Assignment 3
Keras CIFAR-10 Image Classification

CS 519 - Deep Learning
Eugene Seo (OSU ID: 932981978)

Problem 1. Remove the dropout layer after the fully-connected (FC) layer (10 points). Save the model after training.

Here is the modified CNN model after removing the last dropout layer.

- CNN Architecture (Layer Type: Output Shape)

INPUT: 32 32 3
AVGPOOL: 16 16 3
CONV/RELU: 16 16 32
CONV/RELU: 14 14 32
MAXPOOL: 7 7 32
DROPOUT: 7 7 32

CONV/RELU:: 7 7 64
CONV/RELU:: 5 5 64
MAXPOOL: 2 2 64
DROPOUT: 2 2 64

FLATTEN: 256
DENSE/RELU: 512
DENSE/SOFTMAX: 10

Algorithm 1 : Remove the last dropout layer

```
def basicCNNModel(input_shape, nb_classes):
  model = Sequential()
  model.add(AveragePooling2D(pool_size=(2,2), input_shape=input_shape))
  model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same', name='conv1_1
  model.add(Convolution2D(32, 3, 3, activation='relu', name='conv1_2'))
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool1_1'))
  model.add(Dropout(0.25, name='drop1_1'))

  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv2_1
  model.add(Convolution2D(64, 3, 3, activation='relu', name='conv2_2'))
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool2_1'))
  model.add(Dropout(0.25, name='drop2_1'))

  model.add(Flatten())
  model.add(Dense(512, activation='relu', name='dense3_1'))
  model.add(Dropout(0.5, name='drop3_1'))
  model.add(Dense(nb_classes, activation='softmax', name='dense3_2'))

# Remove the last dropout layer
def CNNModel_1(model, nb_classes):
  layer_ds = model.layers.pop()
  layer_do = model.layers.pop()

  model.outputs = [model.layers[-1].output]
  model.layers[-1].outbound_nodes = []

  model.add(Dense(nb_classes, activation='softmax', name='dense3_2'))
  model.output_shape

def training(model, X_train, Y_train, X_test, Y_test, batch_size, nb_epoch, data_augment
  ...
  if not data_augmentation:
    hist = model.fit(X_train, Y_train, batch_size=batch_size,
    nb_epoch=nb_epoch, validation_data=(X_test, Y_test), shuffle=True)
  ...
  model.save(modelName) # save the trained model

def RunMain():
  # 1) Remove the last dropout layer
  model = basicCNNModel(input_shape, nb_classes)
  model = CNNModel_1(model, nb_classes)
  hist = training(model, X_train, Y_train, X_test, Y_test, batch_size, nb_epoch, data_au
```

- Training/validation loss and error (accuracy rate)

  I compare the performance results with ones from the given CNN model. Fig.1 and 2 show the baseline performance generated from the given CNN model, and Fig.3 and 4 show the outcomes of the first changed CNN model.

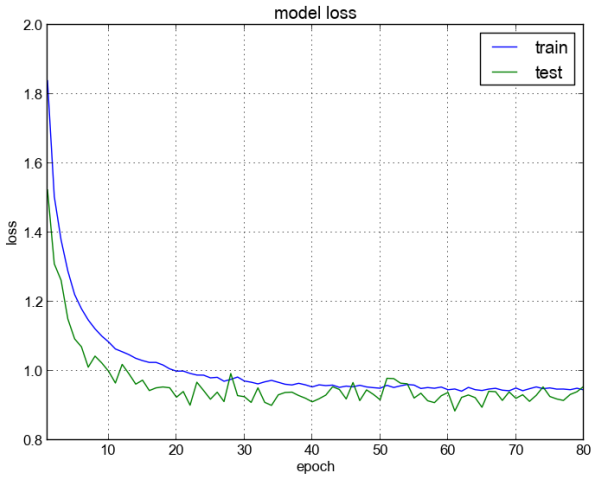

Figure 1: Loss with the last dropout

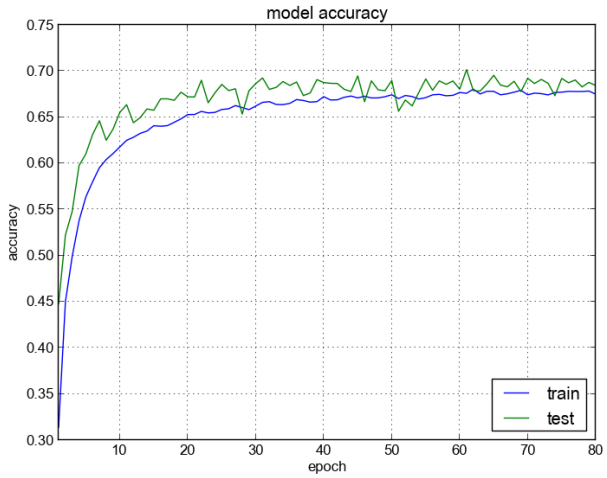

Figure 2: Accuracy with the last dropout

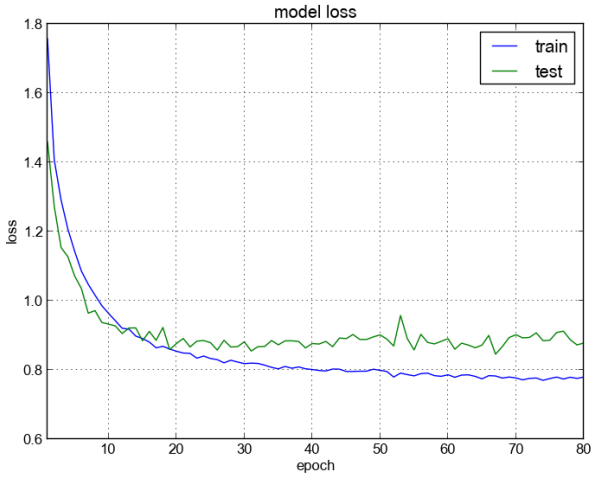

Figure 3: Loss without the last dropout

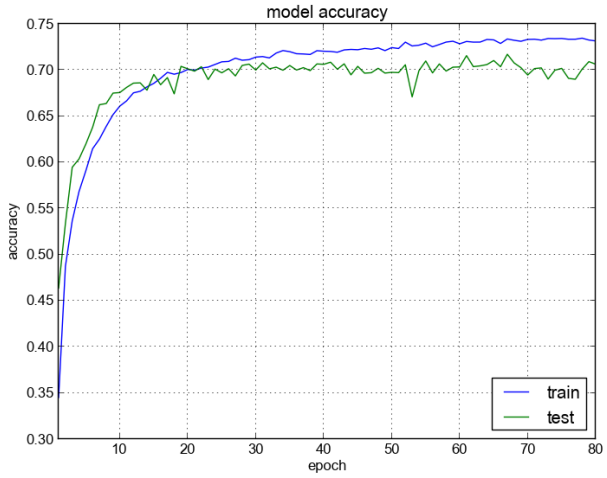

Figure 4: Accuracy without the last dropout

Analysis 1: The best validation accuracy of the given CNN model (with the last dropout layer) is 0.6920 (Fig. 2), and the best validation accuracy of the modified CNN model without the last dropout layer is 0.7171 (Fig. 4). Although dropout helps prevent overfitting, the dropout layer after FC layers does not really improve the validation performance, rather it reduces the gap between train and validation outcomes as generalizing training performance. Thus, I conclude that the dropout after FC layers does not play an important role to improve the classification task and even more we have a little better performance without the last dropout layer.

Problem 2. Load the model you saved at step 1 as initialization to the training. Add another fully connected layer with 512 filters at the second-to-last layer (before the classification layer) (10 points). Train and save the model.

- CNN Architecture

  INPUT: 32 32 3
  AVGPOOL: 16 16 3
  CONV/RELU: 16 16 32
  CONV/RELU: 14 14 32
  MAXPOOL: 7 7 32
  DROPOUT: 7 7 32

  CONV/RELU:: 7 7 64
  CONV/RELU:: 5 5 64
  MAXPOOL: 2 2 64
  DROPOUT: 2 2 64

  FLATTEN: 256
  DENSE/RELU: 512
  DENSE/RELU: 512
  DENSE/SOFTMAX: 10

Algorithm 2 : Add another fully connected layer

```
# add fully connected layer with 512 filters
def CNNModel_2(model, nb_classes):
        layer_ds = model.layers.pop()

        model.outputs = [model.layers[-1].output]
        model.layers[-1].outbound_nodes = []

        model.add(Dense(512, activation='relu'))
        model.add(Dense(nb_classes, activation='softmax'))

def RunMain():
        # 2) Add another fully connected layer with 512 filters
        model = load_model('my_model_1.h5')
        model = CNNModel_2(model, nb_classes)
        hist = training(model, X_train, Y_train, X_test, Y_test, batch_size, nb_epoch, d
```

- Training/validation loss and error after adding another fully connected (FC) layer

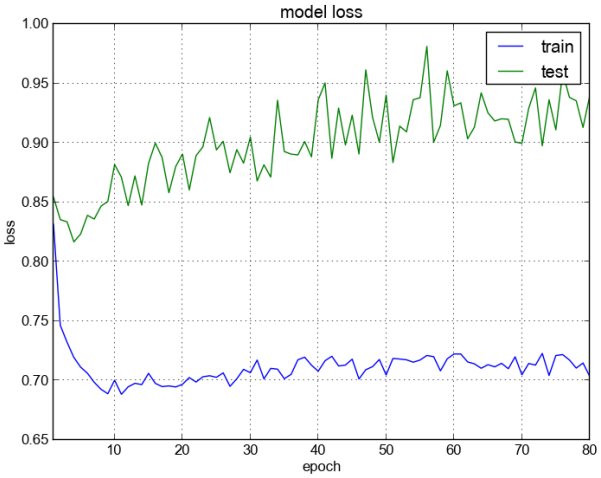

Figure 5: Loss with two FC

Figure 6: Accuracy with two FC

Analysis 2-1: Adding a fully connected layer somehow improves both training and validation performance (The best validation accuracy is 0.7279 in Fig. 6), but it has a great degree of variance for the validation performances (Fig.5 and 6). It may show that deeper CNN layers can increase the performance of classification, but deeper FC layers might not be a good strategy due to the oscillation of validation performance. Rather we can make a deeper CNN layer by adding more CONV layers. I will show the performance results from another deeper CNN network with more CONV layers in the following 4.2 section. Therefore, we need to well design the CNN network architecture, considering its complexity (i.e. layer depth) and generalization (i.e. dropout).

Problem 3. Try to use an adaptive schedule to tune the learning rate, you can choose from RMSprop, Adagrad and Adam (10 points).

I used three different adaptive schedule to tune the learning rate: stochastic gradient descent (SGD), Adagrad, and Adam. While SGD and Adam show fluctuated convergence, Adagrad is quite stable to converge to the optimal points. Adagrad controls the step size based on the size of gradient. Therefore, it reduces oscillation of error updates, however it converges slowly depending on the gradient size. In Fig.7 to 10, we can see that Adagrad shows more smooth convergence of loss/error than SGD and Adam optimizations in a little bit slower rate.

Algorithm 3 : Adagrad Optimizer

```
from keras.optimizers import SGD, Adagrad, Adam
def training(model, X_train, Y_train, X_test, Y_test, batch_size, nb_epoch, data_augment
  if optimizer == "adagrad":
          opt = Adagrad(lr=learningRate, decay=0.0, epsilon=1e-08)
  elif optimizer == "adam":
          opt = Adam(lr=learningRate, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0
  else:
          opt = SGD(lr=learningRate, decay=1e-6, momentum=0.9, nesterov=True)
  model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

I choose Adagrad optimizer to tune the learning rate. I tuned the learning rate with 3 different values, [0.1, 0.01, 0.001] based on the given model as following architecture.

- CNN Architecture

  INPUT: 32 32 3
  AVGPOOL: 16 16 3
  CONV/RELU: 16 16 32
  CONV/RELU: 14 14 32
  MAXPOOL: 7 7 32
  DROPOUT: 7 7 32
  CONV/RELU:: 7 7 64
  CONV/RELU:: 5 5 64
  MAXPOOL: 2 2 64
  DROPOUT: 2 2 64
  FLATTEN: 256
  DENSE/RELU: 512
  DROPOUT: 512
  DENSE/SOFTMAX: 10

Analysis 3: From the results shown in Fig.13 to 18, learning rate 0.1 is not a good choice since it does not learn at all (Fig.13 and 14), and 0.001 is either not a good choice due to the significant slowness to converge to the optimal solution (Fig.17 and 18). Thus, 0.01 is the most proper choice as it converges faster than 0.001 without oscillations (Fig.15 and 16).

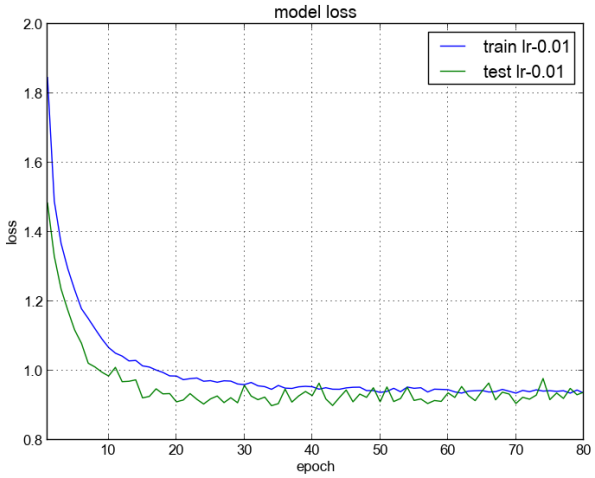

Figure 7: Loss with SGD

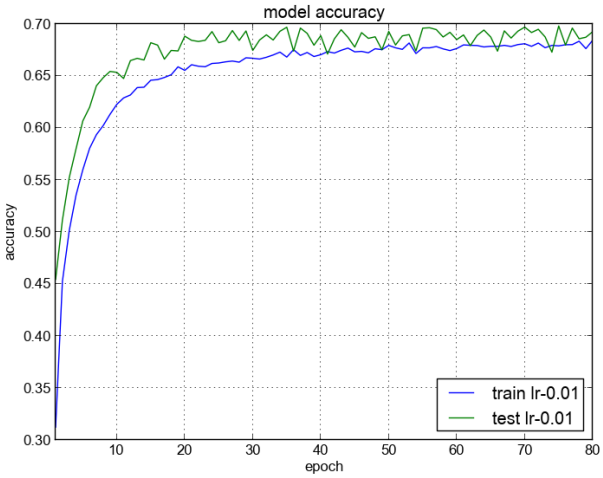

Figure 8: Accuracy with SGD

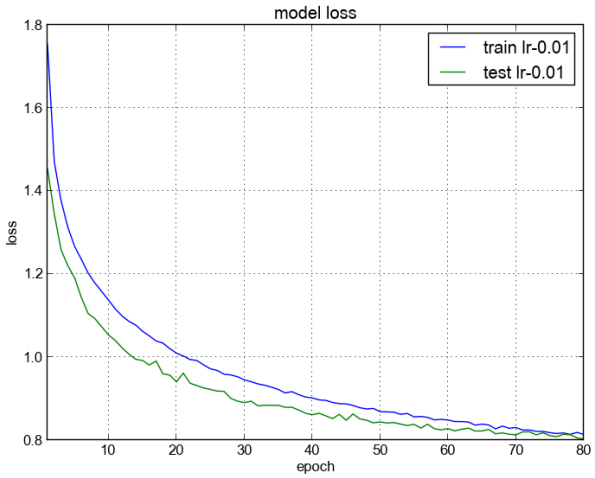

Figure 9: Loss with Adagrad

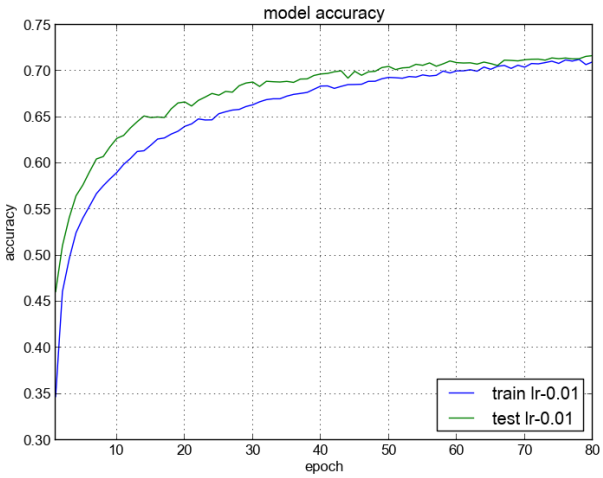

Figure 10: Accuracy with Adagrad

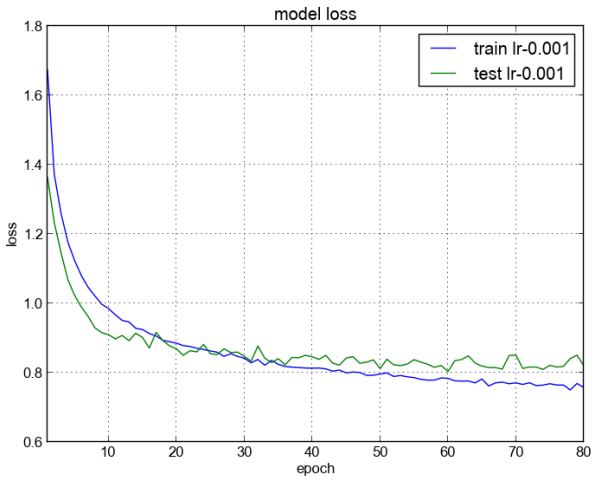

Figure 11: Loss with Adam

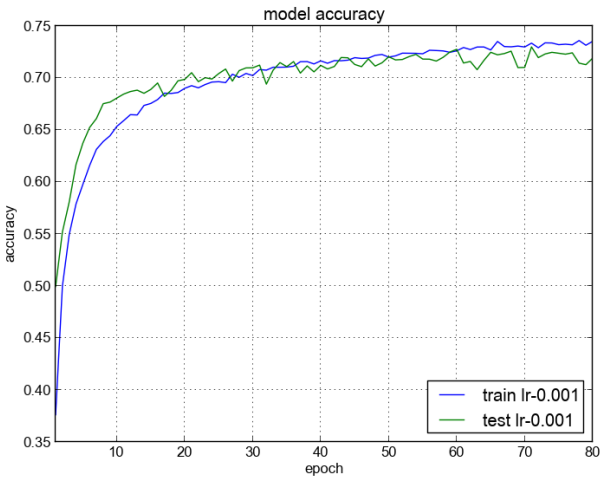

Figure 12: Accuracy with Adam

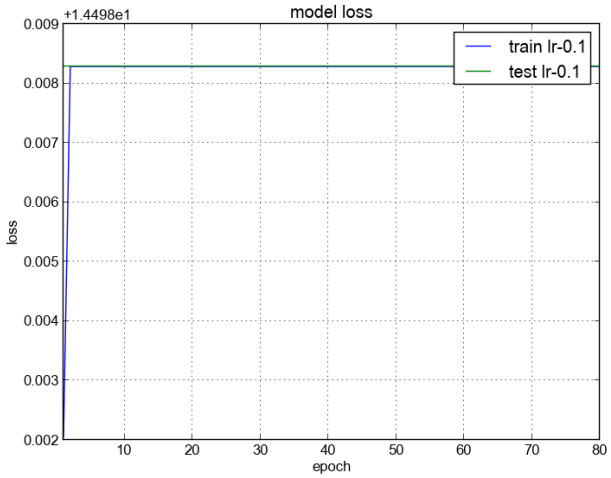

Figure 13: Loss with learning rate 0.1

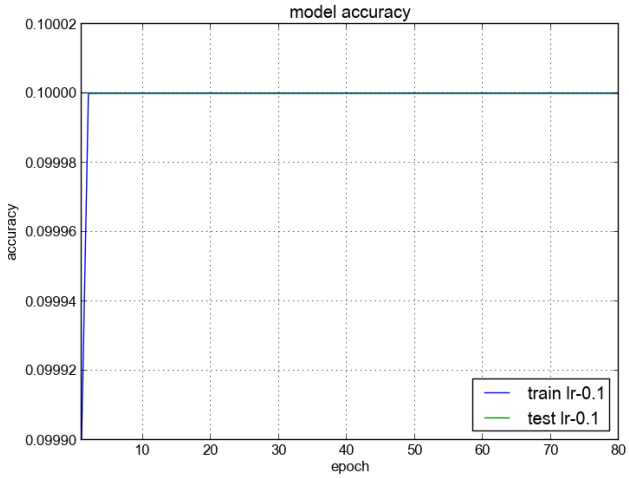

Figure 14: Accuracy with learning rate 0.1

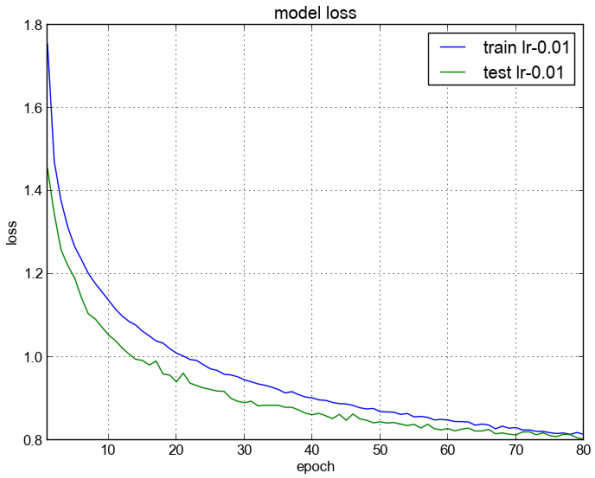

Figure 15: Loss with learning rate 0.01

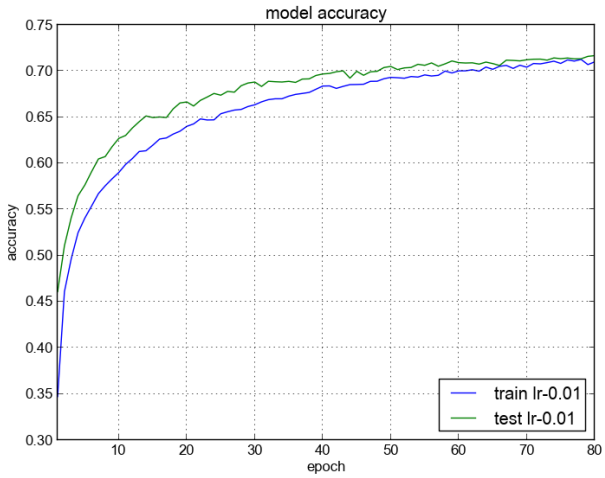

Figure 16: Accuracy with learning rate 0.01

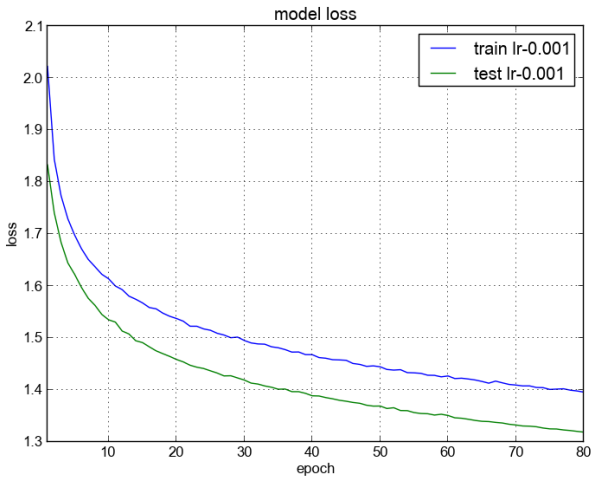

Figure 17: Loss with learning rate 0.001

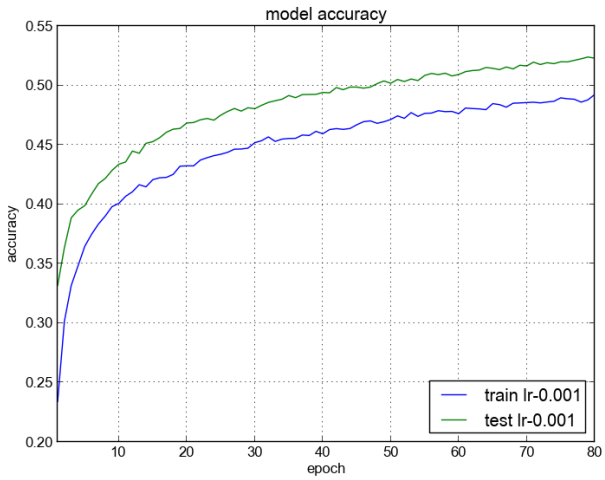

Figure 18: Accuracy with learning rate 0.001

Problem 4. Try to tune your network in two other ways (10 points) (e.g. add/remove a layer, change the activation function, add/remove regularizer, change the number of hidden units) not described in the previous four. You can start from random initializations or previous results as you wish.

1) Remove the AVGPOOL layers and increase the number of hidden units of FC from 512 filters to 4096 filters.

Found that the size of input image greatly diminished by the first AVGPOOL layer and CONV layers with the default border mode, I removed the AVGPOOL layer and let the border mode of CONVs same to keep the previous image size except MAXPOOL layers. As we now have larger parameters from the last CONV layer, I also increase the number of hidden units of FC layer from 512 filters to 4096 filters.

- CNN Architecture

INPUT: 32 32 3
CONV/RELU: 32 32 32
CONV/RELU: 32 32 32
MAXPOOL: 16 16 32
DROPOUT: 16 16 32

CONV/RELU:: 16 16 64
CONV/RELU:: 16 16 64
MAXPOOL: 8 8 64
DROPOUT: 8 8 64

FLATTEN: 4096
DENSE/RELU: 4096
DENSE/SOFTMAX: 10

Analysis 4-1: This model outperformed the previous given model, showing the higher validation accuracy, 0.7675 (Fig. 20). I think it is because this model keeps more non-linear complexed features without losing spatial information. That is, it might be able to better represent the image features as keeping the whole spatial information.

Algorithm 4 : CNNModel 3 without AVGPOOL layer

```
def CNNModel_3(input_shape, nb_classes):
  model = Sequential()
  model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=input_shape, border_m
  model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same', name='conv1_2
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool1_1'))
  model.add(Dropout(0.25, name='drop1_1'))

  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv2_1
  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv2_2
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool2_1'))
  model.add(Dropout(0.25, name='drop2_1'))

  model.add(Flatten())
  model.add(Dense(4096, activation='relu', name='dense3_1'))
  model.add(Dense(nb_classes, activation='softmax', name='dense3_2'))
```

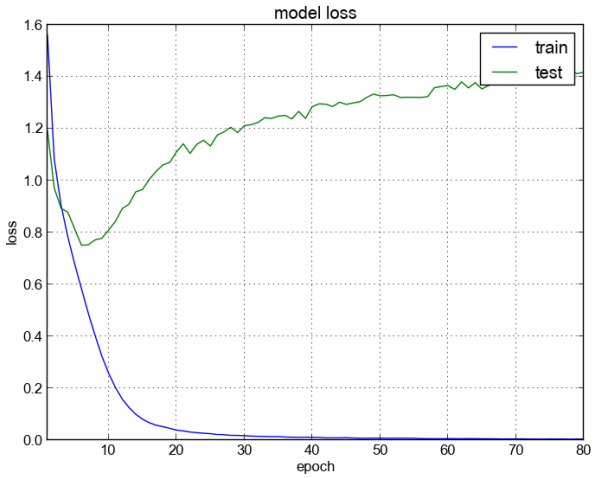

Figure 19: Loss without AVGPOOL

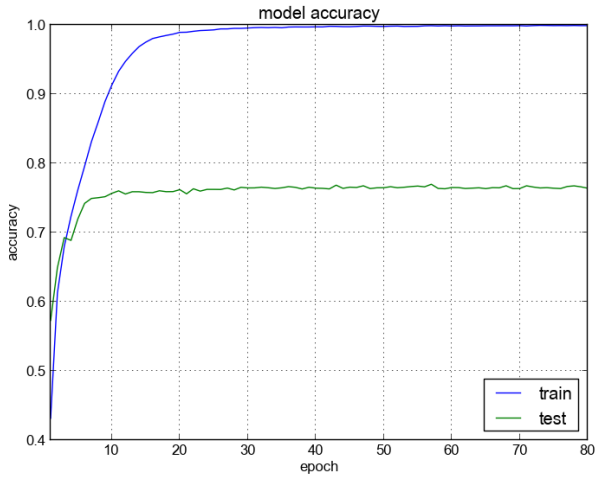

Figure 20: Accuracy without AVGPOOL

2) Add CONV/RELU and MAXPOOL layers.

Based on the previous modification, I also made changes of adding two convolution layers and one MAXPOOL layer to have deeper layers. In other words, I tried to obtain more image features, and to reduce the number of parameters of FC layer. Additionally, I applied data augmentation, and compare its outcomes with ones without it.

- CNN Architecture

INPUT: 32 32 3
CONV/RELU: 32 32 32
CONV/RELU: 32 32 32
MAXPOOL: 16 16 32
DROPOUT: 16 16 32

CONV/RELU:: 16 16 64
CONV/RELU:: 16 16 64
MAXPOOL: 8 8 64
DROPOUT: 8 8 64

CONV/RELU: 8 8 64
CONV/RELU: 8 8 64
MAXPOOL: 4 4 64
DROPOUT: 4 4 64

FLATTEN: 256
DENSE/SIG: 256
DENSE/SOFTMAX: 10

Algorithm 5 : CNNModel 4 with additional CONV layers

```
def CNNModel_4(input_shape, nb_classes):
  model = Sequential()
  model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=input_shape, border_m
  model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same', name='conv1_2
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool1_1'))
  model.add(Dropout(0.25, name='drop1_1'))

  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv2_1
  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv2_2
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool2_1'))
  model.add(Dropout(0.25, name='drop2_1'))

  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv3_1
  model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', name='conv3_2
  model.add(MaxPooling2D(pool_size=(2, 2), name='pool3_1'))
  model.add(Dropout(0.25, name='drop3_1'))

  model.add(Flatten())
  model.add(Dense(256, activation='relu', name='dense4_1'))
  model.add(Dense(nb_classes, activation='softmax', name='dense4_2'))
```

- Training/validation loss and error without data augmentation

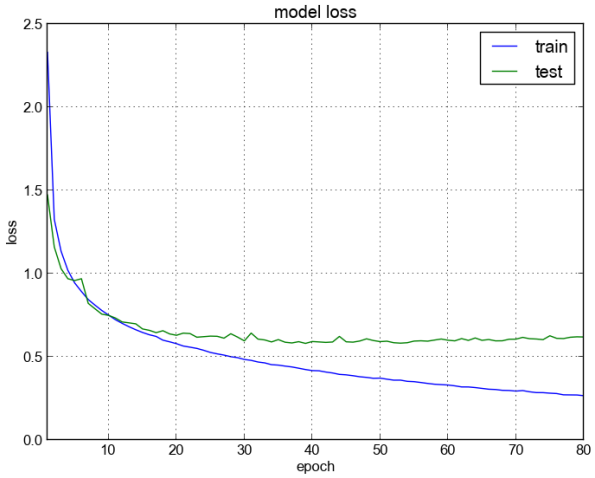

Figure 21: Loss with deeper CONV layers

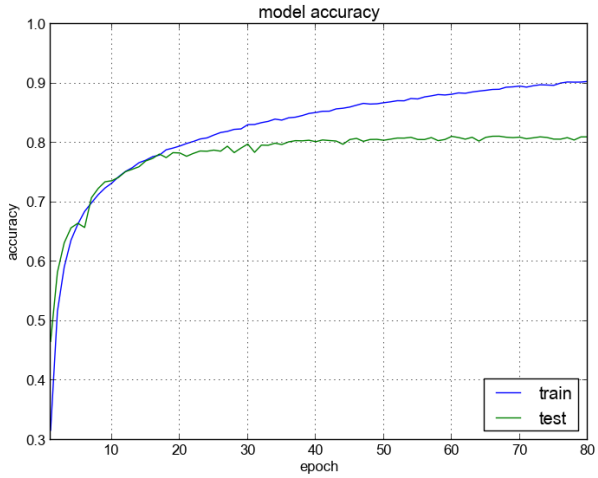

Figure 22: Accuracy with deeper CONV layers

- Training/validation loss and error with data augmentation

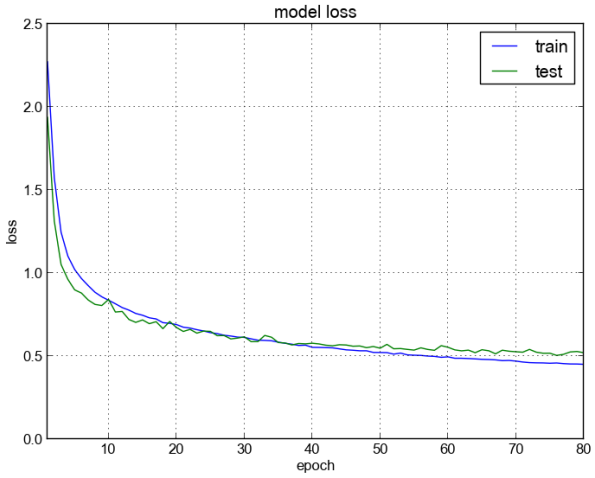

Figure 23: Loss with deeper CONV layers

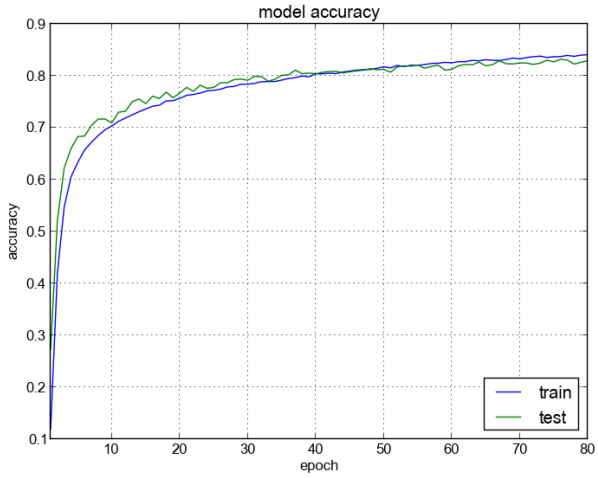

Figure 24: Accuracy with deeper CONV layers

Analysis 4-2: It turns out that having more convolution layers generates much better classification performance comparing to the previous CNN models with smaller number of convolution layers. The convoluted layers are apt for representing the generalized forms of images features, thus, this deeper CNN model with more CONVs layers helps to improve validation classification performance. In addition, since we have a smaller number of hidden units of a FC layer, it also improves the loss/accuracy as well as time/space complexity. Before enabling data augmentation, this model obtained 0.8112

validation accuracy (Fig. 22), which is higher than the previous tested CNN model with fewer convolution layers and a higher number of FC hidden units .

Finally, data augmentation prevents overfitting with many generalized data. This technique vastly reduced the gap between training and validation loss/error, increasing validation performance (Fig. 24). In sum, without AVGPOOL layer, and with more convoluted layers, a simple fully connected layer, and data augmentation, I could achieve the best validation accuracy, 0.8336, which is the highest result among all experiments I tried.