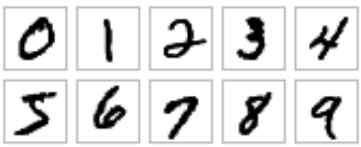
Tutorials

A Guide to TF Layers: Building a Convolutional Neural Network

The TensorFlow `layers` module 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 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

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.

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 to the output to introduce nonlinearities into the model.
- **Pooling layers**, which downsample the image data 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 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.

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). `cnn_mnist.py` takes MNIST feature data, labels, and model mode (`TRAIN`, `EVAL`, `PREDICT`) 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["x"], [-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 == tf.estimator.ModeKeys.TRAIN)

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

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by
the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }

  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

The following sections (with headings corresponding to each code block above) dive deeper into the `tf.layers` code 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`, 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".

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["x"], [-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["x"]`, 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["x"]` will contain 3,920 values (one value for each pixel in each image), and `input_layer` will have a shape of `[5, 28, 28, 1]`. Similarly, if we feed examples in batches of 100, `features["x"]` 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 input 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`.

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 `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 (`pool2` height) * 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 == tf.estimator.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]`.

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` function:

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

The `input` argument specifies the tensor from which to extract maximum values—here `logits`. The `axis` argument 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`:

```
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".

We compile our predictions in a dict, and return an `EstimatorSpec` object:

```
predictions = {
    "classes": tf.argmax(input=logits, axis=1),
    "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
  return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)
```

Calculate Loss

For both training and evaluation, we need to define a loss function that measures how closely the model's predictions match the target classes. For multiclass classification problems like MNIST, 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:

```
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:

```
[[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` 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. We'll use a learning rate of 0.001 and stochastic gradient descent as the optimization algorithm:

```
if mode == tf.estimator.ModeKeys.TRAIN:
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
  train_op = optimizer.minimize(
      loss=loss,
      global_step=tf.train.get_global_step())
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
train_op=train_op)
```

> **Note:** For a more in-depth look at configuring training ops for Estimator model functions, see @{$get_started/custom_estimators#defining-the-training-op-for-the-model$"Defining the training op for the model"} in the @{$get_started/custom_estimators$"Creating Estimations in tf.estimator"} tutorial.

Add evaluation metrics

To add accuracy metric in our model, we define `eval_metric_ops` dict in EVAL mode as follows:

```
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

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 = tf.contrib.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 in `train_data` and `train_labels`, respectively. Similarly, we store the evaluation feature data (10,000 images) and evaluation labels in `eval_data` and `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 = tf.estimator.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 prediction; we pass it the `cnn_model_fn` we created in "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.estimator."

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` to create a `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, our `probabilities`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) or to enable the @{$debugger$TensorFlow Debugger (tfdbg)}.

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 creating `train_input_fn` ans calling `train()` on `mnist_classifier`. Add the following to `main()`:

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=20000,
    hooks=[logging_hook])
```

In the `numpy_input_fn` call, we pass the training feature data and labels to `x` (as a dict) 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).`num_epochs=None` means that the model will train until the specified number of steps is reached. We also set `shuffle=True` to shuffle the training data. In the `train` call, we set `steps=20000` (which means the model will train for 20,000 steps total). We pass our `logging_hook` to the `hooks` argument, 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. We call the `evaluate` method, which evaluates the metrics we specified in `eval_metric_ops` argument in the `model_fn`. Add the following to `main()`:

```
# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

To create `eval_input_fn`, we set `num_epochs=1`, so that the model evaluates the metrics over one epoch of data and returns the result. We also set `shuffle=False` to iterate through the data sequentially.

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 `train()`, 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.estimator provides an introduction to the TensorFlow Estimator API. It walks through configuring an Estimator, writing a model function, calculating loss, and defining a training op.
- Convolutional Neural Networks walks through how to build a MNIST CNN classification model *without estimators*using lower-level TensorFlow operations.

Image Recognition

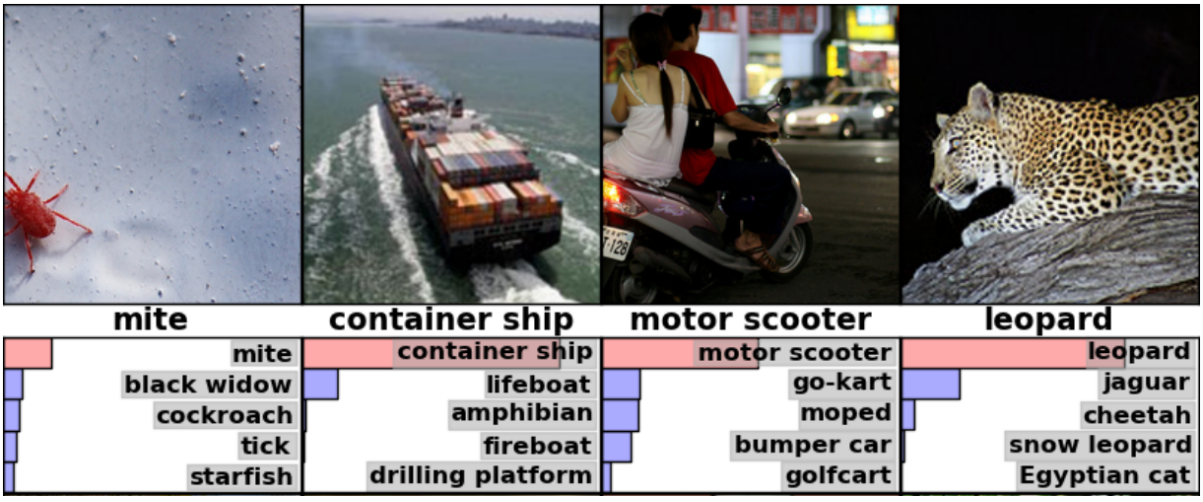
ContentsUsage with Python APIUsage with the C++ APIResources for Learning More

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 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 -- an academic benchmark for computer vision. Successive models continue to show improvements, each time achieving a new state-of-the-art result: QuocNet, AlexNet, Inception (GoogLeNet), BN-Inception-v2. 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.

Inception-v3 is trained for the ImageNet 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, like "Zebra", "Dalmatian", and "Dishwasher". For example, here are the results from AlexNet 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 achieved by setting a top-5 error rate of 15.3% on the 2012 validation data set; Inception (GoogLeNet) achieved 6.67%; BN-Inception-v2 achieved 4.9%; Inception-v3 reaches 3.46%.

> How well do humans do on ImageNet Challenge? There's a blog post 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. You'll learn how to classify images into 1000 classes 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 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 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 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, 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` 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_mean` from 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()` 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` 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()` 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, in this case a 2048 dimensional vector. There's a guide to doing this in the how-to section.

Resources for Learning More

To learn about neural networks in general, Michael Nielsen's free online book is an excellent resource. For convolutional neural networks in particular, Chris Olah has some nice blog posts, and Michael Nielsen's book has a great chaptercovering them.

To find out more about implementing convolutional neural networks, you can jump to the TensorFlow deep convolutional networks tutorial, or start a bit more gently with our MNIST starter tutorial. 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.

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.

Note: A version of this tutorial is also available as a codelab.

Before you start, you must install tensorflow.

Training on Flowers

Image by Kelly Sikkema

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 clone the tensorflow repository using the following command (these examples are not included in the installation):

```
git clone https://github.com/tensorflow/tensorflow
```

Then checkout the version of the tensorflow repository matching your installation and this tutorial as follows:

```
cd tensorflow
git checkout {version}
```

In the simplest cases the retrainer can then be run like this:

```
python tensorflow/examples/image_retraining/retrain.py --image_dir
~/flower_photos
```

The script has many other options. You can get a full listing with:

```
python tensorflow/examples/image_retraining/retrain.py -h
```

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/bottleneck` directory, 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's GitHub 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 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 run the label_image example with your retrained graphs:

```
python tensorflow/examples/label_image/label_image.py \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--input_layer=Mul \
--output_layer=final_result \
--input_mean=128 --input_std=128 \
--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.

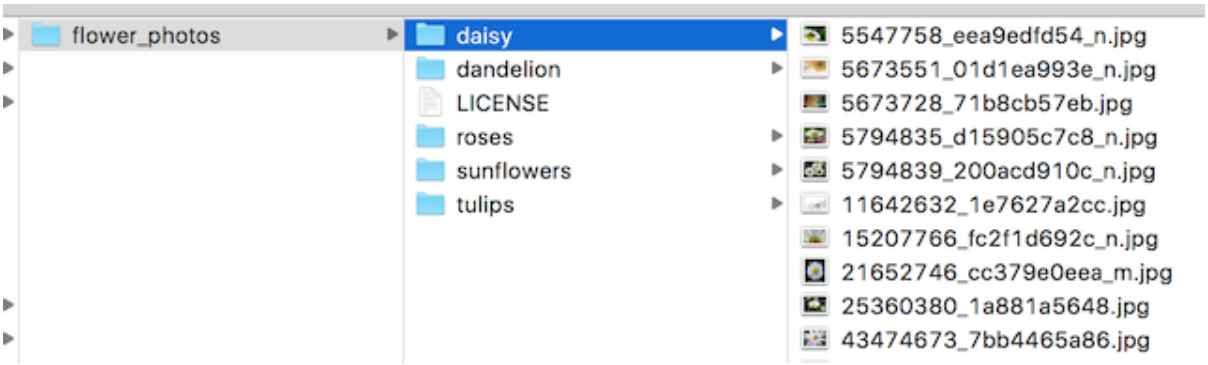
If you'd like to use the retrained model in your own Python program, then the above `label_image` script is a reasonable starting point. The `label_image` directory also contains C++ code which you can use as a template to integrate tensorflow with your own applications.

If you find the default Inception v3 model is too large or slow for your application, take a look at the Other Model Architectures section below for options to speed up and slim down your network.

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.

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_right` will 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.

Other Model Architectures

By default the script uses a pretrained version of the Inception v3 model architecture. This is a good place to start because it provides high accuracy results, but if you intend to deploy your model on mobile devices or other resource-constrained environments you may want to trade off a little accuracy for much smaller file sizes or faster speeds. To help with that, the retrain.py script supports 32 different variations on the Mobilenet architecture.

These are a little less precise than Inception v3, but can result in far smaller file sizes (down to less than a megabyte) and can be many times faster to run. To train with one of these models, pass in the `--architecture` flag, for example:

```
python tensorflow/examples/image_retraining/retrain.py \
    --image_dir ~/flower_photos --architecture
mobilenet_0.25_128_quantized
```

This will create a 941KB model file in `/tmp/output_graph.pb`, with 25% of the parameters of the full Mobilenet, taking 128x128 sized input images, and with its weights quantized down to eight bits on disk. You can choose '1.0', '0.75', '0.50', or '0.25' to control the number of weight parameters, and so the file size (and to some extent the speed), '224', '192', '160', or '128' for the input image size, with smaller sizes giving faster speeds, and an optional '_quantized' at the end to indicate whether the file should contain 8-bit or 32-bit float weights.

The speed and size advantages come at a loss to accuracy of course, but for many purposes this isn't critical. They can also be somewhat offset with improved training data. For example, training with distortions allows me to get above 80% accuracy on the flower data set even with the 0.25/128/quantized graph above.

If you're going to be using the Mobilenet models in label_image or your own programs, you'll need to feed in an image of the specified size converted to a float range into the 'input' tensor. Typically 24-bit images are in the range [0,255], and you must convert them to the [-1,1] float range expected by the model with the formula `(image - 128.)/128.`.

The default arguments for the `label_image` script are set for Inception V3. To use it with a MobileNet, specify the above normalization parameters as `input_mean` and `input_std` on the command line. You also must specify the image size that your model expects, as follows:

```
python tensorflow/examples/label_image/label_image.py \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--input_layer=input \
--output_layer=final_result:0 \
--input_height=224 --input_width=224 \
--input_mean=128 --input_std=128 \
--image=$HOME/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

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 and a Tech Report by Alex Krizhevsky.

Goals

The goal of this tutorial is to build a relatively small 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 (wiki), rectified linear activations (wiki), max pooling (wiki) and local response normalization (Chapter 3.3 in AlexNet paper).
- Visualization of network activities during training, including input images, losses and distributions of activations and gradients.
- Routines for calculating the moving average of learned parameters and using these averages during evaluation to boost predictive performance.
- Implementation of a learning rate schedule that systematically decrements over time.
- Prefetching queues for input data to isolate the model from disk latency and expensive image pre-processing.

We also provide a multi-GPU version 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, 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 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/`.

File	Purpose
`cifar10_input.py`	Reads the native CIFAR-10 binary file format.
`cifar10.py`	Builds the CIFAR-10 model.
`cifar10_train.py`	Trains a CIFAR-10 model on a CPU or GPU.
`cifar10_multi_gpu_train.py`	Trains a CIFAR-10 model on multiple GPUs.
`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`. 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:** `inputs()` and `distorted_inputs()` add operations that read and preprocess CIFAR images for evaluation and training, respectively.
2. **Model prediction:** `inference()` adds operations that perform inference, i.e. classification, on supplied images.
3. **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`. See Reading Data 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 for training.
- They are approximately whitened 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 the image from left to right.

- Randomly distort the image brightness.
- Randomly distort the image contrast.

Please see the Images page for the list of available distortions. We also attach an `tf.summary.image` to the images so that we may visualize them in TensorBoard. This is a good practice to verify that inputs are built correctly.

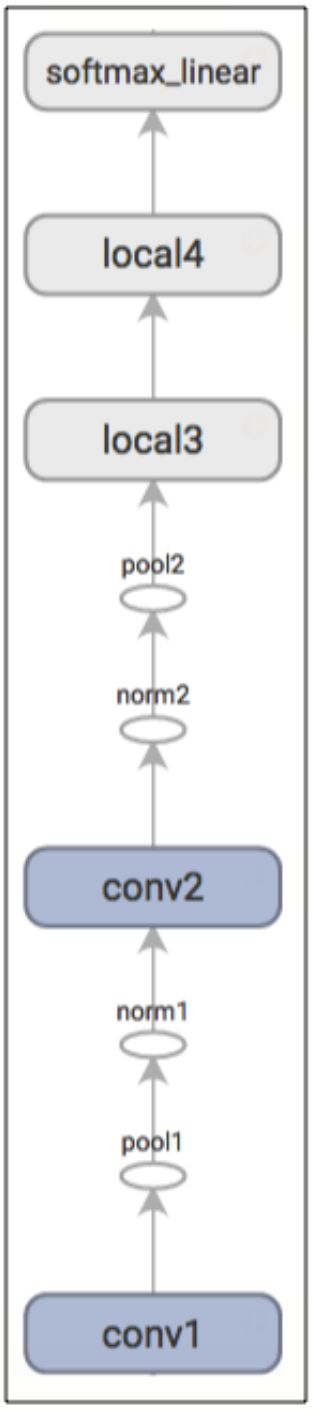
Reading images from disk and distorting them can use a non-trivial amount of processing time. To prevent these operations from slowing down training, we run them inside 16 separate threads which continuously fill a TensorFlowqueue.

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 and rectified linear activation.
`pool1`	max pooling.
`norm1`	local response normalization.
`conv2`	convolution and rectified linear activation.
`norm2`	local response normalization.
`pool2`	max pooling.
`local3`	fully connected layer with rectified linear activation.
`local4`	fully connected layer with rectified linear activation.
`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`.

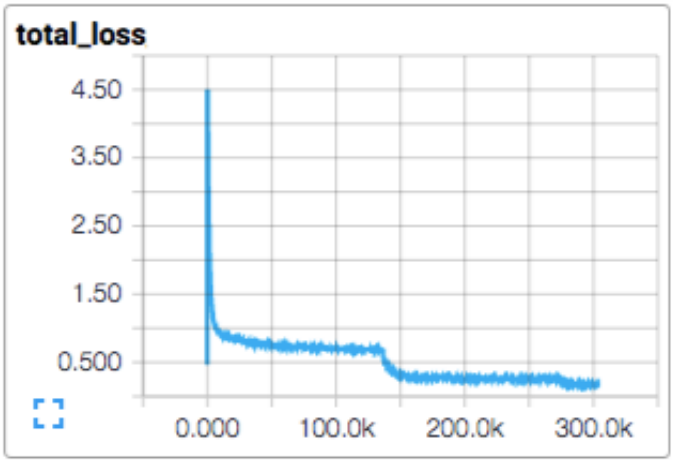
The `inputs()` and `inference()` functions provide all the components necessary to perform an evaluation of a model. We now shift our focus towards building operations for training a model.

> **EXERCISE:** The model architecture in `inference()` differs slightly from the CIFAR-10 model specified in cuda-convnet. 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.

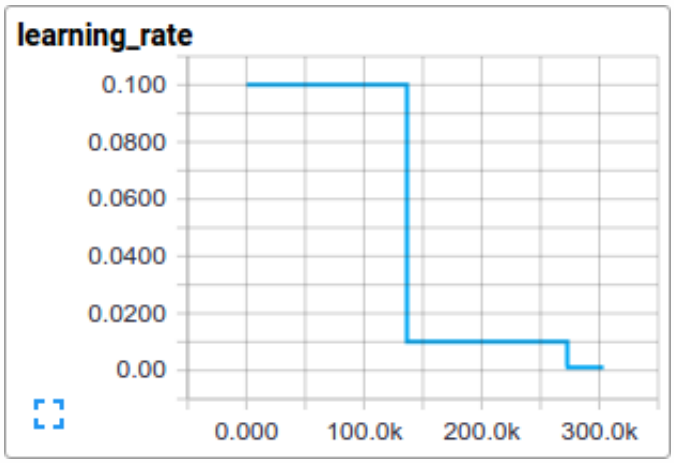
Model Training

The usual method for training a network to perform N-way classification is multinomial logistic regression, aka. *softmax regression*. Softmax regression applies a softmax nonlinearity to the output of the network and calculates the cross-entropy between the normalized predictions and the label index. For regularization, we also apply the usual weight decaylosses 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`:



We train the model using standard gradient descent algorithm (see Training for other methods) with a learning rate thatexponentially decays 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` 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 all model parameters in checkpoint files 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 Modelbelow).

If you followed the previous steps, then you have now started training a CIFAR-10 model. Congratulations!

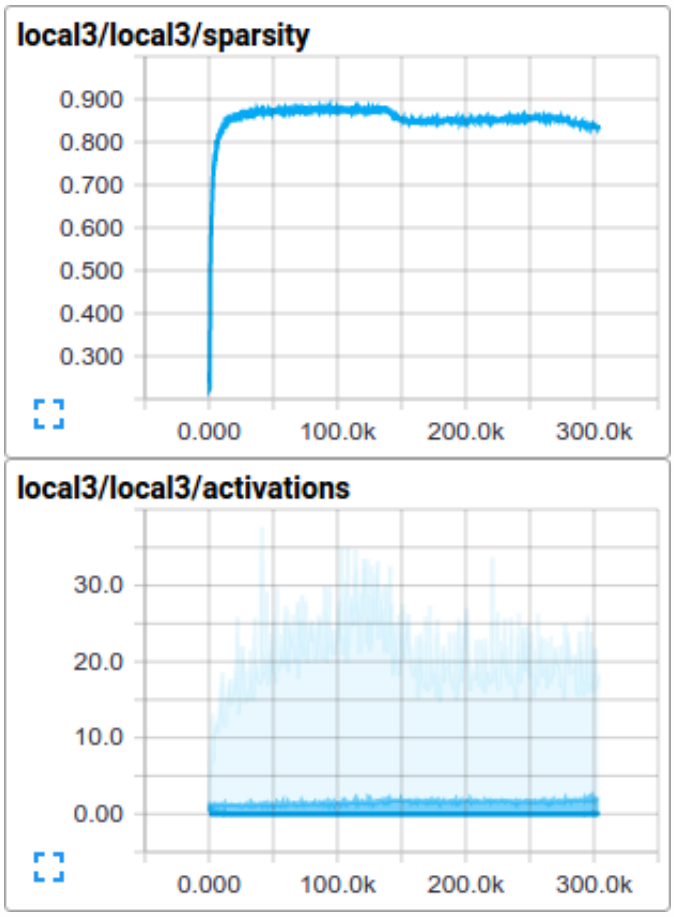
The terminal text returned from `cifar10_train.py` provides minimal insight into how the model is training. We want more insight into the model during training:

- 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 provides this functionality, displaying data exported periodically from `cifar10_train.py` via a`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` 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 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.

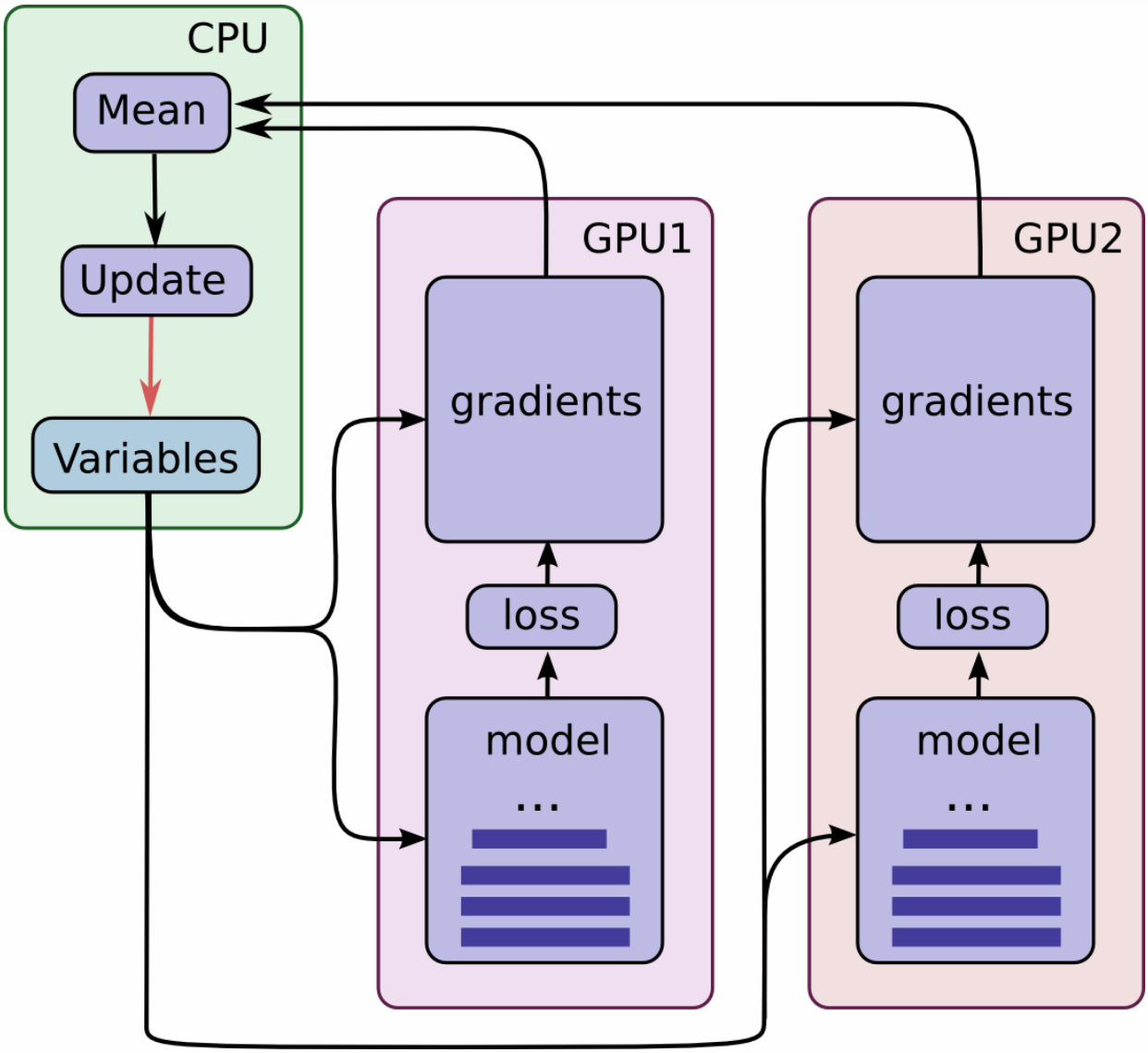
Training a model in a parallel, distributed fashion requires coordinating training processes. For what follows we term *model replica* to be one copy of a model training on a subset of data.

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` 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` specifies this. For instance, all operations in the first tower reside within `device('/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` in order to share them in a multi-GPU version. See how-to on Sharing Variables.

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! 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) 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.

Recurrent Neural Networks

Introduction

Take a look at this great article 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 (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.

For the purpose of this tutorial, we will reproduce the results from Zaremba et al., 2014 (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:

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.

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`. In this example, it is important to note that `current_batch_of_words` does not correspond to a "sentence" of words. Every word in a batch should correspond to a time t. TensorFlow will automatically sum the gradients of each batch for you.

For example:

```
 t=0  t=1    t=2  t=3     t=4
[The, brown, fox, is,     quick]
[The, red,   fox, jumped, high]

words_in_dataset[0] = [The, The]
words_in_dataset[1] = [brown, red]
words_in_dataset[2] = [fox, fox]
words_in_dataset[3] = [is, jumped]
words_in_dataset[4] = [quick, high]
batch_size = 2, time_steps = 5
```

The basic pseudocode is as follows:

```
words_in_dataset = tf.placeholder(tf.float32, [time_steps, batch_size,
num_features])
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
hidden_state = tf.zeros([batch_size, lstm.state_size])
current_state = tf.zeros([batch_size, lstm.state_size])
state = hidden_state, current_state
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) 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=1Nlnptargeti

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=1Nlnptargeti=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.)

Now, clone the TensorFlow models repo 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.

Neural Machine Translation (seq2seq) Tutorial

ContentsBackground on Neural Machine TranslationInstalling the TutorialTraining – How to build our first NMT systemEmbedding

Authors: Thang Luong, Eugene Brevdo, Rui Zhao (Google Research Blogpost, Github)

This version of the tutorial requires TensorFlow Nightly. For using the stable TensorFlow versions, please consider other branches such as tf-1.4.

If make use of this codebase for your research, please cite this.

Introduction

Sequence-to-sequence (seq2seq) models (Sutskever et al., 2014, Cho et al., 2014) have enjoyed great success in a variety of tasks such as machine translation, speech recognition, and text summarization. This tutorial gives readers a full understanding of seq2seq models and shows how to build a competitive seq2seq model from scratch. We focus on the task of Neural Machine Translation (NMT) which was the very first testbed for seq2seq models with wild success. The included code is lightweight, high-quality, production-ready, and incorporated with the latest research ideas. We achieve this goal by:

1. Using the recent decoder / attention wrapper API, TensorFlow 1.2 data iterator
2. Incorporating our strong expertise in building recurrent and seq2seq models
3. Providing tips and tricks for building the very best NMT models and replicating Google's NMT (GNMT) system.

We believe that it is important to provide benchmarks that people can easily replicate. As a result, we have provided full experimental results and pretrained on models on the following publicly available datasets:

1. *Small-scale*: English-Vietnamese parallel corpus of TED talks (133K sentence pairs) provided by the IWSLT Evaluation Campaign.
2. *Large-scale*: German-English parallel corpus (4.5M sentence pairs) provided by the WMT Evaluation Campaign.

We first build up some basic knowledge about seq2seq models for NMT, explaining how to build and train a vanilla NMT model. The second part will go into details of building a competitive NMT model with attention mechanism. We then discuss tips and tricks to build the best possible NMT models (both in speed and translation quality) such as TensorFlow best practices (batching, bucketing), bidirectional RNNs, beam search, as well as scaling up to multiple GPUs using GNMT attention.

Basic

Background on Neural Machine Translation

Back in the old days, traditional phrase-based translation systems performed their task by breaking up source sentences into multiple chunks and then translated them phrase-by-phrase. This led to disfluency in the translation outputs and was not quite like how we, humans, translate. We read the entire source sentence, understand its meaning, and then produce a translation. Neural Machine Translation (NMT) mimics that!

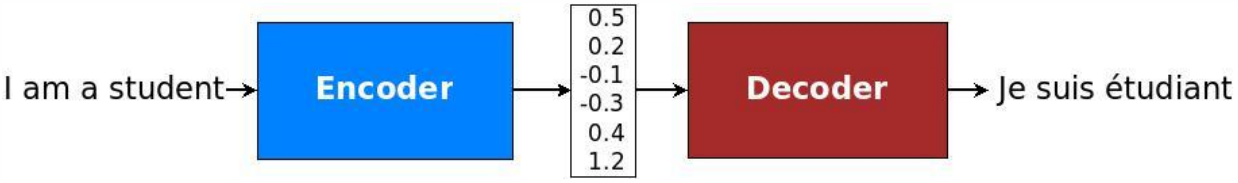


Figure 1. **Encoder-decoder architecture** – example of a general approach for NMT. An encoder converts a source sentence into a "meaning" vector which is passed through a *decoder* to produce a translation.

Specifically, an NMT system first reads the source sentence using an *encoder* to build a "thought" vector, a sequence of numbers that represents the sentence meaning; a *decoder*, then, processes the sentence vector to emit a translation, as illustrated in Figure 1. This is often referred to as the *encoder-decoder architecture*. In this manner, NMT addresses the local translation problem in the traditional phrase-based approach: it can capture *long-range dependencies* in languages, e.g., gender agreements; syntax structures; etc., and produce much more fluent translations as demonstrated by Google Neural Machine Translation systems.

NMT models vary in terms of their exact architectures. A natural choice for sequential data is the recurrent neural network (RNN), used by most NMT models. Usually an RNN is used for both the encoder and decoder. The RNN models, however, differ in terms of: (a) *directionality* – unidirectional or bidirectional; (b) *depth* – single- or multi-layer; and (c) *type*– often either a vanilla RNN, a Long Short-term Memory (LSTM), or a gated recurrent unit (GRU). Interested readers can find more information about RNNs and LSTM on this blog post.

In this tutorial, we consider as examples a *deep multi-layer RNN* which is unidirectional and uses LSTM as a recurrent unit. We show an example of such a model in Figure 2. In this example, we build a model to translate a source sentence "I am a student" into a target sentence "Je suis étudiant". At a high level, the NMT model consists of two recurrent neural networks: the *encoder* RNN simply consumes the input source words without making any prediction; the *decoder*, on the other hand, processes the target sentence while predicting the next words.

For more information, we refer readers to Luong (2016) which this tutorial is based on.

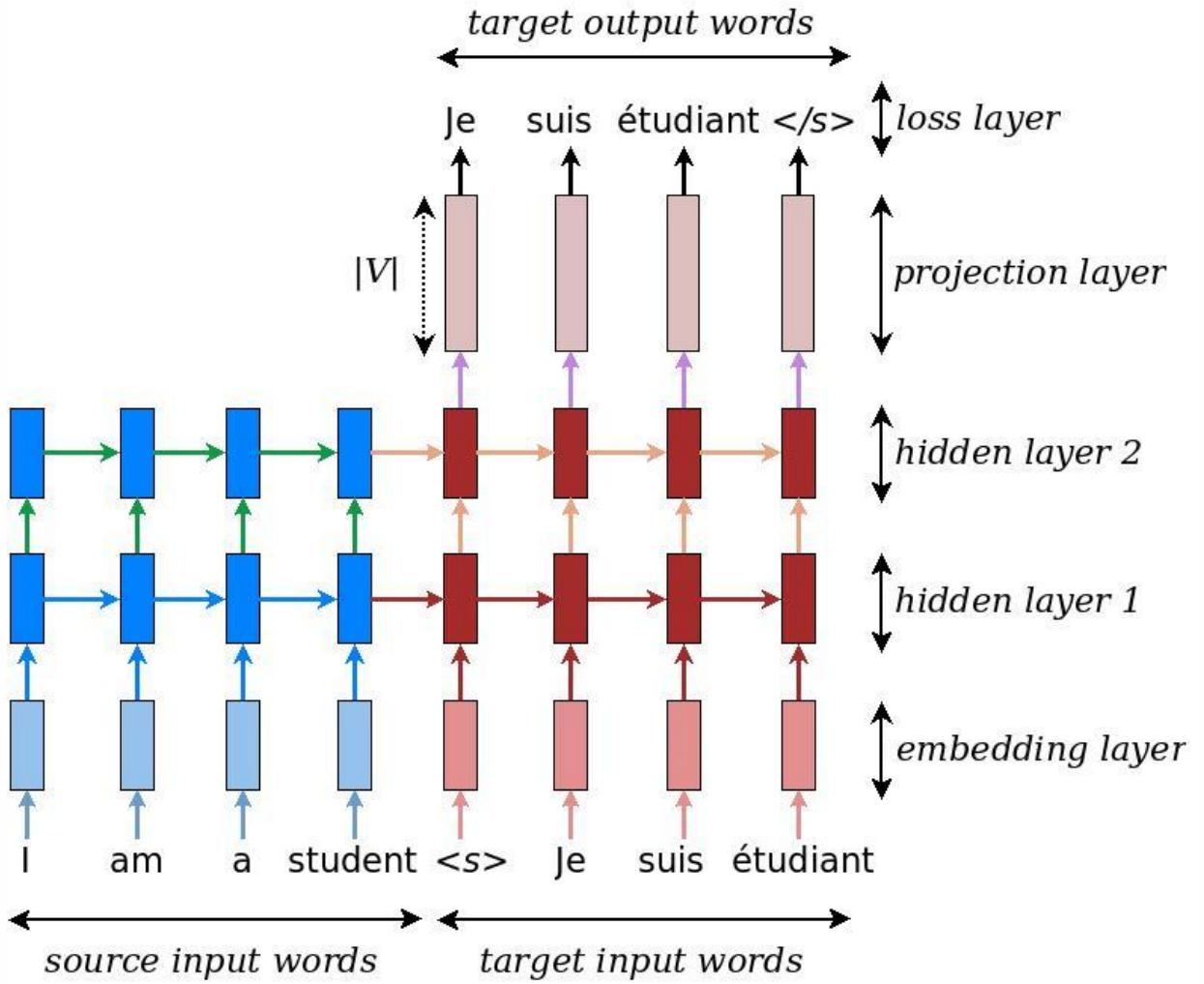


Figure 2. **Neural machine translation** – example of a deep recurrent architecture proposed by for translating a source sentence "I am a student" into a target sentence "Je suis étudiant". Here, "~~" marks the start of the decoding process while "~~"" tells the decoder to stop.

Installing the Tutorial

To install this tutorial, you need to have TensorFlow installed on your system. This tutorial requires TensorFlow Nightly. To install TensorFlow, follow the installation instructions here.

Once TensorFlow is installed, you can download the source code of this tutorial by running:

```
git clone https://github.com/tensorflow/nmt/
```

Training – How to build our first NMT system

Let's first dive into the heart of building an NMT model with concrete code snippets through which we will explain Figure 2 in more detail. We defer data preparation and the full code to later. This part refers to file **model.py**.

At the bottom layer, the encoder and decoder RNNs receive as input the following: first, the source sentence, then a boundary marker "<s>" which indicates the transition from the encoding to the decoding mode, and the target sentence. For *training*, we will feed the system with the following tensors, which are in time-major format and contain word indices:

- **encoder_inputs** [max_encoder_time, batch_size]: source input words.
- **decoder_inputs** [max_decoder_time, batch_size]: target input words.
- **decoder_outputs** [max_decoder_time, batch_size]: target output words, these are decoder_inputs shifted to the left by one time step with an end-of-sentence tag appended on the right.

Here for efficiency, we train with multiple sentences (batch_size) at once. Testing is slightly different, so we will discuss it later.

Embedding

Given the categorical nature of words, the model must first look up the source and target embeddings to retrieve the corresponding word representations. For this *embedding layer* to work, a vocabulary is first chosen for each language. Usually, a vocabulary size V is selected, and only the most frequent V words are treated as unique. All other words are converted to an "unknown" token and all get the same embedding. The embedding weights, one set per language, are usually learned during training.

```
# Embedding
embedding_encoder = variable_scope.get_variable(
    "embedding_encoder", [src_vocab_size, embedding_size], ...)
# Look up embedding:
#   encoder_inputs: [max_time, batch_size]
#   encoder_emb_inp: [max_time, batch_size, embedding_size]
encoder_emb_inp = embedding_ops.embedding_lookup(
    embedding_encoder, encoder_inputs)
```

Similarly, we can build *embedding_decoder* and *decoder_emb_inp*. Note that one can choose to initialize embedding weights with pretrained word representations such as word2vec or Glove vectors. In general, given a large amount of training data we can learn these embeddings from scratch.

Encoder

Once retrieved, the word embeddings are then fed as input into the main network, which consists of two multi-layer RNNs – an encoder for the source language and a decoder for the target language. These two RNNs, in principle, can share the same weights; however, in practice, we often use two different RNN parameters (such models do a better job when fitting large training datasets). The *encoder* RNN uses zero vectors as its starting states and is built as follows:

```
# Build RNN cell
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

# Run Dynamic RNN
#   encoder_outputs: [max_time, batch_size, num_units]
#   encoder_state: [batch_size, num_units]
encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
    encoder_cell, encoder_emb_inp,
    sequence_length=source_sequence_length, time_major=True)
```

Note that sentences have different lengths to avoid wasting computation, we tell *dynamic_rnn* the exact source sentence lengths through *source_sequence_length*. Since our input is time major, we set *time_major=True*. Here, we build only a single layer LSTM, *encoder_cell*. We will describe how to build multi-layer LSTMs, add dropout, and use attention in a later section.

Decoder

The *decoder* also needs to have access to the source information, and one simple way to achieve that is to initialize it with the last hidden state of the encoder, *encoder_state*. In Figure 2, we pass the hidden state at the source word "student" to the decoder side.

```
# Build RNN cell
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)
```

```
# Helper
helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, decoder_lengths, time_major=True)
# Decoder
decoder = tf.contrib.seq2seq.BasicDecoder(
    decoder_cell, helper, encoder_state,
    output_layer=projection_layer)
# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(decoder, ...)
logits = outputs.rnn_output
```

Here, the core part of this code is the *BasicDecoder* object, *decoder*, which receives *decoder_cell* (similar to encoder_cell), a *helper*, and the previous *encoder_state* as inputs. By separating out decoders and helpers, we can reuse different codebases, e.g., *TrainingHelper* can be substituted with *GreedyEmbeddingHelper* to do greedy decoding. See more in[helper.py](helper.py).

Lastly, we haven't mentioned *projection_layer* which is a dense matrix to turn the top hidden states to logit vectors of dimension V. We illustrate this process at the top of Figure 2.

```
projection_layer = layers_core.Dense(
    tgt_vocab_size, use_bias=False)
```

Loss

Given the *logits* above, we are now ready to compute our training loss:

```
crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=decoder_outputs, logits=logits)
train_loss = (tf.reduce_sum(crossent * target_weights) /
    batch_size)
```

Here, *target_weights* is a zero-one matrix of the same size as *decoder_outputs*. It masks padding positions outside of the target sequence lengths with values 0.

Important note: It's worth pointing out that we divide the loss by *batch_size*, so our hyperparameters are "invariant" to batch_size. Some people divide the loss by (*batch_size* * *num_time_steps*), which plays down the errors made on short sentences. More subtly, our hyperparameters (applied to the former way) can't be used for the latter way. For example, if both approaches use SGD with a learning of 1.0, the latter approach effectively uses a much smaller learning rate of 1 / *num_time_steps*.

Gradient computation & optimization

We have now defined the forward pass of our NMT model. Computing the backpropagation pass is just a matter of a few lines of code:

```
# Calculate and clip gradients
params = tf.trainable_variables()
gradients = tf.gradients(train_loss, params)
clipped_gradients, _ = tf.clip_by_global_norm(
    gradients, max_gradient_norm)
```

One of the important steps in training RNNs is gradient clipping. Here, we clip by the global norm. The max value, *max_gradient_norm*, is often set to a value like 5 or 1. The last step is selecting the optimizer. The Adam optimizer is a common choice. We also select a learning rate. The value of *learning_rate* can is usually in the range 0.0001 to 0.001; and can be set to decrease as training progresses.

```
# Optimization
optimizer = tf.train.AdamOptimizer(learning_rate)
update_step = optimizer.apply_gradients(
    zip(clipped_gradients, params))
```

In our own experiments, we use standard SGD (tf.train.GradientDescentOptimizer) with a decreasing learning rate schedule, which yields better performance. See the benchmarks.

Hands-on – Let's train an NMT model

Let's train our very first NMT model, translating from Vietnamese to English! The entry point of our code is **nmt.py**.

We will use a *small-scale parallel corpus of TED talks* (133K training examples) for this exercise. All data we used here can be found at: https://nlp.stanford.edu/projects/nmt/. We will use tst2012 as our dev dataset, and tst2013 as our test dataset.

Run the following command to download the data for training NMT model:\
`nmt/scripts/download_iwslt15.sh /tmp/nmt_data`

Run the following command to start the training:

```
mkdir /tmp/nmt_model
python -m nmt.nmt \
    --src=vi --tgt=en \
    --vocab_prefix=/tmp/nmt_data/vocab  \
    --train_prefix=/tmp/nmt_data/train \
    --dev_prefix=/tmp/nmt_data/tst2012  \
    --test_prefix=/tmp/nmt_data/tst2013 \
    --out_dir=/tmp/nmt_model \
    --num_train_steps=12000 \
    --steps_per_stats=100 \
    --num_layers=2 \
    --num_units=128 \
    --dropout=0.2 \
    --metrics=bleu
```

The above command trains a 2-layer LSTM seq2seq model with 128-dim hidden units and embeddings for 12 epochs. We use a dropout value of 0.2 (keep probability 0.8). If no error, we should see logs similar to the below with decreasing perplexity values as we train.

```
# First evaluation, global step 0
  eval dev: perplexity 17193.66
  eval test: perplexity 17193.27
# Start epoch 0, step 0, lr 1, Tue Apr 25 23:17:41 2017
  sample train data:
    src_reverse: </s> </s> Điều đó , dĩ nhiên , là câu chuyện trích ra từ
học thuyết của Karl Marx .
    ref: That , of course , was the <unk> distilled from the theories of
Karl Marx . </s> </s> </s>
  epoch 0 step 100 lr 1 step-time 0.89s wps 5.78K ppl 1568.62 bleu 0.00
  epoch 0 step 200 lr 1 step-time 0.94s wps 5.91K ppl 524.11 bleu 0.00
  epoch 0 step 300 lr 1 step-time 0.96s wps 5.80K ppl 340.05 bleu 0.00
  epoch 0 step 400 lr 1 step-time 1.02s wps 6.06K ppl 277.61 bleu 0.00
  epoch 0 step 500 lr 1 step-time 0.95s wps 5.89K ppl 205.85 bleu 0.00
```

See **train.py** for more details.

We can start Tensorboard to view the summary of the model during training:

```
tensorboard --port 22222 --logdir /tmp/nmt_model/
```

Training the reverse direction from English and Vietnamese can be done simply by changing:\ `--src=en --tgt=vi`

Inference – How to generate translations

While you're training your NMT models (and once you have trained models), you can obtain translations given previously unseen source sentences. This process is called inference. There is a clear distinction between training and inference (*testing*): at inference time, we only have access to the source sentence, i.e., *encoder_inputs*. There are many ways to perform decoding. Decoding methods include greedy, sampling, and beam-search decoding. Here, we will discuss the greedy decoding strategy.

The idea is simple and we illustrate it in Figure 3:

1. We still encode the source sentence in the same way as during training to obtain an *encoder_state*, and this *encoder_state* is used to initialize the decoder.
2. The decoding (translation) process is started as soon as the decoder receives a starting symbol "<s>" (refer as *tgt_sos_id* in our code);
3. For each timestep on the decoder side, we treat the RNN's output as a set of logits. We choose the most likely word, the id associated with the maximum logit value, as the emitted word (this is the "greedy" behavior). For example in Figure 3, the word "moi" has the highest translation probability in the first decoding step. We then feed this word as input to the next timestep.
4. The process continues until the end-of-sentence marker "</s>" is produced as an output symbol (refer as *tgt_eos_id* in our code).

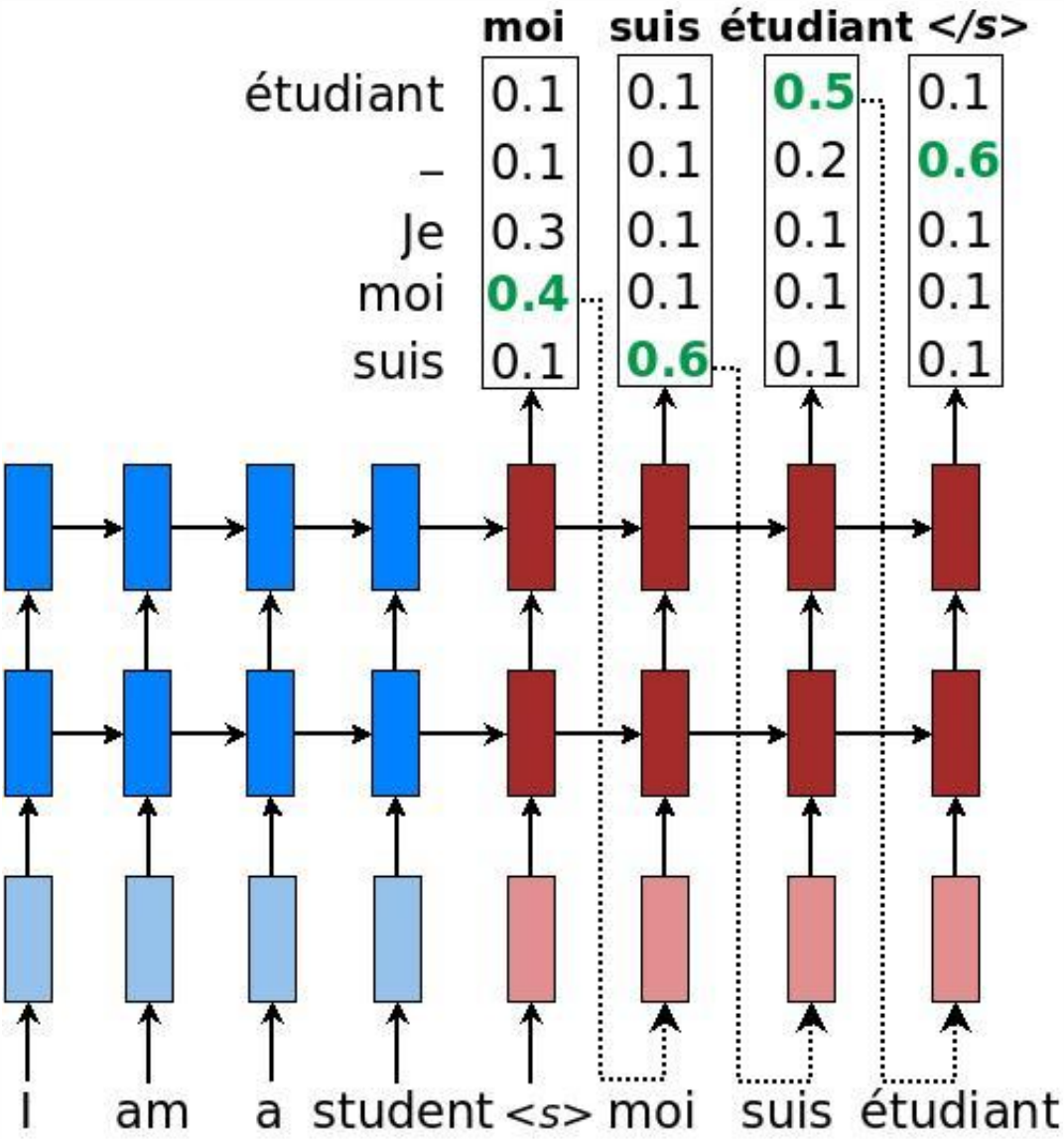


Figure 3. **Greedy decoding** – example of how a trained NMT model produces a translation for a source sentence "Je suis étudiant" using greedy search.

Step 3 is what makes inference different from training. Instead of always feeding the correct target words as an input, inference uses words predicted by the model. Here's the code to achieve greedy decoding. It is very similar to the training decoder.

```
# Helper
helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
    embedding_decoder,
    tf.fill([batch_size], tgt_sos_id), tgt_eos_id)

# Decoder
decoder = tf.contrib.seq2seq.BasicDecoder(
    decoder_cell, helper, encoder_state,
    output_layer=projection_layer)
# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, maximum_iterations=maximum_iterations)
translations = outputs.sample_id
```

Here, we use *GreedyEmbeddingHelper* instead of *TrainingHelper*. Since we do not know the target sequence lengths in advance, we use *maximum_iterations* to limit the translation lengths. One heuristic is to decode up to two times the source sentence lengths.

```
maximum_iterations = tf.round(tf.reduce_max(source_sequence_length) * 2)
```

Having trained a model, we can now create an inference file and translate some sentences:

```
cat > /tmp/my_infer_file.vi
# (copy and paste some sentences from /tmp/nmt_data/tst2013.vi)

python -m nmt.nmt \
    --out_dir=/tmp/nmt_model \
    --inference_input_file=/tmp/my_infer_file.vi \
    --inference_output_file=/tmp/nmt_model/output_infer

cat /tmp/nmt_model/output_infer # To view the inference as output
```

Note the above commands can also be run while the model is still being trained as long as there exists a training checkpoint. See **inference.py** for more details.

Intermediate

Having gone through the most basic seq2seq model, let's get more advanced! To build state-of-the-art neural machine translation systems, we will need more "secret sauce": the *attention mechanism*, which was first introduced by Bahdanau et al., 2015, then later refined by Luong et al., 2015 and others. The key idea of the attention mechanism is to establish direct short-cut connections between the target and the source by paying "attention" to relevant source content as we translate. A nice byproduct of the attention mechanism is an easy-to-visualize alignment matrix between the source and target sentences (as shown in Figure 4).

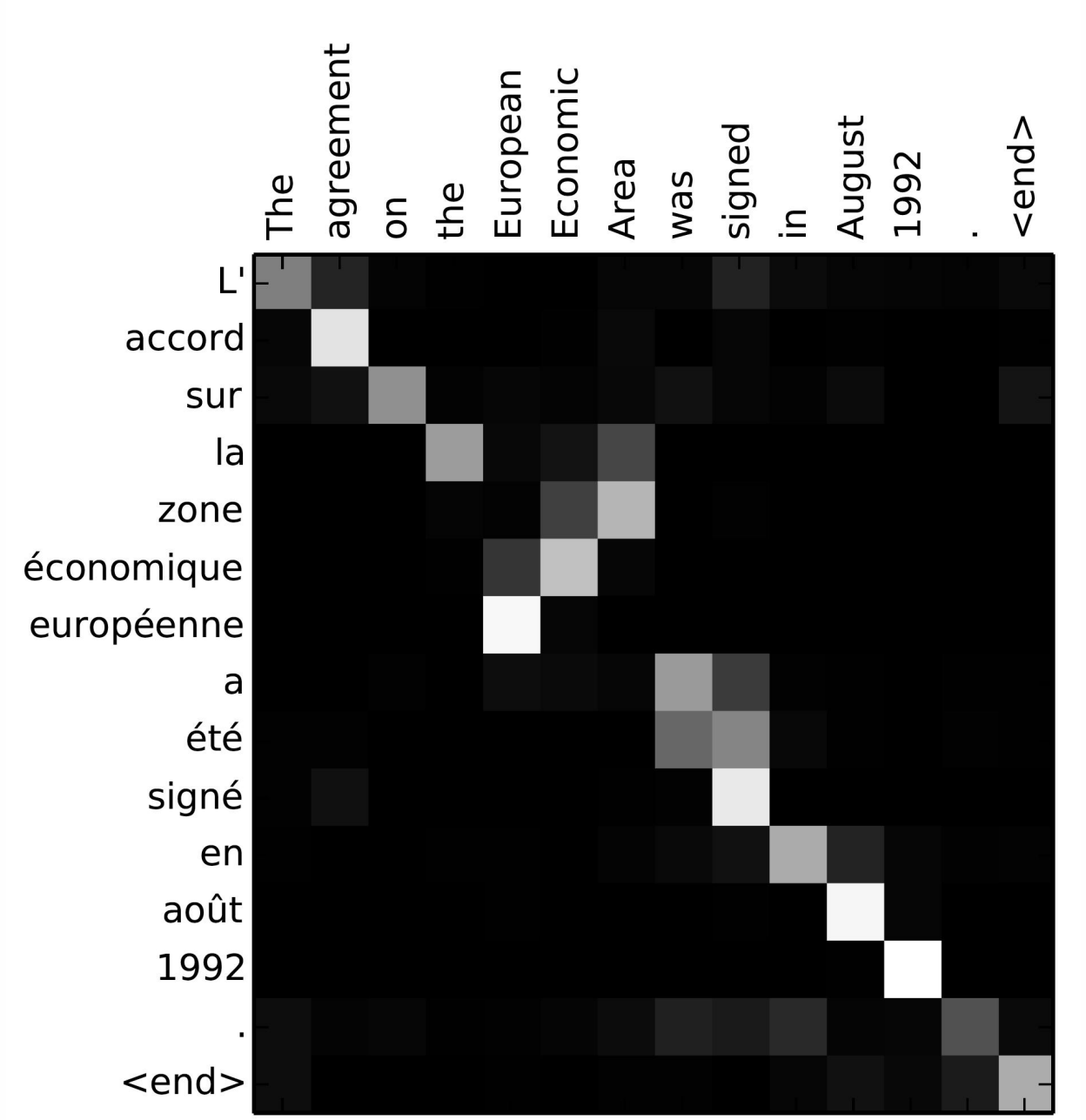


Figure 4. **Attention visualization** – example of the alignments between source and target sentences. Image is taken from (Bahdanau et al., 2015).

Remember that in the vanilla seq2seq model, we pass the last source state from the encoder to the decoder when starting the decoding process. This works well for short and medium-length sentences; however, for long sentences, the single fixed-size hidden state becomes an information bottleneck. Instead of discarding all of the hidden states computed in the source RNN, the attention mechanism provides an approach that allows the decoder to peek at them (treating them as a dynamic memory of the source information). By doing so, the attention mechanism improves the translation of longer sentences. Nowadays, attention mechanisms are the defacto standard and have been successfully applied to many other tasks (including image caption generation, speech recognition, and text summarization).

Background on the Attention Mechanism

We now describe an instance of the attention mechanism proposed in (Luong et al., 2015), which has been used in several state-of-the-art systems including open-source toolkits such as OpenNMT and in the TF seq2seq API in this tutorial. We will also provide connections to other variants of the attention mechanism.

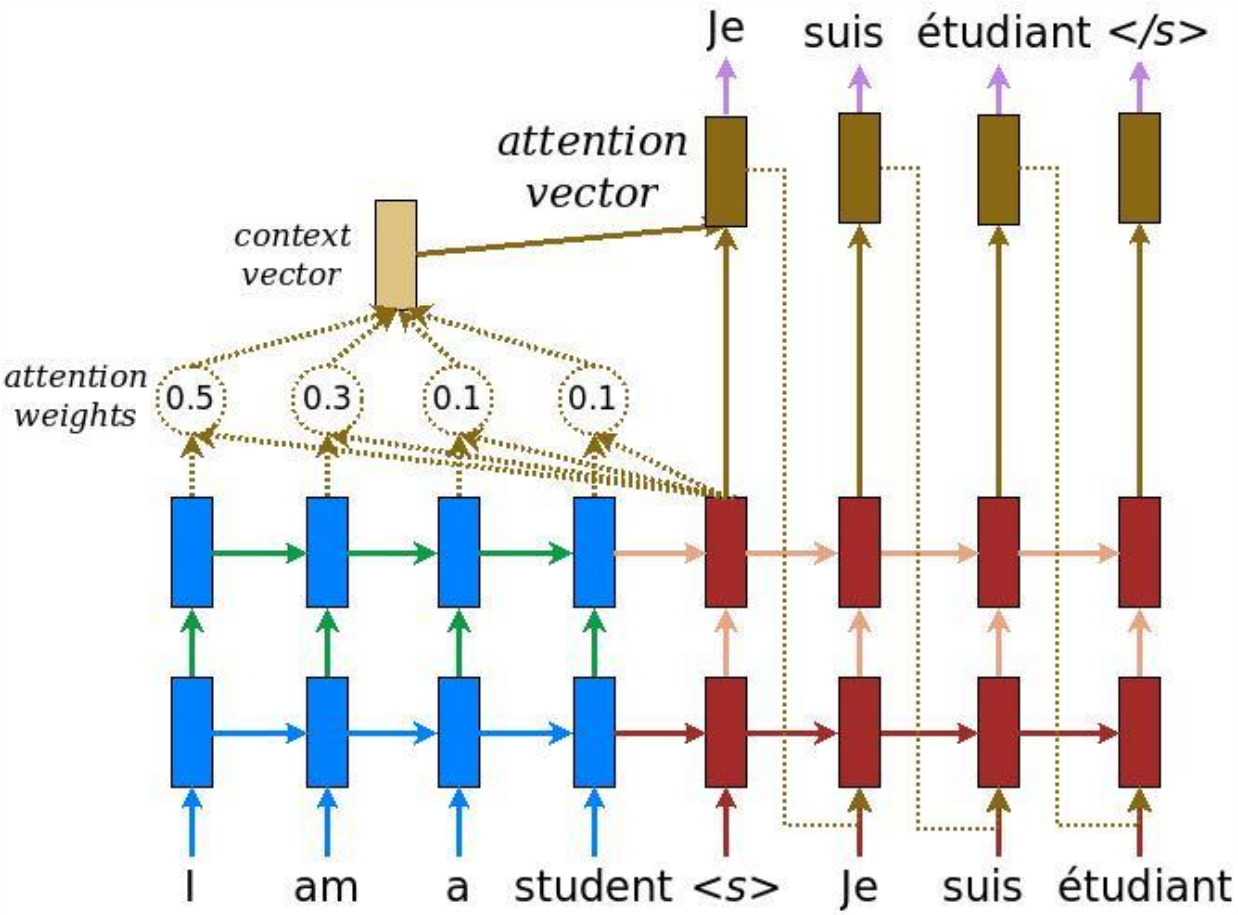


Figure 5. **Attention mechanism** – example of an attention-based NMT system as described in (Luong et al., 2015) . We highlight in detail the first step of the attention computation. For clarity, we don't show the embedding and projection layers in Figure (2).

As illustrated in Figure 5, the attention computation happens at every decoder time step. It consists of the following stages:

1. The current target hidden state is compared with all source states to derive *attention weights* (can be visualized as in Figure 4).
2. Based on the attention weights we compute a *context vector* as the weighted average of the source states.
3. Combine the context vector with the current target hidden state to yield the final *attention vector*
4. The attention vector is fed as an input to the next time step (*input feeding*). The first three steps can be summarized by the equations below:

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

Here, the function `score` is used to compared the target hidden state ht with each of the source hidden states h¯s, and the result is normalized to produced attention weights (a distribution over source positions). There are various choices of the scoring function; popular scoring functions include the multiplicative and additive forms given in Eq. (4). Once computed, the attention vector at is used to derive the softmax logit and loss. This is similar to the target hidden state at the top layer of a vanilla seq2seq model. The function `f` can also take other forms.

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \tag{4}$$

Various implementations of attention mechanisms can be found in attention_wrapper.py.

What matters in the attention mechanism?

As hinted in the above equations, there are many different attention variants. These variants depend on the form of the scoring function and the attention function, and on whether the previous state ht−1 is used instead of ht in the scoring function as originally suggested in (Bahdanau et al., 2015). Empirically, we found that only certain choices matter. First, the basic form of attention, i.e., direct connections between target and source, needs to be present. Second, it's important to feed the attention vector to the next timestep to inform the network about past attention decisions as demonstrated in (Luong et al., 2015). Lastly, choices of the scoring function can often result in different performance. See more in the benchmark results section.

Attention Wrapper API

In our implementation of the AttentionWrapper, we borrow some terminology from (Weston et al., 2015) in their work on*memory networks*. Instead of having readable & writable memory, the attention mechanism presented in this tutorial is a *read-only* memory. Specifically, the set of source hidden states (or their transformed versions, e.g., Wh¯s in Luong's scoring style or W2h¯s in Bahdanau's scoring style) is referred to as the *"memory"*. At each time step, we use the current target hidden state as a *"query"* to decide on which parts of the memory to read. Usually, the query needs to be compared with keys corresponding to individual memory slots. In the above presentation of the attention mechanism, we happen to use the set of source hidden states (or their transformed versions, e.g., W1ht in Bahdanau's scoring style) as "keys". One can be inspired by this memory-network terminology to derive other forms of attention!

Thanks to the attention wrapper, extending our vanilla seq2seq code with attention is trivial. This part refers to file **attention_model.py**

First, we need to define an attention mechanism, e.g., from (Luong et al., 2015):

```
# attention_states: [batch_size, max_time, num_units]
attention_states = tf.transpose(encoder_outputs, [1, 0, 2])

# Create an attention mechanism
attention_mechanism = tf.contrib.seq2seq.LuongAttention(
    num_units, attention_states,
    memory_sequence_length=source_sequence_length)
```

In the previous Encoder section, *encoder_outputs* is the set of all source hidden states at the top layer and has the shape of *[max_time, batch_size, num_units]* (since we use *dynamic_rnn* with *time_major* set to *True* for efficiency). For the attention mechanism, we need to make sure the "memory" passed in is batch major, so we need to transpose*attention_states*. We pass *source_sequence_length* to the attention mechanism to ensure that the attention weights are properly normalized (over non-padding positions only).

Having defined an attention mechanism, we use *AttentionWrapper* to wrap the decoding cell:

```
decoder_cell = tf.contrib.seq2seq.AttentionWrapper(
    decoder_cell, attention_mechanism,
    attention_layer_size=num_units)
```

The rest of the code is almost the same as in the Section Decoder!

Hands-on – building an attention-based NMT model

To enable attention, we need to use one of `luong`, `scaled_luong`, `bahdanau` or `normed_bahdanau` as the value of the `attention` flag during training. The flag specifies which attention mechanism we are going to use. In addition, we need to create a new directory for the attention model, so we don't reuse the previously trained basic NMT model.

Run the following command to start the training:

```
mkdir /tmp/nmt_attention_model

python -m nmt.nmt \
    --attention=scaled_luong \
    --src=vi --tgt=en \
    --vocab_prefix=/tmp/nmt_data/vocab  \
    --train_prefix=/tmp/nmt_data/train \
    --dev_prefix=/tmp/nmt_data/tst2012  \
    --test_prefix=/tmp/nmt_data/tst2013 \
    --out_dir=/tmp/nmt_attention_model \
    --num_train_steps=12000 \
    --steps_per_stats=100 \
    --num_layers=2 \
    --num_units=128 \
    --dropout=0.2 \
    --metrics=bleu
```

After training, we can use the same inference command with the new out_dir for inference:

```
python -m nmt.nmt \
    --out_dir=/tmp/nmt_attention_model \
    --inference_input_file=/tmp/my_infer_file.vi \
    --inference_output_file=/tmp/nmt_attention_model/output_infer
```

Tips & Tricks

Building Training, Eval, and Inference Graphs

When building a machine learning model in TensorFlow, it's often best to build three separate graphs:

- The Training graph, which:
  - Batches, buckets, and possibly subsamples input data from a set of files/external inputs.
  - Includes the forward and backprop ops.
  - Constructs the optimizer, and adds the training op.
- The Eval graph, which:
  - Batches and buckets input data from a set of files/external inputs.
  - Includes the training forward ops, and additional evaluation ops that aren't used for training.
- The Inference graph, which:
  - May not batch input data.

- Does not subsample or bucket input data.
- Reads input data from placeholders (data can be fed directly to the graph via *feed_dict* or from a C++ TensorFlow serving binary).
- Includes a subset of the model forward ops, and possibly additional special inputs/outputs for storing state between session.run calls.

Building separate graphs has several benefits:

- The inference graph is usually very different from the other two, so it makes sense to build it separately.
- The eval graph becomes simpler since it no longer has all the additional backprop ops.
- Data feeding can be implemented separately for each graph.
- Variable reuse is much simpler. For example, in the eval graph there's no need to reopen variable scopes with *reuse=True* just because the Training model created these variables already. So the same code can be reused without sprinkling *reuse=* arguments everywhere.
- In distributed training, it is commonplace to have separate workers perform training, eval, and inference. These need to build their own graphs anyway. So building the system this way prepares you for distributed training.

The primary source of complexity becomes how to share Variables across the three graphs in a single machine setting. This is solved by using a separate session for each graph. The training session periodically saves checkpoints, and the eval session and the infer session restore parameters from checkpoints. The example below shows the main differences between the two approaches.

Before: Three models in a single graph and sharing a single Session

```
with tf.variable_scope('root'):
  train_inputs = tf.placeholder()
  train_op, loss = BuildTrainModel(train_inputs)
  initializer = tf.global_variables_initializer()

with tf.variable_scope('root', reuse=True):
  eval_inputs = tf.placeholder()
  eval_loss = BuildEvalModel(eval_inputs)

with tf.variable_scope('root', reuse=True):
  infer_inputs = tf.placeholder()
  inference_output = BuildInferenceModel(infer_inputs)

sess = tf.Session()

sess.run(initializer)

for i in itertools.count():
  train_input_data = ...
  sess.run([loss, train_op], feed_dict={train_inputs: train_input_data})

  if i % EVAL_STEPS == 0:
    while data_to_eval:
      eval_input_data = ...
      sess.run([eval_loss], feed_dict={eval_inputs: eval_input_data})

  if i % INFER_STEPS == 0:
    sess.run(inference_output, feed_dict={infer_inputs: infer_input_data})
```

After: Three models in three graphs, with three Sessions sharing the same Variables

```
train_graph = tf.Graph()
eval_graph = tf.Graph()
infer_graph = tf.Graph()

with train_graph.as_default():
  train_iterator = ...
  train_model = BuildTrainModel(train_iterator)
  initializer = tf.global_variables_initializer()

with eval_graph.as_default():
  eval_iterator = ...
  eval_model = BuildEvalModel(eval_iterator)

with infer_graph.as_default():
  infer_iterator, infer_inputs = ...
  infer_model = BuildInferenceModel(infer_iterator)

checkpoints_path = "/tmp/model/checkpoints"

train_sess = tf.Session(graph=train_graph)
eval_sess = tf.Session(graph=eval_graph)
infer_sess = tf.Session(graph=infer_graph)

train_sess.run(initializer)
train_sess.run(train_iterator.initializer)

for i in itertools.count():

  train_model.train(train_sess)

  if i % EVAL_STEPS == 0:
    checkpoint_path = train_model.saver.save(train_sess, checkpoints_path,
global_step=i)
    eval_model.saver.restore(eval_sess, checkpoint_path)
    eval_sess.run(eval_iterator.initializer)
    while data_to_eval:
      eval_model.eval(eval_sess)

  if i % INFER_STEPS == 0:
    checkpoint_path = train_model.saver.save(train_sess, checkpoints_path,
global_step=i)
    infer_model.saver.restore(infer_sess, checkpoint_path)
    infer_sess.run(infer_iterator.initializer, feed_dict={infer_inputs:
infer_input_data})
    while data_to_infer:
      infer_model.infer(infer_sess)
```

Notice how the latter approach is "ready" to be converted to a distributed version.

One other difference in the new approach is that instead of using *feed_dicts* to feed data at each *session.run* call (and thereby performing our own batching, bucketing, and manipulating of data), we use stateful iterator objects. These iterators make the input pipeline much easier in both the single-machine and distributed setting. We will cover the new input data pipeline (as introduced in TensorFlow 1.2) in the next section.

Data Input Pipeline

Prior to TensorFlow 1.2, users had two options for feeding data to the TensorFlow training and eval pipelines:

1. Feed data directly via *feed_dict* at each training *session.run* call.
2. Use the queueing mechanisms in *tf.train* (e.g. *tf.train.batch*) and *tf.contrib.train*.
3. Use helpers from a higher level framework like *tf.contrib.learn* or *tf.contrib.slim* (which effectively use #2).

The first approach is easier for users who aren't familiar with TensorFlow or need to do exotic input modification (i.e., their own minibatch queueing) that can only be done in Python. The second and third approaches are more standard but a little less flexible; they also require starting multiple python threads (queue runners). Furthermore, if used incorrectly queues can lead to deadlocks or opaque error messages. Nevertheless, queues are significantly more efficient than using *feed_dict* and are the standard for both single-machine and distributed training.

Starting in TensorFlow 1.2, there is a new system available for reading data into TensorFlow models: dataset iterators, as found in the **tf.data** module. Data iterators are flexible, easy to reason about and to manipulate, and provide efficiency and multithreading by leveraging the TensorFlow C++ runtime.

A **dataset** can be created from a batch data Tensor, a filename, or a Tensor containing multiple filenames. Some examples:

```
# Training dataset consists of multiple files.
train_dataset = tf.data.TextLineDataset(train_files)

# Evaluation dataset uses a single file, but we may
# point to a different file for each evaluation round.
eval_file = tf.placeholder(tf.string, shape=())
eval_dataset = tf.data.TextLineDataset(eval_file)

# For inference, feed input data to the dataset directly via feed_dict.
infer_batch = tf.placeholder(tf.string, shape=(num_infer_examples,))
infer_dataset = tf.data.Dataset.from_tensor_slices(infer_batch)
```

All datasets can be treated similarly via input processing. This includes reading and cleaning the data, bucketing (in the case of training and eval), filtering, and batching.

To convert each sentence into vectors of word strings, for example, we use the dataset map transformation:

```
dataset = dataset.map(lambda string: tf.string_split([string]).values)
```

We can then switch each sentence vector into a tuple containing both the vector and its dynamic length:

```
dataset = dataset.map(lambda words: (words, tf.size(words))
```

Finally, we can perform a vocabulary lookup on each sentence. Given a lookup table object table, this map converts the first tuple elements from a vector of strings to a vector of integers.

```
dataset = dataset.map(lambda words, size: (table.lookup(words), size))
```

Joining two datasets is also easy. If two files contain line-by-line translations of each other and each one is read into its own dataset, then a new dataset containing the tuples of the zipped lines can be created via:

```
source_target_dataset = tf.data.Dataset.zip((source_dataset,
target_dataset))
```

Batching of variable-length sentences is straightforward. The following transformation batches *batch_size* elements from *source_target_dataset,* and respectively pads the source and target vectors to the length of the longest source and target vector in each batch.

```
batched_dataset = source_target_dataset.padded_batch(
        batch_size,
        padded_shapes=((tf.TensorShape([None]),  # source vectors of unknown
size
                        tf.TensorShape([])),     # size(source)
                       (tf.TensorShape([None]),  # target vectors of unknown
size
                        tf.TensorShape([]))),    # size(target)
        padding_values=((src_eos_id,  # source vectors padded on the right
with src_eos_id
                         0),          # size(source) -- unused
                        (tgt_eos_id,  # target vectors padded on the right
with tgt_eos_id
                         0)))         # size(target) -- unused
```

Values emitted from this dataset will be nested tuples whose tensors have a leftmost dimension of size *batch_size*. The structure will be:

- iterator[0][0] has the batched and padded source sentence matrices.
- iterator[0][1] has the batched source size vectors.

- iterator[1][0] has the batched and padded target sentence matrices.
- iterator[1][1] has the batched target size vectors.

Finally, bucketing that batches similarly-sized source sentences together is also possible. Please see the fileutils/iterator_utils.py for more details and the full implementation.

Reading data from a Dataset requires three lines of code: create the iterator, get its values, and initialize it.

```
batched_iterator = batched_dataset.make_initializable_iterator()

((source, source_lengths), (target, target_lengths)) =
batched_iterator.get_next()

# At initialization time.
session.run(batched_iterator.initializer, feed_dict={...})
```

Once the iterator is initialized, every *session.run* call that accesses source or target tensors will request the next minibatch from the underlying dataset.

Other details for better NMT models

Bidirectional RNNs

Bidirectionality on the encoder side generally gives better performance (with some degradation in speed as more layers are used). Here, we give a simplified example of how to build an encoder with a single bidirectional layer:

```
# Construct forward and backward cells
forward_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)
backward_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

bi_outputs, encoder_state = tf.nn.bidirectional_dynamic_rnn(
    forward_cell, backward_cell, encoder_emb_inp,
    sequence_length=source_sequence_length, time_major=True)
encoder_outputs = tf.concat(bi_outputs, -1)
```

The variables *encoder_outputs* and *encoder_state* can be used in the same way as in Section Encoder. Note that, for multiple bidirectional layers, we need to manipulate the encoder_state a bit, see model.py, method *_build_bidirectional_rnn()* for more details.

Beam search

While greedy decoding can give us quite reasonable translation quality, a beam search decoder can further boost performance. The idea of beam search is to better explore the search space of all possible translations by keeping around a small set of top candidates as we translate. The size of the beam is called *beam width*; a minimal beam width of, say size 10, is generally sufficient. For more information, we refer readers to Section 7.2.3 of Neubig, (2017). Here's an example of how beam search can be done:

```
# Replicate encoder infos beam_width times
decoder_initial_state = tf.contrib.seq2seq.tile_batch(
    encoder_state, multiplier=hparams.beam_width)

# Define a beam-search decoder
decoder = tf.contrib.seq2seq.BeamSearchDecoder(
        cell=decoder_cell,
        embedding=embedding_decoder,
        start_tokens=start_tokens,
        end_token=end_token,
        initial_state=decoder_initial_state,
        beam_width=beam_width,
        output_layer=projection_layer,
        length_penalty_weight=0.0)

# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(decoder, ...)
```

Note that the same *dynamic_decode()* API call is used, similar to the Section Decoder. Once decoded, we can access the translations as follows:

```
translations = outputs.predicted_ids
# Make sure translations shape is [batch_size, beam_width, time]
if self.time_major:
   translations = tf.transpose(translations, perm=[1, 2, 0])
```

See model.py, method *_build_decoder()* for more details.

Hyperparameters

There are several hyperparameters that can lead to additional performances. Here, we list some based on our own experience [Disclaimers: others might not agree on things we wrote!].

Optimizer: while Adam can lead to reasonable results for "unfamiliar" architectures, SGD with scheduling will generally lead to better performance if you can train with SGD.

Attention: Bahdanau-style attention often requires bidirectionality on the encoder side to work well; whereas Luong-style attention tends to work well for different settings. For this tutorial code, we recommend using the two improved variants of Luong & Bahdanau-style attentions: *scaled_luong* & *normed bahdanau*.

Multi-GPU training

Training a NMT model may take several days. Placing different RNN layers on different GPUs can improve the training speed. Here's an example to create RNN layers on multiple GPUs.

```
cells = []
for i in range(num_layers):
  cells.append(tf.contrib.rnn.DeviceWrapper(
      tf.contrib.rnn.LSTMCell(num_units),
      "/gpu:%d" % (num_layers % num_gpus)))
cell = tf.contrib.rnn.MultiRNNCell(cells)
```

In addition, we need to enable the `colocate_gradients_with_ops` option in `tf.gradients` to parallelize the gradients computation.

You may notice the speed improvement of the attention based NMT model is very small as the number of GPUs increases. One major drawback of the standard attention architecture is using the top (final) layer's output to query attention at each time step. That means each decoding step must wait its previous step completely finished; hence, we can't parallelize the decoding process by simply placing RNN layers on multiple GPUs.

The GNMT attention architecture parallelizes the decoder's computation by using the bottom (first) layer's output to query attention. Therefore, each decoding step can start as soon as its previous step's first layer and attention computation finished. We implemented the architecture in GNMTAttentionMultiCell, a subclass of *tf.contrib.rnn.MultiRNNCell*. Here's an example of how to create a decoder cell with the *GNMTAttentionMultiCell*.

```
cells = []
for i in range(num_layers):
  cells.append(tf.contrib.rnn.DeviceWrapper(
      tf.contrib.rnn.LSTMCell(num_units),
      "/gpu:%d" % (num_layers % num_gpus)))
attention_cell = cells.pop(0)
attention_cell = tf.contrib.seq2seq.AttentionWrapper(
    attention_cell,
    attention_mechanism,
    attention_layer_size=None,  # don't add an additional dense layer.
    output_attention=False,)
cell = GNMTAttentionMultiCell(attention_cell, cells)
```

Benchmarks

IWSLT English-Vietnamese

Train: 133K examples, vocab=vocab.(vi|en), train=train.(vi|en) dev=tst2012.(vi|en), test=tst2013.(vi|en), download script.

Training details. We train 2-layer LSTMs of 512 units with bidirectional encoder (i.e., 1 bidirectional layers for the encoder), embedding dim is 512. LuongAttention (scale=True) is used together with dropout keep_prob of 0.8. All parameters are uniformly. We use SGD with learning rate 1.0 as follows: train for 12K steps (~ 12 epochs); after 8K steps, we start halving learning rate every 1K step.

Results.

Below are the averaged results of 2 models (model 1, model 2).\ We measure the translation quality in terms of BLEU scores (Papineni et al., 2002).

Systems	**tst2012 (dev)**	**test2013 (test)**
NMT (greedy)	23.2	25.5
NMT (beam=10)	23.8	**26.1**
(Luong & Manning, 2015)	-	23.3

Training Speed: (0.37s step-time, 15.3K wps) on *K40m* & (0.17s step-time, 32.2K wps) on *TitanX*.\ Here, step-time means the time taken to run one mini-batch (of size 128). For wps, we count words on both the source and target.

WMT German-English

Train: 4.5M examples, vocab=vocab.bpe.32000.(de|en), train=train.tok.clean.bpe.32000.(de|en), dev=newstest2013.tok.bpe.32000.(de|en), test=newstest2015.tok.bpe.32000.(de|en), download script

Training details. Our training hyperparameters are similar to the English-Vietnamese experiments except for the following details. The data is split into subword units using BPE (32K operations). We train 4-layer LSTMs of 1024 units with bidirectional encoder (i.e., 2 bidirectional layers for the encoder), embedding dim is 1024. We train for 350K steps (~ 10 epochs); after 170K steps, we start halving learning rate every 17K step.

Results.

The first 2 rows are the averaged results of 2 models (model 1, model 2). Results in the third row is with GNMT attention (model) ; trained with 4 GPUs.

Systems	newstest2013 (dev)	newstest2015
NMT (greedy)	27.1	27.6
NMT (beam=10)	28.0	28.9
NMT + GNMT attention (beam=10)	29.0	**29.9**
WMT SOTA	-	29.3

These results show that our code builds strong baseline systems for NMT.\ (Note that WMT systems generally utilize a huge amount monolingual data which we currently do not.)

Training Speed: (2.1s step-time, 3.4K wps) on *Nvidia K40m* & (0.7s step-time, 8.7K wps) on *Nvidia TitanX* for standard models.\ To see the speed-ups with GNMT attention, we benchmark on *K40m* only:

Systems	1 gpu	4 gpus	8 gpus
NMT (4 layers)	2.2s, 3.4K	1.9s, 3.9K	-
NMT (8 layers)	3.5s, 2.0K	-	2.9s, 2.4K
NMT + GNMT attention (4 layers)	2.6s, 2.8K	1.7s, 4.3K	-
NMT + GNMT attention (8 layers)	4.2s, 1.7K	-	1.9s, 3.8K

These results show that without GNMT attention, the gains from using multiple gpus are minimal.\ With GNMT attention, we obtain from 50%-100% speed-ups with multiple gpus.

WMT English-German — Full Comparison

The first 2 rows are our models with GNMT attention: model 1 (4 layers), model 2 (8 layers).

Systems	newstest2014	newstest2015
Ours — NMT + GNMT attention (4 layers)	23.7	26.5
Ours — NMT + GNMT attention (8 layers)	24.4	**27.6**
WMT SOTA	20.6	24.9
OpenNMT (Klein et al., 2017)	19.3	-
tf-seq2seq (Britz et al., 2017)	22.2	25.2
GNMT (Wu et al., 2016)	**24.6**	-

The above results show our models are very competitive among models of similar architectures.\
[Note that OpenNMT uses smaller models and the current best result (as of this writing) is 28.4 obtained by the Transformer network (Vaswani et al., 2017) which has a significantly different architecture.]

Standard HParams

We have provided a set of standard hparams for using pre-trained checkpoint for inference or training NMT architectures used in the Benchmark.

We will use the WMT16 German-English data, you can download the data by the following command.

```
nmt/scripts/wmt16_en_de.sh /tmp/wmt16
```

Here is an example command for loading the pre-trained GNMT WMT German-English checkpoint for inference.

```
python -m nmt.nmt \
    --src=de --tgt=en \
    --ckpt=/path/to/checkpoint/translate.ckpt \
    --hparams_path=nmt/standard_hparams/wmt16_gnmt_4_layer.json \
    --out_dir=/tmp/deen_gnmt \
    --vocab_prefix=/tmp/wmt16/vocab.bpe.32000 \
    --inference_input_file=/tmp/wmt16/newstest2014.tok.bpe.32000.de \
    --inference_output_file=/tmp/deen_gnmt/output_infer \
    --inference_ref_file=/tmp/wmt16/newstest2014.tok.bpe.32000.en
```

Here is an example command for training the GNMT WMT German-English model.

```
python -m nmt.nmt \
    --src=de --tgt=en \
    --hparams_path=nmt/standard_hparams/wmt16_gnmt_4_layer.json \
    --out_dir=/tmp/deen_gnmt \
    --vocab_prefix=/tmp/wmt16/vocab.bpe.32000 \
    --train_prefix=/tmp/wmt16/train.tok.clean.bpe.32000 \
    --dev_prefix=/tmp/wmt16/newstest2013.tok.bpe.32000 \
    --test_prefix=/tmp/wmt16/newstest2015.tok.bpe.32000
```

Other resources

For deeper reading on Neural Machine Translation and sequence-to-sequence models, we highly recommend the following materials by Luong, Cho, Manning, (2016); Luong, (2016); and Neubig, (2017).

There's a wide variety of tools for building seq2seq models, so we pick one per language:\ Stanford NMThttps://nlp.stanford.edu/projects/nmt/ *[Matlab]* \ tf-seq2seq https://github.com/google/seq2seq *[TensorFlow]* \ Nemantus https://github.com/rsennrich/nematus *[Theano]* \ OpenNMT http://opennmt.net/ *[Torch]*\ OpenNMT-py https://github.com/OpenNMT/OpenNMT-py *[PyTorch]*

Acknowledgment

We would like to thank Denny Britz, Anna Goldie, Derek Murray, and Cinjon Resnick for their work bringing new features to TensorFlow and the seq2seq library. Additional thanks go to Lukasz Kaiser for the initial help on the seq2seq codebase; Quoc Le for the suggestion to replicate GNMT; Yonghui Wu and Zhifeng Chen for details on the GNMT systems; as well as the Google Brain team for their support and feedback!

References

- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. ICLR.
- Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. EMNLP.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. NIPS.

BibTex

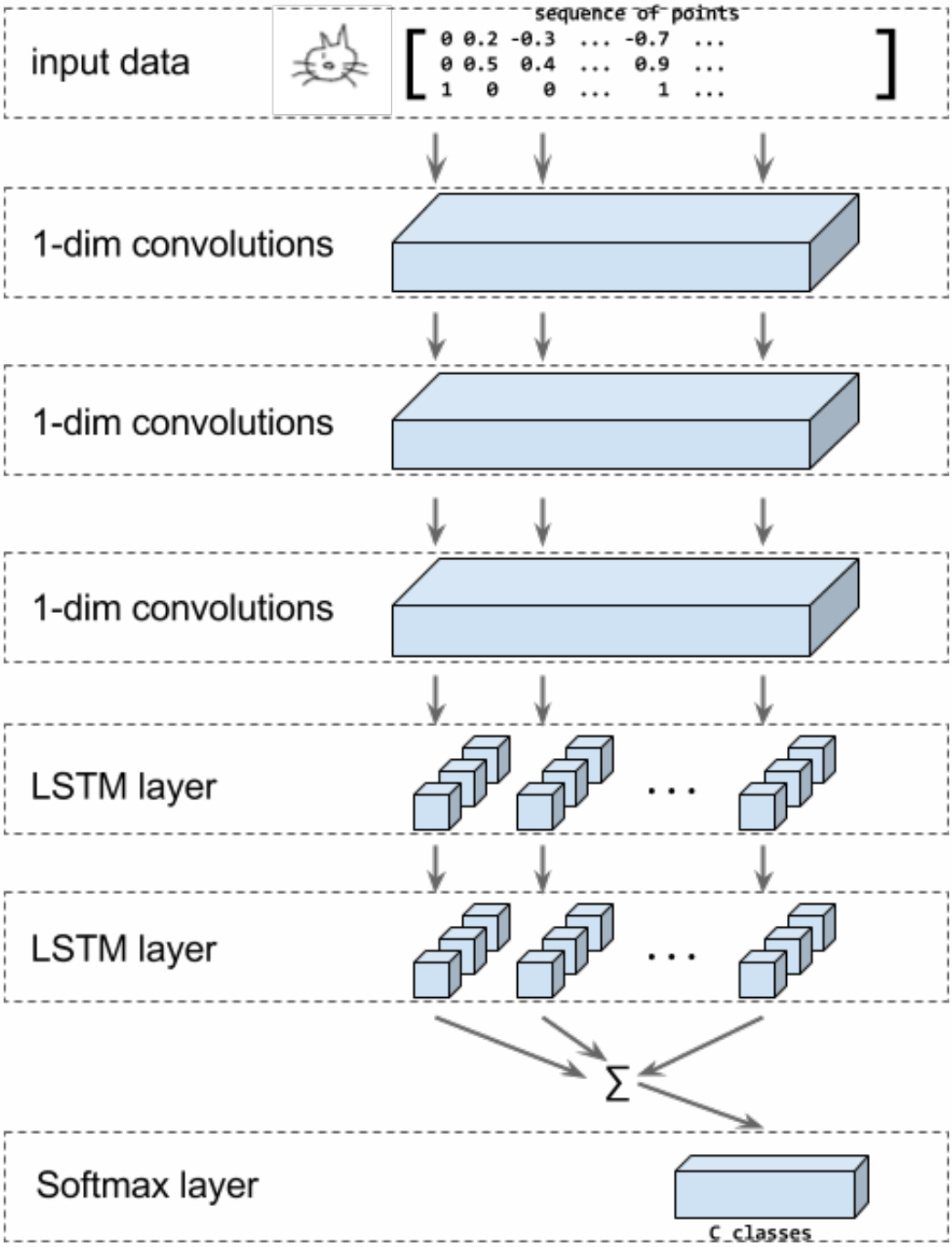
```
@article{luong17,
  author  = {Minh{-}Thang Luong and Eugene Brevdo and Rui Zhao},
  title   = {Neural Machine Translation (seq2seq) Tutorial},
  journal = {https://github.com/tensorflow/nmt},
  year    = {2017},
}
```

Recurrent Neural Networks for Drawing Classification

Quick, Draw! is a game where a player is challenged to draw a number of objects and see if a computer can recognize the drawing.

The recognition in Quick, Draw! is performed by a classifier that takes the user input, given as a sequence of strokes of points in x and y, and recognizes the object category that the user tried to draw.

In this tutorial we'll show how to build an RNN-based recognizer for this problem. The model will use a combination of convolutional layers, LSTM layers, and a softmax output layer to classify the drawings:



The figure above shows the structure of the model that we will build in this tutorial. The input is a drawing that is encoded as a sequence of strokes of points in x, y, and n, where n indicates whether a the point is the first point in a new stroke.

Then, a series of 1-dimensional convolutions is applied. Then LSTM layers are applied and the sum of the outputs of all LSTM steps is fed into a softmax layer to make a classification decision among the classes of drawings that we know.

This tutorial uses the data from actual Quick, Draw! games that is publicly available. This dataset contains of 50M drawings in 345 categories.

Run the tutorial code

To try the code for this tutorial:

1. Install TensorFlow if you haven't already.
2. Download the tutorial code .
3. Download the data in `TFRecord` format from here and unzip it. More details about how to obtain the original Quick, Draw! data and how to convert that to `TFRecord` files is available below.
4. Execute the tutorial code with the following command to train the RNN-based model described in this tutorial. Make sure to adjust the paths to point to the unzipped data from the download in step 3.

```
python train_model.py \
  --training_data=rnn_tutorial_data/training.tfrecord-?????-of-????? \
  --eval_data=rnn_tutorial_data/eval.tfrecord-?????-of-????? \
  --classes_file=rnn_tutorial_data/training.tfrecord.classes
```

Tutorial details

Download the data

We make the data that we use in this tutorial available as `TFRecord` files containing `TFExamples`. You can download the data from here:

http://download.tensorflow.org/data/quickdraw_tutorial_dataset_v1.tar.gz

Alternatively you can download the original data in `ndjson` format from the Google cloud and convert it to the `TFRecord` files containing `TFExamples` yourself as described in the next section.

Optional: Download the full Quick Draw Data

The full Quick, Draw! dataset is available on Google Cloud Storage as ndjson files separated by category. You can browse the list of files in Cloud Console.

To download the data we recommend using gsutil to download the entire dataset. Note that the original .ndjson files require downloading ~22GB.

Then use the following command to check that your gsutil installation works and that you can access the data bucket:

```
gsutil ls -r "gs://quickdraw_dataset/full/simplified/*"
```

which will output a long list of files like the following:

```
gs://quickdraw_dataset/full/simplified/The Eiffel Tower.ndjson
gs://quickdraw_dataset/full/simplified/The Great Wall of China.ndjson
gs://quickdraw_dataset/full/simplified/The Mona Lisa.ndjson
gs://quickdraw_dataset/full/simplified/aircraft carrier.ndjson
...
```

Then create a folder and download the dataset there.

```
mkdir rnn_tutorial_data
cd rnn_tutorial_data
gsutil -m cp "gs://quickdraw_dataset/full/simplified/*" .
```

This download will take a while and download a bit more than 23GB of data.

Optional: Converting the data

To convert the `ndjson` files to TFRecord files containing ${tf.train.Example} protos run the following command.

```
   python create_dataset.py --ndjson_path rnn_tutorial_data \
      --output_path rnn_tutorial_data
```

This will store the data in 10 shards of TFRecord files with 10000 items per class for the training data and 1000 items per class as eval data.

This conversion process is described in more detail in the following.

The original QuickDraw data is formatted as `ndjson` files where each line contains a JSON object like the following:

```
{"word":"cat",
 "countrycode":"VE",
 "timestamp":"2017-03-02 23:25:10.07453 UTC",
 "recognized":true,
 "key_id":"5201136883597312",
 "drawing":[
   [

[130,113,99,109,76,64,55,48,48,51,59,86,133,154,170,203,214,217,215,208,186,
176,162,157,132],

[72,40,27,79,82,88,100,120,134,152,165,184,189,186,179,152,131,114,100,89,76
,0,31,65,70]
   ],[
     [76,28,7],
     [136,128,128]
   ],[
     [76,23,0],
     [160,164,175]
   ],[
     [87,52,37],
     [175,191,204]
   ],[
     [174,220,246,251],
     [134,132,136,139]
   ],[
     [175,255],
     [147,168]
   ],[
     [171,208,215],
     [164,198,210]
   ],[
     [130,110,108,111,130,139,139,119],
     [129,134,137,144,148,144,136,130]
   ],[
     [107,106],
     [96,113]
   ]
 ]
}
```

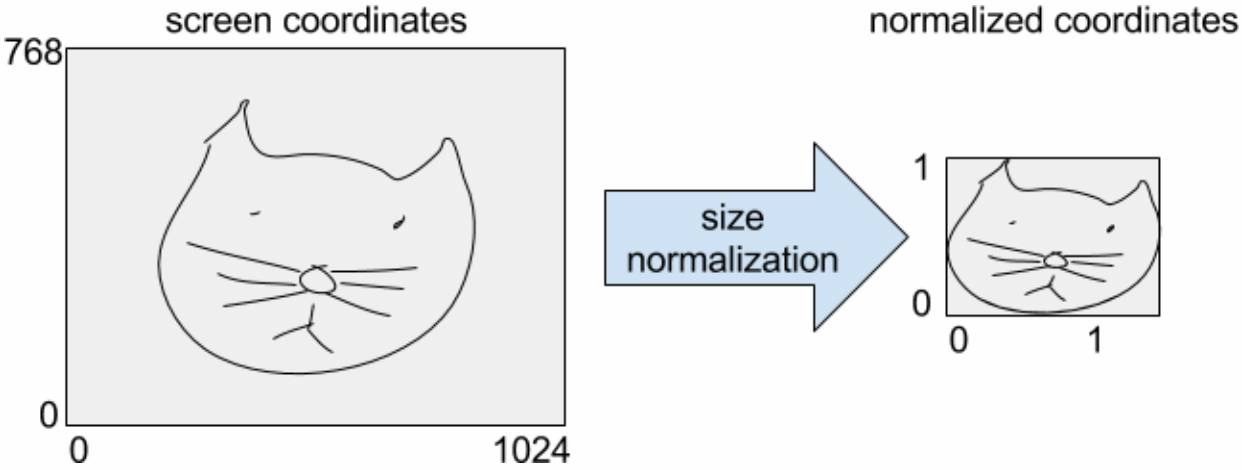

For our purpose of building a classifier we only care about the fields "`word`" and "`drawing`". While parsing the ndjson files, we process them line by line using a function that converts the strokes from the `drawing` field into a tensor of size `[number of points, 3]` containing the differences of consecutive points. This function also returns the class name as a string.

```
def parse_line(ndjson_line):
  """Parse an ndjson line and return ink (as np array) and classname."""
  sample = json.loads(ndjson_line)
  class_name = sample["word"]
  inkarray = sample["drawing"]
  stroke_lengths = [len(stroke[0]) for stroke in inkarray]
  total_points = sum(stroke_lengths)
  np_ink = np.zeros((total_points, 3), dtype=np.float32)
  current_t = 0
  for stroke in inkarray:
    for i in [0, 1]:
      np_ink[current_t:(current_t + len(stroke[0])), i] = stroke[i]
    current_t += len(stroke[0])
    np_ink[current_t - 1, 2] = 1  # stroke_end
  # Preprocessing.
  # 1. Size normalization.
  lower = np.min(np_ink[:, 0:2], axis=0)
  upper = np.max(np_ink[:, 0:2], axis=0)
  scale = upper - lower
  scale[scale == 0] = 1
  np_ink[:, 0:2] = (np_ink[:, 0:2] - lower) / scale
  # 2. Compute deltas.
  np_ink = np_ink[1:, 0:2] - np_ink[0:-1, 0:2]
  return np_ink, class_name
```

Since we want the data to be shuffled for writing we read from each of the category files in random order and write to a random shard.

For the training data we read the first 10000 items for each class and for the eval data we read the next 1000 items for each class.

This data is then reformatted into a tensor of shape `[num_training_samples, max_length, 3]`. Then we determine the bounding box of the original drawing in screen coordinates and normalize the size such that the drawing has unit height.



Finally, we compute the differences between consecutive points and store these as a `VarLenFeature` in a[tensorflow.Example](https://www.tensorflow.org/api_docs/python/tf/train/Example) under the key `ink`. In addition we store the `class_index` as a single entry `FixedLengthFeature`and the `shape` of the `ink` as a `FixedLengthFeature` of length 2.

Defining the model

To define the model we create a new `Estimator`. If you want to read more about estimators, we recommend this tutorial.

To build the model, we:

1. reshape the input back into the original shape - where the mini batch is padded to the maximal length of its contents. In addition to the ink data we also have the lengths for each example and the target class. This happens in the function `_get_input_tensors`.
2. pass the input through to a series of convolution layers in `_add_conv_layers`.
3. pass the output of the convolutions into a series of bidirectional LSTM layers in `_add_rnn_layers`. At the end of that, the outputs for each time step are summed up to have a compact, fixed length embedding of the input.
4. classify this embedding using a softmax layer in `_add_fc_layers`.

In code this looks like:

```
inks, lengths, targets = _get_input_tensors(features, targets)
convolved = _add_conv_layers(inks)
final_state = _add_rnn_layers(convolved, lengths)
logits =_add_fc_layers(final_state)
```

_get_input_tensors

To obtain the input features we first obtain the shape from the features dict and then create a 1D tensor of size `[batch_size]` containing the lengths of the input sequences. The ink is stored as a SparseTensor in the features dict which we convert into a dense tensor and then reshape to be `[batch_size, ?, 3]`. And finally, if targets were passed in we make sure they are stored as a 1D tensor of size `[batch_size]`

In code this looks like this:

```
shapes = features["shape"]
lengths = tf.squeeze(
    tf.slice(shapes, begin=[0, 0], size=[params["batch_size"], 1]))
inks = tf.reshape(
    tf.sparse_tensor_to_dense(features["ink"]),
    [params["batch_size"], -1, 3])
if targets is not None:
  targets = tf.squeeze(targets)
```

_add_conv_layers

The desired number of convolution layers and the lengths of the filters is configured through the parameters `num_conv`and `conv_len` in the `params` dict.

The input is a sequence where each point has dimensionality 3. We are going to use 1D convolutions where we treat the 3 input features as channels. That means that the input is a `[batch_size, length, 3]` tensor and the output will be a`[batch_size, length, number_of_filters]` tensor.

```
convolved = inks
for i in range(len(params.num_conv)):
  convolved_input = convolved
  if params.batch_norm:
    convolved_input = tf.layers.batch_normalization(
        convolved_input,
        training=(mode == tf.estimator.ModeKeys.TRAIN))
  # Add dropout layer if enabled and not first convolution layer.
  if i > 0 and params.dropout:
    convolved_input = tf.layers.dropout(
        convolved_input,
        rate=params.dropout,
        training=(mode == tf.estimator.ModeKeys.TRAIN))
  convolved = tf.layers.conv1d(
      convolved_input,
      filters=params.num_conv[i],
      kernel_size=params.conv_len[i],
      activation=None,
      strides=1,
      padding="same",
      name="conv1d_%d" % i)
return convolved, lengths
```

_add_rnn_layers

We pass the output from the convolutions into bidirectional LSTM layers for which we use a helper function from contrib.

```
outputs, _, _ = contrib_rnn.stack_bidirectional_dynamic_rnn(
    cells_fw=[cell(params.num_nodes) for _ in range(params.num_layers)],
    cells_bw=[cell(params.num_nodes) for _ in range(params.num_layers)],
    inputs=convolved,
    sequence_length=lengths,
    dtype=tf.float32,
    scope="rnn_classification")
```

see the code for more details and how to use `CUDA` accelerated implementations.

To create a compact, fixed-length embedding, we sum up the output of the LSTMs. We first zero out the regions of the batch where the sequences have no data.

```
mask = tf.tile(
    tf.expand_dims(tf.sequence_mask(lengths, tf.shape(outputs)[1]), 2),
    [1, 1, tf.shape(outputs)[2]])
zero_outside = tf.where(mask, outputs, tf.zeros_like(outputs))
outputs = tf.reduce_sum(zero_outside, axis=1)
```

_add_fc_layers

The embedding of the input is passed into a fully connected layer which we then use as a softmax layer.

```
tf.layers.dense(final_state, params.num_classes)
```

Loss, predictions, and optimizer

Finally, we need to add a loss, a training op, and predictions to create the `ModelFn`:

```
cross_entropy = tf.reduce_mean(
    tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels=targets, logits=logits))
# Add the optimizer.
train_op = tf.contrib.layers.optimize_loss(
    loss=cross_entropy,
    global_step=tf.train.get_global_step(),
    learning_rate=params.learning_rate,
    optimizer="Adam",
    # some gradient clipping stabilizes training in the beginning.
    clip_gradients=params.gradient_clipping_norm,
    summaries=["learning_rate", "loss", "gradients", "gradient_norm"])
predictions = tf.argmax(logits, axis=1)
return model_fn_lib.ModelFnOps(
    mode=mode,
    predictions={"logits": logits,
                 "predictions": predictions},
    loss=cross_entropy,
    train_op=train_op,
    eval_metric_ops={"accuracy": tf.metrics.accuracy(targets, predictions)})
```

Training and evaluating the model

To train and evaluate the model we can rely on the functionalities of the `Estimator` APIs and easily run training and evaluation with the `Experiment` APIs:

```
  estimator = tf.estimator.Estimator(
      model_fn=model_fn,
      model_dir=output_dir,
      config=config,
      params=model_params)
  # Train the model.
  tf.contrib.learn.Experiment(
      estimator=estimator,
      train_input_fn=get_input_fn(
          mode=tf.contrib.learn.ModeKeys.TRAIN,
          tfrecord_pattern=FLAGS.training_data,
          batch_size=FLAGS.batch_size),
      train_steps=FLAGS.steps,
      eval_input_fn=get_input_fn(
          mode=tf.contrib.learn.ModeKeys.EVAL,
          tfrecord_pattern=FLAGS.eval_data,
          batch_size=FLAGS.batch_size),
      min_eval_frequency=1000)
```

Note that this tutorial is just a quick example on a relatively small dataset to get you familiar with the APIs of recurrent neural networks and estimators. Such models can be even more powerful if you try them on a large dataset.

When training the model for 1M steps you can expect to get an accuracy of approximately of approximately 70% on the top-1 candidate. Note that this accuracy is sufficient to build the quickdraw game because of the game dynamics the user will be able to adjust their drawing until it is ready. Also, the game does not use the top-1 candidate only but accepts a drawing as correct if the target category shows up with a score better than a fixed threshold.

Simple Audio Recognition

This tutorial will show you how to build a basic speech recognition network that recognizes ten different words. It's important to know that real speech and audio recognition systems are much more complex, but like MNIST for images, it should give you a basic understanding of the techniques involved. Once you've completed this tutorial, you'll have a model that tries to classify a one second audio clip as either silence, an unknown word, "yes", "no", "up", "down", "left", "right", "on", "off", "stop", or "go". You'll also be able to take this model and run it in an Android application.

Preparation

You should make sure you have TensorFlow installed, and since the script downloads over 1GB of training data, you'll need a good internet connection and enough free space on your machine. The training process itself can take several hours, so make sure you have a machine available for that long.

Training

To begin the training process, go to the TensorFlow source tree and run:

```
python tensorflow/examples/speech_commands/train.py
```

The script will start off by downloading the Speech Commands dataset, which consists of 65,000 WAVE audio files of people saying thirty different words. This data was collected by Google and released under a CC BY license, and you can help improve it by contributing five minutes of your own voice. The archive is over 1GB, so this part may take a while, but you should see progress logs, and once it's been downloaded once you won't need to do this step again.

Once the downloading has completed, you'll see logging information that looks like this:

```
I0730 16:53:44.766740   55030 train.py:176] Training from step: 1
I0730 16:53:47.289078   55030 train.py:217] Step #1: rate 0.001000, accuracy
7.0%, cross entropy 2.611571
```

This shows that the initialization process is done and the training loop has begun. You'll see that it outputs information for every training step. Here's a break down of what it means:

`Step #1` shows that we're on the first step of the training loop. In this case there are going to be 18,000 steps in total, so you can look at the step number to get an idea of how close it is to finishing.

`rate 0.001000` is the learning rate that's controlling the speed of the network's weight updates. Early on this is a comparatively high number (0.001), but for later training cycles it will be reduced 10x, to 0.0001.

`accuracy 7.0%` is the how many classes were correctly predicted on this training step. This value will often fluctuate a lot, but should increase on average as training progresses. The model outputs an array of numbers, one for each label, and each number is the predicted likelihood of the input being that class. The predicted label is picked by choosing the entry with the highest score. The scores are always between zero and one, with higher values representing more confidence in the result.

`cross entropy 2.611571` is the result of the loss function that we're using to guide the training process. This is a score that's obtained by comparing the vector of scores from the current training run to the correct labels, and this should trend downwards during training.

After a hundred steps, you should see a line like this:

```
I0730 16:54:41.813438 55030 train.py:252] Saving to
"/tmp/speech_commands_train/conv.ckpt-100"
```

This is saving out the current trained weights to a checkpoint file. If your training script gets interrupted, you can look for the last saved checkpoint and then restart the script with `--start_checkpoint=/tmp/speech_commands_train/conv.ckpt-100` as a command line argument to start from that point.

Confusion Matrix

After four hundred steps, this information will be logged:

```
I0730 16:57:38.073667   55030 train.py:243] Confusion Matrix:
 [[258   0   0   0   0   0   0   0   0   0   0   0]
 [  7   6  26  94   7  49   1  15  40   2   0  11]
 [ 10   1 107  80  13  22   0  13  10   1   0   4]
 [  1   3  16 163   6  48   0   5  10   1   0  17]
 [ 15   1  17 114  55  13   0   9  22   5   0   9]
 [  1   1   6  97   3  87   1  12  46   0   0  10]
 [  8   6  86  84  13  24   1   9   9   1   0   6]
 [  9   3  32 112   9  26   1  36  19   0   0   9]
 [  8   2  12  94   9  52   0   6  72   0   0   2]
 [ 16   1  39  74  29  42   0   6  37   9   0   3]
 [ 15   6  17  71  50  37   0   6  32   2   1   9]
 [ 11   1   6 151   5  42   0   8  16   0   0  20]]
```

The first section is a confusion matrix. To understand what it means, you first need to know the labels being used, which in this case are "*silence*", "*unknown*", "yes", "no", "up", "down", "left", "right", "on", "off", "stop", and "go". Each column represents a set of samples that were predicted to be each label, so the first column represents all the clips that were predicted to be silence, the second all those that were predicted to be unknown words, the third "yes", and so on.

Each row represents clips by their correct, ground truth labels. The first row is all the clips that were silence, the second clips that were unknown words, the third "yes", etc.

This matrix can be more useful than just a single accuracy score because it gives a good summary of what mistakes the network is making. In this example you can see that all of the entries in the first row are zero, apart from the initial one. Because the first row is all the clips that are actually silence, this means that none of them were mistakenly labeled as words, so we have no false negatives for silence. This shows the network is already getting pretty good at distinguishing silence from words.

If we look down the first column though, we see a lot of non-zero values. The column represents all the clips that were predicted to be silence, so positive numbers outside of the first cell are errors. This means that some clips of real spoken words are actually being predicted to be silence, so we do have quite a few false positives.

A perfect model would produce a confusion matrix where all of the entries were zero apart from a diagonal line through the center. Spotting deviations from that pattern can help you figure out how the model is most easily confused, and once you've identified the problems you can address them by adding more data or cleaning up categories.

Validation

After the confusion matrix, you should see a line like this:

```
I0730 16:57:38.073777 55030 train.py:245] Step 400: Validation accuracy = 26.3% (N=3093)
```

It's good practice to separate your data set into three categories. The largest (in this case roughly 80% of the data) is used for training the network, a smaller set (10% here, known as "validation") is reserved for evaluation of the accuracy during training, and another set (the last 10%, "testing") is used to evaluate the accuracy once after the training is complete.

The reason for this split is that there's always a danger that networks will start memorizing their inputs during training. By keeping the validation set separate, you can ensure that the model works with data it's never seen before. The testing set is an additional safeguard to make sure that you haven't just been tweaking your model in a way that happens to work for both the training and validation sets, but not a broader range of inputs.

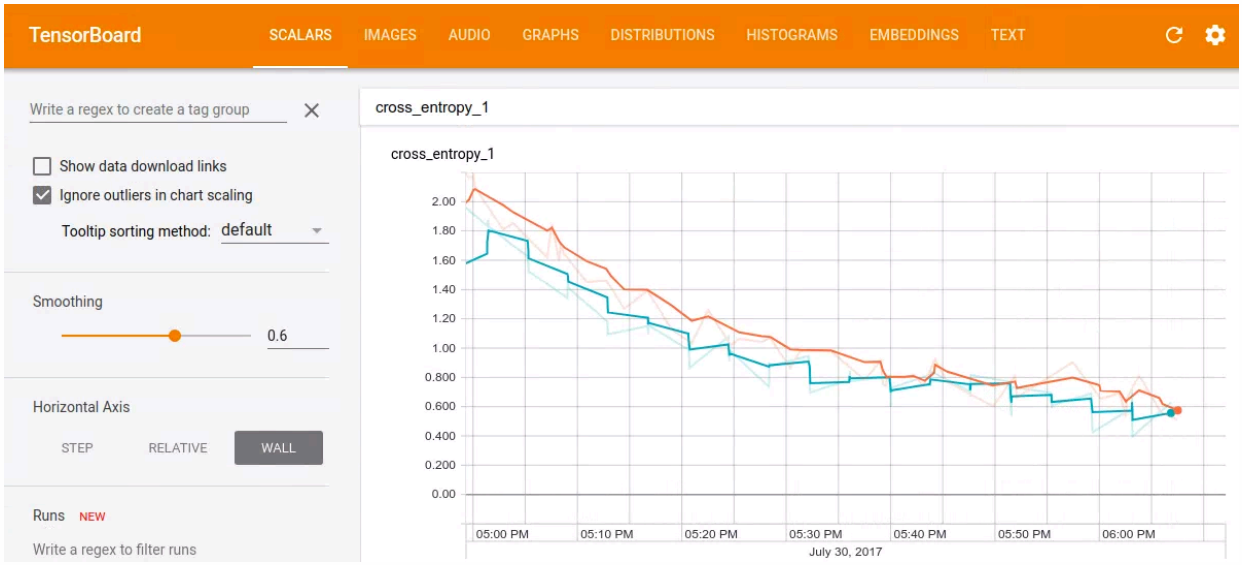
The training script automatically separates the data set into these three categories, and the logging line above shows the accuracy of model when run on the validation set. Ideally, this should stick fairly close to the training accuracy. If the training accuracy increases but the validation doesn't, that's a sign that overfitting is occurring, and your model is only learning things about the training clips, not broader patterns that generalize.

Tensorboard

A good way to visualize how the training is progressing is using Tensorboard. By default, the script saves out events to /tmp/retrain_logs, and you can load these by running:

```
tensorboard --logdir /tmp/retrain_logs
```

Then navigate to http://localhost:6006 in your browser, and you'll see charts and graphs showing your models progress.



Training Finished

After a few hours of training (depending on your machine's speed), the script should have completed all 18,000 steps. It will print out a final confusion matrix, along with an accuracy score, all run on the testing set. With the default settings, you should see an accuracy of between 85% and 90%.

Because audio recognition is particularly useful on mobile devices, next we'll export it to a compact format that's easy to work with on those platforms. To do that, run this command line:

```
python tensorflow/examples/speech_commands/freeze.py \
--start_checkpoint=/tmp/speech_commands_train/conv.ckpt-18000 \
--output_file=/tmp/my_frozen_graph.pb
```

Once the frozen model has been created, you can test it with the `label_wav.py` script, like this:

```
python tensorflow/examples/speech_commands/label_wav.py \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_dataset/left/a5d485dc_nohash_0.wav
```

This should print out three labels:

```
left (score = 0.81477)
right (score = 0.14139)
_unknown_ (score = 0.03808)
```

Hopefully "left" is the top score since that's the correct label, but since the training is random it may not for the first file you try. Experiment with some of the other .wav files in that same folder to see how well it does.

The scores are between zero and one, and higher values mean the model is more confident in its prediction.

Running the Model in an Android App

The easiest way to see how this model works in a real application is to download the prebuilt Android demo applicationsand install them on your phone. You'll see 'TF Speech' appear in your app list, and opening it will show you the same list of action words we've just trained our model on, starting with "Yes" and "No". Once you've given the app permission to use the microphone, you should be able to try saying those words and see them highlighted in the UI when the model recognizes one of them.

You can also build this application yourself, since it's open source and available as part of the TensorFlow repository on github. By default it downloads a pretrained model from tensorflow.org, but you can easily replace it with a model you've trained yourself. If you do this, you'll need to make sure that the constants in the main SpeechActivity Java source file like `SAMPLE_RATE` and `SAMPLE_DURATION` match any changes you've made to the defaults while training. You'll also see that there's a Java version of the RecognizeCommands module that's very similar to the C++ version in this tutorial. If you've tweaked parameters for that, you can also update them in SpeechActivity to get the same results as in your server testing.

The demo app updates its UI list of results automatically based on the labels text file you copy into assets alongside your frozen graph, which means you can easily try out different models without needing to make any code changes. You will need to updaye `LABEL_FILENAME` and `MODEL_FILENAME` to point to the files you've added if you change the paths though.

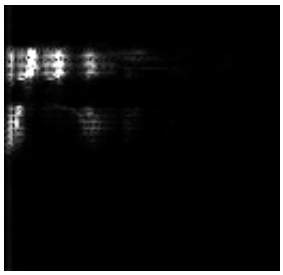
How does this Model Work?

The architecture used in this tutorial is based on some described in the paper Convolutional Neural Networks for Small-footprint Keyword Spotting. It was chosen because it's comparatively simple, quick to train, and easy to understand, rather than being state of the art. There are lots of different approaches to building neural network models to work with audio, including recurrent networks or dilated (atrous) convolutions. This tutorial is based on the kind of convolutional network that will feel very familiar to anyone who's worked with image recognition. That may seem surprising at first though, since audio is inherently a one-dimensional continuous signal across time, not a 2D spatial problem.

We solve that issue by defining a window of time we believe our spoken words should fit into, and converting the audio signal in that window into an image. This is done by grouping the incoming audio samples into short segments, just a few milliseconds long, and calculating the strength of the frequencies across a set of bands. Each set of frequency strengths from a segment is treated as a vector of numbers, and those vectors are arranged in time order to form a two-dimensional array. This array of values can then be treated like a single-channel image, and is known as a spectrogram. If you want to view what kind of image an audio sample produces, you can run the `wav_to_spectrogram tool:

```
bazel run tensorflow/examples/wav_to_spectrogram:wav_to_spectrogram -- \
--input_wav=/tmp/speech_dataset/happy/ab00c4b2_nohash_0.wav \
--output_png=/tmp/spectrogram.png
```

If you open up `/tmp/spectrogram.png` you should see something like this:



Because of TensorFlow's memory order, time in this image is increasing from top to bottom, with frequencies going from left to right, unlike the usual convention for spectrograms where time is left to right. You should be able to see a couple of distinct parts, with the first syllable "Ha" distinct from "ppy".

Because the human ear is more sensitive to some frequencies than others, it's been traditional in speech recognition to do further processing to this representation to turn it into a set of Mel-Frequency Cepstral Coefficients, or MFCCs for short. This is also a two-dimensional, one-channel representation so it can be treated like an image too. If you're targeting general sounds rather than speech you may find you can skip this step and operate directly on the spectrograms.

The image that's produced by these processing steps is then fed into a multi-layer convolutional neural network, with a fully-connected layer followed by a softmax at the end. You can see the definition of this portion intensorflow/examples/speech_commands/models.py.

Streaming Accuracy

Most audio recognition applications need to run on a continuous stream of audio, rather than on individual clips. A typical way to use a model in this environment is to apply it repeatedly at different offsets in time and average the results over a short window to produce a smoothed prediction. If you think of the input as an image, it's continuously scrolling along the time axis. The words we want to recognize can start at any time, so we need to take a series of snapshots to have a chance of having an alignment that captures most of the utterance in the time window we feed into the model. If we sample at a high enough rate, then we have a good chance of capturing the word in multiple windows, so averaging the results improves the overall confidence of the prediction.

For an example of how you can use your model on streaming data, you can look at test_streaming_accuracy.cc. This uses the RecognizeCommands class to run through a long-form input audio, try to spot words, and compare those predictions against a ground truth list of labels and times. This makes it a good example of applying a model to a stream of audio signals over time.

You'll need a long audio file to test it against, along with labels showing where each word was spoken. If you don't want to record one yourself, you can generate some synthetic test data using the `generate_streaming_test_wav` utility. By default this will create a ten minute .wav file with words roughly every three seconds, and a text file containing the ground truth of when each word was spoken. These words are pulled from the test portion of your current dataset, mixed in with background noise. To run it, use:

```
bazel run tensorflow/examples/speech_commands:generate_streaming_test_wav
```

This will save a .wav file to `/tmp/speech_commands_train/streaming_test.wav`, and a text file listing the labels to`/tmp/speech_commands_train/streaming_test_labels.txt`. You can then run accuracy testing with:

```
bazel run tensorflow/examples/speech_commands:test_streaming_accuracy -- \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_commands_train/streaming_test.wav \
--ground_truth=/tmp/speech_commands_train/streaming_test_labels.txt \
--verbose
```

This will output information about the number of words correctly matched, how many were given the wrong labels, and how many times the model triggered when there was no real word spoken. There are various parameters that control how the signal averaging works, including `--average_window_ms` which sets the length of time to average results over, `--clip_stride_ms` which is the time between applications of the model, `--suppression_ms` which stops subsequent word detections from triggering for a certain time after an initial one is found, and `--detection_threshold`, which controls how high the average score must be before it's considered a solid result.

You'll see that the streaming accuracy outputs three numbers, rather than just the one metric used in training. This is because different applications have varying requirements, with some being able to tolerate frequent incorrect results as long as real words are found (high recall), while others very focused on ensuring the predicted labels are highly likely to be correct even if some aren't detected (high precision). The numbers from the tool give you an idea of how your model will perform in an application, and you can try tweaking the signal averaging parameters to tune it to give the kind of performance you want. To understand what the right parameters are for your application, you can look at generating an ROC curve to help you understand the tradeoffs.

RecognizeCommands

The streaming accuracy tool uses a simple decoder contained in a small C++ class called RecognizeCommands. This class is fed the output of running the TensorFlow model over time, it averages the signals, and returns information about a label when it has enough evidence to think that a recognized word has been found. The implementation is fairly small, just keeping track of the last few predictions and averaging them, so it's easy to port to other platforms and languages as needed. For example, it's convenient to do something similar at the Java level on Android, or Python on the Raspberry Pi. As long as these implementations share the same logic, you can tune the parameters that control the averaging using the streaming test tool, and then transfer them over to your application to get similar results.

Advanced Training

The defaults for the training script are designed to produce good end to end results in a comparatively small file, but there are a lot of options you can change to customize the results for your own requirements.

Custom Training Data

By default the script will download the Speech Commands dataset, but you can also supply your own training data. To train on your own data, you should make sure that you have at least several hundred recordings of each sound you would like to recognize, and arrange them into folders by class. For example, if you were trying to recognize dog barks from cat miaows, you would create a root folder called `animal_sounds`, and then within that two sub-folders called `bark` and `miaow`. You would then organize your audio files into the appropriate folders.

To point the script to your new audio files, you'll need to set `--data_url=` to disable downloading of the Speech Commands dataset, and `--data_dir=/your/data/folder/` to find the files you've just created.

The files themselves should be 16-bit little-endian PCM-encoded WAVE format. The sample rate defaults to 16,000, but as long as all your audio is consistently the same rate (the script doesn't support resampling) you can change this with the `--sample_rate` argument. The clips should also all be roughly the same duration. The default expected duration is one second, but you can set this with the `--clip_duration_ms` flag. If you have clips with variable amounts of silence at the start, you can look at word alignment tools to standardize them (here's a quick and dirty approach you can use too).

One issue to watch out for is that you may have very similar repetitions of the same sounds in your dataset, and these can give misleading metrics if they're spread across your training, validation, and test sets. For example, the Speech Commands set has people repeating the same word multiple times. Each one of those repetitions is likely to be pretty close to the others, so if training was overfitting and memorizing one, it could perform unrealistically well when it saw a very similar copy in the test set. To avoid this danger, Speech Commands trys to ensure that all clips featuring the same word spoken by a single person are put into the same partition. Clips are assigned to training, test, or validation sets based on a hash of their filename, to ensure that the assignments remain steady even as new clips are added and avoid any training samples migrating into the other sets. To make sure that all a given speaker's words are in the same bucket, the hashing function ignores anything in a filename after '*nohash*' when calculating the assignments. This means that if you have file names like `pete_nohash_0.wav` and `pete_nohash_1.wav`, they're guaranteed to be in the same set.

Unknown Class

It's likely that your application will hear sounds that aren't in your training set, and you'll want the model to indicate that it doesn't recognize the noise in those cases. To help the network learn what sounds to ignore, you need to provide some clips of audio that are neither of your classes. To do this, you'd create `quack`, `oink`, and `moo` subfolders and populate them with noises from other animals your users might encounter. The `--wanted_words` argument to the script defines which classes you care about, all the others mentioned in subfolder names will be used to populate an `_unknown_` class during training. The Speech Commands dataset has twenty words in its unknown classes, including the digits zero through nine and random names like "Sheila".

By default 10% of the training examples are picked from the unknown classes, but you can control this with the `--unknown_percentage` flag. Increasing this will make the model less likely to mistake unknown words for wanted ones, but making it too large can backfire as the model might decide it's safest to categorize all words as unknown!

Background Noise

Real applications have to recognize audio even when there are other irrelevant sounds happening in the environment. To build a model that's robust to this kind of interference, we need to train against recorded audio with similar properties. The files in the Speech Commands dataset were captured on a variety of devices by users in many different environments, not in a studio, so that helps add some realism to the training. To add even more, you can mix in random segments of environmental audio to the training inputs. In the Speech Commands set there's a special folder called `_background_noise_` which contains minute-long WAVE files with white noise and recordings of machinery and everyday household activity.

Small snippets of these files are chosen at random and mixed at a low volume into clips during training. The loudness is also chosen randomly, and controlled by the `--background_volume` argument as a proportion where 0 is silence, and 1 is full volume. Not all clips have background added, so the `--background_frequency` flag controls what proportion have them mixed in.

Your own application might operate in its own environment with different background noise patterns than these defaults, so you can supply your own audio clips in the `_background_noise_` folder. These should be the same sample rate as your main dataset, but much longer in duration so that a good set of random segments can be selected from them.

Silence

In most cases the sounds you care about will be intermittent and so it's important to know when there's no matching audio. To support this, there's a special `_silence_` label that indicates when the model detects nothing interesting. Because there's never complete silence in real environments, we actually have to supply examples with quiet and irrelevant audio. For this, we reuse the `_background_noise_` folder that's also mixed in to real clips, pulling short sections of the audio data and feeding those in with the ground truth class of `_silence_`. By default 10% of the training data is supplied like this, but the `--silence_percentage` can be used to control the proportion. As with unknown words, setting this higher can weight the model results in favor of true positives for silence, at the expense of false negatives for words, but too large a proportion can cause it to fall into the trap of always guessing silence.

Time Shifting

Adding in background noise is one way of distorting the training data in a realistic way to effectively increase the size of the dataset, and so increase overall accuracy, and time shifting is another. This involves a random offset in time of the training sample data, so that a small part of the start or end is cut off and the opposite section is padded with zeroes. This mimics the natural variations in starting time in the training data, and is controlled with the `--time_shift_ms` flag, which defaults to 100ms. Increasing this value will provide more variation, but at the risk of cutting off important parts of the audio. A related way of augmenting the data with realistic distortions is by using time stretching and pitch scaling, but that's outside the scope of this tutorial.

Customizing the Model

The default model used for this script is pretty large, taking over 800 million FLOPs for each inference and using 940,000 weight parameters. This runs at usable speeds on desktop machines or modern phones, but it involves too many calculations to run at interactive speeds on devices with more limited resources. To support these use cases, there's a couple of alternatives available:

low_latency_conv Based on the 'cnn-one-fstride4' topology described in the Convolutional Neural Networks for Small-footprint Keyword Spotting paper. The accuracy is slightly lower than 'conv' but the number of weight parameters is about the same, and it only needs 11 million FLOPs to run one prediction, making it much faster.

To use this model, you specify `--model_architecture=low_latency_conv` on the command line. You'll also need to update the training rates and the number of steps, so the full command will look like:

```
python tensorflow/examples/speech_commands/train \
--model_architecture=low_latency_conv \
--how_many_training_steps=20000,6000 \
--learning_rate=0.01,0.001
```

This asks the script to train with a learning rate of 0.01 for 20,000 steps, and then do a fine-tuning pass of 6,000 steps with a 10x smaller rate.

low_latency_svdf Based on the topology presented in the Compressing Deep Neural Networks using a Rank-Constrained Topology paper. The accuracy is also lower than 'conv' but it only uses about 750 thousand parameters, and most significantly, it allows for an optimized execution at test time (i.e. when you will actually use it in your application), resulting in 750 thousand FLOPs.

To use this model, you specify `--model_architecture=low_latency_svdf` on the command line, and update the training rates and the number of steps, so the full command will look like:

```
python tensorflow/examples/speech_commands/train \
--model_architecture=low_latency_svdf \
--how_many_training_steps=100000,35000 \
--learning_rate=0.01,0.005
```

Note that despite requiring a larger number of steps than the previous two topologies, the reduced number of computations means that training should take about the same time, and at the end reach an accuracy of around 85%. You can also further tune the topology fairly easily for computation and accuracy by changing these parameters in the SVDF layer:

- rank - The rank of the approximation (higher typically better, but results in more computation).
- num_units - Similar to other layer types, specifies the number of nodes in the layer (more nodes better quality, and more computation).

Regarding runtime, since the layer allows optimizations by caching some of the internal neural network activations, you need to make sure to use a consistent stride (e.g. 'clip_stride_ms' flag) both when you freeze the graph, and when executing the model in streaming mode (e.g. test_streaming_accuracy.cc).

Other parameters to customize If you want to experiment with customizing models, a good place to start is by tweaking the spectrogram creation parameters. This has the effect of altering the size of the input image to the model, and the creation code in models.py will adjust the number of computations and weights automatically to fit with different dimensions. If you make the input smaller, the model will need fewer computations to process it, so it can be a great way to trade off some accuracy for improved latency. The `--window_stride_ms` controls how far apart each frequency analysis sample is from the previous. If you increase this value, then fewer samples will be taken for a given duration, and the time axis of the input will shrink. The `--dct_coefficient_count` flag controls how many buckets are used for the frequency counting, so reducing this will shrink the input in the other dimension. The `--window_size_ms` argument doesn't affect the size, but does control how wide the area used to calculate the frequencies is for each sample. Reducing the duration of the training samples, controlled by `--clip_duration_ms`, can also help if the sounds you're looking for are short, since that also reduces the time dimension of the input. You'll need to make sure that all your training data contains the right audio in the initial portion of the clip though.

If you have an entirely different model in mind for your problem, you may find that you can plug it into models.py and have the rest of the script handle all of the preprocessing and training mechanics. You would add a new clause to `create_model`, looking for the name of your architecture and then calling a model creation function. This function is given the size of the spectrogram input, along with other model information, and is expected to create TensorFlow ops to read that in and produce an output prediction vector, and a placeholder to control the dropout rate. The rest of the script will handle integrating this model into a larger graph doing the input calculations and applying softmax and a loss function to train it.

One common problem when you're adjusting models and training hyper-parameters is that not-a-number values can creep in, thanks to numerical precision issues. In general you can solve these by reducing the magnitude of things like learning rates and weight initialization functions, but if they're persistent you can enable the `--check_nans` flag to track down the source of the errors. This will insert check ops between most regular operations in TensorFlow, and abort the training process with a useful error message when they're encountered.

TensorFlow Linear Model Tutorial

In this tutorial, we will use the tf.estimator API in TensorFlow to solve a binary classification problem: Given census data about a person such as age, education, marital status, 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 if you haven't already.
2. Download the tutorial code.
3. Execute the data download script we provide to you:

```
$ python data_download.py
```

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

```
$ python wide_deep.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. We have provided data_download.py which downloads the code and performs some additional cleanup.

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. For reference, see `input_fn` in wide_deep.py.

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 relationship status of a person (wife, husband, unmarried, 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.

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). Final weight 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.estimator 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.estimator.Estimator methods such as `train` and `evaluate`. The purpose of this function is to construct the input data, which is represented in the form of `tf.Tensor`s or `tf.SparseTensor`s. In more detail, the input builder function returns the following as a pair:

1. `features`: A dict from feature column names to `Tensors` or `SparseTensors`.
2. `labels`: A `Tensor` containing the label column.

The keys of the `features` will be used to construct columns in the next section. Because we want to call the `train`and `evaluate` methods with different data, we define a method that returns an input function based on the given data. Note that the returned input function 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`).

Each continuous column in the train or test data 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. Our `input_fn` uses the `tf.data` API, which makes it easy to apply transformations to our dataset:

```
def input_fn(data_file, num_epochs, shuffle, batch_size):
  """Generate an input function for the Estimator."""
  assert tf.gfile.Exists(data_file), (
      '%s not found. Please make sure you have either run data_download.py
or '
      'set both arguments --train_data and --test_data.' % data_file)

  def parse_csv(value):
    print('Parsing', data_file)
    columns = tf.decode_csv(value, record_defaults=_CSV_COLUMN_DEFAULTS)
    features = dict(zip(_CSV_COLUMNS, columns))
    labels = features.pop('income_bracket')
    return features, tf.equal(labels, '>50K')

  # Extract lines from input files using the Dataset API.
  dataset = tf.data.TextLineDataset(data_file)

  if shuffle:
    dataset = dataset.shuffle(buffer_size=_SHUFFLE_BUFFER)

  dataset = dataset.map(parse_csv, num_parallel_calls=5)

  # We call repeat after shuffling, rather than before, to prevent separate
  # epochs from blending together.
  dataset = dataset.repeat(num_epochs)
  dataset = dataset.batch(batch_size)

  iterator = dataset.make_one_shot_iterator()
  features, labels = iterator.get_next()
  return features, labels
```

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 `CategoricalColumn` using the tf.feature_column API. If you know the set of all possible feature values of a column and there are only a few of them, you can use `categorical_column_with_vocabulary_list`. Each key in the list will get assigned an auto-incremental ID starting from 0. For example, for the `relationship` column we can assign the feature string "Husband" to an integer ID of 0 and "Not-in-family" to 1, etc., by doing:

```
relationship = tf.feature_column.categorical_column_with_vocabulary_list(
    'relationship', [
        'Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried',
        'Other-relative'])
```

What if we don't know the set of possible values in advance? Not a problem. We can use `categorical_column_with_hash_bucket` instead:

```
occupation = tf.feature_column.categorical_column_with_hash_bucket(
    'occupation', hash_bucket_size=1000)
```

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

ID	Feature
...	
9	`"Machine-op-inspct"`
...	
103	`"Farming-fishing"`
...	
375	`"Protective-serv"`
...	

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:

```
education = tf.feature_column.categorical_column_with_vocabulary_list(
    'education', [
        'Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',
        'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',
        '5th-6th', '10th', '1st-4th', 'Preschool', '12th'])

marital_status = tf.feature_column.categorical_column_with_vocabulary_list(
    'marital_status', [
        'Married-civ-spouse', 'Divorced', 'Married-spouse-absent',
        'Never-married', 'Separated', 'Married-AF-spouse', 'Widowed'])

relationship = tf.feature_column.categorical_column_with_vocabulary_list(
    'relationship', [
        'Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried',
        'Other-relative'])

workclass = tf.feature_column.categorical_column_with_vocabulary_list(
    'workclass', [
        'Self-emp-not-inc', 'Private', 'State-gov', 'Federal-gov',
        'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'])

# To show an example of hashing:
occupation = tf.feature_column.categorical_column_with_hash_bucket(
    'occupation', hash_bucket_size=1000)
```

Base Continuous Feature Columns

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

```
age = tf.feature_column.numeric_column('age')
education_num = tf.feature_column.numeric_column('education_num')
capital_gain = tf.feature_column.numeric_column('capital_gain')
capital_loss = tf.feature_column.numeric_column('capital_loss')
hours_per_week = tf.feature_column.numeric_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.feature_column.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.feature_column.crossed_column(
    ['education', 'occupation'], hash_bucket_size=1000)
```

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.feature_column.crossed_column(
    [age_buckets, 'education', 'occupation'], hash_bucket_size=1000)
```

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:

- `CategoricalColumn`
- `NumericColumn`
- `BucketizedColumn`
- `CrossedColumn`

All of these are subclasses of the abstract `FeatureColumn` class, and can be added to the `feature_columns` field of a model:

```
base_columns = [
    education, marital_status, relationship, workclass, occupation,
    age_buckets,
]
crossed_columns = [
    tf.feature_column.crossed_column(
        ['education', 'occupation'], hash_bucket_size=1000),
    tf.feature_column.crossed_column(
        [age_buckets, 'education', 'occupation'], hash_bucket_size=1000),
]

model_dir = tempfile.mkdtemp()
model = tf.estimator.LinearClassifier(
    model_dir=model_dir, feature_columns=base_columns + crossed_columns)
```

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 single command using the tf.estimator API:

```
model.train(input_fn=lambda: input_fn(train_data, num_epochs, True,
batch_size))
```

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

```
results = model.evaluate(input_fn=lambda: input_fn(
    test_data, 1, False, batch_size))
for key in sorted(results):
  print('%s: %s' % (key, results[key]))
```

The first line of the final 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 and set the `model_type` flag 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:

```
model = tf.estimator.LinearClassifier(
    model_dir=model_dir, feature_columns=base_columns + crossed_columns,
    optimizer=tf.train.FtrlOptimizer(
        learning_rate=0.1,
        l1_regularization_strength=1.0,
        l2_regularization_strength=1.0))
```

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:

$$P(Y=1 \mid x)=11+\exp(-(wTx+b))$$

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.

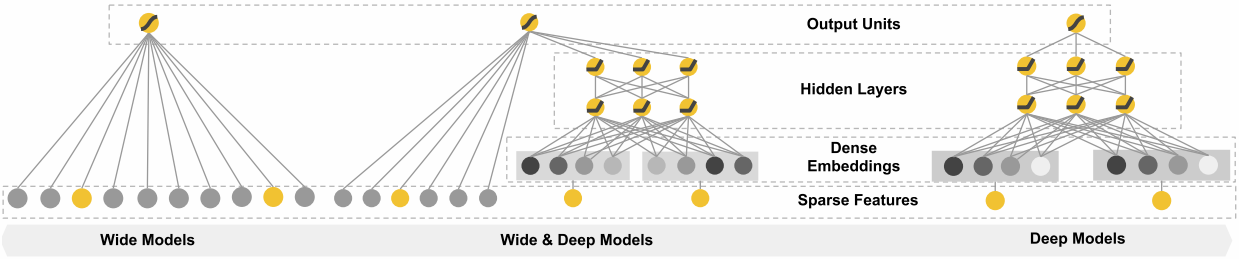
Learn Deeper

If you're interested in learning more, check out our Wide & Deep Learning Tutorial where we'll show you how to combine the strengths of linear models and deep neural networks by jointly training them using the tf.estimator API.

TensorFlow Wide & Deep Learning Tutorial

In the previous TensorFlow Linear Model Tutorial, 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. 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.estimator 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.



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.estimator 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 if you haven't already.
2. Download the tutorial code.
3. Execute the data download script we provide to you:

```
$ python data_download.py
```

4. Execute the tutorial code with the following command to train the wide and deep model described in this tutorial:

```
$ python wide_deep.py
```

Read on to find out how this code builds its 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

# Continuous columns
age = tf.feature_column.numeric_column('age')
education_num = tf.feature_column.numeric_column('education_num')
capital_gain = tf.feature_column.numeric_column('capital_gain')
capital_loss = tf.feature_column.numeric_column('capital_loss')
hours_per_week = tf.feature_column.numeric_column('hours_per_week')

education = tf.feature_column.categorical_column_with_vocabulary_list(
    'education', [
        'Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',
        'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',
        '5th-6th', '10th', '1st-4th', 'Preschool', '12th'])

marital_status = tf.feature_column.categorical_column_with_vocabulary_list(
    'marital_status', [
        'Married-civ-spouse', 'Divorced', 'Married-spouse-absent',
        'Never-married', 'Separated', 'Married-AF-spouse', 'Widowed'])

relationship = tf.feature_column.categorical_column_with_vocabulary_list(
    'relationship', [
        'Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried',
        'Other-relative'])

workclass = tf.feature_column.categorical_column_with_vocabulary_list(
    'workclass', [
        'Self-emp-not-inc', 'Private', 'State-gov', 'Federal-gov',
        'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'])

# To show an example of hashing:
occupation = tf.feature_column.categorical_column_with_hash_bucket(
    'occupation', hash_bucket_size=1000)

# Transformations.
age_buckets = tf.feature_column.bucketized_column(
    age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
```

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:

```
base_columns = [
    education, marital_status, relationship, workclass, occupation,
    age_buckets,
]

crossed_columns = [
    tf.feature_column.crossed_column(
        ['education', 'occupation'], hash_bucket_size=1000),
    tf.feature_column.crossed_column(
        [age_buckets, 'education', 'occupation'], hash_bucket_size=1000),
]
```

You can also see the TensorFlow Linear Model Tutorial for more details.

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 or Word embedding on Wikipedia.

Another way to represent categorical columns to feed into a neural network is via a one-hot or multi-hot representation. This is often appropriate for categorical columns with only a few possible values. As an example of a one-hot representation, for the relationship column, `"Husband"` can be represented as [1, 0, 0, 0, 0, 0], and `"Not-in-family"` as [0, 1, 0, 0, 0, 0], etc. This is a fixed representation, whereas embeddings are more flexible and calculated at training time.

We'll configure the embeddings for the categorical columns using `embedding_column`, and concatenate them with the continuous columns. We also use `indicator_column` to create multi-hot representations of some categorical columns.

```
deep_columns = [
    age,
    education_num,
    capital_gain,
    capital_loss,
    hours_per_week,
    tf.feature_column.indicator_column(workclass),
    tf.feature_column.indicator_column(education),
    tf.feature_column.indicator_column(marital_status),
    tf.feature_column.indicator_column(relationship),
    # To show an example of embedding
    tf.feature_column.embedding_column(occupation, dimension=8),
]
```

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`:

```
model = tf.estimator.DNNLinearCombinedClassifier(
    model_dir='/tmp/census_model',
    linear_feature_columns=base_columns + crossed_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. See `data_download.py` as well as `input_fn` within `wide_deep.py`.

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

```
# Train and evaluate the model every `FLAGS.epochs_per_eval` epochs.
for n in range(FLAGS.train_epochs // FLAGS.epochs_per_eval):
  model.train(input_fn=lambda: input_fn(
      FLAGS.train_data, FLAGS.epochs_per_eval, True, FLAGS.batch_size))

  results = model.evaluate(input_fn=lambda: input_fn(
      FLAGS.test_data, 1, False, FLAGS.batch_size))

  # Display evaluation metrics
  print('Results at epoch', (n + 1) * FLAGS.epochs_per_eval)
  print('-' * 30)

  for key in sorted(results):
    print('%s: %s' % (key, results[key]))
```

The final output accuracy should be somewhere around 85.5%. If you'd like to see a working end-to-end example, you can download our example code.

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 for more ideas about how to apply Wide & Deep Learning in real-world large-scale machine learning problems.

Vector Representations of Words

ContentsHighlightsMotivation: Why Learn Word Embeddings?Scaling up with Noise-Contrastive TrainingThe Skip-gram Model

In this tutorial we look at the word2vec model by Mikolov et al. 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.

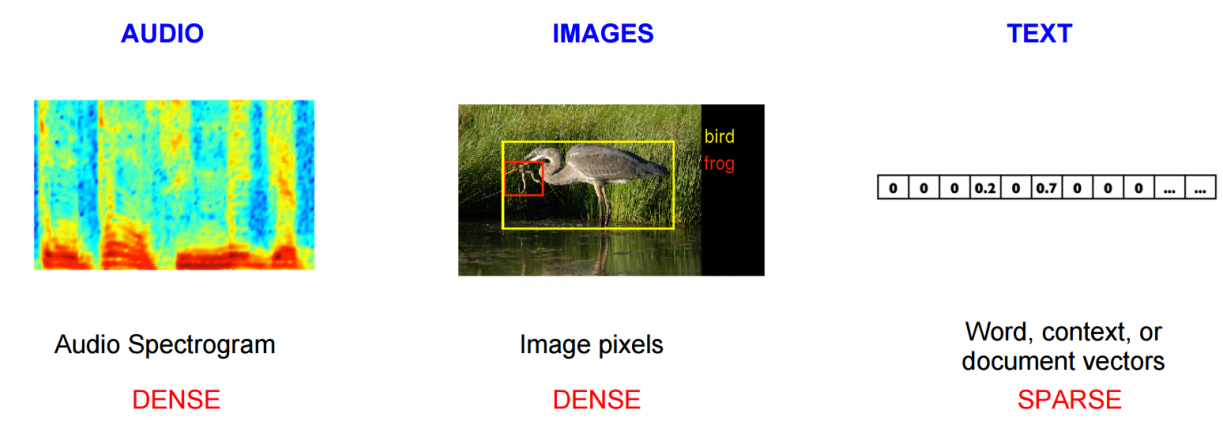
- We start by giving the motivation for why we would want to represent words as vectors.
- We look at the intuition behind the model and how it is trained (with a splash of math for good measure).
- We also show a simple implementation of the model in TensorFlow.
- Finally, we look at ways to make the 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 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 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 (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, 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), and *predictive methods* (e.g. neural probabilistic language models).

This distinction is elaborated in much more detail by Baroni et al., 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.). 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

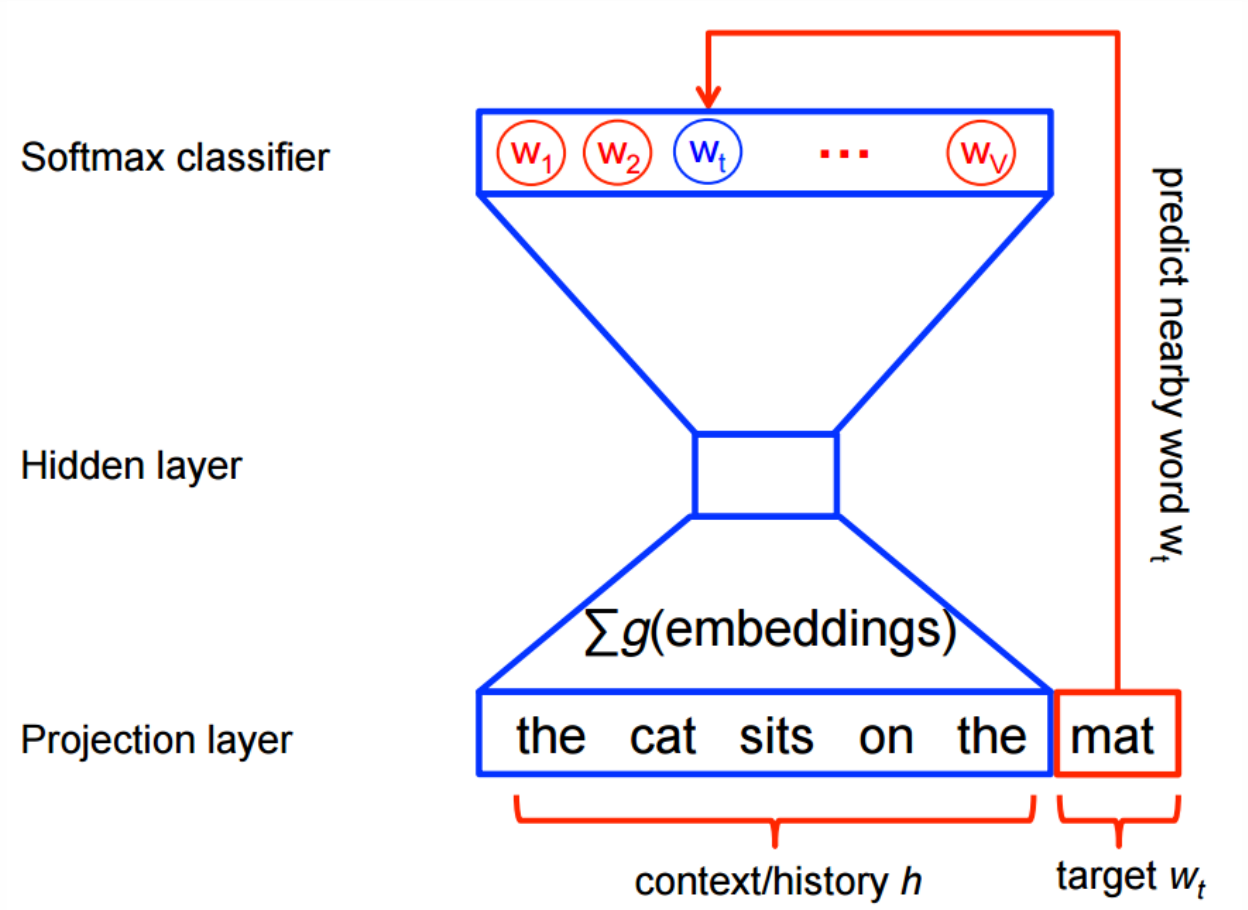
Neural probabilistic language models are traditionally trained using the maximum likelihood (ML) principle to maximize the probability of the next word wt (for "target") given the previous words h (for "history") in terms of a *softmax* function,

$$P(w_t \mid h) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}}$$

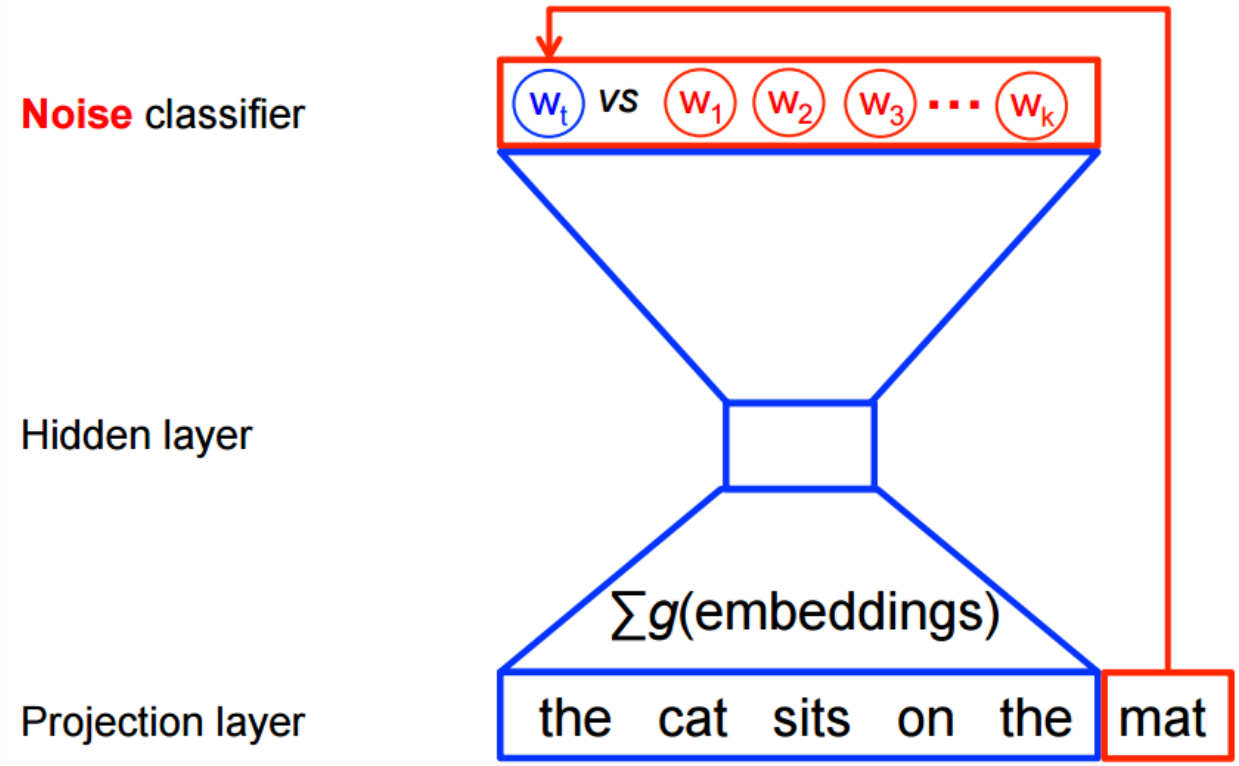
where score(wt,h) computes the compatibility of word wt with the context h (a dot product is commonly used). We train this model by maximizing its log-likelihood on the training set, i.e. by maximizing

$$J_{ML} = \log P(w_t \mid h) = \text{score}(w_t, h) - \log\left(\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}\right).$$

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) 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

$$J_{NEG} = \log Q_\theta(D=1 \mid w_t, h) + k E_{w \sim P_{noise}}[\log Q_\theta(D=0 \mid w, h)]$$

where $Q_\theta(D=1 \mid w,h)$ 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).

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, 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) 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.), 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), ...
```

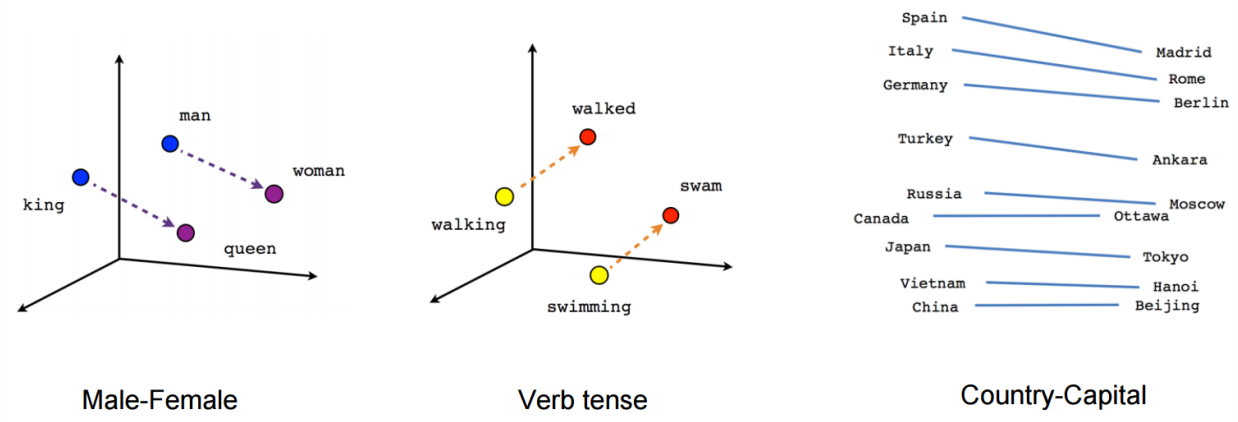
of `(input, output)` pairs. The objective function is defined over the entire dataset, but we typically optimize this withstochastic gradient descent (SGD) using one example at a time (or a 'minibatch' of `batch_size` examples, where typically `16 <= batch_size <= 512`). So let's look at one step of 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 t becomes

$$J_{NEG}(t) = \log Q_\theta(D=1 \mid \text{the, quick}) + \log(Q_\theta(D=0 \mid \text{sheep, quick}))$$

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. $\partial\partial\theta J_{NEG}$(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. 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-capital* relationships between words, as illustrated in the figure below (see also for example Mikolov et al., 2013).



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 (pdf), or follow-up work by Turian et al., 2010).

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 (seetensorflow/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

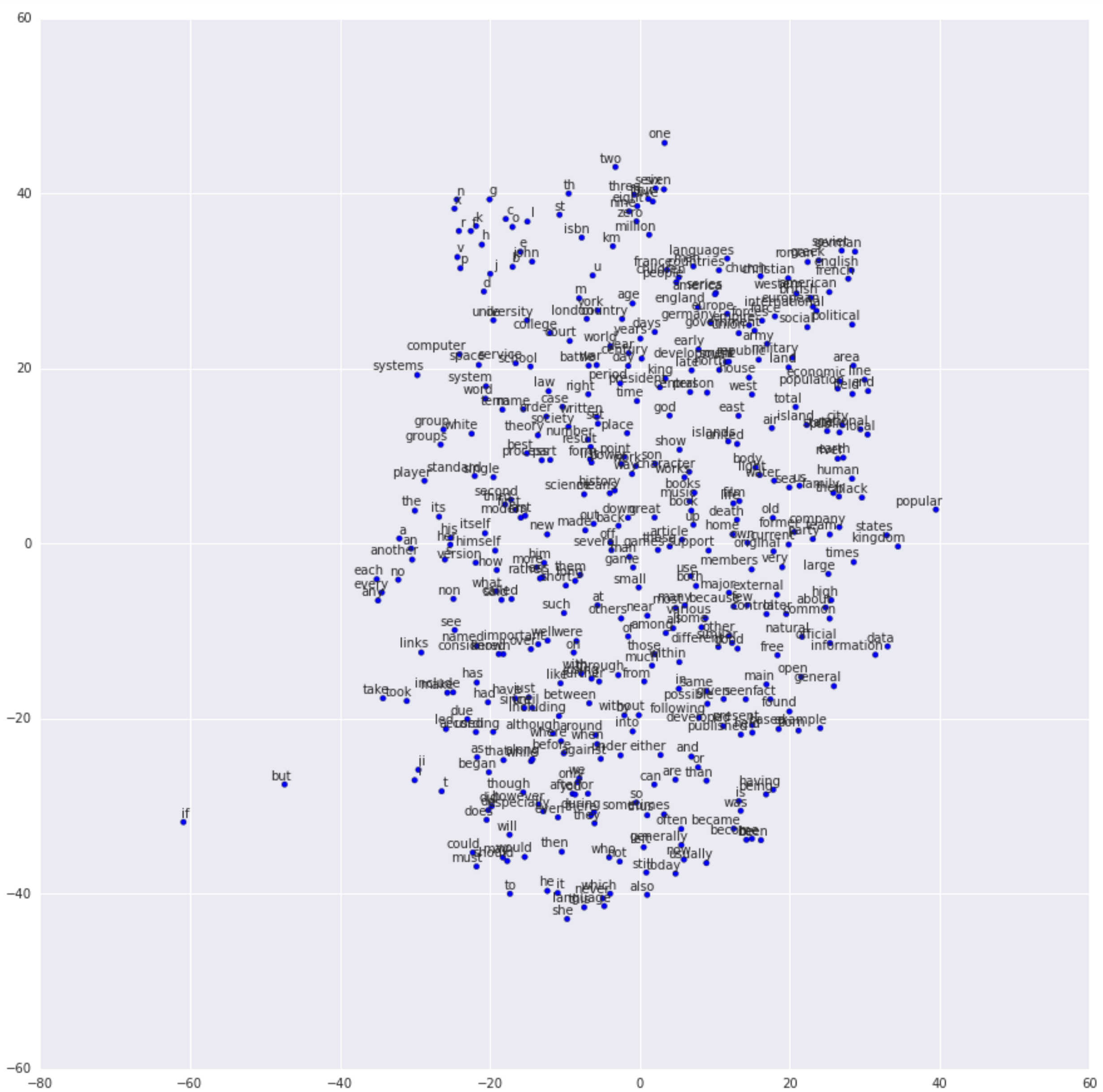
Training the model is then as simple as using a `feed_dict` to push data into the placeholders and calling`tf.Session.run` with this new data in a loop.

```
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.

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 inmodels/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 reasoning* and the task was introduced by Mikolov and colleagues . Download the dataset for this task from download.tensorflow.org.

To see how we do this evaluation, have a look at the `build_eval_graph()` and `eval()` functions inmodels/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 as`tf.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. For the case of Skip-Gram modeling, we've actually already done this for you as an example in 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. Again we've provided an example of this for the Skip-Gram casemodels/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.

Improving Linear Models Using Explicit Kernel Methods

ContentsLoad and prepare MNIST data for classificationTraining a simple linear modelUsing explicit kernel mappings with the linear model.Technical details

Note: This document uses a deprecated version of ${tf.estimator}, which has a ${tf.contrib.learn.estimator$different interface}. It also uses other `contrib` methods whose ${$version_compat#not_covered$API may not be stable}.

In this tutorial, we demonstrate how combining (explicit) kernel methods with linear models can drastically increase the latters' quality of predictions without significantly increasing training and inference times. Unlike dual kernel methods, explicit (primal) kernel methods scale well with the size of the training dataset both in terms of training/inference times and in terms of memory requirements.

Intended audience: Even though we provide a high-level overview of concepts related to explicit kernel methods, this tutorial primarily targets readers who already have at least basic knowledge of kernel methods and Support Vector Machines (SVMs). If you are new to kernel methods, refer to either of the following sources for an introduction:

- If you have a strong mathematical background: Kernel Methods in Machine Learning
- Kernel method wikipedia page

Currently, TensorFlow supports explicit kernel mappings for dense features only; TensorFlow will provide support for sparse features at a later release.

This tutorial uses tf.contrib.learn (TensorFlow's high-level Machine Learning API) Estimators for our ML models. If you are not familiar with this API, tf.estimator Quickstart is a good place to start. We will use the MNIST dataset. The tutorial consists of the following steps:

- Load and prepare MNIST data for classification.
- Construct a simple linear model, train it, and evaluate it on the eval data.
- Replace the linear model with a kernelized linear model, re-train, and re-evaluate.

Load and prepare MNIST data for classification

Run the following utility command to load the MNIST dataset:

```
data = tf.contrib.learn.datasets.mnist.load_mnist()
```

The preceding method loads the entire MNIST dataset (containing 70K samples) and splits it into train, validation, and test data with 55K, 5K, and 10K samples respectively. Each split contains one numpy array for images (with shape [sample_size, 784]) and one for labels (with shape [sample_size, 1]). In this tutorial, we only use the train and validation splits to train and evaluate our models respectively.

In order to feed data to a `tf.contrib.learn Estimator`, it is helpful to convert it to Tensors. For this, we will use an `input function` which adds Ops to the TensorFlow graph that, when executed, create mini-batches of Tensors to be used downstream. For more background on input functions, check this section on input functions. In this example, we will use the `tf.train.shuffle_batch` Op which, besides converting numpy arrays to Tensors, allows us to specify the batch_size and whether to randomize the input every time the input_fn Ops are executed (randomization typically expedites convergence during training). The full code for loading and preparing the data is shown in the snippet below. In this example, we use mini-batches of size 256 for training and the entire sample (5K entries) for evaluation. Feel free to experiment with different batch sizes.

```
import numpy as np
import tensorflow as tf

def get_input_fn(dataset_split, batch_size, capacity=10000,
min_after_dequeue=3000):

  def _input_fn():
    images_batch, labels_batch = tf.train.shuffle_batch(
        tensors=[dataset_split.images,
dataset_split.labels.astype(np.int32)],
        batch_size=batch_size,
        capacity=capacity,
        min_after_dequeue=min_after_dequeue,
        enqueue_many=True,
        num_threads=4)
    features_map = {'images': images_batch}
    return features_map, labels_batch

  return _input_fn

data = tf.contrib.learn.datasets.mnist.load_mnist()

train_input_fn = get_input_fn(data.train, batch_size=256)
eval_input_fn = get_input_fn(data.validation, batch_size=5000)
```

Training a simple linear model

We can now train a linear model over the MNIST dataset. We will use the `tf.contrib.learn.LinearClassifier`estimator with 10 classes representing the 10 digits. The input features form a 784-dimensional dense vector which can be specified as follows:

```
image_column = tf.contrib.layers.real_valued_column('images', dimension=784)
```

The full code for constructing, training and evaluating a LinearClassifier estimator is as follows:

```
import time

# Specify the feature(s) to be used by the estimator.
image_column = tf.contrib.layers.real_valued_column('images', dimension=784)
estimator = tf.contrib.learn.LinearClassifier(feature_columns=
[image_column], n_classes=10)

# Train.
start = time.time()
estimator.fit(input_fn=train_input_fn, steps=2000)
end = time.time()
print('Elapsed time: {} seconds'.format(end - start))

# Evaluate and report metrics.
eval_metrics = estimator.evaluate(input_fn=eval_input_fn, steps=1)
print(eval_metrics)
```

The following table summarizes the results on the eval data.

metric	**value**
loss	0.25 to 0.30
accuracy	92.5%
training time	~25 seconds on my machine

Note: Metrics will vary depending on various factors.

In addition to experimenting with the (training) batch size and the number of training steps, there are a couple other parameters that can be tuned as well. For instance, you can change the optimization method used to minimize the loss by explicitly selecting another optimizer from the collection of available optimizers. As an example, the following code constructs a LinearClassifier estimator that uses the Follow-The-Regularized-Leader (FTRL) optimization strategy with a specific learning rate and L2-regularization.

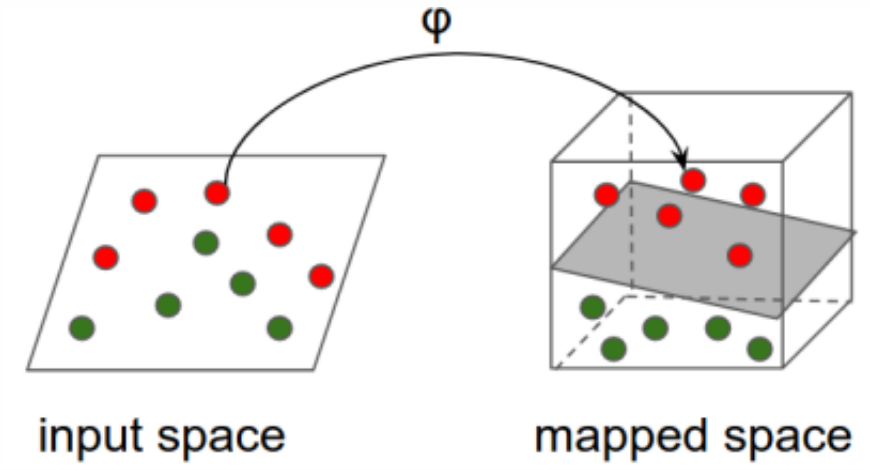
```
optimizer = tf.train.FtrlOptimizer(learning_rate=5.0,
l2_regularization_strength=1.0)
estimator = tf.contrib.learn.LinearClassifier(
    feature_columns=[image_column], n_classes=10, optimizer=optimizer)
```

Regardless of the values of the parameters, the maximum accuracy a linear model can achieve on this dataset caps at around **93%**.

Using explicit kernel mappings with the linear model.

The relatively high error (~7%) of the linear model over MNIST indicates that the input data is not linearly separable. We will use explicit kernel mappings to reduce the classification error.

Intuition: The high-level idea is to use a non-linear map to transform the input space to another feature space (of possibly higher dimension) where the (transformed) features are (almost) linearly separable and then apply a linear model on the mapped features. This is shown in the following figure:



Technical details

In this example we will use **Random Fourier Features**, introduced in the "Random Features for Large-Scale Kernel Machines" paper by Rahimi and Recht, to map the input data. Random Fourier Features map a vector $x \in \mathbb{R}^d$ to $x' \in \mathbb{R}^D$ via the following mapping:

$$RFFM(\cdot): \mathbb{R}^d \to \mathbb{R}^D, RFFM(x) = \cos(\Omega \cdot x + b)$$

where $\Omega \in \mathbb{R}^{D \times d}$, $x \in \mathbb{R}^d$, $b \in \mathbb{R}^D$ and the cosine is applied element-wise.

In this example, the entries of Ω and b are sampled from distributions such that the mapping satisfies the following property:

$$RFFM(x)^T \cdot RFFM(y) \approx e^{-\|x-y\|_2^2 \sigma^2}$$

The right-hand-side quantity of the expression above is known as the RBF (or Gaussian) kernel function. This function is one of the most-widely used kernel functions in Machine Learning and implicitly measures similarity in a different, much higher dimensional space than the original one. See Radial basis function kernel for more details.

Kernel classifier

`tf.contrib.kernel_methods.KernelLinearClassifier` is a pre-packaged `tf.contrib.learn` estimator that combines the power of explicit kernel mappings with linear models. Its constructor is almost identical to that of the LinearClassifier estimator with the additional option to specify a list of explicit kernel mappings to be applied to each feature the classifier uses. The following code snippet demonstrates how to replace LinearClassifier with KernelLinearClassifier.

```
# Specify the feature(s) to be used by the estimator. This is identical to
the
# code used for the LinearClassifier.
image_column = tf.contrib.layers.real_valued_column('images', dimension=784)
optimizer = tf.train.FtrlOptimizer(
   learning_rate=50.0, l2_regularization_strength=0.001)

kernel_mapper = tf.contrib.kernel_methods.RandomFourierFeatureMapper(
  input_dim=784, output_dim=2000, stddev=5.0, name='rffm')
kernel_mappers = {image_column: [kernel_mapper]}
estimator = tf.contrib.kernel_methods.KernelLinearClassifier(
   n_classes=10, optimizer=optimizer, kernel_mappers=kernel_mappers)

# Train.
start = time.time()
estimator.fit(input_fn=train_input_fn, steps=2000)
end = time.time()
print('Elapsed time: {} seconds'.format(end - start))

# Evaluate and report metrics.
eval_metrics = estimator.evaluate(input_fn=eval_input_fn, steps=1)
print(eval_metrics)
```

The only additional parameter passed to `KernelLinearClassifier` is a dictionary from feature_columns to a list of kernel mappings to be applied to the corresponding feature column. The following lines instruct the classifier to first map the initial 784-dimensional images to 2000-dimensional vectors using random Fourier features and then learn a linear model on the transformed vectors:

```
kernel_mapper = tf.contrib.kernel_methods.RandomFourierFeatureMapper(
  input_dim=784, output_dim=2000, stddev=5.0, name='rffm')
kernel_mappers = {image_column: [kernel_mapper]}
estimator = tf.contrib.kernel_methods.KernelLinearClassifier(
   n_classes=10, optimizer=optimizer, kernel_mappers=kernel_mappers)
```

Notice the `stddev` parameter. This is the standard deviation (σ) of the approximated RBF kernel and controls the similarity measure used in classification. `stddev` is typically determined via hyperparameter tuning.

The results of running the preceding code are summarized in the following table. We can further increase the accuracy by increasing the output dimension of the mapping and tuning the standard deviation.

metric	**value**
loss	0.10
accuracy	97%
training time	~35 seconds on my machine

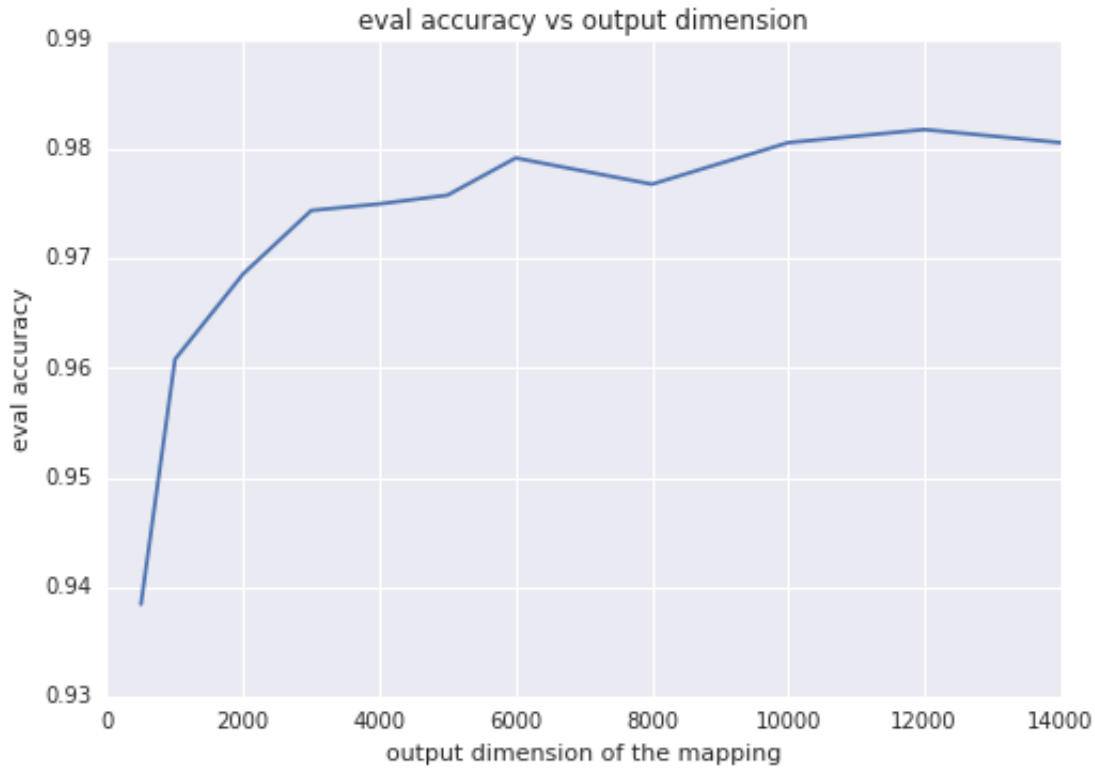
stddev

The classification quality is very sensitive to the value of stddev. The following table shows the accuracy of the classifier on the eval data for different values of stddev. The optimal value is stddev=5.0. Notice how too small or too high stddev values can dramatically decrease the accuracy of the classification.

stddev	**eval accuracy**
1.0	0.1362
2.0	0.4764
4.0	0.9654
5.0	0.9766
8.0	0.9714
16.0	0.8878

Output dimension

Intuitively, the larger the output dimension of the mapping, the closer the inner product of two mapped vectors approximates the kernel, which typically translates to better classification accuracy. Another way to think about this is that the output dimension equals the number of weights of the linear model; the larger this dimension, the larger the "degrees of freedom" of the model. However, after a certain threshold, higher output dimensions increase the accuracy by very little, while making training take more time. This is shown in the following two Figures which depict the eval accuracy as a function of the output dimension and the training time, respectively.



Summary

Explicit kernel mappings combine the predictive power of nonlinear models with the scalability of linear models. Unlike traditional dual kernel methods, explicit kernel methods can scale to millions or hundreds of millions of samples. When using explicit kernel mappings, consider the following tips:

- Random Fourier Features can be particularly effective for datasets with dense features.
- The parameters of the kernel mapping are often data-dependent. Model quality can be very sensitive to these parameters. Use hyperparameter tuning to find the optimal values.
- If you have multiple numerical features, concatenate them into a single multi-dimensional feature and apply the kernel mapping to the concatenated vector.

Mandelbrot Set

Visualizing the 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...

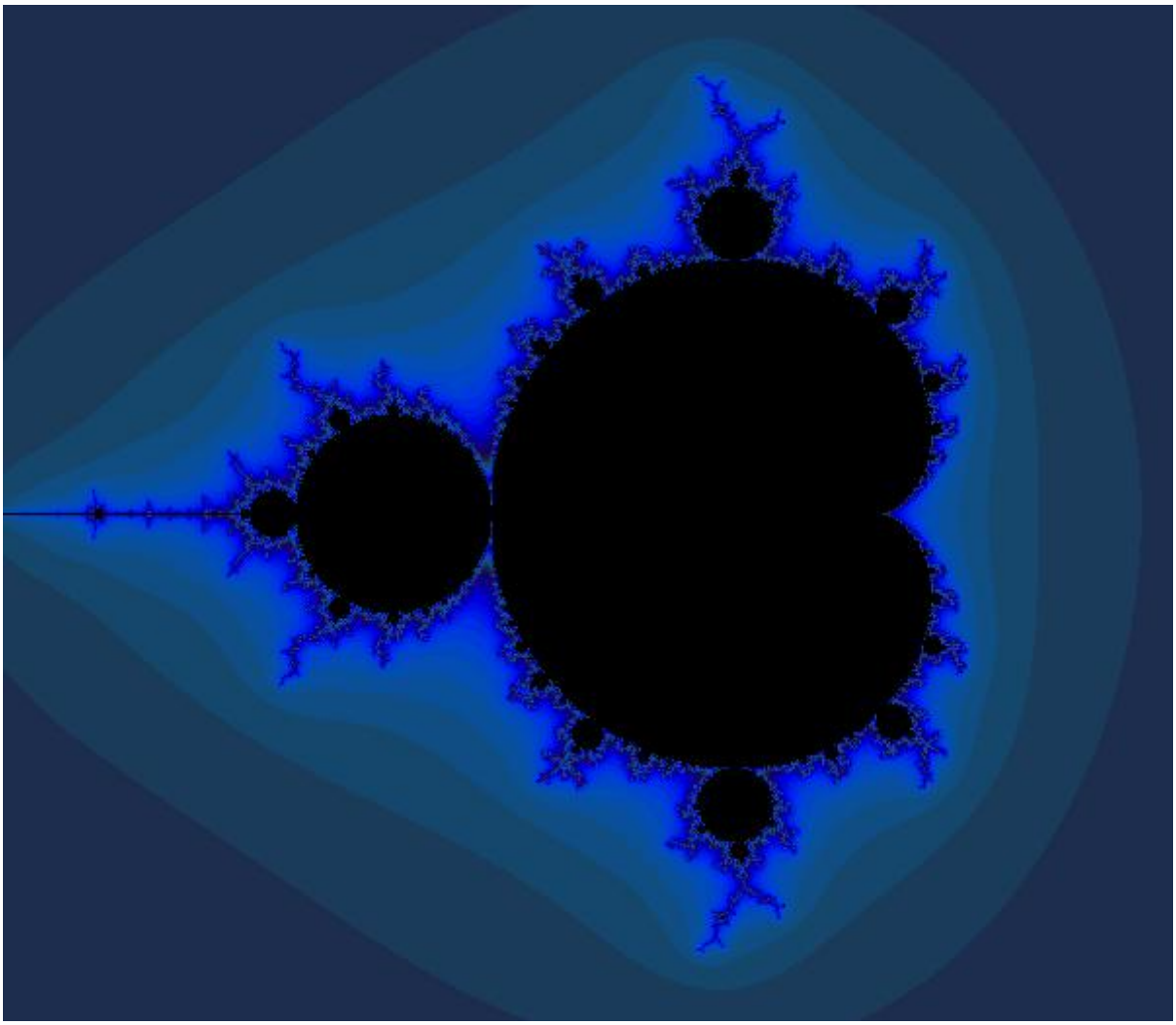
```
# Compute the new values of z: z^2 + x
zs_ = zs*zs + xs

# 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. 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)
```

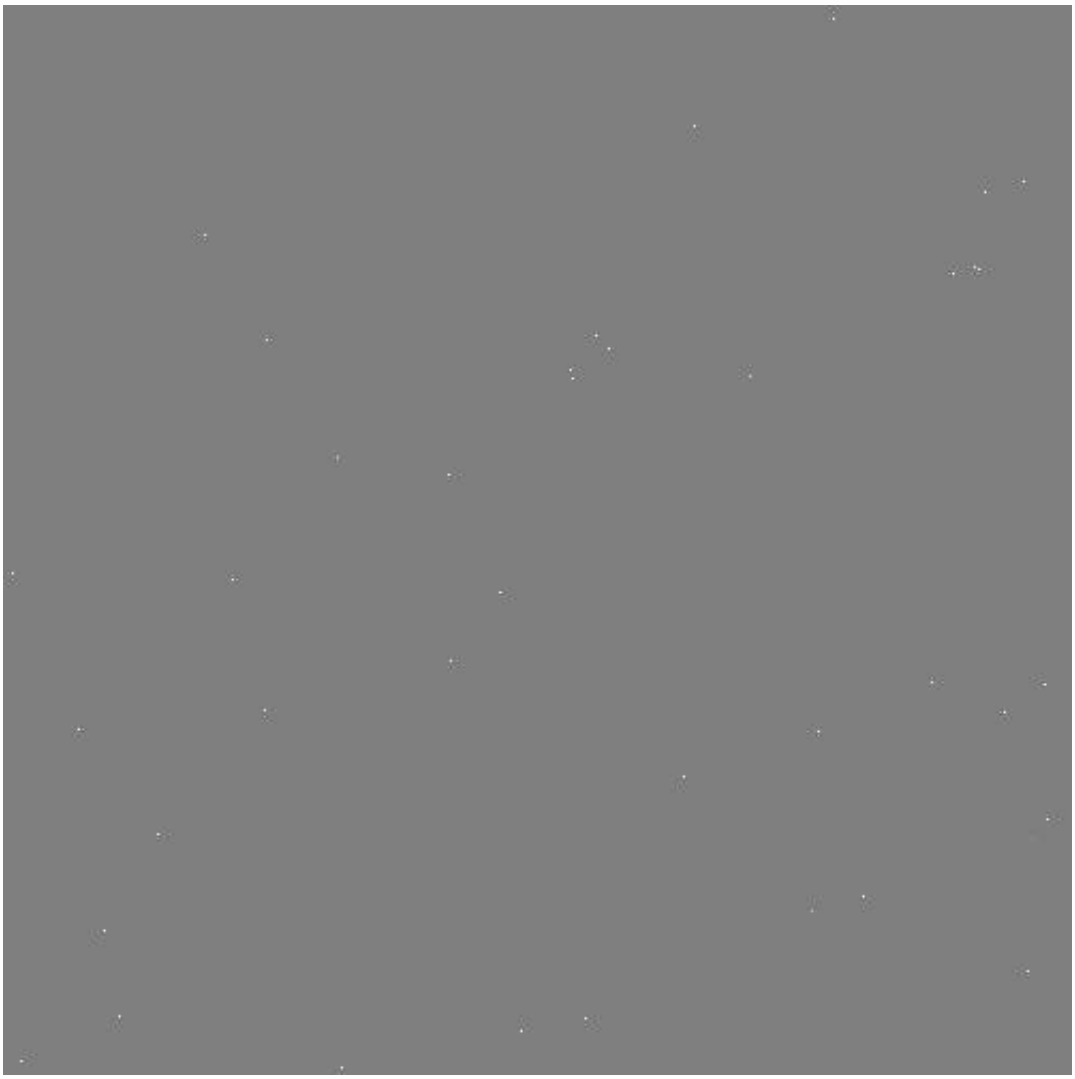
Define the PDE

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])
```



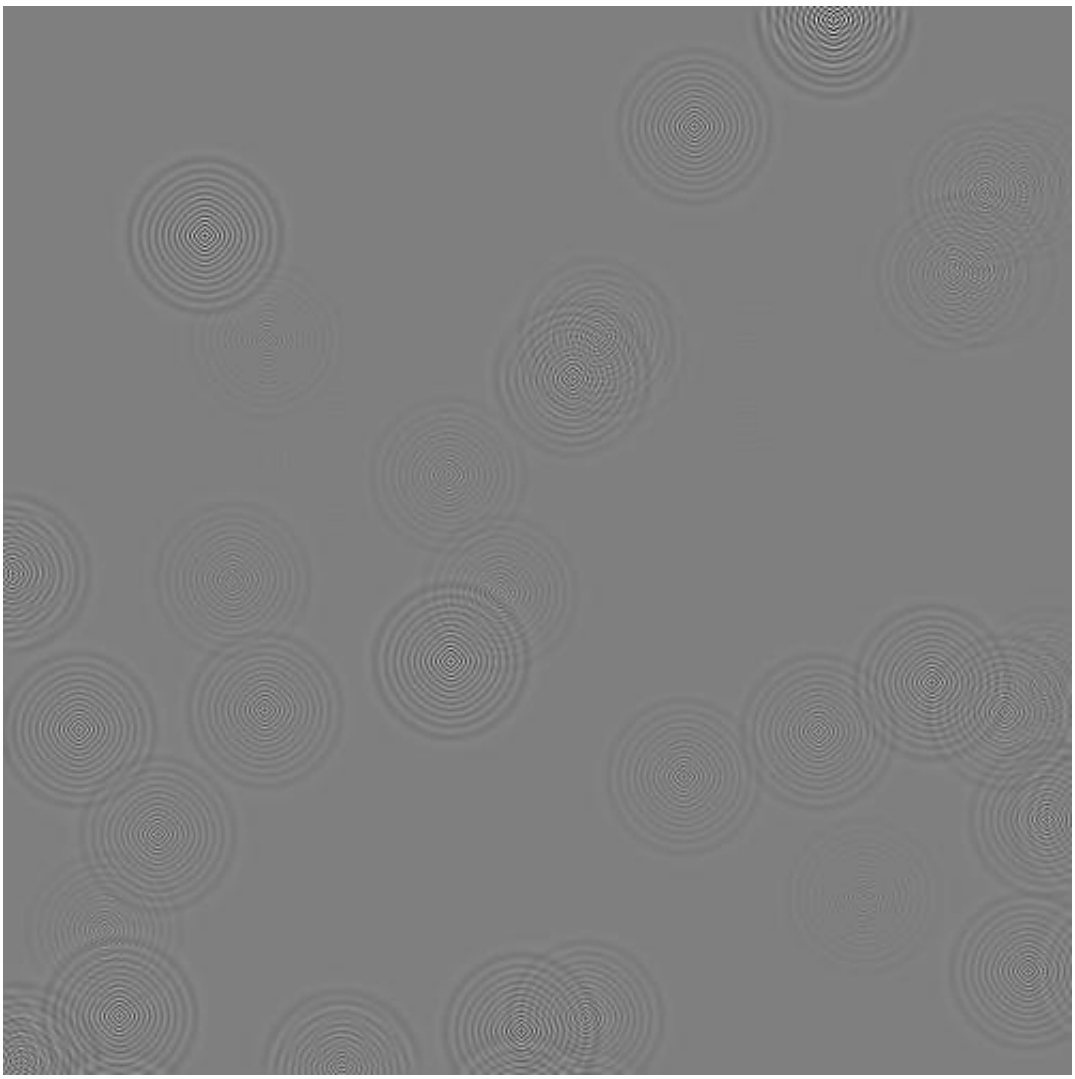
Now let's specify the details of the differential equation.

```
# 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!