BTT004_week7_lab-solution

June 3, 2024

0.1 Lab 7: Implementing a Convolutional Neural Network Using Keras

[1]:
```
import tensorflow.keras as keras
import math
import time
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

A very common problem in computer vision is recognizing hand-written digits. The images of numerals are commonly used by data scientists and machine learning experts to train supervised learning models that specialize in decoding human handwriting. This is a classic problem that is often used in exercises and documentation. In this lab, you will train a convolutional neural network to classify hand-written digits. You will complete the following tasks:

1. Define your ML problem:
    - Define the label - what are you predicting?
    - Identify the features
2. Import the data and split the data into training and test data sets
3. Inspect and visualize the data
4. Prepare your data so that it is ready for modeling.
5. Construct a convolutional neural network
6. Train the convolutional neural network.
7. Evaluate the neural network model's performance on the training and test data.

For this lab, use the demo Implementing a Neural Network Using Keras that is contained in this unit as a reference.

Note: some of the code cells in this notebook may take a while to run

0.2 Part 1. Define Your ML Problem

We will implement a convolutional neural network to solve a handwriting recognition problem. The neural network will classify a hand-written digit.

Define the Label We will work with the MNIST data set, a famous collection of images used for handwriting recognition. It contains labeled images of handwritten digits from 0 to 9. Therefore, the label is a digit from 0 and 9. This is a multiclass classification problem.

Identify Features Each example corresponds to one hand-written image. The features will be comprised of numerical feature vectors (an n-dimensional array) that contain grey-scale pixel values that range from 0 to 255.

0.3 Part 2. Import the Data Set and Create Training and Test Sets

The MNIST data set comes preloaded in Keras. The `load_data()` function returns the data set split into training and test subsets. The cell below loads the data set and contains training and test data.

[2]:
```
# The mnist data set comes preloaded
mnist = keras.datasets.mnist

# Create training and test sets
(X_train, y_train),(X_test, y_test) = mnist.load_data()
```

0.3.1 Inspect the Data

Task: In the code cell below, inspect the datatype and dimensions (shape) of the training and test data (`X_train`, `y_train`, `X_test`, `y_test`).

[3]:
```
# YOUR CODE HERE

print(X_train.shape)
print(type(X_train))
print(y_train.shape)
print(type(y_train))
print(X_test.shape)
print(type(X_test))
print(y_test.shape)
print(type(y_test))
```

```
(60000, 28, 28)
<class 'numpy.ndarray'>
(60000,)
<class 'numpy.ndarray'>
(10000, 28, 28)
<class 'numpy.ndarray'>
(10000,)
<class 'numpy.ndarray'>
```

Notice that the training and test data sets are NumPy arrays.

- Training data: `X_train` is a three-dimensional array of shape `(60000, 28, 28)`. It contains grayscale image data. Pixel values range from 0 to 255. `y_train` is a one-dimensional array with shape `(6000,)`. It contains digit labels (integers in range 0-9).
- Test data: `X_test` is a three-dimensional array of shape `(10000, 28, 28)`. It contains grayscale image data. Pixel values range from 0 to 255.`y_test` is a one-dimensional array with shape `(1000,)`. It contains digit labels (integers in range 0-9).

Let's take a look at the data in more detail. Let's inspect the first example (which contains an image) in X_train:

[4]:
```
X_train[0].shape
```

[4]:
```
(28, 28)
```

[5]:
```
X_train[0]
```

[5]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
        253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
        205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
         90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
        190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
        253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
```

```
 241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
 148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0]], dtype=uint8)
```

As expected, the first example in the training data is a 28 x 28 array. This array encodes the grayscale value of the hand-written image, i.e., each entry in the 28 x 28 array encodes the intensity (darkness) of the corresponding pixel.

0.3.2 Visualize the Data

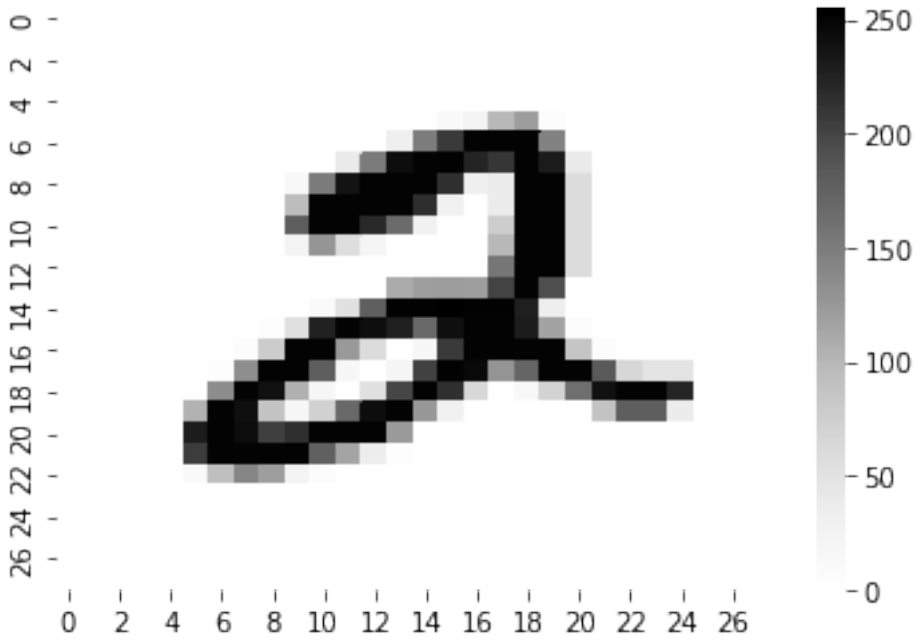
Let's visualize an image below.

Task: In the code cell below, use the Seaborn`heatmap()` function to display any image contained in `X_train`.

[6]:
```
# YOUR CODE HERE

# Solutions may vary. stutdents do not have to add the greyscale param
sns.heatmap(X_train[5], cmap=plt.cm.Greys)
```

[6]:
```
<AxesSubplot:>
```



Task: Inspect the corresponding label in `y_train` to confirm that the label matches the image you see in the heatmap above.

[7]:
```
# YOUR CODE HERE

# Solution
# solutions will vary depending on the image they chose to inspect in X_train
y_train[5]
```

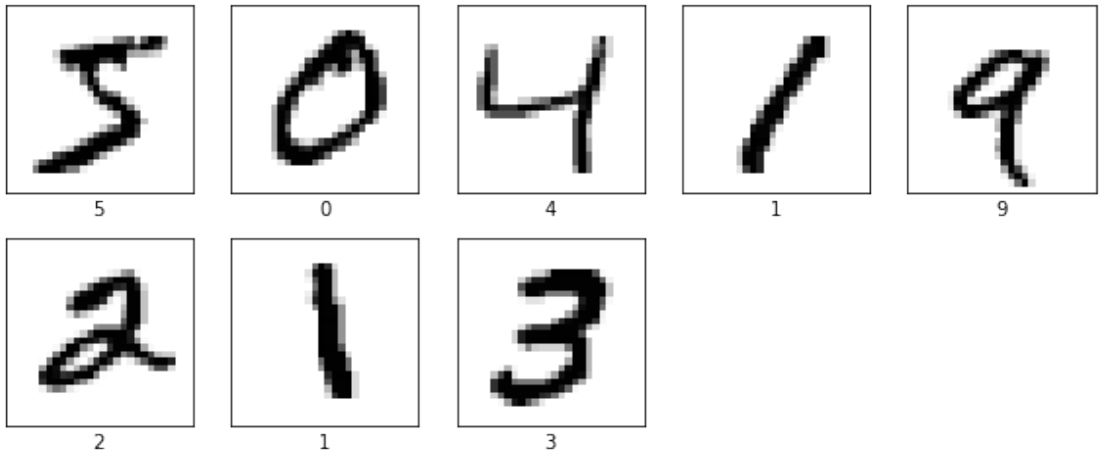
[7]:
```
2
```

Task: Which digit appeared in your heatmap? Did it match its label? Record your findings in the cell below.

We've created a function `plot_imgs()` to help us visualize the image data. Let's use this function to inspect a few more examples in the training data. Execute the two code cells below.

[8]:
```
# Function to visualize the training data
def plot_imgs(images, labels=None):
    subplots_x = int(math.ceil(len(images) / 5))
    plt.figure(figsize=(10,2*subplots_x))
    for i in range(min(len(images), subplots_x*5)):
        plt.subplot(subplots_x,5,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(images[i], cmap=plt.cm.binary)
        if labels is not None:
            plt.xlabel(labels[i])
    plt.show()
```

[9]:
```
# Visualize some example training images
plot_imgs(X_train[:8], y_train[:8])
```



0.4 Part 3. Prepare the Data

Let's now prepare our data to be suitable for a CNN.

Scale the Data Our MNIST data is raw data containing pixel values between 0 and 255. Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the training process. Therefore, it is a good practice to normalize the pixel values so that each pixel has a value between 0 and 1. This can be done by dividing all pixels values by the largest pixel value; that is 255.

Task: In the code cell below, normalize the pixel values in `X_train` and `X_test` to be between 0 and 1 by dividing all feature values by 255.0.

[10]:
```
# YOUR CODE HERE

# Solution (solutions may vary)

X_train, X_test = X_train / 255.0, X_test / 255.0
```

Reshape the Data A CNN in Keras requires a 4-dimensional array as input in the form: `(num_examples, image_dimension_X, image_dimension_Y, num_channels)`.

Since grayscale has only one color channel, every example in `X_train` would have the shape `(28, 28, 1)`. `X_test` should have the same dimensions.

Task: In the code cell below: 1. reshape every example in `X_train` to have the shape `(num_examples_X_train, 28, 28, 1)`. 1. reshape every example in `X_test` to have the shape `(num_examples_X_test, 28, 28, 1)`.

Hint: use the NumPy `reshape()` function. Consult the online documentation for more information.

[11]:
```
# YOUR CODE HERE

# Solution
X_train = np.reshape(X_train, (X_train.shape[0], 28, 28, 1))
X_test = np.reshape(X_test, (X_test.shape[0], 28, 28, 1))
```

0.5 Part 4. Construct the Convolutional Neural Network

0.5.1 Step 1. Define Model Structure

Next we will create our convolutional neural network structure. A CNN has three different types of hidden layers: a convolutional layer, a pooling layer, and a fully connected layer. When constructing a convolutional hidden layer, we will compose a 2D convolution, followed by a batch normalization, followed by an activation function.

Let's create the CNN structure. We will create an input layer, five hidden layers and an output layer (Note that there are different ways one can choose to construct a CNN in Keras):

- Input layer: The input layer will have the input shape corresponding to the number of features.
- Hidden layers: We will create five hidden layers:
  - Four hidden layers will be convolutional layers. They will be comprised of a 2D convolution, followed by a batch normalization, followed by an activation function. In this case, the activation function of choice is ReLU.
  - One hidden layer will be a pooling layer. We will add a layer that uses Global Average Pooling. This is a pooling operation designed to replace the final fully connected layer in classical CNN.
- Output layer: The output layer will have a width of 10.

To construct the CNN model using Keras, we will do the following:

- As before, we will use the Keras `Sequential` class to group a stack of layers. This will be our CNN model object. For more information, consult the Keras online Sequential class documentation.
- We will use the `InputLayer` class to create the input layer. For more information, consult the Keras online InputLayer class documentation.
- We will use the `Conv2D` class to create the convolutional layers. For more information, consult the Keras online Conv2D class documentation.
  - For batch normalization, we will use the `BatchNormalization` class. For more information, consult the Keras online BatchNormalization class documentation.
  - For the activation function, we will use the `ReLU` class. For more information, consult the Keras online ReLU class documentation.
- We will use the `GlobalAveragePooling2D` class to create the pooling layer. For more information, consult the Keras online GlobalAveragePooling2D class documentation.
- Finally, we will use the `Dense` class to create the output layer. For more information, consult the Keras online Dense class documentation.
- We will add each layer to the CNN model object.

Task: Follow these steps to complete the code in the cell below:

1. Create the CNN model object.

   - Use `keras.Sequential()` to create a model object, and assign the result to the variable `cnn_model`.

2. Create the input layer:

   - Call `keras.layers.InputLayer()` with the argument `input_shape` to specify the dimensions of the input. In this case, the dimensions will be the shape of each example (image) in `X_train` — assign this value to the argument `input_shape`.
   - Assign the result to the variable `input_layer`.
   - Add `input_layer` to the neural network model object `cnn_model`.

3. Create the first convolutional layer. You will accomplish this by doing the following:

   - Call `keras.layers.Conv2D()` and assign the result to the variable `conv_1`. You will pass two arguments to `Conv2D()`:
     1. The number of filters: `Conv2D()` requires an argument indicating the number of filters in the convolution. Layers in the network architecture that are closer to the input layer learn fewer convolutional filters whereas layers closer to the output layer learn more filters. Let's choose a value of 16 for the first layer.
     2. The kernal size: this argument specifies the size of the convolution window. We will choose a kernal size of 3.
   - Call `keras.layers.BatchNormalization()` without arguments. Assign the result to variable `batchNorm_1`.
   - Call `keras.layers.ReLU()` without arguments. Assign the result to avariable `ReLU_1`.

   - Add each of these items (`conv_1`, `batchNorm_1` and `ReLU_1`) in order to the neural network model object `cnn_model`.

4. Create the second convolutional layer using the same approach that you used to create the first convolutional layer, specifying 32 filters and a kernal size of 3. Add the layer to the neural network model object `cnn_model`.

5. Create the third convolutional layer using the same approach that you used to create the first convolutional layer, specifying 64 filters and a kernal size of 3. Add the layer to the neural network model object `cnn_model`.

6. Create the fourth convolutional layer using the same approach that you used to create the first convolutional layer, specifying 128 filters and a kernal size of 3. Add the layer to the neural network model object `cnn_model`.

7. Create the pooling layer:

    - Call `keras.layers.GlobalAveragePooling2D()` without arguments.
    - Assign the result to the variable `pooling_layer`.
    - Add `pooling_layer` to the neural network model object `cnn_model`.

8. Create the output layer:

    - Call `keras.layers.Dense()`. We will have one node per class. We have ten classes (digits from 0-9). Therefore, when creating the output later, specify 10 units. Do not specify an activation function.
    - Assign the result to the variable `output_layer`.
    - Add `output_layer` to the neural network model object `cnn_model`.

[12]:
```
# 1. Create CNN model object

#YOUR CODE HERE
# Solution
cnn_model = keras.Sequential()

# 2. Create the input layer and add it to the model object:
# YOUR CODE HERE

# SOLUTION:
input_layer = keras.layers.InputLayer(input_shape=(28, 28, 1))
cnn_model.add(input_layer)

# 3. Create the first convolutional layer and add it to the model object:
# YOUR CODE HERE

# SOLUTION:
conv_1= keras.layers.Conv2D(16, 3)
batchNorm_1 =  keras.layers.BatchNormalization()
relu_1 = keras.layers.ReLU()
cnn_model.add(conv_1)
cnn_model.add(batchNorm_1)
cnn_model.add(relu_1)
```

```
# 4. Create the second convolutional layer and add it to the model object:
# YOUR CODE HERE

# SOLUTION:
conv_2= keras.layers.Conv2D(32, 3)
batchNorm_2 =  keras.layers.BatchNormalization()
relu_2 = keras.layers.ReLU()
cnn_model.add(conv_2)
cnn_model.add(batchNorm_2)
cnn_model.add(relu_2)


# 5. Create the third convolutional layer and add it to the model object:
# YOUR CODE HERE

# SOLUTION:
conv_3= keras.layers.Conv2D(64, 3)
batchNorm_3 = keras.layers.BatchNormalization()
relu_3 = keras.layers.ReLU()
cnn_model.add(conv_3)
cnn_model.add(batchNorm_3)
cnn_model.add(relu_3)

# 6. Create the fourth convolutional layer and add it to the model object:

# YOUR CODE HERE

# SOLUTION:
conv_4= keras.layers.Conv2D(128, 3)
batchNorm_4 =  keras.layers.BatchNormalization()
relu_4 = keras.layers.ReLU()
cnn_model.add(conv_4)
cnn_model.add(batchNorm_4)
cnn_model.add(relu_4)

# 7. Create the pooling layer and add it to the model object:

#  YOUR CODE HERE

# SOLUTION:
pooling_layer = keras.layers.GlobalAveragePooling2D()
cnn_model.add(pooling_layer)


# 8. Create the output layer and add it to the model object:
#  YOUR CODE HERE
```

```
# SOLUTION:
output_layer = keras.layers.Dense(units=10)
cnn_model.add(output_layer)

cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 16)        160
_________________________________________________________________
batch_normalization (BatchNo (None, 26, 26, 16)        64
_________________________________________________________________
re_lu (ReLU)                 (None, 26, 26, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 32)        4640
_________________________________________________________________
batch_normalization_1 (Batch (None, 24, 24, 32)        128
_________________________________________________________________
re_lu_1 (ReLU)               (None, 24, 24, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 22, 22, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 22, 22, 64)        256
_________________________________________________________________
re_lu_2 (ReLU)               (None, 22, 22, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 20, 20, 128)       73856
_________________________________________________________________
batch_normalization_3 (Batch (None, 20, 20, 128)       512
_________________________________________________________________
re_lu_3 (ReLU)               (None, 20, 20, 128)       0
_________________________________________________________________
global_average_pooling2d (Gl (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 10)                1290
=================================================================
Total params: 99,402
Trainable params: 98,922
Non-trainable params: 480
_________________________________________________________________
```

0.5.2 Step 2. Define the Optimization Function

Task: In the code cell below, create a stochastic gradient descent optimizer using `keras.optimizers.SGD()`. Specify a learning rate of 0.1 using the `learning_rate` parameter. As-

sign the result to the variablesgd_optimizer.

[13]:
```
# sgd_optimizer = #YOUR CODE HERE


# SOLUTION
sgd_optimizer = keras.optimizers.SGD(learning_rate=0.1)
```

0.5.3 Step 3. Define the loss function

Task: In the code cell below, create a sparse categorical cross entropy loss function using `keras.losses.SparseCategoricalCrossentropy()`. This is an extension of the categorical cross entropy loss function. It is used when there are two or more label classes and the labels are integers. For more information, consult the online SparseCategoricalCrossentropy documentation. Use the parameter `from_logits=True`. Assign the result to the variable `loss_fn`.

[14]:
```
# loss_fn = #YOUR CODE HERE


# SOLUTION
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

0.5.4 Step 4. Compile the model

Task: In the code cell below, package the network architecture with the optimizer and the loss function using the `cnn_model.compile()` method. Specify the optimizer, loss function and the accuracy evaluation metric as arguments.

[15]:
```
# YOUR CODE HERE


# SOLUTION
cnn_model.compile(optimizer=sgd_optimizer, loss=loss_fn, metrics=['accuracy'])
```

0.6 Part 5. Fit the Model to the Training Data

We can now fit the CNN model to the training data. Since there are 60,000 training examples and nearly 100,000 parameters to fit, this may take a while to run. Therefore, we will only choose one epoch in this assignment.

Task: In the code cell below, fit the CNN model to the training data using the `fit()` method. Call `cnn_model.fit()` with the following arguments: 1. The training data sets. 2. The number of epochs.

Save the results to the variable `history`.

Note: This may take a while to run.

[16]:
```
num_epochs = 1 # Number of epochs

t0 = time.time() # start time
```

```
# history = # YOUR CODE HERE

# solution
history = cnn_model.fit(X_train, y_train, epochs=num_epochs)


t1 = time.time() # stop time

print('Elapsed time: %.2fs' % (t1-t0))
```

```
1875/1875 [==============================] - 169s 90ms/step - loss: 0.3453 -
accuracy: 0.9164
Elapsed time: 169.60s
```

0.7 Part 6. Evaluate the Model's Performance

Let's now evaluate our CNN model's performance on our test data and see how it did.

Task: In the code cell below, call the `cnn_model.evaluate()` method with the test data sets as arguments. The `evaluate()` method returns a list containing two values. The first value is the loss and the second value is the accuracy score.

[17]:
```
#loss, accuracy = # YOUR CODE HERE

# Solution
loss, accuracy = cnn_model.evaluate(X_test, y_test, verbose=0)

print('Loss: ', str(loss) , 'Accuracy: ', str(accuracy))
```

```
Loss:  0.23454897105693817 Accuracy:  0.9258000254631042
```

Next we'll make some predictions on the test set and see for ourselves how accurate these predictions are.

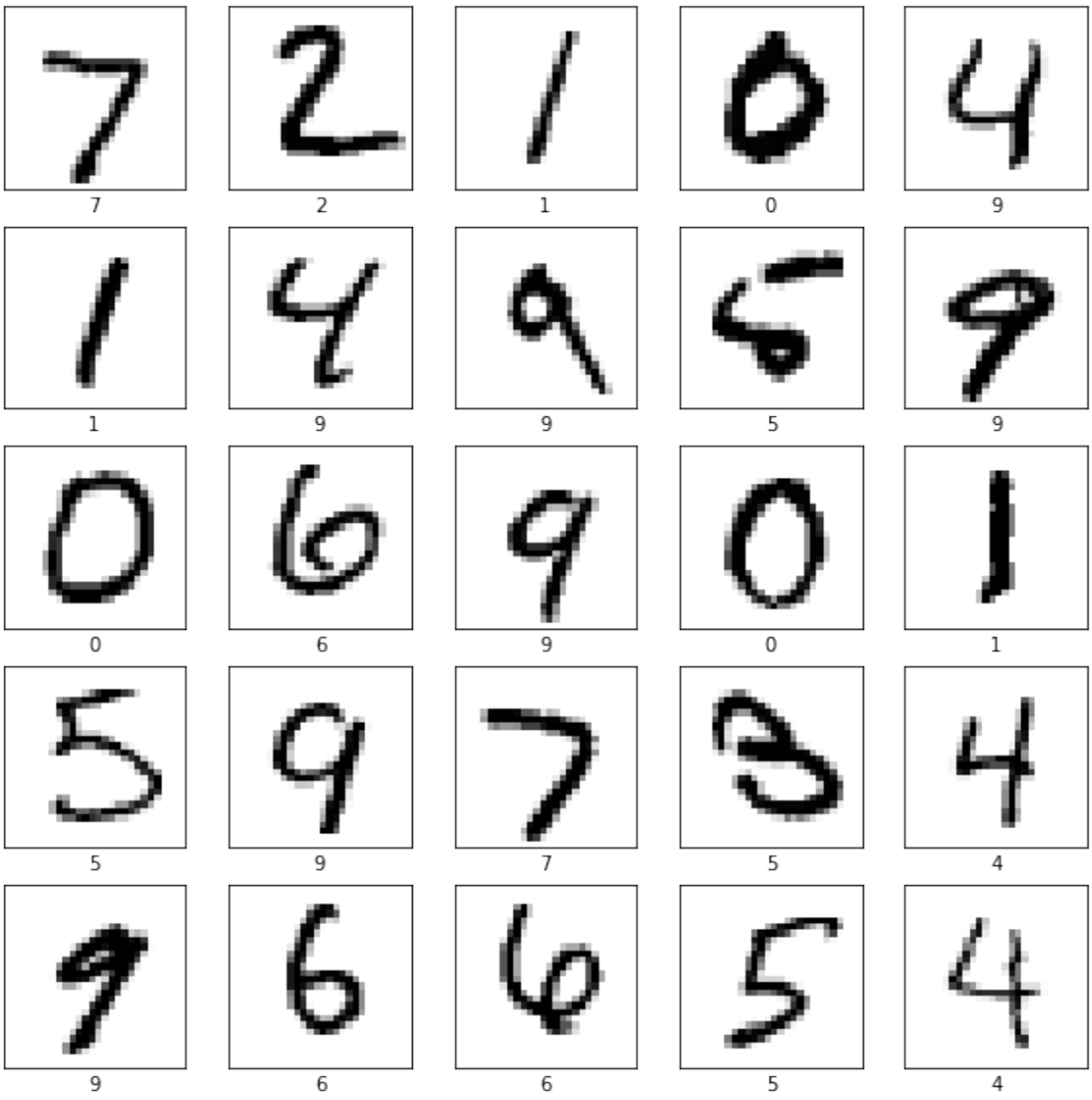
Task: In the code cell below, call the `plot_imgs()` functions with the first 25 images in `X_test` as the first argument, and the first 25 labels in `predictions` as the second argument.

The result should be a display of the first 25 images in the test set `X_test`, and below each image, a display of the predicted digit. How well did we do?

[18]:
```
# Make predictions on the test set
logits = cnn_model.predict(X_test)
predictions = logits.argmax(axis = 1)


## Plot individual predictions
# YOUR CODE HERE

# Solution
plot_imgs(X_test[:25], predictions[:25])
```


7 2 1 0 9
1 9 9 5 9
0 6 9 0 1
5 9 7 5 4
9 6 6 5 4

[ ]: