Artificial Neural Network 101

With Iris dataset and tensorflow

Jian Dai, PhD (daij12@gene.com)

Goal

- In genneral, understand the foundation of artificial neural network including model architecture, and training/optimization using Iris dataset as example and tensorflow as toolkit
- More specifically, **Exercise 1** compute gradient and weight update with single batch, and compare with the result of calling tensorflow for multinomial model to fully understand *gradient descent*
- **Exercise 2** compute gradient and weight update with single batch, and compare with the result of calling tensorflow for 1-hidden-layer model to fully understand *backpropagation*

Plan

- Iris datase, Tensorflow
- Multinomial regression (aka softmax regression)
  - Gradient descent
- 1-Hidden layer feedback forward neural network
  - Backprogagation
  - Tensor computation
- Some to-do for next steps:
  - Gradient descent with momentum

Iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set)

In the following we will use Iris dataset provided by the tensorflow tutorial (https://en.wikipedia.org/wiki/Iris_flower_data_set)

In [1]:
```
# Assume these two csv's 'iris_training.csv', 'iris_test.csv' are in the working dir
# Otherwise download from
# http://download.tensorflow.org/data/iris_training.csv
# http://download.tensorflow.org/data/iris_test.csv

import pandas as pd
training_df = pd.read_csv('iris_training.csv',skiprows=1,header=None)
training_features = training_df.iloc[:,0:4].values
training_labels = training_df.iloc[:,4].values
test_df = pd.read_csv('iris_test.csv',skiprows=1,header=None)
test_features = test_df.iloc[:,0:4].values
test_labels = test_df.iloc[:,4].values
print(training_df.describe())
```

```
                0           1           2           3           4
count  120.000000  120.000000  120.000000  120.000000  120.000000
mean     5.845000    3.065000    3.739167    1.196667    1.000000
std      0.868578    0.427156    1.822100    0.782039    0.840168
min      4.400000    2.000000    1.000000    0.100000    0.000000
25%      5.075000    2.800000    1.500000    0.300000    0.000000
50%      5.800000    3.000000    4.400000    1.300000    1.000000
75%      6.425000    3.300000    5.100000    1.800000    2.000000
max      7.900000    4.400000    6.900000    2.500000    2.000000
```

For R users, refer the following line for reading the data

```
read.csv('iris_training.csv',skip=1,header=FALSE)
```

In [2]:
```
# One-hot coding for class labels
from tensorflow.contrib.learn.python.learn.datasets.mnist import DataSet, dense_to_one_hot
training_labels_1h = dense_to_one_hot(training_labels,3)
training_dataset = DataSet(training_features, training_labels_1h,reshape=False)
test_labels_1h = dense_to_one_hot(test_labels,3)
test_dataset = DataSet(test_features, test_labels_1h,reshape=False)

print(training_dataset.num_examples)
print(test_dataset.num_examples)
```

```
120
30
```

Tensorflow

[https://www.tensorflow.org/ (https://www.tensorflow.org/)](https://www.tensorflow.org/)

In [3]:
```
import tensorflow as tf
```

Multinomial regression model

Here is a brief setup for a multinomial regression (aka softmax regression)

Introduce the multinomial logit as

$$\eta_k = x_\mu W_k^\mu$$

where x_μ^i the input with an implicit sample index i and the feature index μ including the *bias* term. For Iris data, k indexes 3 classes, μ indexes 4 predictors and the bias, and i indexes the batched sample. There is a summation over μ in the above expression understood. We keep a freedom to specify the summary on a pair of indices implicitly or explicitly. If the summation is implied, the convention is known as [*Einstein summation (http://mathworld.wolfram.com/EinsteinSummation.html)*](http://mathworld.wolfram.com/EinsteinSummation.html) which is commonly used in classical tensor analysis in mathematics and physics.

With logit, the conditional probability given data is

$$p_k = \frac{e^{\eta_k}}{Z}$$

where $Z = \sum_k e^{\eta_k}$ is known as *partition function*. Or if we specify the implicit sample index, $p_k^i = e^{\eta_k^i}/Z^i$. Please be cautious not to confuse superindex and power.

The loss function is defined as

$$l = -\left\langle \sum_{k=1}^{3} y_k \log p_k \right\rangle$$

where y the observed class with an implicit sample index i. In this formalism, y is expressed by *one-hot encoding*.

Here the average operation $\left\langle \dots \right\rangle$ can be specified differently in different context. For example,

$$\left\langle \dots \right\rangle = \int dP(x, y)$$

if the joint distribution is known, or

$$\left\langle \dots \right\rangle = \sum_i$$

for batch or mini-batch empirical loss, or just a single case in Stochastic Gradient Descent (https://en.wikipedia.org/wiki/Stochastic_gradient_descent). Also one can choose a normalization in the average. Commonly used normalization includes 2 for Deviance (https://en.wikipedia.org/wiki/Deviance_(statistics)) in classical statistical analysis, $1/N$ for categorical cross-entropy (https://en.wikipedia.org/wiki/Cross_entropy) in ML. In our case, we multiple a factor $\log 3$ such the *uninformative* prediction gives raise to $l = 1$. To restate for our practice and for Iris data

$$l = -\frac{1}{N \log 3} \sum_{i}^{N} \sum_{k=1}^{3} y_k^i \log p_k^i$$

where N is the sample size in a mini batch.

Note the constraints $\sum_k p_k = 1$ and $\sum_k y_k = 1$. And the loss can be transformed to a form which is easier to derive gradient

$$l = -\frac{1}{N \log 3} \sum_{i}^{N} \Big(\sum_{k=1}^{3} y_k^i \eta_k^i - \log Z^i \Big)$$

In [4]:
```
# Tensorflow approach to build a Multinomial regression model for Iris data
# Predictors
x = tf.placeholder(tf.float32,[None,4])
# Weight
W = tf.Variable(tf.random_normal([4,3]),tf.float32)
# bias
b = tf.Variable(tf.zeros([3]),tf.float32)
# logits
logits = tf.add(tf.matmul(x,W),b)
# convert to probability
prob = tf.nn.softmax(logits)
# Actual class
y = tf.placeholder(tf.float32,[None,3])

# Define the loss
cross_entropy = -tf.reduce_sum(y * tf.log(prob),reduction_indices=[1]) / tf.log(3.)
# normalize by log(3) just so the ``uninformative'' prob gives cross-entropy 1.
loss = tf.reduce_mean(cross_entropy)
```

In [5]:
```
lr = .05
batch_size = 7
```

In [6]:
```
# Define the optimization method
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lr)
fit = optimizer.minimize(loss)
```

Derive the gradient of loss function with respect to the weight

$$\frac{\partial l}{\partial W_k^\mu} = -\Big\langle x_\mu (y_k - p_k) \Big\rangle$$

In [7]:
```
# Gradient of loss /w respect to W
grad_W = -tf.matmul(tf.transpose(x),(y-prob))/tf.log(3.)/batch_size
```

In [8]:
```
# Gradient of loss /w respect to bias
grad_b = -tf.matmul(tf.ones([1,batch_size]),(y-prob))/tf.log(3.)/batch_size
```

Gradient descent optimization (https://en.wikipedia.org/wiki/Gradient_descent) is to solve the following *auxilary 1st order ODE*

$$\dot{W}_k^\mu = -\frac{\partial l}{\partial W_k^\mu}$$

where we introduce a pseodo time τ and $\dot{f} = df/d\tau$.

With *forward Euler method* (https://en.wikipedia.org/wiki/Euler_method)

$$W_k^\mu(\tau_{n+1}) = W_k^\mu(\tau_n) + \delta\tau \Big\langle x_\mu (y_k - p_k) \Big\rangle$$

which is the standard *Gradient Descent* formalism, where $\delta\tau$ is referred as *learning rate*.

In [9]:
```
#sess = tf.Session()
sess = tf.InteractiveSession()
```

In [10]:
```
sess.run(tf.global_variables_initializer())
```

In [11]:
```
# Record the weight before a single batch
W0,b0 = sess.run([W,b])
```

In [12]:
```
# Get next mini batch
btch_x,btch_y = training_dataset.next_batch(batch_size)
```

In [13]:
```
# Compute the gradient
delta_W, delta_b = sess.run([-lr*grad_W, -lr*grad_b],{x:btch_x,y:btch_y})
```

In [14]:
```
# tensorflow computed model update
sess.run(fit,{x:btch_x,y:btch_y})
```

In [15]:
```
# Updated weight per a single batch
W1,b1 = sess.run([W,b])
```

The following two Jupyter Notebook cells compare the tensorflow weight update and our computed weight update

In [16]:
```
# Compare the tf output of weight update and our own
W1 - W0, delta_W
```

Out[16]:
```
(array([[  1.99981034e-04,  -1.77860260e-04,  -2.20686197e-05],
        [  1.60366297e-04,  -1.17093325e-04,  -4.32729721e-05],
        [ -1.68681145e-05,  -7.68601894e-05,   9.36985016e-05],
        [ -2.62260437e-05,  -2.33799219e-05,   4.96506691e-05]], dtype=float32),
 array([[  1.99981368e-04,  -1.77911483e-04,  -2.20698967e-05],
        [  1.60354990e-04,  -1.17100346e-04,  -4.32546476e-05],
        [ -1.68490951e-05,  -7.68700775e-05,   9.37191653e-05],
        [ -2.62725116e-05,  -2.33828596e-05,   4.96553621e-05]], dtype=float32))
```

In [17]:
```
# Compare the tf output of bias update and our own
b1 - b0, delta_b
```

Out[17]:
```
(array([ 0.01044421, -0.00835915, -0.00208506], dtype=float32),
 array([[ 0.01044421, -0.00835915, -0.00208506]], dtype=float32))
```

1-hidden layer ANN

Now we add a single hidden layer to our softmax model.

$$h_\alpha = \sigma(x_\mu W^\mu_\alpha)$$

where $\mu = 0, 1, \ldots, 4$ with 0 indexing the bias, and α indexes the hidden nodes with 0 indicating the hidden bias. σ is the standard logistic function (https://en.wikipedia.org/wiki/Logistic_function) aka sigmoid function and for this exercise we choose it as the *activation function*. Note the derivative property $\sigma' = \sigma(1-\sigma)$ that we will use in deriving the backpropagation rule.

Logit for the output layer is given by

$$\eta_k = h_\alpha W^\alpha_k$$

Probability prediction is

$$p_k = \frac{e^{\eta_k}}{Z}$$

Loss function is (again) defined as

$$l = -\left\langle \sum_k y_k \log p_k \right\rangle$$

In [18]:
```
# Implement the 1-hidden layer ANN
x = tf.placeholder(tf.float32,[None,4])
# input -> hidden
number_of_hidden_nodes = 5
W1 = tf.Variable(tf.random_normal([4,number_of_hidden_nodes]),tf.float32)
b1 = tf.Variable(tf.zeros([number_of_hidden_nodes]),tf.float32)
eta1 =  tf.add(tf.matmul(x,W1),b1)
h1 = tf.sigmoid(eta1)
W2 = tf.Variable(tf.random_normal([number_of_hidden_nodes,3]),tf.float32)
b2 = tf.Variable(tf.zeros([3]),tf.float32)
eta2 = tf.add(tf.matmul(h1,W2),b2)
prob = tf.nn.softmax(eta2)
# Actual class
y = tf.placeholder(tf.float32,[None,3])

# Define the loss
cross_entropy = -tf.reduce_sum(y * tf.log(prob),reduction_indices=[1]) / tf.log(3.)
# normalize by log(3) just so the ``uninformative'' prob gives cross-entropy 1.
loss = tf.reduce_mean(cross_entropy)
```

In [19]:
```
lr = .05
batch_size = 7
```

In [20]:
```
# Define the optimization method
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lr)
fit = optimizer.minimize(loss)
```

Backpropagation (https://en.wikipedia.org/wiki/Backpropagation)

Take the 1st order differential of the loss function

$$dl = -\left\langle \sum_k (y_k - p_k) d\eta_k \right\rangle$$

The gradient with respect to the output layer weight is of the same form as that for softmax regression except x is replaced with h

$$\frac{\partial l}{\partial W_k^\alpha} = -\left\langle h_\alpha (y_k - p_k) \right\rangle$$

The gradient with respect to the hidden layer weight, however, is computed by chain rule (https://en.wikipedia.org/wiki/Chain_rule) that is referred as *backpropagation* in this context. We spell out all the implicit indices and summation in the following expression

$$\frac{\partial l}{\partial W_\alpha^\mu} = -\frac{1}{N \log 3} \sum_{i=1}^{N} x_\mu^i \sum_{k=1}^{3} (y_k^i - p_k^i) W_k^\alpha (\sigma(1-\sigma))_\alpha^i$$

Note there is no summation over α and we already used the property $\sigma' = \sigma(1-\sigma)$.

To implement this particular backpropagation, we first contract class index k, then carry out the element-wise multiplication over α, and then contract on sample index i.

In [21]:
```
# Gradient of loss /w respect to W2
grad_W2 = -tf.matmul(tf.transpose(h1),(y-prob))/tf.log(3.)/batch_size
# Gradient of loss /w respect to b2
grad_b2 = -tf.matmul(tf.ones([1,batch_size]),(y-prob))/tf.log(3.)/batch_size
# Gradient of loss /w respect to W1
grad_W1 = -tf.matmul(tf.transpose(x),tf.matmul((y-prob), tf.transpose(W2)) * h1 * (1-h1)) /tf.log(3.)/batch_s
ize
# Gradient of loss /w respect to b1
grad_b1 = -tf.matmul(tf.ones([1,batch_size]),tf.matmul((y-prob), tf.transpose(W2)) * h1 * (1-h1))
/tf.log(3.)/batch_size
```

In [22]:
```
# Initiate the computation
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
```

In [23]:
```
# Record the weight before a single batch
W1_0,b1_0,W2_0,b2_0 = sess.run([W1,b1,W2,b2])
```

In [24]:
```
# Get next mini batch
btch_x,btch_y = training_dataset.next_batch(batch_size)
```

In [25]:
```
# Compute weight update for the readout layer
delta_W2, delta_b2 = sess.run([-lr*grad_W2, -lr*grad_b2],{x:btch_x,y:btch_y})
```

In [26]:
```
# Compute weight update for the hidden layer
delta_W1, delta_b1 = sess.run([-lr*grad_W1, -lr*grad_b1],{x:btch_x,y:btch_y})
```

In [27]:
```
# Run one mini batch
sess.run(fit,{x:btch_x,y:btch_y})
```

In [28]:
```
# Tf output of the updated weight per a single batch
W1_1,b1_1,W2_1,b2_1 = sess.run([W1,b1,W2,b2])
```

The following two Jupyter Notebook cells compare the tensorflow weight update and our computed weight update

In [29]:
```
# Compare the tf weight update and hand-coded weight update in the hidden layer
W2_1 - W2_0, delta_W2, b2_1 - b2_0, delta_b2
```

Out[29]:
```
(array([[ 0.00627661,  0.00026166, -0.0065383 ],
        [ 0.00620437,  0.0002889 , -0.00649327],
        [ 0.00610174,  0.00023231, -0.00633407],
        [ 0.00620842,  0.00030261, -0.00651109],
        [ 0.00619113,  0.00023773, -0.00642884]], dtype=float32),
 array([[ 0.0062766 ,  0.00026169, -0.0065383 ],
        [ 0.00620435,  0.00028892, -0.00649327],
        [ 0.00610174,  0.0002323 , -0.00633404],
        [ 0.00620842,  0.00030261, -0.00651104],
        [ 0.00619111,  0.00023773, -0.00642884]], dtype=float32),
 array([ 0.01234427,  0.000517  , -0.01286126], dtype=float32),
 array([[ 0.01234426,  0.000517  , -0.01286126]], dtype=float32))
```

In [30]:

```
# Compare the tf weight update and hand-coded weight update in the hidden layer
W1_1 - W1_0, delta_W1, b1_1 - b1_0, delta_b1
```

Out[30]:

```
(array([[ -1.04904175e-04,  -4.33474779e-05,  -1.72495842e-04,
           1.39117241e-04,   5.57899475e-05],
        [ -9.52482224e-05,  -1.80602074e-05,  -1.13487244e-04,
           9.54866409e-05,   3.00407410e-05],
        [  9.14186239e-06,  -3.50475311e-05,  -6.38365746e-05,
           4.42266464e-05,   3.27825546e-05],
        [  7.91251659e-06,  -9.13441181e-06,  -1.21593475e-05,
           7.49714673e-06,   7.80820847e-06]], dtype=float32),
 array([[ -1.04884326e-04,  -4.33545829e-05,  -1.72494823e-04,
           1.39112613e-04,   5.58346910e-05],
        [ -9.52763148e-05,  -1.80727347e-05,  -1.13459188e-04,
           9.54725328e-05,   3.00681040e-05],
        [  9.13818349e-06,  -3.50982373e-05,  -6.38418787e-05,
           4.42430755e-05,   3.28093920e-05],
        [  7.90586910e-06,  -9.13498479e-06,  -1.21612884e-05,
           7.49766104e-06,   7.81073322e-06]], dtype=float32),
 array([-0.00545712, -0.00178968, -0.00802938,  0.00656228,  0.00245341], dtype=float32),
 array([[-0.00545712, -0.00178968, -0.00802938,  0.00656228,  0.00245341]], dtype=float32))
```