LESSON 24 Image Classification Last update June3, 2021

Conventions used in these lessons:

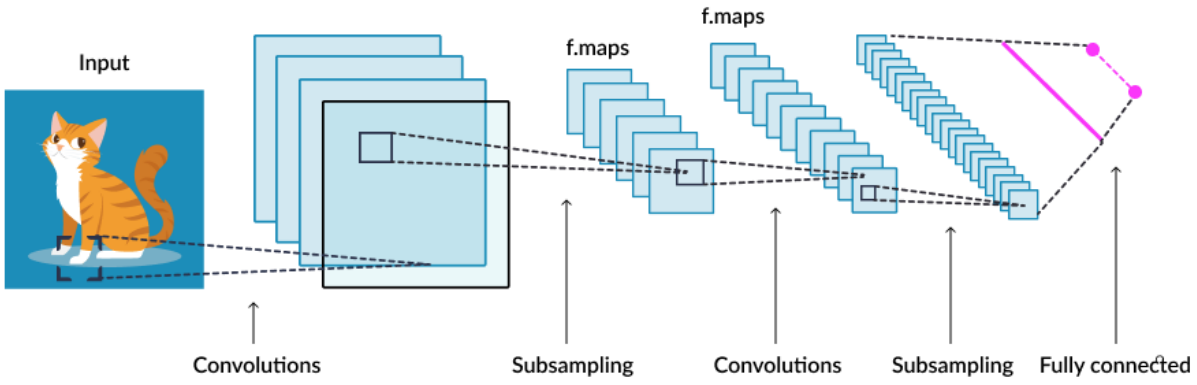
bold - headings, keywords, code
italics - code syntax
underline - important words

Image classification is used everywhere, facial recognition, hand writing analysis, postal code identification, self driving cars, identifying rooms in a house etc. Image classification uses Deep Learning. Deep learning refers to neural networks with multiple hidden layers that can learn with increasingly abstract representations of the input data.
Deep learning has led to major advances in computer vision. We're now able to classify images, find objects in them, and even label them with captions. Deep neural networks with many hidden layers can sequentially learn more complex features from the raw input image:

- The first hidden layers might only learn local edge patterns.
- Then, each subsequent layer (or filter) learns more complex representations.
- Finally, the last layer can classify the image as a cat or dog

These types of deep neural networks are called Convolutional Neural Networks (CNN).

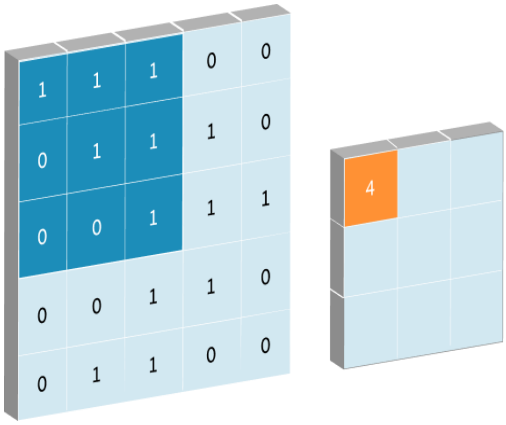
Convolutional Neural Networks (CNN's) are multi-layer neural networks (sometimes up to 17 or more layers) that assume the input data to be images.



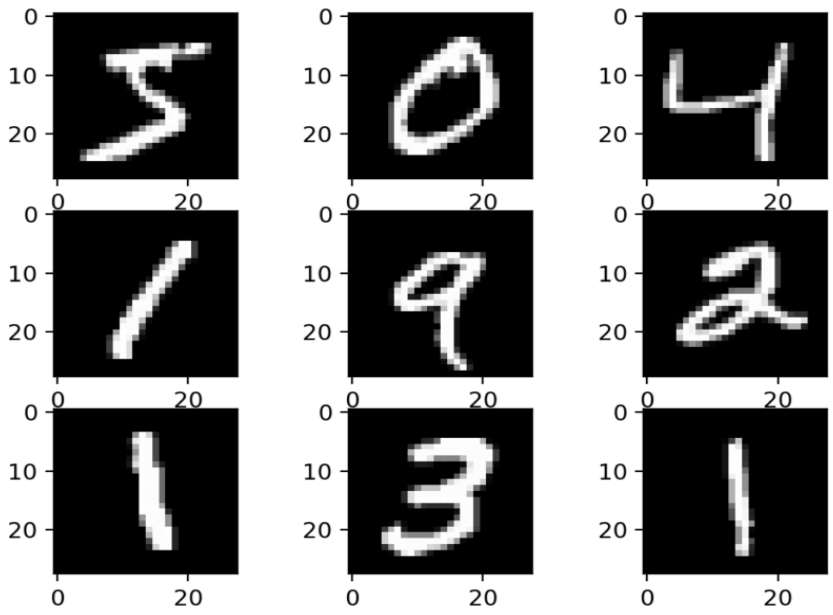
By making this requirement, CNN's can drastically reduce the number of parameters that need to be tuned. Therefore, CNN's can efficiently handle the high dimensionality of raw images.

Convolution

2D convolutional layers take a three-dimensional input, typically an image with three color channels. They pass a filter, also called a convolution kernel, over the image, inspecting a small window of pixels at a time. A kernel filter window may ne 3x3 or 5x5 or 7x7 always a odd number. The convolution operation calculates the dot product of the pixel values in the current filter window with the weights defined in the filter. The window is moved until they have scanned the entire image. The kernel weights are tuned in the training of the model to achieve the most accurate predictions. A 2D convolution layer means that the input of the convolution operation is three-dimensional, for example, a color image which has a value for each pixel across three layers: red, blue and green It is called a “2D convolution” because the movement of the filter across the image happens in two dimensions.



After the convolution ends, the features are down sampled, and then the same convolutional structure repeats again. At first, the convolution identifies features in the original image (for example in a cat, the body, legs, tail, head), then it identifies sub-features within smaller parts of the image (for example, within the head, the ears, whiskers, eyes). Eventually, this process is meant to identify the essential features that can help classify the image.
In this lesson we will classify hand written digits. We will use the MINST data set of hand written digits available from Keras.



The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset.

It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.

Out task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

Deep learning convolutional neural networks achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on this data set.

The images are all pre-aligned and each image only contains a hand-drawn digit, all images have the same square size of 28×28 pixels, and that the images are all grayscale.

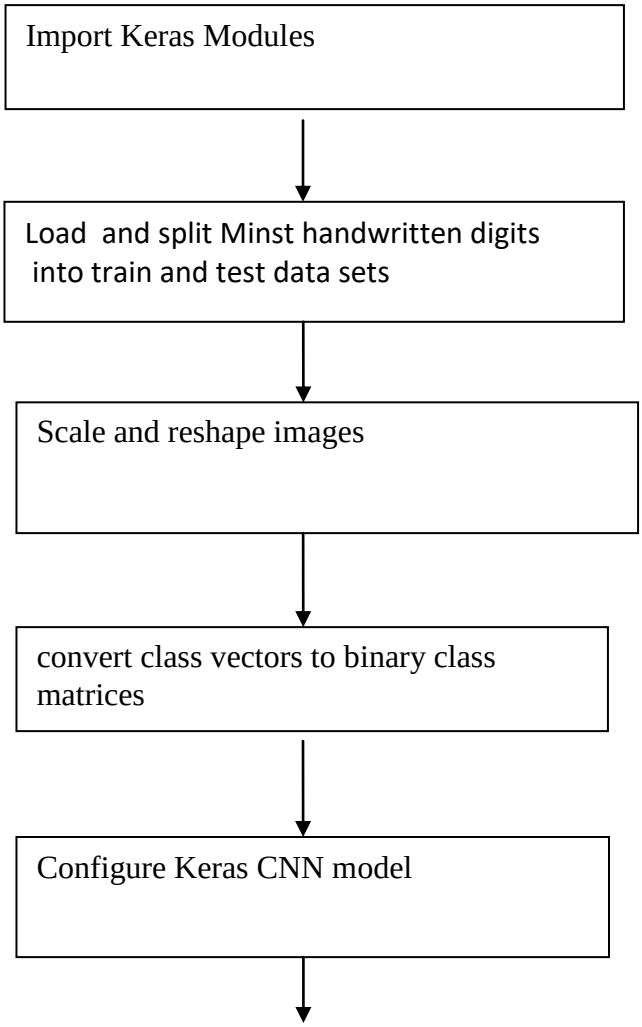
We can load the images and reshape the data arrays to have a single color channel.

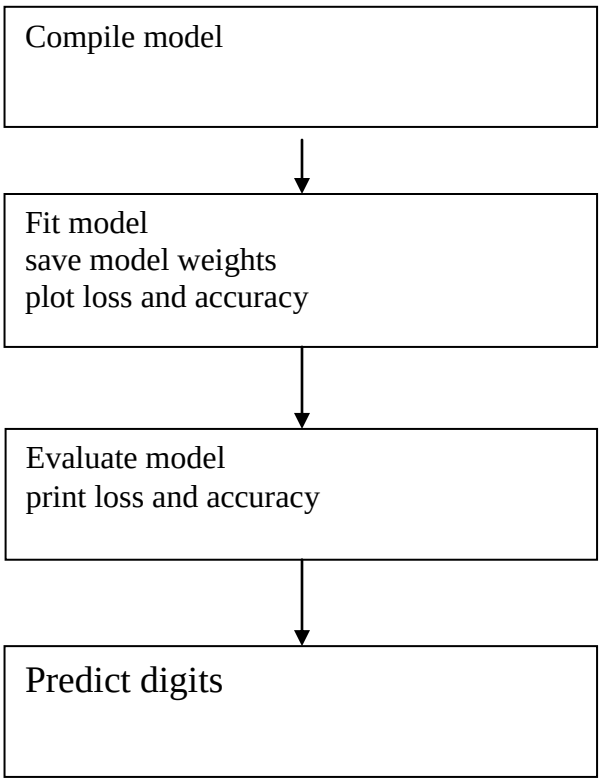
There are 10 classes digits 0 to 9 and that classes are represented as unique integers.

We use a **one hot** encoding for the class element of each sample, transforming the integer into a 10 element binary vector with a 1 for the index of the class value, and 0 values for all other classes. We can achieve this with the **to_categorical()** utility function.

We are using the Sequential model type from Keras. This is simply a linear stack of neural network layers, and it's perfect for this type of feed-forward CNN.

Our program flow is as follows:





(1) Import Keras Modules

```
import numpy as np
import keras
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from matplotlib import pyplot as plt
from os import path
```

(2) Load and split Minst handwritten digits into train and test data sets

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

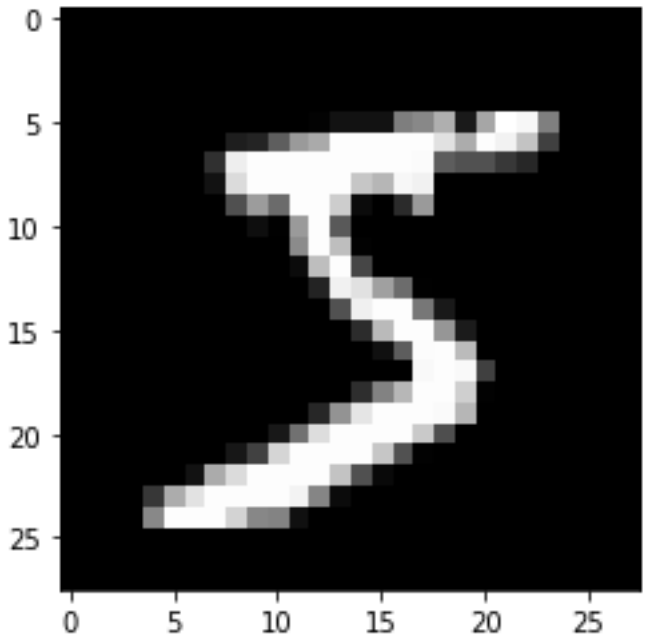
Each digit image is a 28 * 28 array. There are 60,000 training sets and 10000 test sets. The x sets are the images whereas the y sets are the digit numbers 0 to 9,

```
print(x_train[0].shape)
print("x_train: ",x_train.shape)
print("y_train: ",y_train.shape)
print("x_test: ",x_test.shape)
print("x_test: ",y_test.shape)
print("image: ",x_train[0].shape)
```

```
x_train:  (60000, 28, 28)
y_train:  (60000,)
x_test:  (10000, 28, 28)
x_test:  (10000,)
image:  (28, 28)
```

We can even plot one of your images for your curiosity, using a grey scale.

```
plt.imshow(x_train[0],cmap ='gray')
plt.show()
```



(3) Scale and reshape images

We scale the images 0 to 1 from grey scale image 0 to 255.

```
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

We add an additional axes to each image using the numpy function **np.expand_dims**

```
# convert images to shape (28, 28, 1)
# -1 means add new dimension to original shape
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# print out new shapes
print("x_train shape:", x_train.shape)
print(x_train.shape, "train samples")
```

```
x_train shape: (60000, 28, 28, 1)
x_train shape: (60000, 28, 28, 1)
```

We have basically added another dimension where each image. Each image shape is now **(28, 28, 1)** rather than **(28, 28)** and we have 60000 images**.**

We have converted **[28][28]** 2D array to **[[28][28]]** 3D array for each image

(4) convert class vectors to binary class matrices

```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

Basically we are taking the 10 digits and transforming them into a 10 element binary vector with a 1 for the index of the class value, and 0 values for all other classes. This is known as a **one hot encoding** we use the **keras to_categorical()** function to do this.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	0	0	0	0	0	0	0	0
0	**1**	0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0	0	0
0	0	0	**1**	0	0	0	0	0	0
0	0	0	0	**1**	0	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	**1**	0	0	0
0	0	0	0	0	0	0	**1**	0	0
0	0	0	0	0	0	0	0	**1**	0
0	0	0	0	0	0	0	0	0	**1**

(5) Configure Keras CNN model

We are using the Sequential model type from Keras. This is simply a linear stack of neural network layers, and it's perfect for the type of feed-forward CNN.

```
model = keras.Sequential()
```

Our input layer has shape 28,28,1 with 32 filters to be filters used in the convolution operation and a kernel filter size of 3 by 3 and activation relu. Relu means **rectified linear activation function.** The relu activation function is max(x,0) so no output value goes below , otherwise the output follows the input.

```
model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=(28,28,1)))
```

2D convolutional layers, closer to the input learn less filters, while later convolutional layers, closer to the output, learn more filters. The number of filters you select should depend on the complexity of your dataset and the depth of your neural network. A common setting to start with is [32, 64, 128] for three layers, and if there are more layers, increasing to [256, 512, 1024], etc.

Filter size may be determined by the CNN architecture you are using – for example VGGNet exclusively uses (3, 3) filters. If not, use a 5×5 or 7×7 filter to learn larger features and then quickly reduce to 3×3. If your images are smaller than 128×128, consider working with smaller filters of 1×1 and 3×3.

We also have a additional Convolution2D layer of 64 filters with kernel filter size of 3 by 3 and activation relu. This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs

```
model.add(Convolution2D(64, 3, 3, activation='relu'))
```

Convolutional layers in a convolutional neural network summarize the presence of features in an input image.

A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down sample the feature maps. This has the effect of making the resulting down sampled feature

maps more robust to changes in the position of the feature in the image, referred to by the technical phrase “local translation invariance.”

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model.

The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps.

Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels.

This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels).

The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

- **Average Pooling**: Calculate the average value for each patch on the feature map.
- **Maximum Pooling (or Max Pooling)**: Calculate the maximum value for each patch of the feature map.

The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model’s invariance to local translation.

model.add(MaxPooling2D(pool_size=(2,2)))

Dropout is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to complex co-adaptations.

You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to over fit the training data.

model.add(Dropout(0.25))

Flattening a tensor means to remove all of the dimensions except for one. This is exactly what the Flatten layer do. A flatten operation on a tensor reshapes the tensor to have the shape that is equal to the number of elements contained in tensor **non including the batch dimension**.

model.add(Flatten())

Again we drop out 0.5

model.add(Dropout(0.5))

Our output layer has 10 output neuron using softmax activation.

Softmax activation is called the logistic function. Regardless of the input, the function always outputs a value between 0 and 1. The form of the function is an S-shape between 0 and 1 with the vertical or middle of the "*S*" at 0.5.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

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

The summary method prints a string summary of the network.

model.summary()

```
Model: "sequential_32"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_59 (Conv2D)           (None, 9, 9, 32)          320
_________________________________________________________________
conv2d_60 (Conv2D)           (None, 3, 3, 64)          18496
_________________________________________________________________
max_pooling2d_28 (MaxPooling (None, 1, 1, 64)          0
_________________________________________________________________
dropout_52 (Dropout)         (None, 1, 1, 64)          0
_________________________________________________________________
flatten_26 (Flatten)         (None, 64)                0
_________________________________________________________________
dropout_53 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_26 (Dense)             (None, 10)                650
=================================================================
Total params: 19,466
Trainable params: 19,466
Non-trainable params: 0
```

(6) Compile model

The compile defines the loss function, the optimizer and the metrics.

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

The Loss function is used to find error or deviation in the learning process.
We are using the **categorical_crossentropy loss function.**

The **categorical_crossentropy** computes the cross entropy loss between the labels and predictions. The **categorical_crossentropy** expect labels to be provided in a `one hot` representation.

The optimizer is used optimize the input weights by comparing the prediction and the loss function.
There are many optimizers available we are using the Adam optimizer
metrics is a list of metrics to be evaluated by the model during training and testing.
We are interested in accuracy

(7) Fit model and save model weights plot loss and accuracy

Models are trained by using the fit function.

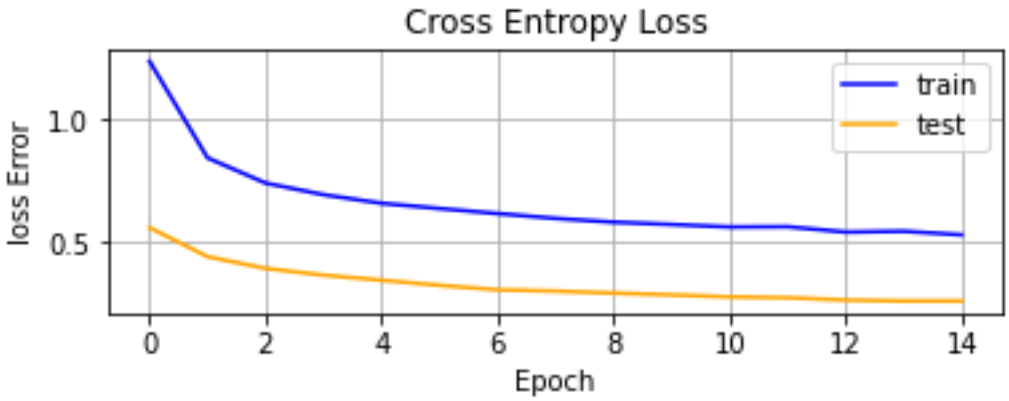
```
# check for weights file
if not path.exists("digits.h5"):

    history = model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1)
    model.save_weights('digits.h5')

    # plot loss
    plt.subplot(2, 1, 1)
    plt.title('Cross Entropy Loss')
    plt.plot(history.history['loss'], color='blue', label='train')
    plt.plot(history.history['val_loss'], color='orange', label='test')
    plt.xlabel('Epoch')
    plt.ylabel('loss Error')
    plt.legend()
    plt.grid(True)
    plt.show()
```

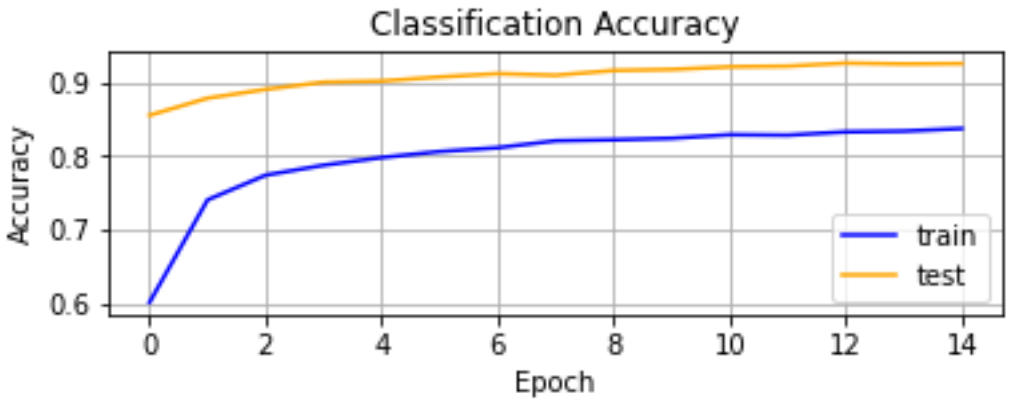
```
    # plot accuracy
    plt.subplot(2, 1, 2)
    plt.title('Classification Accuracy')
    plt.plot(history.history['accuracy'], color='blue', label='train')
    plt.plot(history.history['val_accuracy'], color='orange', label='test')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)

    plt.show()
```

```
else:
   model.load_weights('digits.h5')
```

- **X, y** – is the training data where X are the inputs and y is the expected output.
- **epochs** − no of times the model is needed to be evaluated during training.
- **batch_size** − training instances.
- validation_split: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the x and y data provided, before shuffling.

we save the node weights in a file called digits.h5. so we do not have to refit the data every time we run the program





(7) Evaluate model print loss and accuracy

Returns the loss value & metrics values for the model in test mode.

```
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.2815539538860321
Test accuracy: 0.9193999767303467
```

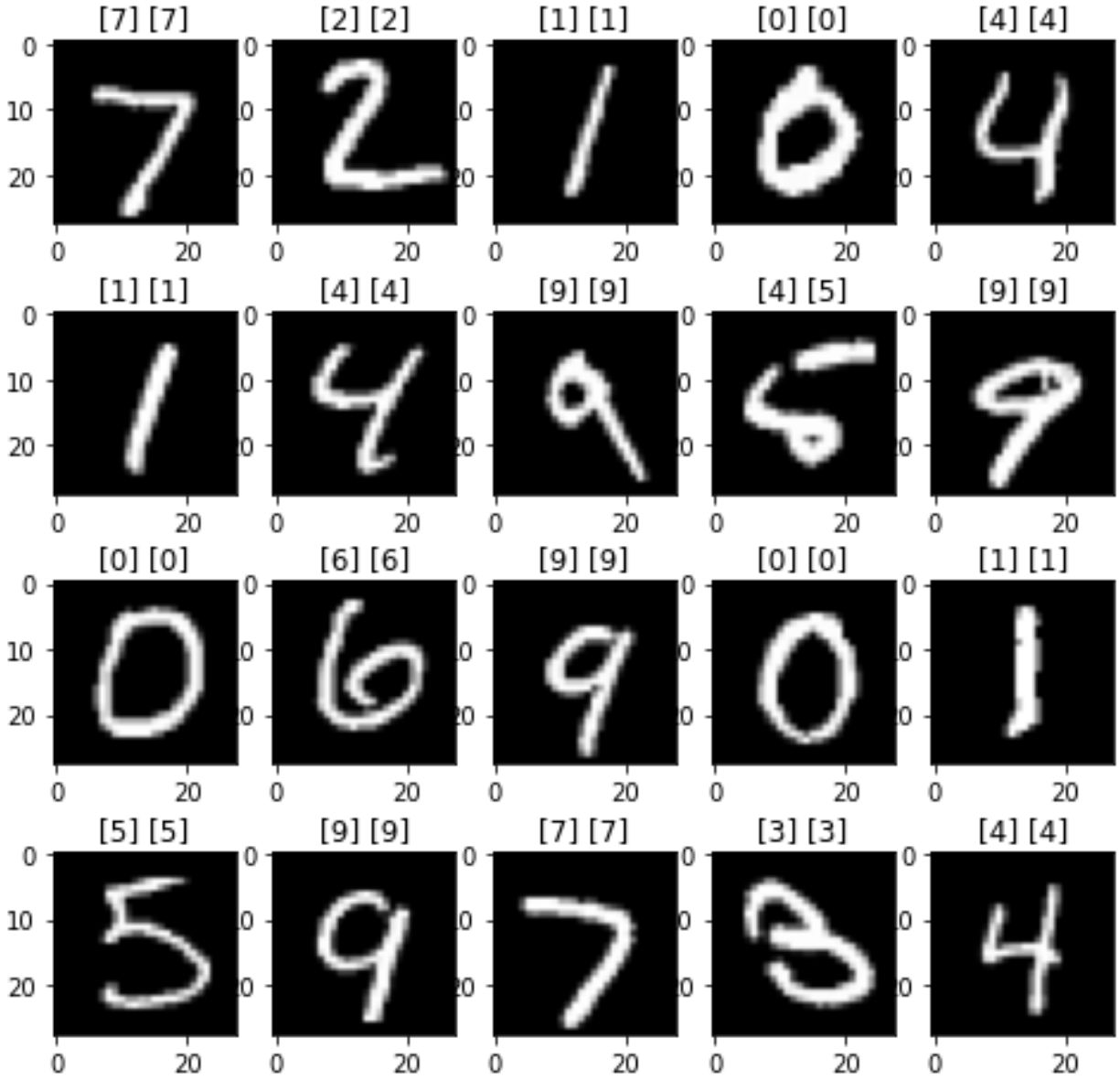
(8) Predict digits

Test some images and display the results.

```
# print predictions
fig=plt.figure(figsize=(8, 8))

# for all predictions
for i in range(1,21):

   print("train shape:",x_test[i-1].shape)

   # restore image from (28 28 1) to (28,28)
   test_image = np.squeeze(x_test[i-1])
   print("test image shape: ",test_image.shape)

   #plt.imshow(test_image)
   #plt.show()
```

```
    # put test image in a in an array
    # (1, 28, 28, 1)
    test_input= np.array([x_test[i-1]])
    print("test image input: ",test_input.shape)

    # get result vector
    result = model.predict(test_input)
    print(result)

    # get digit
    digit = model.predict_classes(test_input)
    print(digit)

    # plot digit
    ax=fig.add_subplot(4,5,i)
    plt.imshow(test_image , cmap ='gray')

    # get expected digit
    expected = np.where(y_test[i-1]==1)
    print(expected[0])

    # print digit and expected digit
    ax.title.set_text(str(digit) + " " +str(expected[0]))

plt.show()
```

Here is the output digit classifications:



Here some text output:

predictions
train shape: (28, 28, 1)
test image shape: (28, 28)
test image input: (1, 28, 28, 1)
[[2.0969494e-03 6.1242071e-07 1.7464131e-02 8.2260283e-04 1.5419604e-06
3.8172093e-05 5.8084720e-07 9.7914141e-01 1.2636275e-04 3.0757807e-04]]
predicted digit: [7]
expected: [7]
train shape: (28, 28, 1)

```
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[1.19813420e-01 3.20754748e-06 8.40809643e-01 1.40412981e-02
  1.02095444e-04 8.41708574e-03 9.47131682e-03 1.87996989e-07
  7.34133366e-03 3.25709721e-07]]
predicted digit:  [2]
expected: [2]
train shape: (28, 28, 1)
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[1.20701198e-03 9.62696254e-01 1.13644153e-02 1.10921734e-04
  8.02435796e-04 2.38432203e-05 2.06528697e-02 3.09327617e-03
  3.68627443e-05 1.19965134e-05]]
predicted digit:  [1]
expected: [1]
train shape: (28, 28, 1)
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[9.9977976e-01 6.6347303e-11 1.3765336e-06 8.7874568e-08 1.0106389e-08
  2.4050081e-05 1.1705151e-04 7.7566612e-05 3.3977024e-08 4.0155015e-08]]
predicted digit:  [0]
expected: [0]
train shape: (28, 28, 1)
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[1.57036920e-04 2.82374355e-08 3.34225275e-04 1.90365824e-06
  9.66693461e-01 1.05879386e-04 4.00955975e-03 4.48661798e-04
  8.01882183e-04 2.74474584e-02]]
predicted digit:  [4]
expected: [4]
train shape: (28, 28, 1)
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[2.6677575e-04 9.9634749e-01 1.6227933e-03 3.4617657e-05 8.2334976e-05
  6.5361032e-06 1.0028239e-03 6.2007591e-04 1.1812633e-05 4.8475431e-06]]
predicted digit:  [1]
expected: [1]
train shape: (28, 28, 1)
test image shape:  (28, 28)
test image input:  (1, 28, 28, 1)
[[4.84060351e-04 1.11178655e-04 4.84405132e-03 6.03228866e-04
  9.62571323e-01 5.54710208e-03 2.27452861e-03 3.06898449e-03
  2.22955691e-03 1.82659049e-02]]
```

Here is the complete program:

```
"""
cnn_digits.py
"""
import numpy as np
import keras
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from matplotlib import pyplot as plt
from os import path

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# print out train and test shapes
print("x_train: ",x_train.shape)
print("y_train: ",y_train.shape)
print("x_test: ",x_test.shape)
print("x_test: ",y_test.shape)
print("image: ",x_train[0].shape)

# show a digit
plt.imshow(x_train[0],cmap ='gray')
plt.show()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# swt image shape to (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print("x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

```
# define model
model = keras.Sequential()

model.add(Convolution2D(32, (3, 3), activation='relu', input_shape=(28,28,1)))
model.add(Convolution2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
#model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# print mode summary
model.summary()

# compile model
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

# check for weights file
if not path.exists("cnn_digits.h5"):

   # fit model (train model)
   history = model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1)

   # save weights
   model.save_weights('cnn_digits.h5')

   # plot loss
   plt.subplot(2, 1, 1)
   plt.title('Cross Entropy Loss')
   plt.plot(history.history['loss'], color='blue', label='train')
   plt.plot(history.history['val_loss'], color='orange', label='test')
   plt.xlabel('Epoch')
   plt.ylabel('loss Error')
   plt.legend()
   plt.grid(True)
   plt.show()

   # plot accuracy
   plt.subplot(2, 1, 2)
   plt.title('Classification Accuracy')
   plt.plot(history.history['accuracy'], color='blue', label='train')
```

```
    plt.plot(history.history['val_accuracy'], color='orange', label='test')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

# load weights
else:
    model.load_weights('cnn_digits.h5')

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

# predict

# print predictions
print("predictions")

fig=plt.figure(figsize=(8, 8))

# for all predictions
for i in range(1,21):

    # print train shape
    print("train shape:",x_test[i-1].shape)

    # restore image from (28 28 1) to (28,28)
    test_image = np.squeeze(x_test[i-1])
    print("test image shape: ",test_image.shape)

    # put test image in a in an array
    # (1, 28, 28, 1)
    test_input= np.array([x_test[i-1]])
    print("test image input: ",test_input.shape)

    # get result vector
    result = model.predict(test_input)
    print(result)
```

```
    # get digit
    digit = model.predict_classes(test_input)
    print("predicted digit: ",digit)

    # plot digit
    ax=fig.add_subplot(4,5,i)
    plt.imshow(test_image , cmap ='gray')

    # get expected digit
    expected = np.where(y_test[i-1]==1)
    print("expected:",expected[0])

    # print digit and expected digit
    ax.title.set_text(str(digit) + " " +str(expected[0]))

plt.show()
```

todo: Type in or copy and paste in to a file called cnn_digits.py and run it.
Note: This program may not work with a theano back end completely.

IMAGE CLASSIFICATION HOMEWORK Question 1

Using a drawing program and make some numbers 0 to 9 and some letters with a black background, and see if you can predict them. Save you image as a png file. Test you digits in a program. Plot out your image and the test prediction. Call your python program classification_homework.py

You can use the following code to load in the image

```
# load the image
img = load_img(filename, grayscale=True, target_size=(28, 28))

# convert to array
img = img_to_array(img)

# reshape into a single sample with 1 channel
img = img.reshape(1, 28, 28, 1)

# prepare pixel data
img = img.astype('float32')
img = img / 255.0
```

END