Lab 05: Softmax

Dr. Vassilis Diakoloukas
Dr. Vassilis Tsiaras

SoftMax vs Logistic Function

- Linear Regression predicts a continuous value (real number)
- Logistic Regression produces binary output
  - Yes – No or 0 – 1
  - Uses the logistic function (sigmoid):

$$f(x) = \frac{1}{1 + e^{-x}}$$

  - Input x can be the linear regression prediction
  - Output is the probability of the answer being “yes”
  - Decide “Yes” or “No” based on a threshold (typically 0.5)
- Use softmax function is a generalization of logistic function

$$f(x)_c = \frac{e^{-x_c}}{\sum_{j=0}^{C-1} e^{-x_j}}, \qquad c = 0 \cdots C - 1$$

  - Returns the probability distribution over mutually exclusive output classes
  - Used to decide among multiple choices $c = 0 \cdots C - 1$
  - Often used as the activation function of the output layer of a classifier

SoftMax for IRIS classification

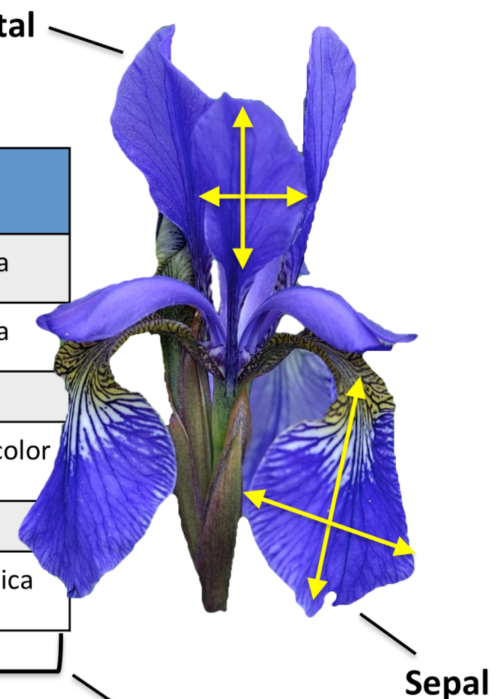
- **IRIS**: One of the most popular datasets for Machine Learning
  - Download from: https://archive.ics.uci.edu/ml/datasets/Iris
- **Goal**: Classify an Iris flower into one of the 3 classes:
  - Iris Setosa
  - Iris Versicolour
  - Iris Virginica
- **Attributes**:
  - Sepal length
  - Sepal width
  - Petal length
  - Petal width

Samples
(instances, observations)

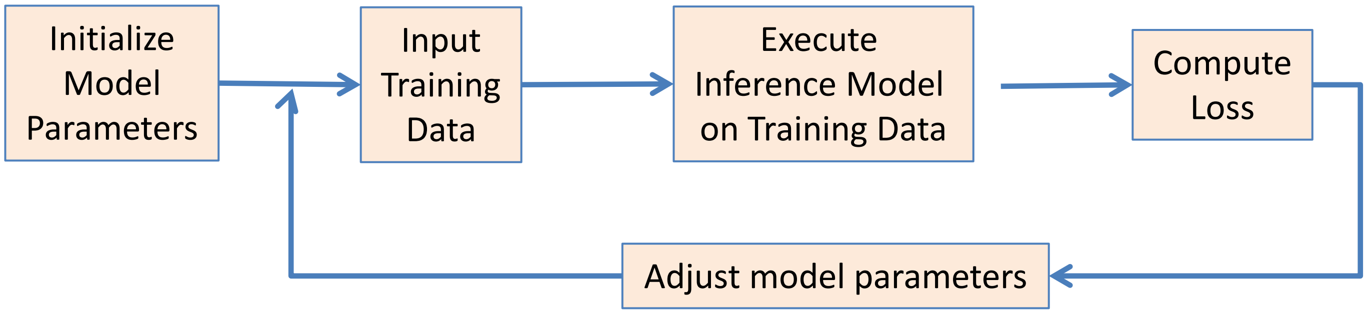
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Features
(attributes, measurements, dimensions)

Class labels
(targets)



Using functions to define the training loop



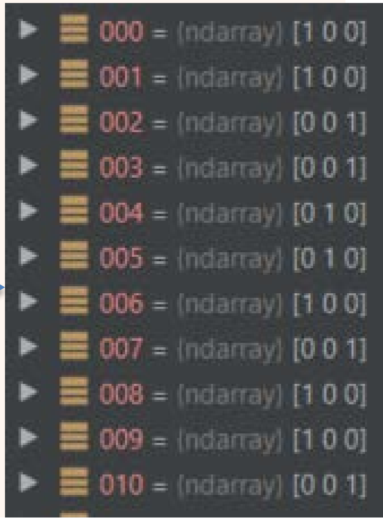

```python
import tensorflow as tf
import pandas as pd
import numpy as np
# initialize variables/model parameters

# define the training loop operations
def input_data():
    # read input data and generate features X and expected outputs Y

def inference(X):
    # compute inference model over data X and return the result

def loss(X, Y):
    # compute loss over training data X and expected outputs Y

def train(total_loss):
    # train / adjust model parameters according to computed total loss

def evaluate(sess, X, Y):
    # evaluate the trained model
```

Input Data

- Read the csv dataset
- Generate training-test feature vectors and the expected output

```
def input_data():
    input_file = 'Iris.csv'
    IRIS_fname = os.path.dirname(__file__) + "/data/" + input_file
    iris = pd.read_csv(IRIS_fname)

    X = iris.drop(labels=['Id', 'Species'], axis=1).values
    X.astype(np.float32)
    #Convert categorical data to one-hot encoding
    Y = pd.get_dummies(iris['Species']).values

    #Create 80% training and 20% test
    train_index = np.random.choice(len(X), int(round(len(X) * 0.8)), replace=False)
    test_index = np.array(list(set(range(len(X))) - set(train_index)))
    Y_train = Y[train_index]
    Y_test = Y[test_index]
    X_train = normalize(X[train_index])  # Normalize training sets
    X_test = normalize(X[test_index]) # Normalize test sets

    return X_train, Y_train, X_test, Y_test

# min-max mormalization
def normalize(X):
    col_max = np.max(X, axis=0)
    col_min = np.min(X, axis=0)
    normX = np.divide(X - col_min, col_max - col_min)
    return normX
```

Sample of output Y



SoftMax Initialization

- SoftMax computes C outputs instead of one (one-hot encoding)
  - We need C weight groups, one for each output
    - Initialize a weight matrix variable $W \in \mathbb{R}^{d \times C}$, where d is the number of features and C the number of classes
- **IRIS Initialization**: Weight matrix has a 4×3 dimension

```python
import tensorflow as tf
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
import os

# initialize variables/model parameters
n_dim = 4 #Feature Vector dimension
n_classes = 3 #Number of classes
X = tf.placeholder(dtype=tf.float32, shape=[None, n_dim])
Y = tf.placeholder(dtype=tf.int32, shape=(None, n_classes))
# Weights form a matrix, of a feature column per output class.
W = tf.Variable(tf.zeros([n_dim, n_classes]), name="weights", dtype=tf.float32)
# Biases, one per output class.
b = tf.Variable(tf.zeros([n_classes]), name="bias", dtype=tf.float32)
```

SoftMax Loss

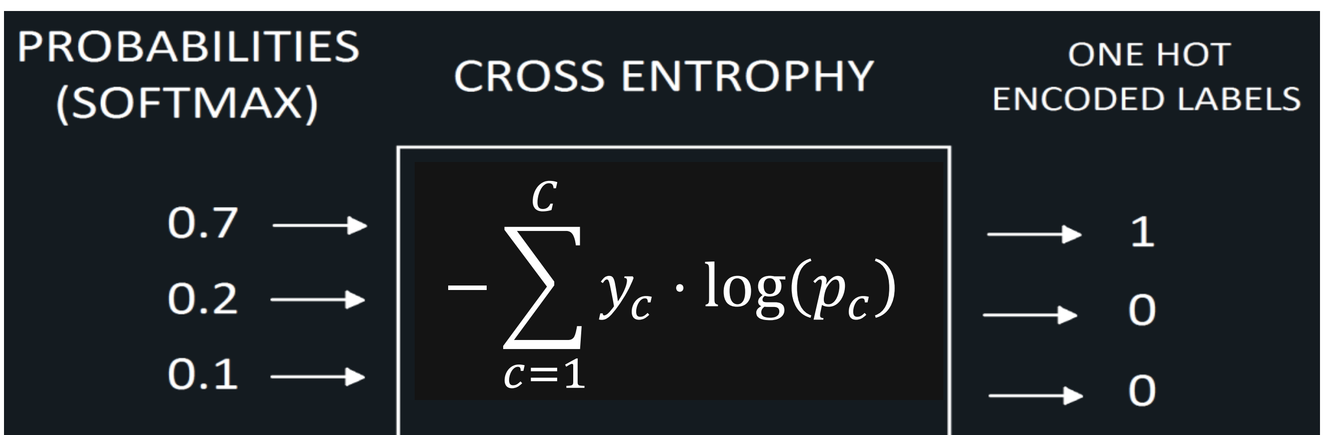
- Use the cross-entropy as loss function, appropriately adapted for multiple classes
  - For a single training sample i with predicted output probability p_c for the class c

$$\mathcal{L}_i = -\sum_{c=1}^{C} y_c \cdot \log(p_c) \qquad \text{or}$$

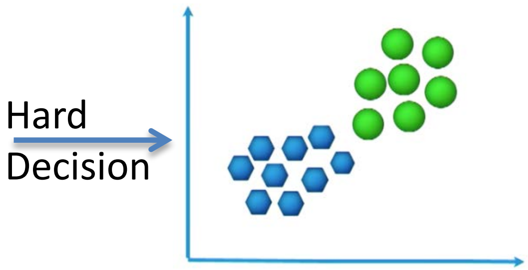
$$\mathcal{L}_i = -\sum_{c=1}^{C} y_c \cdot \log(p_c) + (1 - y_c) \cdot \log(1 - p_c)$$

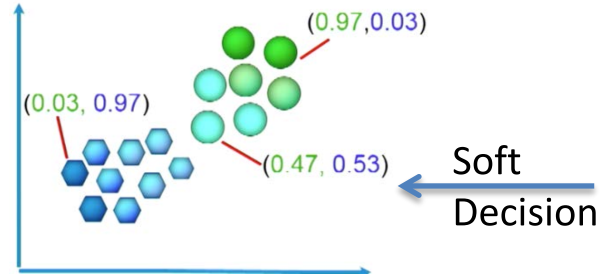
  - Total loss among the total number N of training samples:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{c_i} \cdot \log(p_c)$$



SoftMax Cross-Entropy Versions





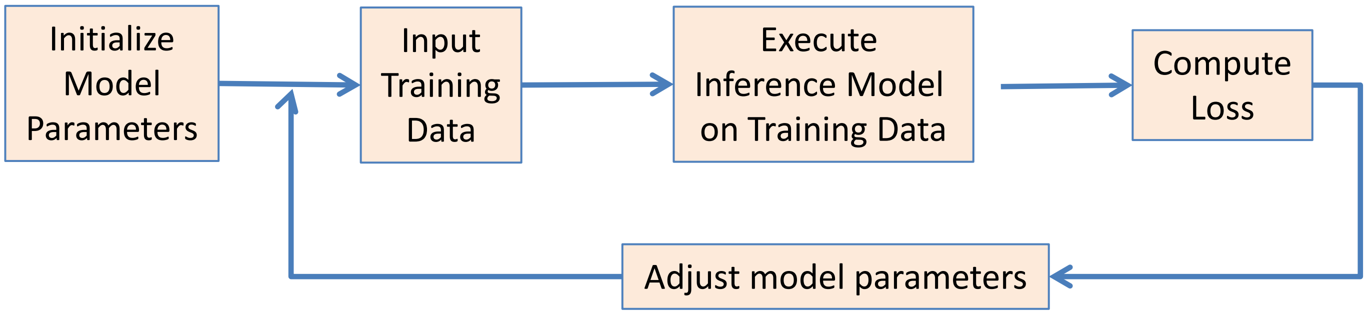
- Softmax cross-entropy function versions:
  – **Hard Decision**: For training sets with a single class value per example
    - For example: an image represents a dog or a truck but not both
    - **Labels**=one number per example (0 … num_classes – 1)
    - **Logits**=Softmax probabilities per class

```
tf.nn.sparse_softmax_cross_entropy_with_logits(logits, labels)
```

  – **Soft Decision**: When training examples have a probability to belong to each class
    - For example: an image represents 70% a dog and 30% a truck
    - **Labels**=one hot encoding or probabilities per class
    - **Logits**=Softmax probabilities per class
    - The dimensions of **Logits** and **Labels** are the same

```
tf.nn.softmax_cross_entropy_with_logits_v2(logits, labels)
```

Other Main Functions




```python
# former inference is now used for combining inputs
def combine_inputs(X):
    return tf.matmul(X, W) + b

def inference(X):
    return tf.nn.softmax(combine_inputs(X))

def loss(X, Y):
    Yhat = combine_inputs(X)
    SoftCE = tf.nn.softmax_cross_entropy_with_logits_v2(logits=Yhat, labels=Y)
    return tf.reduce_mean(SoftCE)

def train(total_loss):
    learning_rate = 0.1
    opt = tf.train.GradientDescentOptimizer(learning_rate)
    goal = opt.minimize(total_loss)
    return goal
```

Evaluate Function

- Redefine the evaluation function appropriately

```python
def evaluate(sess, Xtest, Ytest):
    Yhat = inference(X)
    #Return the index with the largest value across axis
    Ypredict = tf.argmax(Yhat, axis=1, output_type=tf.int32) #in [0,1,2]
    Ycorrect = tf.argmax(Y, axis=1, output_type=tf.int32) #in [0,1,2]
    #Cast a boolean tensor to float32
    correct = tf.cast(tf.equal(Ypredict, Ycorrect), tf.float32)
    accuracy_graph = tf.reduce_mean(correct)
    accuracy = sess.run(accuracy_graph, feed_dict={X: Xtest, Y: Ytest})
    return accuracy
```

Helper Functions

- Get the data in random order ➔reshuffle(X,Y)
- Get the data in batches ➔ read_next_batch()

```
# Shuffle the training data
def reshuffle(X, Y):
    data_index = 0
    NData = len(X)
    perm_indices = np.arange(NData)
    np.random.shuffle(perm_indices)
    X = X[perm_indices]
    Y = Y[perm_indices]
    return X, Y

# Read next training batch
def read_next_batch(X, Y, batch_size, train_index=0):
    n_train_examples = len(X)
    if train_index + batch_size < n_train_examples:
        X_train_batch = X[train_index:train_index + batch_size]
        Y_train_batch = Y[train_index:train_index + batch_size]
        train_index = train_index + batch_size
        return X_train_batch, Y_train_batch, train_index
    else:
        return None, None, None
```

Session Execution

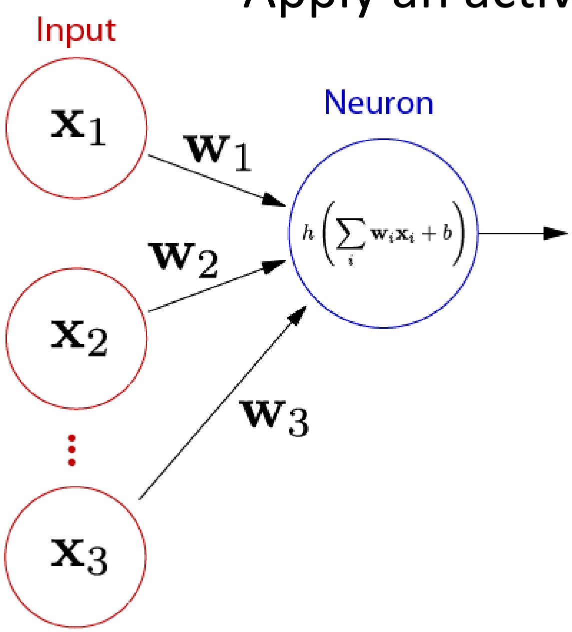
```
batch_size = 30  # Training batch size
Xtrain, Ytrain, Xtest, Ytest = input_data()  # Get the data samples
init = tf.global_variables_initializer()
with tf.Session() as sess:
  sess.run(init) # variables initialization
  total_loss = loss(X, Y)
  train_op = train(total_loss)
  # actual training loop
  num_epochs = 200
  for epoch in range(num_epochs):
    Xtrain, Ytrain = reshuffle(Xtrain, Ytrain)
    train_index = 0
    train_loss = 0
    loss_trace_list = []
    Xtrain_batch, Ytrain_batch, train_index = read_next_batch(Xtrain, Ytrain, batch_size,
                                                                            train_index)
    while Xtrain_batch is not None:
      temp_loss, _ = sess.run([total_loss, train_op], feed_dict={X:Xtrain_batch, Y:Ytrain_batch})
      loss_trace_list.append(temp_loss)
      train_loss += temp_loss
      Xtrain_batch, Ytrain_batch, train_index = read_next_batch(Xtrain, Ytrain, batch_size,
                                                                            train_index)

    # see how the loss decreases
    if epoch % 10 == 0:
       train_acc = evaluate(sess, Xtrain, Ytrain)
       test_acc = evaluate(sess, Xtest, Ytest)
       print('epoch: {:4d} loss: {:5f} train_acc: {:5f} test_acc: {:5f}'.format(epoch + 1,
                                                              train_loss, train_acc, test_acc))
```

Results

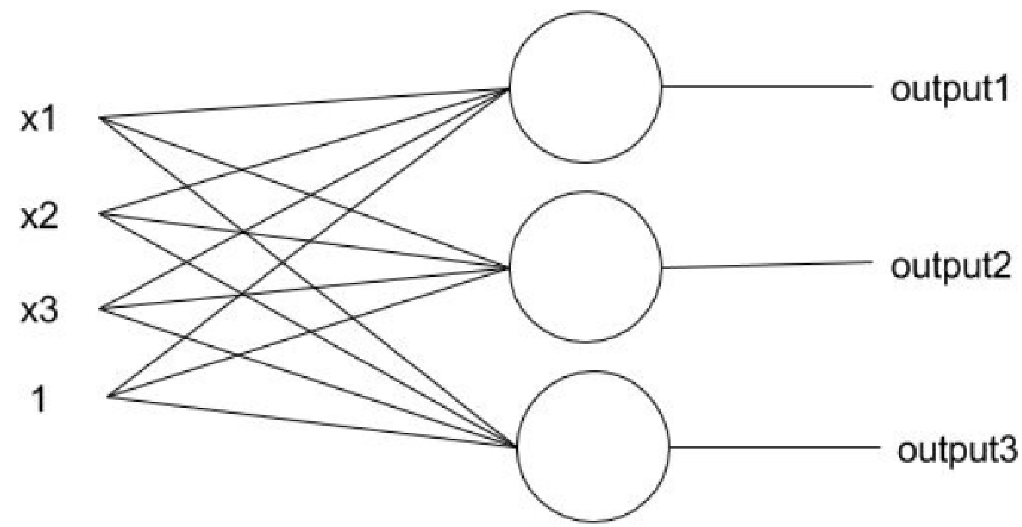
```
epoch:    1 loss: 3.274638 train_acc: 0.666667 test_acc: 0.600000
epoch:   11 loss: 2.805722 train_acc: 0.683333 test_acc: 0.600000
epoch:   21 loss: 2.493774 train_acc: 0.683333 test_acc: 0.600000
epoch:   31 loss: 2.345563 train_acc: 0.700000 test_acc: 0.666667
epoch:   41 loss: 2.075058 train_acc: 0.725000 test_acc: 0.666667
epoch:   51 loss: 1.964237 train_acc: 0.741667 test_acc: 0.700000
epoch:   61 loss: 1.899966 train_acc: 0.791667 test_acc: 0.700000
epoch:   71 loss: 1.719347 train_acc: 0.800000 test_acc: 0.733333
epoch:   81 loss: 1.696884 train_acc: 0.816667 test_acc: 0.800000
epoch:   91 loss: 1.690177 train_acc: 0.850000 test_acc: 0.866667
epoch:  101 loss: 1.534325 train_acc: 0.816667 test_acc: 0.766667
epoch:  111 loss: 1.592387 train_acc: 0.858333 test_acc: 0.866667
epoch:  121 loss: 1.515490 train_acc: 0.833333 test_acc: 0.866667
epoch:  131 loss: 1.385782 train_acc: 0.850000 test_acc: 0.866667
epoch:  141 loss: 1.408420 train_acc: 0.866667 test_acc: 0.866667
epoch:  151 loss: 1.414920 train_acc: 0.866667 test_acc: 0.866667
epoch:  161 loss: 1.323434 train_acc: 0.891667 test_acc: 0.866667
epoch:  171 loss: 1.286453 train_acc: 0.850000 test_acc: 0.866667
epoch:  181 loss: 1.345704 train_acc: 0.891667 test_acc: 0.866667
epoch:  191 loss: 1.252267 train_acc: 0.900000 test_acc: 0.866667
```

From simple NNs to Multi-layer neural networks

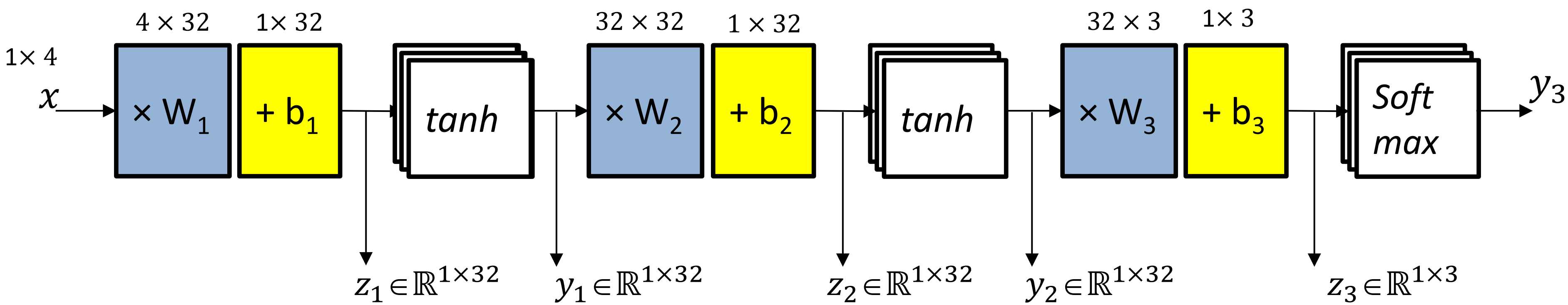
- Linear and logistic regression models are single neurons
  - Perform a weighted sum of the input features
  - Apply an activation function to calculate output



- SoftMax classification is a network of C neurons (one for each output class)



Define a multi-layer network



- Network characteristics:
  - Use two hidden layers with 32 nodes each
  - Use ***tanh*** as activation function of the first two layers
  - Use ***SoftMax*** as the activation function of the output layer

Input Data:

Output is the class value instead of one-hot

- Generate training-test feature vectors and the expected output

```
def input_data():
    input_file = 'Iris.csv'
    IRIS_fname = os.path.dirname(__file__) + "/data/" + input_file
    iris = pd.read_csv(IRIS_fname)
    #Replace categorical labels with numerical values
    iris.Species = iris.Species.replace(to_replace=['Iris-setosa', 'Iris-versicolor', 'Iris-
virginica'], value=[0, 1, 2])
    X = iris.drop(labels=['Id', 'Species'], axis=1).values
    X.astype(np.float32)
    Y = iris.Species.values.astype(np.int32)
    #Create 80% training and 20% test
    train_index = np.random.choice(len(X), int(round(len(X) * 0.8)), replace=False)
    test_index = np.array(list(set(range(len(X))) - set(train_index)))
    Y_train = Y[train_index]
    Y_test = Y[test_index]
    X_train = normalize(X[train_index])  # Normalize training sets
    X_test = normalize(X[test_index]) # Normalize test sets

    return X_train, Y_train, X_test, Y_test

# min-max mormalization
def normalize(X):
    col_max = np.max(X, axis=0)
    col_min = np.min(X, axis=0)
    normX = np.divide(X - col_min, col_max - col_min)
    return normX
```

Sample of output Y

Network Initialization

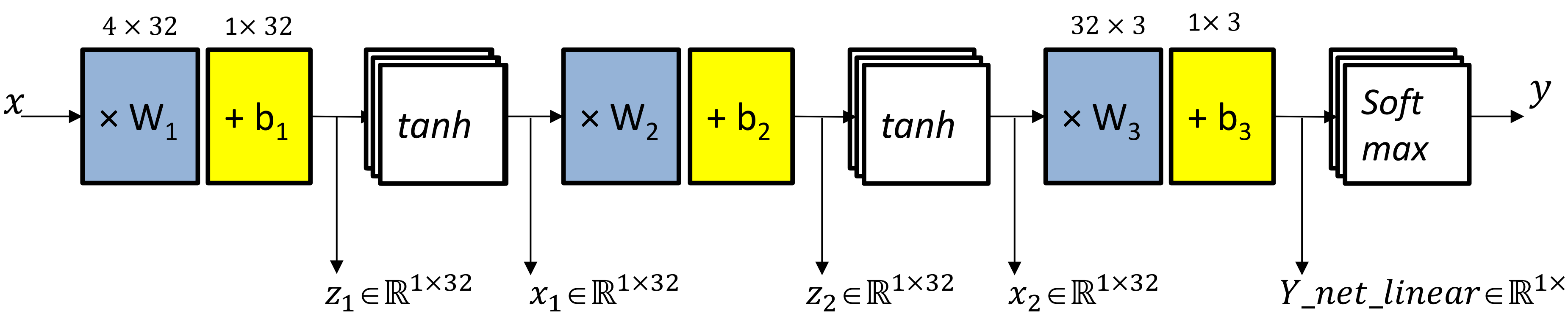
- Define Weight and bias Variables for each layer
- Use random values to initialize them

```
import tensorflow as tf
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
import os

n_dim = 4 #Feature Vector dimension
n_classes = 3 #Number of classes
n_hidden = 32 #Number of Hidden nodes
X = tf.placeholder(dtype=tf.float32, shape=[None, n_dim])
Y = tf.placeholder(dtype=tf.int32, shape=(None, ))
# Weights form a matrix, of a feature column per output class.
W1 = tf.Variable(tf.random_normal(shape=[n_dim, n_hidden]), dtype=tf.float32)
b1 = tf.Variable(tf.random_normal(shape=(n_hidden,)), dtype=tf.float32)

W2 = tf.Variable(tf.random_normal(shape=[n_hidden, n_hidden]), dtype=tf.float32)
b2 = tf.Variable(tf.random_normal(shape=(n_hidden,)), dtype=tf.float32)

W3 = tf.Variable(tf.random_normal(shape=[n_hidden, n_classes]), dtype=tf.float32)
b3 = tf.Variable(tf.random_normal(shape=(n_classes,)), dtype=tf.float32)
```

Network Inference Functions

- Redefine the combine_inputs function appropriately

```
# Get the linear output of the network combining inputs
def combine_inputs(X):
    X1 = tf.nn.tanh(tf.matmul(X, W1) + b1)
    X2 = tf.nn.tanh(tf.matmul(X1, W2) + b2)
    Y_net_linear = tf.matmul(X2, W3) + b3
    return Y_net_linear
```




```
def inference(X):
    return tf.nn.softmax(combine_inputs(X))
```

Loss Function

- Use the sparse version instead:

`tf.nn.`**`sparse`**`_softmax_cross_entropy_with_logits`

```
def loss(X, Y):
    Yhat = combine_inputs(X)
    SoftCE = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=Yhat, labels=Y)
    return tf.reduce_mean(SoftCE)
```

Evaluate Function

- Redefine the evaluation function appropriately

```python
def evaluate(sess, Xtest, Ytest):
    Yhat = inference(X)
    #Return the index with the largest value across axis
    Ypredict = tf.argmax(Yhat, axis=1, output_type=tf.int32) #in [0,1,2]
    #Cast a boolean tensor to float32
    correct = tf.cast(tf.equal(Ypredict, Y), tf.float32)
    accuracy_graph = tf.reduce_mean(correct)
    accuracy = sess.run(accuracy_graph, feed_dict={X: Xtest, Y: Ytest})
    return accuracy
```

Session Execution

```
batch_size = 30  # Training batch size
Xtrain, Ytrain, Xtest, Ytest = input_data()  # Get the data samples
init = tf.global_variables_initializer()
with tf.Session() as sess:
  sess.run(init) # variables initialization
  total_loss = loss(X, Y)
  train_op = train(total_loss)
  # actual training loop
  num_epochs = 200
  for epoch in range(num_epochs):
    Xtrain, Ytrain = reshuffle(Xtrain, Ytrain)
    train_index = 0
    train_loss = 0
    loss_trace_list = []
    Xtrain_batch, Ytrain_batch, train_index = read_next_batch(Xtrain, Ytrain, batch_size,
                                                                            train_index)
    while Xtrain_batch is not None:
      temp_loss, _ = sess.run([total_loss, train_op], feed_dict={X:Xtrain_batch, Y:Ytrain_batch})
      loss_trace_list.append(temp_loss)
      train_loss += temp_loss
      Xtrain_batch, Ytrain_batch, train_index = read_next_batch(Xtrain, Ytrain, batch_size,
                                                                              train_index)

    # see how the loss decreases
    if epoch % 10 == 0:
       train_acc = evaluate(sess, Xtrain, Ytrain)
       test_acc = evaluate(sess, Xtest, Ytest)
       print('epoch: {:4d} loss: {:5f} train_acc: {:5f} test_acc: {:5f}'.format(epoch + 1,
                                                              train_loss, train_acc, test_acc))
```

Results

```
epoch:   1 loss: 8.095600 train_acc: 0.925000 test_acc: 0.866667
epoch:  11 loss: 0.360769 train_acc: 0.958333 test_acc: 0.933333
epoch:  21 loss: 0.186995 train_acc: 0.983333 test_acc: 0.933333
epoch:  31 loss: 0.212801 train_acc: 0.966667 test_acc: 0.933333
epoch:  41 loss: 0.251649 train_acc: 0.983333 test_acc: 0.933333
epoch:  51 loss: 0.225448 train_acc: 0.983333 test_acc: 0.900000
epoch:  61 loss: 0.186641 train_acc: 0.975000 test_acc: 0.933333
epoch:  71 loss: 0.205213 train_acc: 0.983333 test_acc: 0.900000
epoch:  81 loss: 0.072361 train_acc: 0.983333 test_acc: 0.900000
epoch:  91 loss: 0.098235 train_acc: 0.991667 test_acc: 0.933333
epoch: 101 loss: 0.157500 train_acc: 0.983333 test_acc: 0.933333
epoch: 111 loss: 0.152627 train_acc: 0.991667 test_acc: 0.933333
epoch: 121 loss: 0.158958 train_acc: 0.991667 test_acc: 0.933333
epoch: 131 loss: 0.076640 train_acc: 0.983333 test_acc: 0.900000
epoch: 141 loss: 0.059961 train_acc: 0.991667 test_acc: 0.933333
epoch: 151 loss: 0.186251 train_acc: 0.991667 test_acc: 0.933333
epoch: 161 loss: 0.053375 train_acc: 0.991667 test_acc: 0.933333
epoch: 171 loss: 0.155159 train_acc: 0.991667 test_acc: 0.933333
epoch: 181 loss: 0.145963 train_acc: 0.983333 test_acc: 0.933333
epoch: 191 loss: 0.157552 train_acc: 0.991667 test_acc: 0.933333
```

Exercise: Implement RNN networks

- Define network architecture consisting of one or more layers
- Define the cell type (i.e. GRU or LSTM)
- Define possible dropouts between the ayers
- Build RNNs from the cells that wrap each other

```
num_units = 64
num_layers = 3
shape = tf.shape(X)
X=tf.reshape(X, (1, shape[0], shape[1]))
cells = []
for _ in range(num_layers):
    cell = tf.contrib.rnn.GRUCell(64)
    cell = tf.contrib.rnn.BasicLSTMCell(num_units=64, reuse=tf.AUTO_REUSE)
    cell = tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob=1.0 - dropout)
    cells.append(cell)
cell = tf.contrib.rnn.MultiRNNCell(cells)
Y, state = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
Y = tf.reshape(Y, shape)
```