Machine Learning Practical: Coursework 3

Christopher Sipola (s1667278)

16th February 2017

Contents

1 Introduction

This paper will build a simple neural network model[1] to predict labels in the CIFAR-10 and CIFAR-100 data. The process will be iterative: Each time a new aspect of the architecture is tested (for example, the number of hidden layers), the best performing architecture from that experiment will be used as the basis for the next experiment. The final model will act as a baseline in Coursework 4.

Each experiment has its own section, each with three subsections:

- a description of the experiment, which will start with a summary of all changes made to the starting architecture up to that point;
- an outline of the code, which will highlight the salient details explained in the rest of the section; and
- a discussion of the results.

The majority of the tests will be performed on CIFAR-10 data. After a reasonable baseline architecture has been established, this architecture will be tested on the CIFAR-100 data. There is no strong theoretical reason why this will be done outside of convenience: Testing on both data sets would add another layer of complexity that may not be informative enough to justify the cost. There is, however, an expectation that the best-performing architectures for each of the data sets will resemble one another, since some degree of discrimination between subclasses (that is, the 100 classes in CIFAR-100) must occur naturally for accurate prediction of the CIFAR-10 labels.

Most experiments will use Softmax cross entropy error instead of accuracy as a measure of performance, since it is arguably more nuanced and comprehensive. Accuracy will be used mostly as a means of getting an intuitive sense of how well the model is doing.

For the sake of brevity, this paper will assume most of the background information regarding neural networks and techniques used to tune models. For example, this paper will not discuss what L2 regularization is, what the ReLU activation function is, or why they may be useful when training neural networks.

[1] The architecture will be simple in that it will not be a convolutional or recurrent neural network model.

1.1 Starting architecture, hyperparameters and initialization settings

The experiments in this paper will be based on the architecture provided in lab 9a. This architecture has one input layer, one hidden layer and one output layer, with ReLU nonlinearities wrapping the affine transformations. The weights are initialized using a truncated normal initialization[2] with a standard deviation of $2/\sqrt{input\_dim + output\_dim}$. The biases are initialized with zeros. The training uses an Adam optimizer[3] with a learning rate of 0.001. Multi-class cross entropy error with Softmax[4] is applied to outputs. The data is trained for 20 epochs with batches of 50 training examples.

The code from lab 9a has been provided in the Appendix.

Figure 1 shows a visualization of the architecture, while figure 2 shows the internal architecture of the hidden layer (`fc-layer-1`).

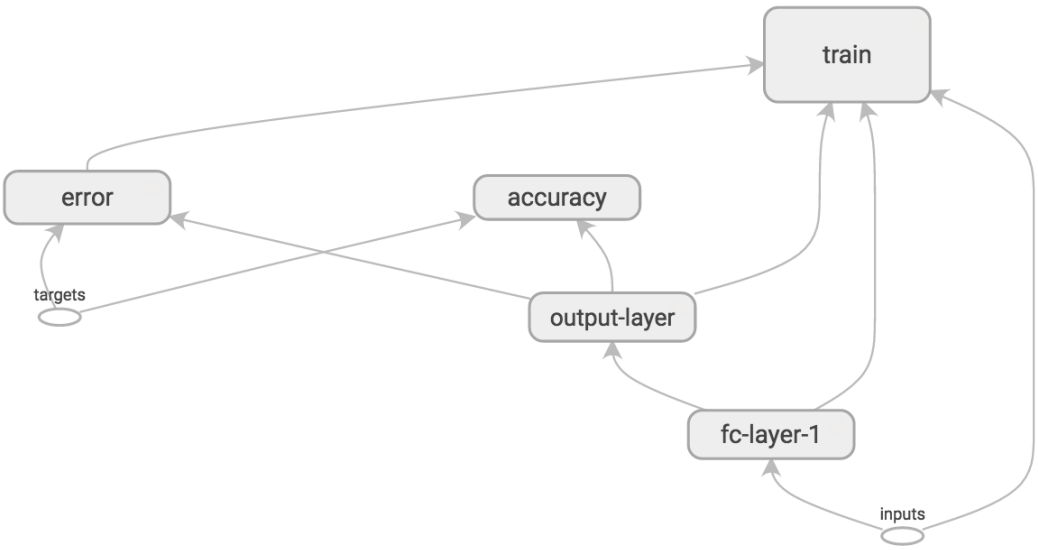


Figure 1: Base architecture given in lab 9a, as visualized by TensorBoard.

[2] `https://www.tensorflow.org/api_docs/python/constant_op/random_tensors#truncated_normal`
[3] `https://www.tensorflow.org/api_docs/python/train/optimizers#AdamOptimizer`
[4] `https://www.tensorflow.org/api_docs/python/nn/classification#softmax_cross_entropy_with_logits`

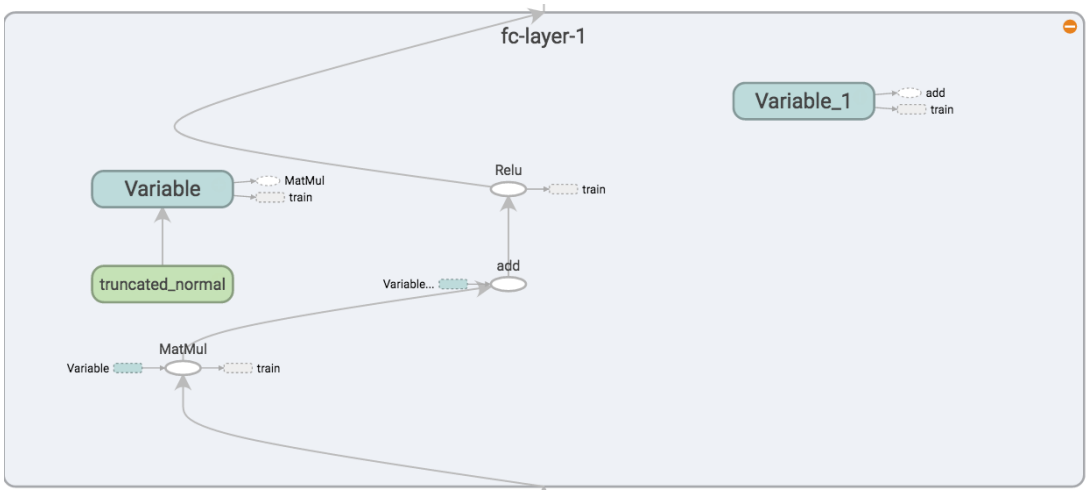


Figure 2: Internal structure of the hidden layer in architecture given in lab 9a.

2 Testing the number of layers and hidden units

2.1 Description of experiment

This section will use the architecture described in the Introduction, except that it will vary the number of hidden layers in $\{1, 2, 3, 4, 5\}$ and the number of hidden units in $\{50, 100, 200, 300\}$, giving twenty combinations that will be tested. The number of hidden units will be kept constant across all hidden layers. The metrics used to measure performance will be validation error, validation accuracy and run time.[5]

If the results showed that 5 hidden layers performed best, the experiment would have been expanded to include architectures with more than 5 hidden layers. Similar reasoning applies for 300 hidden units.

2.2 Code outline

```
...

num_hlayers_list = [1, 2, 3, 4, 5]
num_hidden_list = [50, 100, 200, 300]

for num_hlayers in num_hlayers_list:
    for num_hidden in num_hidden_list:

        ...
```

[5] In other experiments, there will be heavy emphasis on the validation error.

```
        hlayers = []

        # Create first hidden layer
        with tf.name_scope('fc-layer-1'):
            hlayers.append(fully_connected_layer(inputs, train_data.inputs.shape[1],
    num_hidden))

        # Create second, third, etc. hidden layers
        for hl in range(num_hlayers - 1):
            fc_layer_num = hl + 2
            with tf.name_scope('fc-layer-{}'.format(fc_layer_num)):
                hlayers.append(fully_connected_layer(hlayers[-1], num_hidden,
    num_hidden))

        # Create output layer
        with tf.name_scope('output-layer'):
            outputs = fully_connected_layer(hlayers[-1], num_hidden, train_data.
    num_classes, tf.identity)

        ...
...
```

2.3 Results and comments

An architecture with 4 hidden layers and 100 hidden units performed best when balancing validation error, validation accuracy and training time (figure 3).[6] The validation error was 1.50 (the lowest of all combinations), the validation accuracy was 0.477 (close to the best values), and the run time was 144 seconds (which was much lower than those with comparable performance). Note that the validation error and accuracy do not necessarily improve when the number of layers and the number of hidden units are increased.

All curves were plotted to see if there was any unexpected behavior in the convergence of the error curves (figure 4). It seems all models converged in more or less the same manner, reaching a steady validation error value between 5 and 10 epochs without any serious overfitting.

Num. hidden layers \ Num. hidden units	50	100	200	300
1	1.82	1.69	1.58	1.54
2	1.59	1.56	1.51	1.50
3	1.51	1.51	1.51	1.56
4	1.57	1.50	1.60	1.62
5	1.51	1.56	1.57	1.61

(a) **Validation error.**

Num. hidden layers \ Num. hidden units	50	100	200	300
1	0.349	0.411	0.438	0.459
2	0.439	0.453	0.475	0.491
3	0.467	0.473	0.490	0.486
4	0.447	0.477	0.457	0.480
5	0.468	0.454	0.479	0.483

(b) **Validation accuracy.**

Num. hidden layers \ Num. hidden units	50	100	200	300
1	82	112	165	270
2	84	130	229	335
3	96	146	233	408
4	94	144	234	
5	91	124	209	319

(c) **Training time** (seconds).

Figure 3: Validation error and validation accuracy by number of hidden layers and number of hidden units. Green represents better performance and red represents worse performance.

[6] The training time for 4 layers and 300 units was omitted since it was unusually large, likely due to the machine being held up due to other operations.

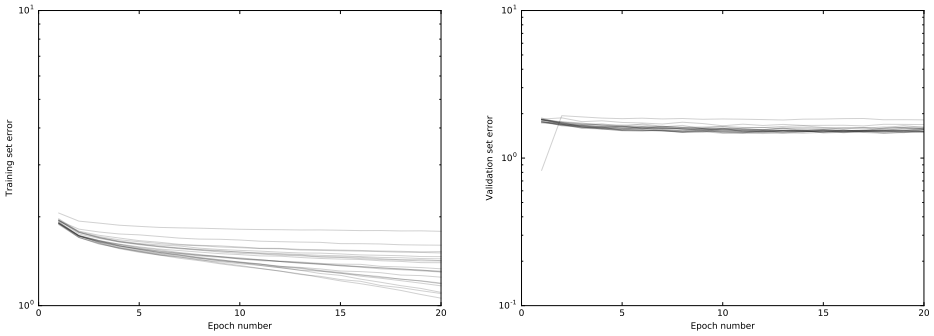


Figure 4: Training and validation error for all combinations of the number of hidden layers and number of hidden units.

3 Testing different activation functions

3.1 Description of experiment

The previous experiment showed that **4 hidden layers, each with 100 hidden units** performed better than 1 hidden layer of 200 hidden units.

This experiment varies the activation function used on the affine transformations. The activation functions that are tested are ReLU (which is what was used in the previous section), dropout (with a dropout percentage of 50%[7]) sigmoid, and tanh.[8] The algorithm uses 30 epochs, since pilot runs showed that validation error curves for some activation functions did not converge.

3.2 Code outline

```
...

def dropout2(x):
    return tf.nn.dropout(x, 0.5, noise_shape=None, seed=None, name=None)

funs = [tf.nn.relu, dropout2, tf.sigmoid, tf.tanh]

for fun in funs:

    ...

    # Set argument nonlinearity=fun in all instances of fully_connected_layer
    function.
```

[7]The course slides recommended using 50% as the dropout percentage: http://www.inf.ed.ac.uk/teaching/courses/mlp/2016/mlp05-hid.pdf.

[8]These were chosen because they represent some of the most widely used activation functions.

```
    ...
...
```

3.3 Results and comments

ReLU performed the best when looking at validation error (figure 5). Validation error for ReLU is lowest at all points in the curve, although it does look like dropout and sigmoid may have been able to train for a few more epochs before reaching convergence. There is no severe overfitting, and the number of epochs could likely be reduced to around 10 or 15.

In short, this experiment results in no change to the architecture.

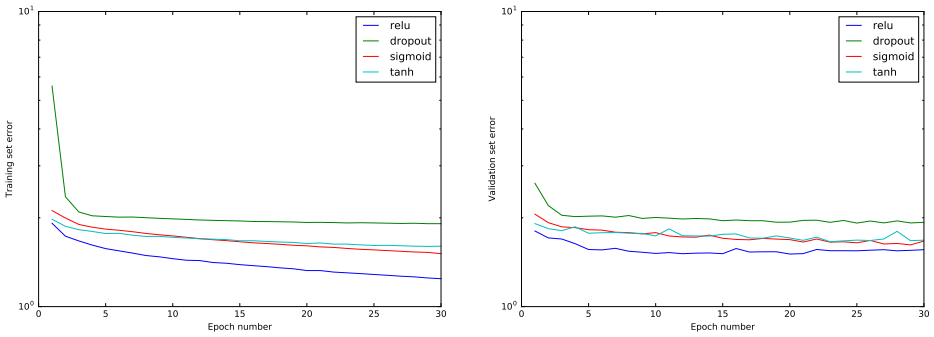


Figure 5: Training and validation error by activation function.

4 Testing learning rate schedules

4.1 Description of experiment

The previous experiments have shown that:

- **4 hidden layers, each with 100 hidden units** perform better than one hidden layer of 200 hidden units; and
- ReLU is the best-performing activation function.

This section will now vary the learning rate schedules. In particular, it will test plain gradient descent, Adagrad, a momentum scheduler, Adam, and RMSProp.[9] Note that Adam was used in the previous experiments.

[9]For technical information regarding these functions, see: `https://www.tensorflow.org/api_docs/python/train/`.

Since the Adam learning rate of 0.001 would not work for all optimizers, a few pilot tests were conducted to determine reasonable learning rates for the various optimizers. For example, Adagrad and gradient descent were not close to reaching convergence, so their learning rates were increased by an order of magnitude to 0.01. The momentum rate used for the momentum scheduler was 0.9, as suggested in the lecture slides.[10]

Thirty epochs were used in order to better observe the behavior of the validation error curves.

4.2 Code outline

```
...

def gradientdescent():
    return tf.train.GradientDescentOptimizer(0.01)

def adagrad():
    return tf.train.AdagradOptimizer(0.01)

def momentum():
    return tf.train.MomentumOptimizer(0.001, 0.9)

def adam():
    return tf.train.AdamOptimizer(0.001)

def rmsprop():
    return tf.train.RMSPropOptimizer(0.001)

optimizers = [gradientdescent(), adagrad(), momentum(), adam(), rmsprop()]

for optimizer in optimizers:

    ...

     with tf.name_scope('train'):
         train_step = optimizer.minimize(error)

    ...
...
```

4.3 Results and comments

Adagrad seems to have the lowest validation error, as well as the most stable curve (figure 6). Additionally, it does not overfit like Adam does. The momentum and gradient descent schedulers also perform well, with gradient descent a bit noisier than the momentum scheduler. RMSProp performs worst, both in training and validation error.

[10] http://www.inf.ed.ac.uk/teaching/courses/mlp/2016/mlp04-learn.pdf

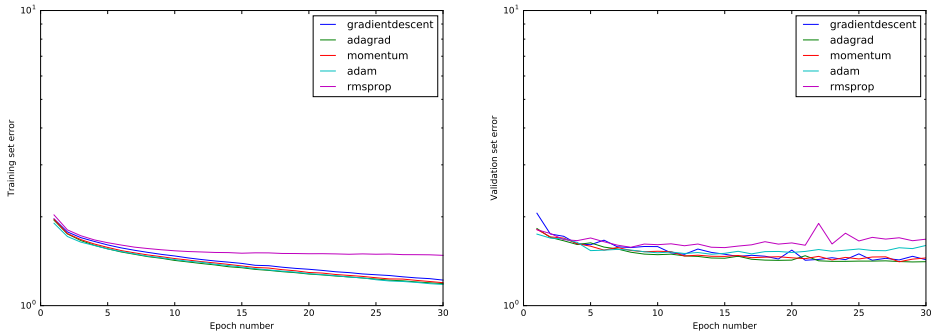


Figure 6: Training and validation error by learning rate schedule.

5 Testing regularization

5.1 Description of experiment

The previous experiments have shown that:

- **4 hidden layers, each with 100 hidden units** perform better than one hidden layer of 200 hidden units;
- ReLU is the best-performing activation function; and
- **Adagrad**, with a learning rate of 0.01, is the best-performing learning rate schedule.

This experiment will test different L2 regularization penalty values in $\{0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3\}$. The same regularization penalty will be used across layers. Again, thirty epochs will be used to more fully observe the behavior of the validation error curves. The learning rate was increased to 0.03 (from 0.01) because pilot runs hinted that higher penalty values may have validation error curves that take longer to converge. The expectation is that regularization will help to prevent overfitting.

5.2 Code outline

```
...

penalties = [0., 0.001, 0.003, 0.01, 0.03, 0.1, 0.3]

for penalty in penalties:

    ...
```

```
    with tf.name_scope('error'):
        # http://stackoverflow.com/a/38466108
        lossL2 = tf.add_n([tf.nn.l2_loss(v) for v in tf.trainable_variables() if '
    bias' not in v.name]) * penalty
        error = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(outputs, targets) + lossL2)

    ...
...
```

5.3 Results and comments

Regularization does not seem to improve validation error in the current architecture (figure 7). It seems like the model with the L2 penalty of 0.001 is perhaps still in the process of converging at 30 epochs; however, pilot runs showed that increasing the learning rate did not result in this model having a better eventual validation error than the implementation without regularization. Regularization penalties above 0.03 converged quickly to poor validation error values.

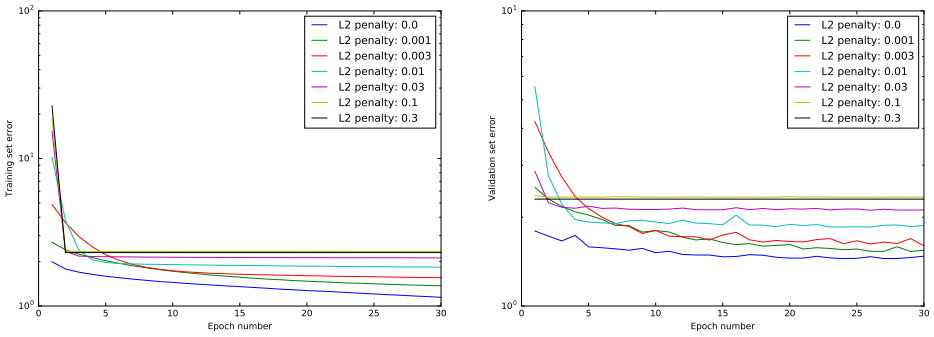


Figure 7: Training and validation error by L2 penalty.

6 Batch normalization

6.1 Description of experiment

The previous experiments have shown that:

- **4 hidden layers, each with 100 hidden units** perform better than one hidden layer of 200 hidden units;
- ReLU is the best-performing activation function;
- **Adagrad**, with a learning rate of 0.01, is the best-performing learning rate schedule; and

- regularization does not seem to improve the convergence of the validation error.

This experiment will simply test batch normalization against the current fully connected layer type. The batch normalization will be implemented within every fully connected layer. The expectation is that this will result in faster convergence and perhaps improved validation error.

6.2 Code outline

Note: Please see Appendix for `fully_connected_layer_batch_norm` code.

```
...
layer_types = [fully_connected_layer, fully_connected_layer_batch_norm]
for layer_type for layer_types:
    ...
    # Replaced all instances of fully_connected_layer with layer_type.
    ...
...
```

6.3 Results and comments

Batch normalization does not seem to improve the convergence of the validation error (figure 8). As expected, it converges more quickly, but the minimum validation error is higher than error values for the current fully connected layer.

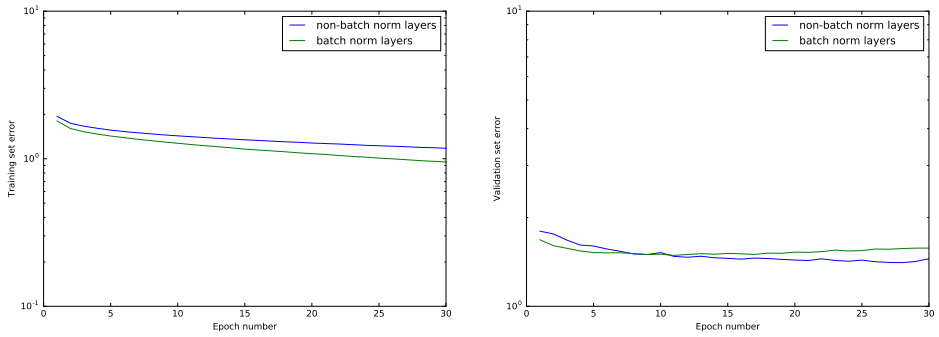
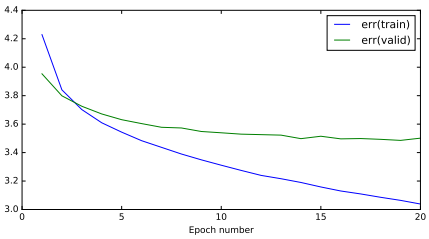


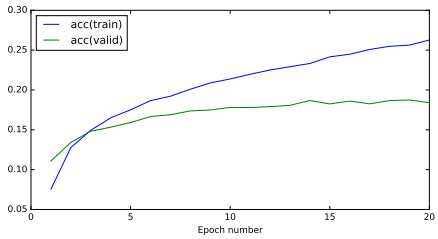
Figure 8: Training and validation error for batch norm when compared against current fully connected layer type.

7 Applying architecture to CIFAR-100

When applying the above architecture to the CIFAR-100 data, the validation error was 3.50 and validation accuracy was 0.18 (figure 9). The metrics are clearly worse than those of CIFAR-10 due to the increased difficulty of the classification task. It is possible that changing the architecture—perhaps by adding hidden layers or hidden units—will help the model to discriminate a bit better between subclasses.



(a) Training and validation error.



(b) Training and validation accuracy.

Figure 9: Error and accuracy for CIFAR-100 given best-performing architecture for CIFAR-10.

8 Summary of experiment results

In this paper, each experiment iterated on the best-performing architecture up to that point, accumulating the improved performance from the prior architectures. In order of their implementation, here are the results of the experiments:

- One hidden layer of 200 hidden units was changed to **4 hidden layers, each with 100 hidden units** because it had the highest validation accuracy, a reasonable validation error and a fast training time when compared to architectures that had similar performance.
- The ReLU activation function was left unchanged because it had the lowest validation error.
- The Adam optimizer was changed to **Adagrad with a learning rate of 0.01** because it had the lowest validation error.
- Regularization was not implemented because it does not seem to improve the convergence of the validation error.
- Batch normalization was not implemented because it does not seem to improve the convergence of the validation error.

The cumulative effect of these changes are that they decreased validation error from 1.58 to around 1.42 and increased validation accuracy from 0.44 to around 0.51 on the CIFAR-10 data set. Run time increased from 165 to 205, which is a small computational cost given the improvements.

9 Further work

Coursework 4 will test a convolutional neural network, since the current architecture does not take advantage of the information from the inherent structure of the image data. Additionally, data augmentation using a techniques such as a Gaussian filter could help to improve performance.

10 Appendix

10.1 Starting code (lab 9a)

```
import os
import tensorflow as tf
import numpy as np
from mlp.data_providers import CIFAR10DataProvider, CIFAR100DataProvider
import matplotlib.pyplot as plt
%matplotlib inline

train_data = CIFAR10DataProvider('train', batch_size=50)
valid_data = CIFAR10DataProvider('valid', batch_size=50)

def fully_connected_layer(inputs, input_dim, output_dim, nonlinearity=tf.nn.relu):
    weights = tf.Variable(
        tf.truncated_normal(
            [input_dim, output_dim], stddev=2. / (input_dim + output_dim)**0.5), '
    weights')
    biases = tf.Variable(tf.zeros([output_dim]), 'biases')
    outputs = nonlinearity(tf.matmul(inputs, weights) + biases)
    return outputs

inputs = tf.placeholder(tf.float32, [None, train_data.inputs.shape[1]], 'inputs')
targets = tf.placeholder(tf.float32, [None, train_data.num_classes], 'targets')
num_hidden = 200

with tf.name_scope('fc-layer-1'):
    hidden_1 = fully_connected_layer(inputs, train_data.inputs.shape[1], num_hidden)
with tf.name_scope('output-layer'):
    outputs = fully_connected_layer(hidden_1, num_hidden, train_data.num_classes, tf
    .identity)

with tf.name_scope('error'):
    error = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(outputs, targets)
    )
with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(outputs, 1), tf.argmax(
    targets, 1)),
tf.float32))

with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer().minimize(error)

init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
    sess.run(init)
    for e in range(10):
        running_error = 0.
        running_accuracy = 0.
        for input_batch, target_batch in train_data:
            _, batch_error, batch_acc = sess.run(
                [train_step, error, accuracy],
                feed_dict={inputs: input_batch, targets: target_batch})
            running_error += batch_error
            running_accuracy += batch_acc
        running_error /= train_data.num_batches
        running_accuracy /= train_data.num_batches
        print('End of epoch {0:02d}: err(train)={1:.2f} acc(train)={2:.2f}'
            .format(e + 1, running_error, running_accuracy))
        if (e + 1) % 5 == 0:
            valid_error = 0.
            valid_accuracy = 0.
            for input_batch, target_batch in valid_data:
                batch_error, batch_acc = sess.run(
                    [error, accuracy],
            feed_dict={inputs: input_batch, targets: target_batch})
            valid_error += batch_error
        valid_accuracy += batch_acc
        valid_error /= valid_data.num_batches
        valid_accuracy /= valid_data.num_batches
        print('                err(valid)={0:.2f} acc(valid)={1:.2f}'
            .format(valid_error, valid_accuracy))
```

10.2 Batch normalization function

```
# http://r2rt.com/implementing-batch-normalization-in-tensorflow.html
def fully_connected_layer_batch_norm(inputs, input_dim, output_dim, nonlinearity=tf.
    nn.relu, epsilon=1e-6):
    weights = tf.Variable(
    tf.truncated_normal(
        [input_dim, output_dim], stddev=2. / (input_dim + output_dim)**0.5),
            'weights')
    z1_BN = tf.matmul(inputs, weights)
    batch_mean1, batch_var1 = tf.nn.moments(z1_BN, [0])
    z1_hat = (z1_BN - batch_mean1) / tf.sqrt(batch_var1 + epsilon)
    scale1 = tf.Variable(tf.ones([output_dim]))
    beta1 = tf.Variable(tf.zeros([output_dim]))
    BN1 = scale1 * z1_hat + beta1
    outputs = nonlinearity(BN1)
    return outputs
```