GR5293/GU4293 Topics in Modern Statistics: Group Project Report

Team members:

TaeYoung Choi tc2777
Aakanksha Joshi aj2771
Xiaohui Guo xg2225

Detailed explanation of how CNN works

A convolutional neural network is a class of deep, feed-forward artificial neural networks which consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers. The components of a neural network are as follows:

1. Weights: CNNs share weights in convolutional layers, which means that the same filter (weights bank) is used for each receptive field in the layer; this reduces memory footprint and improves performance.

2. Fully-Connected Layers: Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network.

3. Pooling: Convolutional networks may include local or global pooling layers, which combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For example, *max pooling* uses the maximum value from each of a cluster of neurons at the prior layer. Another example is *average pooling*, which uses the average value from each of a cluster of neurons at the prior layer.

(Source: https://en.wikipedia.org/wiki/Convolutional_neural_network)

4. Dropout: Dropout is a technique that prevent the network from overfitting by turning off a proportion of units. Dropout will randomly choose units according to given proportion each epoch, then training the remaining units that did get dropped out.

5. Batch Normalization: After applying non-linearity to activations, it is helpful in practice to normalize these outputs to univariate normal. We define a mini-batch to learn the batch mean and variance and normalize each mini-batch. Therefore, we have extra parameters the learn. However, this technique can reduce internal covariate shift in the network training steps.

Explanation of each and every parameter/hyperparameter of CNN with its function

1. Conv2D: the first parameter determines the number of layers after applying convolution. The kernel_size is the size of the region from the previous step that we apply convolution to. For example, if we apply a 5 by 5 kernel to a 28 by 28 image, we can find 24 by 24 different sub-regions. Therefore, the dimension of the convoluted layer becomes 24 by 24. Activation parameter sets the activation function. This non-linear activation function allows us to create a non-linear decision boundaries. 'Relu' is a piecewise linear function, which is 0 for $x < 0$ and identity for all x greater than or equal to 0. 'Softmax' is used for classification, which returns probabilities for different classes. Input_shape is necessary for the first hidden layer, because the network does not know the shape of our input. However, the subsequent layers can infer how many parameters to initialize by looking at their previous layer.

2. Dropout: This sets the proportion of units in hidden layers to turn off while training. The units are selected randomly and we drop off a different set of units for each epoch. As explained above, dropout reduces the change of overfitting.

3. MaxPooling2D: Pool_size definise the size of the region that we are extracting maximum from. Unlike convolution, the regions cannot intersect. As a result, a 3 by 3 pool_size reduce the layer size by ⅓.

4. Compile: The loss function we are trying to minimize is cross-entropy, because we are fitting a classification model. We could also use squared loss for regression. The optimizer sets the methodology for backpropagation in training. Different optimizers employ different algorithms, but they all try to efficiently update weights between units each epoch they run. Metrics decides which evaluation function to use to judge the performance of the network. This is a classification model and using categorical accuracy is appropriate here.

5. Fit: batch_size decides the batch size of inputs. If we set the batch size to 128. Then a batch of 128 images is used to train the network together and we go through each batch subsequently. The complete run through of all the batches define a epoch. We can run this process for multiple times and epochs sets the number of iterations. Verbose controls the text output from training process and validation_split sets the proportion from training data that we keep separately for validation. For our project, we tried both ⅙ and 0.1.

Detailed explanation of your code logic

For our best model, we get accuracy of 0.9957. Here is the detailed explanation of our code logic:

```
import keras                                    # import keras
from keras.datasets import mnist                # import mnist dataset
from keras.models import Sequential             # import Sequetial model
from keras.layers import Dense, Dropout, Flatten, BatchNormalization     # import these layers
from keras.layers import Conv2D, MaxPooling2D      # import layers
from keras import backend as K                  # import backend

batch_size = 128       # set batch_size to be 128
num_classes = 10       # set number of classes to be 10
epochs = 25            # set epochs to be 25

# input image dimensions
img_rows, img_cols = 28, 28    # The input image dimensions are 28 by 28

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

if K.image_data_format() == 'channels_first':
   x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
   x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
   input_shape = (1, img_rows, img_cols)

# if  K.image_data_format() is "channels_first", reshape x_train data to have the dimensions:
#number of rows of x_train, 1, img_rows which iw 28, img_cols which is 28
# reshape x_test data to have the dimensions:
# number of rows of x_test, 1, 28,28 . here 28 is img_rows and also img_cols,

else:
   x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
   x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
   input_shape = (img_rows, img_cols, 1)

# If  K.image_data_format() is not "channels_first", reshape x_train to have the dimensions:
# number of rows of x_train, 28,28,1
# reshape x_test to have the dimensions:
# number of rows of x_test, 28,28,1.
# here 28 is the  img_rows and img_cols we defined above.


x_train = x_train.astype('float32')      # reset the data type of x_train to be 'float32'
x_test = x_test.astype('float32')        # reset the test data type pf x_test to be 'float32'
x_train /= 255    # Divides x_train with 255 and assign the result to x_train
x_test /= 255     # Divides x_test with 255 and assign the result to x_test
print('x_train shape:', x_train.shape)    # print x_train shape
print(x_train.shape[0], 'train samples') # print x_train row numbers
print(x_test.shape[0], 'test samples')  # print x_test row numbers

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

model = Sequential()     # set up the basic model " Sequential".
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', input_shape=input_shape))
# add Conv2D layer : Set filter to be 32, which means the number of output filters in the convolution
# is 32. Set kernel size to be 5 by 5, which is the width and height of the 2D convolution window.
# Set the activation to be ‘relu’ and set the input_shape to be the shape we defined just now.

model.add(BatchNormalization())
# add a batch normalization layer

model.add(Conv2D(64, (5, 5), activation='relu'))
# Add a Conv2D layer with filter of 64 and kernel size of 5 by 5. Use the activation ‘relu’.

model.add(BatchNormalization())
# Add a batch normalization layer

model.add(Conv2D(64, (5, 5), activation='relu'))
# Add a Conv2D layer with filter of 64 and kernel size of 5 by 5. Use the activation ‘relu’.

model.add(BatchNormalization())
# Add a batch normalization layer

model.add(MaxPooling2D(pool_size=(3, 3)))
```

```
# add a maxpooling2D layer with pool size of 3 by 3.

model.add(Dropout(0.25))
# add a dropout layer with the rate of 0.25, which means we set 25% of the input units to drop.

model.add(Flatten())
# add a flatten layer

model.add(Dense(256, activation='relu'))
# add a dense layer with dimensionality of the output space to be 256 and use the activation to be ‘relu’.

model.add(Dropout(0.5))
# add a dropout layer with the rate of 0.5, which means we set 50% of the input units to drop.

model.add(Dense(256, activation='relu'))
# add a dense layer with dimensionality of the output space to be 256 and use the activation to be ‘relu’.

model.add(Dropout(0.5))
# add a dropout layer with the rate of 0.5, which means we set 50% of the input units to drop.

model.add(Dense(num_classes, activation='softmax'))
# add a dense layer with dimensionality of the output space to be number of class we set earlier,  and use the activation to be ‘softmax’.

model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(), metrics=['accuracy'])
# compile the model

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_split=0.1)
# fit the model

score = model.evaluate(x_test, y_test, verbose=1)
# calculate the prediction accuracy score when test the model on test dataset.

print('Test loss:', score[0])
print('Test accuracy:', score[1])

# print test loss and test accuracy.
```

<u>Mention your starting model and how you chose that</u>

Starting Model:

https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
Starting model is from the above website.

Using the specified code after fixing the bug which was that the accuracy of the model was being tested on the test set itself instead of a hold-out set within the training, which would have caused the model to not generalize well.

```
Original : model.fit (x_train,
                       y_train,
                      batch_size=batch_size,
                      epochs=epochs,
                      verbose=1,
                     data=(x_test, y_test))
```

```
Optimization: model.fit (x_train,
                          y_train,
                          batch_size=batch_size,
                          epochs=epochs,
                          verbose=1,
                          validation_split=⅙)  # fixed the bug
```

The original code was validating each epoch on test data. We can double check that by look at the sample output. After the run the validation accuracy is exactly same as the test accuracy. We should not use test data until we finalize our model. There was information leakage in the original model. For our model, we decide to keep 10,000 samples separately from training set as validation set, which is the ⅙ of training dataset.

Accuracy: 99.11%

<u>Explain how you reached final model from starting model with intermediate stages as well</u>

Intermediate Models:

<u>Changed parameters in the original code:</u>

1. epoch = 15, kernel size = (5,5), batch_size = 256, Dense = 512, PoolSize = (3,3), ***Accuracy: 0.9936***

2. epoch=15, kernelsize=(5,5), batch_size=128, + batchnormalizer, ***Accuracy: 0.9938***

3. epoch = 15, kernel size = (5,5), batch_size = 256, Dense = 256 ***Accuracy: 0.994***

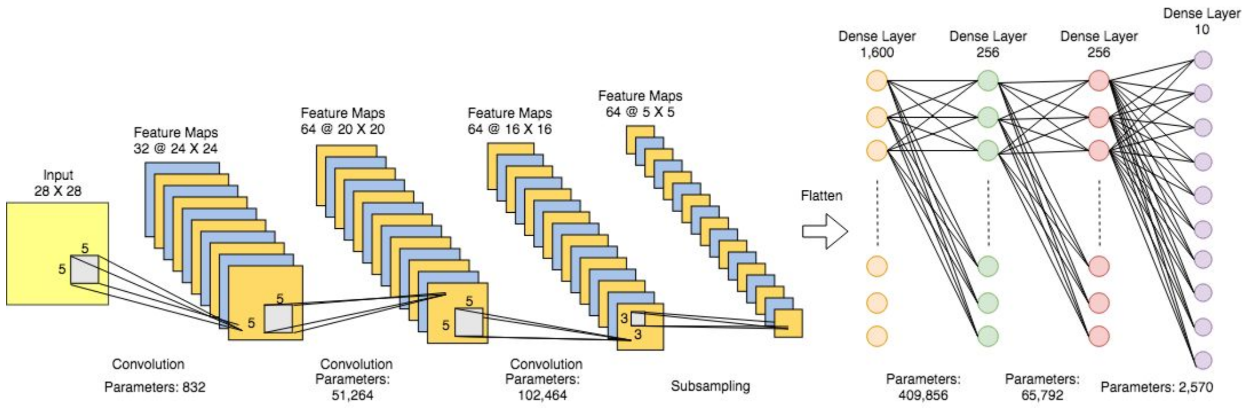
<u>Added additional layers:</u>

Epoch = 15, kernel size = (5,5), batch_size = 128

```
model.add(BatchNormalization())
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Accuracy: 0.9949

Final Model:



Epoch = 25, kernel size = (5,5), batch_size = 128

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
metrics=['accuracy'])
```

Accuracy: 0.9957

<u>What are the difficulties faced while implementing the model?</u>

- The code can take considerable amount of time to run on CPU. However it is not easy to set up GPU environment on google cloud. In order to train the model faster and save time, we spent a whole day trying to set up GPU environment on google cloud. However, we met a lot of problems, such as "ssh: connect to host xxxxx port 22: Connection timed out', 'failed to install keras on google cloud GPU". It cost us a lot of time by googling methods of set up GPU and install all the required packages in the GPU environment.

- Many possible combinations of parameters. We would not be able t test all the combination of parameters since for each epoch, it cost more than 20 minutes. It is really time consuming when tuned the parameters.

- Intuition and previous experience may not help here. Such as based on the experience in other project, the higher the value of epochs is, the better accuracy we will get. But for this case, when we set epochs to be 25, we get test accuracy 0.9957. However, when we set epochs to be 100, the test accuracy is 0.9956. The intuition and previous experience doesn't help here.

<u>How can you improve the model further?</u>

- Add more layers and compare the test accuracy results.
  Such as add the layers
  ```
  model.add(Dense(256, activation='relu'))
  model.add(Dropout(0.5))
  ```

- Tune the parameters of the new layers already added.