3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 7

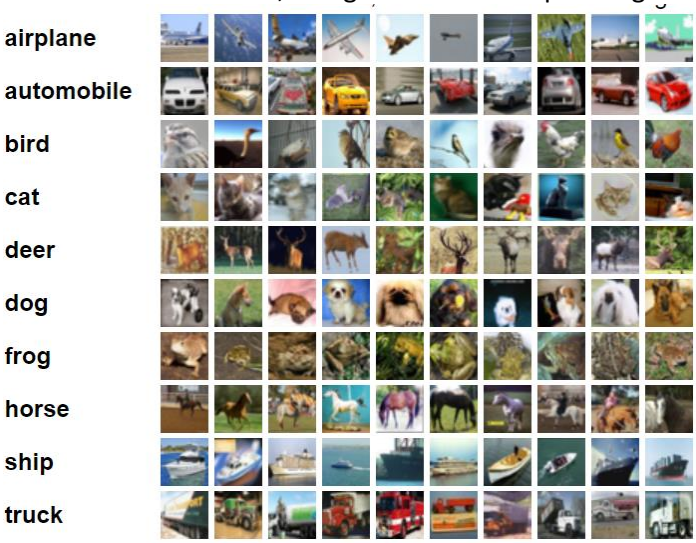
Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Complex Data: Image Classification with TensorFlow

For this lab we are using Convolution Neural Networks to classify images from the well-known CIFAR-10 dataset. From [1]; the dataset description: "The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class."

Here are the 10 classes, along with some sample images:



There are two ways to import the dataset, one by downloading the tar.gz file and using 'pickle' to unpack it into python dicts. Or we can import from the tensorflow keras datasets.

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
##OR##
from tensorflow.python.keras.datasets import cifar10
```

For the main CNN implementation, I am using the unpickle approach as given in the exercise sheet. For proving the data augmentation and showing classes, I am importing from keras datasets as it is easier.

Part 1: Convolutional Neural Networks for Image Classification (CNN)

This part is to build the simple CNN architecture, and establish a training pipeline. The purpose is to have it ready for the expanded CNN in Part 2. Here, I detail the approach for data pre-processing, data augmentation, and effects of regularization on neural networks.

Architecture of CNN:

1. Conv1: convolution and rectified linear activation (RELU)
2. Pool1: max pooling
3. FC1: fully connected layer with rectified linear activation (RELU)
4. Softmax layer: final output predictions i.e. classify into one of the ten classes.

Build a training pipeline:

Step 1: Data pre-processing

There are many available resources for pre-processing CIFAR-10 dataset, for this exercise I follow the approach as detailed in [4].

- Normalize data. Here we are using Min-Max normalization. The original image data will be transformed in range of 0 to 1.

```
def normalize_data(x):  # Min-max normalization of pixel values (to [0-1] range)
    min_val = np.min(x)
    max_val = np.max(x)
    x = (x-min_val) / (max_val-min_val)
    return x
```

- We have to one-hot encode the target / labels. Since this is a classification task, the labels are in ranges 1-10 for each class of image category. The output will be an [Nx10] matrix.

```
def onehotencoding(labels, nclasses):
    outlabels = np.zeros((len(labels), nclasses))
    for i, l in enumerate(labels):
        outlabels[i, l] = 1
    return outlabels
```

- Function below is defined, in order to split up the dataset into data (X) and labels (Y). Within this function, the above normalize and one-hot functions are called. Additionally, my laptop RAM does not support a full one batch, hence I pass 'range' parameter to limit the number of rows. This is a trade-off in terms of model accuracy, but I have no other choice in order to develop the stated architecture.

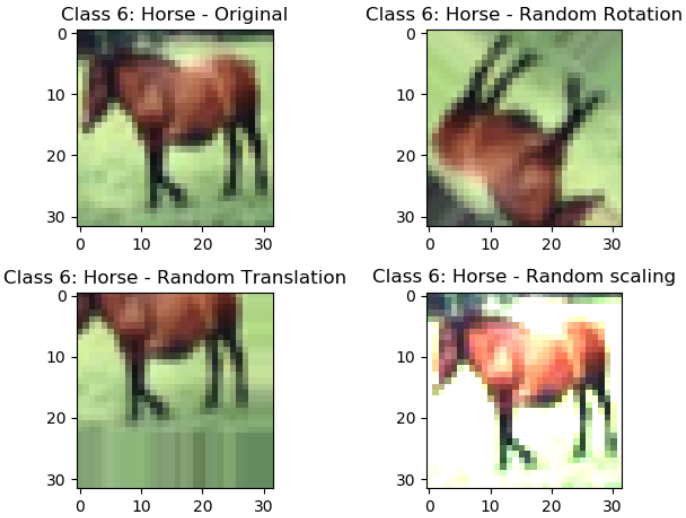
```
dir = 'C:/PythonProjects/cifar-10-batches-py/'
data_train = unpickle(dir+'data_batch_1')
data_test = unpickle(dir+'test_batch')

def generate_data(input_data,range):
    X_data = input_data[b'data'] # m * n
    X_data = X_data[0:range]
    X_data = normalize_data(X_data)
    X_data = X_data.reshape(-1, 32, 32, 3)

    Y_data = np.array(input_data[b'labels'])
    Y_data = Y_data[0:range]
    Y_data = onehotencoding(Y_data,10) # 10 image classes

    return X_data,Y_data
```

- Finally, we can generate our test/train dataset by running the two commands as show below. The return output shows the data rows, number of channels (3), image width and height (32), and the number of classes (10).

```
xTrain, yTrain = generate_data(data_train,7000)
xTest, yTest = generate_data(data_test,3000)
# Print output
Shape of X_Train data:  (7000, 32, 32, 3)
Shape of Y_Train labels:  (7000, 10)
Shape of X_Test data:  (3000, 32, 32, 3)
Shape of Y_Test labels:  (3000, 10)
```

Step 2: Data Augmentation

Machine learning works best when there is plenty of training data (in this case images), because this prevents overfitting of the model [5]. If the training data is limited, we can artificially increase it by data augmentation. Here additional images are created by randomly scaling, rotating and translating on the original training image. We can use Keras ImageDataGenerator for this part, showing the effects of each image transformation. The ImageDataGenerator class that is used to generate batches of tensor image data with real-time data augmentation, meaning that the data will be looped over in batches indefinitely. The code for below proof is in "Lab7DataAug.py" and it's based upon the code in [5].



We can combine all the operations into one block as shown below. This must be implemented on the training dataset only, using datagen.fit(x_train). And we can use by calling datagen.flow(batches).

```
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=180, # Rotate by degrees
    width_shift_range=0.4,  # For translating image vertically
    height_shift_range=0.4, # For translating image horizontally
    horizontal_flip=True,
    rescale=2, # For rescaling images
    fill_mode='nearest', # Fill empty pixels
)
datagen.fit(x_train)
x_batch,y_batch = datagen.flow(x_train,y_train,batch_size=5)
```

Step 3: TensorFlow CNN architecture

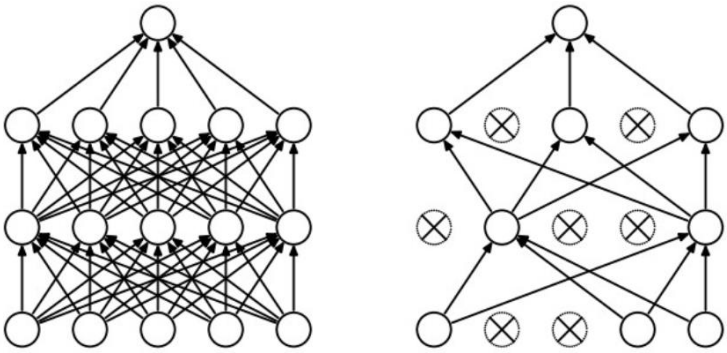
Based on the data exploration, we can define the following variables:

```
nInput = 32
nChannels = 3
nClasses = 10

X = tf.placeholder(tf.float32, [None, nInput, nInput, nChannels])
Y = tf.placeholder(tf.float32, [None, nClasses])
keep_prob = tf.placeholder(tf.float32)
```

Note that 'keep_prob' hyperparameter is used for dropout. Dropout in machine learning is a way of neural network regularization [6]. During training, dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. Dropout is implemented by only keeping a neuron active with some probability as we define it, 'keep_prob' which I set as 0.4 after trial and error testing.

```
def drop_out(fc, keep_prob=0.4):
    drop_out = tf.layers.dropout(fc, rate=keep_prob)
    return drop_out
```

This image from [6] shows the description of NN dropout:



(a) Standard Neural Net (b) After applying dropout.

We have the following helper functions, which will be called in the CNN architecture:

1. Conv2D. Strides are simply set to '1', since I found it takes too long for larger strides. We maintain the zero padding ('SAME'). Also, we are using ReLU activation function here.

```
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

2. Maxpool2D. This function to take the largest value after striding (convolving) the filter across it. I just set the strides = 2 in this case.

```
def maxpool2d(x, k=2):
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')
```

3. Based on the architecture given, we define the conv_net as follows:

```
weights = {
    'WC1': tf.Variable(tf.random_normal([5, 5, 3, 32]), name='W0'),
    'WD1': tf.Variable(tf.random_normal([16*16*32, 64]), name='W1'),
    'out': tf.Variable(tf.random_normal([64, nClasses]), name='W5')
}

biases = {
    'BC1': tf.Variable(tf.random_normal([32]), name='B0'),
    'BC2': tf.Variable(tf.random_normal([64]), name='B1'),
    'out': tf.Variable(tf.random_normal([nClasses]), name='B5')
}

def conv_net(x, weights, biases):
    conv1 = conv2d(x, weights['WC1'], biases['BC1'])
    conv1 = maxpool2d(conv1)

    fc1 = tf.reshape(conv1, [-1, weights['WD1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['WD1']), biases['BC2'])
    fc1 = tf.nn.relu(fc1)
    fc1 = drop_out(fc1)  # NN regularization

    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    out = tf.nn.softmax(out)

    return out
```

Here we add some additional parts in the implementation:

Convolution layer ==> Pooling layer ==> Flattening layer ==> Fully connected layer==>Output layer

After applying 1 layer of convolution and 1 layer of max-pooling operation, we then down-sample the input image from 32x32x3 to 16x16x3. Then we flatten (reshape) this down-sampled output to feed this as the input into the fully connected layer. We return the softmax / dense of it.

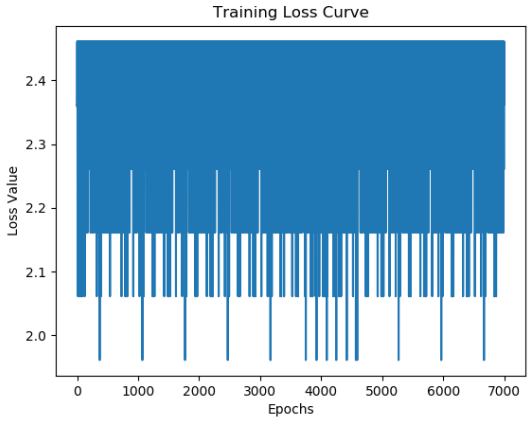
```
training_epochs = 10
learning_rate = 0.05
batch_size = 10
total_batches = xTrain.shape[0] // batch_size

pred = conv_net(X, weights, biases)
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=Y))
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(loss)
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

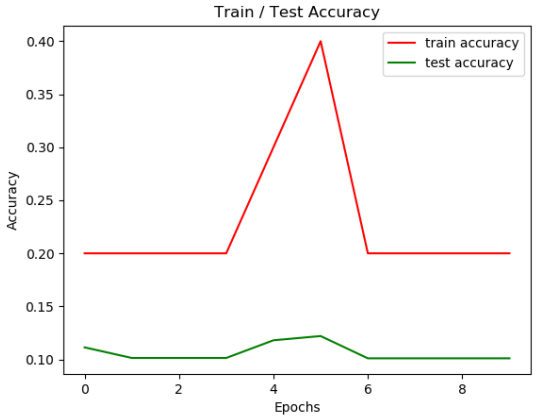
The hyperparameters are shown as above, from trial and error, the best learning rate seems to be 0.05. The prediction, loss, optimizer, correct prediction, and accuracy formulas are exactly the same as my Lab 6 code.

The TensorFlow session block shown below is also exactly the same as previous Labs, we just feed the testing and training data and labels into it, and plot the loss curve and accuracies.

```
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    summary_writer = tf.summary.FileWriter('./tensorboard/train', graph=tf.get_default_graph())
    for epoch in range(training_epochs):
        for batch in range(total_batches):  # Mini-batch for SGD
            batch_x = xTrain[batch * batch_size:min((batch + 1) * batch_size, len(xTrain))]
            batch_y = yTrain[batch * batch_size:min((batch + 1) * batch_size, len(yTrain))]
            _, c, acc,summary = sess.run([optimizer, loss,accuracy, merged_all_], feed_dict={X: batch_x, Y: batch_y})
            training_loss.append(c)
            summary_writer.add_summary(summary, epoch)
        print('Epoch: ', epoch, 'Training Loss =', '{:.9f}'.format(c), 'Training accuracy = {:.5f}'.format(acc))
        train_accuracy.append(acc)
        _, corrtest = sess.run([optimizer, accuracy], feed_dict={X: xTest, Y: yTest})
        test_accuracy.append(corrtest)

    print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: xTest, Y: yTest}))
```

Results:



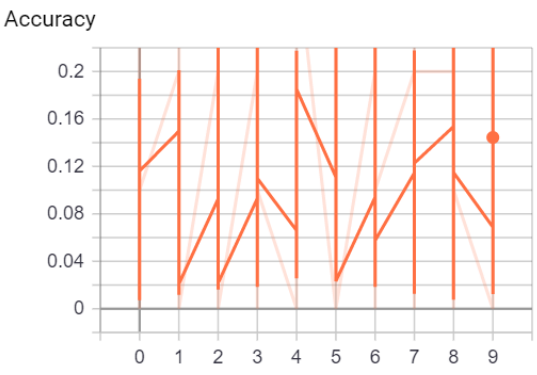


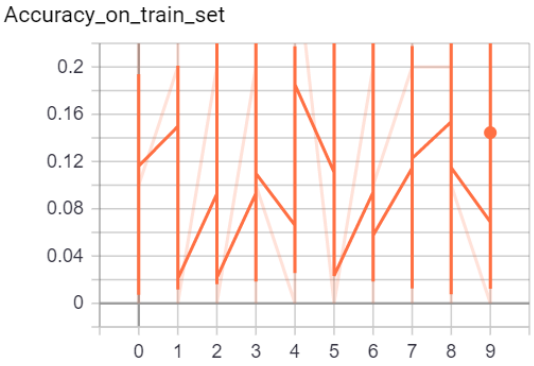
As the above plots show, the results are quite erratic. However, this is to be expected, as I was forced to reduce the volume of training data. That was still not enough in fact, as I would get the following error when running any iteration of the CNN.

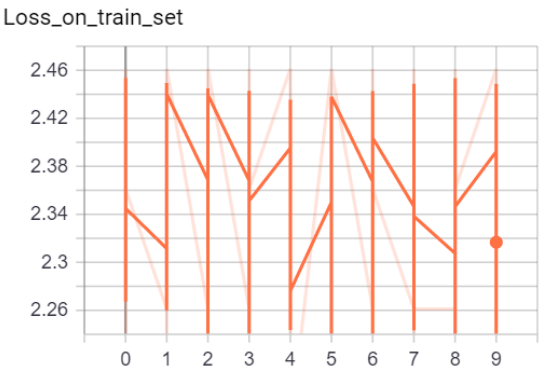
```
Epoch:  0 Training Loss = 2.361150265 Training accuracy = 0.10000
2019-06-23 16:12:10.551231: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
2019-06-23 16:12:11.695853: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
```

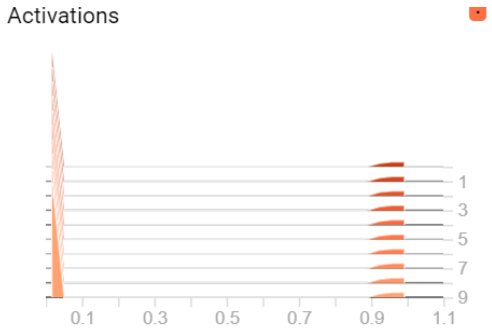
I have to limit the number of epochs to 10, otherwise my laptop will run out of RAM and crash. Anyway, we have achieved the desired output, which is the simple CNN implementation. This results are further explained in the Discussion sections, which follows after Part 2 results.

Tensorboard graphs:









Part 2: Convolutional Neural Networks for Image Classification (CNN)

The steps for implementation the CNN architecture here are mostly similar as in Part 1, except here we have a few more layers for the convolution.

1. conv1: convolution and self-normalization activation
2. pool1: max pooling
3. norm: batch normalization
4. pool2: max pooling
5. FC1: fully connected layer with self-normalization activation
6. FC2: fully connected layer with self-normalization activation
7. Output (softmax) layer: final output predictions i.e. classify into one of the ten classes.

We can visualize the flow:

Convolution layer (+batch norm) ==> Pooling layer ==> Batch normalization ==> Pooling layer ==> Flatten ==> Fully connected layer (+dropout) ==> Fully connected layer (+dropout) ==> Output layer

The main addition here is extra convolution and fully connected layer. And, we introduce batch normalization to the architecture. Batch normalization is a method we can use to normalize the inputs of each layer, in order to fight the internal covariate shift problem [6]. During training time, the batch normalization layer does the following:

- Calculate the mean and variance of the layers input.
- Normalize the layer inputs using the previously calculated batch statistics.
- Scale and shift in order to obtain the output of the layer.

We implement the batch norms using the following helper function. It is called between the maxpool layers.

```
def normalize_layer(pooling):
    norm = tf.contrib.layers.batch_norm(pooling, center=True, scale=True)
    return norm
```

The full CNN architecture is shown below. One thing to note is the SeLU activation function.

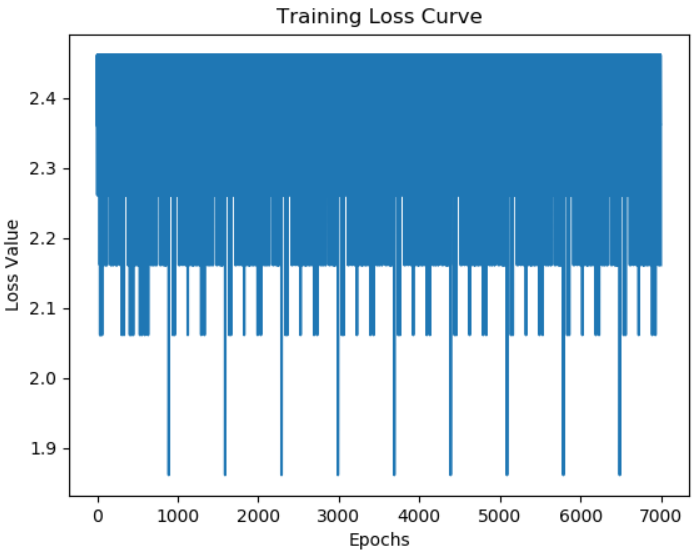
```
weights = {
    'WC1': tf.Variable(tf.random_normal([5, 5, 3, 32]), name='W0'),
    'WC2': tf.Variable(tf.random_normal([5, 5, 32, 64]), name='W1'),
    'WD1': tf.Variable(tf.random_normal([8 * 8 * 64, 64]), name='W2'),
    'WD2': tf.Variable(tf.random_normal([64, 128]), name='W3'),
    'out': tf.Variable(tf.random_normal([128, nClasses]), name='W5')
}

biases = {
    'BC1': tf.Variable(tf.random_normal([32]), name='B0'),
    'BC2': tf.Variable(tf.random_normal([64]), name='B1'),
    'BD1': tf.Variable(tf.random_normal([64]), name='B2'),
    'BD2': tf.Variable(tf.random_normal([128]), name='B3'),
    'out': tf.Variable(tf.random_normal([nClasses]), name='B5')
}
```

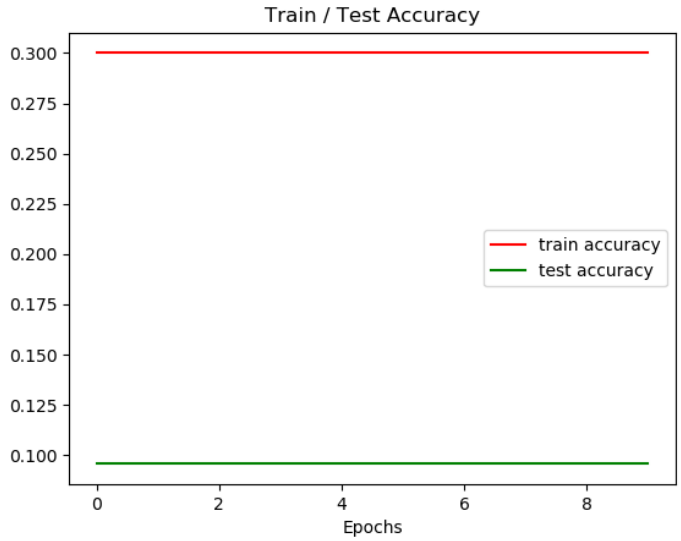
```python
def conv_net(x, weights, biases):
    conv1 = conv2d(x, weights['WC1'], biases['BC1'])
    conv1 = maxpool2d(conv1)
    conv1 = normalize_layer(conv1)

    conv2 = conv2d(conv1, weights['WC2'], biases['BC2'])
    conv2 = maxpool2d(conv2)

    fc1 = tf.reshape(conv2, [-1, weights['WD1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['WD1']), biases['BD1'])
    fc1 = tf.nn.selu(fc1)  # Using self-normalization activation
    fc1 = drop_out(fc1)

    fc2 = tf.add(tf.matmul(fc1, weights['WD2']), biases['BD2'])
    fc2 = tf.nn.selu(fc2)  # Using self-normalization activation
    fc2 = drop_out(fc2)

    out = tf.add(tf.matmul(fc2, weights['out']), biases['out'])
    out = tf.nn.softmax(out)

    return out
```

Besides this, other parts of the code are the same as in Part 1.

Results:

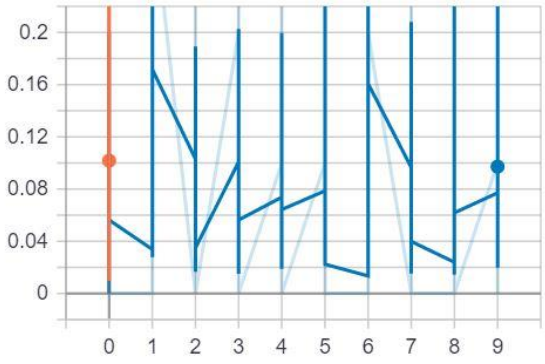
Training and testing accuracy are (slightly) better as compared to Part 1, we can see from the graph. This CNN learns very slowly. Again, due to RAM limitations, the number of epochs is limited to 5 here. But we can see that it performs better with more epochs and utilizing the full training dataset. Besides this, I sometimes get differing results, it really depends when the program hits RAM limitations. I have attached different sample outputs in Appendix A.





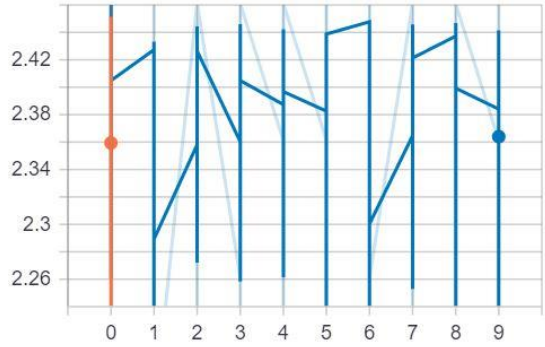
Tensorboard graphs:

Accuracy

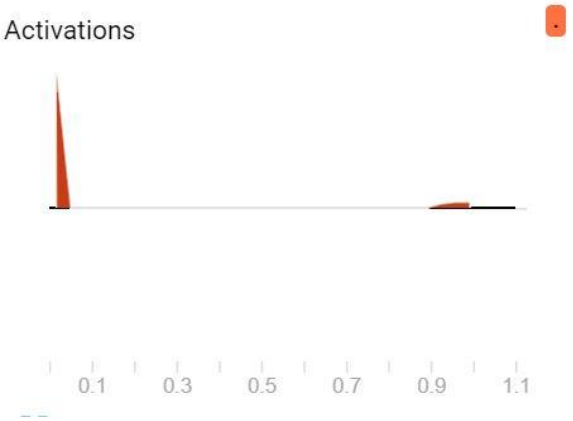


Accuracy_on_train_set

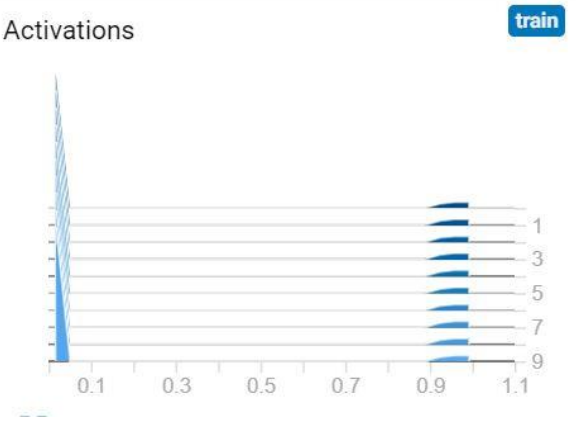
Loss_on_train_set



Activations



Activations



Discussions:

Although both my CNNs achieved low training and test accuracy, this is to be expected due to many reasons.

1. Due to my system limitations (8GB RAM, no GPU) I was forced to limit my training dataset and batch size. The iterations were only set to a small number to prevent system crash.
2. The given architecture with only one or two hidden layers would not provide a good accuracy in any case.
3. As explained in [7], it must be understood that a neural network can't always learn with the same accuracy. This program is just example of an implementation of a convolution neural network. If we want to get a good accuracy, we need to implement more complicated CNN models (such as ResNet, GoogleLeNet, mobileNetV2 etc.)

References:

[1] https://www.cs.toronto.edu/~kriz/cifar.html
[2] https://github.com/moritzhambach/Image-Augmentation-in-Keras-CIFAR-10-
[3] CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.github.io/convolutional-networks/
[4] https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c
[5] https://samyzaf.com/ML/cifar10/cifar10.html#Load-training-and-test-data
[6] http://cs231n.github.io/neural-networks-2/#reg
[7] https://github.com/exelban/tensorflow-cifar-10

Appendix A – Sample outputs on memory issue

```
If you depend on functionality not listed there, please file an issue.

WARNING:tensorflow:From C:/PythonProjects/Lab7Ex1.py:82: dropout (from tens
Instructions for updating:
Use keras.layers.dropout instead.
WARNING:tensorflow:From C:/PythonProjects/Lab7Ex1.py:135: softmax_cross_ent
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.

2019-06-23 13:00:09.810385: I tensorflow/core/platform/cpu_feature_guard.cc
Epoch:  0 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  1 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  2 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  3 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  4 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  5 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  6 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  7 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  8 Training Loss = 2.161150217 Training accuracy = 0.30000
Epoch:  9 Training Loss = 2.161150217 Training accuracy = 0.30000
Testing Accuracy: 0.09566667

Process finished with exit code 0
```

Hitting RAM limitations

```
See `tf.nn.softmax_cross_entropy_with_logits_v2`.

2019-06-23 13:25:29.463536: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this Tensor
Epoch:  0 Training Loss = 2.461150169 Training accuracy = 0.00000
2019-06-23 13:25:51.332020: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
2019-06-23 13:25:57.458452: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
Epoch:  1 Training Loss = 2.461150169 Training accuracy = 0.00000
2019-06-23 13:26:22.095945: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
2019-06-23 13:26:29.011136: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
Epoch:  2 Training Loss = 2.461150169 Training accuracy = 0.00000
2019-06-23 13:26:55.723353: W tensorflow/core/framework/allocator.cc:124] Allocation of 393216000 exceeds 10% of system memory.
Epoch:  3 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  4 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  5 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  6 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  7 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  8 Training Loss = 2.461150169 Training accuracy = 0.00000
Epoch:  9 Training Loss = 2.461150169 Training accuracy = 0.00000
Testing Accuracy: 0.09966667

Process finished with exit code 0
```

Appendix B – TensorBoard graphmap