Training

Initialize and train a TensorFlow neural network using actual training data.

Chapter Goals:

- Learn how to feed data values into a neural network
- Understand how to run a neural network using input values
- Train the neural network on batched input data and labels

A. Running the model

In this chapter and the next, you will be running your model on input data, using a `tf.Session` object and the `tf.placeholder` variables from the previous chapters.

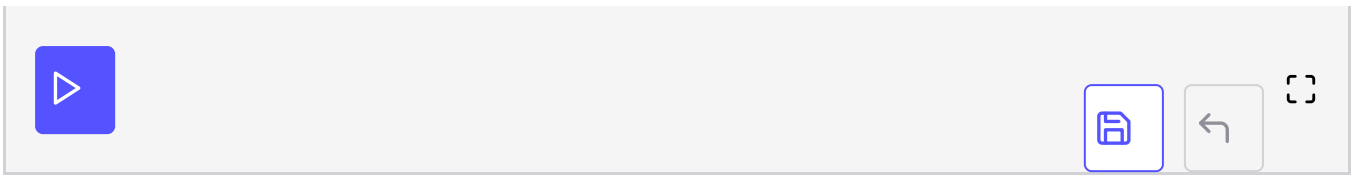
The `tf.Session` object has an extremely important function called `run`. All the code written in the previous chapters was to build the computation graph of the neural network, i.e. its layers and operations. However, we can only train or evaluate the model on real input data using `run`. The function takes in a single required argument and a few keyword arguments.

B. Using `run`

The required argument is normally either a single tensor/operation or a list/tuple of tensors and operations. Calling `run` on a tensor returns the value of that tensor after executing our computation graph. The output of `run` with a tensor input is a NumPy array.

The code below shows usages of `run` on `tf.constant` tensors.

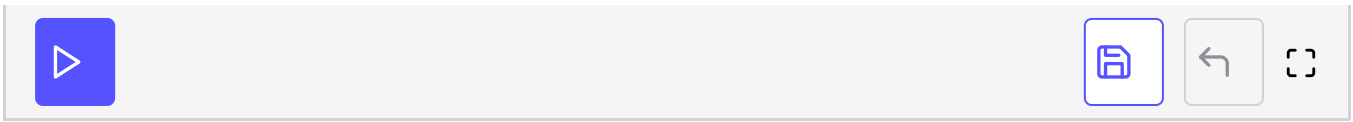
```
t = tf.constant([1, 2, 3])
sess = tf.Session()
arr = sess.run(t)
print('{}\n'.format(repr(arr)))

t2 = tf.constant(4)
tup = sess.run((t, t2))
print('{}\n'.format(repr(tup)))
```



Of the keyword arguments for `run`, the important one for most applications is `feed_dict`. The `feed_dict` is a python dictionary. Each key is a *tensor* from the model's computation graph. The key's value can be a Python scalar, list, or NumPy array.

We use `feed_dict` to pass values into certain tensors in the computation graph. In the code below, we pass in a value for `inputs`, which is a `tf.placeholder` object.

```
inputs = tf.placeholder(tf.float32, shape=(None, 2))
feed_dict = {
  inputs: [[1.1, -0.3],
           [0.2, 0.1]]
}
sess = tf.Session()
arr = sess.run(inputs, feed_dict=feed_dict)
print('{}\n'.format(repr(arr)))
```



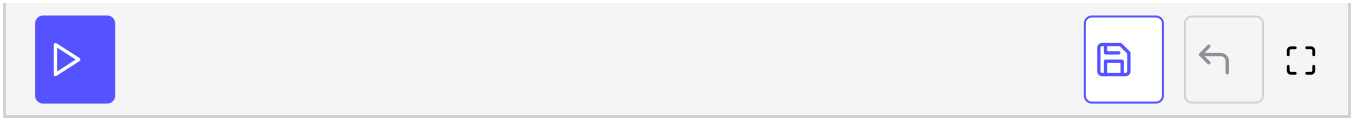
Each `tf.placeholder` object used in the model execution must be included as a key in the `feed_dict`, with the corresponding value's shape and type matching the placeholder's.

C. Initializing variables

When we call `run`, every tensor in the model's computation graph must either already have a value or must be fed in a value through `feed_dict`. However, when we start training from scratch, none of our variables (e.g. weights) have values yet. We need to initialize all the variables using `tf.global_variables_initializer`. This returns an operation that, when used as the required argument in `run`, initializes all the variables in the model.

In the code below, the variables that are initialized are part of `tf.layers.dense`. The variable initialization process is defined internally by the function. In this case, the variables are initialized in a way that results in zero logits.

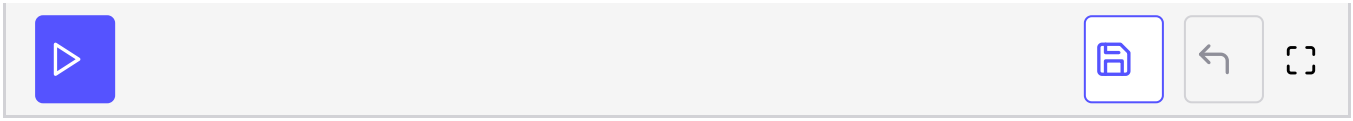
```
inputs = tf.placeholder(tf.float32, shape=(None, 2))
feed_dict = {
  inputs: [[1.1, -0.3],
           [0.2, 0.1]]
}
logits = tf.layers.dense(inputs, 1, name='logits')
init_op = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init_op) # variable initialization
arr = sess.run(logits, feed_dict=feed_dict)
print('{}\n'.format(repr(arr)))
```



D. Training logistics

The `num_steps` argument represents the number of iterations we use to train our model. Each iteration we train the model on a *batch* of data points. So `input_data` is essentially a large dataset divided into chunks (i.e. batches), and each iteration we train on a specific batch of points and their corresponding labels.

```
# predefined dataset
print('Input data:')
print('{}\n'.format(repr(input_data)))

print('Labels:')
print('{}\n'.format(repr(input_labels)))
```



Example dataset with a batch size of 3.

The batch size determines how the model trains. Larger batch sizes usually result in faster training but less accurate models, and vice-versa for smaller batch sizes. Choosing the batch size is a speed-precision tradeoff.

When training a neural network, it's usually a good idea to print out the loss every so often, so you know that the model is training correctly and to stop the training when the loss has converged to a minimum.

Time to Code!

The coding exercise for this chapter sets up the training utility, using the ADAM training operation and model layers from the previous chapters.

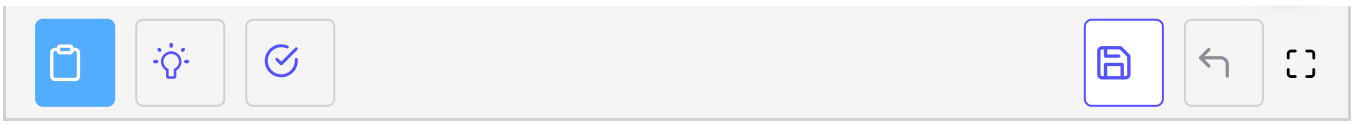
We first need to initialize all the model variables (e.g. the variables via `tf.layers.dense`).

To do this, we'll use `tf.Session` and `tf.global_variables_initializer`.

Set `init_op` equal to `tf.global_variables_initializer()`.

Create a `tf.Session` object named `sess`, and call its `run` function on `init_op`.

```
# CODE HERE
```



After initializing the variables, we can run our model training. We'll run the training for 1000 steps. We provide a `for` loop for you, which iterates 1000 steps.

The rest of the code for this chapter goes inside the `for` loop.

We provide the input dataset (`input_data`) and labels (`input_labels`) as NumPy arrays, initialized in the backend. The placeholders for the data and labels, `inputs` and `labels`, are also predefined in the backend (using your code from Chapter 2).

Each iteration of the `for` loop we will feed in the i^{th} data observation (and its corresponding label) into the model.

Set `feed_dict` equal to a python dictionary with key-value pairs `inputs: input_data[i]` and `labels: input_labels[i]`.

The ADAM training operation (`train_op`) is defined in the backend, using the code from Chapter 5. Using `sess.run`, we can run training on the input data and labels.

Using the `sess` object defined earlier, call the `run` function with first

argument `train_op` and keyword argument `feed_dict=feed_dict`.

```
# input_data, input_labels, inputs, labels, train_op
# are all predefined in the backend
for i in range(1000):
  # CODE HERE
  pass
```