Some Deep Learning Cooking Recipes in TensorFlow

Thursday 7th March, 2019

Matthia Sabatelli[1]

[1]Montefiore Institute, Department of Electrical Engineering and Computer Science, Université de Liège, Belgium

Overview

1. Back in the days...
2. Tensorflow
3. Keras

Back in the days...

Back in the days...

```
def calculate_loss(model):
    num_examples = len(X)

    W1, b1, W2, b2 = model['W1'], model['b1'],
        model['W2'], model['b2']

    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2

    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1,
        keepdims=True)
    corect_logprobs =
        -np.log(probs[range(num_examples), y])

    data_loss = np.sum(corect_logprobs)
    data_loss += reg_lambda / 2 *
        (np.sum(np.square(W1)) +
        np.sum(np.square(W2)))
```

Back in the days...

```java
public class MLP implements Cloneable
{
    protected double fLearningRate = 0.001;
    protected Layer[] fLayers;
    double dVal=0;
    public ArrayList<Integer> checkArray = new ArrayList<Integer>();

    public MLP(int[] layers, double learningRate){
        fLearningRate = learningRate;
        fLayers = new Layer[layers.length];
        for(int i = 0; i < layers.length; i++){
            if(i != 0){
                fLayers[i] = new Layer(layers[i], layers[i - 1]);
            }
            else{
                ...
```

Back in the days...

```
import numpy
import theano as T

w_1 = T.shared(rng.randn(784, 300), name='w1')
b_1 = T.shared(numpy.zeros((300,)), name='b1')

from theano.tensor.nnet import sigmoid

p_1 = sigmoid(-T.dot(sigmoid(-T.dot(x, w_1)-b_1),
    w_2)-b_2)
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
cost = xent.mean() + 0.01 * (w_2** 2).sum()

gw_1, gb_1, gw_2, gb_2 = T.grad(cost, [w_1, b_1, w_2,
    b_2])

train = T.function(inputs = [x, y], outputs =
    [prediction, xent], updates = {w_1 :
    w_1-0.1*gw_1, b_1 : b_1-0.1*gb_1, ... })

predict = T.function(inputs=[x], outputs=prediction)
```

Tensorflow

Some TF facts

- Developed by Google Brain and open-sourced in November 2015
- Not only limited to Deep Learning
- Supports many Python APIs (`Keras`, `TF-slim`, `TF-Learn`)
- Runs efficient `C++` code in the backend
- Provides **Automatic Differentiation!**

Installation Instructions

Install Anaconda and create a Virtual-Env

```
curl -O https://repo.continuum.io/archive/
     Anaconda3-5.0.1-Linux-x86_64.sh

bash Anaconda3-5.0.1-Linux-x86_64.sh

conda create --name my_tf_env python=3

source activate my_tf_env
```

Install appropriate Tf version

```
conda install -c anaconda tensorflow

conda install -c anaconda tensorflow-gpu
```

Check if GPUs are properly seen

```
import tensorflow as tf

sess = tf.Session(config=tf.ConfigProto(
     log_device_placement=True)
```

If everything goes well...

```
name: GeForce GTX 1080 Ti
major: 6 minor: 1 memoryClockRate (GHz) 1.582
pciBusID 0000:82:00.0
Total memory: 10.92GiB
Free memory: 10.00GiB
2019-03-07 09:12:30.130236: I tensorflow/core/common_runtime/gpu/gpu_device.cc:976] DMA: 0
2019-03-07 09:12:30.130254: I tensorflow/core/common_runtime/gpu/gpu_device.cc:986] 0:   Y
2019-03-07 09:12:30.130266: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1045] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX 1080 Ti, pci bus id: 0000:82:00.0)
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: GeForce GTX 1080 Ti, pci bus id: 0000:82:00.0
```

Our First (Basic) Computation Graph

```python
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y+y+2

sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)

result = sess.run(f)

sess.close()
```

```python
import tensorflow as tf

init = tf.global_variables_initializer()

with tf.Session() as sess:
      init.run()
      result = f.eval()
```

Our First (Ugly) Neural Network

```python
import tensorflow as tf

n_inputs = 28*28
n_hidden_1 = 300
n_hidden_2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape(None, n_inputs),
    name="X")  # input layer
y = tf.placeholder(tf.int64, shape(None),
    name="y")     # output layer
```

- We want 2 hidden layers
- Introduce non-linearity

Our First (Ugly) Neural Network

```
def neuron_layer(X, n_neurons, name):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs + n_neurons)
        init = tf.truncated_normal((n_inputs,
            n_neurons), stddev=stddev)

        W = tf.Variable(init, name="weight_matrix")
        b = tf.Variable(tf.ones([n_neurons]),
            name="bias")
        Z = tf.matmul(X,W)+b

        return tf.nn.relu(Z)
```

```
with tf.name_scope("my_net"):
    hidden_1 = neuron_layer(X, n_hidden_1, name =
        "hidden_1")
    hidden_2 = neuron_layer(n_hidden_1, n_hidden_2,
        name = "hidden_2")
    logits = neuron_layer(hidden_2, n_outputs, name
        = "outputs")
```

Who wants to program a layer???!!!

Tf-Layers to the rescue!

```
with tf.name_scope("my_net"):
    hidden_1 = tf.layers.dense(X, n_hidden_1, name =
        "hidden_1", activation = tf.nn.relu)
    hidden_2 = tf.layers.dense(hidden_1, n_hidden_2,
        name = "hidden_2", activation = tf.nn.relu)
    logits = tf.layers.dense(hidden_2, n_outputs,
        name = "outputs")  # output before final
        activation function
```

However we still need to train our graph somehow:

- Cost function
- Optimizer

Our Loss

```
with tf.name_scope("loss"):
    entropy =
        tf.nn.sparse_softmax_cross_entropy_with_logits(labe
        logits=logits)
```

Our Optimizer

```
with tf.name_scope("train"):
    optimizer =
        tf.train.GradientDescentOptimizer(0.0001)
    training_op = optimizer.minimize(loss)
```

Our Evaluation Metric

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)  # Does
        the highest logit match with the class?
    accuracy = tf.reduce_mean(tf.cast(correct,
        tf.float32))
```

What we have done so far

- We created the placeholders for the inputs and targets
- Defined the layers of the network
- Defined an objective function to minimize
- Initialized the optimizer
- Defined a performance measure

$\Rightarrow$ End of the **Construction-Phase** of the graph

$\Rightarrow$ Let's move to the **Execution-Phase!**

Training Time

```
with tf.Session as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in
            range(mnist.train.num_examples //
            batch_size):
            X_batch, y_batch =
                mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X:
                X_batch, y: y_batch})

        training_acc = accuracy.eval(feed_dict={X:
            X_batch, y: y_batch})
        val_acc = accuracy.eval(feed_dict={X:
            mnist.validation_images, y:
            mnist_validation_labels})
```

Testing Time

```
with tf.Session as sess:
    saver.restore(sess, "./my_path/model.ckpt")
    Z =
        logits.eval(feed_dict={X:mnist.testing_images})
    y_pred = np.argmax(Z, axis = 1)
```

Activation Functions

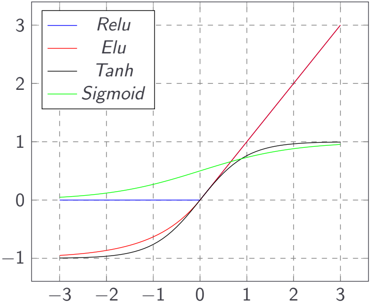

Figure: Graphical representation of some potential activation functions

- tf.nn.relu
- tf.nn.elu
- tf.nn.sigmoid
- tf.nn.tanh
- tf.nn.linear

```python
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden_layer = tf.layers.dense(X, n_hidden1,
    activation=leaky_relu, name="hidden1")
```

Cost Functions

- `tf.nn.mean_squared_error()`
- `tf.nn.sigmoid_cross_entropy()`
- `tf.nn.softmax_cross_entropy()`
- `tf.nn.cosine_distance()`
- `tf.nn.hinge_loss()`

Custom Loss Function

```
self.target_q_t = tf.placeholder('float32', [None],
    name='target_q_t')
self.action = tf.placeholder('int64', [None],
    name='action')

action_one_hot = tf.one_hot(self.action,
    self.env.action_size, 1.0, 0.0,
    name='action_one_hot')

q_acted = tf.reduce_sum(self.q * action_one_hot,
    reduction_indices=1, name='q_acted')

self.delta = self.target_q_t - q_acted

self.loss = tf.reduce_mean(clipped_error(self.delta),
    name='loss')
```

Optimizers

- `tf.train.GradientDescent()`
- `tf.train.AdadeltaOptimizer()`
- `tf.train.AdamOptimizer()`
- `tf.train.RMSPropOptimizer()`
- `tf.train.MomentumOptimizer()`

$\Rightarrow$ It is easy to change the default parameters and to minimize a custom objective function!

```
optim = tf.train.RMSPropOptimizer(
            self.learning_rate_op, momentum=0.95,
                epsilon=0.01).minimize(self.loss)
```

Avoiding Overfitting through Regularization

```
my_regularized_layer = partial(tf.layers.dense,
    activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(0.00

with tf.name_scope("regularized_dnn"):
     hidden_1 = my_regularized_layer(X, 100,
         name="reg_hidden1")
     hidden_2 = my_regularized_layer(hidden_1, 100,
         name="reg_hidden2")
     logits = my_regularized_layer(hidden_2,
         n_outputs, activation=None,
         name="output_layer")
```

Remember to add the regularized losses to your overall loss

```
reg_losses =
    tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name
    ="regularized_loss")
```

Our own regularizer

Max-Norm Regularization: $\sum_k \sum_i \sum_j (w_{i,j}^k)^2 \leq r$

```
def max_norm_regularizer(r, axes=1, name="max_norm",
    collection="max_norm"):

      def max_norm(w):
            clipped = tf.clip_by_norm(w, clip_norm = r,
                axes = axes)
            clip_weights = tf.assign(w, clipped, name =
                name)
            tf.add_to_collection(collection,
                clip_weights)

            return None     # no regularization loss is
                required this time

      return max_norm

hidden = tf.layers.dense(X, 100, activation =
    tf.nn.relu, kernel_regularizer =
    max_norm_regularizer, name = "hidden1")
```

Our own regularizer

```
with tf.name_scope("my_reg_nn"):
    hidden = tf.layers.dense(X, 100, activation =
        tf.nn.relu, kernel_regularizer =
        max_norm_regularizer, name = "hidden1")
```

Dropout

```
training = tf.placeholder_with_default(True,
    shape=(), "training")

dropout_rate = 0.2 # Reminder == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate,
    training=training)

with tf.name_scope("dropout_nn"):
    hidden_1 = tf.layers.dense(X, 100,
        activation=tf.nn.relu, name="hidden_1")
    hidden_1_dropout = tf.layers.dense(X_drop,
        dropout_rate, training=training)
```

Remember to set training to False at test time!

Keras

Sweet construction phase

- Super-simple
- Works only for single-input/output
- No more placeholders/scopes and sessions to initialize

```
import keras
from keras import layers

model = keras.Sequential()

model.add(layers.Dense(20, activation="relu",
    inpute_shape=(784,)))
model.add(layers.Dense(20, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

Sweet evaluation phase

- No more annoying loops
- No more `init.run()` nor `sess.run()`

```
model.compile(loss='categorical_crossentropy',
    optimizer='sgd', metrics=['accuracy'])
model.fit(X, y, epochs = 10, batch_size = 32)

model.evaluate(X_test, y_test)
```

The Functional API

- Similar to LEGO blocks
- Suitable for more complex architectures
- Allows multiple inputs/outputs

```python
import keras
from keras import layers

input_frames = keras.Input(shape=(84, 84, 4))

x = layers.Conv2D(32, (8, 8),
    activation='relu')(input_frames)
x = layers.Conv2D(64, (3, 3), activation='relu')(x)
x = layers.Conv2D(32, (3, 3), activation='relu')(x)

flattened_representation = layers.Flatten()(x)
shared_representation = layers.Dense(512,
    activation='relu')(flattened_representation)
```

The Functional API

```
q_outputs = layers.Dense(3, activation = 'linear')(shared_representation)
v_ouput = layers.Dense(1, activation = 'relu')(shared_representation)

model = keras.Model(inputs, outputs = [q_outputs, v_ouput])
```

Data Generators

```
import numpy as np

from keras import layers

X = np.load('my_huuuuuge_training_set.npy')
y = np.load('my_huuuuuuuuuuuge_labels.npy')

model = Sequential()
[...] # My fancy Neural Net
model.compile()

model.fit(X, y, epochs = 500, batch_size = 128)
```

1) We first define our Generator

```
from keras.preprocessing.image import
    ImageDataGenerator

data_generator = ImageDataGenerator(
                    rescale=1./255,
                    horizontal_flip = True,
                    vertical_flip = False,
                    height_shift_range = 0.15,
                    width_shift_range = 0.15,
                    rotation_range = 5,
                    shear_range = 0.01,
                    fill_mode = 'nearest',
                    zoom_range=0.25)
```

2) We yield appropriate batches of data

```
data_gen =
    data_generator.flow_from_directory(IMAGES_PATH,
                target_size = (224,224),
                class_mode = 'categorical',
                batch_size = 32,
                shuffle=True,
                seed=7)

while True:
     data = data_gen.next()
     yield [data[0], data[1]]
```

- The data_generator.* method depends on your dataset structure!

3) We appropriately fit our model

```
step_size_train = data_gen.n // batch_size

model.fit_generator(generator = data_gen,
                    steps_per_epoch = step_size_train,
                    callbacks=[csv_logger, tbCallBack]
                    epochs=10)
```

- Similarly for the evaluation phase there is a `model.evaluate_generator` method

Using ImageNet as a powerful Source-Domain

```
from keras.applications import ResNet50

base_model =
    ResNet50(weights='imagenet',include_top=False)

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024,activation='relu')(x)

preds = Dense(100,activation='softmax')(x)

model = Model(inputs = X, outputs = predictions)
```

Transfer Learning

- Which layers should use ImageNet's information?
- We can access this information as we would be dealing with lists

```
for layer in model.layers:
    layer.trainable=False

for layer in model.layers[20:]:
    layer.trainable=True
```

- One can explore which layers carry which features
- **Drastically reduce the amount of trainable parameters**

Callbacks

```
model.fit_generator(generator=training_generator
            ,...,
            callbacks=[csv_logger, tbCallBack]
         )
```

Monitoring your results

```
csv_logger = CSVLogger(results_path + '/log_' +
    dataset_name + '.csv', append=True,
    separator='\t')
```

Regularizing training

```
earlyStoppingCallBack =
    keras.callbacks.EarlyStopping(monitor='val_loss',
     patience=5, verbose=0, mode='auto')
```

Tensorboard is da bomb!

```
tbCallBack =
    keras.callbacks.TensorBoard(log_dir=tensorboard_path,
    histogram_freq=0, write_graph=True,
    write_images=True)
```

Callbacks

The CSVLogger Callback

- We can easily keep track of the training progress
- All results are neatily saved for us

```
epoch   tr_acc_1    tr_loss_1     val_loss_1     ...
0        0.854        3.965         6.726        0.654
1        0.8729       3.215         4.792        0.772
2        0.8601       2.624         4.078        0.797
```

⇒ At the end of training we can easily process and plot our results!

References & Credits

- *Géron, Aurélien. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.", 2017.*
- *Francois, Chollet. "Deep learning with Python." (2017).*
- `https://www.tensorflow.org/tutorials`
- `https://github.com/fchollet/keras-resources`

Now it is up to you! Have fun :)