# <https://www.tensorflow.org/versions/master/tutorials>

## MNIST for ML beginners

1 data has 10 classes (0~9)

28 pixels x28 pixels = 784

### Source code

|  |  |  |
| --- | --- | --- |
| # Copyright 2015 The TensorFlow Authors. All Rights Reserved.  # Licensed under the Apache License, Version 2.0 (the "License");  # you may not use this file except in compliance with the License.  # You may obtain a copy of the License at  # http://www.apache.org/licenses/LICENSE-2.0  # Unless required by applicable law or agreed to in writing, software  # distributed under the License is distributed on an "AS IS" BASIS,  # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  # See the License for the specific language governing permissions and  # limitations under the License.  # ==============================================================================  """A very simple MNIST classifier.  See extensive documentation at  http://tensorflow.org/tutorials/mnist/beginners/index.md  """  from \_\_future\_\_ import absolute\_import  from \_\_future\_\_ import division  from \_\_future\_\_ import print\_function  import argparse  import sys  from tensorflow.examples.tutorials.mnist import input\_data  import tensorflow as tf  FLAGS = None | def main(\_):  # Import data  mnist = input\_data.read\_data\_sets(FLAGS.data\_dir, one\_hot=True)  # Create the model  y is predicted class  x = tf.placeholder(tf.float32, [None, 784])  W = tf.Variable(tf.zeros([784, 10]))  b = tf.Variable(tf.zeros([10]))  y = tf.matmul(x, W) + b  # Define loss and optimizer  y\_ = tf.placeholder(tf.float32, [None, 10])  y\_ is the label (class)  # The raw formulation of cross-entropy,  #  # tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(tf.nn.softmax(y)),  # reduction\_indices=[1]))  #  # can be numerically unstable.  #  # So here we use tf.nn.softmax\_cross\_entropy\_with\_logits on the raw  # outputs of 'y', and then average across the batch. | cross\_entropy = tf.reduce\_mean(  tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_, logits=y))  train\_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross\_entropy)  sess = tf.InteractiveSession()  tf.global\_variables\_initializer().run()  # Train  for \_ in range(1000):  batch\_xs, batch\_ys = mnist.train.next\_batch(100)  sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})    # Test trained model  correct\_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))  accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  print(sess.run(accuracy, feed\_dict={x: mnist.test.images,  y\_: mnist.test.labels}))  if \_\_name\_\_ == '\_\_main\_\_':  parser = argparse.ArgumentParser()  parser.add\_argument('--data\_dir', type=str, default='/tmp/tensorflow/mnist/input\_data',  help='Directory for storing input data')  FLAGS, unparsed = parser.parse\_known\_args()  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed) |

### The MNIST data

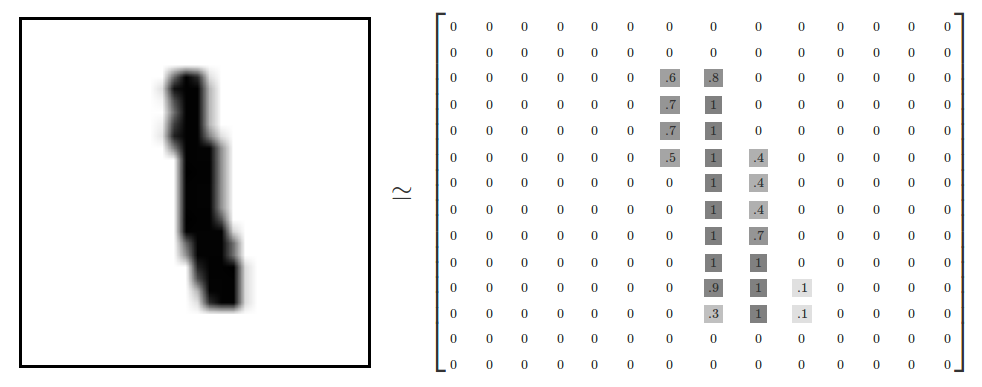
55000 training

10000 test

5000 validation

MNIST database (handwriting)

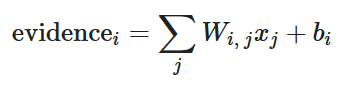
10 labels: 0 to 9, For example, 3 would be [0,0,0,1,0,0,0,0,0,0]



28x28=784

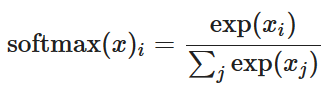
### Softmax Regressions

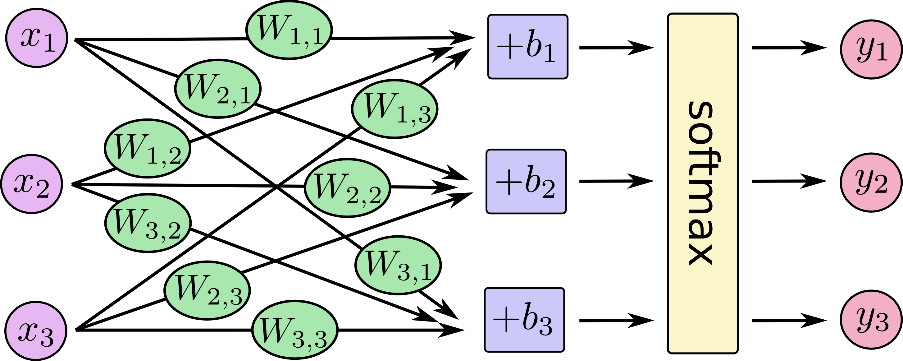
* softmax gives us a list of values between 0 and 1 that add up to 1
* even later on, when we train more sophisticated models, the final step will be a layer of softmax.
* A softmax regression has two steps: first we add up the evidence of our input being in certain classes, and then we convert that evidence into probabilities.

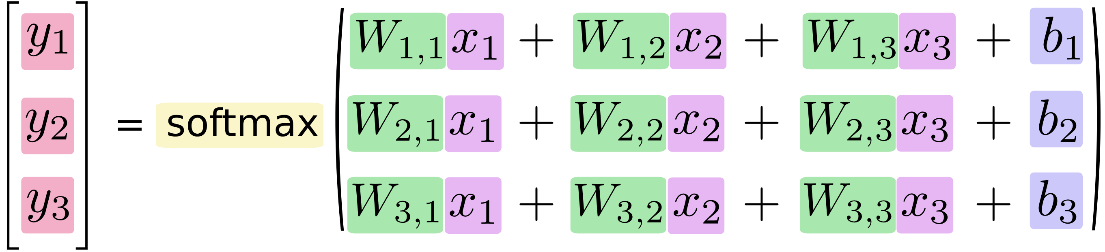


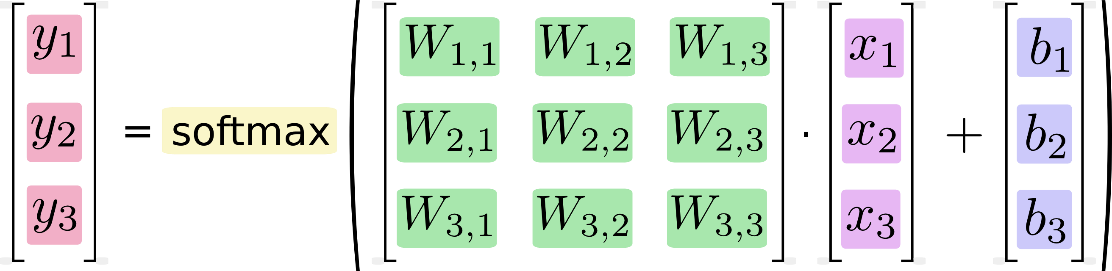












*y*=softmax(*Wx*+*b*)

### Implementing the Regression

* libraries like [NumPy](http://www.numpy.org/) that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language.
* Unfortunately, there can still be a lot of overhead from switching back to Python every operation.
* TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. (Approaches like this can be seen in a few machine learning libraries.)

**(Here None means that a dimension can be of any length.)**

**x isn't a specific value. It's a placeholder, a value that we'll input when we ask TensorFlow to run a computation. We want to be able to input any number of MNIST images, each flattened into a 784-dimensional vector. We represent this as a 2-D tensor of floating-point numbers, with a shape [None, 784].**

import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 784])

W = tf.**Variable**(tf.zeros([784, 10]))  
b = tf.**Variable**(tf.zeros([10]))

**10 means 10 classes**

**A Variable is a modifiable tensor that lives in TensorFlow's graph of interacting operations. It can be used and even modified by the computation. For machine learning applications, one generally has the model parameters be Variable s.**

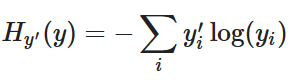
\*\*\*This is 1 layer ANN

we initialize both W and b as tensors full of zeros

y = tf.nn.softmax(tf.matmul(x, W) + b)

### Training

Cross-entropy



Where y is our predicted probability distribution, and y′ is the true distribution (the one-hot vector with the digit labels). In some rough sense, the cross-entropy is measuring how inefficient our predictions are for describing the truth.

Let y’ be:

y\_ = tf.placeholder(tf.float32, [None, 10])

tf.log computes the logarithm of each element of y

then

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y), reduction\_indices=[1]))

we multiply each element of y\_ with the corresponding element of tf.log(y)

# 'x' is [[1, 1, 1]

# [1, 1, 1]]

tf.reduce\_sum(x) ==> 6

tf.reduce\_sum(x, 0) ==> [2, 2, 2]

tf.reduce\_sum(x, 1) ==> [3, 3]

tf.reduce\_sum(x, 1, keep\_dims=True) ==> [[3], [3]]

tf.reduce\_sum(x, [0, 1]) ==> 6

Note that in the source code, we don't use this formulation, because it is numerically unstable. Instead, we apply ***tf.nn.softmax\_cross\_entropy\_with\_logits*** on the unnormalized logits (e.g., we call softmax\_cross\_entropy\_with\_logits on tf.matmul(x, W) + b), because this more numerically stable function internally computes the softmax activation. In your code, consider using tf.nn.softmax\_cross\_entropy\_with\_logits instead.

*cross\_entropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_, logits=y)) #for last layer only*

*train\_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross\_entropy)*

Learning rate

**compute\_gradients(loss, var\_list=None, gate\_gradients=GATE\_OP, aggregation\_method=None, colocate\_gradients\_with\_ops=False, grad\_loss=None)**

Compute gradients of loss for the variables in var\_list.

This is the first part of minimize(). It returns a list of (gradient, variable) pairs where "gradient" is the gradient for "variable". Note that "gradient" can be a Tensor, an IndexedSlices, or None if there is no gradient for the given variable.

**Args:**

* **loss**: A Tensor containing the value to minimize.
* **var\_list**: Optional list of tf.Variable to update to minimize loss. Defaults to the list of variables collected in the graph under the key GraphKey.TRAINABLE\_VARIABLES.
* **gate\_gradients**: How to gate the computation of gradients. Can be GATE\_NONE, GATE\_OP, or GATE\_GRAPH.
* **aggregation\_method**: Specifies the method used to combine gradient terms. Valid values are defined in the class AggregationMethod.
* **colocate\_gradients\_with\_ops**: If True, try colocating gradients with the corresponding op.
* **grad\_loss**: Optional. A Tensor holding the gradient computed for loss.

**Returns:**

A list of (gradient, variable) pairs. Variable is always present, but gradient can be None.

**Raises:**

* **TypeError**: If var\_list contains anything else than Variable objects.
* **ValueError**: If some arguments are invalid.

**minimize(loss, global\_step=None, var\_list=None, gate\_gradients=GATE\_OP, aggregation\_method=None, colocate\_gradients\_with\_ops=False, name=None, grad\_loss=None)**

Add operations to minimize loss by updating var\_list.

This method simply combines calls **compute\_gradients()** and **apply\_gradients().** If you want to process the gradient before applying them call compute\_gradients() and apply\_gradients() explicitly instead of using this function.

**Args:**

* **loss**: A Tensor containing the value to minimize.
* **global\_step**: Optional Variable to increment by one after the variables have been updated.
* **var\_list**: Optional list of Variable objects to update to minimize loss. Defaults to the list of variables collected in the graph under the key GraphKeys.TRAINABLE\_VARIABLES.
* **gate\_gradients**: How to gate the computation of gradients. Can be GATE\_NONE, GATE\_OP, or GATE\_GRAPH.
* **aggregation\_method**: Specifies the method used to combine gradient terms. Valid values are defined in the class AggregationMethod.
* **colocate\_gradients\_with\_ops**: If True, try colocating gradients with the corresponding op.
* **name**: Optional name for the returned operation.
* **grad\_loss**: Optional. A Tensor holding the gradient computed for loss.

**Returns:**

An Operation that updates the variables in var\_list. If global\_step was not None, that operation also increments global\_step.

**Raises:**

* **ValueError**: If some of the variables are not Variable objects.

1st part: Compute Gradient

2nd part: Apply Gradient

**apply\_gradients(grads\_and\_vars, global\_step=None, name=None)**

**Apply gradients to variables.**

**This is the second part of minimize(). It returns an Operation that applies gradients.**

**Args:**

* **grads\_and\_vars: List of (gradient, variable) pairs as returned by compute\_gradients().**
* **global\_step: Optional Variable to increment by one after the variables have been updated.**
* **name: Optional name for the returned operation. Default to the name passed to the Optimizer constructor.**

**Returns:**

**An Operation that applies the specified gradients. If global\_step was not None, that operation also increments global\_step.**

**Raises:**

* **TypeError: If grads\_and\_vars is malformed.**
* **ValueError: If none of the variables have gradients.**

*sess = tf.InteractiveSession()*

*tf.global\_variables\_initializer().run()* #We first have to create an operation to initialize the variables we created:

*for i in range(1000):  
  batch\_xs, batch\_ys = mnist.train.next\_batch(100)  
  sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})*

‘’’

Using small batches of random data is called stochastic training -- in this case, stochastic gradient descent. Ideally, we'd like to use all our data for every step of training because that would give us a better sense of what we should be doing, but that's expensive. So, instead, we use a different subset every time. Doing this is cheap and has much of the same benefit.

**tf.argmax**

Returns the index with the largest value across axes of a tensor.

**tf.equal**

Returns the truth value of (x == y) element-wise.

#### Returns:

A Tensor of type bool.

Defined in tensorflow/python/ops/gen\_math\_ops.py.

Y has a shape of (?,10). The “1” in tf.argmax(y,1) refers to argument[1], the “10” in (?,10) where 10 is the classes

‘’’

*correct\_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))*

*accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))*

*print(sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels}))*

**tf.cast**

# tensor `a` is [1.8, 2.2], dtype=tf.float

tf.cast(a, tf.**int32**) ==> [1, 2] # dtype=tf.int32

**tf.reduce\_mean**(input\_tensor, reduction\_indices=None, keep\_dims=False, name=None)

Computes the mean of elements across dimensions of a tensor.

For example:

# 'x' is [[1., 1.]

# [2., 2.]]

tf.reduce\_mean(x) ==> 1.5

tf.reduce\_mean(x, 0) ==> [1.5, 1.5]

tf.reduce\_mean(x, 1) ==> [1., 2.]

## Deep MNIST for Experts

### Weight Initialization

**tf.truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)**

Outputs random values from a truncated normal distribution.

The generated values follow a normal distribution with specified mean and standard deviation, ***except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.***

def weight\_variable(shape):  
  initial = tf.truncated\_normal(shape, stddev=0.1)  
  return tf.Variable(initial)  
  
def bias\_variable(shape):  
  initial = tf.constant(0.1, shape=shape)  
  return tf.Variable(initial)

### Convolution and Pooling

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks. To keep our code cleaner, let's also abstract those operations into functions.

def conv2d(x, W):  
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')  
  
def max\_pool\_2x2(x):  
  return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1],  
                        strides=[1, 2, 2, 1], padding='SAME')

Since each filter is 1x1, it will take 2x2=4 filters to cover every pixel

***Tf.nn.conv2d() example***

input = tf.Variable(tf.random\_normal([1,2,2,1]))

filter = tf.Variable(tf.random\_normal([1,1,1,1]))

op = tf.nn.conv2d(input, filter, strides=[1, 1, 1, 1], padding='SAME')

init = tf.initialize\_all\_variables()

with tf.Session() as sess:

sess.run(init)

print("input")

print(input.eval())

print("filter")

print(filter.eval())

print("result")

result = sess.run(op)

print(result)

‘’‘

input

[[[[ 1.60314465]

[-0.55022103]]

[[ 0.00595062]

[-0.69889867]]]]

filter

[[[[-0.59594476]]]]

result

[[[[-0.95538563]

[ 0.32790133]]

[[-0.00354624]

[ 0.41650501]]]]

’‘’

|  |  |
| --- | --- |
| Filter 1 | Filter 2 |
| Filter 3 | Filter 4 |

Where

Input[0]=no. of data points #1

Input[1],input[2]=width x height #2,2 or 2x2

Input[3]=channel #1

#hence 1,2,2,1

Where

Filter[0],flter[1]=filter width x height #1,1 or 1x1

Filter[2]=channel #1

Input[4]=no of filter#1

#hence 1,1,1,1

Each filter goes through processing to a new value, creating a 2x2 result

|  |  |
| --- | --- |
| Result 1 | Result 2 |
| Result 3 | Result 4 |

the resulting 2x2, 1 channel image (size 1x2x2x1, number of images x width x height x channels) is the result of multiplying the filter value by each pixel of the image.

### First Convolutional Layer

**W\_conv1 = weight\_variable([5, 5, 1, 32])# 5x5 dimension, input channel, no. of features/filter  
b\_conv1 = bias\_variable([32])**

**x\_image = tf.reshape(x, [-1,28,28,1])**

**no of colour channels**

img shape

**h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)  
h\_pool1 = max\_pool\_2x2(h\_conv1)**

this reduce 28x28 to 14x14

### Second Convolutional Layer

**W\_conv2 = weight\_variable([5, 5, 32, 64])  
b\_conv2 = bias\_variable([64])  
  
h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)  
h\_pool2 = max\_pool\_2x2(h\_conv2)**

this reduce 14x14 to 7x7

### Densely Connected Layer

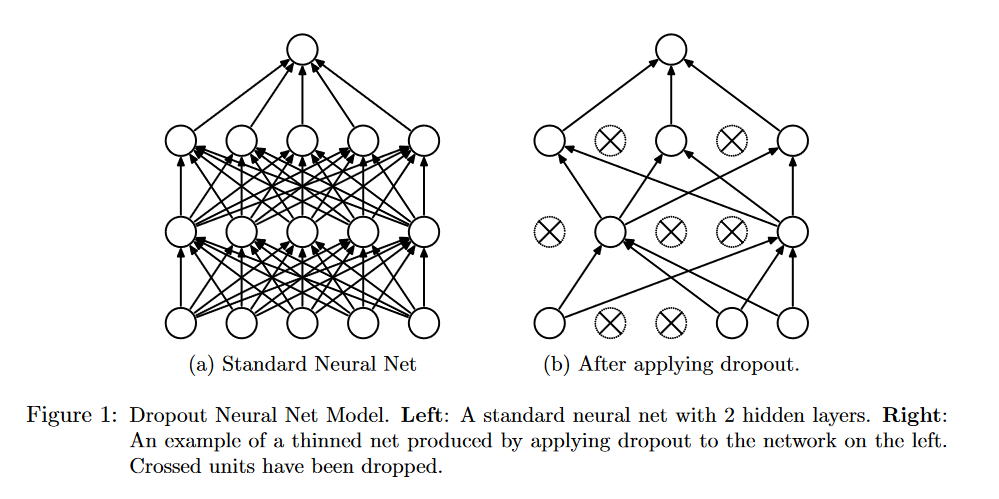
**W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024]) #1024 neurons  
b\_fc1 = bias\_variable([1024])  
  
h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])**

**h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)**

### Dropout

To reduce overfitting, we will apply [dropout](https://www.cs.toronto.edu/%7Ehinton/absps/JMLRdropout.pdf) before the readout layer. We create a placeholder for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's tf.nn.dropout op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.[1](https://www.tensorflow.org/get_started/mnist/pros#f1)

**keep\_prob = tf.placeholder(tf.float32)  
h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)**

****

### Readout Layer

**W\_fc2 = weight\_variable([1024, 10])  
b\_fc2 = bias\_variable([10])  
  
y\_conv = tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2**

### Train and Evaluate the Model

How well does this model do? To train and evaluate it we will use code that is nearly identical to that for the simple one layer SoftMax network above.

The differences are that:

* We will replace the steepest gradient descent optimizer with the more sophisticated ADAM optimizer.
* We will include the additional parameter keep\_prob in feed\_dict to control the dropout rate.
* We will add logging to every 100th iteration in the training process.

Feel free to go ahead and run this code, but it does 20,000 training iterations and may take a while (possibly up to half an hour), depending on your processor.

**cross\_entropy = tf.reduce\_mean(  
    tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_, logits=y\_conv))**

**train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)**

**correct\_prediction = tf.equal(tf.argmax(y\_conv,1), tf.argmax(y\_,1))**

**accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))**

**sess.run(tf.global\_variables\_initializer())**

**for i in range(20000):  
  batch = mnist.train.next\_batch(50)  
  if i%100 == 0:  
    train\_accuracy = accuracy.eval(feed\_dict={  
        x:batch[0], y\_: batch[1], keep\_prob: 1.0})  
    print("step %d, training accuracy %g"%(i, train\_accuracy))  
  train\_step.run(feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})  
  
print("test accuracy %g"%accuracy.eval(feed\_dict={  
    x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.0}))**

The final test set accuracy after running this code should be approximately 99.2%.

We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

**1**: For this small convolutional network, performance is actually nearly identical with and without dropout. Dropout is often very effective at reducing overfitting, but it is most useful when training very large neural networks.

## Visualization

let

***W = tf.Variable(tf.zeros([784, 10]))***

***b = tf.Variable(tf.zeros([10]))***

then, accessing W, b arrays would simpily be:

***sess.run(W)***

***sess.run(b)***

To record loss and entropy, do as following:

***for \_ in range(10):***

***batch\_xs, batch\_ys = mnist.train.next\_batch(100)***

***xx,xEn,wVal=sess.run([train\_step,cross\_entropy,W], feed\_dict={x: batch\_xs, y\_: batch\_ys})***

# END