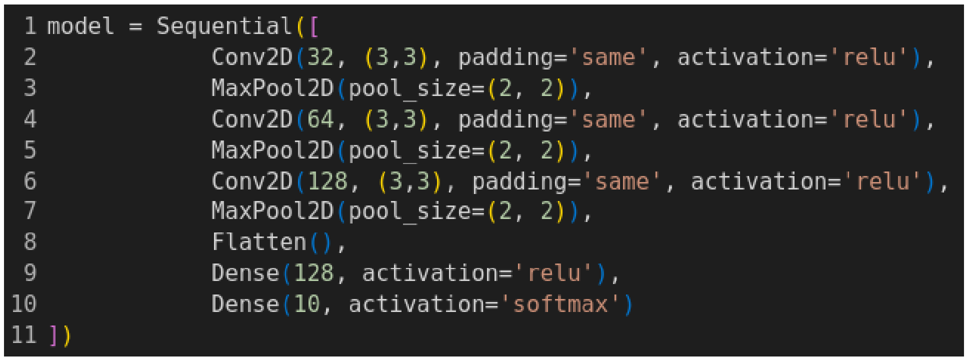
```
shape of x_train: (60000, 28, 28, 1)
shape of x_test: (10000, 28, 28, 1)
```



```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

print("shape of y_train:", y_train.shape)
print("shape of y_test:", y_test.shape)
```

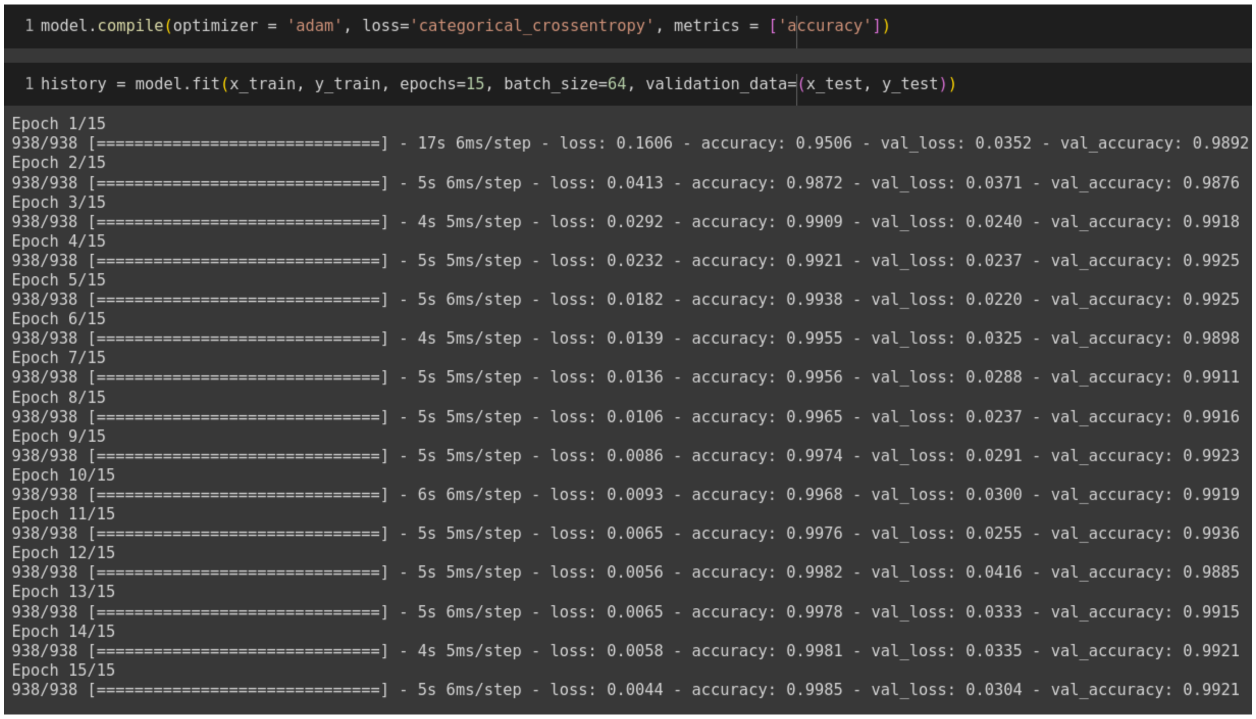
```
shape of y_train: (60000, 10)
shape of y_test: (10000, 10)
```

Now the model is implemented based on question explanations.



```
model = Sequential([
            Conv2D(32, (3,3), padding='same', activation='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Conv2D(64, (3,3), padding='same', activation='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Conv2D(128, (3,3), padding='same', activation='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Flatten(),
            Dense(128, activation='relu'),
            Dense(10, activation='softmax')
])
```

Then the model is trained and validated.



```
model.compile(optimizer = 'adam', loss='categorical_crossentropy', metrics = ['accuracy'])
```

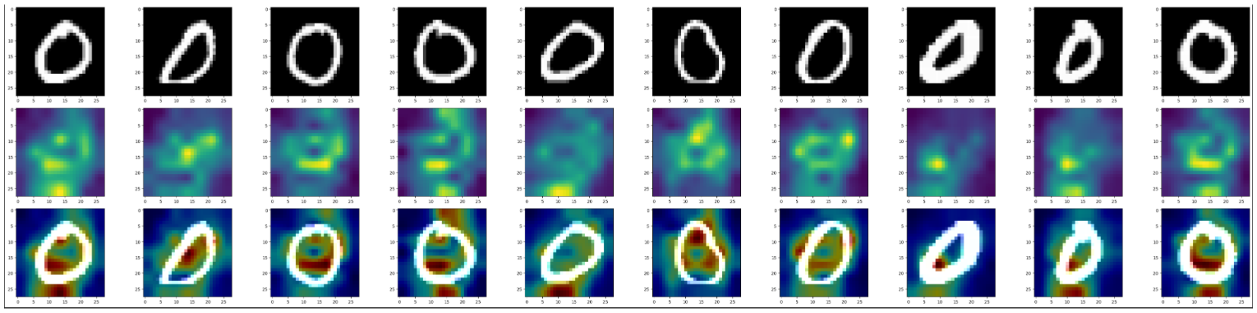
```
history = model.fit(x_train, y_train, epochs=15, batch_size=64, validation_data=(x_test, y_test))
```

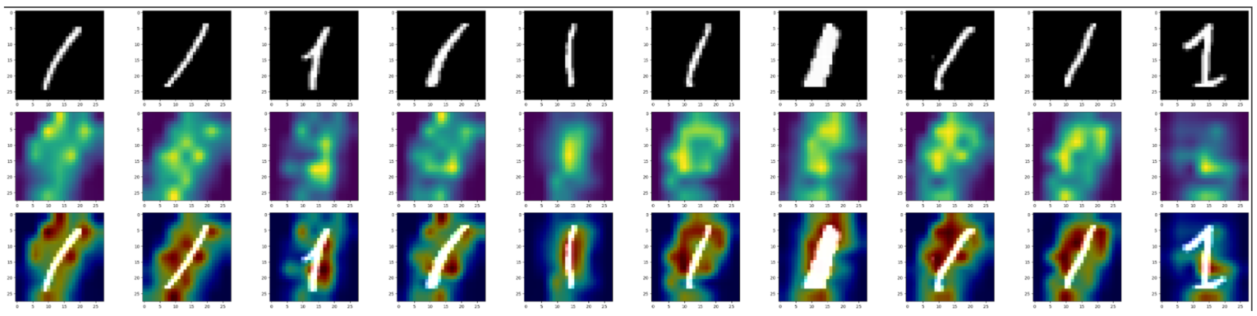
```
Epoch 1/15
938/938 [==============================] - 17s 6ms/step - loss: 0.1606 - accuracy: 0.9506 - val_loss: 0.0352 - val_accuracy: 0.9892
Epoch 2/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0413 - accuracy: 0.9872 - val_loss: 0.0371 - val_accuracy: 0.9876
Epoch 3/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0292 - accuracy: 0.9909 - val_loss: 0.0240 - val_accuracy: 0.9918
Epoch 4/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0232 - accuracy: 0.9921 - val_loss: 0.0237 - val_accuracy: 0.9925
Epoch 5/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0182 - accuracy: 0.9938 - val_loss: 0.0220 - val_accuracy: 0.9925
Epoch 6/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0139 - accuracy: 0.9955 - val_loss: 0.0325 - val_accuracy: 0.9898
Epoch 7/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0136 - accuracy: 0.9956 - val_loss: 0.0288 - val_accuracy: 0.9911
Epoch 8/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0106 - accuracy: 0.9965 - val_loss: 0.0237 - val_accuracy: 0.9916
Epoch 9/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0086 - accuracy: 0.9974 - val_loss: 0.0291 - val_accuracy: 0.9923
Epoch 10/15
938/938 [==============================] - 6s 6ms/step - loss: 0.0093 - accuracy: 0.9968 - val_loss: 0.0300 - val_accuracy: 0.9919
Epoch 11/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0065 - accuracy: 0.9976 - val_loss: 0.0255 - val_accuracy: 0.9936
Epoch 12/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0056 - accuracy: 0.9982 - val_loss: 0.0416 - val_accuracy: 0.9885
Epoch 13/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0065 - accuracy: 0.9978 - val_loss: 0.0333 - val_accuracy: 0.9915
Epoch 14/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0058 - accuracy: 0.9981 - val_loss: 0.0335 - val_accuracy: 0.9921
Epoch 15/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0044 - accuracy: 0.9985 - val_loss: 0.0304 - val_accuracy: 0.9921
```

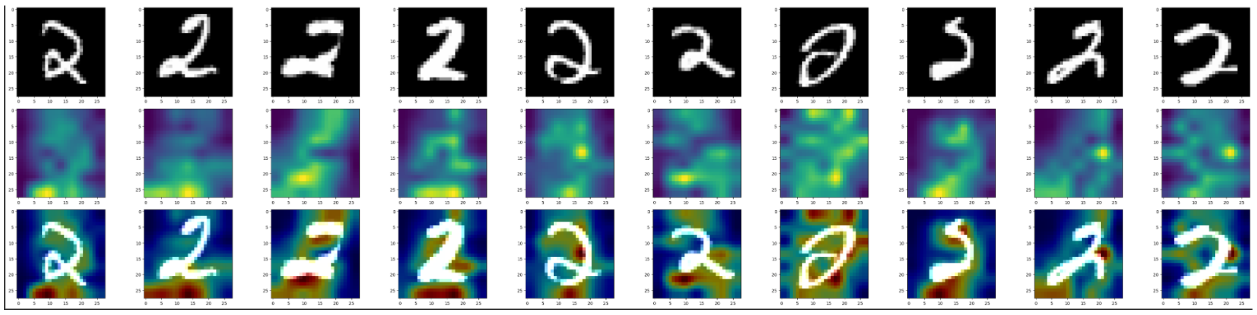
After training our model, we are supposed to implement the Grad-CAM algorithm which is taught in slides using Keras.

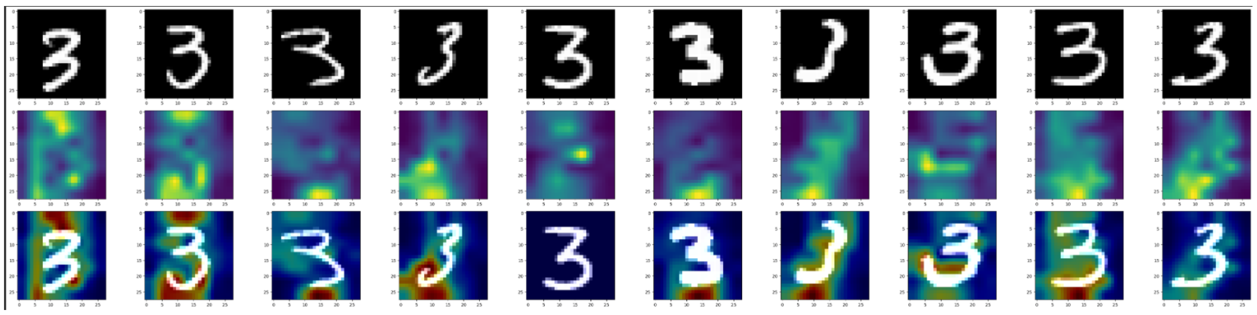
There is a grad_cam function which takes an image and the last convolutional layer to make its activation function as None. This function computes the gradient of output for a specific class of the image due to the last convolutional layer. This leads to having a heatmap matrix of shape 7*7 that shows the important parts of the image.

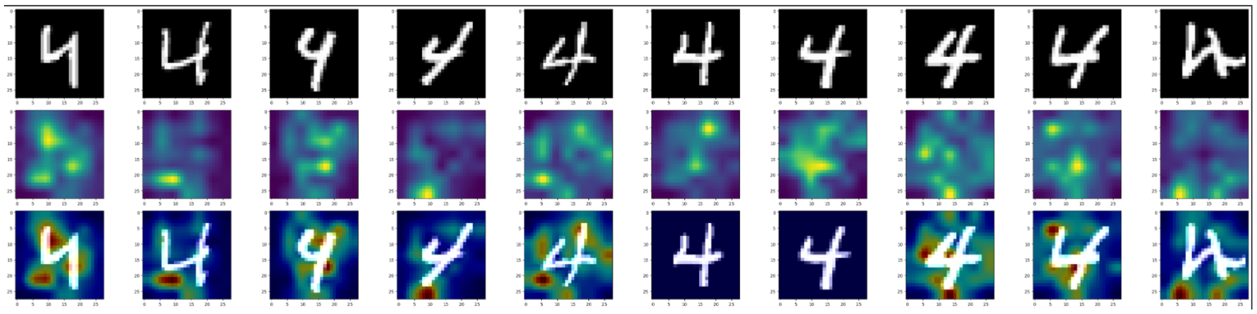
The superimpose function takes an image and the heatmap of that and makes an image combining these two.

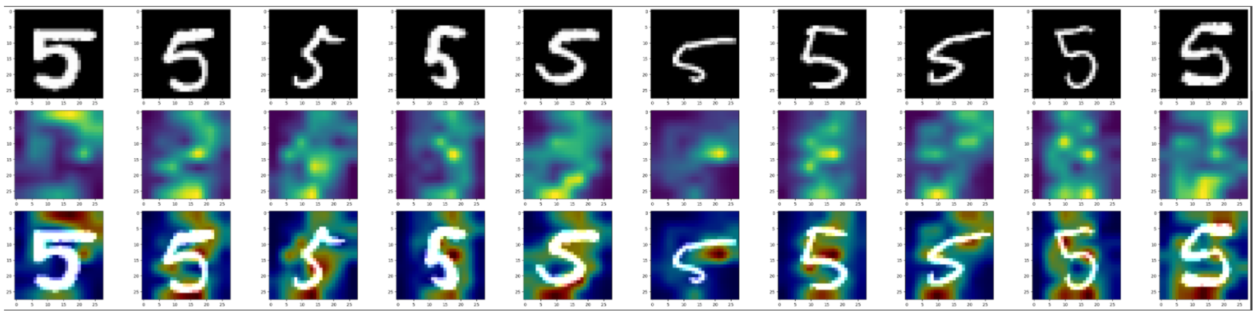


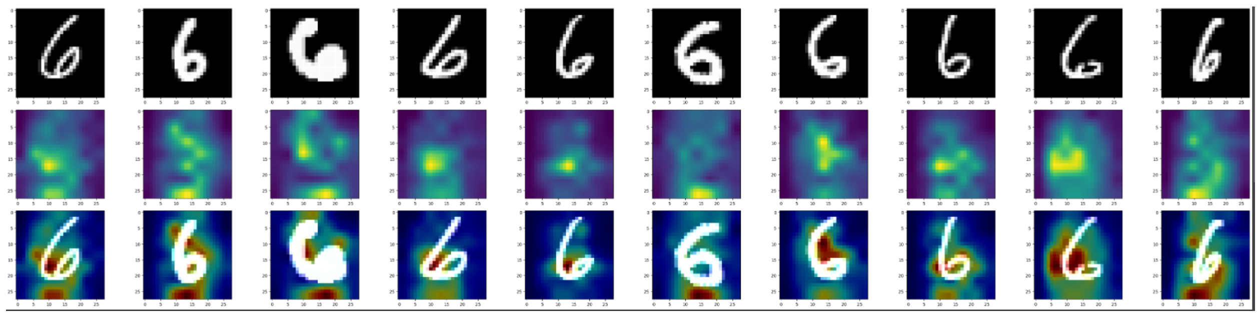


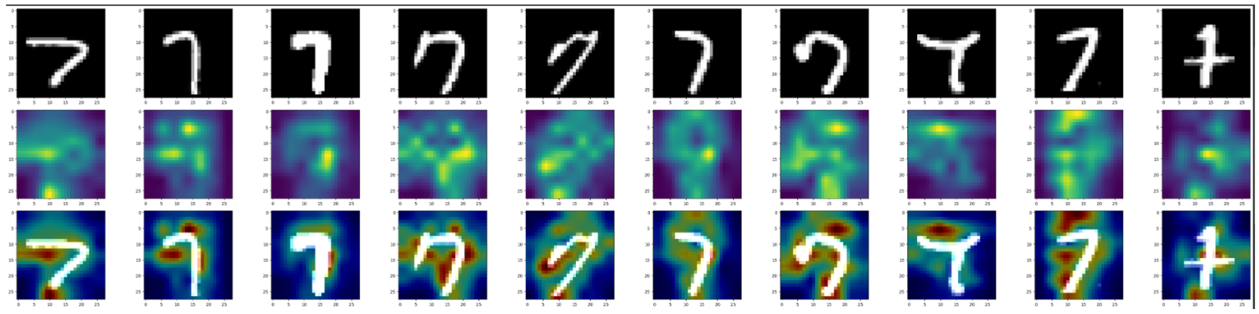


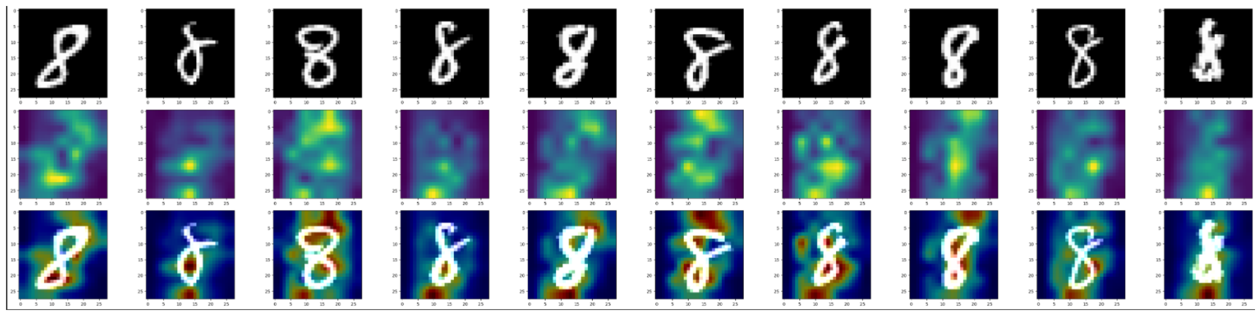


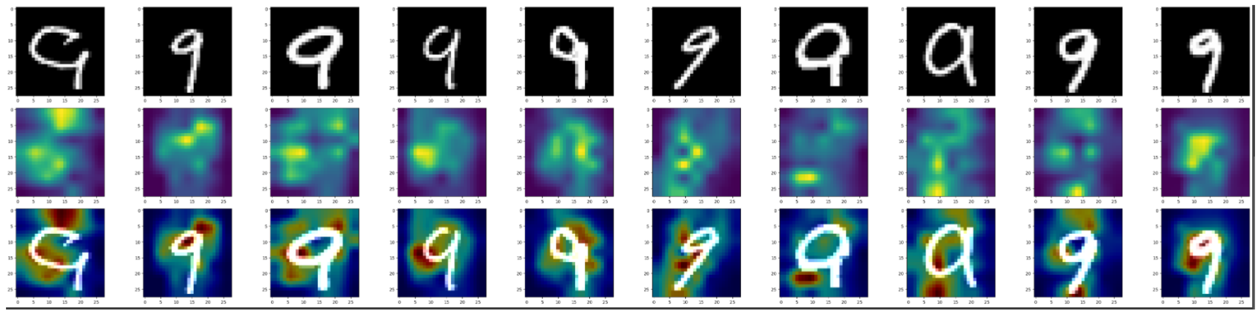












As it is shown in images, the last convolutional layer has more attention to the specific parts of images of each class which means this algorithm helps the model to classify easier based on the heatmap.

I use this site.