Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle, then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, but will also require much less training data.

For example, suppose that you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, so you should try to reuse parts of the first network (see Figure 11-4).

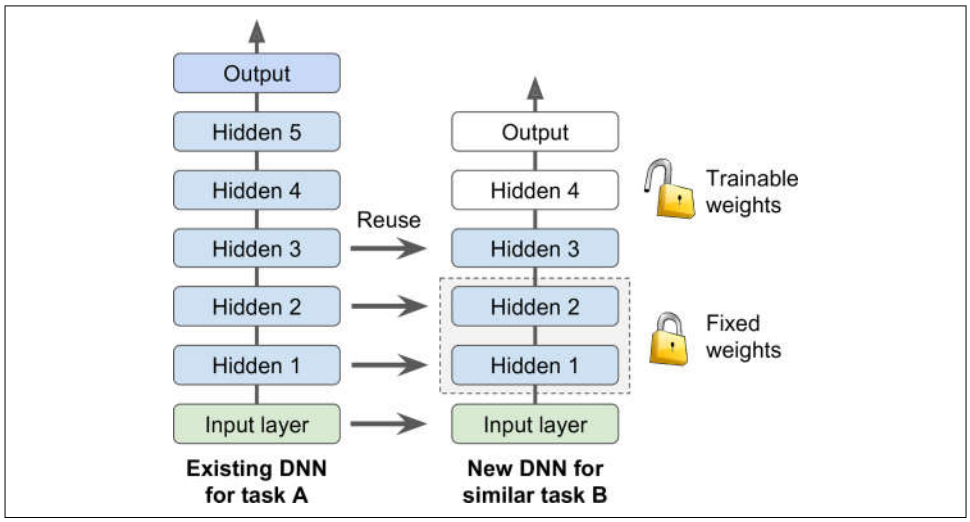

Figure 11-4. Reusing pretrained layers

If the input pictures of your new task don't have the same size as the ones used in the original task, you will have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will only work well if the inputs have similar low-level features.

Reusing a TensorFlow Model

If the original model was trained using TensorFlow, you can simply restore it and train it on the new task. As we discussed in Chapter 9, you can use the `import_meta_graph()` function to import the operations into the default graph. This returns a `Saver` that you can later use to load the model's state:

```
saver = tf.train.import_meta_graph("./my_model_final.ckpt.meta")
```

You must then get a handle on the operations and tensors you will need for training. For this, you can use the graph's `get_operation_by_name()` and `get_tensor_by_name()` methods. The name of a tensor is the name of the operation that outputs it followed by `:0` (or `:1` if it is the second output, `:2` if it is the third, and so on):

```
X = tf.get_default_graph().get_tensor_by_name("X:0")
y = tf.get_default_graph().get_tensor_by_name("y:0")
accuracy = tf.get_default_graph().get_tensor_by_name("eval/accuracy:0")
training_op = tf.get_default_graph().get_operation_by_name("GradientDescent")
```

If the pretrained model is not well documented, then you will have to explore the graph to find the names of the operations you will need. In this case, you can either explore the graph using TensorBoard (for this you must first export the graph using a `FileWriter`, as discussed in Chapter 9), or you can use the graph's `get_operations()` method to list all the operations:

```
for op in tf.get_default_graph().get_operations():
    print(op.name)
```

If you are the author of the original model, you could make things easier for people who will reuse your model by giving operations very clear names and documenting them. Another approach is to create a collection containing all the important operations that people will want to get a handle on:

```
for op in (X, y, accuracy, training_op):
    tf.add_to_collection("my_important_ops", op)
```

This way people who reuse your model will be able to simply write:

```
X, y, accuracy, training_op = tf.get_collection("my_important_ops")
```

You can then restore the model's state using the `Saver` and continue training using your own data:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    [...] # train the model on your own data
```

Alternatively, if you have access to the Python code that built the original graph, you can use it instead of `import_meta_graph()`.

In general, you will want to reuse only part of the original model, typically the lower layers. If you use `import_meta_graph()` to restore the graph, it will load the entire original graph, but nothing prevents you from just ignoring the layers you do not care about. For example, as shown in Figure 11-4, you could build new layers (e.g., one hidden layer and one output layer) on top of a pretrained layer (e.g., pretrained hidden layer 3). You would also need to compute the loss for this new output, and create an optimizer to minimize that loss.

If you have access to the pretrained graph's Python code, you can just reuse the parts you need and chop out the rest. However, in this case you need a `Saver` to restore the pretrained model (specifying which variables you want to restore; otherwise, TensorFlow will complain that the graphs do not match), and another `Saver` to save the new model. For example, the following code restores only hidden layers 1, 2, and 3:

```
[...] # build the new model with the same hidden layers 1-3 as before

reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]") # regular expression
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # to restore layers 1-3

init = tf.global_variables_initializer() # to init all variables, old and new
saver = tf.train.Saver() # to save the new model

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")
    [...] # train the model
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

First we build the new model, making sure to copy the original model's hidden layers 1 to 3. Then we get the list of all variables in hidden layers 1 to 3, using the regular expression `"hidden[123]"`. Next, we create a dictionary that maps the name of each variable in the original model to its name in the new model (generally you want to keep the exact same names). Then we create a `Saver` that will restore only these variables. We also create an operation to initialize all the variables (old and new) and a second `Saver` to save the entire new model, not just layers 1 to 3. We then start a session and initialize all variables in the model, then restore the variable values from the original model's layers 1 to 3. Finally, we train the model on the new task and save it.

The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, you can try keeping all the hidden layers and just replace the output layer.

Reusing Models from Other Frameworks

If the model was trained using another framework, you will need to load the model parameters manually (e.g., using Theano code if it was trained with Theano), then assign them to the appropriate variables. This can be quite tedious. For example, the following code shows how you would copy the weight and biases from the first hidden layer of a model trained using another framework:

```
original_w = [...] # Load the weights from the other framework
original_b = [...] # Load the biases from the other framework
```

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
[...] # Build the rest of the model

# Get a handle on the assignment nodes for the hidden1 variables
graph = tf.get_default_graph()
assign_kernel = graph.get_operation_by_name("hidden1/kernel/Assign")
assign_bias = graph.get_operation_by_name("hidden1/bias/Assign")
init_kernel = assign_kernel.inputs[1]
init_bias = assign_bias.inputs[1]

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init, feed_dict={init_kernel: original_w, init_bias: original_b})
    # [...] Train the model on your new task
```

In this implementation, we first load the pretrained model using the other framework (not shown here), and we extract from it the model parameters we want to reuse. Next, we build our TensorFlow model as usual. Then comes the tricky part: every TensorFlow variable has an associated assignment operation that is used to initialize it. We start by getting a handle on these assignment operations (they have the same name as the variable, plus `"/Assign"`). We also get a handle on each assignment operation's second input: in the case of an assignment operation, the second input corresponds to the value that will be assigned to the variable, so in this case it is the variable's initialization value. Once we start the session, we run the usual initialization operation, but this time we feed it the values we want for the variables we want to reuse. Alternatively, we could have created new assignment operations and placeholders, and used them to set the values of the variables after initialization. But why create new nodes in the graph when everything we need is already there?

Freezing the Lower Layers

It is likely that the lower layers of the first DNN have learned to detect low-level features in pictures that will be useful across both image classification tasks, so you can just reuse these layers as they are. It is generally a good idea to "freeze" their weights when training the new DNN: if the lower-layer weights are fixed, then the higher-layer weights will be easier to train (because they won't have to learn a moving target). To freeze the lower layers during training, one solution is to give the optimizer the list of variables to train, excluding the variables from the lower layers:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

The first line gets the list of all trainable variables in hidden layers 3 and 4 and in the output layer. This leaves out the variables in the hidden layers 1 and 2. Next we pro-

vide this restricted list of trainable variables to the optimizer's `minimize()` function. Ta-da! Layers 1 and 2 are now frozen: they will not budge during training (these are often called *frozen layers*).

Another option is to add a `stop_gradient()` layer in the graph. Any layer below it will be frozen:

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                              name="hidden1") # reused frozen
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                              name="hidden2") # reused frozen
    hidden2_stop = tf.stop_gradient(hidden2)
    hidden3 = tf.layers.dense(hidden2_stop, n_hidden3, activation=tf.nn.relu,
                              name="hidden3") # reused, not frozen
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu,
                              name="hidden4") # new!
    logits = tf.layers.dense(hidden4, n_outputs, name="outputs") # new!
```

Caching the Frozen Layers

Since the frozen layers won't change, it is possible to cache the output of the topmost frozen layer for each training instance. Since training goes through the whole dataset many times, this will give you a huge speed boost as you will only need to go through the frozen layers once per training instance (instead of once per epoch). For example, you could first run the whole training set through the lower layers (assuming you have enough RAM), then during training, instead of building batches of training instances, you would build batches of outputs from hidden layer 2 and feed them to the training operation:

```
import numpy as np

n_batches = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")

    h2_cache = sess.run(hidden2, feed_dict={X: mnist.train.images})

    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(mnist.train.num_examples)
        hidden2_batches = np.array_split(h2_cache[shuffled_idx], n_batches)
        y_batches = np.array_split(mnist.train.labels[shuffled_idx], n_batches)
        for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
            sess.run(training_op, feed_dict={hidden2:hidden2_batch, y:y_batch})

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

The last line of the training loop runs the training operation defined earlier (which does not touch layers 1 and 2), and feeds it a batch of outputs from the second hidden layer (as well as the targets for that batch). Since we give TensorFlow the output of hidden layer 2, it does not try to evaluate it (or any node it depends on).

Tweaking, Dropping, or Replacing the Upper Layers

The output layer of the original model should usually be replaced since it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

Try freezing all the copied layers first, then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freeze all remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even add more hidden layers.

Model Zoos

Where can you find a neural network trained for a task similar to the one you want to tackle? The first place to look is obviously in your own catalog of models. This is one good reason to save all your models and organize them so you can retrieve them later easily. Another option is to search in a *model zoo*. Many people train Machine Learning models for various tasks and kindly release their pretrained models to the public.

TensorFlow has its own model zoo available at *https://github.com/tensorflow/models*. In particular, it contains most of the state-of-the-art image classification nets such as VGG, Inception, and ResNet (see Chapter 13, and check out the *models/slim* directory), including the code, the pretrained models, and tools to download popular image datasets.

Another popular model zoo is Caffe's Model Zoo. It also contains many computer vision models (e.g., LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, inception) trained on various datasets (e.g., ImageNet, Places Database, CIFAR10, etc.). Saumitro Dasgupta wrote a converter, which is available at *https://github.com/ethereon/caffe-tensorflow*.

Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose all hope! First, you should of course try to gather more labeled training data, but if this is too hard or too expensive, you may still be able to perform *unsupervised pretraining* (see Figure 11-5). That is, if you have plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer and then going up, using an unsupervised feature detector algorithm such as *Restricted Boltzmann Machines* (RBMs; see Appendix E) or autoencoders (see Chapter 15). Each layer is trained on the output of the previously trained layers (all layers except the one being trained are frozen). Once all layers have been trained this way, you can fine-tune the network using supervised learning (i.e., with backpropagation).

This is a rather long and tedious process, but it often works well; in fact, it is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning. Until 2010, unsupervised pretraining (typically using RBMs) was the norm for deep nets, and it was only after the vanishing gradients problem was alleviated that it became much more common to train DNNs purely using backpropagation. However, unsupervised pretraining (today typically using autoencoders rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

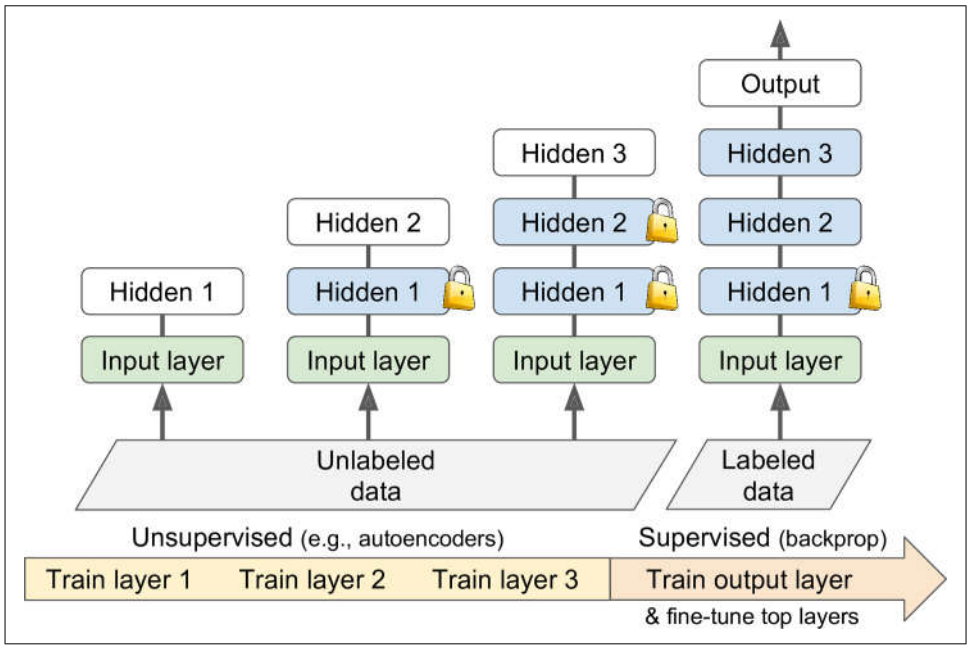

Figure 11-5. Unsupervised pretraining

Pretraining on an Auxiliary Task

One last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. However, you could gather a lot of pictures of random people on the internet and train a first neural network to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier using little training data.

It is often rather cheap to gather unlabeled training examples, but quite expensive to label them. In this situation, a common technique is to label all your training examples as "good," then generate many new training instances by corrupting the good ones, and label these corrupted instances as "bad." Then you can train a first neural network to classify instances as good or bad. For example, you could download millions of sentences, label them as "good," then randomly change a word in each sentence and label the resulting sentences as "bad." If a neural network can tell that "The

dog sleeps" is a good sentence but "The dog they" is bad, it probably knows quite a lot about language. Reusing its lower layers will likely help in many language processing tasks.

Another approach is to train a first network to output a score for each training instance, and use a cost function that ensures that a good instance's score is greater than a bad instance's score by at least some margin. This is called *max margin learning*.

Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network. Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam optimization.

Momentum Optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*, proposed by Boris Polyak in 1964.[10] In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regards to the weights ($\nabla_\theta J(\theta)$) multiplied by the learning rate η. The equation is: $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* **m** (multiplied by the learning rate η), and it updates the weights by simply adding this momentum vector (see Equation 11-4). In other words, the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperpara-

10 "Some methods of speeding up the convergence of iteration methods," B. Polyak (1964).