Parallelizing Neural Network Training with TensorFlow

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

Outline

- Building, Compiling, and Running Machine Learning Models with TensorFlow
- Training Neural Networks Efficiently with High-Level TensorFlow APIs
- Choosing Activation Functions for Multilayer Networks

Building, Compiling, and Running Machine Learning Models with TensorFlow

Computation Efficiency

- By default, Python is limited to execution on one core due to Global Interpreter Lock (GIL)
- Parallel processing capability of GPUs
  - CUDA or OpenCL is not convenient for common people

Specifications	Intel® Core™ i7-6900K Processor Extreme Ed.	NVIDIA GeForce® GTX™ 1080 Ti
Base Clock Frequency	3.2 GHz	< 1.5 GHz
Cores	8	3584
Memory Bandwidth	64 GB/s	484 GB/s
Floating-Point Calculations	409 GFLOPS	11300 GFLOPS
Cost	~ $1000.00	~ $700.00

By Aug. 2017

TensorFlow (1/2)

- A ***scalable*** and ***multi-platform*** programming interface for implementing and running machine learning algorithms, including convenient wrappers for deep learning
  - In hardware, TensorFlow supports both CPUs and CUDA-based GPUs (for OpenCL-enabled devices is experimental)
  - In programming languages, TensorFlow has an official APIs for Python and C++

TensorFlow (2/2)

- TensorFlow is built around a computation graph composed of a set of nodes
  - Each node represents an operation that may have zero or more inputs or outputs
  - The value that flows through the edges of the computation graph are called ***tensors***
- Two level of TensorFlow APIs
  - Low-level: Giving more flexibility as programmers to combine the basic operations and develop complex machine learning models
  - High-level: Built on top of the low-level TensorFlow APIs, allowing building and prototyping models much faster
    - TensorFlow **Layers** and **Keras**

First Step with Low Level TensorFlow API

- Tensor can be understood as a generalization of scalars, vectors, matrices, and so on.
  - A scalar can be defined as a rank-0 tensor, a vector as a rank-1 tensor, a matrix as a rank-2 tensor, and matrices stacked in a third dimension as rank-3 tensor
- In a computation graph,
  - A ***placeholder*** is to hold input data
    - A placeholder with *shape=(None)* can take input data of any size along the corresponding axis
    - In above, the input ***x*** is a scalar
  - A ***variable*** is to hold a parameter tensor

First Step with Low Level TensorFlow API

$$z = w \times x + b$$

Construction Phase

Execution Phase

```
import tensorflow as tf

## create a graph
g = tf.Graph()
with g.as_default():
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None), name='x')
    w = tf.Variable(2.0, name='weight')
    b = tf.Variable(0.7, name='bias')

    z = w*x + b
    init = tf.global_variables_initializer()

## create a session and pass in graph g
with tf.Session(graph=g) as sess:
    ## initialize w and b:
    sess.run(init)
    ## evaluate z:
    for t in [1.0, 0.6, -1.8]:
        print('x=%4.1f --> z=%4.1f'%(
              t, sess.run(z, feed_dict={x:t})))
```

```
x= 1.0 --> z= 2.7
x= 0.6 --> z= 1.9
x=-1.8 --> z=-2.9
```

First Step with Low Level TensorFlow API

- In the previous example, the input is fed in an element-by-element form
- Below, we feed the input *x* as a minibatch of size 3

```
with tf.Session(graph=g) as sess:
    sess.run(init)
    print(sess.run(z, feed_dict={x:[1., 2., 3.]}))
```

```
[ 2.70000005  4.69999981  6.69999981]
```

Working with Array Structures

- Create a rank-3 tensor of size *batchsize* x 2 x 3, reshape it, and calculate the column sums and means using TensorFlow's optimized sessions
- When reshaping a tensor, if use '-1' for a specific axis, the size of the axis will be computed according to the total size of the tensor and the shape of the remaining axes

Working with Array Structures

```python
import tensorflow as tf
import numpy as np


g = tf.Graph()
with g.as_default():
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None, 2, 3),
                       name='input_x')

    x2 = tf.reshape(x, shape=(-1, 6),
                    name='x2')

    ## calculate the sum of each column
    xsum = tf.reduce_sum(x2, axis=0, name='col_sum')

    ## calculate the mean of each column
    xmean = tf.reduce_mean(x2, axis=0, name='col_mean')
with tf.Session(graph=g) as sess:
    x_array = np.arange(18).reshape(3, 2, 3)
    print('input shape: ', x_array.shape)
    print('Reshaped:\n',
          sess.run(x2, feed_dict={x:x_array}))
    print('Column Sums:\n',
          sess.run(xsum, feed_dict={x:x_array}))
    print('Column Means:\n',
          sess.run(xmean, feed_dict={x:x_array}))
```

```
input shape:  (3, 2, 3)
Reshaped:
 [[  0.   1.   2.   3.   4.   5.]
 [  6.   7.   8.   9.  10.  11.]
 [ 12.  13.  14.  15.  16.  17.]]
Column Sums:
 [ 18.  21.  24.  27.  30.  33.]
Column Means:
 [  6.   7.   8.   9.  10.  11.]
```

Developing a Simple Model with Low-Level TensorFlow APIs

- Implement the Ordinary Least Square regression in a class with low-level TensorFlow API
  - Training *X*:10 instances with 1 dimensional feature vector
  - Training label *y*:10 corresponding target labels
  - Two placeholders are needed, one for *X* and the other *y*.
  - MSE as cost function with gradient descent optimizer

```python
import tensorflow as tf
import numpy as np

X_train = np.arange(10).reshape((10, 1))
y_train = np.array([1.0, 1.3, 3.1,
                    2.0, 5.0, 6.3,
                    6.6, 7.4, 8.0,
                    9.0])
```

Linear Regression Model Definition

```
class TfLinreg(object):

    def __init__(self, x_dim, learning_rate=0.01,
                 random_seed=None):
        self.x_dim = x_dim
        self.learning_rate = learning_rate
        self.g = tf.Graph()
        ## build the model
        with self.g.as_default():
            ## set graph-level random-seed
            tf.set_random_seed(random_seed)

            self.build()
            ## create initializer
            self.init_op = tf.global_variables_initializer()
```

```
def build(self):
    ## define placeholders for inputs
    self.X = tf.placeholder(dtype=tf.float32,
                            shape=(None, self.x_dim),
                            name='x_input')
    self.y = tf.placeholder(dtype=tf.float32,
                            shape=(None),
                            name='y_input')
    print(self.X)
    print(self.y)
    ## define weight matrix and bias vector
    w = tf.Variable(tf.zeros(shape=(1)),
                    name='weight')
    b = tf.Variable(tf.zeros(shape=(1)),
                    name="bias")
    print(w)
    print(b)

    self.z_net = tf.squeeze(w*self.X + b,
                            name='z_net')
    print(self.z_net)
    sqr_errors = tf.square(self.y - self.z_net,
                           name='sqr_errors')
    print(sqr_errors)
    self.mean_cost = tf.reduce_mean(sqr_errors,
                                    name='mean_cost')

    optimizer = tf.train.GradientDescentOptimizer(
                learning_rate=self.learning_rate,
                name='GradientDescent')
    self.optimizer = optimizer.minimize(self.mean_cost)
```

Create an Instance of OLS Regression

```
lrmodel = TfLinreg(x_dim=X_train.shape[1], learning_rate=0.01)
```

```
Tensor("x_input:0", shape=(?, 1), dtype=float32)
Tensor("y_input:0", dtype=float32)
<tf.Variable 'weight:0' shape=(1,) dtype=float32_ref>
<tf.Variable 'bias:0' shape=(1,) dtype=float32_ref>
Tensor("z_net:0", dtype=float32)
Tensor("sqr_errors:0", dtype=float32)
```

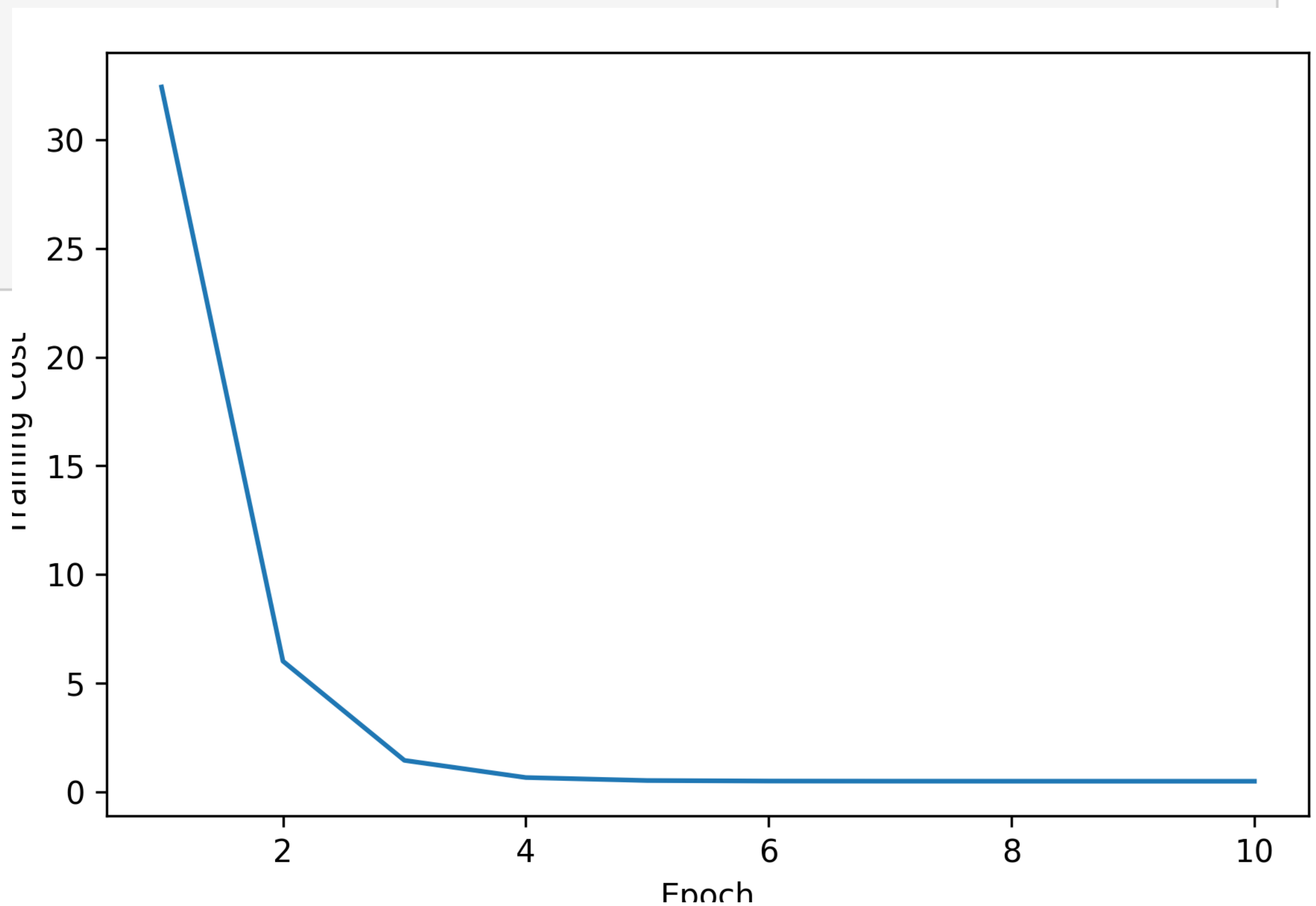
Implementing a Training Function

```
def train_linreg(sess, model, X_train, y_train, num_epochs=10):
    ## initialiaze all variables: W & b
    sess.run(model.init_op)

    training_costs = []
    for i in range(num_epochs):
        _, cost = sess.run([model.optimizer, model.mean_cost],
                           feed_dict={model.X:X_train,
                                      model.y:y_train})
        training_costs.append(cost)

    return training_costs
```

Train the Model

```
sess = tf.Session(graph=lrmodel.g)
training_costs = train_linreg(sess, lrmodel, X_train, y_train)
```

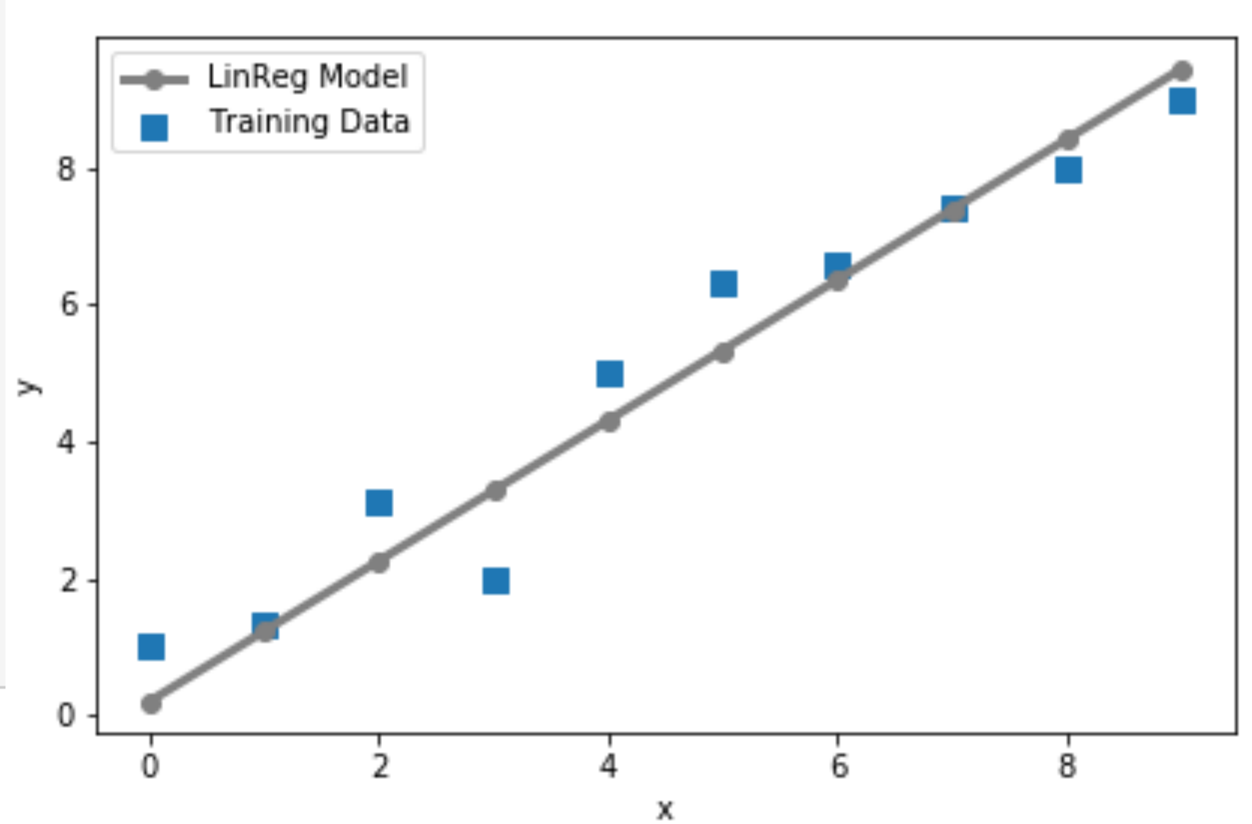
```
import matplotlib.pyplot as plt

plt.plot(range(1,len(training_costs) + 1), training_costs)
plt.tight_layout()
plt.xlabel('Epoch')
plt.ylabel('Training Cost')
#plt.savefig('images/13_01.png', dpi=300)
plt.show()
```



Make Prediction

```
def predict_linreg(sess, model, X_test):
    y_pred = sess.run(model.z_net,
                      feed_dict={model.X:X_test})
    return y_pred
```

```
plt.scatter(X_train, y_train,
            marker='s', s=50,
            label='Training Data')
plt.plot(range(X_train.shape[0]),
         predict_linreg(sess, lrmodel, X_train),
         color='gray', marker='o',
         markersize=6, linewidth=3,
         label='LinReg Model')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.tight_layout()
#plt.savefig('images/13_02.png')
plt.show()
```



Training Neural Networks Efficiently with High-Level TensorFlow APIs

TensorFlow High-Level API Examples

- The Layers API
  - tensorflow.layers or tf.layers
- The Keras APS
  - tensor flow.contrib.keras

Building Multilayer Neural Networks Using TensorFlow's Layers API

- Implement a MLP to classify the handwritten digits from the MNIST dataset

```python
# unzips mnist

import sys
import gzip
import shutil
import os

if (sys.version_info > (3, 0)):
    writemode = 'wb'
else:
    writemode = 'w'

zipped_mnist = [f for f in os.listdir('./') if f.endswith('ubyte.gz')]
for z in zipped_mnist:
    with gzip.GzipFile(z, mode='rb') as decompressed, open(z[:-3], writemode) as outfile:
        outfile.write(decompressed.read())
```

Preprocessing of the Dataset

```python
import struct

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
        images = ((images / 255.) - .5) * 2

    return images, labels
```

Load the Dataset

```
## loading the data
X_train, y_train = load_mnist('.', kind='train')
print('Rows: %d,  Columns: %d' %(X_train.shape[0],
                                 X_train.shape[1]))

X_test, y_test = load_mnist('.', kind='t10k')
print('Rows: %d,  Columns: %d' %(X_test.shape[0],
                                     X_test.shape[1]))
## mean centering and normalization:
mean_vals = np.mean(X_train, axis=0)
std_val = np.std(X_train)

X_train_centered = (X_train - mean_vals)/std_val
X_test_centered = (X_test - mean_vals)/std_val

del X_train, X_test

print(X_train_centered.shape, y_train.shape)

print(X_test_centered.shape, y_test.shape)
```

```
Rows: 60000,  Columns: 784
Rows: 10000,  Columns: 784
(60000, 784) (60000,)
(10000, 784) (10000,)
```

Build a Computation Graph for 3-layer MLP

- Add additional hidden layer
- Replace logistic units in hidden layer with *hyperbolic tangent* activation functions, output layer with *softmax*

```
import tensorflow as tf

n_features = X_train_centered.shape[1]
n_classes = 10
random_seed = 123
np.random.seed(random_seed)

g = tf.Graph()
with g.as_default():
    tf.set_random_seed(random_seed)
    tf_x = tf.placeholder(dtype=tf.float32,
                          shape=(None, n_features),
                          name='tf_x')

    tf_y = tf.placeholder(dtype=tf.int32,
                           shape=None, name='tf_y')
    y_onehot = tf.one_hot(indices=tf_y, depth=n_classes)

    h1 = tf.layers.dense(inputs=tf_x, units=50,
                          activation=tf.tanh,
                          name='layer1')
    h2 = tf.layers.dense(inputs=h1, units=50,
                          activation=tf.tanh,
                          name='layer2')

    logits = tf.layers.dense(inputs=h2,
                              units=10,
                              activation=None,
                              name='layer3')

    predictions = {
        'classes' : tf.argmax(logits, axis=1,
                               name='predicted_classes'),
        'probabilities' : tf.nn.softmax(logits,
                               name='softmax_tensor')
    }
```

Define Cost Functions and Optimizer

```
## define cost function and optimizer:
with g.as_default():
    cost = tf.losses.softmax_cross_entropy(
            onehot_labels=y_onehot, logits=logits)

    optimizer = tf.train.GradientDescentOptimizer(
            learning_rate=0.001)

    train_op = optimizer.minimize(loss=cost)

    init_op = tf.global_variables_initializer()
```

Generate Batches of Data to Train the Network

```
def create_batch_generator(X, y, batch_size=128, shuffle=False):
    X_copy = np.array(X)
    y_copy = np.array(y)

    if shuffle:
        data = np.column_stack((X_copy, y_copy))
        np.random.shuffle(data)
        X_copy = data[:, :-1]
        y_copy = data[:, -1].astype(int)

    for i in range(0, X.shape[0], batch_size):
        yield (X[i:i+batch_size, :], y[i:i+batch_size])
```

Create a TensorFlow Session and Start Training

```
## create a session to launch the graph
sess =  tf.Session(graph=g)
## run the variable initialization operator
sess.run(init_op)

## 50 epochs of training:
training_costs = []
for epoch in range(50):
    training_loss = []
    batch_generator = create_batch_generator(
            X_train_centered, y_train,
            batch_size=64, shuffle=True)
    for batch_X, batch_y in batch_generator:
        ## prepare a dict to feed data to our network:
        feed = {tf_x:batch_X, tf_y:batch_y}
        _, batch_cost = sess.run([train_op, cost],
                                 feed_dict=feed)
        training_costs.append(batch_cost)
    print(' -- Epoch %2d  '
          'Avg. Training Loss: %.4f' % (
              epoch+1, np.mean(training_costs)
    ))
```

```
-- Epoch  1  Avg. Training Loss: 1.5573
-- Epoch  2  Avg. Training Loss: 1.2532
-- Epoch  3  Avg. Training Loss: 1.0854
-- Epoch  4  Avg. Training Loss: 0.9738
-- Epoch  5  Avg. Training Loss: 0.8924
-- Epoch  6  Avg. Training Loss: 0.8296
-- Epoch  7  Avg. Training Loss: 0.7794
-- Epoch  8  Avg. Training Loss: 0.7381
-- Epoch  9  Avg. Training Loss: 0.7032
-- Epoch 10  Avg. Training Loss: 0.6734
-- Epoch 11  Avg. Training Loss: 0.6475
-- Epoch 12  Avg. Training Loss: 0.6247
-- Epoch 13  Avg. Training Loss: 0.6045
-- Epoch 14  Avg. Training Loss: 0.5864
-- Epoch 15  Avg. Training Loss: 0.5700
-- Epoch 16  Avg. Training Loss: 0.5551
-- Epoch 17  Avg. Training Loss: 0.5415
-- Epoch 18  Avg. Training Loss: 0.5290
-- Epoch 19  Avg. Training Loss: 0.5175
-- Epoch 20  Avg. Training Loss: 0.5068
-- Epoch 21  Avg. Training Loss: 0.4968
-- Epoch 22  Avg. Training Loss: 0.4875
-- Epoch 23  Avg. Training Loss: 0.4788
```

Make Prediction on Test Dataset

```
## do prediction on the test set:
feed = {tf_x : X_test_centered}
y_pred = sess.run(predictions['classes'],
                  feed_dict=feed)

print('Test Accuracy: %.2f%%' % (
      100*np.sum(y_pred == y_test)/y_test.shape[0]))
```

```
Test Accuracy: 93.89%
```

Developing MLP with Keras

- Keras has been integrated into TensorFlow since version TensorFlow 1.1.0
- Currently Keras is a part of the contrib module of TensorFlow
- In the future release, Keras may be moved to become a separate module in the TensorFlow main API

Load the Dataset

```
X_train, y_train = load_mnist('./', kind='train')
print('Rows: %d,  Columns: %d' %(X_train.shape[0],
                                 X_train.shape[1]))
X_test, y_test = load_mnist('./', kind='t10k')
print('Rows: %d,  Columns: %d' %(X_test.shape[0],
                                 X_test.shape[1]))

## mean centering and normalization:
mean_vals = np.mean(X_train, axis=0)
std_val = np.std(X_train)

X_train_centered = (X_train - mean_vals)/std_val
X_test_centered = (X_test - mean_vals)/std_val

del X_train, X_test

print(X_train_centered.shape, y_train.shape)

print(X_test_centered.shape, y_test.shape)
```

```
Rows: 60000,  Columns: 784
Rows: 10000,  Columns: 784
(60000, 784) (60000,)
(10000, 784) (10000,)
```

Initialization

- Use same graph-level random seed as in TensorFlow's Layers API
- Keras provides a convenient tool to convert the integer class labels into the 1-hot format

```
import tensorflow as tf
import tensorflow.contrib.keras as keras

np.random.seed(123)
tf.set_random_seed(123)
```

```
y_train_onehot = keras.utils.to_categorical(y_train)

print('First 3 labels: ', y_train[:3])
print('\nFirst 3 labels (one-hot):\n', y_train_onehot[:3])
```

```
First 3 labels:  [5 0 4]

First 3 labels (one-hot):
 [[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```

LaRC NTHU EE Laboratory for Reliable Computing

Use Keras to Build Model

```
model = keras.models.Sequential()

model.add(
    keras.layers.Dense(
        units=50,
        input_dim=X_train_centered.shape[1],
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='tanh'))

model.add(
    keras.layers.Dense(
        units=50,
        input_dim=50,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='tanh'))
model.add(
    keras.layers.Dense(
        units=y_train_onehot.shape[1],
        input_dim=50,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        activation='softmax'))


sgd_optimizer = keras.optimizers.SGD(
        lr=0.001, decay=1e-7, momentum=.9)

model.compile(optimizer=sgd_optimizer,
              loss='categorical_crossentropy')
```

Training the Model

```
history = model.fit(X_train_centered, y_train_onehot,
                    batch_size=64, epochs=50,
                    verbose=1,
                    validation_split=0.1)
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/50
54000/54000 [==============================] - 2s - loss: 0.7247 - val_loss: 0.3616
Epoch 2/50
54000/54000 [==============================] - 2s - loss: 0.3718 - val_loss: 0.2815
Epoch 3/50
54000/54000 [==============================] - 2s - loss: 0.3087 - val_loss: 0.2447
Epoch 4/50
54000/54000 [==============================] - 2s - loss: 0.2728 - val_loss: 0.2216
Epoch 5/50
54000/54000 [==============================] - 2s - loss: 0.2475 - val_loss: 0.2042
Epoch 6/50
54000/54000 [==============================] - 2s - loss: 0.2277 - val_loss: 0.1918
Epoch 7/50
54000/54000 [==============================] - 2s - loss: 0.2115 - val_loss: 0.1810
```

Make Predictions

```
y_train_pred = model.predict_classes(X_train_centered, verbose=0)
print('First 3 predictions: ', y_train_pred[:3])
```

```
First 3 predictions:  [5 0 4]
```

```
y_train_pred = model.predict_classes(X_train_centered,
                                     verbose=0)
correct_preds = np.sum(y_train == y_train_pred, axis=0)
train_acc = correct_preds / y_train.shape[0]

print('First 3 predictions: ', y_train_pred[:3])
print('Training accuracy: %.2f%%' % (train_acc * 100))
```

```
First 3 predictions:  [5 0 4]
Training accuracy: 98.88%
```

```
y_test_pred = model.predict_classes(X_test_centered,
                                    verbose=0)

correct_preds = np.sum(y_test == y_test_pred, axis=0)
test_acc = correct_preds / y_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
```

```
Test accuracy: 96.04%
```

Choosing Activation Functions for Multilayer Networks

Logistic Function Recap

$$\phi_{logistic}(z) = \frac{1}{1+e^{-z}}$$

- The logistic function has a range (0,1) and gives the likelihood P(y=1 | x) of the prediction to be positive given a data point x
- It is the inverse of the logit (log odds) function

Logistic Function Recap

```
import numpy as np

X = np.array([1, 1.4, 2.5]) ## first value must be 1
w = np.array([0.4, 0.3, 0.5])

def net_input(X, w):
    return np.dot(X, w)

def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))

def logistic_activation(X, w):
    z = net_input(X, w)
    return logistic(z)

print('P(y=1|x) = %.3f' % logistic_activation(X, w))
```

```
P(y=1|x) = 0.888
```

Issues with Multiple Logistic Activation Units in Output Layer

```
# W : array with shape = (n_output_units, n_hidden_units+1)
#     note that the first column are the bias units

W = np.array([[1.1, 1.2, 0.8, 0.4],
              [0.2, 0.4, 1.0, 0.2],
              [0.6, 1.5, 1.2, 0.7]])

# A : data array with shape = (n_hidden_units + 1, n_samples)
#     note that the first column of this array must be 1

A = np.array([[1, 0.1, 0.4, 0.6]])

Z = np.dot(W, A[0])
y_probas = logistic(Z)

print('Net Input: \n', Z)

print('Output Units:\n', y_probas)
```

```
Net Input:
 [ 1.78  0.76  1.65]
Output Units:
 [ 0.85569687  0.68135373  0.83889105]
```

NTHU EE
Laboratory for Reliable Computing

Softmax Function

- A soft form of argmax function
  - Instead of giving a single class index, it provides the probability of each class

$$p\left(y=i \middle| z\right)=\phi\left(z\right)=\frac{e^{z_i}}{\sum_{i=1}^{M} e^{z_j}}$$

```
def softmax(z):
    return np.exp(z) / np.sum(np.exp(z))

y_probas = softmax(Z)
print('Probabilities:\n', y_probas)
```

```
Probabilities:
 [ 0.44668973  0.16107406  0.39223621]
```

```
np.sum(y_probas)
```

```
1.0
```

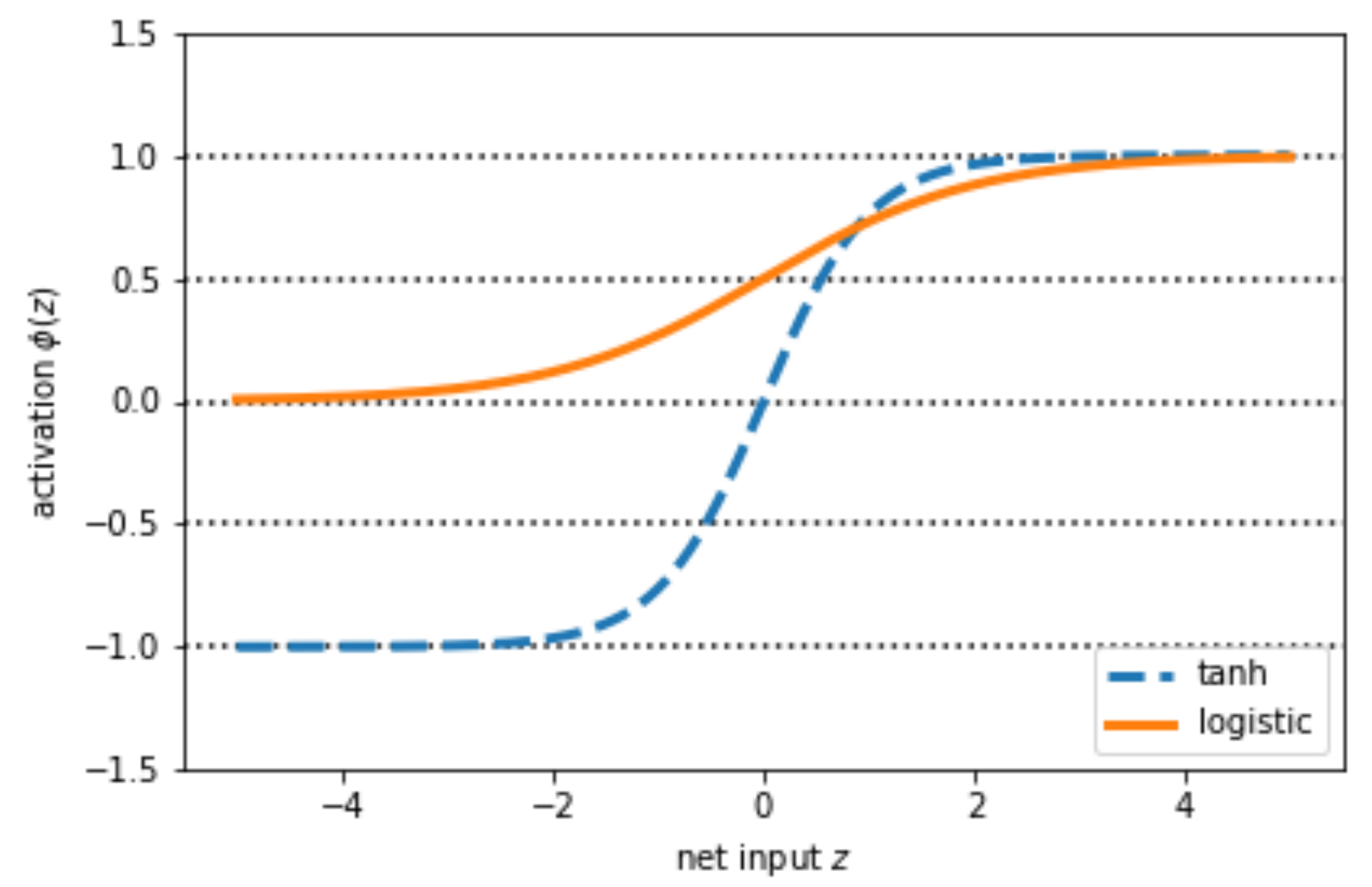
Hyperbolic Tangent Function

- Rescaled version of the logistic function

$$\phi_{logistic}(z) = \frac{1}{1+e^{-z}} \qquad \phi_{tanh}(z) = 2\times\phi_{logistic}(2z)-1 = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

- The tanh has a broader output spectrum and ranges in the open interval (-1,1), which can improve the performance of the back propagation algorithm

```python
import matplotlib.pyplot as plt

def tanh(z):
    e_p = np.exp(z)
    e_m = np.exp(-z)
    return (e_p - e_m) / (e_p + e_m)

z = np.arange(-5, 5, 0.005)
log_act = logistic(z)
tanh_act = tanh(z)

plt.ylim([-1.5, 1.5])
plt.xlabel('net input $z$')
plt.ylabel('activation $\phi(z)$')
plt.axhline(1, color='black', linestyle=':')
plt.axhline(0.5, color='black', linestyle=':')
plt.axhline(0, color='black', linestyle=':')
plt.axhline(-0.5, color='black', linestyle=':')
plt.axhline(-1, color='black', linestyle=':')
plt.plot(z, tanh_act,
         linewidth=3, linestyle='--',
         label='tanh')

plt.plot(z, log_act,
         linewidth=3,
         label='logistic')
plt.legend(loc='lower right')
plt.tight_layout()
#plt.savefig('images/13_03.png')
plt.show()
```

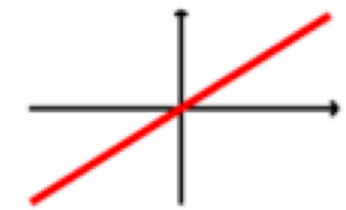
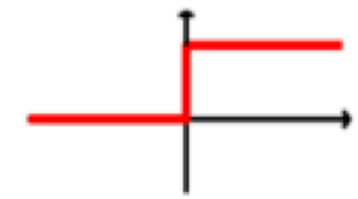
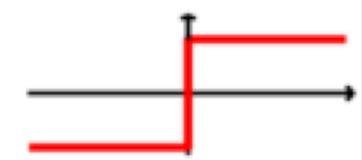
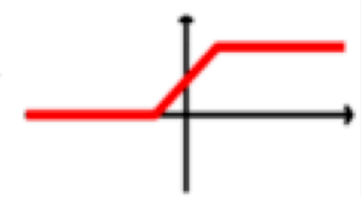
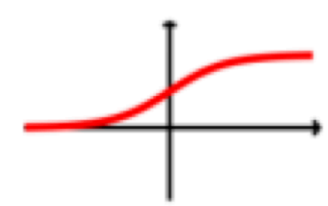
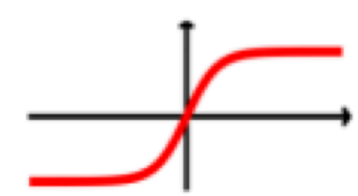


Rectified Linear Unit (ReLU) Function

- ReLU solves the problem of vanishing gradients for logistic and tanh functions at large input values
- The derivative of ReLU, with respect to its inputs, is always 1 for positive input values and always 0 for negative inout values
- ReLU is commonly used in convolutional neural networks

$$\phi(z) = \max(0, z)$$

Activation Functions

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z)= \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z)= \begin{cases} 0 & z \le -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \le z \le \frac{1}{2} \\ 1 & z \ge \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z)= \dfrac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z)= \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z)= \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs