Digit Classification Using Convolutional Neural Net

Kanishk Kumar

July 21, 2022

Table of Contents

Objective

This project will be focused on **classification** using a convolutional neural net model. The main objectives of this project are as follows:

- To apply data preprocessing and preparation techniques in order to obtain clean and crunchable data.
- To build at least three variations of a Convolutional Neural Net model with patterns of different types and numbers of layers to improve its performance in classifying digits between 0 to 9 using image data.
- To analyze and compare performance of each model variation in order to choose the best hyper-parameters and layer patterns.

Data Description

We will be using MNIST database of handwritten digits which has a training set of 60,000 28x28 grayscale images of the 10 digits, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

```python
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

```python
print(x_train.shape, y_train.shape,
      x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
```

Preprocessing Data

First, we import the necessary libraries:

```python
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import SGD
from tensorflow.keras import backend as K
import matplotlib.pyplot as plt
%matplotlib inline
```

When dealing with images & convolutions, it is paramount to handle `image_data_format` properly:

```python
img_rows, img_cols = 28, 28

if K.image_data_format() == 'channels_first':
    shape_ord = (1, img_rows, img_cols)
else:  # channel_last
    shape_ord = (img_rows, img_cols, 1)
```

We'll preprocess by scaling image pixels to be between 0 and 1:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

```
print(x_train.shape, y_train.shape,
      x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
```

Then convert the classes to its binary categorical form. Since there are 10 digits in total. Our test set now has (10000, 10) dimensions.

```
nb_classes = 10
y_train = to_categorical(y_train, nb_classes)
y_test = to_categorical(y_test, nb_classes)
```

```
print(x_train.shape, y_train.shape,
      x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000, 10) (10000, 28, 28) (10000, 10)
```

A Simple CNN

Now we will initialize the values for our convolutional neural network model. In our first run, we'll go through a hundred epochs (`nb_epoch`). Batch size (`batch_size`) is 64, I'll use 32 convolutional filters (`nb_filters`) and size of pooling areas for max pooling (`nb_pool`) is 2, just to see how it goes. Convolutional kernel size (`nb_conv`) is 3 here.

```
nb_epoch = 100
batch_size = 64
nb_filters = 32
nb_pool = 2
nb_conv = 3
sgd = SGD()
```

I'm using vanilla settings for our Gradient descent (with momentum) optimizer `SGD()` because I feel like it already has the optimum settings for this type of dataset. So, `learning_rate` would be set to default, i.e., 0.01.

Now, let's define our model. We are using Keras's Sequential model with plain stack of layers. The very first layer (Conv2D here) must always specify the `input_shape`, which we defined during the preprocessing. In this case, it is a 28x28 image with three color channels.

In the following layers I've used the `'sigmoid'` activation function because I've studied it extensively and want to see how it performs here.

```
model = Sequential()
model.add(Conv2D(nb_filters, (nb_conv, nb_conv),
                 padding='valid', input_shape=shape_ord))
model.add(Activation('sigmoid'))

model.add(Flatten())
model.add(Dense(nb_classes))
model.add(Activation('sigmoid'))
```

Now let's compile and run our model:

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

```
run_hist = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=nb_epoch, verbose=1,
                     validation_data=(x_test, y_test))
```

As can be seen, our model also validated data during the run and noted down some metrics:

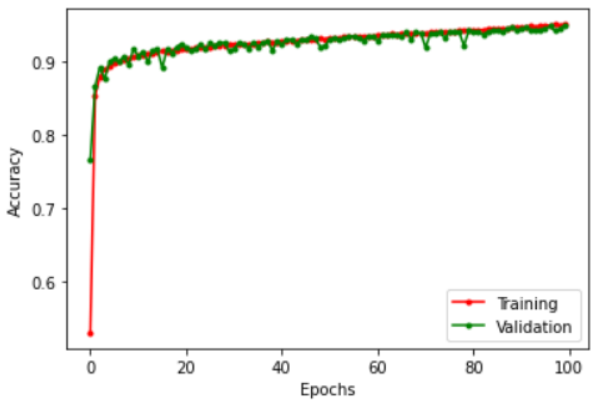
```
run_hist.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Let's plot the training accuracy and the validation accuracy over the different epochs and see how it looks.

```
fig, ax = plt.subplots()
ax.plot(run_hist.history["accuracy"],'r',
        marker='.', label="Training")
ax.plot(run_hist.history["val_accuracy"],'g',
        marker='.', label="Validation")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x26fa9e414c0>
```



```
print('Available Metrics in Model: {}'.format(model.metrics_names))
```

```
Available Metrics in Model: ['loss', 'accuracy']
```

```
# Evaluating the model on the test data
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)
```

```
Test Loss: 0.18352389335632324
Test Accuracy: 0.9498000144958496
```

We achieved a test accuracy of 95% and Loss of 0.18%. Not bad for a first run.

Adding More Dense Layers

Let's change some hyperparameters in our model and add some dense layers. I've kept the first layer as it is with `padding = 'valid'` so that the input image gets fully covered by the filter.

In following layers I've changed the activation functions to `'relu'` because of the vanishing gradient issue with `'sigmoid'` in this particular case. I also used `'softmax'` because it is supposed to work good with this type of training dataset with 28x28 images.

Let's compile and run our model.

```
model = Sequential()
model.add(Conv2D(nb_filters, (nb_conv, nb_conv),
                 padding='valid',
                 input_shape=shape_ord))
model.add(Activation('relu'))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

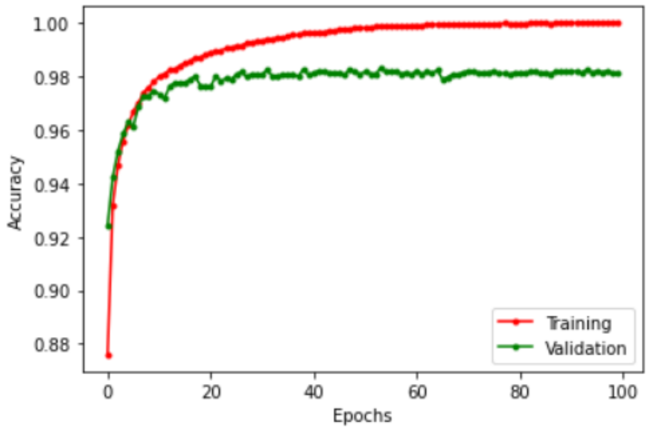
```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

run_hist = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=nb_epoch,verbose=1,
                     validation_data=(x_test, y_test))
```

Now, let's plot the training accuracy and the validation accuracy over the different epochs and see how it looks.

```
fig, ax = plt.subplots()
ax.plot(run_hist.history["accuracy"],'r',
        marker='.', label="Training")
ax.plot(run_hist.history["val_accuracy"],'g',
        marker='.', label="Validation")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x270861095e0>
```



As can be seen in our evaluation, we achieved a test accuracy of 98% and Loss of 0.08%. Significantly better than our first run.

```
#Evaluating the model on the test data
score, accuracy = model.evaluate(x_test,  y_test, verbose=0)
print('Test score:', score)
print('Test accuracy:', accuracy)
```

```
Test score: 0.08079393953084946
Test accuracy: 0.9812999963760376
```

Adding A Dropout Layer

Let’s add a dropout layer with 0.5 rate to prevent over fitting, compile and run the model:

```
model = Sequential()

model.add(Conv2D(nb_filters, (nb_conv, nb_conv),
                        padding='valid',
                        input_shape=shape_ord))
model.add(Activation('relu'))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

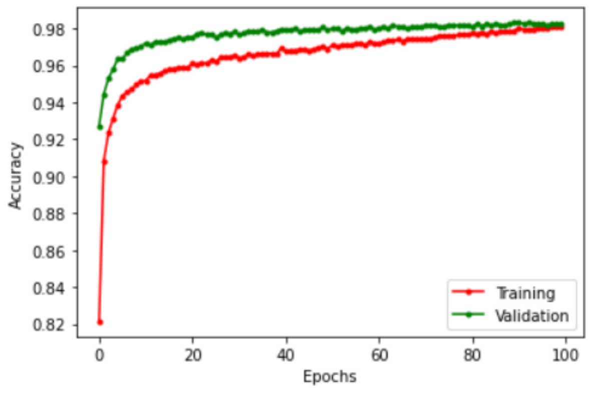
```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

run_hist = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=nb_epoch,verbose=1,
                     validation_data=(x_test, y_test))
```

Let's plot the training accuracy and the validation accuracy over the different epochs and see how it looks.

```
fig, ax = plt.subplots()
ax.plot(run_hist.history["accuracy"],'r',
        marker='.', label="Training")
ax.plot(run_hist.history["val_accuracy"],'g',
        marker='.', label="Validation")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x270922ec910>
```



As can be seen in our evaluation, we achieved a test accuracy of 98.2% and Loss of 0.05%. Just a little bit better than our last run.

```
#Evaluating the model on the test data
score, accuracy = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score)
print('Test accuracy:', accuracy)
```

```
Test score: 0.0594780333340168
Test accuracy: 0.9825999736785889
```

Adding More Convolutional Layers

Let’s add more convolutional layers to our model. Since this convolutional layer has the padding set to `‘valid’`, its output width and height will not remain the same, and the number of output channel will not be equal to the number of filters learned by the layer, i.e., 16.

These convolutional layers, instead, have the default padding, and therefore reduce width and height by (k−1), where k is the size of the kernel. `MaxPooling2D` layers, instead, reduce width and height of the input tensor, but keep the same number of channels.

Activation layers, of course, don't change the shape.

```
model = Sequential()
model.add(Conv2D(nb_filters,
                 (nb_conv, nb_conv),
                 padding='valid',
                 input_shape=shape_ord))
model.add(Activation('relu'))
model.add(Conv2D(nb_filters,
                 (nb_conv, nb_conv)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(nb_pool,
                                  nb_pool)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

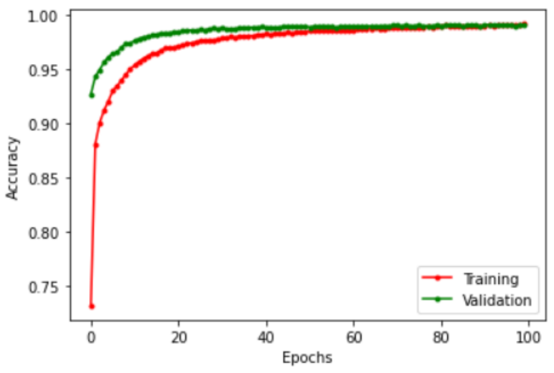
```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

run_hist = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=nb_epoch,verbose=1,
                     validation_data=(x_test, y_test))
```

Let's plot the training accuracy and the validation accuracy over the different epochs and see how it looks.

```
fig, ax = plt.subplots()
ax.plot(run_hist.history["accuracy"],'r',
        marker='.', label="Training")
ax.plot(run_hist.history["val_accuracy"],'g',
        marker='.', label="Validation")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x26fce23ae20>
```



As can be seen in our evaluation, we achieved a test accuracy of **99%** and Loss of 0.02%. With my current understanding, I think this is as good as I can get.

```
#Evaluating the model on the test data
score, accuracy = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score)
print('Test accuracy:', accuracy)
```

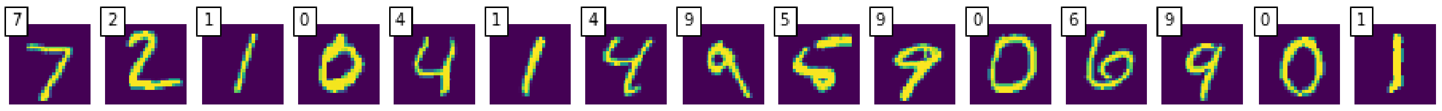
```
Test score: 0.028617434203624725
Test accuracy: 0.9911999702453613
```

Let’s plot our model’s classification and see if it really works:

```
slice = 15
predicted = model.predict(x_test[:slice]).argmax(-1)

plt.figure(figsize=(16,8))
for i in range(slice):
    plt.subplot(1, slice, i+1)
    plt.imshow(x_test[i], interpolation='nearest')
    plt.text(0, 0, predicted[i], color='black',
             bbox=dict(facecolor='white', alpha=1))
    plt.axis('off')
```

10 out of 10. Perfect outcome:



Key Findings

After running many variations of this model, I can clearly say that ‘Sigmoid’ activation function is clearly not suited here. Vanilla settings for our SGD optimizer works nearly perfectly. Our models are converging at around ~40 epochs but to get to 99% accuracy we need at least 100 epochs which is quite GPU intensive.

Advanced Steps

I don’t think our final model needs any further improvements unless there is some change in the dataset features. It’d be interesting to run these models with more data-points, and I’m eying for the original (NIST) version of this dataset which is quite larger than this, it’d be interesting to see how this model performs on that. I’d also like to see ‘Leaky ReLU’ activation with this many epochs to see if the accuracy can somehow increase.