Tutorials

This section contains tutorials demonstrating how to do specific tasks in TensorFlow. If you are new to TensorFlow, we recommend reading the documents in the "Get Started" section before reading these tutorials.

The following tutorial explains the interaction of CPUs and GPUs on a TensorFlow system:

* [Using GPUs](https://www.tensorflow.org/tutorials/using_gpu)

The following tutorials cover different aspects of image recognition:

* [Image Recognition](https://www.tensorflow.org/tutorials/image_recognition), which introduces the field of image recognition and a model (Inception) for recognizing images.
* [How to Retrain Inception's Final Layer for New Categories](https://www.tensorflow.org/tutorials/image_retraining), which has a wonderfully self-explanatory title.
* [A Guide to TF Layers: Building a Convolutional Neural Network](https://www.tensorflow.org/tutorials/layers), which introduces convolutional neural networks (CNNs) and demonstrates how to build a CNN in TensorFlow.
* [Convolutional Neural Networks](https://www.tensorflow.org/tutorials/deep_cnn), which demonstrates how to build a small CNN for recognizing images. This tutorial is aimed at advanced TensorFlow users.

The following tutorials focus on machine learning problems in human language:

* [Vector Representations of Words](https://www.tensorflow.org/tutorials/word2vec), which demonstrates how to create an embedding for words.
* [Recurrent Neural Networks](https://www.tensorflow.org/tutorials/recurrent), which demonstrates how to use a recurrent neural network to predict the next word in a sentence.
* [Sequence-to-Sequence Models](https://www.tensorflow.org/tutorials/seq2seq), which demonstrates how to use a sequence-to-sequence model to translate text from English to French.

The following tutorials focus on linear models:

* [Large-Scale Linear Models with TensorFlow](https://www.tensorflow.org/tutorials/linear), which introduces linear models and demonstrates how to build them with the high-level API.
* [TensorFlow Linear Model Tutorial](https://www.tensorflow.org/tutorials/wide), which demonstrates how to solve a binary classification problem in TensorFlow.
* [TensorFlow Wide & Deep Learning Tutorial](https://www.tensorflow.org/tutorials/wide_and_deep), which explains how to use the high-level API to jointly train both a wide linear model and a deep feed-forward neural network.

Although TensorFlow specializes in machine learning, you may also use TensorFlow to solve other kinds of math problems. For example:

* [Mandelbrot Set](https://www.tensorflow.org/tutorials/mandelbrot)
* [Partial Differential Equations](https://www.tensorflow.org/tutorials/pdes)

# Using GPUs

## Supported devices

On a typical system, there are multiple computing devices. In TensorFlow, the supported device types are CPU and GPU. They are represented as strings. For example:

* "/cpu:0": The CPU of your machine.
* "/gpu:0": The GPU of your machine, if you have one.
* "/gpu:1": The second GPU of your machine, etc.

If a TensorFlow operation has both CPU and GPU implementations, the GPU devices will be given priority when the operation is assigned to a device. For example, matmul has both CPU and GPU kernels. On a system with devices cpu:0 and gpu:0, gpu:0 will be selected to run matmul.

## Logging Device placement

To find out which devices your operations and tensors are assigned to, create the session with log\_device\_placement configuration option set to True.

# Creates a graph.  
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')  
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')  
c = tf.matmul(a, b)  
# Creates a session with log\_device\_placement set to True.  
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))  
# Runs the op.  
print(sess.run(c))

You should see the following output:

Device mapping:  
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus  
id: 0000:05:00.0  
b: /job:localhost/replica:0/task:0/gpu:0  
a: /job:localhost/replica:0/task:0/gpu:0  
MatMul: /job:localhost/replica:0/task:0/gpu:0  
[[ 22.  28.]  
 [ 49.  64.]]

## Manual device placement

If you would like a particular operation to run on a device of your choice instead of what's automatically selected for you, you can use with tf.device to create a device context such that all the operations within that context will have the same device assignment.

# Creates a graph.  
with tf.device('/cpu:0'):  
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')  
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')  
  c = tf.matmul(a, b)  
# Creates a session with log\_device\_placement set to True.  
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))  
# Runs the op.  
print(sess.run(c))

You will see that now a and b are assigned to cpu:0.

Device mapping:  
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus  
id: 0000:05:00.0  
b: /job:localhost/replica:0/task:0/cpu:0  
a: /job:localhost/replica:0/task:0/cpu:0  
MatMul: /job:localhost/replica:0/task:0/gpu:0  
[[ 22.  28.]  
 [ 49.  64.]]

## Allowing GPU memory growth

By default, TensorFlow maps nearly all of the GPU memory of all GPUs (subject to[CUDA\_VISIBLE\_DEVICES](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#env-vars)) visible to the process. This is done to more efficiently use the relatively precious GPU memory resources on the devices by reducing [memory fragmentation](https://en.wikipedia.org/wiki/Fragmentation_(computing)).

In some cases it is desirable for the process to only allocate a subset of the available memory, or to only grow the memory usage as is needed by the process. TensorFlow provides two Config options on the Session to control this.

The first is the allow\_growth option, which attempts to allocate only as much GPU memory based on runtime allocations: it starts out allocating very little memory, and as Sessions get run and more GPU memory is needed, we extend the GPU memory region needed by the TensorFlow process. Note that we do not release memory, since that can lead to even worse memory fragmentation. To turn this option on, set the option in the ConfigProto by:

config = tf.ConfigProto()  
config.gpu\_options.allow\_growth = True  
session = tf.Session(config=config, ...)

The second method is the per\_process\_gpu\_memory\_fraction option, which determines the fraction of the overall amount of memory that each visible GPU should be allocated. For example, you can tell TensorFlow to only allocate 40% of the total memory of each GPU by:

config = tf.ConfigProto()  
config.gpu\_options.per\_process\_gpu\_memory\_fraction = 0.4  
session = tf.Session(config=config, ...)

This is useful if you want to truly bound the amount of GPU memory available to the TensorFlow process.

## Using a single GPU on a multi-GPU system

If you have more than one GPU in your system, the GPU with the lowest ID will be selected by default. If you would like to run on a different GPU, you will need to specify the preference explicitly:

# Creates a graph.  
with tf.device('/gpu:2'):  
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')  
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')  
  c = tf.matmul(a, b)  
# Creates a session with log\_device\_placement set to True.  
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))  
# Runs the op.  
print(sess.run(c))

If the device you have specified does not exist, you will get InvalidArgumentError:

InvalidArgumentError: Invalid argument: Cannot assign a device to node 'b':  
Could not satisfy explicit device specification '/gpu:2'  
   [[Node: b = Const[dtype=DT\_FLOAT, value=Tensor<type: float shape: [3,2]  
   values: 1 2 3...>, \_device="/gpu:2"]()]]

If you would like TensorFlow to automatically choose an existing and supported device to run the operations in case the specified one doesn't exist, you can set allow\_soft\_placement to True in the configuration option when creating the session.

# Creates a graph.  
with tf.device('/gpu:2'):  
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')  
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')  
  c = tf.matmul(a, b)  
# Creates a session with allow\_soft\_placement and log\_device\_placement set  
# to True.  
sess = tf.Session(config=tf.ConfigProto(  
      allow\_soft\_placement=True, log\_device\_placement=True))  
# Runs the op.  
print(sess.run(c))

## Using multiple GPUs

If you would like to run TensorFlow on multiple GPUs, you can construct your model in a multi-tower fashion where each tower is assigned to a different GPU. For example:

# Creates a graph.  
c = []  
for d in ['/gpu:2', '/gpu:3']:  
  with tf.device(d):  
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])  
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])  
    c.append(tf.matmul(a, b))  
with tf.device('/cpu:0'):  
  sum = tf.add\_n(c)  
# Creates a session with log\_device\_placement set to True.  
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))  
# Runs the op.  
print(sess.run(sum))

You will see the following output.

Device mapping:  
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K20m, pci bus  
id: 0000:02:00.0  
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K20m, pci bus  
id: 0000:03:00.0  
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K20m, pci bus  
id: 0000:83:00.0  
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K20m, pci bus  
id: 0000:84:00.0  
Const\_3: /job:localhost/replica:0/task:0/gpu:3  
Const\_2: /job:localhost/replica:0/task:0/gpu:3  
MatMul\_1: /job:localhost/replica:0/task:0/gpu:3  
Const\_1: /job:localhost/replica:0/task:0/gpu:2  
Const: /job:localhost/replica:0/task:0/gpu:2  
MatMul: /job:localhost/replica:0/task:0/gpu:2  
AddN: /job:localhost/replica:0/task:0/cpu:0  
[[  44.   56.]  
 [  98.  128.]]

The [cifar10 tutorial](https://www.tensorflow.org/tutorials/deep_cnn) is a good example demonstrating how to do training with multiple GPUs.

# Image Recognition

Our brains make vision seem easy. It doesn't take any effort for humans to tell apart a lion and a jaguar, read a sign, or recognize a human's face. But these are actually hard problems to solve with a computer: they only seem easy because our brains are incredibly good at understanding images.

In the last few years the field of machine learning has made tremendous progress on addressing these difficult problems. In particular, we've found that a kind of model called a deep [convolutional neural network](http://colah.github.io/posts/2014-07-Conv-Nets-Modular/) can achieve reasonable performance on hard visual recognition tasks -- matching or exceeding human performance in some domains.

Researchers have demonstrated steady progress in computer vision by validating their work against [ImageNet](http://www.image-net.org/) -- an academic benchmark for computer vision. Successive models continue to show improvements, each time achieving a new state-of-the-art result: [QuocNet](http://static.googleusercontent.com/media/research.google.com/en/archive/unsupervised_icml2012.pdf), [AlexNet](http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf), [Inception (GoogLeNet)](http://arxiv.org/abs/1409.4842), [BN-Inception-v2](http://arxiv.org/abs/1502.03167). Researchers both internal and external to Google have published papers describing all these models but the results are still hard to reproduce. We're now taking the next step by releasing code for running image recognition on our latest model, [Inception-v3](http://arxiv.org/abs/1512.00567).

Inception-v3 is trained for the [ImageNet](http://image-net.org/) Large Visual Recognition Challenge using the data from 2012. This is a standard task in computer vision, where models try to classify entire images into [1000 classes](http://image-net.org/challenges/LSVRC/2014/browse-synsets), like "Zebra", "Dalmatian", and "Dishwasher". For example, here are the results from [AlexNet](http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf) classifying some images:

To compare models, we examine how often the model fails to predict the correct answer as one of their top 5 guesses -- termed "top-5 error rate". [AlexNet](http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf) achieved by setting a top-5 error rate of 15.3% on the 2012 validation data set; [Inception (GoogLeNet)](http://arxiv.org/abs/1409.4842) achieved 6.67%; [BN-Inception-v2](http://arxiv.org/abs/1502.03167) achieved 4.9%; [Inception-v3](http://arxiv.org/abs/1512.00567) reaches 3.46%.

How well do humans do on ImageNet Challenge? There's a [blog post](http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/) by Andrej Karpathy who attempted to measure his own performance. He reached 5.1% top-5 error rate.

This tutorial will teach you how to use [Inception-v3](http://arxiv.org/abs/1512.00567). You'll learn how to classify images into [1000 classes](http://image-net.org/challenges/LSVRC/2014/browse-synsets) in Python or C++. We'll also discuss how to extract higher level features from this model which may be reused for other vision tasks.

We're excited to see what the community will do with this model.

## Usage with Python API

classify\_image.py downloads the trained model from tensorflow.org when the program is run for the first time. You'll need about 200M of free space available on your hard disk.

Start by cloning the [TensorFlow models repo](https://github.com/tensorflow/models) from GitHub. Run the following commands:

cd models/tutorials/image/imagenet  
python classify\_image.py

The above command will classify a supplied image of a panda bear.

If the model runs correctly, the script will produce the following output:

giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score = 0.88493)  
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00878)  
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score = 0.00317)  
custard apple (score = 0.00149)  
earthstar (score = 0.00127)

If you wish to supply other JPEG images, you may do so by editing the --image\_file argument.

If you download the model data to a different directory, you will need to point --model\_dir to the directory used.

## Usage with the C++ API

You can run the same [Inception-v3](http://arxiv.org/abs/1512.00567) model in C++ for use in production environments. You can download the archive containing the GraphDef that defines the model like this (running from the root directory of the TensorFlow repository):

curl -L "https://storage.googleapis.com/download.tensorflow.org/models/inception\_v3\_2016\_08\_28\_frozen.pb.tar.gz" |  
  tar -C tensorflow/examples/label\_image/data -xz

Next, we need to compile the C++ binary that includes the code to load and run the graph. If you've followed [the instructions to download the source installation of TensorFlow](https://www.tensorflow.org/install/install_sources) for your platform, you should be able to build the example by running this command from your shell terminal:

bazel build tensorflow/examples/label\_image/...

That should create a binary executable that you can then run like this:

bazel-bin/tensorflow/examples/label\_image/label\_image

This uses the default example image that ships with the framework, and should output something similar to this:

I tensorflow/examples/label\_image/main.cc:206] military uniform (653): 0.834306  
I tensorflow/examples/label\_image/main.cc:206] mortarboard (668): 0.0218692  
I tensorflow/examples/label\_image/main.cc:206] academic gown (401): 0.0103579  
I tensorflow/examples/label\_image/main.cc:206] pickelhaube (716): 0.00800814  
I tensorflow/examples/label\_image/main.cc:206] bulletproof vest (466): 0.00535088

In this case, we're using the default image of [Admiral Grace Hopper](https://en.wikipedia.org/wiki/Grace_Hopper), and you can see the network correctly identifies she's wearing a military uniform, with a high score of 0.8.

Next, try it out on your own images by supplying the --image= argument, e.g.

bazel-bin/tensorflow/examples/label\_image/label\_image --image=my\_image.png

If you look inside the [tensorflow/examples/label\_image/main.cc](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc) file, you can find out how it works. We hope this code will help you integrate TensorFlow into your own applications, so we will walk step by step through the main functions:

The command line flags control where the files are loaded from, and properties of the input images. The model expects to get square 299x299 RGB images, so those are the input\_width and input\_height flags. We also need to scale the pixel values from integers that are between 0 and 255 to the floating point values that the graph operates on. We control the scaling with the input\_mean and input\_std flags: we first subtract input\_meanfrom each pixel value, then divide it by input\_std.

These values probably look somewhat magical, but they are just defined by the original model author based on what he/she wanted to use as input images for training. If you have a graph that you've trained yourself, you'll just need to adjust the values to match whatever you used during your training process.

You can see how they're applied to an image in the [ReadTensorFromImageFile()](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L88) function.

// Given an image file name, read in the data, try to decode it as an image,  
// resize it to the requested size, and then scale the values as desired.  
Status ReadTensorFromImageFile(string file\_name, const int input\_height,  
                               const int input\_width, const float input\_mean,  
                               const float input\_std,  
                               std::vector<Tensor>\* out\_tensors) {  
  tensorflow::GraphDefBuilder b;

We start by creating a GraphDefBuilder, which is an object we can use to specify a model to run or load.

  string input\_name = "file\_reader";  
  string output\_name = "normalized";  
  tensorflow::Node\* file\_reader =  
      tensorflow::ops::ReadFile(tensorflow::ops::Const(file\_name, b.opts()),  
                                b.opts().WithName(input\_name));

We then start creating nodes for the small model we want to run to load, resize, and scale the pixel values to get the result the main model expects as its input. The first node we create is just a Const op that holds a tensor with the file name of the image we want to load. That's then passed as the first input to the ReadFile op. You might notice we're passing b.opts() as the last argument to all the op creation functions. The argument ensures that the node is added to the model definition held in the GraphDefBuilder. We also name the ReadFile operator by making the WithName() call to b.opts(). This gives a name to the node, which isn't strictly necessary since an automatic name will be assigned if you don't do this, but it does make debugging a bit easier.

  // Now try to figure out what kind of file it is and decode it.  
  const int wanted\_channels = 3;  
  tensorflow::Node\* image\_reader;  
  if (tensorflow::StringPiece(file\_name).ends\_with(".png")) {  
    image\_reader = tensorflow::ops::DecodePng(  
        file\_reader,  
        b.opts().WithAttr("channels", wanted\_channels).WithName("png\_reader"));  
  } else {  
    // Assume if it's not a PNG then it must be a JPEG.  
    image\_reader = tensorflow::ops::DecodeJpeg(  
        file\_reader,  
        b.opts().WithAttr("channels", wanted\_channels).WithName("jpeg\_reader"));  
  }  
  // Now cast the image data to float so we can do normal math on it.  
  tensorflow::Node\* float\_caster = tensorflow::ops::Cast(  
      image\_reader, tensorflow::DT\_FLOAT, b.opts().WithName("float\_caster"));  
  // The convention for image ops in TensorFlow is that all images are expected  
  // to be in batches, so that they're four-dimensional arrays with indices of  
  // [batch, height, width, channel]. Because we only have a single image, we  
  // have to add a batch dimension of 1 to the start with ExpandDims().  
  tensorflow::Node\* dims\_expander = tensorflow::ops::ExpandDims(  
      float\_caster, tensorflow::ops::Const(0, b.opts()), b.opts());  
  // Bilinearly resize the image to fit the required dimensions.  
  tensorflow::Node\* resized = tensorflow::ops::ResizeBilinear(  
      dims\_expander, tensorflow::ops::Const({input\_height, input\_width},  
                                            b.opts().WithName("size")),  
      b.opts());  
  // Subtract the mean and divide by the scale.  
  tensorflow::ops::Div(  
      tensorflow::ops::Sub(  
          resized, tensorflow::ops::Const({input\_mean}, b.opts()), b.opts()),  
      tensorflow::ops::Const({input\_std}, b.opts()),  
      b.opts().WithName(output\_name));

We then keep adding more nodes, to decode the file data as an image, to cast the integers into floating point values, to resize it, and then finally to run the subtraction and division operations on the pixel values.

  // This runs the GraphDef network definition that we've just constructed, and  
  // returns the results in the output tensor.  
  tensorflow::GraphDef graph;  
  TF\_RETURN\_IF\_ERROR(b.ToGraphDef(&graph));

At the end of this we have a model definition stored in the b variable, which we turn into a full graph definition with the ToGraphDef() function.

  std::unique\_ptr<tensorflow::Session> session(  
      tensorflow::NewSession(tensorflow::SessionOptions()));  
  TF\_RETURN\_IF\_ERROR(session->Create(graph));  
  TF\_RETURN\_IF\_ERROR(session->Run({}, {output\_name}, {}, out\_tensors));  
  return Status::OK();

Then we create a [tf.Session](https://www.tensorflow.org/api_docs/python/tf/Session) object, which is the interface to actually running the graph, and run it, specifying which node we want to get the output from, and where to put the output data.

This gives us a vector of Tensor objects, which in this case we know will only be a single object long. You can think of a Tensor as a multi-dimensional array in this context, and it holds a 299 pixel high, 299 pixel wide, 3 channel image as float values. If you have your own image-processing framework in your product already, you should be able to use that instead, as long as you apply the same transformations before you feed images into the main graph.

This is a simple example of creating a small TensorFlow graph dynamically in C++, but for the pre-trained Inception model we want to load a much larger definition from a file. You can see how we do that in the LoadGraph() function.

// Reads a model graph definition from disk, and creates a session object you  
// can use to run it.  
Status LoadGraph(string graph\_file\_name,  
                 std::unique\_ptr<tensorflow::Session>\* session) {  
  tensorflow::GraphDef graph\_def;  
  Status load\_graph\_status =  
      ReadBinaryProto(tensorflow::Env::Default(), graph\_file\_name, &graph\_def);  
  if (!load\_graph\_status.ok()) {  
    return tensorflow::errors::NotFound("Failed to load compute graph at '",  
                                        graph\_file\_name, "'");  
  }

If you've looked through the image loading code, a lot of the terms should seem familiar. Rather than using a GraphDefBuilder to produce a GraphDef object, we load a protobuf file that directly contains the GraphDef.

  session->reset(tensorflow::NewSession(tensorflow::SessionOptions()));  
  Status session\_create\_status = (\*session)->Create(graph\_def);  
  if (!session\_create\_status.ok()) {  
    return session\_create\_status;  
  }  
  return Status::OK();  
}

Then we create a Session object from that GraphDef and pass it back to the caller so that they can run it at a later time.

The GetTopLabels() function is a lot like the image loading, except that in this case we want to take the results of running the main graph, and turn it into a sorted list of the highest-scoring labels. Just like the image loader, it creates a GraphDefBuilder, adds a couple of nodes to it, and then runs the short graph to get a pair of output tensors. In this case they represent the sorted scores and index positions of the highest results.

// Analyzes the output of the Inception graph to retrieve the highest scores and  
// their positions in the tensor, which correspond to categories.  
Status GetTopLabels(const std::vector<Tensor>& outputs, int how\_many\_labels,  
                    Tensor\* indices, Tensor\* scores) {  
  tensorflow::GraphDefBuilder b;  
  string output\_name = "top\_k";  
  tensorflow::ops::TopK(tensorflow::ops::Const(outputs[0], b.opts()),  
                        how\_many\_labels, b.opts().WithName(output\_name));  
  // This runs the GraphDef network definition that we've just constructed, and  
  // returns the results in the output tensors.  
  tensorflow::GraphDef graph;  
  TF\_RETURN\_IF\_ERROR(b.ToGraphDef(&graph));  
  std::unique\_ptr<tensorflow::Session> session(  
      tensorflow::NewSession(tensorflow::SessionOptions()));  
  TF\_RETURN\_IF\_ERROR(session->Create(graph));  
  // The TopK node returns two outputs, the scores and their original indices,  
  // so we have to append :0 and :1 to specify them both.  
  std::vector<Tensor> out\_tensors;  
  TF\_RETURN\_IF\_ERROR(session->Run({}, {output\_name + ":0", output\_name + ":1"},  
                                  {}, &out\_tensors));  
  \*scores = out\_tensors[0];  
  \*indices = out\_tensors[1];  
  return Status::OK();

The PrintTopLabels() function takes those sorted results, and prints them out in a friendly way. The CheckTopLabel() function is very similar, but just makes sure that the top label is the one we expect, for debugging purposes.

At the end, [main()](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L252) ties together all of these calls.

int main(int argc, char\* argv[]) {  
  // We need to call this to set up global state for TensorFlow.  
  tensorflow::port::InitMain(argv[0], &argc, &argv);  
  Status s = tensorflow::ParseCommandLineFlags(&argc, argv);  
  if (!s.ok()) {  
    LOG(ERROR) << "Error parsing command line flags: " << s.ToString();  
    return -1;  
  }  
  
  // First we load and initialize the model.  
  std::unique\_ptr<tensorflow::Session> session;  
  string graph\_path = tensorflow::io::JoinPath(FLAGS\_root\_dir, FLAGS\_graph);  
  Status load\_graph\_status = LoadGraph(graph\_path, &session);  
  if (!load\_graph\_status.ok()) {  
    LOG(ERROR) << load\_graph\_status;  
    return -1;  
  }

We load the main graph.

  // Get the image from disk as a float array of numbers, resized and normalized  
  // to the specifications the main graph expects.  
  std::vector<Tensor> resized\_tensors;  
  string image\_path = tensorflow::io::JoinPath(FLAGS\_root\_dir, FLAGS\_image);  
  Status read\_tensor\_status = ReadTensorFromImageFile(  
      image\_path, FLAGS\_input\_height, FLAGS\_input\_width, FLAGS\_input\_mean,  
      FLAGS\_input\_std, &resized\_tensors);  
  if (!read\_tensor\_status.ok()) {  
    LOG(ERROR) << read\_tensor\_status;  
    return -1;  
  }  
  const Tensor& resized\_tensor = resized\_tensors[0];

Load, resize, and process the input image.

  // Actually run the image through the model.  
  std::vector<Tensor> outputs;  
  Status run\_status = session->Run({ {FLAGS\_input\_layer, resized\_tensor}},  
                                   {FLAGS\_output\_layer}, {}, &outputs);  
  if (!run\_status.ok()) {  
    LOG(ERROR) << "Running model failed: " << run\_status;  
    return -1;  
  }

Here we run the loaded graph with the image as an input.

  // This is for automated testing to make sure we get the expected result with  
  // the default settings. We know that label 866 (military uniform) should be  
  // the top label for the Admiral Hopper image.  
  if (FLAGS\_self\_test) {  
    bool expected\_matches;  
    Status check\_status = CheckTopLabel(outputs, 866, &expected\_matches);  
    if (!check\_status.ok()) {  
      LOG(ERROR) << "Running check failed: " << check\_status;  
      return -1;  
    }  
    if (!expected\_matches) {  
      LOG(ERROR) << "Self-test failed!";  
      return -1;  
    }  
  }

For testing purposes we can check to make sure we get the output we expect here.

  // Do something interesting with the results we've generated.  
  Status print\_status = PrintTopLabels(outputs, FLAGS\_labels);

Finally we print the labels we found.

  if (!print\_status.ok()) {  
    LOG(ERROR) << "Running print failed: " << print\_status;  
    return -1;  
  }

The error handling here is using TensorFlow's Status object, which is very convenient because it lets you know whether any error has occurred with the ok() checker, and then can be printed out to give a readable error message.

In this case we are demonstrating object recognition, but you should be able to use very similar code on other models you've found or trained yourself, across all sorts of domains. We hope this small example gives you some ideas on how to use TensorFlow within your own products.

**EXERCISE**: Transfer learning is the idea that, if you know how to solve a task well, you should be able to transfer some of that understanding to solving related problems. One way to perform transfer learning is to remove the final classification layer of the network and extract the [next-to-last layer of the CNN](http://arxiv.org/abs/1310.1531), in this case a 2048 dimensional vector. There's a guide to doing this [in the how-to section](https://www.tensorflow.org/tutorials/image_retraining).

## Resources for Learning More

To learn about neural networks in general, Michael Nielsen's [free online book](http://neuralnetworksanddeeplearning.com/chap1.html) is an excellent resource. For convolutional neural networks in particular, Chris Olah has some [nice blog posts](http://colah.github.io/posts/2014-07-Conv-Nets-Modular/), and Michael Nielsen's book has a [great chapter](http://neuralnetworksanddeeplearning.com/chap6.html) covering them.

To find out more about implementing convolutional neural networks, you can jump to the TensorFlow [deep convolutional networks tutorial](https://www.tensorflow.org/tutorials/deep_cnn), or start a bit more gently with our [ML beginner](https://www.tensorflow.org/get_started/mnist/beginners) or [ML expert](https://www.tensorflow.org/get_started/mnist/pros) MNIST starter tutorials. Finally, if you want to get up to speed on research in this area, you can read the recent work of all the papers referenced in this tutorial.

# How to Retrain Inception's Final Layer for New Categories

Modern object recognition models have millions of parameters and can take weeks to fully train. Transfer learning is a technique that shortcuts a lot of this work by taking a fully-trained model for a set of categories like ImageNet, and retrains from the existing weights for new classes. In this example we'll be retraining the final layer from scratch, while leaving all the others untouched. For more information on the approach you can see [this paper on Decaf](http://arxiv.org/pdf/1310.1531v1.pdf).

Though it's not as good as a full training run, this is surprisingly effective for many applications, and can be run in as little as thirty minutes on a laptop, without requiring a GPU. This tutorial will show you how to run the example script on your own images, and will explain some of the options you have to help control the training process.

## Training on Flowers

 [Image by Kelly Sikkema](https://www.flickr.com/photos/95072945@N05/9922116524/)

Before you start any training, you'll need a set of images to teach the network about the new classes you want to recognize. There's a later section that explains how to prepare your own images, but to make it easy we've created an archive of creative-commons licensed flower photos to use initially. To get the set of flower photos, run these commands:

cd ~  
curl -O http://download.tensorflow.org/example\_images/flower\_photos.tgz  
tar xzf flower\_photos.tgz

Once you have the images, you can build the retrainer like this, from the root of your TensorFlow source directory:

bazel build tensorflow/examples/image\_retraining:retrain

If you have a machine which supports [the AVX instruction set](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions) (common in x86 CPUs produced in the last few years) you can improve the running speed of the retraining by building for that architecture, like this (after choosing appropriate options in configure):

bazel build --config opt tensorflow/examples/image\_retraining:retrain

The retrainer can then be run like this:

bazel-bin/tensorflow/examples/image\_retraining/retrain --image\_dir ~/flower\_photos

This script loads the pre-trained Inception v3 model, removes the old top layer, and trains a new one on the flower photos you've downloaded. None of the flower species were in the original ImageNet classes the full network was trained on. The magic of transfer learning is that lower layers that have been trained to distinguish between some objects can be reused for many recognition tasks without any alteration.

## Bottlenecks

The script can take thirty minutes or more to complete, depending on the speed of your machine. The first phase analyzes all the images on disk and calculates the bottleneck values for each of them. 'Bottleneck' is an informal term we often use for the layer just before the final output layer that actually does the classification. This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize. That means it has to be a meaningful and compact summary of the images, since it has to contain enough information for the classifier to make a good choice in a very small set of values. The reason our final layer retraining can work on new classes is that it turns out the kind of information needed to distinguish between all the 1,000 classes in ImageNet is often also useful to distinguish between new kinds of objects.

Because every image is reused multiple times during training and calculating each bottleneck takes a significant amount of time, it speeds things up to cache these bottleneck values on disk so they don't have to be repeatedly recalculated. By default they're stored in the /tmp/bottleneckdirectory, and if you rerun the script they'll be reused so you don't have to wait for this part again.

## Training

Once the bottlenecks are complete, the actual training of the top layer of the network begins. You'll see a series of step outputs, each one showing training accuracy, validation accuracy, and the cross entropy. The training accuracy shows what percent of the images used in the current training batch were labeled with the correct class. The validation accuracy is the precision on a randomly-selected group of images from a different set. The key difference is that the training accuracy is based on images that the network has been able to learn from so the network can overfit to the noise in the training data. A true measure of the performance of the network is to measure its performance on a data set not contained in the training data -- this is measured by the validation accuracy. If the train accuracy is high but the validation accuracy remains low, that means the network is overfitting and memorizing particular features in the training images that aren't helpful more generally. Cross entropy is a loss function which gives a glimpse into how well the learning process is progressing. The training's objective is to make the loss as small as possible, so you can tell if the learning is working by keeping an eye on whether the loss keeps trending downwards, ignoring the short-term noise.

By default this script will run 4,000 training steps. Each step chooses ten images at random from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions. Those predictions are then compared against the actual labels to update the final layer's weights through the back-propagation process. As the process continues you should see the reported accuracy improve, and after all the steps are done, a final test accuracy evaluation is run on a set of images kept separate from the training and validation pictures. This test evaluation is the best estimate of how the trained model will perform on the classification task. You should see an accuracy value of between 90% and 95%, though the exact value will vary from run to run since there's randomness in the training process. This number is based on the percent of the images in the test set that are given the correct label after the model is fully trained.

## Visualizing the Retraining with TensorBoard

The script includes TensorBoard summaries that make it easier to understand, debug, and optimize the retraining. For example, you can visualize the graph and statistics, such as how the weights or accuracy varied during training.

To launch TensorBoard, run this command during or after retraining:

tensorboard --logdir /tmp/retrain\_logs

Once TensorBoard is running, navigate your web browser to localhost:6006 to view the TensorBoard.

The script will log TensorBoard summaries to /tmp/retrain\_logs by default. You can change the directory with the --summaries\_dir flag.

The [TensorBoard README](https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/tensorboard/README.md) has a lot more information on TensorBoard usage, including tips & tricks, and debugging information.

## Using the Retrained Model

The script will write out a version of the Inception v3 network with a final layer retrained to your categories to /tmp/output\_graph.pb, and a text file containing the labels to /tmp/output\_labels.txt. These are both in a format that the [C++ and Python image classification examples](https://www.tensorflow.org/tutorials/image_recognition) can read in, so you can start using your new model immediately. Since you've replaced the top layer, you will need to specify the new name in the script, for example with the flag --output\_layer=final\_result if you're using label\_image.

Here's an example of how to build and run the label\_image example with your retrained graphs:

bazel build tensorflow/examples/label\_image:label\_image && \  
bazel-bin/tensorflow/examples/label\_image/label\_image \  
--graph=/tmp/output\_graph.pb --labels=/tmp/output\_labels.txt \  
--output\_layer=final\_result \  
--image=$HOME/flower\_photos/daisy/21652746\_cc379e0eea\_m.jpg

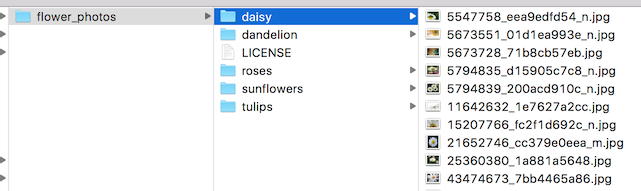
You should see a list of flower labels, in most cases with daisy on top (though each retrained model may be slightly different). You can replace the --image parameter with your own images to try those out, and use the C++ code as a template to integrate with your own applications.

If you'd like to use the retrained model in a Python program [this example from @eldor4do shows what you'll need to do](https://github.com/eldor4do/TensorFlow-Examples/blob/master/retraining-example.py).

## Training on Your Own Categories

If you've managed to get the script working on the flower example images, you can start looking at teaching it to recognize categories you care about instead. In theory all you'll need to do is point it at a set of sub-folders, each named after one of your categories and containing only images from that category. If you do that and pass the root folder of the subdirectories as the argument to --image\_dir, the script should train just like it did for the flowers.

Here's what the folder structure of the flowers archive looks like, to give you and example of the kind of layout the script is looking for:



In practice it may take some work to get the accuracy you want. I'll try to guide you through some of the common problems you might encounter below.

## Creating a Set of Training Images

The first place to start is by looking at the images you've gathered, since the most common issues we see with training come from the data that's being fed in.

For training to work well, you should gather at least a hundred photos of each kind of object you want to recognize. The more you can gather, the better the accuracy of your trained model is likely to be. You also need to make sure that the photos are a good representation of what your application will actually encounter. For example, if you take all your photos indoors against a blank wall and your users are trying to recognize objects outdoors, you probably won't see good results when you deploy.

Another pitfall to avoid is that the learning process will pick up on anything that the labeled images have in common with each other, and if you're not careful that might be something that's not useful. For example if you photograph one kind of object in a blue room, and another in a green one, then the model will end up basing its prediction on the background color, not the features of the object you actually care about. To avoid this, try to take pictures in as wide a variety of situations as you can, at different times, and with different devices. If you want to know more about this problem, you can read about the classic (and possibly apocryphal) [tank recognition problem](http://www.jefftk.com/p/detecting-tanks).

You may also want to think about the categories you use. It might be worth splitting big categories that cover a lot of different physical forms into smaller ones that are more visually distinct. For example instead of 'vehicle' you might use 'car', 'motorbike', and 'truck'. It's also worth thinking about whether you have a 'closed world' or an 'open world' problem. In a closed world, the only things you'll ever be asked to categorize are the classes of object you know about. This might apply to a plant recognition app where you know the user is likely to be taking a picture of a flower, so all you have to do is decide which species. By contrast a roaming robot might see all sorts of different things through its camera as it wanders around the world. In that case you'd want the classifier to report if it wasn't sure what it was seeing. This can be hard to do well, but often if you collect a large number of typical 'background' photos with no relevant objects in them, you can add them to an extra 'unknown' class in your image folders.

It's also worth checking to make sure that all of your images are labeled correctly. Often user-generated tags are unreliable for our purposes, for example using #daisy for pictures of a person named Daisy. If you go through your images and weed out any mistakes it can do wonders for your overall accuracy.

## Training Steps

If you're happy with your images, you can take a look at improving your results by altering the details of the learning process. The simplest one to try is --how\_many\_training\_steps. This defaults to 4,000, but if you increase it to 8,000 it will train for twice as long. The rate of improvement in the accuracy slows the longer you train for, and at some point will stop altogether, but you can experiment to see when you hit that limit for your model.

## Distortions

A common way of improving the results of image training is by deforming, cropping, or brightening the training inputs in random ways. This has the advantage of expanding the effective size of the training data thanks to all the possible variations of the same images, and tends to help the network learn to cope with all the distortions that will occur in real-life uses of the classifier. The biggest disadvantage of enabling these distortions in our script is that the bottleneck caching is no longer useful, since input images are never reused exactly. This means the training process takes a lot longer, so I recommend trying this as a way of fine-tuning your model once you've got one that you're reasonably happy with.

You enable these distortions by passing --random\_crop, --random\_scale and --random\_brightness to the script. These are all percentage values that control how much of each of the distortions is applied to each image. It's reasonable to start with values of 5 or 10 for each of them and then experiment to see which of them help with your application. --flip\_left\_rightwill randomly mirror half of the images horizontally, which makes sense as long as those inversions are likely to happen in your application. For example it wouldn't be a good idea if you were trying to recognize letters, since flipping them destroys their meaning.

## Hyper-parameters

There are several other parameters you can try adjusting to see if they help your results. The --learning\_rate controls the magnitude of the updates to the final layer during training. Intuitively if this is smaller then the learning will take longer, but it can end up helping the overall precision. That's not always the case though, so you need to experiment carefully to see what works for your case. The --train\_batch\_size controls how many images are examined during one training step, and because the learning rate is applied per batch you'll need to reduce it if you have larger batches to get the same overall effect.

## Training, Validation, and Testing Sets

One of the things the script does under the hood when you point it at a folder of images is divide them up into three different sets. The largest is usually the training set, which are all the images fed into the network during training, with the results used to update the model's weights. You might wonder why we don't use all the images for training? A big potential problem when we're doing machine learning is that our model may just be memorizing irrelevant details of the training images to come up with the right answers. For example, you could imagine a network remembering a pattern in the background of each photo it was shown, and using that to match labels with objects. It could produce good results on all the images it's seen before during training, but then fail on new images because it's not learned general characteristics of the objects, just memorized unimportant details of the training images.

This problem is known as overfitting, and to avoid it we keep some of our data out of the training process, so that the model can't memorize them. We then use those images as a check to make sure that overfitting isn't occurring, since if we see good accuracy on them it's a good sign the network isn't overfitting. The usual split is to put 80% of the images into the main training set, keep 10% aside to run as validation frequently during training, and then have a final 10% that are used less often as a testing set to predict the real-world performance of the classifier. These ratios can be controlled using the --testing\_percentage and --validation\_percentage flags. In general you should be able to leave these values at their defaults, since you won't usually find any advantage to training to adjusting them.

Note that the script uses the image filenames (rather than a completely random function) to divide the images among the training, validation, and test sets. This is done to ensure that images don't get moved between training and testing sets on different runs, since that could be a problem if images that had been used for training a model were subsequently used in a validation set.

You might notice that the validation accuracy fluctuates among iterations. Much of this fluctuation arises from the fact that a random subset of the validation set is chosen for each validation accuracy measurement. The fluctuations can be greatly reduced, at the cost of some increase in training time, by choosing --validation\_batch\_size=-1, which uses the entire validation set for each accuracy computation.

Once training is complete, you may find it insightful to examine misclassified images in the test set. This can be done by adding the flag --print\_misclassified\_test\_images. This may help you get a feeling for which types of images were most confusing for the model, and which categories were most difficult to distinguish. For instance, you might discover that some subtype of a particular category, or some unusual photo angle, is particularly difficult to identify, which may encourage you to add more training images of that subtype. Oftentimes, examining misclassified images can also point to errors in the input data set, such as mislabeled, low-quality, or ambiguous images. However, one should generally avoid point-fixing individual errors in the test set, since they are likely to merely reflect more general problems in the (much larger) training set.

# A Guide to TF Layers: Building a Convolutional Neural Network

The TensorFlow [layers module](https://www.tensorflow.org/api_docs/python/tf/layers) provides a high-level API that makes it easy to construct a neural network. It provides methods that facilitate the creation of dense (fully connected) layers and convolutional layers, adding activation functions, and applying dropout regularization. In this tutorial, you'll learn how to use layers to build a convolutional neural network model to recognize the handwritten digits in the MNIST data set.

**The**[**MNIST dataset**](http://yann.lecun.com/exdb/mnist/)**comprises 60,000 training examples and 10,000 test examples of the handwritten digits 0–9, formatted as 28x28-pixel monochrome images.**

## Getting Started

Let's set up the skeleton for our TensorFlow program. Create a file called cnn\_mnist.py, and add the following code:

from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
# Imports  
import numpy as np  
import tensorflow as tf  
  
from tensorflow.contrib import learn  
from tensorflow.contrib.learn.python.learn.estimators import model\_fn as model\_fn\_lib  
  
tf.logging.set\_verbosity(tf.logging.INFO)  
  
# Our application logic will be added here  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  tf.app.run()

As you work through the tutorial, you'll add code to construct, train, and evaluate the convolutional neural network. The complete, final code can be [found here](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/layers/cnn_mnist.py).

## Intro to Convolutional Neural Networks

Convolutional neural networks (CNNs) are the current state-of-the-art model architecture for image classification tasks. CNNs apply a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification. CNNs contains three components:

* **Convolutional layers**, which apply a specified number of convolution filters to the image. For each subregion, the layer performs a set of mathematical operations to produce a single value in the output feature map. Convolutional layers then typically apply a [ReLU activation function](https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) to the output to introduce nonlinearities into the model.
* **Pooling layers**, which [downsample the image data](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer) extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time. A commonly used pooling algorithm is max pooling, which extracts subregions of the feature map (e.g., 2x2-pixel tiles), keeps their maximum value, and discards all other values.
* **Dense (fully connected) layers**, which perform classification on the features extracted by the convolutional layers and downsampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.

Typically, a CNN is composed of a stack of convolutional modules that perform feature extraction. Each module consists of a convolutional layer followed by a pooling layer. The last convolutional module is followed by one or more dense layers that perform classification. The final dense layer in a CNN contains a single node for each target class in the model (all the possible classes the model may predict), with a [softmax](https://en.wikipedia.org/wiki/Softmax_function) activation function to generate a value between 0–1 for each node (the sum of all these softmax values is equal to 1). We can interpret the softmax values for a given image as relative measurements of how likely it is that the image falls into each target class.

**Note:** For a more comprehensive walkthrough of CNN architecture, see Stanford University's [Convolutional Neural Networks for Visual Recognition course materials](http://cs231n.github.io/convolutional-networks/).

## Building the CNN MNIST Classifier

Let's build a model to classify the images in the MNIST dataset using the following CNN architecture:

1. **Convolutional Layer #1**: Applies 32 5x5 filters (extracting 5x5-pixel subregions), with ReLU activation function
2. **Pooling Layer #1**: Performs max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap)
3. **Convolutional Layer #2**: Applies 64 5x5 filters, with ReLU activation function
4. **Pooling Layer #2**: Again, performs max pooling with a 2x2 filter and stride of 2
5. **Dense Layer #1**: 1,024 neurons, with dropout regularization rate of 0.4 (probability of 0.4 that any given element will be dropped during training)
6. **Dense Layer #2 (Logits Layer)**: 10 neurons, one for each digit target class (0–9).

The tf.layers module contains methods to create each of the three layer types above:

* conv2d(). Constructs a two-dimensional convolutional layer. Takes number of filters, filter kernel size, padding, and activation function as arguments.
* max\_pooling2d(). Constructs a two-dimensional pooling layer using the max-pooling algorithm. Takes pooling filter size and stride as arguments.
* dense(). Constructs a dense layer. Takes number of neurons and activation function as arguments.

Each of these methods accepts a tensor as input and returns a transformed tensor as output. This makes it easy to connect one layer to another: just take the output from one layer-creation method and supply it as input to another.

Open cnn\_mnist.py and add the following cnn\_model\_fn function, which conforms to the interface expected by TensorFlow's Estimator API (more on this later in [Create the Estimator](https://www.tensorflow.org/tutorials/layers#create_the_estimator)). cnn\_mnist.py takes MNIST feature data, labels, and [model mode](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/ModeKeys) (TRAIN, EVAL, INFER) as arguments; configures the CNN; and returns predictions, loss, and a training operation:

def cnn\_model\_fn(features, labels, mode):  
  """Model function for CNN."""  
  # Input Layer  
  input\_layer = tf.reshape(features, [-1, 28, 28, 1])  
  
  # Convolutional Layer #1  
  conv1 = tf.layers.conv2d(  
      inputs=input\_layer,  
      filters=32,  
      kernel\_size=[5, 5],  
      padding="same",  
      activation=tf.nn.relu)  
  
  # Pooling Layer #1  
  pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)  
  
  # Convolutional Layer #2 and Pooling Layer #2  
  conv2 = tf.layers.conv2d(  
      inputs=pool1,  
      filters=64,  
      kernel\_size=[5, 5],  
      padding="same",  
      activation=tf.nn.relu)  
  pool2 = tf.layers.max\_pooling2d(inputs=conv2, pool\_size=[2, 2], strides=2)  
  
  # Dense Layer  
  pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])  
  dense = tf.layers.dense(inputs=pool2\_flat, units=1024, activation=tf.nn.relu)  
  dropout = tf.layers.dropout(  
      inputs=dense, rate=0.4, training=mode == learn.ModeKeys.TRAIN)  
  
  # Logits Layer  
  logits = tf.layers.dense(inputs=dropout, units=10)  
  
  loss = None  
  train\_op = None  
  
  # Calculate Loss (for both TRAIN and EVAL modes)  
  if mode != learn.ModeKeys.INFER:  
    onehot\_labels = tf.one\_hot(indices=tf.cast(labels, tf.int32), depth=10)  
    loss = tf.losses.softmax\_cross\_entropy(  
        onehot\_labels=onehot\_labels, logits=logits)  
  
  # Configure the Training Op (for TRAIN mode)  
  if mode == learn.ModeKeys.TRAIN:  
    train\_op = tf.contrib.layers.optimize\_loss(  
        loss=loss,  
        global\_step=tf.contrib.framework.get\_global\_step(),  
        learning\_rate=0.001,  
        optimizer="SGD")  
  
  # Generate Predictions  
  predictions = {  
      "classes": tf.argmax(  
          input=logits, axis=1),  
      "probabilities": tf.nn.softmax(  
          logits, name="softmax\_tensor")  
  }  
  
  # Return a ModelFnOps object  
  return model\_fn\_lib.ModelFnOps(  
      mode=mode, predictions=predictions, loss=loss, train\_op=train\_op)

The following sections (with headings corresponding to each code block above) dive deeper into the tf.layerscode used to create each layer, as well as how to calculate loss, configure the training op, and generate predictions. If you're already experienced with CNNs and [TensorFlow Estimators](https://www.tensorflow.org/extend/estimators), and find the above code intuitive, you may want to skim these sections or just skip ahead to ["Training and Evaluating the CNN MNIST Classifier"](https://www.tensorflow.org/tutorials/layers#training-and-evaluating-the-cnn-mnist-classifier).

### Input Layer

The methods in the layers module for creating convolutional and pooling layers for two-dimensional image data expect input tensors to have a shape of [batch\_size, image\_width, image\_height, channels], defined as follows:

* batch\_size. Size of the subset of examples to use when performing gradient descent during training.
* image\_width. Width of the example images.
* image\_height. Height of the example images.
* channels. Number of color channels in the example images. For color images, the number of channels is 3 (red, green, blue). For monochrome images, there is just 1 channel (black).

Here, our MNIST dataset is composed of monochrome 28x28 pixel images, so the desired shape for our input layer is [batch\_size, 28, 28, 1].

To convert our input feature map (features) to this shape, we can perform the following reshape operation:

input\_layer = tf.reshape(features, [-1, 28, 28, 1])

Note that we've indicated -1 for batch size, which specifies that this dimension should be dynamically computed based on the number of input values in features, holding the size of all other dimensions constant. This allows us to treat batch\_size as a hyperparameter that we can tune. For example, if we feed examples into our model in batches of 5, features will contain 3,920 values (one value for each pixel in each image), and input\_layerwill have a shape of [5, 28, 28, 1]. Similarly, if we feed examples in batches of 100, features will contain 78,400 values, and input\_layer will have a shape of [100, 28, 28, 1].

### Convolutional Layer #1

In our first convolutional layer, we want to apply 32 5x5 filters to the input layer, with a ReLU activation function. We can use the conv2d() method in the layers module to create this layer as follows:

conv1 = tf.layers.conv2d(  
    inputs=input\_layer,  
    filters=32,  
    kernel\_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)

The inputs argument specifies our input tensor, which must have the shape [batch\_size, image\_width,image\_height, channels]. Here, we're connecting our first convolutional layer to input\_layer, which has the shape [batch\_size, 28, 28, 1].

**Note:** **conv2d()** will instead accept a shape of **[channels, batch\_size, image\_width, image\_height]** when passed the argument **data\_format=channels\_first**.

The filters argument specifies the number of filters to apply (here, 32), and kernel\_size specifies the dimensions of the filters as [width, height] (here, [5, 5]).

**TIP:** If filter width and height have the same value, you can instead specify a single integer for kernel\_size—e.g., kernel\_size=5.

The padding argument specifies one of two enumerated values (case-insensitive): valid (default value) or same. To specify that the output tensor should have the same width and height values as the input tensor, we set padding=same here, which instructs TensorFlow to add 0 values to the edges of the output tensor to preserve width and height of 28. (Without padding, a 5x5 convolution over a 28x28 tensor will produce a 24x24 tensor, as there are 24x24 locations to extract a 5x5 tile from a 28x28 grid.)

The activation argument specifies the activation function to apply to the output of the convolution. Here, we specify ReLU activation with [tf.nn.relu](https://www.tensorflow.org/api_docs/python/tf/nn/relu).

Our output tensor produced by conv2d() has a shape of [batch\_size, 28, 28, 32]: the same width and height dimensions as the input, but now with 32 channels holding the output from each of the filters.

### Pooling Layer #1

Next, we connect our first pooling layer to the convolutional layer we just created. We can use the max\_pooling2d() method in layers to construct a layer that performs max pooling with a 2x2 filter and stride of 2:

pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2, 2], strides=2)

Again, inputs specifies the input tensor, with a shape of [batch\_size, image\_width, image\_height,channels]. Here, our input tensor is conv1, the output from the first convolutional layer, which has a shape of [batch\_size, 28, 28, 32].

**Note:** As with **conv2d()**, **max\_pooling2d()** will instead accept a shape of **[channels, batch\_size,image\_width, image\_height]** when passed the argument **data\_format=channels\_first**.

The pool\_size argument specifies the size of the max pooling filter as [width, height] (here, [2, 2]). If both dimensions have the same value, you can instead specify a single integer (e.g., pool\_size=2).

The strides argument specifies the size of the stride. Here, we set a stride of 2, which indicates that the subregions extracted by the filter should be separated by 2 pixels in both the width and height dimensions (for a 2x2 filter, this means that none of the regions extracted will overlap). If you want to set different stride values for width and height, you can instead specify a tuple or list (e.g., stride=[3, 6]).

Our output tensor produced by max\_pooling2d() (pool1) has a shape of [batch\_size, 14, 14, 32]: the 2x2 filter reduces width and height by 50% each.

### Convolutional Layer #2 and Pooling Layer #2

We can connect a second convolutional and pooling layer to our CNN using conv2d() and max\_pooling2d()as before. For convolutional layer #2, we configure 64 5x5 filters with ReLU activation, and for pooling layer #2, we use the same specs as pooling layer #1 (a 2x2 max pooling filter with stride of 2):

conv2 = tf.layers.conv2d(  
    inputs=pool1,  
    filters=64,  
    kernel\_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)  
  
pool2 = tf.layers.max\_pooling2d(inputs=conv2, pool\_size=[2, 2], strides=2)

Note that convolutional layer #2 takes the output tensor of our first pooling layer (pool1) as input, and produces the tensor h\_conv2 as output. conv2 has a shape of [batch\_size, 14, 14, 64], the same width and height as pool1 (due to padding="same"), and 64 channels for the 64 filters applied.

Pooling layer #2 takes conv2 as input, producing pool2 as output. pool2 has shape [batch\_size, 7, 7, 64] (50% reduction of width and height from conv2).

### Dense Layer

Next, we want to add a dense layer (with 1,024 neurons and ReLU activation) to our CNN to perform classification on the features extracted by the convolution/pooling layers. Before we connect the layer, however, we'll flatten our feature map (pool2) to shape [batch\_size, features], so that our tensor has only two dimensions:

pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])

In the reshape() operation above, the -1 signifies that the batch\_size dimension will be dynamically calculated based on the number of examples in our input data. Each example has 7 (pool2 width) \* 7 (pool2height) \* 64 (pool2 channels) features, so we want the features dimension to have a value of 7 \* 7 \* 64 (3136 in total). The output tensor, pool2\_flat, has shape [batch\_size, 3136].

Now, we can use the dense() method in layers to connect our dense layer as follows:

dense = tf.layers.dense(inputs=pool2\_flat, units=1024, activation=tf.nn.relu)

The inputs argument specifies the input tensor: our flattened feature map, pool2\_flat. The units argument specifies the number of neurons in the dense layer (1,024). The activation argument takes the activation function; again, we'll use tf.nn.relu to add ReLU activation.

To help improve the results of our model, we also apply dropout regularization to our dense layer, using the dropout method in layers:

dropout = tf.layers.dropout(  
    inputs=dense, rate=0.4, training=mode == learn.ModeKeys.TRAIN)

Again, inputs specifies the input tensor, which is the output tensor from our dense layer (dense).

The rate argument specifies the dropout rate; here, we use 0.4, which means 40% of the elements will be randomly dropped out during training.

The training argument takes a boolean specifying whether or not the model is currently being run in training mode; dropout will only be performed if training is True. Here, we check if the mode passed to our model function cnn\_model\_fn is TRAIN mode.

Our output tensor dropout has shape [batch\_size, 1024].

### Logits Layer

The final layer in our neural network is the logits layer, which will return the raw values for our predictions. We create a dense layer with 10 neurons (one for each target class 0–9), with linear activation (the default):

logits = tf.layers.dense(inputs=dropout, units=10)

Our final output tensor of the CNN, logits, has shape [batch\_size, 10].

### Calculate Loss

For both training and evaluation, we need to define a [loss function](https://en.wikipedia.org/wiki/Loss_function) that measures how closely the model's predictions match the target classes. For multiclass classification problems like MNIST, [cross entropy](https://en.wikipedia.org/wiki/Cross_entropy) is typically used as the loss metric. The following code calculates cross entropy when the model runs in either TRAIN or EVAL mode:

loss = None  
train\_op = None  
  
# Calculate loss for both TRAIN and EVAL modes  
if mode != learn.ModeKeys.INFER:  
  onehot\_labels = tf.one\_hot(indices=tf.cast(labels, tf.int32), depth=10)  
  loss = tf.losses.softmax\_cross\_entropy(  
      onehot\_labels=onehot\_labels, logits=logits)

Let's take a closer look at what's happening above.

Our labels tensor contains a list of predictions for our examples, e.g. [1, 9, ...]. In order to calculate cross-entropy, first we need to convert labels to the corresponding [one-hot encoding](https://www.quora.com/What-is-one-hot-encoding-and-when-is-it-used-in-data-science):

[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
 ...]

We use the [tf.one\_hot](https://www.tensorflow.org/api_docs/python/tf/one_hot) function to perform this conversion. tf.one\_hot() has two required arguments:

* indices. The locations in the one-hot tensor that will have "on values"—i.e., the locations of 1 values in the tensor shown above.
* depth. The depth of the one-hot tensor—i.e., the number of target classes. Here, the depth is 10.

The following code creates the one-hot tensor for our labels, onehot\_labels:

onehot\_labels = tf.one\_hot(indices=tf.cast(labels, tf.int32), depth=10)

Because labels contains a series of values from 0–9, indices is just our labels tensor, with values cast to integers. The depth is 10 because we have 10 possible target classes, one for each digit.

Next, we compute cross-entropy of onehot\_labels and the softmax of the predictions from our logits layer. tf.losses.softmax\_cross\_entropy() takes onehot\_labels and logits as arguments, performs softmax activation on logits, calculates cross-entropy, and returns our loss as a scalar Tensor:

loss = tf.losses.softmax\_cross\_entropy(  
        onehot\_labels=onehot\_labels, logits=logits)

### Configure the Training Op

In the previous section, we defined loss for our CNN as the softmax cross-entropy of the logits layer and our labels. Let's configure our model to optimize this loss value during training, using the[tf.contrib.layers.optimize\_loss](https://www.tensorflow.org/api_docs/python/tf/contrib/layers/optimize_loss) method in tf.contrib.layers. We'll use a learning rate of 0.001 and[stochastic gradient descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) as the optimization algorithm:

# Configure the Training Op (for TRAIN mode)  
if mode == learn.ModeKeys.TRAIN:  
    train\_op = tf.contrib.layers.optimize\_loss(  
        loss=loss,  
        global\_step=tf.contrib.framework.get\_global\_step(),  
        learning\_rate=0.001,  
        optimizer="SGD")

**Note:** For a more in-depth look at configuring training ops for Estimator model functions, see ["Defining the training op for the model"](https://www.tensorflow.org/extend/estimators#defining-the-training-op-for-the-model) in the ["Creating Estimations in tf.contrib.learn"](https://www.tensorflow.org/extend/estimators) tutorial.

### Generate Predictions

The logits layer of our model returns our predictions as raw values in a [batch\_size, 10]-dimensional tensor. Let's convert these raw values into two different formats that our model function can return:

* The **predicted class** for each example: a digit from 0–9.
* The **probabilities** for each possible target class for each example: the probability that the example is a 0, is a 1, is a 2, etc.

For a given example, our predicted class is the element in the corresponding row of the logits tensor with the highest raw value. We can find the index of this element using the [tf.argmax](https://www.tensorflow.org/api_docs/python/tf/argmax) function:

tf.argmax(input=logits, axis=1)

The input argument specifies the tensor from which to extract maximum values—here logits. The axisargument specifies the axis of the input tensor along which to find the greatest value. Here, we want to find the largest value along the dimension with index of 1, which corresponds to our predictions (recall that our logits tensor has shape [batch\_size, 10]).

We can derive probabilities from our logits layer by applying softmax activation using [tf.nn.softmax](https://www.tensorflow.org/api_docs/python/tf/nn/softmax):

tf.nn.softmax(logits, name="softmax\_tensor")

**Note:** We use the **name** argument to explicitly name this operation **softmax\_tensor**, so we can reference it later. (We'll set up logging for the softmax values in ["Set Up a Logging Hook"](https://www.tensorflow.org/tutorials/layers#set_up_a_logging_hook).

We compile our predictions in a dict as follows:

predictions = {  
    "classes": tf.argmax(  
        input=logits, axis=1),  
    "probabilities": tf.nn.softmax(  
        logits, name="softmax\_tensor")  
}

Finally, now that we've got our predictions, loss, and train\_op, we can return them, along with our modeargument, in a [tf.contrib.learn.ModelFnOps](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/ModelFnOps) object:

# Return a ModelFnOps object  
return model\_fn\_lib.ModelFnOps(  
    mode=mode, predictions=predictions, loss=loss, train\_op=train\_op)

## Training and Evaluating the CNN MNIST Classifier

We've coded our MNIST CNN model function; now we're ready to train and evaluate it.

### Load Training and Test Data

First, let's load our training and test data. Add a main() function to cnn\_mnist.py with the following code:

def main(unused\_argv):  
  # Load training and eval data  
  mnist = learn.datasets.load\_dataset("mnist")  
  train\_data = mnist.train.images # Returns np.array  
  train\_labels = np.asarray(mnist.train.labels, dtype=np.int32)  
  eval\_data = mnist.test.images # Returns np.array  
  eval\_labels = np.asarray(mnist.test.labels, dtype=np.int32)

We store the training feature data (the raw pixel values for 55,000 images of hand-drawn digits) and training labels (the corresponding value from 0–9 for each image) as [numpy arrays](https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html) in train\_data and train\_labels, respectively. Similarly, we store the evalulation feature data (10,000 images) and evaluation labels in eval\_dataand eval\_labels, respectively.

### Create the Estimator

Next, let's create an Estimator (a TensorFlow class for performing high-level model training, evaluation, and inference) for our model. Add the following code to main():

# Create the Estimator  
mnist\_classifier = learn.Estimator(  
      model\_fn=cnn\_model\_fn, model\_dir="/tmp/mnist\_convnet\_model")

The model\_fn argument specifies the model function to use for training, evaluation, and inference; we pass it the cnn\_model\_fn we created in ["Building the CNN MNIST Classifier."](https://www.tensorflow.org/tutorials/layers#building_the_cnn_mnist_classifier) The model\_dir argument specifies the directory where model data (checkpoints) will be saved (here, we specify the temp directory /tmp/mnist\_convnet\_model, but feel free to change to another directory of your choice).

**Note:** For an in-depth walkthrough of the TensorFlow **Estimator** API, see the tutorial ["Creating Estimators in tf.contrib.learn."](https://www.tensorflow.org/extend/estimators)

### Set Up a Logging Hook

Since CNNs can take a while to train, let's set up some logging so we can track progress during training. We can use TensorFlow's [tf.train.SessionRunHook](https://www.tensorflow.org/api_docs/python/tf/train/SessionRunHook) to create a [tf.train.LoggingTensorHook](https://www.tensorflow.org/api_docs/python/tf/train/LoggingTensorHook) that will log the probability values from the softmax layer of our CNN. Add the following to main():

# Set up logging for predictions  
  tensors\_to\_log = {"probabilities": "softmax\_tensor"}  
  logging\_hook = tf.train.LoggingTensorHook(  
      tensors=tensors\_to\_log, every\_n\_iter=50)

We store a dict of the tensors we want to log in tensors\_to\_log. Each key is a label of our choice that will be printed in the log output, and the corresponding label is the name of a Tensor in the TensorFlow graph. Here, ourprobabilities can be found in softmax\_tensor, the name we gave our softmax operation earlier when we generated the probabilities in cnn\_model\_fn.

**Note:** If you don't explicitly assign a name to an operation via the **name** argument, TensorFlow will assign a default name. A couple easy ways to discover the names applied to operations are to visualize your graph on[TensorBoard](https://www.tensorflow.org/get_started/graph_viz)) or to enable the [TensorFlow Debugger (tfdbg)](https://www.tensorflow.org/programmers_guide/debugger).

Next, we create the LoggingTensorHook, passing tensors\_to\_log to the tensors argument. We set every\_n\_iter=50, which specifies that probabilities should be logged after every 50 steps of training.

### Train the Model

Now we're ready to train our model, which we can do by calling fit() on mnist\_classifier. Add the following to main():

# Train the model  
mnist\_classifier.fit(  
    x=train\_data,  
    y=train\_labels,  
    batch\_size=100,  
    steps=20000,  
    monitors=[logging\_hook])

In the fit call, we pass the training feature data and labels to x and y, respectively. We set a batch\_size of 100 (which means that the model will train on minibatches of 100 examples at each step), and steps of 20000(which means the model will train for 20,000 steps total). We pass our logging\_hook to the monitorsargument, so that it will be triggered during training.

### Evaluate the Model

Once training is complete, we want to evaluate our model to determine its accuracy on the MNIST test set. To set up the accuracy metric for our model, we need to create a metrics dict with a [tf.contrib.learn.MetricSpec](https://www.tensorflow.org/api_docs/python/tf/contrib/learn/MetricSpec)that calculates accuracy. Add the following to main():

# Configure the accuracy metric for evaluation  
metrics = {  
    "accuracy":  
        learn.MetricSpec(  
            metric\_fn=tf.metrics.accuracy, prediction\_key="classes"),  
}

We create our MetricSpecs with the following two arguments:

* metric\_fn. The function that calculates and returns the value of our metric. Here, we can use the predefined accuracy function in the [tf.metrics](https://www.tensorflow.org/api_docs/python/tf/metrics) module.
* prediction\_key. The key of the tensor that contains the predictions returned by the model function. Here, because we're building a classification model, the prediction key is "classes", which we specified back in ["Generate Predictions."](https://www.tensorflow.org/tutorials/layers#generate_predictions)

Now that we've set up our metrics dict, we can evaluate the model. Add the following code, which performs evaluation and prints the results:

# Evaluate the model and print results  
eval\_results = mnist\_classifier.evaluate(  
    x=eval\_data, y=eval\_labels, metrics=metrics)  
print(eval\_results)

We pass our evaluation feature data and labels to evaluate() in the x and y arguments, respectively. The metrics argument takes the metrics dict we just defined.

### Run the Model

We've coded the CNN model function, Estimator, and the training/evaluation logic; now let's see the results. Run cnn\_mnist.py.

**Note:** Training CNNs is quite computationally intensive. Estimated completion time of **cnn\_mnist.py** will vary depending on your processor, but will likely be upwards of 1 hour on CPU. To train more quickly, you can decrease the number of **steps** passed to **fit()**, but note that this will affect accuracy.

As the model trains, you'll see log output like the following:

INFO:tensorflow:loss = 2.36026, step = 1  
INFO:tensorflow:probabilities = [[ 0.07722801  0.08618255  0.09256398, ...]]  
...  
INFO:tensorflow:loss = 2.13119, step = 101  
INFO:tensorflow:global\_step/sec: 5.44132  
...  
INFO:tensorflow:Loss for final step: 0.553216.  
  
INFO:tensorflow:Restored model from /tmp/mnist\_convnet\_model  
INFO:tensorflow:Eval steps [0,inf) for training step 20000.  
INFO:tensorflow:Input iterator is exhausted.  
INFO:tensorflow:Saving evaluation summary for step 20000: accuracy = 0.9733, loss = 0.0902271  
{'loss': 0.090227105, 'global\_step': 20000, 'accuracy': 0.97329998}

Here, we've achieved an accuracy of 97.3% on our test data set.

## Additional Resources

To learn more about TensorFlow Estimators and CNNs in TensorFlow, see the following resources:

* [Creating Estimators in tf.contrib.learn](https://www.tensorflow.org/extend/estimators). An introduction to the TensorFlow Estimator API, which walks through configuring an Estimator, writing a model function, calculating loss, and defining a training op.
* [Deep MNIST for Experts: Building a Multilayer CNN](https://www.tensorflow.org/get_started/mnist/pros#build_a_multilayer_convolutional_network). Walks through how to build a MNIST CNN classification model without layers using lower-level TensorFlow operations.

# Convolutional Neural Networks

**NOTE:** This tutorial is intended for advanced users of TensorFlow and assumes expertise and experience in machine learning.

## Overview

CIFAR-10 classification is a common benchmark problem in machine learning. The problem is to classify RGB 32x32 pixel images across 10 categories:

airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

For more details refer to the [CIFAR-10 page](http://www.cs.toronto.edu/~kriz/cifar.html) and a [Tech Report](http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf) by Alex Krizhevsky.

### Goals

The goal of this tutorial is to build a relatively small [convolutional neural network](https://en.wikipedia.org/wiki/Convolutional_neural_network) (CNN) for recognizing images. In the process, this tutorial:

1. Highlights a canonical organization for network architecture, training and evaluation.
2. Provides a template for constructing larger and more sophisticated models.

The reason CIFAR-10 was selected was that it is complex enough to exercise much of TensorFlow's ability to scale to large models. At the same time, the model is small enough to train fast, which is ideal for trying out new ideas and experimenting with new techniques.

### Highlights of the Tutorial

The CIFAR-10 tutorial demonstrates several important constructs for designing larger and more sophisticated models in TensorFlow:

* Core mathematical components including [convolution](https://www.tensorflow.org/api_docs/python/tf/nn/conv2d) ([wiki](https://en.wikipedia.org/wiki/Convolution)), [rectified linear activations](https://www.tensorflow.org/api_docs/python/tf/nn/relu) ([wiki](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))), [max pooling](https://www.tensorflow.org/api_docs/python/tf/nn/max_pool)([wiki](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer)) and [local response normalization](https://www.tensorflow.org/api_docs/python/tf/nn/local_response_normalization) (Chapter 3.3 in [AlexNet paper](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf)).
* [Visualization](https://www.tensorflow.org/get_started/summaries_and_tensorboard) of network activities during training, including input images, losses and distributions of activations and gradients.
* Routines for calculating the [moving average](https://www.tensorflow.org/api_docs/python/tf/train/ExponentialMovingAverage) of learned parameters and using these averages during evaluation to boost predictive performance.
* Implementation of a [learning rate schedule](https://www.tensorflow.org/api_docs/python/tf/train/exponential_decay) that systematically decrements over time.
* Prefetching [queues](https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch) for input data to isolate the model from disk latency and expensive image pre-processing.

We also provide a [multi-GPU version](https://www.tensorflow.org/tutorials/deep_cnn#training_a_model_using_multiple_gpu_cards) of the model which demonstrates:

* Configuring a model to train across multiple GPU cards in parallel.
* Sharing and updating variables among multiple GPUs.

We hope that this tutorial provides a launch point for building larger CNNs for vision tasks on TensorFlow.

### Model Architecture

The model in this CIFAR-10 tutorial is a multi-layer architecture consisting of alternating convolutions and nonlinearities. These layers are followed by fully connected layers leading into a softmax classifier. The model follows the architecture described by [Alex Krizhevsky](https://code.google.com/p/cuda-convnet/), with a few differences in the top few layers.

This model achieves a peak performance of about 86% accuracy within a few hours of training time on a GPU. Please see [below](https://www.tensorflow.org/tutorials/deep_cnn#evaluating_a_model) and the code for details. It consists of 1,068,298 learnable parameters and requires about 19.5M multiply-add operations to compute inference on a single image.

## Code Organization

The code for this tutorial resides in [models/tutorials/image/cifar10/](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/).

| File | Purpose |
| --- | --- |
| [cifar10\_input.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10_input.py) | Reads the native CIFAR-10 binary file format. |
| [cifar10.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10.py) | Builds the CIFAR-10 model. |
| [cifar10\_train.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10_train.py) | Trains a CIFAR-10 model on a CPU or GPU. |
| [cifar10\_multi\_gpu\_train.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10_multi_gpu_train.py) | Trains a CIFAR-10 model on multiple GPUs. |
| [cifar10\_eval.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10_eval.py) | Evaluates the predictive performance of a CIFAR-10 model. |

## CIFAR-10 Model

The CIFAR-10 network is largely contained in [cifar10.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/image/cifar10/cifar10.py). The complete training graph contains roughly 765 operations. We find that we can make the code most reusable by constructing the graph with the following modules:

1. [**Model inputs:**](https://www.tensorflow.org/tutorials/deep_cnn#model_inputs) inputs() and distorted\_inputs() add operations that read and preprocess CIFAR images for evaluation and training, respectively.
2. [**Model prediction:**](https://www.tensorflow.org/tutorials/deep_cnn#model_prediction) inference() adds operations that perform inference, i.e. classification, on supplied images.
3. [**Model training:**](https://www.tensorflow.org/tutorials/deep_cnn#model_training) loss() and train() add operations that compute the loss, gradients, variable updates and visualization summaries.

### Model Inputs

The input part of the model is built by the functions inputs() and distorted\_inputs() which read images from the CIFAR-10 binary data files. These files contain fixed byte length records, so we use[tf.FixedLengthRecordReader](https://www.tensorflow.org/api_docs/python/tf/FixedLengthRecordReader). See [Reading Data](https://www.tensorflow.org/programmers_guide/reading_data#reading_from_files) to learn more about how the Reader class works.

The images are processed as follows:

* They are cropped to 24 x 24 pixels, centrally for evaluation or [randomly](https://www.tensorflow.org/api_docs/python/tf/random_crop) for training.
* They are [approximately whitened](https://www.tensorflow.org/api_docs/python/tf/image/per_image_standardization) to make the model insensitive to dynamic range.

For training, we additionally apply a series of random distortions to artificially increase the data set size:

* [Randomly flip](https://www.tensorflow.org/api_docs/python/tf/image/random_flip_left_right) the image from left to right.
* Randomly distort the [image brightness](https://www.tensorflow.org/api_docs/python/tf/image/random_brightness).
* Randomly distort the [image contrast](https://www.tensorflow.org/api_docs/python/tf/image/random_contrast).

Please see the [Images](https://www.tensorflow.org/api_guides/python/image) page for the list of available distortions. We also attach an [tf.summary.image](https://www.tensorflow.org/api_docs/python/tf/summary/image) to the images so that we may visualize them in [TensorBoard](https://www.tensorflow.org/get_started/summaries_and_tensorboard). This is a good practice to verify that inputs are built correctly.

Reading images from disk and distorting them can use a non-trivial amount of processing time. To prevent these operations from slowing down training, we run them inside 16 separate threads which continuously fill a TensorFlow [queue](https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch).

### Model Prediction

The prediction part of the model is constructed by the inference() function which adds operations to compute the logits of the predictions. That part of the model is organized as follows:

| Layer Name | Description |
| --- | --- |
| conv1 | [convolution](https://www.tensorflow.org/api_docs/python/tf/nn/conv2d) and [rectified linear](https://www.tensorflow.org/api_docs/python/tf/nn/relu) activation. |
| pool1 | [max pooling](https://www.tensorflow.org/api_docs/python/tf/nn/max_pool). |
| norm1 | [local response normalization](https://www.tensorflow.org/api_docs/python/tf/nn/local_response_normalization). |
| conv2 | [convolution](https://www.tensorflow.org/api_docs/python/tf/nn/conv2d) and [rectified linear](https://www.tensorflow.org/api_docs/python/tf/nn/relu) activation. |
| norm2 | [local response normalization](https://www.tensorflow.org/api_docs/python/tf/nn/local_response_normalization). |
| pool2 | [max pooling](https://www.tensorflow.org/api_docs/python/tf/nn/max_pool). |
| local3 | [fully connected layer with rectified linear activation](https://www.tensorflow.org/api_guides/python/nn). |
| local4 | [fully connected layer with rectified linear activation](https://www.tensorflow.org/api_guides/python/nn). |
| softmax\_linear | linear transformation to produce logits. |

Here is a graph generated from TensorBoard describing the inference operation:

**EXERCISE**: The output of inference are un-normalized logits. Try editing the network architecture to return normalized predictions using [tf.nn.softmax](https://www.tensorflow.org/api_docs/python/tf/nn/softmax).

The inputs() and inference() functions provide all the components necessary to perform evaluation on a model. We now shift our focus towards building operations for training a model.

**EXERCISE:** The model architecture in inference() differs slightly from the CIFAR-10 model specified in [cuda-convnet](https://code.google.com/p/cuda-convnet/). In particular, the top layers of Alex's original model are locally connected and not fully connected. Try editing the architecture to exactly reproduce the locally connected architecture in the top layer.

### Model Training

The usual method for training a network to perform N-way classification is [multinomial logistic regression](https://en.wikipedia.org/wiki/Multinomial_logistic_regression), aka. softmax regression. Softmax regression applies a [softmax](https://www.tensorflow.org/api_docs/python/tf/nn/softmax) nonlinearity to the output of the network and calculates the [cross-entropy](https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits) between the normalized predictions and a [1-hot encoding](https://www.tensorflow.org/api_docs/python/tf/sparse_to_dense) of the label. For regularization, we also apply the usual [weight decay](https://www.tensorflow.org/api_docs/python/tf/nn/l2_loss) losses to all learned variables. The objective function for the model is the sum of the cross entropy loss and all these weight decay terms, as returned by the loss() function.

We visualize it in TensorBoard with a [tf.summary.scalar](https://www.tensorflow.org/api_docs/python/tf/summary/scalar):

We train the model using standard [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent) algorithm (see [Training](https://www.tensorflow.org/api_guides/python/train) for other methods) with a learning rate that [exponentially decays](https://www.tensorflow.org/api_docs/python/tf/train/exponential_decay) over time.

The train() function adds the operations needed to minimize the objective by calculating the gradient and updating the learned variables (see [tf.train.GradientDescentOptimizer](https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer) for details). It returns an operation that executes all the calculations needed to train and update the model for one batch of images.

## Launching and Training the Model

We have built the model, let's now launch it and run the training operation with the script cifar10\_train.py.

python cifar10\_train.py

**NOTE:** The first time you run any target in the CIFAR-10 tutorial, the CIFAR-10 dataset is automatically downloaded. The data set is ~160MB so you may want to grab a quick cup of coffee for your first run.

You should see the output:

Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.  
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)  
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)  
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)  
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)  
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)  
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)  
...

The script reports the total loss every 10 steps as well as the speed at which the last batch of data was processed. A few comments:

* The first batch of data can be inordinately slow (e.g. several minutes) as the preprocessing threads fill up the shuffling queue with 20,000 processed CIFAR images.
* The reported loss is the average loss of the most recent batch. Remember that this loss is the sum of the cross entropy and all weight decay terms.
* Keep an eye on the processing speed of a batch. The numbers shown above were obtained on a Tesla K40c. If you are running on a CPU, expect slower performance.

**EXERCISE:** When experimenting, it is sometimes annoying that the first training step can take so long. Try decreasing the number of images that initially fill up the queue. Search for min\_fraction\_of\_examples\_in\_queue in cifar10\_input.py.

cifar10\_train.py periodically [saves](https://www.tensorflow.org/api_docs/python/tf/train/Saver) all model parameters in [checkpoint files](https://www.tensorflow.org/programmers_guide/variables#saving_and_restoring) but it does not evaluate the model. The checkpoint file will be used by cifar10\_eval.py to measure the predictive performance (see [Evaluating a Model](https://www.tensorflow.org/tutorials/deep_cnn#evaluating_a_model) below).

If you followed the previous steps, then you have now started training a CIFAR-10 model. [Congratulations!](https://www.youtube.com/watch?v=9bZkp7q19f0)

The terminal text returned from cifar10\_train.py provides minimal insight into how the model is training. We want more insight into the model during training:

* Is the loss really decreasing or is that just noise?
* Is the model being provided appropriate images?
* Are the gradients, activations and weights reasonable?
* What is the learning rate currently at?

[TensorBoard](https://www.tensorflow.org/get_started/summaries_and_tensorboard) provides this functionality, displaying data exported periodically from cifar10\_train.py via a[tf.summary.FileWriter](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter).

For instance, we can watch how the distribution of activations and degree of sparsity in local3 features evolve during training:

Individual loss functions, as well as the total loss, are particularly interesting to track over time. However, the loss exhibits a considerable amount of noise due to the small batch size employed by training. In practice we find it extremely useful to visualize their moving averages in addition to their raw values. See how the scripts use[tf.train.ExponentialMovingAverage](https://www.tensorflow.org/api_docs/python/tf/train/ExponentialMovingAverage) for this purpose.

## Evaluating a Model

Let us now evaluate how well the trained model performs on a hold-out data set. The model is evaluated by the script cifar10\_eval.py. It constructs the model with the inference() function and uses all 10,000 images in the evaluation set of CIFAR-10. It calculates the precision at 1: how often the top prediction matches the true label of the image.

To monitor how the model improves during training, the evaluation script runs periodically on the latest checkpoint files created by the cifar10\_train.py.

python cifar10\_eval.py

Be careful not to run the evaluation and training binary on the same GPU or else you might run out of memory. Consider running the evaluation on a separate GPU if available or suspending the training binary while running the evaluation on the same GPU.

You should see the output:

2015-11-06 08:30:44.391206: precision @ 1 = 0.860  
...

The script merely returns the precision @ 1 periodically -- in this case it returned 86% accuracy. cifar10\_eval.py also exports summaries that may be visualized in TensorBoard. These summaries provide additional insight into the model during evaluation.

The training script calculates the [moving average](https://www.tensorflow.org/api_docs/python/tf/train/ExponentialMovingAverage) version of all learned variables. The evaluation script substitutes all learned model parameters with the moving average version. This substitution boosts model performance at evaluation time.

**EXERCISE:** Employing averaged parameters may boost predictive performance by about 3% as measured by precision @ 1. Edit cifar10\_eval.py to not employ the averaged parameters for the model and verify that the predictive performance drops.

## Training a Model Using Multiple GPU Cards

Modern workstations may contain multiple GPUs for scientific computation. TensorFlow can leverage this environment to run the training operation concurrently across multiple cards.

Training a model in a parallel, distributed fashion requires coordinating training processes. For what follows we term model replica to be one copy of a model training on a subset of data.

Naively employing asynchronous updates of model parameters leads to sub-optimal training performance because an individual model replica might be trained on a stale copy of the model parameters. Conversely, employing fully synchronous updates will be as slow as the slowest model replica.

In a workstation with multiple GPU cards, each GPU will have similar speed and contain enough memory to run an entire CIFAR-10 model. Thus, we opt to design our training system in the following manner:

* Place an individual model replica on each GPU.
* Update model parameters synchronously by waiting for all GPUs to finish processing a batch of data.

Here is a diagram of this model:

Note that each GPU computes inference as well as the gradients for a unique batch of data. This setup effectively permits dividing up a larger batch of data across the GPUs.

This setup requires that all GPUs share the model parameters. A well-known fact is that transferring data to and from GPUs is quite slow. For this reason, we decide to store and update all model parameters on the CPU (see green box). A fresh set of model parameters is transferred to the GPU when a new batch of data is processed by all GPUs.

The GPUs are synchronized in operation. All gradients are accumulated from the GPUs and averaged (see green box). The model parameters are updated with the gradients averaged across all model replicas.

### Placing Variables and Operations on Devices

Placing operations and variables on devices requires some special abstractions.

The first abstraction we require is a function for computing inference and gradients for a single model replica. In the code we term this abstraction a "tower". We must set two attributes for each tower:

* A unique name for all operations within a tower. [tf.name\_scope](https://www.tensorflow.org/api_docs/python/tf/name_scope) provides this unique name by prepending a scope. For instance, all operations in the first tower are prepended with tower\_0, e.g. tower\_0/conv1/Conv2D.
* A preferred hardware device to run the operation within a tower. [tf.device](https://www.tensorflow.org/api_docs/python/tf/device) specifies this. For instance, all operations in the first tower reside within device('/gpu:0') scope indicating that they should be run on the first GPU.

All variables are pinned to the CPU and accessed via [tf.get\_variable](https://www.tensorflow.org/api_docs/python/tf/get_variable) in order to share them in a multi-GPU version. See how-to on [Sharing Variables](https://www.tensorflow.org/programmers_guide/variable_scope).

### Launching and Training the Model on Multiple GPU cards

If you have several GPU cards installed on your machine you can use them to train the model faster with the cifar10\_multi\_gpu\_train.py script. This version of the training script parallelizes the model across multiple GPU cards.

python cifar10\_multi\_gpu\_train.py --num\_gpus=2

Note that the number of GPU cards used defaults to 1. Additionally, if only 1 GPU is available on your machine, all computations will be placed on it, even if you ask for more.

**EXERCISE:** The default settings for cifar10\_train.py is to run on a batch size of 128. Try running cifar10\_multi\_gpu\_train.py on 2 GPUs with a batch size of 64 and compare the training speed.

## Next Steps

[Congratulations!](https://www.youtube.com/watch?v=9bZkp7q19f0) You have completed the CIFAR-10 tutorial.

If you are now interested in developing and training your own image classification system, we recommend forking this tutorial and replacing components to address your image classification problem.

**EXERCISE:** Download the [Street View House Numbers (SVHN)](http://ufldl.stanford.edu/housenumbers/) data set. Fork the CIFAR-10 tutorial and swap in the SVHN as the input data. Try adapting the network architecture to improve predictive performance.

# Vector Representations of Words

In this tutorial we look at the word2vec model by [Mikolov et al.](http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) This model is used for learning vector representations of words, called "word embeddings".

## Highlights

This tutorial is meant to highlight the interesting, substantive parts of building a word2vec model in TensorFlow.

* We start by giving the motivation for why we would want to represent words as vectors.
* We look at the intuition behind the model and how it is trained (with a splash of math for good measure).
* We also show a simple implementation of the model in TensorFlow.
* Finally, we look at ways to make the naive version scale better.

We walk through the code later during the tutorial, but if you'd prefer to dive straight in, feel free to look at the minimalistic implementation in [tensorflow/examples/tutorials/word2vec/word2vec\_basic.py](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/word2vec/word2vec_basic.py) This basic example contains the code needed to download some data, train on it a bit and visualize the result. Once you get comfortable with reading and running the basic version, you can graduate to[models/tutorials/embedding/word2vec.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/embedding/word2vec.py) which is a more serious implementation that showcases some more advanced TensorFlow principles about how to efficiently use threads to move data into a text model, how to checkpoint during training, etc.

But first, let's look at why we would want to learn word embeddings in the first place. Feel free to skip this section if you're an Embedding Pro and you'd just like to get your hands dirty with the details.

## Motivation: Why Learn Word Embeddings?

Image and audio processing systems work with rich, high-dimensional datasets encoded as vectors of the individual raw pixel-intensities for image data, or e.g. power spectral density coefficients for audio data. For tasks like object or speech recognition we know that all the information required to successfully perform the task is encoded in the data (because humans can perform these tasks from the raw data). However, natural language processing systems traditionally treat words as discrete atomic symbols, and therefore 'cat' may be represented as Id537 and 'dog' as Id143. These encodings are arbitrary, and provide no useful information to the system regarding the relationships that may exist between the individual symbols. This means that the model can leverage very little of what it has learned about 'cats' when it is processing data about 'dogs' (such that they are both animals, four-legged, pets, etc.). Representing words as unique, discrete ids furthermore leads to data sparsity, and usually means that we may need more data in order to successfully train statistical models. Using vector representations can overcome some of these obstacles.

[Vector space models](https://en.wikipedia.org/wiki/Vector_space_model) (VSMs) represent (embed) words in a continuous vector space where semantically similar words are mapped to nearby points ('are embedded nearby each other'). VSMs have a long, rich history in NLP, but all methods depend in some way or another on the [Distributional Hypothesis](https://en.wikipedia.org/wiki/Distributional_semantics#Distributional_Hypothesis), which states that words that appear in the same contexts share semantic meaning. The different approaches that leverage this principle can be divided into two categories: count-based methods (e.g. [Latent Semantic Analysis](https://en.wikipedia.org/wiki/Latent_semantic_analysis)), and predictive methods (e.g.[neural probabilistic language models](http://www.scholarpedia.org/article/Neural_net_language_models)).

This distinction is elaborated in much more detail by [Baroni et al.](http://clic.cimec.unitn.it/marco/publications/acl2014/baroni-etal-countpredict-acl2014.pdf), but in a nutshell: Count-based methods compute the statistics of how often some word co-occurs with its neighbor words in a large text corpus, and then map these count-statistics down to a small, dense vector for each word. Predictive models directly try to predict a word from its neighbors in terms of learned small, dense embedding vectors (considered parameters of the model).

Word2vec is a particularly computationally-efficient predictive model for learning word embeddings from raw text. It comes in two flavors, the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model (Section 3.1 and 3.2 in [Mikolov et al.](http://arxiv.org/pdf/1301.3781.pdf)). Algorithmically, these models are similar, except that CBOW predicts target words (e.g. 'mat') from source context words ('the cat sits on the'), while the skip-gram does the inverse and predicts source context-words from the target words. This inversion might seem like an arbitrary choice, but statistically it has the effect that CBOW smoothes over a lot of the distributional information (by treating an entire context as one observation). For the most part, this turns out to be a useful thing for smaller datasets. However, skip-gram treats each context-target pair as a new observation, and this tends to do better when we have larger datasets. We will focus on the skip-gram model in the rest of this tutorial.

## Scaling up with Noise-Contrastive Training

Neural probabilistic language models are traditionally trained using the [maximum likelihood](https://en.wikipedia.org/wiki/Maximum_likelihood) (ML) principle to maximize the probability of the next word wt (for "target") given the previous words h (for "history") in terms of a[softmax function](https://en.wikipedia.org/wiki/Softmax_function),

P(wt|h)=softmax(score(wt,h))=exp⁡{score(wt,h)}∑Word w' in Vocabexp⁡{score(w′,h)}

where score(wt,h) computes the compatibility of word wt with the context h (a dot product is commonly used). We train this model by maximizing its [log-likelihood](https://en.wikipedia.org/wiki/Likelihood_function) on the training set, i.e. by maximizing

JML=log⁡P(wt|h)=score(wt,h)−log⁡(∑Word w' in Vocabexp⁡{score(w′,h)}).

This yields a properly normalized probabilistic model for language modeling. However this is very expensive, because we need to compute and normalize each probability using the score for all other V words w′ in the current context h, at every training step.

On the other hand, for feature learning in word2vec we do not need a full probabilistic model. The CBOW and skip-gram models are instead trained using a binary classification objective ([logistic regression](https://en.wikipedia.org/wiki/Logistic_regression)) to discriminate the real target words wt from k imaginary (noise) words w~, in the same context. We illustrate this below for a CBOW model. For skip-gram the direction is simply inverted.

Mathematically, the objective (for each example) is to maximize

JNEG=log⁡Qθ(D=1|wt,h)+kEw~∼Pnoise⁡[log⁡Qθ(D=0|w~,h)]

where Qθ(D=1|w,h) is the binary logistic regression probability under the model of seeing the word w in the context h in the dataset D, calculated in terms of the learned embedding vectors θ. In practice we approximate the expectation by drawing k contrastive words from the noise distribution (i.e. we compute a [Monte Carlo average](https://en.wikipedia.org/wiki/Monte_Carlo_integration)).

This objective is maximized when the model assigns high probabilities to the real words, and low probabilities to noise words. Technically, this is called [Negative Sampling](http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf), and there is good mathematical motivation for using this loss function: The updates it proposes approximate the updates of the softmax function in the limit. But computationally it is especially appealing because computing the loss function now scales only with the number of noise words that we select (k), and not all words in the vocabulary (V). This makes it much faster to train. We will actually make use of the very similar [noise-contrastive estimation (NCE)](http://papers.nips.cc/paper/5165-learning-word-embeddings-efficiently-with-noise-contrastive-estimation.pdf) loss, for which TensorFlow has a handy helper function tf.nn.nce\_loss().

Let's get an intuitive feel for how this would work in practice!

## The Skip-gram Model

As an example, let's consider the dataset

the quick brown fox jumped over the lazy dog

We first form a dataset of words and the contexts in which they appear. We could define 'context' in any way that makes sense, and in fact people have looked at syntactic contexts (i.e. the syntactic dependents of the current target word, see e.g. [Levy et al.](https://levyomer.files.wordpress.com/2014/04/dependency-based-word-embeddings-acl-2014.pdf)), words-to-the-left of the target, words-to-the-right of the target, etc. For now, let's stick to the vanilla definition and define 'context' as the window of words to the left and to the right of a target word. Using a window size of 1, we then have the dataset

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

of (context, target) pairs. Recall that skip-gram inverts contexts and targets, and tries to predict each context word from its target word, so the task becomes to predict 'the' and 'brown' from 'quick', 'quick' and 'fox' from 'brown', etc. Therefore our dataset becomes

(quick, the), (quick, brown), (brown, quick), (brown, fox), ...

of (input, output) pairs. The objective function is defined over the entire dataset, but we typically optimize this with [stochastic gradient descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) (SGD) using one example at a time (or a 'minibatch' of batch\_sizeexamples, where typically 16 <= batch\_size <= 512). So let's look at one step of this process.

Let's imagine at training step t we observe the first training case above, where the goal is to predict the from quick. We select num\_noise number of noisy (contrastive) examples by drawing from some noise distribution, typically the unigram distribution, P(w). For simplicity let's say num\_noise=1 and we select sheep as a noisy example. Next we compute the loss for this pair of observed and noisy examples, i.e. the objective at time step tbecomes

JNEG(t)=log⁡Qθ(D=1|the, quick)+log⁡(Qθ(D=0|sheep, quick))

The goal is to make an update to the embedding parameters θ to improve (in this case, maximize) this objective function. We do this by deriving the gradient of the loss with respect to the embedding parameters θ, i.e. ∂∂θJNEG(luckily TensorFlow provides easy helper functions for doing this!). We then perform an update to the embeddings by taking a small step in the direction of the gradient. When this process is repeated over the entire training set, this has the effect of 'moving' the embedding vectors around for each word until the model is successful at discriminating real words from noise words.

We can visualize the learned vectors by projecting them down to 2 dimensions using for instance something like the [t-SNE dimensionality reduction technique](http://lvdmaaten.github.io/tsne/). When we inspect these visualizations it becomes apparent that the vectors capture some general, and in fact quite useful, semantic information about words and their relationships to one another. It was very interesting when we first discovered that certain directions in the induced vector space specialize towards certain semantic relationships, e.g. male-female, verb tense and even country-capitalrelationships between words, as illustrated in the figure below (see also for example [Mikolov et al., 2013](http://www.aclweb.org/anthology/N13-1090)).

This explains why these vectors are also useful as features for many canonical NLP prediction tasks, such as part-of-speech tagging or named entity recognition (see for example the original work by [Collobert et al., 2011](http://arxiv.org/abs/1103.0398)([pdf](http://arxiv.org/pdf/1103.0398.pdf)), or follow-up work by [Turian et al., 2010](http://www.aclweb.org/anthology/P10-1040)).

But for now, let's just use them to draw pretty pictures!

## Building the Graph

This is all about embeddings, so let's define our embedding matrix. This is just a big random matrix to start. We'll initialize the values to be uniform in the unit cube.

embeddings = tf.Variable(  
    tf.random\_uniform([vocabulary\_size, embedding\_size], -1.0, 1.0))

The noise-contrastive estimation loss is defined in terms of a logistic regression model. For this, we need to define the weights and biases for each word in the vocabulary (also called the output weights as opposed to the input embeddings). So let's define that.

nce\_weights = tf.Variable(  
  tf.truncated\_normal([vocabulary\_size, embedding\_size],  
                      stddev=1.0 / math.sqrt(embedding\_size)))  
nce\_biases = tf.Variable(tf.zeros([vocabulary\_size]))

Now that we have the parameters in place, we can define our skip-gram model graph. For simplicity, let's suppose we've already integerized our text corpus with a vocabulary so that each word is represented as an integer (see[tensorflow/examples/tutorials/word2vec/word2vec\_basic.py](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/word2vec/word2vec_basic.py) for the details). The skip-gram model takes two inputs. One is a batch full of integers representing the source context words, the other is for the target words. Let's create placeholder nodes for these inputs, so that we can feed in data later.

# Placeholders for inputs  
train\_inputs = tf.placeholder(tf.int32, shape=[batch\_size])  
train\_labels = tf.placeholder(tf.int32, shape=[batch\_size, 1])

Now what we need to do is look up the vector for each of the source words in the batch. TensorFlow has handy helpers that make this easy.

embed = tf.nn.embedding\_lookup(embeddings, train\_inputs)

Ok, now that we have the embeddings for each word, we'd like to try to predict the target word using the noise-contrastive training objective.

# Compute the NCE loss, using a sample of the negative labels each time.  
loss = tf.reduce\_mean(  
  tf.nn.nce\_loss(weights=nce\_weights,  
                 biases=nce\_biases,  
                 labels=train\_labels,  
                 inputs=embed,  
                 num\_sampled=num\_sampled,  
                 num\_classes=vocabulary\_size))

Now that we have a loss node, we need to add the nodes required to compute gradients and update the parameters, etc. For this we will use stochastic gradient descent, and TensorFlow has handy helpers to make this easy as well.

# We use the SGD optimizer.  
optimizer = tf.train.GradientDescentOptimizer(learning\_rate=1.0).minimize(loss)

## Training the Model

Training the model is then as simple as using a feed\_dict to push data into the placeholders and calling[tf.Session.run](https://www.tensorflow.org/api_docs/python/tf/Session#run) with this new data in a loop.

for inputs, labels in generate\_batch(...):  
  feed\_dict = {train\_inputs: inputs, train\_labels: labels}  
  \_, cur\_loss = session.run([optimizer, loss], feed\_dict=feed\_dict)

See the full example code in [tensorflow/examples/tutorials/word2vec/word2vec\_basic.py](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/word2vec/word2vec_basic.py).

## Visualizing the Learned Embeddings

After training has finished we can visualize the learned embeddings using t-SNE.

Et voila! As expected, words that are similar end up clustering nearby each other. For a more heavyweight implementation of word2vec that showcases more of the advanced features of TensorFlow, see the implementation in [models/tutorials/embedding/word2vec.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/embedding/word2vec.py).

## Evaluating Embeddings: Analogical Reasoning

Embeddings are useful for a wide variety of prediction tasks in NLP. Short of training a full-blown part-of-speech model or named-entity model, one simple way to evaluate embeddings is to directly use them to predict syntactic and semantic relationships like king is to queen as father is to ?. This is called analogical reasoningand the task was introduced by [Mikolov and colleagues](http://msr-waypoint.com/en-us/um/people/gzweig/Pubs/NAACL2013Regularities.pdf). Download the dataset for this task from[download.tensorflow.org](http://download.tensorflow.org/data/questions-words.txt).

To see how we do this evaluation, have a look at the build\_eval\_graph() and eval() functions in[models/tutorials/embedding/word2vec.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/embedding/word2vec.py).

The choice of hyperparameters can strongly influence the accuracy on this task. To achieve state-of-the-art performance on this task requires training over a very large dataset, carefully tuning the hyperparameters and making use of tricks like subsampling the data, which is out of the scope of this tutorial.

## Optimizing the Implementation

Our vanilla implementation showcases the flexibility of TensorFlow. For example, changing the training objective is as simple as swapping out the call to tf.nn.nce\_loss() for an off-the-shelf alternative such astf.nn.sampled\_softmax\_loss(). If you have a new idea for a loss function, you can manually write an expression for the new objective in TensorFlow and let the optimizer compute its derivatives. This flexibility is invaluable in the exploratory phase of machine learning model development, where we are trying out several different ideas and iterating quickly.

Once you have a model structure you're satisfied with, it may be worth optimizing your implementation to run more efficiently (and cover more data in less time). For example, the naive code we used in this tutorial would suffer compromised speed because we use Python for reading and feeding data items -- each of which require very little work on the TensorFlow back-end. If you find your model is seriously bottlenecked on input data, you may want to implement a custom data reader for your problem, as described in [New Data Formats](https://www.tensorflow.org/extend/new_data_formats). For the case of Skip-Gram modeling, we've actually already done this for you as an example in[models/tutorials/embedding/word2vec.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/embedding/word2vec.py).

If your model is no longer I/O bound but you want still more performance, you can take things further by writing your own TensorFlow Ops, as described in [Adding a New Op](https://www.tensorflow.org/extend/adding_an_op). Again we've provided an example of this for the Skip-Gram case [models/tutorials/embedding/word2vec\_optimized.py](https://www.tensorflow.org/code/tensorflow_models/tutorials/embedding/word2vec_optimized.py). Feel free to benchmark these against each other to measure performance improvements at each stage.

## Conclusion

In this tutorial we covered the word2vec model, a computationally efficient model for learning word embeddings. We motivated why embeddings are useful, discussed efficient training techniques and showed how to implement all of this in TensorFlow. Overall, we hope that this has show-cased how TensorFlow affords you the flexibility you need for early experimentation, and the control you later need for bespoke optimized implementation.

# Recurrent Neural Networks

## Introduction

Take a look at [this great article](http://colah.github.io/posts/2015-08-Understanding-LSTMs/) for an introduction to recurrent neural networks and LSTMs in particular.

## Language Modeling

In this tutorial we will show how to train a recurrent neural network on a challenging task of language modeling. The goal of the problem is to fit a probabilistic model which assigns probabilities to sentences. It does so by predicting next words in a text given a history of previous words. For this purpose we will use the [Penn Tree Bank](https://catalog.ldc.upenn.edu/ldc99t42)(PTB) dataset, which is a popular benchmark for measuring the quality of these models, whilst being small and relatively fast to train.

Language modeling is key to many interesting problems such as speech recognition, machine translation, or image captioning. It is also fun -- take a look [here](http://karpathy.github.io/2015/05/21/rnn-effectiveness/).

For the purpose of this tutorial, we will reproduce the results from [Zaremba et al., 2014](http://arxiv.org/abs/1409.2329) ([pdf](http://arxiv.org/pdf/1409.2329.pdf)), which achieves very good quality on the PTB dataset.

## Tutorial Files

This tutorial references the following files from models/tutorials/rnn/ptb in the [TensorFlow models repo](https://github.com/tensorflow/models):

| File | Purpose |
| --- | --- |
| ptb\_word\_lm.py | The code to train a language model on the PTB dataset. |
| reader.py | The code to read the dataset. |

## Download and Prepare the Data

The data required for this tutorial is in the data/ directory of the [PTB dataset from Tomas Mikolov's webpage](http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz).

The dataset is already preprocessed and contains overall 10000 different words, including the end-of-sentence marker and a special symbol (\<unk>) for rare words. In reader.py, we convert each word to a unique integer identifier, in order to make it easy for the neural network to process the data.

## The Model

### LSTM

The core of the model consists of an LSTM cell that processes one word at a time and computes probabilities of the possible values for the next word in the sentence. The memory state of the network is initialized with a vector of zeros and gets updated after reading each word. For computational reasons, we will process data in mini-batches of size batch\_size.

The basic pseudocode is as follows:

lstm = tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
# Initial state of the LSTM memory.  
state = tf.zeros([batch\_size, lstm.state\_size])  
probabilities = []  
loss = 0.0  
for current\_batch\_of\_words in words\_in\_dataset:  
    # The value of state is updated after processing each batch of words.  
    output, state = lstm(current\_batch\_of\_words, state)  
  
    # The LSTM output can be used to make next word predictions  
    logits = tf.matmul(output, softmax\_w) + softmax\_b  
    probabilities.append(tf.nn.softmax(logits))  
    loss += loss\_function(probabilities, target\_words)

### Truncated Backpropagation

By design, the output of a recurrent neural network (RNN) depends on arbitrarily distant inputs. Unfortunately, this makes backpropagation computation difficult. In order to make the learning process tractable, it is common practice to create an "unrolled" version of the network, which contains a fixed number (num\_steps) of LSTM inputs and outputs. The model is then trained on this finite approximation of the RNN. This can be implemented by feeding inputs of length num\_steps at a time and performing a backward pass after each such input block.

Here is a simplified block of code for creating a graph which performs truncated backpropagation:

# Placeholder for the inputs in a given iteration.  
words = tf.placeholder(tf.int32, [batch\_size, num\_steps])  
  
lstm = tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
# Initial state of the LSTM memory.  
initial\_state = state = tf.zeros([batch\_size, lstm.state\_size])  
  
for i in range(num\_steps):  
    # The value of state is updated after processing each batch of words.  
    output, state = lstm(words[:, i], state)  
  
    # The rest of the code.  
    # ...  
  
final\_state = state

And this is how to implement an iteration over the whole dataset:

# A numpy array holding the state of LSTM after each batch of words.  
numpy\_state = initial\_state.eval()  
total\_loss = 0.0  
for current\_batch\_of\_words in words\_in\_dataset:  
    numpy\_state, current\_loss = session.run([final\_state, loss],  
        # Initialize the LSTM state from the previous iteration.  
        feed\_dict={initial\_state: numpy\_state, words: current\_batch\_of\_words})  
    total\_loss += current\_loss

### Inputs

The word IDs will be embedded into a dense representation (see the [Vector Representations Tutorial](https://www.tensorflow.org/tutorials/word2vec)) before feeding to the LSTM. This allows the model to efficiently represent the knowledge about particular words. It is also easy to write:

# embedding\_matrix is a tensor of shape [vocabulary\_size, embedding size]  
word\_embeddings = tf.nn.embedding\_lookup(embedding\_matrix, word\_ids)

The embedding matrix will be initialized randomly and the model will learn to differentiate the meaning of words just by looking at the data.

### Loss Function

We want to minimize the average negative log probability of the target words:

loss=−1N∑i=1Nln⁡ptargeti

It is not very difficult to implement but the function sequence\_loss\_by\_example is already available, so we can just use it here.

The typical measure reported in the papers is average per-word perplexity (often just called perplexity), which is equal to

e−1N∑i=1Nln⁡ptargeti=eloss

and we will monitor its value throughout the training process.

### Stacking multiple LSTMs

To give the model more expressive power, we can add multiple layers of LSTMs to process the data. The output of the first layer will become the input of the second and so on.

We have a class called MultiRNNCell that makes the implementation seamless:

def lstm\_cell():  
  return tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
stacked\_lstm = tf.contrib.rnn.MultiRNNCell(  
    [lstm\_cell() for \_ in range(number\_of\_layers)])  
  
initial\_state = state = stacked\_lstm.zero\_state(batch\_size, tf.float32)  
for i in range(num\_steps):  
    # The value of state is updated after processing each batch of words.  
    output, state = stacked\_lstm(words[:, i], state)  
  
    # The rest of the code.  
    # ...  
  
final\_state = state

## Run the Code

Before running the code, download the PTB dataset, as discussed at the beginning of this tutorial. Then, extract the PTB dataset underneath your home directory as follows:

tar xvfz simple-examples.tgz -C $HOME

(Note: On Windows, you may need to use [*other tools*](https://wiki.haskell.org/How_to_unpack_a_tar_file_in_Windows).)

Now, clone the [TensorFlow models repo](https://github.com/tensorflow/models) from GitHub. Run the following commands:

cd models/tutorials/rnn/ptb  
python ptb\_word\_lm.py --data\_path=$HOME/simple-examples/data/ --model=small

There are 3 supported model configurations in the tutorial code: "small", "medium" and "large". The difference between them is in size of the LSTMs and the set of hyperparameters used for training.

The larger the model, the better results it should get. The small model should be able to reach perplexity below 120 on the test set and the large one below 80, though it might take several hours to train.

## What Next?

There are several tricks that we haven't mentioned that make the model better, including:

* decreasing learning rate schedule,
* dropout between the LSTM layers.

Study the code and modify it to improve the model even further.

# Sequence-to-Sequence Models

Recurrent neural networks can learn to model language, as already discussed in the [RNN Tutorial](https://www.tensorflow.org/tutorials/recurrent) (if you did not read it, please go through it before proceeding with this one). This raises an interesting question: could we condition the generated words on some input and generate a meaningful response? For example, could we train a neural network to translate from English to French? It turns out that the answer is yes.

This tutorial will show you how to build and train such a system end-to-end. Clone the [TensorFlow main repo](https://github.com/tensorflow/tensorflow) and the [TensorFlow models repo](https://github.com/tensorflow/models) from GitHub. You can then start by running the translate program:

cd models/tutorials/rnn/translate  
python translate.py --data\_dir [your\_data\_directory]

It will download English-to-French translation data from the [WMT'15 Website](http://www.statmt.org/wmt15/translation-task.html), prepare it for training, and train. It takes about 20GB of disk space, and a while to download and prepare (see [later](https://www.tensorflow.org/tutorials/seq2seq#lets_run_it) for details), so you can start and leave it running while reading this tutorial.

This tutorial references the following files.

| File | What's in it? |
| --- | --- |
| tensorflow/tensorflow/python/ops/seq2seq.py | Library for building sequence-to-sequence models. |
| models/tutorials/rnn/translate/seq2seq\_model.py | Neural translation sequence-to-sequence model. |
| models/tutorials/rnn/translate/data\_utils.py | Helper functions for preparing translation data. |
| models/tutorials/rnn/translate/translate.py | Binary that trains and runs the translation model. |

## Sequence-to-sequence basics

A basic sequence-to-sequence model, as introduced in [Cho et al., 2014](http://arxiv.org/abs/1406.1078) ([pdf](http://arxiv.org/pdf/1406.1078.pdf)), consists of two recurrent neural networks (RNNs): an encoder that processes the input and a decoder that generates the output. This basic architecture is depicted below.

Each box in the picture above represents a cell of the RNN, most commonly a GRU cell or an LSTM cell (see the [RNN Tutorial](https://www.tensorflow.org/tutorials/recurrent) for an explanation of those). Encoder and decoder can share weights or, as is more common, use a different set of parameters. Multi-layer cells have been successfully used in sequence-to-sequence models too, e.g. for translation [Sutskever et al., 2014](http://arxiv.org/abs/1409.3215) ([pdf](http://arxiv.org/pdf/1409.3215.pdf)).

In the basic model depicted above, every input has to be encoded into a fixed-size state vector, as that is the only thing passed to the decoder. To allow the decoder more direct access to the input, an attention mechanism was introduced in [Bahdanau et al., 2014](http://arxiv.org/abs/1409.0473) ([pdf](http://arxiv.org/pdf/1409.0473.pdf)). We will not go into the details of the attention mechanism (see the paper); suffice it to say that it allows the decoder to peek into the input at every decoding step. A multi-layer sequence-to-sequence network with LSTM cells and attention mechanism in the decoder looks like this.

## TensorFlow seq2seq library

As you can see above, there are many different sequence-to-sequence models. Each of these models can use different RNN cells, but all of them accept encoder inputs and decoder inputs. This motivates the interfaces in the TensorFlow seq2seq library (tensorflow/tensorflow/python/ops/seq2seq.py). The basic RNN encoder-decoder sequence-to-sequence model works as follows.

outputs, states = basic\_rnn\_seq2seq(encoder\_inputs, decoder\_inputs, cell)

In the above call, encoder\_inputs are a list of tensors representing inputs to the encoder, i.e., corresponding to the letters A, B, C in the first picture above. Similarly, decoder\_inputs are tensors representing inputs to the decoder, GO, W, X, Y, Z on the first picture.

The cell argument is an instance of the tf.contrib.rnn.RNNCell class that determines which cell will be used inside the model. You can use an existing cell, such as GRUCell or LSTMCell, or you can write your own. Moreover, tf.contrib.rnn provides wrappers to construct multi-layer cells, add dropout to cell inputs or outputs, or to do other transformations. See the [RNN Tutorial](https://www.tensorflow.org/tutorials/recurrent) for examples.

The call to basic\_rnn\_seq2seq returns two arguments: outputs and states. Both of them are lists of tensors of the same length as decoder\_inputs. Naturally, outputs correspond to the outputs of the decoder in each time-step, in the first picture above that would be W, X, Y, Z, EOS. The returned states represent the internal state of the decoder at every time-step.

In many applications of sequence-to-sequence models, the output of the decoder at time t is fed back and becomes the input of the decoder at time t+1. At test time, when decoding a sequence, this is how the sequence is constructed. During training, on the other hand, it is common to provide the correct input to the decoder at every time-step, even if the decoder made a mistake before. Functions in seq2seq.py support both modes using the feed\_previous argument. For example, let's analyze the following use of an embedding RNN model.

outputs, states = embedding\_rnn\_seq2seq(  
    encoder\_inputs, decoder\_inputs, cell,  
    num\_encoder\_symbols, num\_decoder\_symbols,  
    embedding\_size, output\_projection=None,  
    feed\_previous=False)

In the embedding\_rnn\_seq2seq model, all inputs (both encoder\_inputs and decoder\_inputs) are integer-tensors that represent discrete values. They will be embedded into a dense representation (see the [Vectors Representations Tutorial](https://www.tensorflow.org/tutorials/word2vec) for more details on embeddings), but to construct these embeddings we need to specify the maximum number of discrete symbols that will appear: num\_encoder\_symbols on the encoder side, and num\_decoder\_symbols on the decoder side.

In the above invocation, we set feed\_previous to False. This means that the decoder will use decoder\_inputstensors as provided. If we set feed\_previous to True, the decoder would only use the first element of decoder\_inputs. All other tensors from this list would be ignored, and instead the previous output of the decoder would be used. This is used for decoding translations in our translation model, but it can also be used during training, to make the model more robust to its own mistakes, similar to [Bengio et al., 2015](http://arxiv.org/abs/1506.03099) ([pdf](http://arxiv.org/pdf/1506.03099.pdf)).

One more important argument used above is output\_projection. If not specified, the outputs of the embedding model will be tensors of shape batch-size by num\_decoder\_symbols as they represent the logits for each generated symbol. When training models with large output vocabularies, i.e., when num\_decoder\_symbolsis large, it is not practical to store these large tensors. Instead, it is better to return smaller output tensors, which will later be projected onto a large output tensor using output\_projection. This allows to use our seq2seq models with a sampled softmax loss, as described in [Jean et. al., 2014](http://arxiv.org/abs/1412.2007) ([pdf](http://arxiv.org/pdf/1412.2007.pdf)).

In addition to basic\_rnn\_seq2seq and embedding\_rnn\_seq2seq there are a few more sequence-to-sequence models in seq2seq.py; take a look there. They all have similar interfaces, so we will not describe them in detail. We will use embedding\_attention\_seq2seq for our translation model below.

## Neural translation model

While the core of the sequence-to-sequence model is constructed by the functions in tensorflow/tensorflow/python/ops/seq2seq.py, there are still a few tricks that are worth mentioning that are used in our translation model in models/tutorials/rnn/translate/seq2seq\_model.py.

### Sampled softmax and output projection

For one, as already mentioned above, we want to use sampled softmax to handle large output vocabulary. To decode from it, we need to keep track of the output projection. Both the sampled softmax loss and the output projections are constructed by the following code in seq2seq\_model.py.

if num\_samples > 0 and num\_samples < self.target\_vocab\_size:  
  w\_t = tf.get\_variable("proj\_w", [self.target\_vocab\_size, size], dtype=dtype)  
  w = tf.transpose(w\_t)  
  b = tf.get\_variable("proj\_b", [self.target\_vocab\_size], dtype=dtype)  
  output\_projection = (w, b)  
  
  def sampled\_loss(labels, inputs):  
    labels = tf.reshape(labels, [-1, 1])  
    # We need to compute the sampled\_softmax\_loss using 32bit floats to  
    # avoid numerical instabilities.  
    local\_w\_t = tf.cast(w\_t, tf.float32)  
    local\_b = tf.cast(b, tf.float32)  
    local\_inputs = tf.cast(inputs, tf.float32)  
    return tf.cast(  
        tf.nn.sampled\_softmax\_loss(  
            weights=local\_w\_t,  
            biases=local\_b,  
            labels=labels,  
            inputs=local\_inputs,  
            num\_sampled=num\_samples,  
            num\_classes=self.target\_vocab\_size),  
        dtype)

First, note that we only construct a sampled softmax if the number of samples (512 by default) is smaller than the target vocabulary size. For vocabularies smaller than 512, it might be a better idea to just use a standard softmax loss.

Then, as you can see, we construct an output projection. It is a pair, consisting of a weight matrix and a bias vector. If used, the rnn cell will return vectors of shape batch-size by size, rather than batch-size by target\_vocab\_size. To recover logits, we need to multiply by the weight matrix and add the biases, as is done in lines 124-126 in seq2seq\_model.py.

if output\_projection is not None:  
  for b in xrange(len(buckets)):  
    self.outputs[b] = [tf.matmul(output, output\_projection[0]) +  
                       output\_projection[1] for ...]

### Bucketing and padding

In addition to sampled softmax, our translation model also makes use of bucketing, which is a method to efficiently handle sentences of different lengths. Let us first clarify the problem. When translating English to French, we will have English sentences of different lengths L1 on input, and French sentences of different lengths L2 on output. Since the English sentence is passed as encoder\_inputs, and the French sentence comes as decoder\_inputs (prefixed by a GO symbol), we should in principle create a seq2seq model for every pair (L1, L2+1) of lengths of an English and French sentence. This would result in an enormous graph consisting of many very similar subgraphs. On the other hand, we could just pad every sentence with a special PAD symbol. Then we'd need only one seq2seq model, for the padded lengths. But on shorter sentence our model would be inefficient, encoding and decoding many PAD symbols that are useless.

As a compromise between constructing a graph for every pair of lengths and padding to a single length, we use a number of buckets and pad each sentence to the length of the bucket above it. In translate.py we use the following default buckets.

buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]

This means that if the input is an English sentence with 3 tokens, and the corresponding output is a French sentence with 6 tokens, then they will be put in the first bucket and padded to length 5 for encoder inputs, and length 10 for decoder inputs. If we have an English sentence with 8 tokens and the corresponding French sentence has 18 tokens, then they will not fit into the (10, 15) bucket, and so the (20, 25) bucket will be used, i.e. the English sentence will be padded to 20, and the French one to 25.

Remember that when constructing decoder inputs we prepend the special GO symbol to the input data. This is done in the get\_batch() function in seq2seq\_model.py, which also reverses the input English sentence. Reversing the inputs was shown to improve results for the neural translation model in [Sutskever et al., 2014](http://arxiv.org/abs/1409.3215) ([pdf](http://arxiv.org/pdf/1409.3215.pdf)). To put it all together, imagine we have the sentence "I go.", tokenized as ["I", "go", "."] as input and the sentence "Je vais." as output, tokenized ["Je", "vais", "."]. It will be put in the (5, 10) bucket, with encoder inputs representing [PAD PAD "." "go" "I"] and decoder inputs [GO "Je" "vais" "." EOS PAD PAD PAD PAD PAD].

## Let's run it

To train the model described above, we need to a large English-French corpus. We will use the 10^9-French-English corpus from the [WMT'15 Website](http://www.statmt.org/wmt15/translation-task.html) for training, and the 2013 news test from the same site as development set. Both data-sets will be downloaded to data\_dir and training will start, saving checkpoints in train\_dir, when this command is run.

python translate.py  
  --data\_dir [your\_data\_directory] --train\_dir [checkpoints\_directory]  
  --en\_vocab\_size=40000 --fr\_vocab\_size=40000

It takes about 18GB of disk space and several hours to prepare the training corpus. It is unpacked, vocabulary files are created in data\_dir, and then the corpus is tokenized and converted to integer ids. Note the parameters that determine vocabulary sizes. In the example above, all words outside the 40K most common ones will be converted to an UNK token representing unknown words. So if you change vocabulary size, the binary will re-map the corpus to token-ids again.

After the data is prepared, training starts. Default parameters in translate are set to quite large values. Large models trained over a long time give good results, but it might take too long or use too much memory for your GPU. You can request to train a smaller model as in the following example.

python translate.py  
  --data\_dir [your\_data\_directory] --train\_dir [checkpoints\_directory]  
  --size=256 --num\_layers=2 --steps\_per\_checkpoint=50

The above command will train a model with 2 layers (the default is 3), each layer with 256 units (default is 1024), and will save a checkpoint every 50 steps (the default is 200). You can play with these parameters to find out how large a model can be to fit into the memory of your GPU.

During training, every steps\_per\_checkpoint steps the binary will print out statistics from recent steps. With the default parameters (3 layers of size 1024), first messages look like this.

global step 200 learning rate 0.5000 step-time 1.39 perplexity 1720.62  
  eval: bucket 0 perplexity 184.97  
  eval: bucket 1 perplexity 248.81  
  eval: bucket 2 perplexity 341.64  
  eval: bucket 3 perplexity 469.04  
global step 400 learning rate 0.5000 step-time 1.38 perplexity 379.89  
  eval: bucket 0 perplexity 151.32  
  eval: bucket 1 perplexity 190.36  
  eval: bucket 2 perplexity 227.46  
  eval: bucket 3 perplexity 238.66

You can see that each step takes just under 1.4 seconds, the perplexity on the training set and the perplexities on the development set for each bucket. After about 30K steps, we see perplexities on short sentences (bucket 0 and 1) going into single digits. Since the training corpus contains ~22M sentences, one epoch (going through the training data once) takes about 340K steps with batch-size of 64. At this point the model can be used for translating English sentences to French using the --decode option.

python translate.py --decode  
  --data\_dir [your\_data\_directory] --train\_dir [checkpoints\_directory]  
  
Reading model parameters from /tmp/translate.ckpt-340000  
>  Who is the president of the United States?  
 Qui est le président des États-Unis ?

## What next?

The example above shows how you can build your own English-to-French translator, end-to-end. Run it and see how the model performs for yourself. While it has reasonable quality, the default parameters will not give you the best translation model. Here are a few things you can improve.

First of all, we use a very primitive tokenizer, the basic\_tokenizer function in data\_utils. A better tokenizer can be found on the [WMT'15 Website](http://www.statmt.org/wmt15/translation-task.html). Using that tokenizer, and a larger vocabulary, should improve your translations.

Also, the default parameters of the translation model are not tuned. You can try changing the learning rate, decay, or initializing the weights of your model in a different way. You can also change the defaultGradientDescentOptimizer in seq2seq\_model.py to a more advanced one, such as AdagradOptimizer. Try these things and see how they improve your results!

Finally, the model presented above can be used for any sequence-to-sequence task, not only for translation. Even if you want to transform a sequence to a tree, for example to generate a parsing tree, the same model as above can give state-of-the-art results, as demonstrated in [Vinyals & Kaiser et al., 2014](http://arxiv.org/abs/1412.7449) ([pdf](http://arxiv.org/pdf/1412.7449.pdf)). So you can not only build your own translator, you can also build a parser, a chat-bot, or any program that comes to your mind. Experiment!

# Large-scale Linear Models with TensorFlow

The tf.learn API provides (among other things) a rich set of tools for working with linear models in TensorFlow. This document provides an overview of those tools. It explains:

* what a linear model is.
* why you might want to use a linear model.
* how tf.learn makes it easy to build linear models in TensorFlow.
* how you can use tf.learn to combine linear models with deep learning to get the advantages of both.

Read this overview to decide whether the tf.learn linear model tools might be useful to you. Then do the [Linear Models tutorial](https://www.tensorflow.org/tutorials/wide) to give it a try. This overview uses code samples from the tutorial, but the tutorial walks through the code in greater detail.

To understand this overview it will help to have some familiarity with basic machine learning concepts, and also with [tf.learn](https://www.tensorflow.org/get_started/tflearn).

## What is a linear model?

A linear model uses a single weighted sum of features to make a prediction. For example, if you have [data](https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names) on age, years of education, and weekly hours of work for a population, you can learn weights for each of those numbers so that their weighted sum estimates a person's salary. You can also use linear models for classification.

Some linear models transform the weighted sum into a more convenient form. For example, logistic regressionplugs the weighted sum into the logistic function to turn the output into a value between 0 and 1. But you still just have one weight for each input feature.

## Why would you want to use a linear model?

Why would you want to use so simple a model when recent research has demonstrated the power of more complex neural networks with many layers?

Linear models:

* train quickly, compared to deep neural nets.
* can work well on very large feature sets.
* can be trained with algorithms that don't require a lot of fiddling with learning rates, etc.
* can be interpreted and debugged more easily than neural nets. You can examine the weights assigned to each feature to figure out what's having the biggest impact on a prediction.
* provide an excellent starting point for learning about machine learning.
* are widely used in industry.

## How does tf.learn help you build linear models?

You can build a linear model from scratch in TensorFlow without the help of a special API. But tf.learn provides some tools that make it easier to build effective large-scale linear models.

### Feature columns and transformations

Much of the work of designing a linear model consists of transforming raw data into suitable input features. tf.learn uses the FeatureColumn abstraction to enable these transformations.

A FeatureColumn represents a single feature in your data. A FeatureColumn may represent a quantity like 'height', or it may represent a category like 'eye\_color' where the value is drawn from a set of discrete possibilities like {'blue', 'brown', 'green'}.

In the case of both continuous features like 'height' and categorical features like 'eye\_color', a single value in the data might get transformed into a sequence of numbers before it is input into the model. The FeatureColumnabstraction lets you manipulate the feature as a single semantic unit in spite of this fact. You can specify transformations and select features to include without dealing with specific indices in the tensors you feed into the model.

#### Sparse columns

Categorical features in linear models are typically translated into a sparse vector in which each possible value has a corresponding index or id. For example, if there are only three possible eye colors you can represent 'eye\_color' as a length 3 vector: 'brown' would become [1, 0, 0], 'blue' would become [0, 1, 0] and 'green' would become [0, 0, 1]. These vectors are called "sparse" because they may be very long, with many zeros, when the set of possible values is very large (such as all English words).

While you don't need to use sparse columns to use tf.learn linear models, one of the strengths of linear models is their ability to deal with large sparse vectors. Sparse features are a primary use case for the tf.learn linear model tools.

##### Encoding sparse columns

FeatureColumn handles the conversion of categorical values into vectors automatically, with code like this:

eye\_color = tf.contrib.layers.sparse\_column\_with\_keys(  
  column\_name="eye\_color", keys=["blue", "brown", "green"])

where eye\_color is the name of a column in your source data.

You can also generate FeatureColumns for categorical features for which you don't know all possible values. For this case you would use sparse\_column\_with\_hash\_bucket(), which uses a hash function to assign indices to feature values.

education = tf.contrib.layers.sparse\_column\_with\_hash\_bucket(\  
    "education", hash\_bucket\_size=1000)

##### Feature Crosses

Because linear models assign independent weights to separate features, they can't learn the relative importance of specific combinations of feature values. If you have a feature 'favorite\_sport' and a feature 'home\_city' and you're trying to predict whether a person likes to wear red, your linear model won't be able to learn that baseball fans from St. Louis especially like to wear red.

You can get around this limitation by creating a new feature 'favorite\_sport\_x\_home\_city'. The value of this feature for a given person is just the concatenation of the values of the two source features: 'baseball\_x\_stlouis', for example. This sort of combination feature is called a feature cross.

The crossed\_column() method makes it easy to set up feature crosses:

sport = tf.contrib.layers.sparse\_column\_with\_hash\_bucket(\  
    "sport", hash\_bucket\_size=1000)  
city = tf.contrib.layers.sparse\_column\_with\_hash\_bucket(\  
    "city", hash\_bucket\_size=1000)  
sport\_x\_city = tf.contrib.layers.crossed\_column(  
    [sport, city], hash\_bucket\_size=int(1e4))

#### Continuous columns

You can specify a continuous feature like so:

age = tf.contrib.layers.real\_valued\_column("age")

Although, as a single real number, a continuous feature can often be input directly into the model, tf.learn offers useful transformations for this sort of column as well.

##### Bucketization

Bucketization turns a continuous column into a categorical column. This transformation lets you use continuous features in feature crosses, or learn cases where specific value ranges have particular importance.

Bucketization divides the range of possible values into subranges called buckets:

age\_buckets = tf.contrib.layers.bucketized\_column(  
    age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])

The bucket into which a value falls becomes the categorical label for that value.

#### Input function

FeatureColumns provide a specification for the input data for your model, indicating how to represent and transform the data. But they do not provide the data itself. You provide the data through an input function.

The input function must return a dictionary of tensors. Each key corresponds to the name of a FeatureColumn. Each key's value is a tensor containing the values of that feature for all data instances. See [Building Input Functions with tf.contrib.learn](https://www.tensorflow.org/get_started/input_fn) for a more comprehensive look at input functions, and input\_fn in the [linear models tutorial code](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/learn/wide_n_deep_tutorial.py) for an example implementation of an input function.

The input function is passed to the fit() and evaluate() calls that initiate training and testing, as described in the next section.

### Linear estimators

tf.learn's estimator classes provide a unified training and evaluation harness for regression and classification models. They take care of the details of the training and evaluation loops and allow the user to focus on model inputs and architecture.

To build a linear estimator, you can use either the tf.contrib.learn.LinearClassifier estimator or thetf.contrib.learn.LinearRegressor estimator, for classification and regression respectively.

As with all tf.learn estimators, to run the estimator you just:

1. Instantiate the estimator class. For the two linear estimator classes, you pass a list of FeatureColumns to the constructor.
2. Call the estimator's fit() method to train it.
3. Call the estimator's evaluate() method to see how it does.

For example:

e = tf.contrib.learn.LinearClassifier(feature\_columns=[  
  native\_country, education, occupation, workclass, marital\_status,  
  race, age\_buckets, education\_x\_occupation, age\_buckets\_x\_race\_x\_occupation],  
  model\_dir=YOUR\_MODEL\_DIRECTORY)  
e.fit(input\_fn=input\_fn\_train, steps=200)  
# Evaluate for one step (one pass through the test data).  
results = e.evaluate(input\_fn=input\_fn\_test, steps=1)  
  
# Print the stats for the evaluation.  
for key in sorted(results):  
    print("%s: %s" % (key, results[key]))

### Wide and deep learning

The tf.learn API also provides an estimator class that lets you jointly train a linear model and a deep neural network. This novel approach combines the ability of linear models to "memorize" key features with the generalization ability of neural nets. Use tf.contrib.learn.DNNLinearCombinedClassifier to create this sort of "wide and deep" model:

e = tf.contrib.learn.DNNLinearCombinedClassifier(  
    model\_dir=YOUR\_MODEL\_DIR,  
    linear\_feature\_columns=wide\_columns,  
    dnn\_feature\_columns=deep\_columns,  
    dnn\_hidden\_units=[100, 50])

For more information, see the [Wide and Deep Learning tutorial](https://www.tensorflow.org/tutorials/wide_and_deep).

# TensorFlow Linear Model Tutorial

In this tutorial, we will use the TF.Learn API in TensorFlow to solve a binary classification problem: Given census data about a person such as age, gender, education and occupation (the features), we will try to predict whether or not the person earns more than 50,000 dollars a year (the target label). We will train a **logistic regression**model, and given an individual's information our model will output a number between 0 and 1, which can be interpreted as the probability that the individual has an annual income of over 50,000 dollars.

## Setup

To try the code for this tutorial:

1. [Install TensorFlow](https://www.tensorflow.org/install/index) if you haven't already.
2. Download [the tutorial code](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/learn/wide_n_deep_tutorial.py).
3. Install the pandas data analysis library. tf.learn doesn't require pandas, but it does support it, and this tutorial uses pandas. To install pandas:

a. Get pip:

# Ubuntu/Linux 64-bit  
$ sudo apt-get install python-pip python-dev  
  
# Mac OS X  
$ sudo easy\_install pip  
$ sudo easy\_install --upgrade six

b. Use pip to install pandas:

$ sudo pip install pandas

If you have trouble installing pandas, consult the [instructions](http://pandas.pydata.org/pandas-docs/stable/install.html) on the pandas site.

1. Execute the tutorial code with the following command to train the linear model described in this tutorial:

$ python wide\_n\_deep\_tutorial.py --model\_type=wide

Read on to find out how this code builds its linear model.

## Reading The Census Data

The dataset we'll be using is the [Census Income Dataset](https://archive.ics.uci.edu/ml/datasets/Census+Income). You can download the [training data](https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data) and [test data](https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.test)manually or use code like this:

import tempfile  
import urllib  
train\_file = tempfile.NamedTemporaryFile()  
test\_file = tempfile.NamedTemporaryFile()  
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.data", train\_file.name)  
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.test", test\_file.name)

Once the CSV files are downloaded, let's read them into [Pandas](http://pandas.pydata.org/) dataframes.

import pandas as pd  
COLUMNS = ["age", "workclass", "fnlwgt", "education", "education\_num",  
           "marital\_status", "occupation", "relationship", "race", "gender",  
           "capital\_gain", "capital\_loss", "hours\_per\_week", "native\_country",  
           "income\_bracket"]  
df\_train = pd.read\_csv(train\_file, names=COLUMNS, skipinitialspace=True)  
df\_test = pd.read\_csv(test\_file, names=COLUMNS, skipinitialspace=True, skiprows=1)

Since the task is a binary classification problem, we'll construct a label column named "label" whose value is 1 if the income is over 50K, and 0 otherwise.

LABEL\_COLUMN = "label"  
df\_train[LABEL\_COLUMN] = (df\_train["income\_bracket"].apply(lambda x: ">50K" in x)).astype(int)  
df\_test[LABEL\_COLUMN] = (df\_test["income\_bracket"].apply(lambda x: ">50K" in x)).astype(int)

Next, let's take a look at the dataframe and see which columns we can use to predict the target label. The columns can be grouped into two types—categorical and continuous columns:

* A column is called **categorical** if its value can only be one of the categories in a finite set. For example, the native country of a person (U.S., India, Japan, etc.) or the education level (high school, college, etc.) are categorical columns.
* A column is called **continuous** if its value can be any numerical value in a continuous range. For example, the capital gain of a person (e.g. $14,084) is a continuous column.

CATEGORICAL\_COLUMNS = ["workclass", "education", "marital\_status", "occupation",  
                       "relationship", "race", "gender", "native\_country"]  
CONTINUOUS\_COLUMNS = ["age", "education\_num", "capital\_gain", "capital\_loss", "hours\_per\_week"]

Here's a list of columns available in the Census Income dataset:

| Column Name | Type | Description |
| --- | --- | --- |
| age | Continuous | The age of the individual |
| workclass | Categorical | The type of employer the individual has (government, military, private, etc.). |
| fnlwgt | Continuous | The number of people the census takers believe that observation represents (sample weight). This variable will not be used. |
| education | Categorical | The highest level of education achieved for that individual. |
| education\_num | Continuous | The highest level of education in numerical form. |
| marital\_status | Categorical | Marital status of the individual. |
| occupation | Categorical | The occupation of the individual. |
| relationship | Categorical | Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried. |
| race | Categorical | White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black. |
| gender | Categorical | Female, Male. |
| capital\_gain | Continuous | Capital gains recorded. |
| capital\_loss | Continuous | Capital Losses recorded. |
| hours\_per\_week | Continuous | Hours worked per week. |
| native\_country | Categorical | Country of origin of the individual. |
| income | Categorical | ">50K" or "<=50K", meaning whether the person makes more than \$50,000 annually. |

## Converting Data into Tensors

When building a TF.Learn model, the input data is specified by means of an Input Builder function. This builder function will not be called until it is later passed to TF.Learn methods such as fit and evaluate. The purpose of this function is to construct the input data, which is represented in the form of [tf.Tensor](https://www.tensorflow.org/api_docs/python/tf/Tensor)s or[tf.SparseTensor](https://www.tensorflow.org/api_docs/python/tf/SparseTensor)s. In more detail, the Input Builder function returns the following as a pair:

1. feature\_cols: A dict from feature column names to Tensors or SparseTensors.
2. label: A Tensor containing the label column.

The keys of the feature\_cols will be used to construct columns in the next section. Because we want to call the fit and evaluate methods with different data, we define two different input builder functions,train\_input\_fn and test\_input\_fn which are identical except that they pass different data to input\_fn. Note that input\_fn will be called while constructing the TensorFlow graph, not while running the graph. What it is returning is a representation of the input data as the fundamental unit of TensorFlow computations, a Tensor(or SparseTensor).

Our model represents the input data as constant tensors, meaning that the tensor represents a constant value, in this case the values of a particular column of df\_train or df\_test. This is the simplest way to pass data into TensorFlow. Another more advanced way to represent input data would be to construct an [Inputs And Readers](https://www.tensorflow.org/api_guides/python/io_ops#inputs_and_readers)that represents a file or other data source, and iterates through the file as TensorFlow runs the graph. Each continuous column in the train or test dataframe will be converted into a Tensor, which in general is a good format to represent dense data. For categorical data, we must represent the data as a SparseTensor. This data format is good for representing sparse data.

import tensorflow as tf  
  
def input\_fn(df):  
  # Creates a dictionary mapping from each continuous feature column name (k) to  
  # the values of that column stored in a constant Tensor.  
  continuous\_cols = {k: tf.constant(df[k].values)  
                     for k in CONTINUOUS\_COLUMNS}  
  # Creates a dictionary mapping from each categorical feature column name (k)  
  # to the values of that column stored in a tf.SparseTensor.  
  categorical\_cols = {k: tf.SparseTensor(  
      indices=[[i, 0] for i in range(df[k].size)],  
      values=df[k].values,  
      dense\_shape=[df[k].size, 1])  
                      for k in CATEGORICAL\_COLUMNS}  
  # Merges the two dictionaries into one.  
  feature\_cols = dict(continuous\_cols.items() + categorical\_cols.items())  
  # Converts the label column into a constant Tensor.  
  label = tf.constant(df[LABEL\_COLUMN].values)  
  # Returns the feature columns and the label.  
  return feature\_cols, label  
  
def train\_input\_fn():  
  return input\_fn(df\_train)  
  
def eval\_input\_fn():  
  return input\_fn(df\_test)

## Selecting and Engineering Features for the Model

Selecting and crafting the right set of feature columns is key to learning an effective model. A **feature column** can be either one of the raw columns in the original dataframe (let's call them **base feature columns**), or any new columns created based on some transformations defined over one or multiple base columns (let's call them **derived feature columns**). Basically, "feature column" is an abstract concept of any raw or derived variable that can be used to predict the target label.

### Base Categorical Feature Columns

To define a feature column for a categorical feature, we can create a SparseColumn using the TF.Learn API. If you know the set of all possible feature values of a column and there are only a few of them, you can usesparse\_column\_with\_keys. Each key in the list will get assigned an auto-incremental ID starting from 0. For example, for the gender column we can assign the feature string "Female" to an integer ID of 0 and "Male" to 1 by doing:

gender = tf.contrib.layers.sparse\_column\_with\_keys(  
  column\_name="gender", keys=["Female", "Male"])

What if we don't know the set of possible values in advance? Not a problem. We can use sparse\_column\_with\_hash\_bucket instead:

education = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("education", hash\_bucket\_size=1000)

What will happen is that each possible value in the feature column education will be hashed to an integer ID as we encounter them in training. See an example illustration below:

| ID | Feature |
| --- | --- |
| ... |  |
| 9 | "Bachelors" |
| ... |  |
| 103 | "Doctorate" |
| ... |  |
| 375 | "Masters" |
| ... |  |

No matter which way we choose to define a SparseColumn, each feature string will be mapped into an integer ID by looking up a fixed mapping or by hashing. Note that hashing collisions are possible, but may not significantly impact the model quality. Under the hood, the LinearModel class is responsible for managing the mapping and creating tf.Variable to store the model parameters (also known as model weights) for each feature ID. The model parameters will be learned through the model training process we'll go through later.

We'll do the similar trick to define the other categorical features:

race = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("race", hash\_bucket\_size=100)  
marital\_status = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("marital\_status", hash\_bucket\_size=100)  
relationship = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("relationship", hash\_bucket\_size=100)  
workclass = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("workclass", hash\_bucket\_size=100)  
occupation = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("occupation", hash\_bucket\_size=1000)  
native\_country = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("native\_country", hash\_bucket\_size=1000)

### Base Continuous Feature Columns

Similarly, we can define a RealValuedColumn for each continuous feature column that we want to use in the model:

age = tf.contrib.layers.real\_valued\_column("age")  
education\_num = tf.contrib.layers.real\_valued\_column("education\_num")  
capital\_gain = tf.contrib.layers.real\_valued\_column("capital\_gain")  
capital\_loss = tf.contrib.layers.real\_valued\_column("capital\_loss")  
hours\_per\_week = tf.contrib.layers.real\_valued\_column("hours\_per\_week")

### Making Continuous Features Categorical through Bucketization

Sometimes the relationship between a continuous feature and the label is not linear. As an hypothetical example, a person's income may grow with age in the early stage of one's career, then the growth may slow at some point, and finally the income decreases after retirement. In this scenario, using the raw age as a real-valued feature column might not be a good choice because the model can only learn one of the three cases:

1. Income always increases at some rate as age grows (positive correlation),
2. Income always decreases at some rate as age grows (negative correlation), or
3. Income stays the same no matter at what age (no correlation)

If we want to learn the fine-grained correlation between income and each age group separately, we can leverage **bucketization**. Bucketization is a process of dividing the entire range of a continuous feature into a set of consecutive bins/buckets, and then converting the original numerical feature into a bucket ID (as a categorical feature) depending on which bucket that value falls into. So, we can define a bucketized\_column over age as:

age\_buckets = tf.contrib.layers.bucketized\_column(age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])

where the boundaries is a list of bucket boundaries. In this case, there are 10 boundaries, resulting in 11 age group buckets (from age 17 and below, 18-24, 25-29, ..., to 65 and over).

### Intersecting Multiple Columns with CrossedColumn

Using each base feature column separately may not be enough to explain the data. For example, the correlation between education and the label (earning > 50,000 dollars) may be different for different occupations. Therefore, if we only learn a single model weight for education="Bachelors" and education="Masters", we won't be able to capture every single education-occupation combination (e.g. distinguishing between education="Bachelors" AND occupation="Exec-managerial" and education="Bachelors" AND occupation="Craft-repair"). To learn the differences between different feature combinations, we can add **crossed feature columns** to the model.

education\_x\_occupation = tf.contrib.layers.crossed\_column([education, occupation], hash\_bucket\_size=int(1e4))

We can also create a CrossedColumn over more than two columns. Each constituent column can be either a base feature column that is categorical (SparseColumn), a bucketized real-valued feature column (BucketizedColumn), or even another CrossColumn. Here's an example:

age\_buckets\_x\_education\_x\_occupation = tf.contrib.layers.crossed\_column(  
  [age\_buckets, education, occupation], hash\_bucket\_size=int(1e6))

## Defining The Logistic Regression Model

After processing the input data and defining all the feature columns, we're now ready to put them all together and build a Logistic Regression model. In the previous section we've seen several types of base and derived feature columns, including:

* SparseColumn
* RealValuedColumn
* BucketizedColumn
* CrossedColumn

All of these are subclasses of the abstract FeatureColumn class, and can be added to the feature\_columnsfield of a model:

model\_dir = tempfile.mkdtemp()  
m = tf.contrib.learn.LinearClassifier(feature\_columns=[  
  gender, native\_country, education, occupation, workclass, marital\_status, race,  
  age\_buckets, education\_x\_occupation, age\_buckets\_x\_education\_x\_occupation],  
  model\_dir=model\_dir)

The model also automatically learns a bias term, which controls the prediction one would make without observing any features (see the section "How Logistic Regression Works" for more explanations). The learned model files will be stored in model\_dir.

## Training and Evaluating Our Model

After adding all the features to the model, now let's look at how to actually train the model. Training a model is just a one-liner using the TF.Learn API:

m.fit(input\_fn=train\_input\_fn, steps=200)

After the model is trained, we can evaluate how good our model is at predicting the labels of the holdout data:

results = m.evaluate(input\_fn=eval\_input\_fn, steps=1)  
for key in sorted(results):  
    print("%s: %s" % (key, results[key]))

The first line of the output should be something like accuracy: 0.83557522, which means the accuracy is 83.6%. Feel free to try more features and transformations and see if you can do even better!

If you'd like to see a working end-to-end example, you can download our [example code](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/learn/wide_n_deep_tutorial.py). and set the model\_typeflag to wide.

## Adding Regularization to Prevent Overfitting

Regularization is a technique used to avoid **overfitting**. Overfitting happens when your model does well on the data it is trained on, but worse on test data that the model has not seen before, such as live traffic. Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observed training data. Regularization allows for you to control your model's complexity and makes the model more generalizable to unseen data.

In the Linear Model library, you can add L1 and L2 regularizations to the model as:

m = tf.contrib.learn.LinearClassifier(feature\_columns=[  
  gender, native\_country, education, occupation, workclass, marital\_status, race,  
  age\_buckets, education\_x\_occupation, age\_buckets\_x\_education\_x\_occupation],  
  optimizer=tf.train.FtrlOptimizer(  
    learning\_rate=0.1,  
    l1\_regularization\_strength=1.0,  
    l2\_regularization\_strength=1.0),  
  model\_dir=model\_dir)

One important difference between L1 and L2 regularization is that L1 regularization tends to make model weights stay at zero, creating sparser models, whereas L2 regularization also tries to make the model weights closer to zero but not necessarily zero. Therefore, if you increase the strength of L1 regularization, you will have a smaller model size because many of the model weights will be zero. This is often desirable when the feature space is very large but sparse, and when there are resource constraints that prevent you from serving a model that is too large.

In practice, you should try various combinations of L1, L2 regularization strengths and find the best parameters that best control overfitting and give you a desirable model size.

## How Logistic Regression Works

Finally, let's take a minute to talk about what the Logistic Regression model actually looks like in case you're not already familiar with it. We'll denote the label as Y, and the set of observed features as a feature vector x=[x1,x2,...,xd]. We define Y=1 if an individual earned > 50,000 dollars and Y=0 otherwise. In Logistic Regression, the probability of the label being positive (Y=1) given the features x is given as:

P(Y=1|x)=11+exp⁡(−(wTx+b))

where w=[w1,w2,...,wd] are the model weights for the features x=[x1,x2,...,xd]. b is a constant that is often called the **bias** of the model. The equation consists of two parts—A linear model and a logistic function:

* **Linear Model**: First, we can see that wTx+b=b+w1x1+...+wdxd is a linear model where the output is a linear function of the input features x. The bias b is the prediction one would make without observing any features. The model weight wi reflects how the feature xi is correlated with the positive label. If xi is positively correlated with the positive label, the weight wi increases, and the probability P(Y=1|x) will be closer to 1. On the other hand, if xi is negatively correlated with the positive label, then the weight wi decreases and the probability P(Y=1|x) will be closer to 0.
* **Logistic Function**: Second, we can see that there's a logistic function (also known as the sigmoid function) S(t)=1/(1+exp⁡(−t)) being applied to the linear model. The logistic function is used to convert the output of the linear model wTx+b from any real number into the range of [0,1], which can be interpreted as a probability.

Model training is an optimization problem: The goal is to find a set of model weights (i.e. model parameters) to minimize a **loss function** defined over the training data, such as logistic loss for Logistic Regression models. The loss function measures the discrepancy between the ground-truth label and the model's prediction. If the prediction is very close to the ground-truth label, the loss value will be low; if the prediction is very far from the label, then the loss value would be high.

## Learn Deeper

If you're interested in learning more, check out our [Wide & Deep Learning Tutorial](https://www.tensorflow.org/tutorials/wide_and_deep) where we'll show you how to combine the strengths of linear models and deep neural networks by jointly training them using the TF.Learn API.

# TensorFlow Wide & Deep Learning Tutorial

In the previous [TensorFlow Linear Model Tutorial](https://www.tensorflow.org/tutorials/wide), we trained a logistic regression model to predict the probability that the individual has an annual income of over 50,000 dollars using the [Census Income Dataset](https://archive.ics.uci.edu/ml/datasets/Census+Income). TensorFlow is great for training deep neural networks too, and you might be thinking which one you should choose—Well, why not both? Would it be possible to combine the strengths of both in one model?

In this tutorial, we'll introduce how to use the TF.Learn API to jointly train a wide linear model and a deep feed-forward neural network. This approach combines the strengths of memorization and generalization. It's useful for generic large-scale regression and classification problems with sparse input features (e.g., categorical features with a large number of possible feature values). If you're interested in learning more about how Wide & Deep Learning works, please check out our [research paper](http://arxiv.org/abs/1606.07792).

The figure above shows a comparison of a wide model (logistic regression with sparse features and transformations), a deep model (feed-forward neural network with an embedding layer and several hidden layers), and a Wide & Deep model (joint training of both). At a high level, there are only 3 steps to configure a wide, deep, or Wide & Deep model using the TF.Learn API:

1. Select features for the wide part: Choose the sparse base columns and crossed columns you want to use.
2. Select features for the deep part: Choose the continuous columns, the embedding dimension for each categorical column, and the hidden layer sizes.
3. Put them all together in a Wide & Deep model (DNNLinearCombinedClassifier).

And that's it! Let's go through a simple example.

## Setup

To try the code for this tutorial:

1. [Install TensorFlow](https://www.tensorflow.org/install/index) if you haven't already.
2. Download [the tutorial code](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/learn/wide_n_deep_tutorial.py).
3. Install the pandas data analysis library. tf.learn doesn't require pandas, but it does support it, and this tutorial uses pandas. To install pandas:

a. Get pip:

# Ubuntu/Linux 64-bit  
$ sudo apt-get install python-pip python-dev  
  
# Mac OS X  
$ sudo easy\_install pip  
$ sudo easy\_install --upgrade six

b. Use pip to install pandas:

$ sudo pip install pandas

If you have trouble installing pandas, consult the [instructions](http://pandas.pydata.org/pandas-docs/stable/install.html) on the pandas site.

1. Execute the tutorial code with the following command to train the linear model described in this tutorial:

$ python wide\_n\_deep\_tutorial.py --model\_type=wide\_n\_deep

Read on to find out how this code builds its linear model.

## Define Base Feature Columns

First, let's define the base categorical and continuous feature columns that we'll use. These base columns will be the building blocks used by both the wide part and the deep part of the model.

import tensorflow as tf  
  
# Categorical base columns.  
gender = tf.contrib.layers.sparse\_column\_with\_keys(column\_name="gender", keys=["Female", "Male"])  
race = tf.contrib.layers.sparse\_column\_with\_keys(column\_name="race", keys=[  
  "Amer-Indian-Eskimo", "Asian-Pac-Islander", "Black", "Other", "White"])  
education = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("education", hash\_bucket\_size=1000)  
relationship = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("relationship", hash\_bucket\_size=100)  
workclass = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("workclass", hash\_bucket\_size=100)  
occupation = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("occupation", hash\_bucket\_size=1000)  
native\_country = tf.contrib.layers.sparse\_column\_with\_hash\_bucket("native\_country", hash\_bucket\_size=1000)  
  
# Continuous base columns.  
age = tf.contrib.layers.real\_valued\_column("age")  
age\_buckets = tf.contrib.layers.bucketized\_column(age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])  
education\_num = tf.contrib.layers.real\_valued\_column("education\_num")  
capital\_gain = tf.contrib.layers.real\_valued\_column("capital\_gain")  
capital\_loss = tf.contrib.layers.real\_valued\_column("capital\_loss")  
hours\_per\_week = tf.contrib.layers.real\_valued\_column("hours\_per\_week")

## The Wide Model: Linear Model with Crossed Feature Columns

The wide model is a linear model with a wide set of sparse and crossed feature columns:

wide\_columns = [  
  gender, native\_country, education, occupation, workclass, relationship, age\_buckets,  
  tf.contrib.layers.crossed\_column([education, occupation], hash\_bucket\_size=int(1e4)),  
  tf.contrib.layers.crossed\_column([native\_country, occupation], hash\_bucket\_size=int(1e4)),  
  tf.contrib.layers.crossed\_column([age\_buckets, education, occupation], hash\_bucket\_size=int(1e6))]

Wide models with crossed feature columns can memorize sparse interactions between features effectively. That being said, one limitation of crossed feature columns is that they do not generalize to feature combinations that have not appeared in the training data. Let's add a deep model with embeddings to fix that.

## The Deep Model: Neural Network with Embeddings

The deep model is a feed-forward neural network, as shown in the previous figure. Each of the sparse, high-dimensional categorical features are first converted into a low-dimensional and dense real-valued vector, often referred to as an embedding vector. These low-dimensional dense embedding vectors are concatenated with the continuous features, and then fed into the hidden layers of a neural network in the forward pass. The embedding values are initialized randomly, and are trained along with all other model parameters to minimize the training loss. If you're interested in learning more about embeddings, check out the TensorFlow tutorial on [Vector Representations of Words](https://www.tensorflow.org/versions/r0.9/tutorials/word2vec/index.html), or [Word Embedding](https://en.wikipedia.org/wiki/Word_embedding) on Wikipedia.

We'll configure the embeddings for the categorical columns using embedding\_column, and concatenate them with the continuous columns:

deep\_columns = [  
  tf.contrib.layers.embedding\_column(workclass, dimension=8),  
  tf.contrib.layers.embedding\_column(education, dimension=8),  
  tf.contrib.layers.embedding\_column(gender, dimension=8),  
  tf.contrib.layers.embedding\_column(relationship, dimension=8),  
  tf.contrib.layers.embedding\_column(native\_country, dimension=8),  
  tf.contrib.layers.embedding\_column(occupation, dimension=8),  
  age, education\_num, capital\_gain, capital\_loss, hours\_per\_week]

The higher the dimension of the embedding is, the more degrees of freedom the model will have to learn the representations of the features. For simplicity, we set the dimension to 8 for all feature columns here. Empirically, a more informed decision for the number of dimensions is to start with a value on the order of log2⁡(n) or kn4, where n is the number of unique features in a feature column and k is a small constant (usually smaller than 10).

Through dense embeddings, deep models can generalize better and make predictions on feature pairs that were previously unseen in the training data. However, it is difficult to learn effective low-dimensional representations for feature columns when the underlying interaction matrix between two feature columns is sparse and high-rank. In such cases, the interaction between most feature pairs should be zero except a few, but dense embeddings will lead to nonzero predictions for all feature pairs, and thus can over-generalize. On the other hand, linear models with crossed features can memorize these “exception rules” effectively with fewer model parameters.

Now, let's see how to jointly train wide and deep models and allow them to complement each other’s strengths and weaknesses.

## Combining Wide and Deep Models into One

The wide models and deep models are combined by summing up their final output log odds as the prediction, then feeding the prediction to a logistic loss function. All the graph definition and variable allocations have already been handled for you under the hood, so you simply need to create a DNNLinearCombinedClassifier:

import tempfile  
model\_dir = tempfile.mkdtemp()  
m = tf.contrib.learn.DNNLinearCombinedClassifier(  
    model\_dir=model\_dir,  
    linear\_feature\_columns=wide\_columns,  
    dnn\_feature\_columns=deep\_columns,  
    dnn\_hidden\_units=[100, 50])

## Training and Evaluating The Model

Before we train the model, let's read in the Census dataset as we did in the [TensorFlow Linear Model tutorial](https://www.tensorflow.org/tutorials/wide). The code for input data processing is provided here again for your convenience:

import pandas as pd  
import urllib  
  
# Define the column names for the data sets.  
COLUMNS = ["age", "workclass", "fnlwgt", "education", "education\_num",  
  "marital\_status", "occupation", "relationship", "race", "gender",  
  "capital\_gain", "capital\_loss", "hours\_per\_week", "native\_country", "income\_bracket"]  
LABEL\_COLUMN = 'label'  
CATEGORICAL\_COLUMNS = ["workclass", "education", "marital\_status", "occupation",  
                       "relationship", "race", "gender", "native\_country"]  
CONTINUOUS\_COLUMNS = ["age", "education\_num", "capital\_gain", "capital\_loss",  
                      "hours\_per\_week"]  
  
# Download the training and test data to temporary files.  
# Alternatively, you can download them yourself and change train\_file and  
# test\_file to your own paths.  
train\_file = tempfile.NamedTemporaryFile()  
test\_file = tempfile.NamedTemporaryFile()  
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.data", train\_file.name)  
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.test", test\_file.name)  
  
# Read the training and test data sets into Pandas dataframe.  
df\_train = pd.read\_csv(train\_file, names=COLUMNS, skipinitialspace=True)  
df\_test = pd.read\_csv(test\_file, names=COLUMNS, skipinitialspace=True, skiprows=1)  
df\_train[LABEL\_COLUMN] = (df\_train['income\_bracket'].apply(lambda x: '>50K' in x)).astype(int)  
df\_test[LABEL\_COLUMN] = (df\_test['income\_bracket'].apply(lambda x: '>50K' in x)).astype(int)  
  
def input\_fn(df):  
  # Creates a dictionary mapping from each continuous feature column name (k) to  
  # the values of that column stored in a constant Tensor.  
  continuous\_cols = {k: tf.constant(df[k].values)  
                     for k in CONTINUOUS\_COLUMNS}  
  # Creates a dictionary mapping from each categorical feature column name (k)  
  # to the values of that column stored in a tf.SparseTensor.  
  categorical\_cols = {k: tf.SparseTensor(  
      indices=[[i, 0] for i in range(df[k].size)],  
      values=df[k].values,  
      dense\_shape=[df[k].size, 1])  
                      for k in CATEGORICAL\_COLUMNS}  
  # Merges the two dictionaries into one.  
  feature\_cols = dict(continuous\_cols.items() + categorical\_cols.items())  
  # Converts the label column into a constant Tensor.  
  label = tf.constant(df[LABEL\_COLUMN].values)  
  # Returns the feature columns and the label.  
  return feature\_cols, label  
  
def train\_input\_fn():  
  return input\_fn(df\_train)  
  
def eval\_input\_fn():  
  return input\_fn(df\_test)

After reading in the data, you can train and evaluate the model:

m.fit(input\_fn=train\_input\_fn, steps=200)  
results = m.evaluate(input\_fn=eval\_input\_fn, steps=1)  
for key in sorted(results):  
    print("%s: %s" % (key, results[key]))

The first line of the output should be something like accuracy: 0.84429705. We can see that the accuracy was improved from about 83.6% using a wide-only linear model to about 84.4% using a Wide & Deep model. If you'd like to see a working end-to-end example, you can download our [example code](https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/learn/wide_n_deep_tutorial.py).

Note that this tutorial is just a quick example on a small dataset to get you familiar with the API. Wide & Deep Learning will be even more powerful if you try it on a large dataset with many sparse feature columns that have a large number of possible feature values. Again, feel free to take a look at our [research paper](http://arxiv.org/abs/1606.07792) for more ideas about how to apply Wide & Deep Learning in real-world large-scale machine learning problems.

# Mandelbrot Set

Visualizing the [Mandelbrot set](https://en.wikipedia.org/wiki/Mandelbrot_set) doesn't have anything to do with machine learning, but it makes for a fun example of how one can use TensorFlow for general mathematics. This is actually a pretty naive implementation of the visualization, but it makes the point. (We may end up providing a more elaborate implementation down the line to produce more truly beautiful images.)

## Basic Setup

We'll need a few imports to get started.

# Import libraries for simulation  
import tensorflow as tf  
import numpy as np  
  
# Imports for visualization  
import PIL.Image  
from io import BytesIO  
from IPython.display import Image, display

Now we'll define a function to actually display the image once we have iteration counts.

def DisplayFractal(a, fmt='jpeg'):  
  """Display an array of iteration counts as a  
     colorful picture of a fractal."""  
  a\_cyclic = (6.28\*a/20.0).reshape(list(a.shape)+[1])  
  img = np.concatenate([10+20\*np.cos(a\_cyclic),  
                        30+50\*np.sin(a\_cyclic),  
                        155-80\*np.cos(a\_cyclic)], 2)  
  img[a==a.max()] = 0  
  a = img  
  a = np.uint8(np.clip(a, 0, 255))  
  f = BytesIO()  
  PIL.Image.fromarray(a).save(f, fmt)  
  display(Image(data=f.getvalue()))

## Session and Variable Initialization

For playing around like this, we often use an interactive session, but a regular session would work as well.

sess = tf.InteractiveSession()

It's handy that we can freely mix NumPy and TensorFlow.

# Use NumPy to create a 2D array of complex numbers  
  
Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]  
Z = X+1j\*Y

Now we define and initialize TensorFlow tensors.

xs = tf.constant(Z.astype(np.complex64))  
zs = tf.Variable(xs)  
ns = tf.Variable(tf.zeros\_like(xs, tf.float32))

TensorFlow requires that you explicitly initialize variables before using them.

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

## Defining and Running the Computation

Now we specify more of the computation...

# Compute the new values of z: z^2 + x  
zs\_ = zs\*zs + xs  
  
# Have we diverged with this new value?  
not\_diverged = tf.abs(zs\_) < 4  
  
# Operation to update the zs and the iteration count.  
#  
# Note: We keep computing zs after they diverge! This  
#       is very wasteful! There are better, if a little  
#       less simple, ways to do this.  
#  
step = tf.group(  
  zs.assign(zs\_),  
  ns.assign\_add(tf.cast(not\_diverged, tf.float32))  
  )

... and run it for a couple hundred steps

for i in range(200): step.run()

Let's see what we've got.

DisplayFractal(ns.eval())

Not bad!

# Partial Differential Equations

TensorFlow isn't just for machine learning. Here we give a (somewhat pedestrian) example of using TensorFlow for simulating the behavior of a [partial differential equation](https://en.wikipedia.org/wiki/Partial_differential_equation). We'll simulate the surface of square pond as a few raindrops land on it.

## Basic Setup

A few imports we'll need.

#Import libraries for simulation  
import tensorflow as tf  
import numpy as np  
  
#Imports for visualization  
import PIL.Image  
from io import BytesIO  
from IPython.display import clear\_output, Image, display

A function for displaying the state of the pond's surface as an image.

def DisplayArray(a, fmt='jpeg', rng=[0,1]):  
  """Display an array as a picture."""  
  a = (a - rng[0])/float(rng[1] - rng[0])\*255  
  a = np.uint8(np.clip(a, 0, 255))  
  f = BytesIO()  
  PIL.Image.fromarray(a).save(f, fmt)  
  clear\_output(wait = True)  
  display(Image(data=f.getvalue()))

Here we start an interactive TensorFlow session for convenience in playing around. A regular session would work as well if we were doing this in an executable .py file.

sess = tf.InteractiveSession()

## Computational Convenience Functions

def make\_kernel(a):  
  """Transform a 2D array into a convolution kernel"""  
  a = np.asarray(a)  
  a = a.reshape(list(a.shape) + [1,1])  
  return tf.constant(a, dtype=1)  
  
def simple\_conv(x, k):  
  """A simplified 2D convolution operation"""  
  x = tf.expand\_dims(tf.expand\_dims(x, 0), -1)  
  y = tf.nn.depthwise\_conv2d(x, k, [1, 1, 1, 1], padding='SAME')  
  return y[0, :, :, 0]  
  
def laplace(x):  
  """Compute the 2D laplacian of an array"""  
  laplace\_k = make\_kernel([[0.5, 1.0, 0.5],  
                           [1.0, -6., 1.0],  
                           [0.5, 1.0, 0.5]])  
  return simple\_conv(x, laplace\_k)

## Define the PDE

Our pond is a perfect 500 x 500 square, as is the case for most ponds found in nature.

N = 500

Here we create our pond and hit it with some rain drops.

# Initial Conditions -- some rain drops hit a pond  
  
# Set everything to zero  
u\_init = np.zeros([N, N], dtype=np.float32)  
ut\_init = np.zeros([N, N], dtype=np.float32)  
  
# Some rain drops hit a pond at random points  
for n in range(40):  
  a,b = np.random.randint(0, N, 2)  
  u\_init[a,b] = np.random.uniform()  
  
DisplayArray(u\_init, rng=[-0.1, 0.1])

Now let's specify the details of the differential equation.

# Parameters:  
# eps -- time resolution  
# damping -- wave damping  
eps = tf.placeholder(tf.float32, shape=())  
damping = tf.placeholder(tf.float32, shape=())  
  
# Create variables for simulation state  
U  = tf.Variable(u\_init)  
Ut = tf.Variable(ut\_init)  
  
# Discretized PDE update rules  
U\_ = U + eps \* Ut  
Ut\_ = Ut + eps \* (laplace(U) - damping \* Ut)  
  
# Operation to update the state  
step = tf.group(  
  U.assign(U\_),  
  Ut.assign(Ut\_))

## Run The Simulation

This is where it gets fun -- running time forward with a simple for loop.

# Initialize state to initial conditions  
tf.global\_variables\_initializer().run()  
  
# Run 1000 steps of PDE  
for i in range(1000):  
  # Step simulation  
  step.run({eps: 0.03, damping: 0.04})  
  DisplayArray(U.eval(), rng=[-0.1, 0.1])

Look! Ripples!