TensorFlow Machine Learning Cookbook

Second Edition

Over 60 recipes to build intelligent machine learning systems with the power of Python

Packt
www.packt.com

By Nick McClure

TensorFlow Machine Learning Cookbook **Spring 2019**

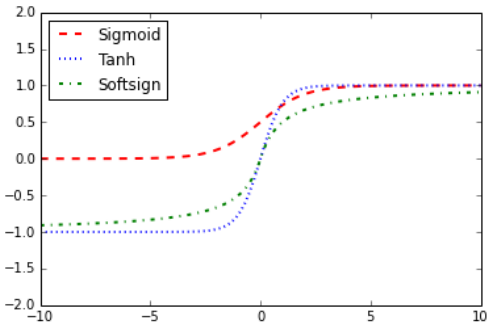
Chapter 1: Getting Started with TensorFlow

Scribe: Roderick Karlemstrand *Date: 11 May 2019*

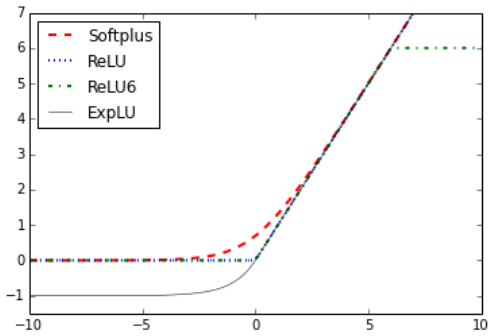
Disclaimer: *These notes have been subjected to the usual scrutiny reserved for formal publications.*

1.1 Activation Functions

The activation functions live in the neural network (nn) library in TensorFlow.



(a) Sigmoid, hyperbolic tangent, and softsign act func



(b) Softplus, ReLU, ReLU6, and exponential LU

Figure 1.1: Some Act Funcs

1. The rectified linear unit, known as ReLU, is the most common and basic way to introduce non-linearity into neural networks. This function is just called max(0,x). It is continuous, but not smooth.
2. To cap the linearly increasing part of the preceding ReLU activation function. Do this by nesting the max(0,x) function into a min() function, called the ReLU6 function.

   It is computationally faster, and does not suffer from vanishing (infinitesimally near zero) or exploding values.
3. Sigmoid, most common continuous and smooth activation function. Not used very often because of its tendency to zero-out the backpropagation terms during training.
4. Softplus function, is a smooth version of the ReLU function.

Some TensorFlow resources:

- A public Docker container that is kept current by TensorFlow is available on Dockerhub at `https://hub.docker.com/r/tensorflow/tensorflow/`
- [**DO NOT MISS THIS**] TensorFlow has also made a site where you can visually explore training a neural network while changing the parameters and datasets. Visit `http://playground.tensorflow.org/` to explore how different settings affect the training of neural networks.

TensorFlow Machine Learning Cookbook **Spring 2019**

Chapter 2: The TensorFlow Way

Scribe: Roderick Karlemstrand *Date: 11 May 2019*

Disclaimer: *These notes have been subjected to the usual scrutiny reserved for formal publications.*

This chapter will cover:

- Operations in a computational graph
- Layering nested operations
- Working with multiple layers
- Implementing loss functions
- Implementing backpropagation
- Working with batch and stochastic training
- Combining everything together
- Evaluating models

2.2 Working with multiple layers

About how to best connect various layers, including custom layers. The data we will generate and use will be representative of small random images.

The first layer we will explore is called a **moving window**. We will perform a small moving window average across a 2D image and then the second layer will be a custom operation layer.

In this section, we will see that the computational graph can get large and hard to look at. To address this, we will also introduce ways to name operations and create scopes for layers.

1. Create our sample 2D image, but we use four dimensions: image number, height, width, and channel, and to make it one image with one channel, we set two of the dimensions to 1, as follows:

```
x_shape = [1, 4, 4, 1]
x_val = np.random.uniform(size=x_shape)
```

2. Create the placeholder in our graph where we can feed in the sample image

```
x_data = tf.placeholder(tf.float32, shape=x_shape)
```

3. Moving window average across the 4 x 4 image, built-in function convolutes a constant across a window of the shape 2 x 2.

   The function we will use is conv2d(); this function is quite commonly used in image processing and in TensorFlow. This function takes a piecewise product of the window and a filter we specify. We must also specify a stride for the moving window in both directions. Here, we will compute four moving window averages: the upper-left, upper-right,lower-left, and lower-right four pixels. We do this by creating a 2 x 2 window and having strides of length 2 in each direction. To take the average, we will convolute the 2 x 2 window with a constant of 0.25, as follows:

```
my_filter = tf.constant(0.25, shape=[2, 2, 1, 1])
my_strides = [1, 2, 2, 1]
mov_avg_layer= tf.nn.conv2d(x_data, my_filter, my_strides,
                        padding='SAME',
                        name='Moving_Avg_Window')
```

4. Define a custom layer that will operate on the 2 x 2 output of the moving window average. The custom function will first multiply the input by another 2 x 2 matrix tensor, and then add 1 to each entry.

   After this, we take the sigmoid of each element and return the 2 x 2 matrix.

   Since matrix multiplication only operates on two-dimensional matrices, we need to drop the extra dimensions of our image that are of size 1. TensorFlow can do this with the built-in squeeze() function. Here, we define the new layer:

```
def custom_layer(input_matrix):
    input_matrix_sqeezed = tf.squeeze(input_matrix)
    A = tf.constant([[1., 2.], [-1., 3.]])
    b = tf.constant(1., shape=[2, 2])
    temp1 = tf.matmul(A, input_matrix_sqeezed)
    temp = tf.add(temp1, b) # Ax + b
    return tf.sigmoid(temp)
```

5. Place the new layer on the graph. We will do this with a named scope so that it is identifiable and collapsible/expandable on the computational graph in Tensorboard.

```
with tf.name_scope('Custom_Layer') as scope:
    custom_layer1 = custom_layer(mov_avg_layer)
```

6. Now, we just feed in the 4 x 4 image to replace the placeholder and tell TensorFlow to run the graph, as follows:

```
print(sess.run(custom_layer1, feed_dict={x_data: x_val}))
[[ 0.91914582 0.96025133]
 [ 0.87262219 0.9469803 ]]
```

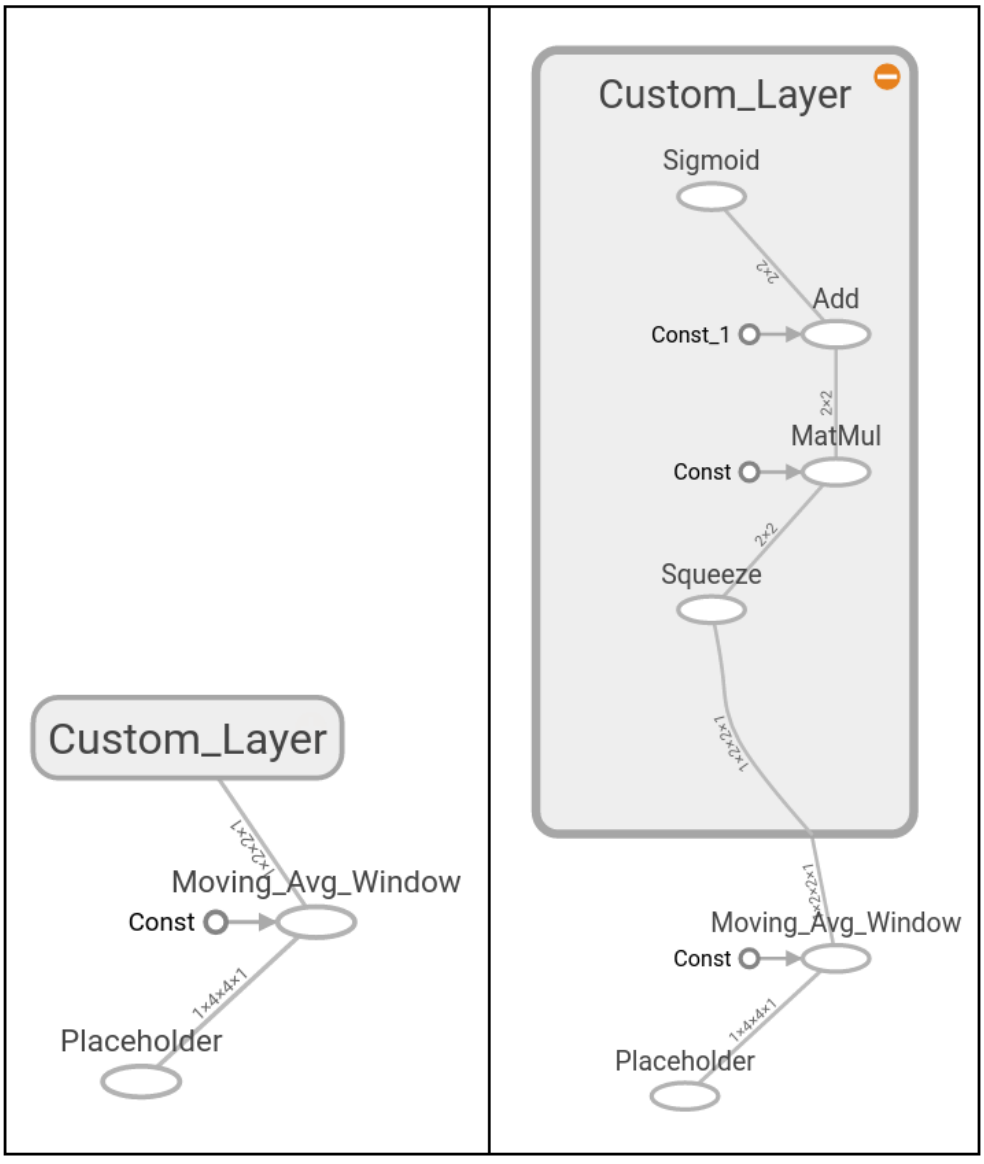


Figure 2.2: Computational graph with two layers

2.3 Implementing loss functions

First, we will talk about loss functions for regression, which means predicting a continuous dependent variable.

The L2 norm loss is also known as the Euclidean loss function. It is just the square of the distance to the target. The L2 norm is a great loss function because it is very curved near the target and algorithms can use this fact to converge to the target more slowly the closer it gets to zero.

The L1 norm loss is also known as the absolute loss function. Instead of squaring the difference, we take the absolute value. The L1 norm is better for outliers than the L2 norm because it is not as steep for larger values. One issue to be aware of is that the L1 norm is not smooth at the target, and this can result in algorithms not converging well.

2.4 Backprop

For a recap and explanation, for both examples we did the following:

1. Created the data. Both examples needed to load data through a placeholder.
2. Initialized placeholders and variables. These were very similar placeholders for the data. The variables were very similar, they both had a multiplicative matrix, A, but the first classification algorithm had a bias term to find the split in the data.
3. Created a loss function, we used the L2 loss for regression and the cross-entropy loss for classification.
4. Defined an optimization algorithm. Both algorithms used gradient descent.
5. Iterated across random data samples to iteratively update our variables.

Learning rate size.. See Section "There's more..." in the book

Ideally, we would like to take larger steps for smaller moving variables and shorter steps for faster changing variables. We will not go into the mathematics of this approach, but a common implementation of this idea is called the Adagrad algorithm.

Adagrad forces the gradients to zero too soon because it takes into account the whole history. A solution to this is to limit how many steps we use. Doing this is called the Adadelta algorithm.

2.5 Working with batch and stochastic training

Operating on **one** training example can make for a very **erratic** learning process, while using **too large** a batch can be **computationally expensive**. Choosing the right type of training is crucial for getting our machine learning algorithms to converge to a solution.

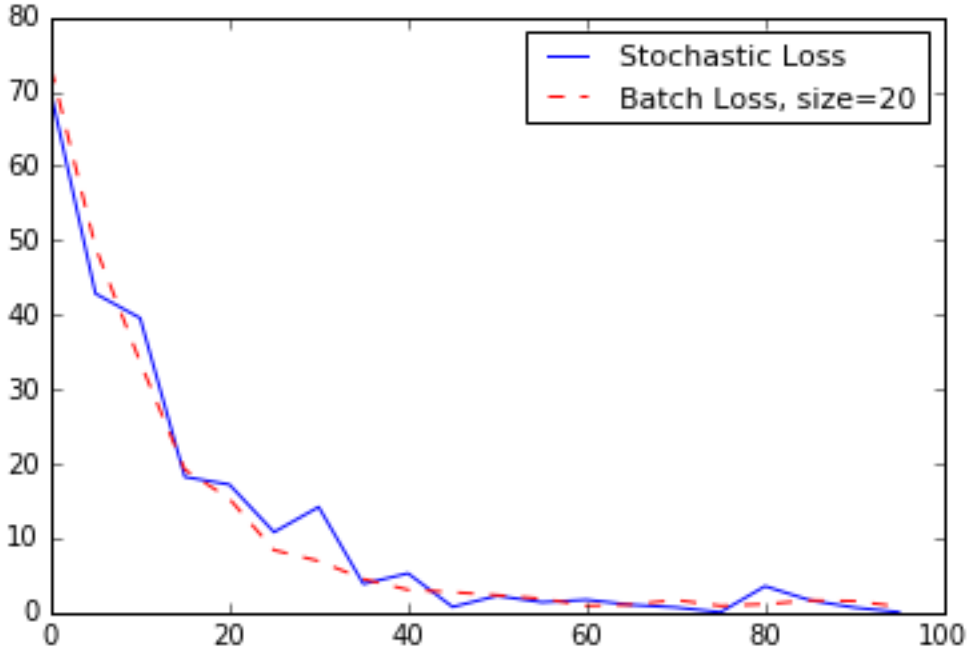


Figure 2.3: Stochastic loss and batch loss (batch size = 20) plotted over 100 iterations.

Note that the batch loss is much smoother and the stochastic loss is much more erratic.

2.6 Evaluating models

After training an algorithm in TensorFlow, we must be able to evaluate the model's predictions to determine how well it did.

Using TensorFlow, we must build this feature (Evaluating models) into the computational graph and call it while our model is training and/or after it has finished training.

Evaluating models during training gives us an insight into the algorithm and may give us hints to **debug it, improve it, or change models entirely.** While evaluation during training isn't always necessary.

After training, we need to **quantify** how the model performs on the data.

Another important aspect of any model we want to evaluate is if it is a regression or classification model.

Regression models attempt to predict a continuous number. The target is not a category, but a desired number. To evaluate these regression predictions against the actual targets, we need an aggregate measure of the distance between the two. Most of the time, a meaningful loss function will satisfy these criteria. This recipe shows you how to change the simple regression algorithm from before into printing out the loss in the training loop and evaluating the loss at the end. For an example, we will revisit and rewrite our regression example in the prior Implementing back propagation recipe in this chapter.

Classification models predict a category based on numerical inputs. The actual targets are a sequence of 1s and 0s, and we must have a measure of how close we are to the truth from our predictions.

The loss function for classification models usually isn't that helpful in interpreting how well our model is doing. Usually, we want some sort of classification accuracy, which is commonly the percentage of correctly predicted categories. For this example, we will use the classification example from the prior Implementing back propagation recipe in this chapter.