https://navoshta.com/facial-with-tensorflow/

Alex Staravoitau

Detecting facial keypoints with TensorFlow

15 minute read

This is a TensorFlow follow-along for an amazing Deep Learning tutorial by Daniel Nouri. Daniel describes ways of approaching a computer vision problem of detecting facial keypoints in an image using various deep learning techniques, while these techniques gradually build upon each other, demonstrating advantages and limitations of each. I highly recommend going through the steps if you are interested in the topic and prefer learning by example.

However, Daniel uses Lasagne as a machine learning framework, and I'm currently learning to use TensorFlow, so I thought I would publish my follow-along tutorial where I'm utilising the very same approach, but using TensorFlow for building models on each of the steps. Daniel is using a set of different models that tend to gradually get more complicated (and perform better), so I did the same and broke down the tutorial into three Jupyter notebooks:

- **First model: a single hidden layer.** A very simple neural network.
- **Second model: convolutions.** Convolutional neural network with data augmentation, learning rate decay and dropout.
- **Third model: training specialists.** A pipeline of specialist CNNs with early stopping and supervised pre-training.

Let's take a look at them and check out the differences when it comes to TensorFlow. You can get the notebooks here:

Star

Fork

Download

First model: a single hidden layer.

This is a fairly simple model, so it was easy to recreate it in TensorFlow. If you are not familiar with TensorFlow framework, here is how it works: you first build a computation graph, which means you specify all variables you are planning to use, as well as all the relations across those variables. Then you evaluate specific variables from that graph that you are interested in, triggering computation of a path in the graph that leads to them. So in our case we will define a neural network structure and its loss, and will then train it by evaluating a TensorFlow loss optimiser, feeding it with batches of training data over and over again.

First, let's introduce a couple of handy functions that will help us defining model architecture.

```python
def fully_connected(input, size):
    weights = tf.get_variable( 'weights',
        shape = [input.get_shape()[1], size],
        initializer = tf.contrib.layers.xavier_initializer()
      )
    biases = tf.get_variable( 'biases',
        shape = [size],
        initializer=tf.constant_initializer(0.0)
      )
    return tf.matmul(input, weights) + biases
```

This function performs a single fully connected neural network layer pass. You only need to provide input and define number of units, it will work out the rest and initialise its weights. It's very handy, since now we can use the same function for defining as many fully connected layers as we like. Let's define our model structure and use this function for defining a hidden and output layers:

```python
def model_pass(input):
    with tf.variable_scope('hidden'):
        hidden = fully_connected(input, size = 100)
    relu_hidden = tf.nn.relu(hidden)
    with tf.variable_scope('out'):
        prediction = fully_connected(relu_hidden, size = num_keypoints)
    return prediction
```

This function performs a full model pass. It takes our array of features, passes it over to hidden layer (containing 100 units), then feeds the hidden output to output layer which in its turn produces vector of output values.

Please, note that we used `fully_connected()` function defined earlier for both layers, and thanks to TensorFlow's concept of `variable_scope` we didn't have to specify variables for weights and biases of each. You can think of it this way: in this example we implicitly create variables with the following names:

- `hidden/weights`
- `hidden/bias`
- `out/weights`
- `out/bias`

You don't have to use full names of those variables each time, instead you simply specify a block with *variable scope* — and whenever you try to get hold of a variable using `tf.get_variable()` within that block, the scope would be appended to each of your variables names.

Ok, now let's define our training graph. First, just as we did for each of the layers, we will use a variable scope for the whole model.

```python
# This model has 1 fully connected layer, we train it using batches of 36 examples
for 1000 epochs.
```

```
model_variable_scope = "1fc_b36_e1000"
```

So our variables would now have names like: `1fc_b36_e1000/hidden/weights`, `1fc_b36_e1000/hidden/bias` and so on.

Next thing we initialise a graph.

```
graph = tf.Graph()

with graph.as_default():
        ...
```

Strictly speaking we didn’t have to do that, as there is always a default graph and we could just use it. But where is fun in that?

Whatever comes in `with graph.as_default():` block defines our graph: all of the graph variables and their relations.

```
with graph.as_default():
    # Input data. For the training data, we use a placeholder that will be fed at run time with a training minibatch.
    tf_x_batch = tf.placeholder(tf.float32, shape = (None, image_size * image_size))
    tf_y_batch = tf.placeholder(tf.float32, shape = (None, num_keypoints))

    # Training computation.
    with tf.variable_scope(model_variable_scope):
        predictions = model_pass(tf_x_batch)

    loss = tf.reduce_mean(tf.square(predictions - tf_y_batch))

    # Optimizer.
    optimizer = tf.train.MomentumOptimizer(
        learning_rate = learning_rate,
        momentum = momentum,
        use_nesterov = True
    ).minimize(loss)
```

Here we define a couple of `tf.placeholder`s — these are not variables per se, they are, well, just placeholders. They are not trainable, and we don’t need to initialise them during graph build time. Instead, we provide what’s going to be in them during run time, while evaluating portions of our graph. Here we will use them to feed model with training examples in batches, and those examples will, of course, change after every weights update. **Note that we don’t explicitly specify batch size during graph build time, and instead use `None` as the first dimension of placeholders’ shapes.**

We then define computation of model predictions and loss, create an optimiser for our model and off we go!

Now we need to run that graph using `tf.Session` object. Every session has a graph, so we specify one when initialising our session. Also, before doing any computation you need to initialise all graph variables by running `tf.global_variables_initializer()`.

```
with tf.Session(graph = graph) as session:
        # Initialise all variables in the graph
    session.run(tf.global_variables_initializer())
        ...
```

Once we are in the scope of initialised session we can actually perform the training procedure:

```
for epoch in range(num_epochs):
    # Train on whole randomised dataset in batches
    batch_iterator = BatchIterator(batch_size = batch_size, shuffle = True)
    for x_batch, y_batch in batch_iterator(x_train, y_train):
        session.run([optimizer], feed_dict = {
                tf_x_batch : x_batch,
                tf_y_batch : y_batch
            }
        )
```

What happens here is that we ask the session to evaluate `optimizer`, which will implicitly run a sub-graph containing every variable that `optimizer` uses. From definition you can see it uses `loss` (value it is optimising), which in its turn uses `predictions`, etc. We also provide values that should be put in our data feeding `tf.placeholder`s by providing `feed_dict` parameter. This means that by the time computation of the path leading to `optimizer` begins, `tf_x_batch` and `tf_y_batch` placeholders would be holding `x_batch` and `y_batch` values respectively.

When training finishes we need to run our trained model on the testing data. We do this within the scope of the same `session`:

```
# Evaluate on test dataset (also in batches).
batch_iterator = BatchIterator(batch_size = 128)
predictions = []
for x_batch, _ in batch_iterator(x_test):
    [p_batch] = session.run([predictions], feed_dict = {
            tf_x_batch : x_batch
        }
    )
    predictions.extend(p_batch)

test_loss = np.mean(np.square(predictions - y_test))
print(" Test score: %.3f (loss = %.8f)" % (np.sqrt(test_loss) * 48.0, test_loss))
```

Since this operation would be performed on GPU by default (if you’re running a GPU version of TensorFlow), you may bump into your GPU’s memory limitations, therefore I’m suggesting batching your testing data as well.

I'm still using a tiny bit of Lasagne here, more specifically its `BatchIterator`. Further in tutorial Daniel uses this `BatchIterator`for data augmentation and it fits perfectly into the workflow. Also, as far as I'm aware TensorFlow lacks a similar plug-and-play component for iterating over data in batches, and one would have to define their own `tf.train.Example` type and setup a pipeline for `tf.TFRecordReader`, feeding it to the model with a `tf.train.QueueRunner`. Although this seems like a bucket of joy, I thought I would go with a plain vanilla `BatchIterator`, and concentrate on building a model instead. Data feeding in TensorFlow seems to be a broad topic, and would make a good article on its own!

As you see we only supply `tf_x_batch` value in the `feed_dict`, since we only evaluate `predictions`variable here, and its path in the graph does not involve `tf_y_batch` — we are not calculating `loss` as a part of this computation after all.

One of the neat Lasagne features is keeping track of training history by logging validation and training losses. As far as I'm aware TensorFlow doesn't do that for you, so we will have to come up with some other solution.

One might be tempted to use `tf.train.SummaryWriter`s and visualise data using `TensorBoard`, and actually that's exactly what I did at first. I even managed to plot training and validation losses on the same graph and overcome a couple of other issues, but in the end `tf.train.SummaryWriter` seemed to slow down training process quite a bit. I'm not sure if it was due to me not using it correctly, or it's just the way it works, but I got much better results in terms of speed using simple arrays, saving them to disk and plotting losses with `matplotlib`.

First let's refactor out part where we evaluate model on the testing dataset into a function. We're going to use it quite a lot, as the plan is to periodically get validation and training datasets predictions during training and logging those losses:

```python
def get_predictions_in_batches(X, session):
    """
    Calculates predictions in batches of 128 examples at a time, using `session`'s 
calculation graph.
    """
    p = []
    batch_iterator = BatchIterator(batch_size = 128)
    for x_batch, _ in batch_iterator(X):
        [p_batch] = session.run([predictions], feed_dict = {
                tf_x_batch : x_batch 
            }
        )
        p.extend(p_batch)
    return p
```

This function here is just a convenient way of getting predictions for a dataset on the model's weights it has learned so far.

Now let's add a couple of arrays: `train_loss_history` and `valid_loss_history` for keeping track of training and validation losses respectively. Let's rewrite our training code as follows:

```
def calc_loss(predictions, labels):
    """
    Squared mean error for given predictions.
    """
    return np.mean(np.square(predictions - labels))

with tf.Session(graph = graph) as session:
    tf.initialize_all_variables().run()

    train_loss_history = np.zeros(num_epochs)
    valid_loss_history = np.zeros(num_epochs)

    print("============ TRAINING =============")
    for epoch in range(num_epochs):
        # Train on whole randomised dataset in batches
        batch_iterator = BatchIterator(batch_size = batch_size, shuffle = True)
        for x_batch, y_batch in batch_iterator(x_train, y_train):
            session.run([optimizer], feed_dict = {
                    tf_x_batch : x_batch,
                    tf_y_batch : y_batch
                }
            )

        # Another epoch ended, let's log our losses.
        # Get training data predictions and log training loss:
        train_loss = calc_loss(
            get_predictions_in_batches(x_train, session),
            y_train
        )
        train_loss_history[epoch] = train_loss

        # Get validation data predictions and log validation loss:
        valid_loss = calc_loss(
            get_predictions_in_batches(x_valid, session),
            y_valid
        )
        valid_loss_history[epoch] = valid_loss

        if (epoch % 100 == 0):
            print("--------- EPOCH %4d/%d ---------" % (epoch, num_epochs))
            print("     Train loss: %.8f" % (train_loss))
            print("Validation loss: %.8f" % (valid_loss))

    # Evaluate on test dataset.
    test_loss = calc_loss(
        get_predictions_in_batches(x_test, session),
        y_test
    )
    print("===================================")
    print(" Test score: %.3f (loss = %.8f)" % (np.sqrt(test_loss) * 48.0, test_loss))
    np.savez(os.getcwd() + "/train_history", train_loss_history = train_loss_history,
valid_loss_history = valid_loss_history)
```

You can now load training history from file and use Daniel’s code to plot learning curves and see how your model is performing.

```
model_history = np.load(os.getcwd() + "/train_history.npz")
train_loss = model_history["train_loss_history"]
valid_loss = model_history["valid_loss_history"]
x_axis = np.arange(num_epochs)
pyplot.plot(x_axis, train_loss, "b-", linewidth=2, label="train")
pyplot.plot(x_axis, valid_loss, "g-", linewidth=2, label="valid")
pyplot.grid()
pyplot.legend()
pyplot.xlabel("epoch")
pyplot.ylabel("loss")
pyplot.ylim(0.0005, 0.01)
pyplot.xlim(0, num_epochs)
pyplot.yscale("log")
pyplot.show()
```

You may want to only log losses every, say, 5 or 10 epochs, as evaluating on the whole training set does take a while. However, you may need validation loss later on in order to implement early stopping.

Second model: convolutions.

In the second model we will add convolutions, which should improve model performance significantly. Let’s declare a couple of additional convenience functions:

```
def conv_relu(input, kernel_size, depth):
    weights = tf.get_variable( 'weights',
        shape = [kernel_size, kernel_size, input.get_shape()[3], depth],
        initializer = tf.contrib.layers.xavier_initializer()
      )
    biases = tf.get_variable( 'biases',
        shape = [depth],
        initializer=tf.constant_initializer(0.0)
      )
    conv = tf.nn.conv2d(input, weights,
        strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)
```

This one will perform a convolutional layer pass followed by a rectified linear unit (since usually those two are applied together). As you see we’re using `tf.get_variable()` again, so we can reuse this function with different layers by simply providing variable scope. Let’s add a couple of other helper functions to make encoding of our model architecture easier:

```
def fully_connected_relu(input, size):
    return tf.nn.relu(fully_connected(input, size))

def pool(input, size):
    return tf.nn.max_pool(
        input,
        ksize=[1, size, size, 1],
        strides=[1, size, size, 1],
        padding='SAME'
    )
```

Ok, with these routines we can now encode our full model pass.

```
def model_pass(input, training):
    # Convolutional layers
    with tf.variable_scope('conv1'):
        conv1 = conv_relu(input, kernel_size = 3, depth = 32)
    with tf.variable_scope('pool1'):
        pool1 = pool(conv1, size = 2)
        # Apply dropout if needed
        pool1 = tf.cond(training, lambda: tf.nn.dropout(pool1, keep_prob = 0.9 if
dropout else 1.0), lambda: pool1)
    with tf.variable_scope('conv2'):
        conv2 = conv_relu(pool1, kernel_size = 2, depth = 64)
    with tf.variable_scope('pool2'):
        pool2 = pool(conv2, size = 2)
        # Apply dropout if needed
        pool2 = tf.cond(training, lambda: tf.nn.dropout(pool2, keep_prob = 0.8 if
dropout else 1.0), lambda: pool2)
    with tf.variable_scope('conv3'):
        conv3 = conv_relu(pool2, kernel_size = 2, depth = 128)
    with tf.variable_scope('pool3'):
        pool3 = pool(conv3, size = 2)
        # Apply dropout if needed
        pool3 = tf.cond(training, lambda: tf.nn.dropout(pool3, keep_prob = 0.7 if
dropout else 1.0), lambda: pool3)

    # Flatten convolutional layers output
    shape = pool3.get_shape().as_list()
    flattened = tf.reshape(pool3, [-1, shape[1] * shape[2] * shape[3]])

    # Fully connected layers
    with tf.variable_scope('fc4'):
        fc4 = fully_connected_relu(flattened, size = 1000)
        # Apply dropout if needed
        fc4 = tf.cond(training, lambda: tf.nn.dropout(fc4, keep_prob = 0.5 if dropout
else 1.0), lambda: fc4)
    with tf.variable_scope('fc5'):
        fc5 = fully_connected_relu(fc4, size = 1000)
    with tf.variable_scope('out'):
        prediction = fully_connected(fc5, size = num_keypoints)
    return prediction
```

Please note those weird assignments:

```
fc4 = tf.cond(training, lambda: tf.nn.dropout(fc4, keep_prob = 0.5 if dropout else
1.0), lambda: fc4)
```

Let’s break it down a bit.

First we calculate `0.5 if dropout else 1.0`, which means that we conditionally apply dropout, if `dropout` flag is set to `True`. This is done so that later you could compare how same model performs with and without dropout.

Furthermore, we only want to apply dropout while training, and not while evaluating our model, that’s why we put the assignment into a `tf.cond(training, lambda: ..., lambda: fc4)` block. It means that if `training` (being a `tf.Variable`) is `True`, we will apply dropout, and will simply assign `fc4` to itself otherwise.

Also note that we have to manually flatten convolutional layers’ output before passing it over to fully connected layers.

A couple of new things you may notice in the TensorFlow graph are `is_training` flag, learning rate decay and momentum increase. The `is_training` flag is another TensorFlow placeholder we use to indicate if we’re training or evaluating. In latter case model pipeline function won’t apply dropout. I implemented momentum increase in plain Python by checking how far have we gone into the maximum number of epochs. As for learning rate decay, there is a nice TensorFlow function for that: `tf.train.exponential_decay()` lets you do exactly that, providing number of decay steps and decay rate.

The rest should be familiar from the first notebook. Please, mind that some optimisation options are defined as flags (for instance, `data_augmentation`, `learning_rate_decay`, etc.) which are encoded into model name. This is done so that you could compare performance with different optimisation techniques applied. Just provide the name of the model as a parameter of `plot_learning_curves()` method and learning curves of that model would be drawn on top of the current plot:

```
new_model_epochs = plot_learning_curves()
old_model_epochs = plot_learning_curves("1fc_b36_e1000", linewidth = 1)
pyplot.ylim(0.001, 0.01)
pyplot.xlim(0, max(new_model_epochs, old_model_epochs))
pyplot.show()
```

Third model: training specialists.

The third notebook implements the most advanced model of this tutorial: training specialists for groups of facial keypoints. It also covers another great technique for battling overfitting: *early stopping*. `EarlyStopping` class from Daniel’s tutorial requires one crucial modification when working with TensorFlow: in order to save and restore trained weights we need a reference to current TensorFlow session and `tf.train.Saver` object. TensorFlow `Saver` is doing exactly what you would expect it to do: lets you easily save and restore variables from your session graph —

for instance, trained weights. You simply call `save()` to save current weights to a checkpoint file, or `restore()` in order to load those weights into your session's graph.

```
saver.save(session, checkpoint_path)
# Weights are now saved in a file located at `checkpoint_path`.

saver.restore(session, checkpoint_path)
# Saved weights are loaded into corresponding variables again.
```

As easy as that! One thing to consider is that when restoring session your graph (e.g. variables's names and relations) is expected to be exactly the same as it was during saving, so that `saver`knows which weights to load where. The easiest way to do so is saving and restoring the graph with `tf.train.export_meta_graph` and `tf.train.import_meta_graph` functions.

However, what if your graph is *not* the same? Well, actually we run into exactly this problem when reusing previously trained model as a specialist. The idea is that we initialise weights for each specialist with values from a pre-trained model (the one we implemented in notebook #2 — *3con_2fc_b36_e1000_aug_lrdec_mominc_dr*). Unfortunately, the graph for a single specialist is not going to be the same due to a different shape of the `out` layer, e.g. number of keypoints the model provides as an output. Also, we are using a different variable scope. In order to fix that we do the following:

```
spec_var_scope = "specialist_variable_scope"
initialising_model = "3con_2fc_b36_e1000_aug_lrdec_mominc_dr"

# Exclude output layer weights from variables we will restore
variables_to_restore = [v for v in tf.global_variables() if "/out/" not in v.op.name]

# Replace variables scope with that of the current model
loader = tf.train.Saver({v.op.name.replace(spec_var_scope, initialising_model): v for
v in variables_to_restore})

loader.restore(session, "/3con_2fc_b36_e1000_aug_lrdec_mominc_dr/model.ckpt")
```

By default `tf.train.Saver` will restore all variables you have in your graph. However, you can provide a list of variables to be restored — that's how we are going to exclude output layer weights from the list of the values we are restoring. Important thing to remember is that variable scope is pretty much a namespace that is encoded in the variable name, separated by slashes (`/`). That's why we simply filter out variables with `/out/` in their names from all variables in the graph:

```
variables_to_restore = [v for v in tf.all_variables() if "/out/" not in v.op.name]
```

Next thing to do is updating the variable scope. This is done, again, by simply updating variables' names, e.g. by replacing occurences of the old model name with your current variable scope:

```
...{v.op.name.replace(spec_var_scope, initialising_model): v for v in
variables_to_restore}
```

Let’s assume this is what your saved model looks like. Say, it has a graph with the following variables:

- `3con_2fc_b36_e1000_aug_lrdec_mominc_dr/fc4/weights`
- `3con_2fc_b36_e1000_aug_lrdec_mominc_dr/fc4/biases`
- `3con_2fc_b36_e1000_aug_lrdec_mominc_dr/out/weights`
- `3con_2fc_b36_e1000_aug_lrdec_mominc_dr/out/biases`

After we apply our transformations it is converted to:

- `specialist_variable_scope/fc4/weights`
- `specialist_variable_scope/fc4/biases`

And `.../out/weights` and `.../out/biases` are gone, since they had `/out/` in their names.

You can now plot learning curves for each of the speclialists, and, as Daniel suggests, explore ways of improving your model even further. As he points out, some specialists overfit more than the others, so there might be sense in using different dropout values for each of them. One might want to also experiment with additional regularisation techniques, like L2 loss, and probably take some further steps with data augmentation.