`latex-mimosis`

A minimal, modern LaTeX package for typesetting your thesis

by

Bastian Rieck

A document submitted in partial fulfillment of the requirements for the degree of

Technical Report

at

Miskatonic University

Abstract

Scientific documents often use LaTeX for typesetting. While numerous packages and templates exist, it makes sense to create a new one. Just because.

Contents

1 Technology

In this chapter I'm describing the technology that I used to implement the set of utilities and experiments described in section ??? This chapter is organized as follows. In 1.1 I'm describing what is TensorFlow and how its computational graphs work. In 1.2 I'm describing Keras, an high-level API for building Machine Learning models without working with a low-level library like TensorFlow. In 1.3 I'm describing CleverHans, a library to generate adversarial examples against a given model. In 1.4 I'm describing scikit-learn, a library for building and training Machine Learning models.

1.1 TensorFlow

TensorFlow is a C++ framework for Machine Learning released by Google. It uses a programming model called Data Flow that aims to allow distributed and parallel computations. While this *paradigm* makes TensorFlow suitable even for research and production environments, it can be quite daunting to use when tinkering. In fact, computational graphs are built and only later executed. This is counter-intuitive at first. For example, in the following snippet of code

```
>>> import tensorflow as tf
>>> symbol = tf.constant(42)
>>> symbol
<tf.Tensor 'Const:0' shape=() dtype=int32>
```

`symbol` doesn't contain a reference to the integer 42. Instead it contains a reference to a node of the computational graph (which will always output 42). In general until you don't run the computational graph it's hard to determine what's going to be the value of a tensor. That's slowing down exploratory analysis.

1.1.1 About computational graphs

TensorFlow computational graphs are a fundamental part of the framework. A computational graph is a directed graph representing a computation involving tensors. A tensor in TensorFlow is a multi-dimensional array: it can be a scalar, a matrix, a batch of RGB images (which is a 4D vector), ... Each node represents an operation on tensors, while edges represent tensors. (This distinction between nodes and edges is more important from a formal standpoint than a practical one: actually thinking of tensors as nodes makes no difference to the best of my knowledge.)
Figure 1.1 shows a very simple computational graph. It's the one associated to an addition of two `tf.float32`'s (i.e. two tensors made of 32-bit float numbers). The first two nodes `a` and `b` outputs

a tensor each. Those are used to feed the `add` node that will output the tensor resulting from the sum of `a` and `b`.

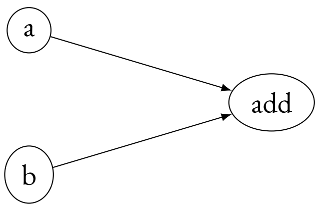

Figure 1.1: An easy computational graph

A more interesting example would be to implement a neural network using the computational graph. The neural network we want to implement consists of 4 input neurons and 4 output neurons. A graphical representation of that is given in Figure 1.2.

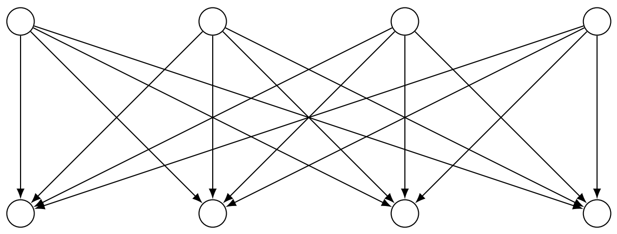

Figure 1.2: A simple neural network

We'll need the weights and the bias and do

$$X * W + b$$

where $W \in \mathbb{R}^{4\times4}$ and $b \in \mathbb{R}^4$, while X is a *batch* of input vectors in $\mathbb{R}^4$. If you imagine the input is a flattened image of 2 by 2 pixels, this neural network learns how to classify 4 different classes of images.
Using the TensorFlow Python API to build the related computational graph, we'll do

```
>>> import tensorflow as tf
>>> W = tf.random_normal(shape=(4, 4))
>>> b = tf.random_normal(shape=(4,))
>>> X = tf.placeholder(shape=(None, 4), dtype=tf.float32)
>>> logits = tf.matmul(X, W) + b
>>> probabilities = tf.nn.softmax(logits)
>>>
>>> probabilities
<tf.Tensor 'Softmax:0' shape=(?, 4) dtype=float32>
```

Notice the `tf.nn.softmax` function is applied to the result of the matrix multiplication. This is done for various reasons: it introduces non-linearity allowing the neural network to learn to compute any function – what's known as the "Universal approximation theorem"; also it normalizes

the results of the neural network in $[0, 1]$ allowing to interpret the output of the neural network in terms of *probabilities*/confidence levels.
To get results out of the graph you need what's called a session in TensorFlow. Basically a session *powers on* the graph, allowing you to run the computational graph with your own data and get actual tensors of actual numbers out of it. In the example above we can use a session to get probabilities out of our neural network for a batch of 2×2 images.

```
>>> import numpy as np
>>> ten_two_by_two_images_batch = np.random.rand(10, 4)
>>> ten_two_by_two_images_batch
array([[0.54485176, 0.33854871, 0.45185129, 0.79884188],
      [0.41204776, 0.23552753, 0.04101023, 0.47883844],
      [0.25544491, 0.7610509 , 0.49307137, 0.6098213 ],
      [0.02545156, 0.70459456, 0.22067103, 0.64743811],
      [0.92359354, 0.96497353, 0.45790538, 0.49380769],
      [0.13330072, 0.22947966, 0.02996348, 0.69954114],
      [0.38397249, 0.30473362, 0.87023559, 0.90153084],
      [0.77056319, 0.94843128, 0.39095345, 0.50572861],
      [0.90112077, 0.19240995, 0.48437166, 0.46200152],
      [0.98589042, 0.2013479 , 0.86091217, 0.55886214]])
>>> with tf.Session() as session:
...     # we're _feeding_ the X placeholder with an actual numpy array
...     session.run(probabilities, feed_dict={X: ten_two_by_two_images_batch})
...
array([[0.03751625, 0.8651346 , 0.05245555, 0.04489357],
      [0.11873736, 0.6301607 , 0.17646323, 0.07463871],
      [0.06252376, 0.82338244, 0.07284217, 0.04125158],
      [0.12086256, 0.6911012 , 0.14362463, 0.04441159],
      [0.03662292, 0.8575108 , 0.04834893, 0.05751737],
      [0.12984137, 0.63064647, 0.1834405 , 0.05607158],
      [0.01711418, 0.93650085, 0.02116078, 0.02522417],
      [0.04886489, 0.83023334, 0.06332602, 0.05757581],
      [0.033136  , 0.84808177, 0.05061754, 0.06816475],
      [0.01250888, 0.9262628 , 0.01797607, 0.0432523 ]], dtype=float32)
>>> #              ^^^^^^^^^ the last image is of the second class with a confidence of 92%
```

Of course building neural networks is pretty useless if you can't train them. In fact the probabilities above are totally non-sense: they're only based on the initial weights and bias randomly extracted from a normal distribution. There's no training set, no training phase yet. To train a network in TensorFlow we need a `tf.Variable`. A variable in TensorFlow is a tensor whose value persists across sessions and that can be modified. This allows you to basically add parameters to your computational graphs, allowing you to compute different functions with the same graph. That, combined with a learning procedure that modifies the `tf.Variables` allows you to train your model.

To build a `tf.Variable` in TensorFlow you can just wrap an existing tensor. In the example above, the part of the graph that you want to parameterize are the weights matrix W and the bias vector b.

```
W = tf.Variable(W)
b = tf.Variable(b)
logits = tf.matmul(X, W) + b # redefine logits: using different tensors now
probabilities = tf.nn.softmax(logits)
```

To perform training we now need a training set. A training set in this example woule consist of lots of 2×2 images and an ID associated to each image representing the class of that image.
Training basically means minimizing a loss function between the provided IDs of the training set and the IDs computed by the neural network. Minimization happens thanks to the parameters of our neural network: the `W` and `b` `tf.Variables`.

```
y_true = tf.placeholder(shape=(None,), dtype=tf.float32)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=y_true)
```

where *cross-entropy* is our loss function that we want to minimize.
To minimize the cross-entropy one can use the Gradient descent algorithm – an optimization procedure that works iteratively. TensorFlow implements that algorithm via a class exposing `.minimize()`: it returns a tensor that each time you run it in a session will modify the graph's `tf.Variables`, trying to lower the value of the loss function.

```
from tf.train import GradientDescentOptimizer as SGD
optimizer = SGD(learning_rate=0.5).minimize(cross_entropy)
```

So a typical training phase would consist of

```
with tf.Session() as session:
    session.run(tf.global_variables_initializer()) # needed.

n_training_steps = 10
for i in range(n_training_steps):
    images, classes = next_batch()
    # fictitious function returning batches from the training set

    with tf.Session() as session:
        session.run(optimizer, feed_dict={X: images, y_true: classes})
```

As you increase `n_training_steps`, `W` and `b` will be modified pushing the value of the loss function towards 0. That makes the model learn how to correctly classify images similar to those in the training set.

1.2 Keras

Keras is a high level library for building Machine Learning models. It consists of a simple API and bindings for a backend of choice, most notably TensorFlow. When you use the library, you're encouraged to use *layers* instead of plain tensors. In fact the whole concept of TensorFlow tensors is hidden away by the library abstractions. To implement your model you stack layers – basically you build a list of layers. For example, to implement a simple neural network one would stack a `Dense` layer of 4 hidden neurons and an `Activation` layer for the Softmax function.

```
from keras.models import Sequential
model = Sequential([
  Dense(batch_input_shape=(None, 4), units=4),
  Activation('softmax')
])
```

Under the hood, a `Layer` is simply a callable object whose `__call()__` method takes a TensorFlow tensor `X` and manipulates it. For example a simple custom one would be

```
from keras.engine.base_layer import Layer
import tensorflow as tf
class Add42(Layer):
    def __init__(self):
        self.fortytwo = tf.constant(42)

    def __call__(self, X):
        return tf.add(X, self.fortytwo)
```

(Of course in a real layer instead of `self.fortytwo` one would have trainable TensorFlow variables.)
While the idea is beginner friendly – as it allows people to immediately concentrate on model's architecture instead of thinking about tensors – at the time of this writing the abstraction can become leaky when things goes wrong, needing the programmer to know about TensorFlow computational graphs to debug her program successfully.

1.3 CleverHans

CleverHans is a library written by Google for building adversarial examples. It implements a variety of attacks against neural networks (FGS, T-FGS, Carlini and Wagner, ...) and it's compatible with models built with Keras or plain TensorFlow. It uses the same computational graphs of TensorFlow to generate adversarial examples; again, as you use the library more and more understanding computational graphs becomes a necessity.

1.4 Scikit-learn

Scikit-learn is a Python library written by Google providing a number of models, learning algorithms and other utilities for Machine Learning. It's perfect for fast prototyping as it uses a simple and consistent API and handles numpy arrays or even Python lists.
I've used scikit-learn to borrow a couple of decomposition algorithms (PCA, FastICA, ...) without having to implement them. This required a bit of reasoning as reusing scikit-learn algorithms on TensorFlow graphs was not straightforward.

Acronyms

PCA	Principal component analysis
SNF	Smith normal form
TDA	Topological data analysis

Glossary

$\LaTeX$	A document preparation system
$\mathbb{R}$	The set of real numbers