ECE 156B: Neural Networks

Vicki Chen
University of California, Santa Barbara

1 Introduction

In this assignment, we are given a large data set to choose or design a machine learning model and get the best classification accuracy. The training data is stored in a 50000x3027 matrix as called *train_data.npy*. Labels for the training data are stored in another numpy array with 50000 integer values called *train_label.npy*. There are 10 types of labels, each specified by an integer 0 to 9 which corresponds to a certain class. We are given the task to generate a neural network model that can accurately classify an unlabeled dataset. To tackle this problem, I first looked into what the numbers in the array represented. After noticing the numbers stood for pixel values of images, I researched image recognition and classification methods.
By using Keras along with tensorflow as backend, I developed a Deep Neural Network and was able to accurately classify 80% of the unknown data:

CodaLab Username: icxiv
Score: 79.9% accuracy

2 Method

2.1 Setup

To load and set up the data, we must first notice that in order to feed an image data into a CNN model, we need change the dimension of the input tensor to (width x height x num_channel).
The row vector for an image has the exact same number of elements when you calculate 32*32*3 == 3072. In order to reshape the row vector into (width x height x num_channel) form, we must first reshape the array and then transpose it.
Reshape transforms an array to a new shape without changing its data.
However, we need to swap the order of each axes by calling transpose. This will change the shape to fit the expected shape for tensorflow.

```
train = np.load('train_data.npy')
```

```
labels = np.load('train_label.npy')
test = np.load('test_data.npy') #test at very end

X = train
y = labels

X_train, X_test, y_train, y_test = \
   train_test_split(X, y, test_size=.15, random_state=42)

# images_train = X_train
# lables_train = y_train
# images_test = X_test
# labels_test = y_test

X_train = X_train.reshape((len(X_train), 3, 32, 32)).transpose(0, 2, 3, 1)
X_test = X_test.reshape((len(X_test), 3, 32, 32)).transpose(0, 2, 3, 1)
test = test.reshape((len(test), 3, 32, 32)).transpose(0, 2, 3, 1)
input_shape = (32, 32, 3)
```

The pixel values are in the range of 0 to 255 for each of the red, green and blue channels. Since the input values are well understood, we can easily normalize to the range 0 to 1 by dividing each value by the maximum observation which is 255. Since data is loaded as integers, we must cast it to floating point values in order to perform the division.

```
# Making sure that the values are float so that we can get decimal points after division
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
test = test.astype('float32')

# Normalizing the RGB codes by dividing it to the max RGB value.
X_train /= 255
X_test /= 255
test /= 255
```

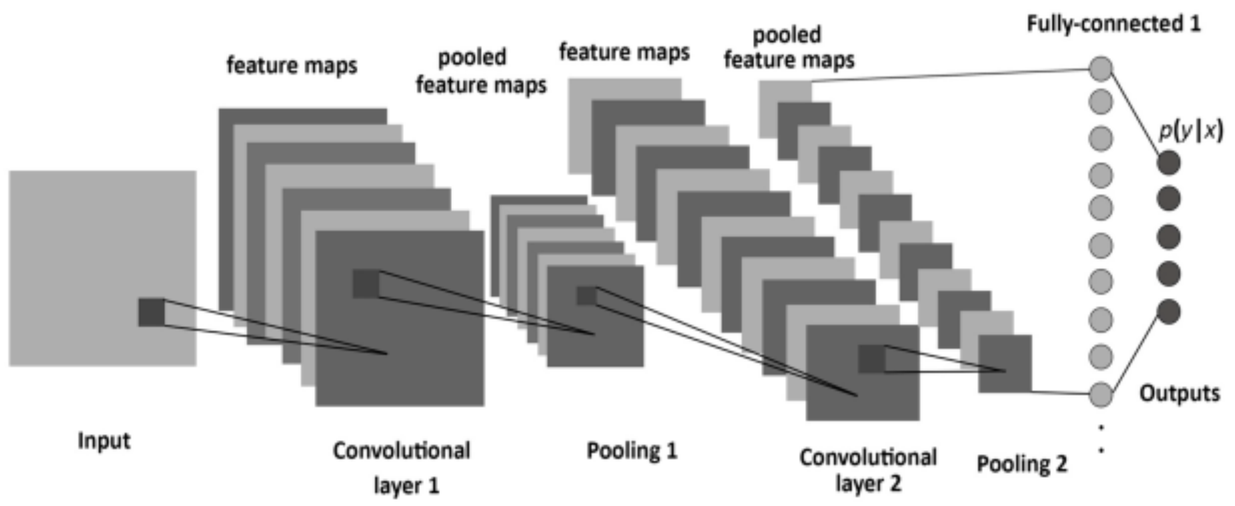
The output variables are defined as a vector of integers from 0 to 1 for each class. We can use a one hot encoding to transform them into a binary matrix in order to best model the classification problem. Since there are 10 classes for this problem, we specify the num_classes (width) as 10.

Lastly, using sklearn's module train_test_split, I split the training data into train and test data that could be used to build the model.

3 Convolutional Neural Network

When I started looking at learning deep learning tutorials with neural networks, I realized that one of the most powerful supervised deep learning techniques is the Convolutional Neural Networks (CNN). The final structure of a CNN is actually very similar to Regular Neural Networks (RegularNets) where there are neurons with weights and biases. Like in RegularNets, we use a loss function and an optimizer (e.g. Adam optimizer) in CNNs. Additionally, there are also Convolutional Layers, Pooling Layers, and Flatten Layers in CNNs.
The process of building a Convolutional Neural Network majorly involves four major blocks show below.

Convolution layer → Pooling layer → Flattening layer → Dense/Output layer



Convolution layer is set of 3 operations: Convolution, Activation & Batch normalization. Therefore, I will import the Sequential Model from Keras and add Conv2D, MaxPooling, Flatten, Dropout, and Dense layers.

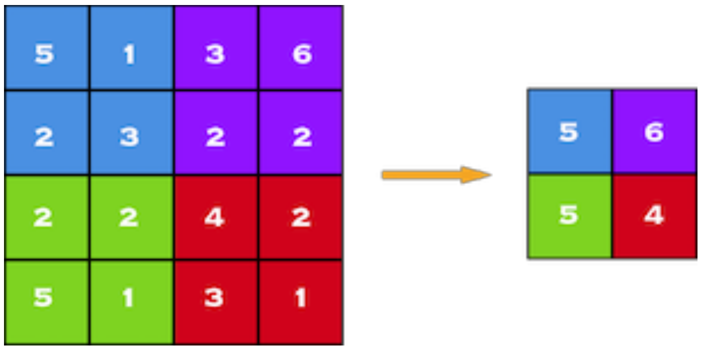
```
model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(Conv2D(32, (3, 3)))
# Batch normalization layer added here
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256))
# Batch normalization layer added here
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
adam = Adam(lr=0.0006, beta_1=0.9, beta_2=0.999, decay=0.0)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer=adam)
```

3.1 Pooling Layer

When constructing my CNN, I inserted a pooling layer after the convolution layer to reduce the spatial size of the representation to reduce the parameter counts which reduces the computational complexity. In addition, pooling layers also helps with the overfitting problem. Basically we select a pooling size to reduce the amount of the parameters by selecting the maximum, average, or sum values inside these pixels. Max Pooling, one of the most common pooling techniques and is the one I used for my model:



Max Pooling by 2 x 2

4 Regularization

In many cases and mine, larger models have a tendency to over fit training data. While getting good performance on the training set, it performs poorly on the test set. Regularization methods are used to prevent overfitting, making these larger models generalize better.

4.1 Dropout

Dropout is a method of regularization and works on a neural network layer by masking a random subset of its outputs (zeroing them) for every input with probability p and scaling up the rest of the outputs by $1/(1 - p)$. Masking prevents gradient backpropagation through the masked outputs. The method selects a random subset of the neural network to train on. During testing, p is set to zero. This can be interpreted as averaging the outputs of the ensemble models. Because of the scaling, the expected layer outputs are the same during training and testing.

```
model.add(Dropout(0.25))
```

4.2 Batch Normalization

Batch Normalization normalizes the activation of the previous layer at each batch. It applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1. Batch Normalization achieves the same accuracy with fewer training steps thus speeding up the training process This speeds up training and improves the final performance of the model.

```
model.add(BatchNormalization())
```

4.3 Data Augmentation

Another boost in model performance can be achieved by using some data augmentation. Methods such as standardization and random shifts and horizontal image flips may be beneficial. In this case, I added in random horizontal and vertical shifts, rotation range, as well as a horizontal flip to further improve the training accuracy. In Keras, we have a I*mageDataGenerator* class that is used to generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches) indefinitely. The image data is generated by transforming the actual training images by rotation, crop, shifts, shear, zoom, flip, reflection, normalization etc.

```
datagen = ImageDataGenerator( rotation_range=90,
            width_shift_range=0.1, height_shift_range=0.1,
            horizontal_flip=True)
datagen.fit(X_train)
```

5 Results

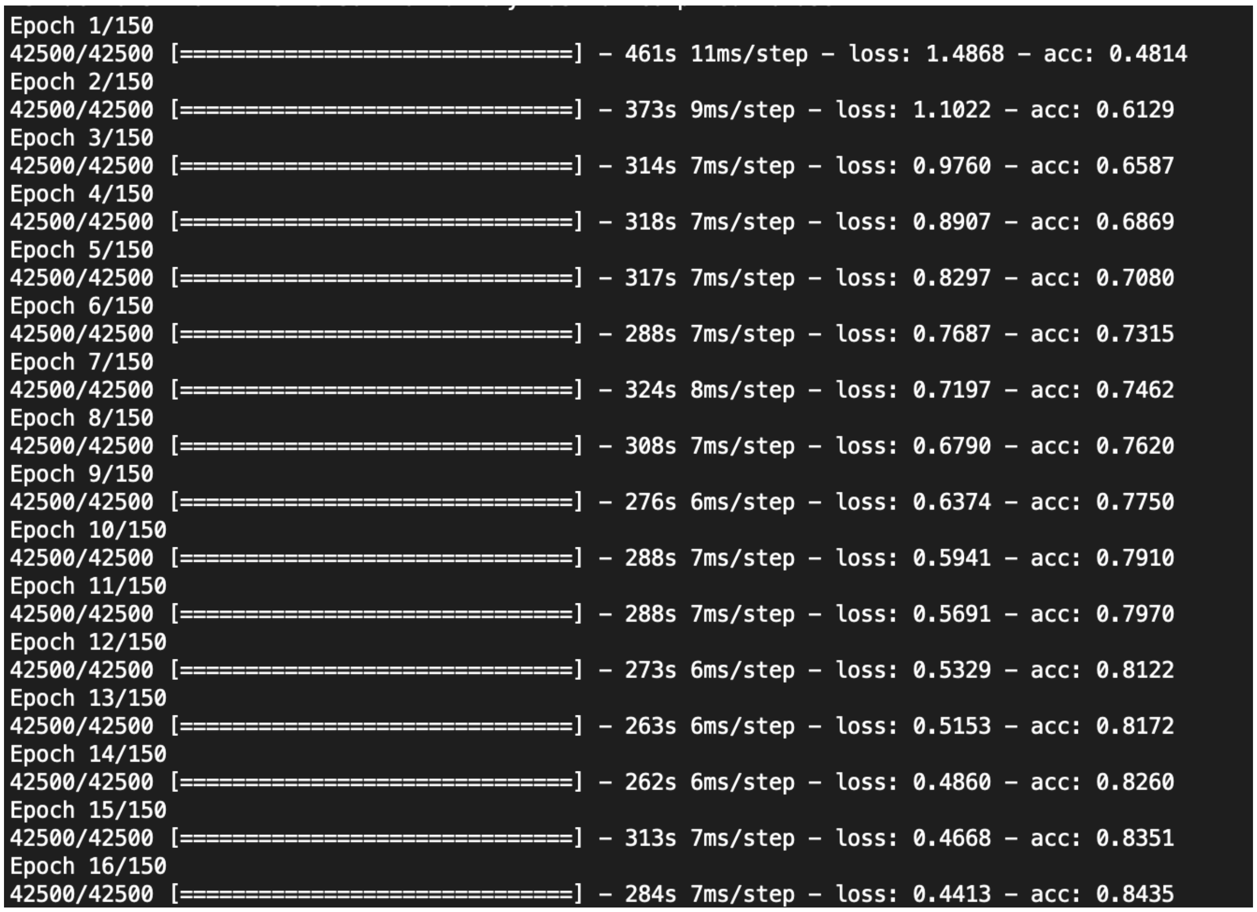
5.1 Compiling and Evaluating the model

```
adam = Adam(lr=0.0006, beta_1=0.9, beta_2=0.999, decay=0.0)

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

After setting up the model, we can then fit the model by using our train data. There's several different optimizers available to use on models, but the one I chose is Adam. After researching the difference between optimizers, I came to the conclusion that Adam learns the fastest and is more stable than the other optimizers since it doesn't suffer any major decreases in accuracy. For learning rate, I set it to 0.0006 since its standard for first time experiments

The training results:



```
Epoch 1/150
42500/42500 [==============================] - 461s 11ms/step - loss: 1.4868 - acc: 0.4814
Epoch 2/150
42500/42500 [==============================] - 373s 9ms/step - loss: 1.1022 - acc: 0.6129
Epoch 3/150
42500/42500 [==============================] - 314s 7ms/step - loss: 0.9760 - acc: 0.6587
Epoch 4/150
42500/42500 [==============================] - 318s 7ms/step - loss: 0.8907 - acc: 0.6869
Epoch 5/150
42500/42500 [==============================] - 317s 7ms/step - loss: 0.8297 - acc: 0.7080
Epoch 6/150
42500/42500 [==============================] - 288s 7ms/step - loss: 0.7687 - acc: 0.7315
Epoch 7/150
42500/42500 [==============================] - 324s 8ms/step - loss: 0.7197 - acc: 0.7462
Epoch 8/150
42500/42500 [==============================] - 308s 7ms/step - loss: 0.6790 - acc: 0.7620
Epoch 9/150
42500/42500 [==============================] - 276s 6ms/step - loss: 0.6374 - acc: 0.7750
Epoch 10/150
42500/42500 [==============================] - 288s 7ms/step - loss: 0.5941 - acc: 0.7910
Epoch 11/150
42500/42500 [==============================] - 288s 7ms/step - loss: 0.5691 - acc: 0.7970
Epoch 12/150
42500/42500 [==============================] - 273s 6ms/step - loss: 0.5329 - acc: 0.8122
Epoch 13/150
42500/42500 [==============================] - 263s 6ms/step - loss: 0.5153 - acc: 0.8172
Epoch 14/150
42500/42500 [==============================] - 262s 6ms/step - loss: 0.4860 - acc: 0.8260
Epoch 15/150
42500/42500 [==============================] - 313s 7ms/step - loss: 0.4668 - acc: 0.8351
Epoch 16/150
42500/42500 [==============================] - 284s 7ms/step - loss: 0.4413 - acc: 0.8435
```

I also set my program up so that after every 50 epochs, it saves a model.

```
mc = keras.callbacks.ModelCheckpoint('newmodel{epoch}.h5', save_weights_only=False,
period=50)
model.fit(x=X_train, y=y_train, callbacks=[mc], epochs=150)
```

When the model is done training, I have another program called predict.py which runs the newest model on the unknown data and saves the output class labels as a numpy file named **Prediction.npy**

```
model = keras.models.load_model("newmodel99.h5")
predictions = model.predict_classes(test)
np.save("prediction", predictions)
```

6 Conclusion and Improvements

After testing through several different epoch models on CodaLab, I came to the conclusion that the model trained with 100 epochs had the highest accuracy. Although I was able to get 80% accuracy, I had a huge problem with training speed since I did not have access to an available GPU.

Some improvements can be done such as:

- **Using a GPU**: My biggest problem with this lab was the training speed my CPU provided. In comparison to running the program with a GPU, a CPU trains 10x slower. My google cloud account request was not approved and google colab gave me an unexpected loading error which hindered me from training and testing more programs.
- **Train for More Epochs**: Each model was trained for a very small number of epochs, 20, 50, 100, 150. It is common to train large convolutional neural networks for hundreds or thousands of epochs.
- **Changing the parameters**: Changing different parameters such as learning rate, activation, dropout can significantly change your model accuracy.
- **Deeper Network**: Although I'm already implementing a deep neural network, larger networks can definitely be designed for the problem. This may involve more feature maps closer to the input and perhaps less aggressive pooling.

References

https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1