1)
between input layer and hidden layer 1 = 4 x 5 = 20
between hidden layer 1 and hidden layer 2 = 5 x 3 = 15
between hidden layer 2 and output = 3 x 1 = 3
20 + 15 + 3 = 38
38 trainable parameters.

2)

```
# TODO :: define the cross entorpy computation graph in tensorflow; expect 10-15 lines of code (Requirement : create your own graph with tf.Graph an run yo
# use placeholder to define variable instead of tf.Variable)

graph3 = tf.Graph()
with graph3.as_default():

    pos1 = tf.constant(1, dtype=tf.float32)
    neg1 = tf.constant(-1, dtype=tf.float32)
    y = tf.placeholder(tf.float32, name = "y")
    p = tf.placeholder(tf.float32, name = "p")

    part1 = tf.multiply(y, tf.log(p))
    part2 = tf.multiply(tf.subtract(pos1, y), tf.log(tf.subtract(pos1, p)))
    tf.multiply(neg1, tf.add(part1, part2))
```

Linear Regression

Using the Normal Equation

```
import numpy as np
from sklearn.datasets import fetch_california_housing

reset_graph()

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)

# TODO :: write down the normal equation, for more detail of the normal equation, you can refer to http://mlwiki.org/index.php/Normal_Equation
# hint : you may want to use tf.matrix_inverse, tf.matrix_inverse and tf.matmul
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT,X)),XT),y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

```
X = housing_data_plus_bias
y = housing.target.reshape(-1, 1)
# TODO :: implement the same normal equation with numpy
# hint : you may want to use np.linalg.inv
theta_numpy=np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

print(theta_numpy)
```

```
from sklearn.linear_model import LinearRegression
# TODO :: define the linear regression model and fit the training data. the model name should be lin_reg
lin_reg =LinearRegression().fit(housing.data, housing.target.reshape(-1, 1))

print(np.r_[lin_reg.intercept_.reshape(-1, 1), lin_reg.coef_.T])
```

```
mse = tf.reduce_mean(tf.square(error), name="mse")
```

In [17]:
```
# TODO :: define the GradientDescentOptimizer and call minimize on the optimizer, the result should be named as training_op; you can refer to t
training_op = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(mse)
```

```
# TODO :: repeat the same procedure this time use the MomentumOptimizer, you can refer to the tensorflow documentation : https://
tf.train.MomentumOptimizer(learning_rate,0.01).minimize(mse, global_step=tf.Variable(0, trainable=False))
```

```
<tf.Operation 'Momentum' type=AssignAdd>
```

3)

Affine layer: foward

Complete the following code cell to implement the forward propagation for each layer, later you will call this function for each layer when you do forward propagation.

```
: def affine_forward(x, theta, theta0):
    """
      Computes the forward pass for an affine (fully-connected) layer.

      The input x has shape (m, d_1, ..., d_k) and contains a minibatch of m
      examples, where each example x[i] has shape (d_1, ..., d_k). We will
      reshape each input into a vector of dimension d = d_1 * ... * d_k, and
      then transform it to an output vector of dimension h.

      Inputs:
      - x: A numpy array containing input data, of shape (m, d_1, ..., d_k)
      - theta: A numpy array of weights, of shape (d, h)
      - theta0: A numpy array of biases, of shape (h,)

      Returns a tuple of:
      - out: output, of shape (m, h)
      - cache: (x, theta, theta0)
    """
    out = None
    #############################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You     #
    # will need to reshape the input into rows.                                 #
    #############################################################################
    # 2 lines of code expected

    out = np.dot(x.reshape(x.shape[0], np.prod(x.shape[1:])), theta) + theta0


    #############################################################################
    #                             END OF YOUR CODE                              #
    #############################################################################
    cache = (x, theta, theta0)
    return out, cache
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking. The function will be called by each layer when you do backward propagation.

```
]: def affine_backward(dout, cache):
    """
      Computes the backward pass for an affine layer.

      Inputs:
      - dout: Upstream derivative, of shape (m, h)
      - cache: Tuple of:
        - x: Input data, of shape (m, d_1, ... d_k)
        - theta: Weights, of shape (d,h)
        - theta0: biases, of shape (h,)

      Returns a tuple of:
      - dx: Gradient with respect to x, of shape (m, d1, ..., d_k)
      - dtheta: Gradient with respect to theta, of shape (d, h)
      - dtheta0: Gradient with respect to theta0, of shape (h,)
    """
    x, theta, theta0 = cache
    dx, dtheta, dtheta0 = None, None, None
    #############################################################################
    # TODO: Implement the affine backward pass.                                 #
    #############################################################################
    # Hint: do not forget to reshape x into (m,d) form
    # 4-5 lines of code expected

    x, theta, theta0 = cache[0], cache[1], cache[2]

    new = x.reshape(x.shape[0], np.prod(x.shape[1:]))
    dtheta = new.T.dot(dout)
    dx = dout.dot(theta.T).reshape(x.shape)
    dtheta0 = np.sum(dout, axis=0)
    #############################################################################
    #                             END OF YOUR CODE                              #
    #############################################################################
    return dx, dtheta, dtheta0
```

```python
def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    # 1 line of code expected
    out = np.maximum(0, x)

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache
```

```python
def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    dx, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    # 1 line of code expected. Hint: use np.where
    # newx: m,d

    dx = np.where(np.maximum(0, cache) > 0, 1, 0) * dout
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx
```

```python
        self.reg = reg

        ############################################################################
        # TODO: Initialize the weights and biases of the two-layer net. Weights    #
        # should be initialized from a zero-mean Gaussian with stdev equal to      #
        # weight_scale, and biases should be initialized to zero. All weights and  #
        # biases should be stored in the dictionary self.params, with first layer  #
        # weights and biases using the keys 'theta1' and 'theta1_0' and second     #
        # layer weights and biases using the keys 'theta2' and 'theta2_0.          #
        # theta1 has shape (input_dim,hidden-dim), theta1_0 shape is (hidden_dim,) #
        # theta2 shape is (hidden_dim,num_classes), theta2_0 shape is (num_classes,)#
        ############################################################################
        # 4 lines of code expected
        self.params['theta1'] = np.random.normal(scale=weight_scale, size=(input_dim, hidden_dim))
        self.params['theta1_0'] = np.zeros(hidden_dim)

        self.params['theta2'] = np.random.normal(scale=weight_scale, size=(hidden_dim, num_classes))
        self.params['theta2_0'] = np.zeros(num_classes)

        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################
```

```python
        names to gradients of the loss with respect to those parameters.
        """
        scores = None
        ############################################################################
        # TODO: Implement the forward pass for the two-layer net, computing the    #
        # class scores for X and storing them in the scores variable.              #
        ############################################################################
        # Hint: unpack the weight parameters from self.params
        # then calculate output of two layer network using functions defined before
        # 3 lines of code expected
        out_1, cache1 = affine_relu_forward(X, self.params['theta1'], self.params['theta1_0'])
        out_2, cache2 = affine_forward(out_1, self.params['theta2'], self.params['theta2_0'])
        scores = out_2

        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################
```

```
    ############################################################################
    # TODO: Implement the backward pass for the two-layer net. Store the loss  #
    # in the loss variable and gradients in the grads dictionary. Compute data #
    # loss using softmax, and make sure that grads[k] holds the gradients for  #
    # self.params[k]. Don't forget to add L2 regularization!                   #
    #                                                                          #
    # NOTE: To ensure that your implementation matches ours and you pass the   #
    # automated tests, make sure that your L2 regularization includes a factor #
    # of 0.5 to simplify the expression for the gradient.                      #
    ############################################################################

    # 4-8 lines of code expected

    loss, dscores = softmax_loss(scores, y)
    loss += (self.reg*np.sum(self.params['theta1']**2) + self.reg*np.sum(self.params['theta2']**2)) / 2

    dx_2, grads['theta2'], grads['theta2_0'] = affine_backward(dscores, cache2)
    dx_1, grads['theta1'], grads['theta1_0'] = affine_relu_backward(dx_2, cache1)

    grads['theta2'] += self.reg*self.params['theta2']
    grads['theta1'] += self.reg*self.params['theta1']
```

```
  ############################################################################
  # TODO: Initialize the weights and biases of the three-layer net. Weights     #
  # should be initialized from a zero-mean Gaussian with stdev equal to          #
  # weight_scale, and biases should be initialized to zero. All weights and      #
  # biases should be stored in the dictionary self.params, with first layer      #
  # weights and biases using the keys 'theta1' and 'theta1_0' and second         #
  # layer weights and biases using the keys 'theta2' and 'theta2_0. the third    #
  # layer weights and biases using the keys 'theta3' and 'theta3_0'.             #
  # theta1 has shape (input_dim,hidden-dim-1), theta1_0 shape is (hidden-dim-1,)  #
  # theta2 shape is (hidden_dim-1,hidden_dim-2), theta2_0 shape is (hidden-dim-2,)#
  # theta3 shape is (hidden_dim-2, num_classes), theta3_0 shape is (num_classes)  #
  ############################################################################
  # 6 lines of code expected
  self.params['theta1'] = np.random.normal(scale=weight_scale, size=(input_dim, hidden_dim_1))
  self.params['theta1_0'] = np.zeros(hidden_dim_1)

  self.params['theta2'] = np.random.normal(scale=weight_scale, size=(hidden_dim_1, hidden_dim_2))
  self.params['theta2_0'] = np.zeros(hidden_dim_2)

  self.params['theta3'] = np.random.normal(scale=weight_scale, size=(hidden_dim_2, num_classes))
  self.params['theta3_0'] = np.zeros(num_classes)

  ############################################################################
  #                             END OF YOUR CODE                             #
  ############################################################################
```

```
      scores = None
      ############################################################################
      # TODO: Implement the forward pass for the three-layer net, computing the  #
      # class scores for X and storing them in the scores variable.              #
      ############################################################################
      # Hint: unpack the weight parameters from self.params
      # then calculate output of two layer network using functions defined before
      # 3 lines of code expected
      out1, cache1 = affine_relu_forward(X, self.params['theta1'], self.params['theta1_0'])
      out2, cache2 = affine_relu_forward(out1, self.params['theta2'], self.params['theta2_0'])
      out3, cache3 = affine_forward(out2, self.params['theta3'], self.params['theta3_0'])
      scores = out3

      ############################################################################
      #                             END OF YOUR CODE                             #
      ############################################################################
```

Solver

Open the file `solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

```
model = TwoLayerNet()
sgd_solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
# Get the CIFAR-10 data broken up into train, validation and test sets

sgd_solver = Solver(model, data, update_rule='sgd', optim_config={'learning_rate': 1e-3},
                    lr_decay=0.95, num_epochs=7, batch_size=100, print_every=100)
sgd_solver.train()
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

```
            print('training accuracy : ', acc)
            ############################################################################
            # TODO: here we evaluate the training accuracy, you need you evaluate the    #
            # validation accuracy as well and save it in a variable named val_acc        #
            ############################################################################

            val_acc = sess.run(accuracy, feed_dict = {x: data['X_val'][:batch_size], y: data['y_val'][:batch_size]})

            ############################################################################
            # End your code here                                                         #
            ############################################################################
```

Training A Three-Layer Neural Network on MINIST Dataset

You've seen how to train and evaluate your neural network model on CIFAR10 dataset, it's your turn to train a three-layer Neural Network on MINIST Dataset. Implement the model with Tensorflow or Keras. Tune the parameters to find the best hyperparameters for your model.

```
from keras.datasets import mnist
from keras.layers.core import Dense, Dropout, Activation
from keras.utils import np_utils
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("x_test shape", X_test.shape)
print("y_test shape", y_test.shape)
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
x_test shape (10000, 28, 28)
y_test shape (10000,)
```

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255

print("Training matrix shape", x_train.shape)
print("Testing matrix shape", X_test.shape)
```

```
Training matrix shape (60000, 784)
Testing matrix shape (10000, 784)
```

```
nb_classes = 10

Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

```
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 512)               401920
_________________________________________________________________
activation_10 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_8 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_14 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_11 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_12 (Activation)   (None, 10)                0
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

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

```
model.fit(X_train, Y_train, batch_size=128, epochs=8, verbose=1)
```

```
score = model.evaluate(X_test, Y_test)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 26us/step
Test score: 0.10113688079406265
Test accuracy: 0.9842000007629395
```

Question: What did you discover with hyperparameter tuning?

The batch size is how many samples we use for one update to the model weights. Epochs is how many times we want to iterate on the whole training set. Generally with smaller batch, the more unstable the stochastic updates are. Using too high of epochs will cause the model to overfit.