11. Training Deep Neural Nets

Chapter 11. Training Deep Neural Nets

In Chapter 10 we introduced artificial neural networks and trained our first deep neural network. But it was a very shallow DNN, with only two hidden layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with (say) 10 layers, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

- First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, with such a large network, training would be extremely slow.
- Third, a model with millions of parameters would severely risk overfitting the training set.

In this chapter, we will go through each of these problems in turn and present techniques to solve them. We will start by explaining the vanishing gradients problem and exploring some of the most popular solutions to this problem. Next we will look at various optimizers that can speed up training large models tremendously compared to plain Gradient Descent. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets: welcome to Deep Learning!

Vanishing/Exploding Gradients Problems

As we discussed in Chapter 10, the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which is mostly encountered in recurrent neural networks (see Chapter 14). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it. A paper titled “Understanding the Difficulty of Training Deep Feedforward Neural Networks” by Xavier Glorot and Yoshua Bengio found a few suspects, including the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a normal distribution with a mean of 0 and a standard deviation of 1. In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than

the logistic function in deep networks).

Looking at the logistic activation function (see Figure 11-1), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

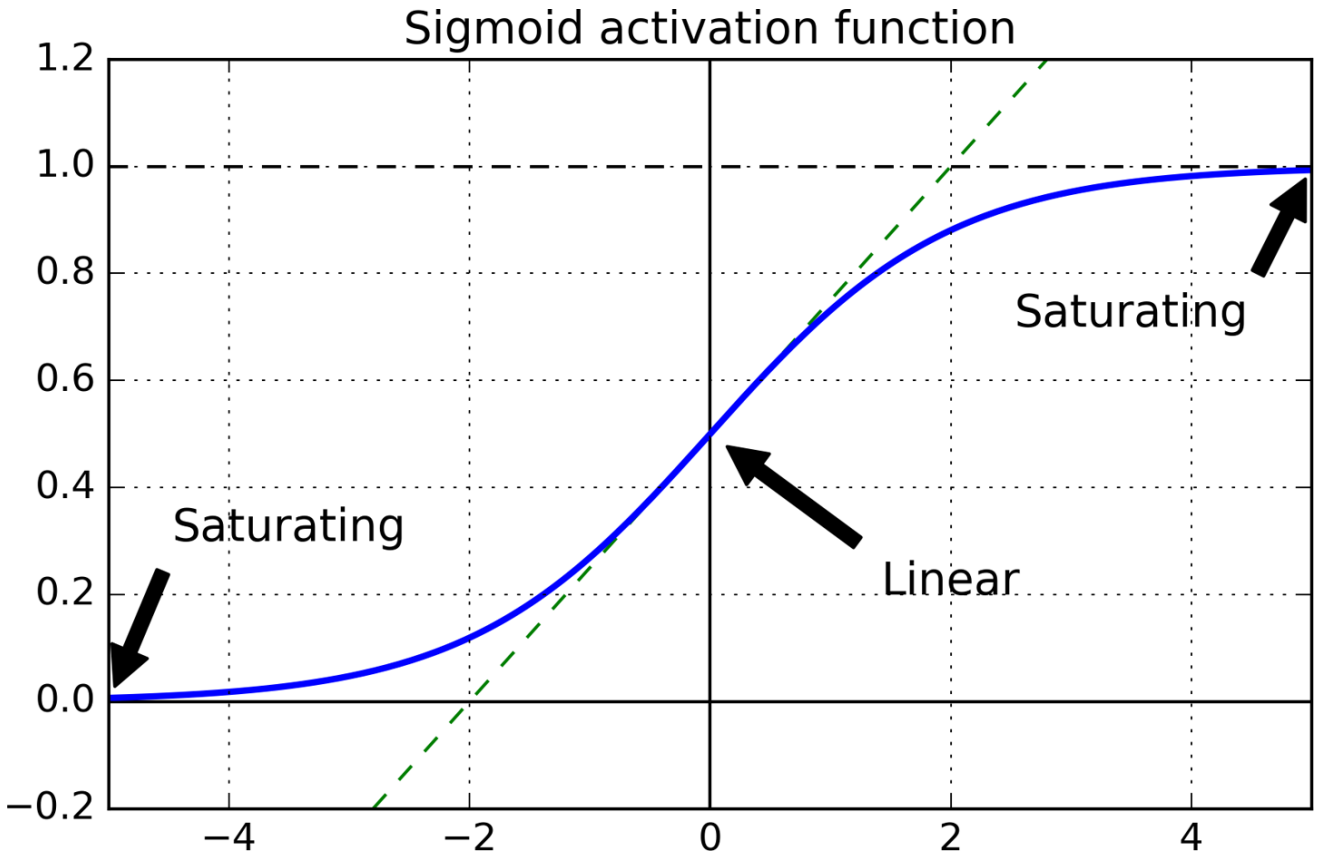

Figure 11-1. Logistic activation function saturation

Xavier and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of input and output connections, but they proposed a good compromise that has proven to work very well in practice: the connection weights must be initialized randomly as described in Equation 11-1, where n_{inputs} and n_{outputs} are the number of input and output connections for the layer whose weights are being initialized (also called *fan-in* and *fan-out*). This initialization strategy is often called *Xavier initialization* (after the author's first name), or sometimes *Glorot initialization*.

Equation 11-1. Xavier initialization (when using the logistic activation function)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

When the number of input connections is roughly equal to the number of output connections, you get simpler equations (e.g., or). We used this simplified strategy in Chapter 10.

$\sigma = 1 / \sqrt{n_{\text{inputs}}}$

Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning. Some recent papers have provided similar strategies for different activation functions, as shown in Table 11-1. The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described shortly) is sometimes called *He initialization* (after the last name of its author).

$r = \sqrt{3} / \sqrt{n_{\text{inputs}}}$

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

By default, the `fully_connected()` function (introduced in Chapter 10) uses Xavier initialization (with a uniform distribution). You can change this to He initialization by using the `variance_scaling_initializer()` function like this:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = fully_connected(X, n_hidden1, weights_initializer=he_init, scope="h1")
```

Note

He initialization considers only the fan-in, not the average between fan-in and fan-out like in Xavier initialization. This is also the default for the `variance_scaling_initializer()` function, but you can change this by setting the argument `mode="FAN_AVG"`.

Nonsaturating Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature

had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*. This function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$, and is typically set to 0.01. This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A recent paper compared several variants of the ReLU activation function and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting $\alpha = 0.2$ (huge leak) seemed to result in better performance than $\alpha = 0.01$ (small leak). They also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training, and it is fixed to an average value during testing. It also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set). Finally, they also evaluated the *parametric leaky ReLU* (PReLU), where α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

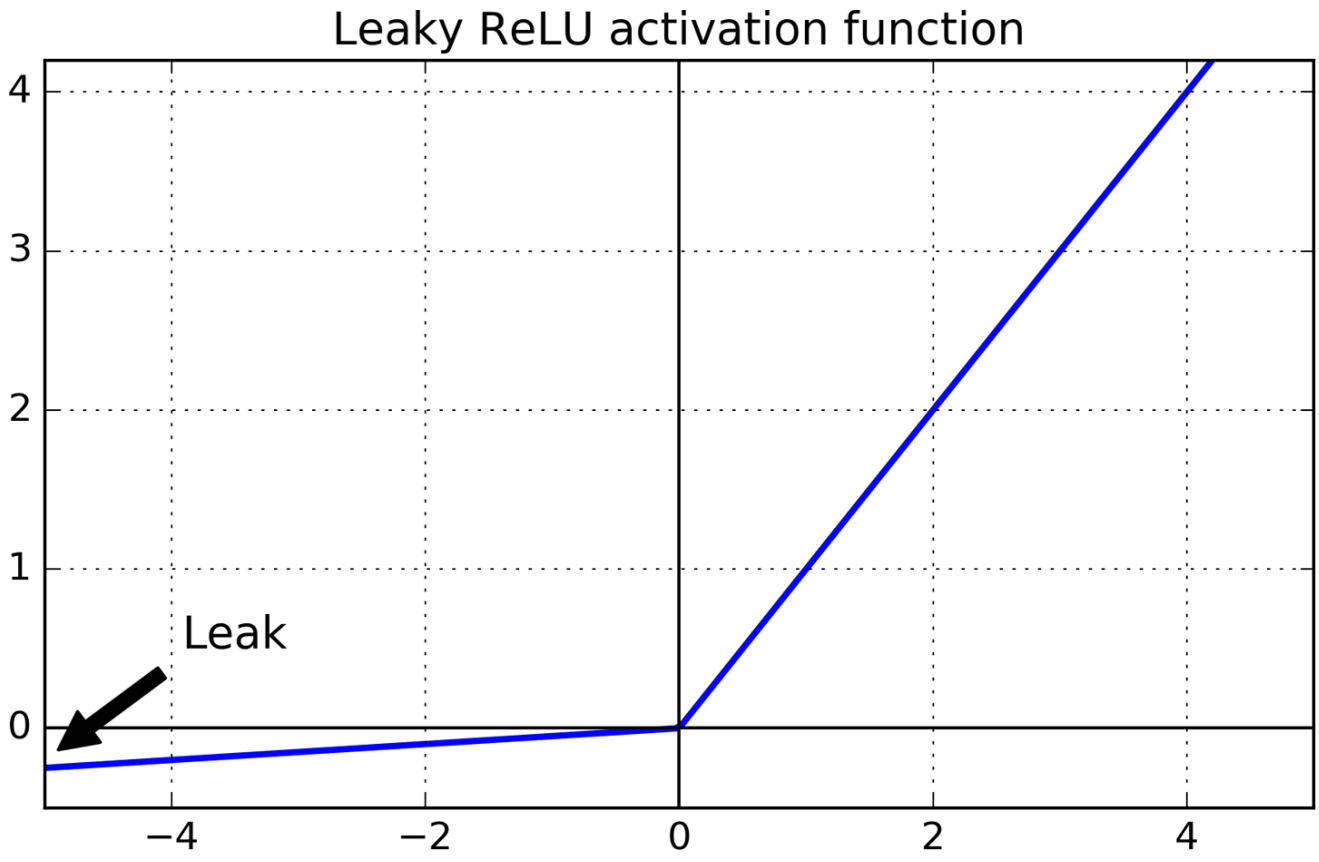

Figure 11-2. Leaky ReLU

Last but not least, a 2015 paper by Djork-Arné Clevert et al. proposed a new activation function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants in their experiments: training time was reduced and the neural network performed better on the test set. It is represented in Figure 11-3, and Equation 11-2 shows its definition.

Equation 11-2. ELU activation function

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

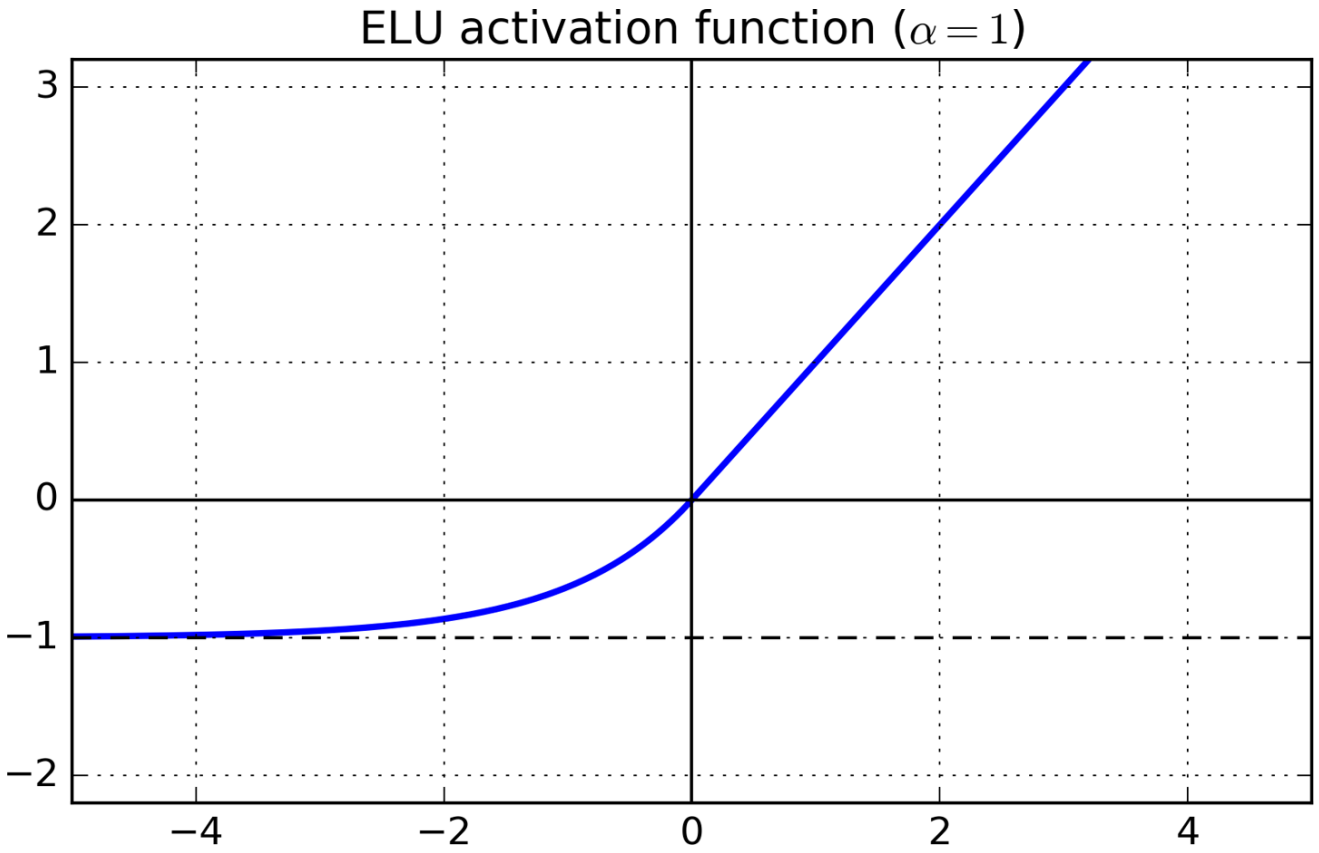

Figure 11-3. ELU activation function

It looks a lot like the ReLU function, with a few major differences:

- First it takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0. This helps alleviate the vanishing gradients problem, as discussed earlier. The hyperparameter α defines the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter if you want.
- Second, it has a nonzero gradient for $z < 0$, which avoids the dying units issue.
- Third, the function is smooth everywhere, including around $z = 0$, which helps speed up Gradient Descent, since it does not bounce as much left and right of $z = 0$.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but during training this is compensated by the faster convergence rate. However, at test time an ELU network will be slower than a ReLU network.

Tip

So which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If you care a lot about runtime performance, then you may prefer leaky ReLUs over ELUs. If you don't want to tweak yet another hyperparameter, you may just use the default α values suggested earlier (0.01 for the leaky ReLU, and 1 for ELU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is overfitting, or PReLU if you have a huge training set.

TensorFlow offers an `elu()` function that you can use to build your neural network. Simply set the `activation_fn` argument when calling the `fully_connected()` function, like this:

```
hidden1 = fully_connected(X, n_hidden1, activation_fn=tf.nn.elu)
```

TensorFlow does not have a predefined function for leaky ReLUs, but it is easy enough to define:

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

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu)
```

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a 2015 paper, Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change (which they call the *Internal Covariate Shift* problem).

The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer.

In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch (hence the name "Batch Normalization"). The whole operation is summarized in Equation 11-3.

Equation 11-3. Batch Normalization algorithm

1. $$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$$

3. $$\mathbf{x}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

4. $$\mathbf{z}^{(i)} = \gamma \mathbf{x}^{(i)} + \beta$$

- μ_B is the empirical mean, evaluated over the whole mini-batch *B*.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- $\mathbf{x}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

At test time, there is no mini-batch to compute the empirical mean and standard deviation, so instead you simply use the whole training set's mean and standard deviation. These are typically efficiently computed during training using a moving average. So, in total, four parameters are learned for each batch-normalized layer: γ (scale), β (offset), μ (mean), and σ (standard deviation).

The authors demonstrated that this technique considerably improved all the deep neural networks they experimented with. The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weight initialization. They were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that "Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. […] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters." Finally, like a gift that keeps on giving, Batch Normalization also acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in the chapter).

Batch Normalization does, however, add some complexity to the model (although it removes the need for normalizing the input data since the first hidden layer will take care of that, provided it is batch-normalized). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. So if you need predictions to be lightning-fast, you may want to check how well plain ELU + He initialization perform before playing with Batch Normalization.

Note

You may find that training is rather slow at first while Gradient Descent is searching for the optimal scales and offsets for each layer, but it accelerates once it has found reasonably good values.

Implementing Batch Normalization with TensorFlow

TensorFlow provides a `batch_normalization()` function that simply centers and normalizes the inputs, but you must compute the mean and standard deviation yourself (based on the mini-batch data during training or on the full dataset during testing, as just discussed) and pass them as parameters to this function, and you must also handle the creation of the scaling and offset parameters (and pass them to this function). It is doable, but not the most convenient approach. Instead, you should use the `batch_norm()` function, which handles all this for you. You can either call it directly or tell the `fully_connected()` function to use it, such as in the following code:

```
import tensorflow as tf
from tensorflow.contrib.layers import batch_norm

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

is_training = tf.placeholder(tf.bool, shape=(), name='is_training')
bn_params = {
    'is_training': is_training,
    'decay': 0.99,
    'updates_collections': None
}

hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          normalizer_fn=batch_norm, normalizer_params=
bn_params)
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2",
                          normalizer_fn=batch_norm, normalizer_params=
bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="outputs",
                         normalizer_fn=batch_norm, normalizer_params=bn_params
)
```

Let's walk through this code. The first lines are fairly self-explanatory, until we define the `is_training` placeholder, which will either be `True` or `False`. This will be used to tell the `batch_norm()` function whether it should use the current mini-batch's mean and standard deviation (during training) or the running averages that it keeps track of (during testing).

Next we define `bn_params`, which is a dictionary that defines the parameters that will be passed to the `batch_norm()` function, including `is_training` of course. The algorithm uses *exponential decay* to compute the running averages, which is why it requires the `decay` parameters. Given a new value *v*, the running average $\hat{v}$ is updated through the equation

$$\hat{v} \leftarrow \hat{v} \times \text{decay} + v \times (1 - \text{decay})$$

. A good decay value is typically close to 1—for example, 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and smaller mini-batches). Finally, `updates_collections` should be set to `None` if you want the `batch_norm()` function to update the running averages right before it performs batch normalization during training (i.e., when `is_training=True`). If you don't set this parameter, by default TensorFlow will just add the operations that update the running averages to a collection of operations that you must run yourself.

Lastly, we create the layers by calling the `fully_connected()` function, just like we did in Chapter 10, but this time we tell it to use the `batch_norm()` function (with the parameters `nb_params`) to normalize the inputs right before calling the activation function.

Note that by default `batch_norm()` only centers, normalizes, and shifts the inputs; it does not scale them (i.e., γ is fixed to 1). This makes sense for layers with no activation function or with the ReLU activation function, since the

"scale":
next layer's weights can take care of scaling, but for any other activation function, you should add `True` to `bn_params`.

You may have noticed that defining the preceding three layers was fairly repetitive since several parameters were identical. To avoid repeating the same parameters over and over again, you can create an *argument scope* using the `arg_scope()` function: the first parameter is a list of functions, and the other parameters will be passed to these functions automatically. The last three lines of the preceding code can be modified like so:

```
[...]

with tf.contrib.framework.arg_scope(
        [fully_connected],
        normalizer_fn=batch_norm,
        normalizer_params=bn_params):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2"
)
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

It may not look much better than before in this small example, but if you have 10 layers and want to set the activation function, the initializers, the normalizers, the regularizers, and so on, it will make your code much more readable.

The rest of the construction phase is the same as in Chapter 10: define the cost function, create an optimizer, tell it to minimize the cost function, define the evaluation operations, create a `Saver`, and so on.

The execution phase is also pretty much the same, with one exception. Whenever you run an operation that depends on the `batch_norm` layer, you need to set the `is_training` placeholder to `True` or `False`:

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op,
                     feed_dict={is_training: True, X: X_batch, y: y_batch
})
        accuracy_score = accuracy.eval(
            feed_dict={is_training: False, X: X_test_scaled, y: y_test}))
        print(accuracy_score)
```

That's all! In this tiny example with just two layers, it's unlikely that Batch Normalization will have a very positive impact, but for deeper networks it can make a tremendous difference.

Gradient Clipping

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks; see Chapter 14). This is called *Gradient Clipping*. In general people now prefer Batch Normalization, but it's still useful to know about Gradient Clipping and how to implement it.

In TensorFlow, the optimizer's `minimize()` function takes care of both computing the gradients and applying them, so you must instead call the optimizer's `compute_gradients()` method first, then create an operation to clip the gradients using the `clip_by_value()` function, and finally create an operation to apply the clipped gradients using the optimizer's `apply_gradients()` method:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

You would then run this `training_op` at every training step, as usual. It will compute the gradients, clip them between –1.0 and 1.0, and apply them. The threshold is a hyperparameter you can tune.

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).

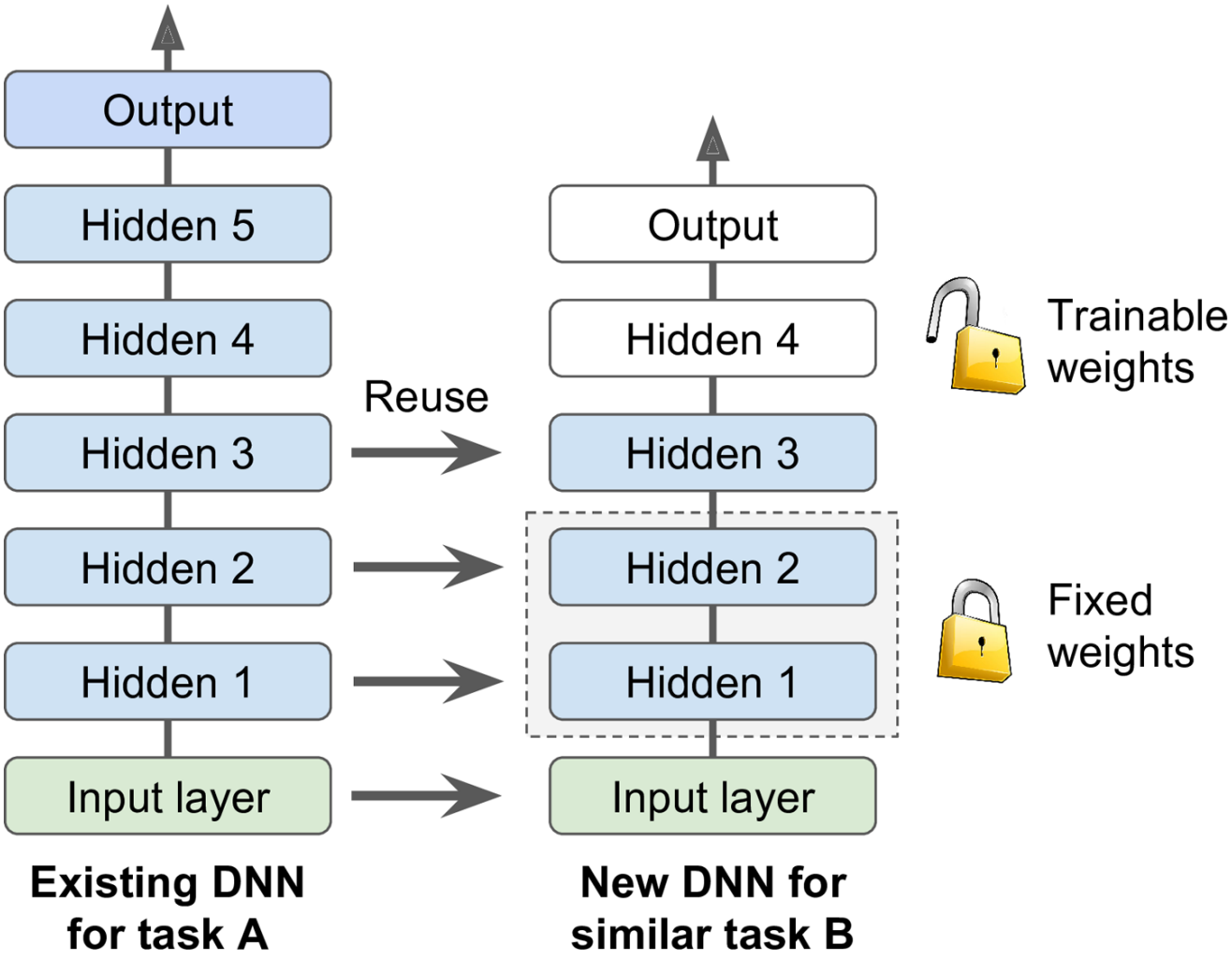

Figure 11-4. Reusing pretrained layers

Note

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 work only 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:

```
      # construct the original
[...] model

with tf.Session() as sess:
    saver.restore(sess, "./my_original_model.ckpt"
)
          # Train it on your new
    [...] task
```

However, in general you will want to reuse only part of the original model (as we will discuss in a moment). A simple solution is to configure the `Saver` to restore only a subset of the variables from the original model. For example, the following code restores only hidden layers 1, 2, and 3:

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

init = tf.global_variables_initializer()

reuse_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[123]")
reuse_vars_dict = dict([(var.name, var.name) for var in reuse_vars])
                                            # saver to restore the original
original_saver = tf.Saver(reuse_vars_dict) model

                        # saver to save the new
new_saver = tf.Saver() model

with tf.Session() as sess:
    sess.run(init)
                                                          # restore layers 1 to
    original_saver.restore("./my_original_model.ckpt") 3
          # train the new
    [...] model
                                           # save the whole
    new_saver.save("./my_new_model.ckpt") model
```

First we build the new model, making sure to copy the original model's hidden layers 1 to 3. We also create a node to initialize all variables. Then we get the list of all variables that were just created with `"trainable=True"` (which is the default), and we keep only the ones whose scope matches the regular expression `"hidden[123]"` (i.e., we get all trainable variables in hidden layers 1 to 3). Next we create a dictionary mapping 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, and we create another `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.

Tip

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 weights 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:

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

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
      # # Build the rest of the
[...] model

# Get a handle on the variables created by
fully_connected()
                                                          # root
with tf.variable_scope("", default_name="", reuse=True):  scope
    hidden1_weights = tf.get_variable("hidden1/weights")
    hidden1_biases = tf.get_variable("hidden1/biases")

# Create nodes to assign arbitrary values to the weights and
biases
original_weights = tf.placeholder(tf.float32, shape=(n_inputs, n_hidden1))
original_biases = tf.placeholder(tf.float32, shape=(n_hidden1))
assign_hidden1_weights = tf.assign(hidden1_weights, original_weights)
assign_hidden1_biases = tf.assign(hidden1_biases, original_biases)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.run(assign_hidden1_weights, feed_dict={original_weights: original_w
})
    sess.run(assign_hidden1_biases, feed_dict={original_biases: original_b})
          # Train the model on your new
    [...] task
```

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, the simplest 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 provide 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*).

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):

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

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_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(len(hidden2_outputs))
    hidden2_batches = np.array_split(hidden2_outputs[shuffled_idx], n_batches
)
    y_batches = np.array_split(y_train[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})
```

The last line runs the training operation defined earlier (which freezes 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.

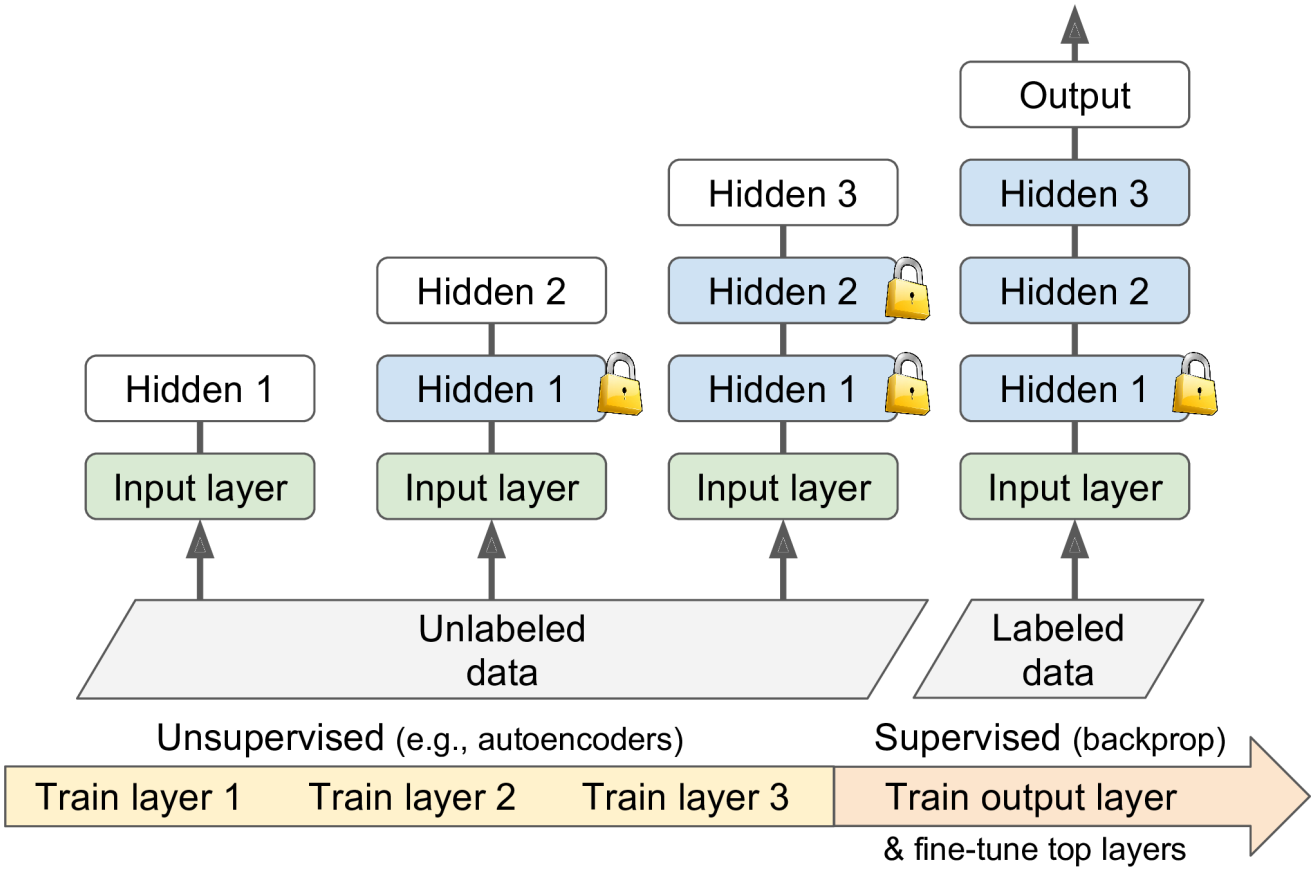

Figure 11-5. Unsupervised pretraining

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.

Spoiler alert: the conclusion of this section is that you should almost always use Adam optimization,[10] so if you don't care about how it works, simply replace your `GradientDescentOptimizer` with an `AdamOptimizer` and skip to the next section! With just this small change, training will typically be several times faster. However, Adam optimization does have three hyperparameters that you can tune (plus the learning rate); the default values usually work fine, but if you ever need to tweak them it may be helpful to know what they do. Adam optimization combines several ideas from other optimization algorithms, so it is useful to look at these algorithms first.

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.[11] 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 adds the local gradient to the *momentum vector* **m** (multiplied by the learning rate η), and it updates the weights by simply subtracting 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 hyperparameter β, simply called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-4. Momentum algorithm

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla_\theta J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \mathbf{m}$$

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$. For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular, we saw in Chapter 4 that when the inputs have very different scales the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, Momentum optimization will roll down the

bottom of the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Note

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in TensorFlow is a no-brainer: just replace the `GradientDescentOptimizer` with the `MomentumOptimizer`, then lie back and profit!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

The one drawback of Momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than Gradient Descent.

Nesterov Accelerated Gradient

One small variant to Momentum optimization, proposed by Yurii Nesterov in 1983,[12] is almost always faster than vanilla Momentum optimization. The idea of *Nesterov Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum (see Equation 11-5). The only difference from vanilla Momentum optimization is that the gradient is measured at $\theta + \beta\mathbf{m}$ rather than at θ.

Equation 11-5. Nesterov Accelerated Gradient algorithm

$$1.\quad \mathbf{m} \leftarrow \beta\mathbf{m} + \eta \nabla_\theta J(\theta + \beta\mathbf{m})$$

$$2.\quad \theta \leftarrow \theta - \mathbf{m}$$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradient at the original position, as you can see in Figure 11-6 (where ∇_1 represents the gradient of the cost function measured at the starting point θ, and ∇_2 represents the gradient at the point located at $\theta + \beta\mathbf{m}$). As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converges faster.

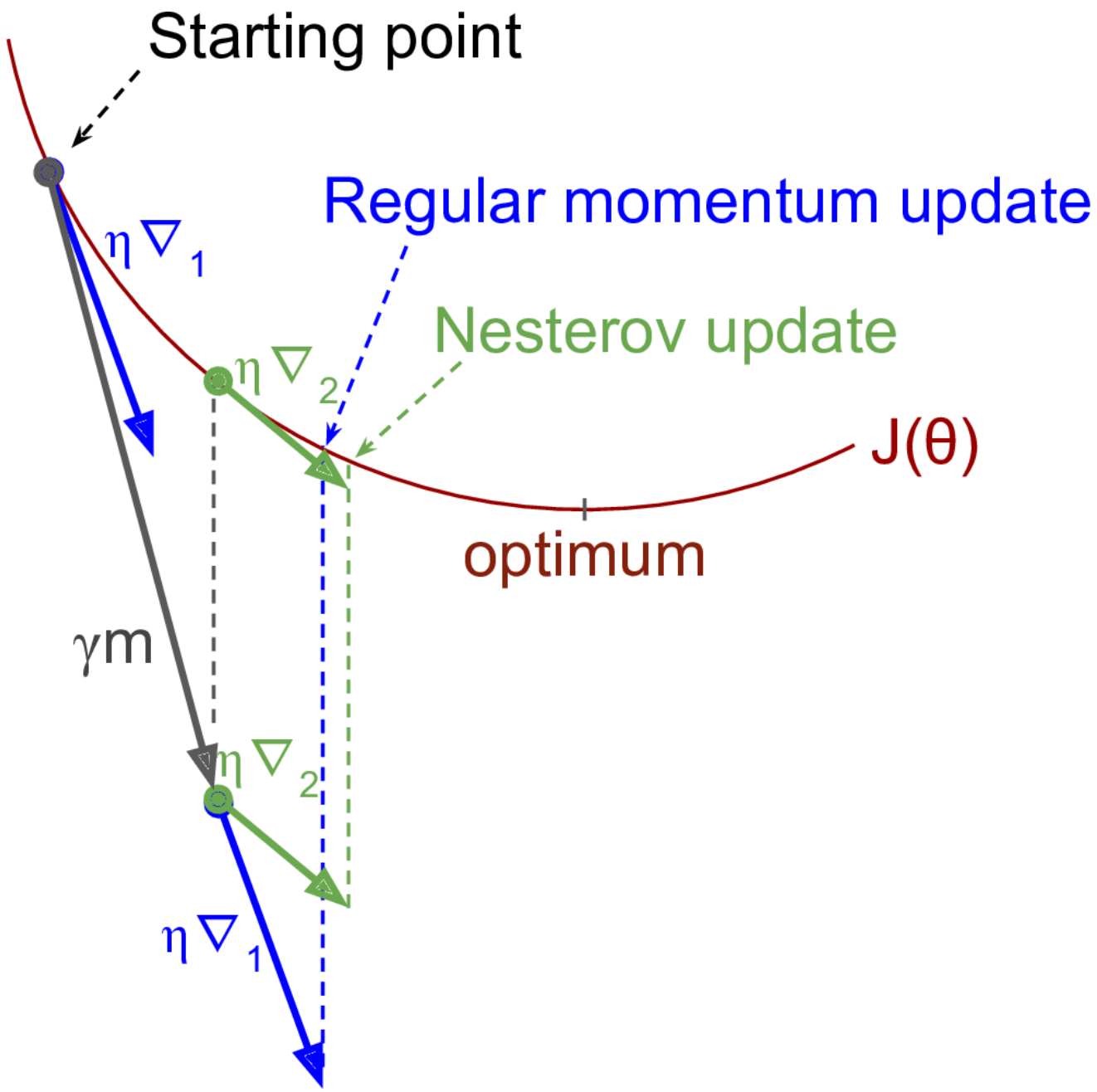

Figure 11-6. Regular versus Nesterov Momentum optimization

NAG will almost always speed up training compared to regular Momentum optimization. To use it, simply set `use_nesterov=True` when creating the `MomentumOptimizer`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=
True)
```

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The *AdaGrad* algorithm[13] achieves this by scaling down the gradient vector along the steepest dimensions (see Equation 11-6):

Equation 11-6. AdaGrad algorithm

$$1. \quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

The first step accumulates the square of the gradients into the vector **s** (the ⊗ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial / \partial \, \theta_i \, J(\theta))^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the i^{th} dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$ (the ⊘ symbol represents the element-wise division, and □ is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing for all parameters θ_i (simultaneously).

$$\theta_i \leftarrow \theta_i - \eta \, \partial / \partial \theta_i \, J(\theta) / \sqrt{s_i + \epsilon}$$

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

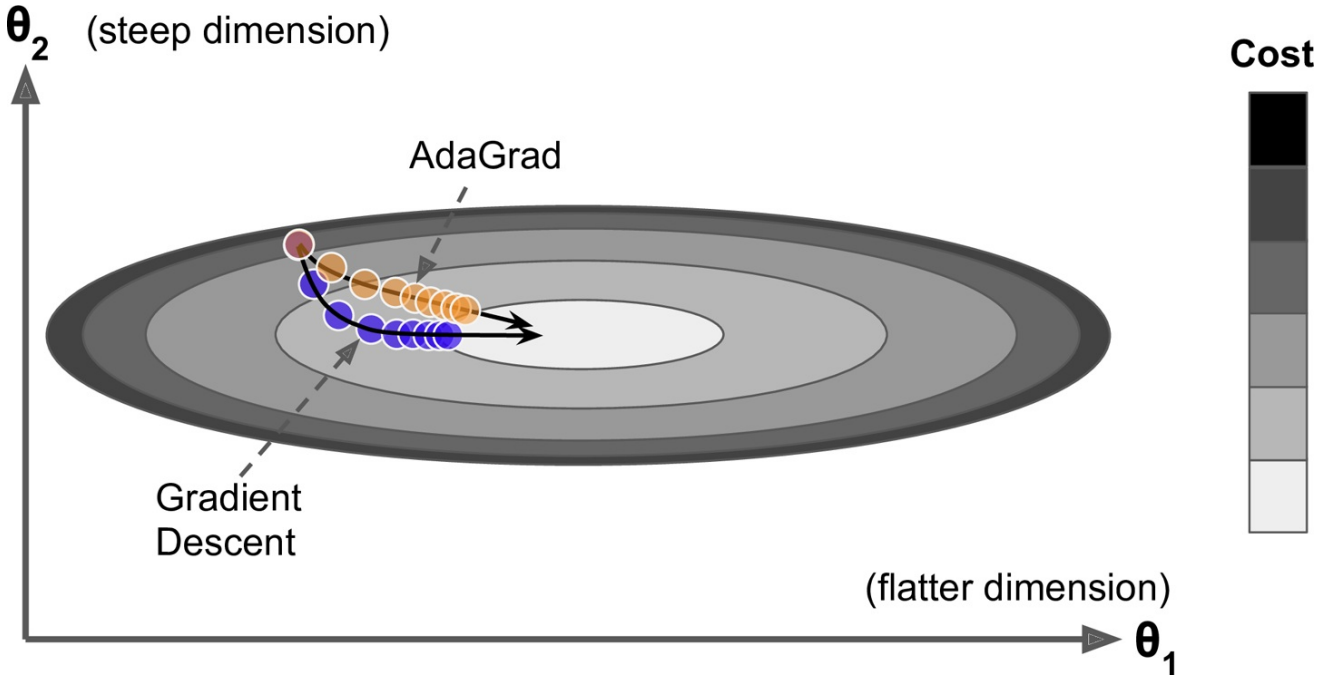

Figure 11-7. AdaGrad versus Gradient Descent

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though TensorFlow has an `AdagradOptimizer`, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though).

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[14] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

$$1. \quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, TensorFlow has an `RMSPropOptimizer` class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=
1e-10)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization and Nesterov Accelerated Gradients. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam Optimization

Adam,[15] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[16]

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$
5. $\theta \leftarrow \theta - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- *T* represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term □ is usually initialized to a tiny number such as 10^{-8}. These are the default values for TensorFlow's `AdamOptimizer` class, so you can simply use:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

In fact, since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

Note

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never settling down (unless you use an adaptive learning rate optimization algorithm such as AdaGrad, RMSProp, or Adam, but even then it may take time to settle). If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

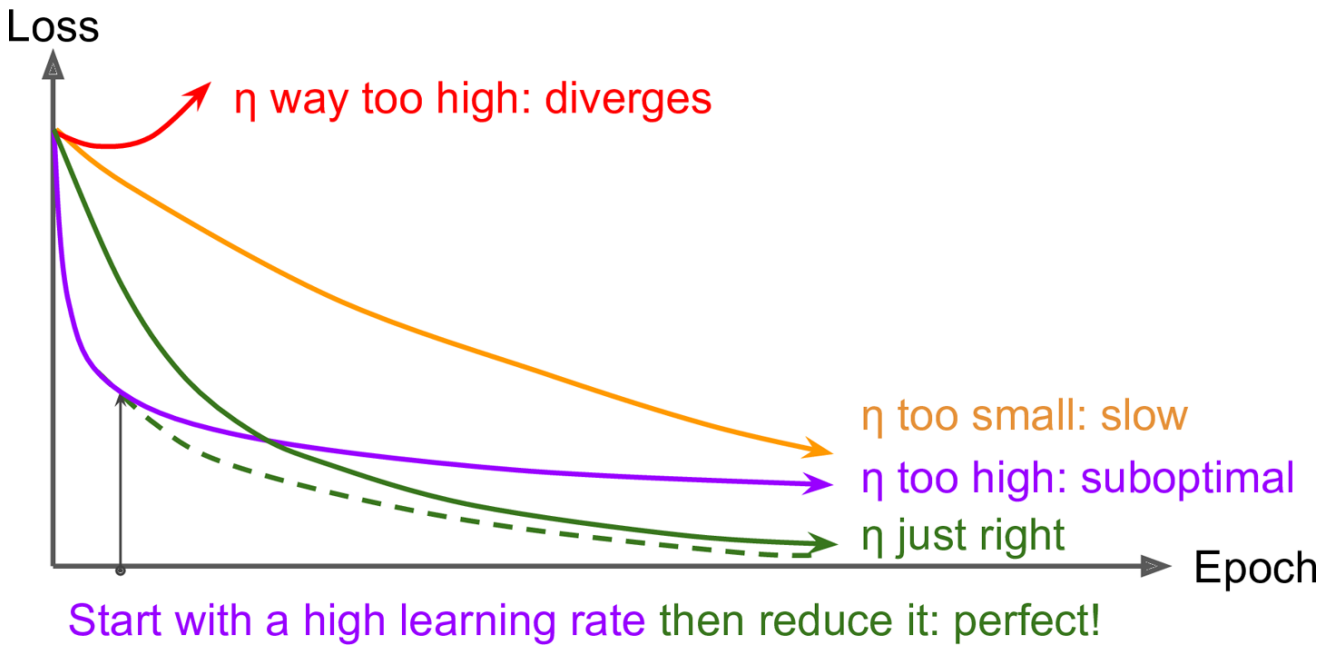

Figure 11-8. Learning curves for various learning rates η

You may be able to find a fairly good learning rate by training your network several times during just a few epochs using various learning rates and comparing the learning curves. The ideal learning rate will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4), the most common of which are:

Predetermined piecewise constant learning rate

> For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling

> Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling

> Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\ 10^{-t/r}$. This works great, but it requires

tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

Power scheduling

Set the learning rate to $\eta(t) = \eta_0 (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

A 2013 paper[19] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well, but they favored exponential scheduling because it is simpler to implement, is easy to tune, and converged slightly faster to the optimal solution.

Implementing a learning schedule with TensorFlow is fairly straightforward:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False)
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                           decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

After setting the hyperparameter values, we create a nontrainable variable `global_step` (initialized to 0) to keep track of the current training iteration number. Then we define an exponentially decaying learning rate (with η_0 = 0.1 and r = 10,000) using TensorFlow's `exponential_decay()` function. Next, we create an optimizer (in this example, a `MomentumOptimizer`) using this decaying learning rate. Finally, we create the training operation by calling the optimizer's `minimize()` method; since we pass it the `global_step` variable, it will kindly take care of incrementing it. That's it!

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms, using exponential decay or performance scheduling can considerably speed up convergence.

Avoiding Overfitting Through Regularization

> *With four parameters I can fit an elephant and with five I can make him wiggle his trunk.*
>
> *John von Neumann, cited by Enrico Fermi in Nature 427*

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.

With millions of parameters you can fit the whole zoo. In this section we will present some of the most popular regularization techniques for neural networks, and how to implement them with TensorFlow: early stopping, ℓ_1 and ℓ_2 regularization, dropout, max-norm regularization, and data augmentation.

Early Stopping

To avoid overfitting the training set, a great solution is early stopping (introduced in Chapter 4): just interrupt training when its performance on the validation set starts dropping.

One way to implement this with TensorFlow is to evaluate the model on a validation set at regular intervals (e.g., every 50 steps), and save a "winner" snapshot if it outperforms previous "winner" snapshots. Count the number of steps since the last "winner" snapshot was saved, and interrupt training when this number reaches some limit (e.g., 2,000 steps). Then restore the last "winner" snapshot.

Although early stopping works very well in practice, you can usually get much higher performance out of your network by combining it with other regularization techniques.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights (but typically not its biases).

One way to do this using TensorFlow is to simply add the appropriate regularization terms to your cost function. For example, assuming you have just one hidden layer with weights `weights1` and one output layer with weights `weights2`, then you can apply ℓ_1 regularization like this:

```
      # construct the neural
[...] network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

However, if there are many layers, this approach is not very convenient. Fortunately, TensorFlow provides a better option. Many functions that create variables (such as `get_variable()` or `fully_connected()`) accept a `*_regularizer` argument for each created variable (e.g., `weights_regularizer`). You can pass any function that takes weights as an argument and returns the corresponding regularization loss. The `l1_regularizer()`, `l2_regularizer()`, and `l1_l2_regularizer()` functions return such functions. The following code puts all this together:

```
with arg_scope(
        [fully_connected],
        weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out"
)
```

This code creates a neural network with two hidden layers and one output layer, and it also creates nodes in the graph to compute the ℓ_1 regularization loss corresponding to each layer's weights. TensorFlow automatically adds these nodes to a special collection containing all the regularization losses. You just need to add these regularization losses to your overall loss, like this:

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

Warning

Don't forget to add the regularization losses to your overall loss, or else they will simply be ignored.

Dropout

The most popular regularization technique for deep neural networks is arguably *dropout*. It was proposed[20] by G. E. Hinton in 2012 and further detailed in a paper[21] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability *p* of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter *p* is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

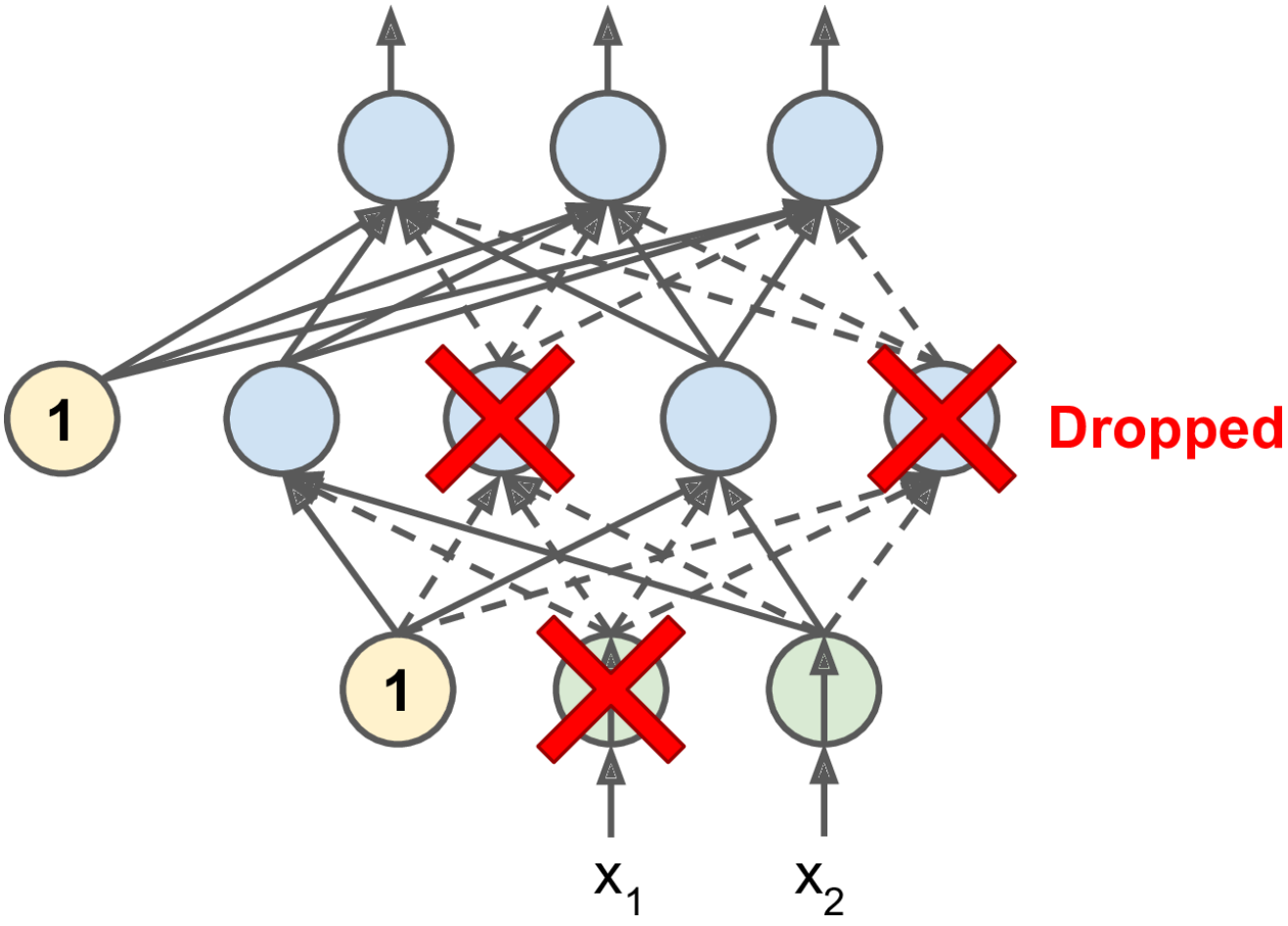

Figure 11-9. Dropout regularization

It is quite surprising at first that this rather brutal technique works at all. Would a company perform better if its

employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there is a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent since they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

There is one small but important technical detail. Suppose $p = 50$, in which case during testing a neuron will be connected to twice as many input neurons as it was (on average) during training. To compensate for this fact, we need to multiply each neuron's input connection weights by 0.5 after training. If we don't, each neuron will get a total input signal roughly twice as large as what the network was trained on, and it is unlikely to perform well. More generally, we need to multiply each input connection weight by the *keep probability* $(1 – p)$ after training. Alternatively, we can divide each neuron's output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement dropout using TensorFlow, you can simply apply the `dropout()` function to the input layer and to the output of every hidden layer. During training, this function randomly drops some items (setting them to 0) and divides the remaining items by the keep probability. After training, this function does nothing at all. The following code applies dropout regularization to our three-layer neural network:

```
from tensorflow.contrib.layers import dropout

[...]
is_training = tf.placeholder(tf.bool, shape=(), name='is_training')

keep_prob = 0.5
X_drop = dropout(X, keep_prob, is_training=is_training)

hidden1 = fully_connected(X_drop, n_hidden1, scope="hidden1")
hidden1_drop = dropout(hidden1, keep_prob, is_training=is_training)

hidden2 = fully_connected(hidden1_drop, n_hidden2, scope="hidden2")
hidden2_drop = dropout(hidden2, keep_prob, is_training=is_training)

logits = fully_connected(hidden2_drop, n_outputs, activation_fn=None,
                         scope="outputs")
```

Warning

You want to use the `dropout()` function in `tensorflow.contrib.layers`, not the one in `tensorflow.nn`. The first one turns off (no-op) when not training, which is what you want, while the second one does not.

Of course, just like you did earlier for Batch Normalization, you need to set `is_training` to `True` when training, and to `False` when testing.

If you observe that the model is overfitting, you can increase the dropout rate (i.e., reduce the `keep_prob` hyperparameter). Conversely, you should try decreasing the dropout rate (i.e., increasing `keep_prob`) if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. So, it is generally well worth the extra time and effort.

Note

Dropconnect is a variant of dropout where individual connections are dropped randomly rather than whole neurons. In general dropout performs better.

Max-Norm Regularization

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

We typically implement this constraint by computing $\| \mathbf{w} \|_2$ after each training step and clipping $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\| \mathbf{w} \|_2}$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems (if you are not using Batch Normalization).

TensorFlow does not provide an off-the-shelf max-norm regularizer, but it is not too hard to implement. The following code creates a node `clip_weights` that will clip the `weights` variable along the second axis so that each row vector has a maximum norm of 1.0:

```
threshold = 1.0
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

You would then apply this operation after each training step, like so:

```
with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch
})
            clip_weights.eval()
```

You may wonder how to get access to the `weights` variable of each layer. For this you can simply use a variable

scope like this:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")

with tf.variable_scope("hidden1", reuse=True):
    weights1 = tf.get_variable("weights")
```

Alternatively, you can use the root variable scope:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
[...]

with tf.variable_scope("", default_name="", reuse=True):
# root
scope
    weights1 = tf.get_variable("hidden1/weights")
    weights2 = tf.get_variable("hidden2/weights")
```

If you don't know what the name of a variable is, you can either use TensorBoard to find out or simply use the `global_variables()` function and print out all the variable names:

```
for variable in tf.global_variables():
    print(variable.name)
```

Although the preceding solution should work fine, it is a bit messy. A cleaner solution is to create a `max_norm_regularizer()` function and use it just like the earlier `l1_regularizer()` function:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes
)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
                    # there is no regularization loss
        return None  term
    return max_norm
```

This function returns a parametrized `max_norm()` function that you can use like any other regularizer:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          weights_regularizer=
max_norm_reg)
```

Note that max-norm regularization does not require adding a regularization loss term to your overall loss function, so the `max_norm()` function returns `None`. But you still need to be able to run the `clip_weights` operation after each

training step, so you need to be able to get a handle on it. This is why the `max_norm()` function adds the `clip_weights` node to a collection of max-norm clipping operations. You need to fetch these clipping operations and run them after each training step:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch
})
            sess.run(clip_all_weights)
```

Much cleaner code, isn't it?

Data Augmentation

One last regularization technique, data augmentation, consists of generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting, making this a regularization technique. The trick is to generate realistic training instances; ideally, a human should not be able to tell which instances were generated and which ones were not. Moreover, simply adding white noise will not help; the modifications you apply should be learnable (white noise is not).

For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 11-10). This forces the model to be more tolerant to the position, orientation, and size of the mushrooms in the picture. If you want the model to be more tolerant to lighting conditions, you can similarly generate many images with various contrasts. Assuming the mushrooms are symmetrical, you can also flip the pictures horizontally. By combining these transformations you can greatly increase the size of your training set.

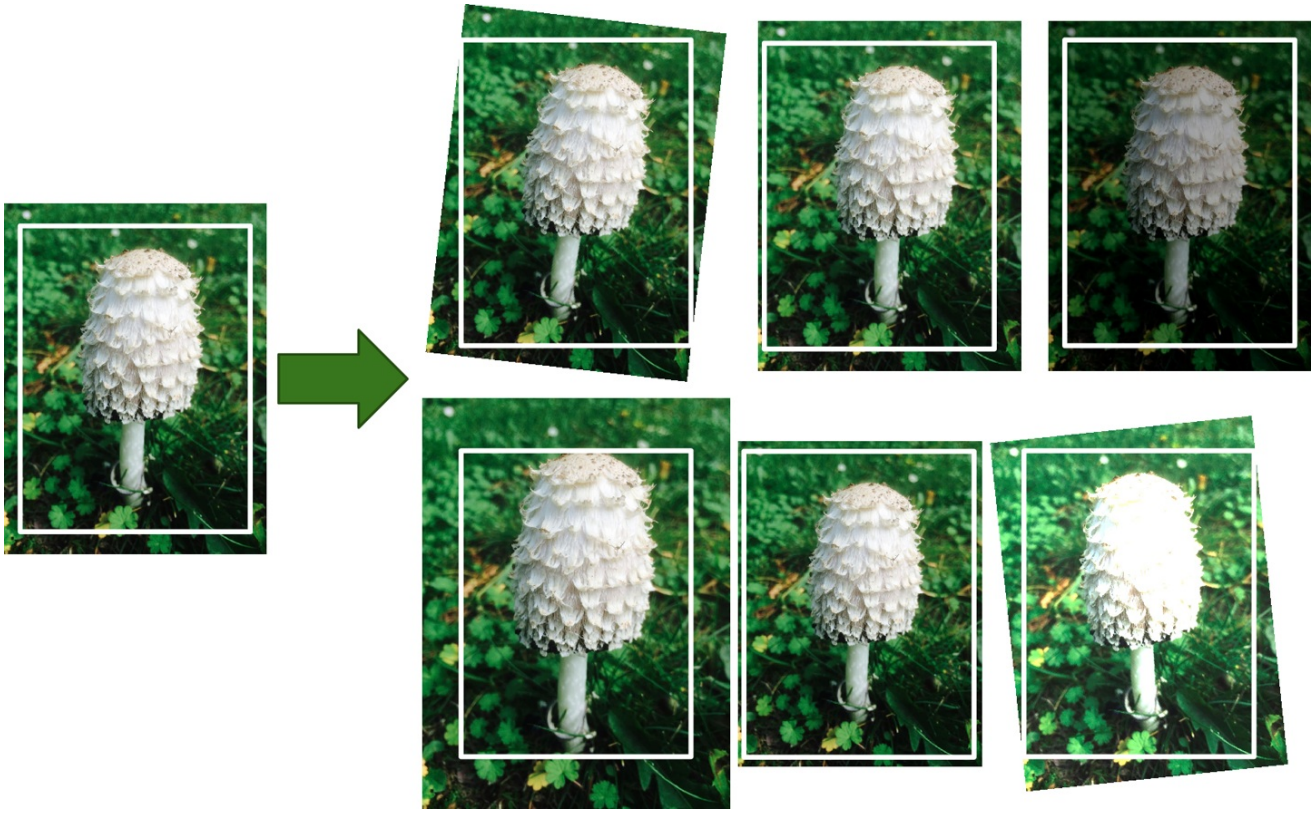

Figure 11-10. Generating new training instances from existing ones

It is often preferable to generate training instances on the fly during training rather than wasting storage space and network bandwidth. TensorFlow offers several image manipulation operations such as transposing (shifting), rotating, resizing, flipping, and cropping, as well as adjusting the brightness, contrast, saturation, and hue (see the API documentation for more details). This makes it easy to implement data augmentation for image datasets.

Note

Another powerful technique to train very deep neural networks is to add *skip connections* (a skip connection is when you add the input of a layer to the output of a higher layer). We will explore this idea in Chapter 13 when we talk about deep residual networks.

Practical Guidelines

In this chapter, we have covered a wide range of techniques and you may be wondering which ones you should use. The configuration in Table 11-2 will work fine in most cases.

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Of course, you should try to reuse parts of a pretrained neural network if you can find one that solves a similar problem.

This default configuration may need to be tweaked:

- If you can't find a good learning rate (convergence was too slow, so you increased the training rate, and now convergence is fast but the network's accuracy is suboptimal), then you can try adding a learning schedule such as exponential decay.
- If your training set is a bit too small, you can implement data augmentation.
- If you need a sparse model, you can add some ℓ_1 regularization to the mix (and optionally zero out the tiny weights after training). If you need an even sparser model, you can try using FTRL instead of Adam optimization, along with ℓ_1 regularization.
- If you need a lightning-fast model at runtime, you may want to drop Batch Normalization, and possibly replace the ELU activation function with the leaky ReLU. Having a sparse model will also help.

With these guidelines, you are now ready to train very deep nets—well, if you are very patient, that is! If you use a single machine, you may have to wait for days or even months for training to complete. In the next chapter we will discuss how to use distributed TensorFlow to train and run models across many servers and GPUs.

Exercises

1. Is it okay to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it okay to initialize the bias terms to 0?
3. Name three advantages of the ELU activation function over ReLU.
4. In which cases would you want to use each of the following activation functions: ELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using a `MomentumOptimizer`?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)?
8. Deep Learning.
    1. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
    2. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
    3. Tune the hyperparameters using cross-validation and see what precision you can achieve.
    4. Now try adding Batch Normalization and compare the learning curves: is it converging faster than

before? Does it produce a better model?

5. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?

9. Transfer learning.

    1. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a fresh new one.

    2. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?

    3. Try caching the frozen layers, and train the model again: how much faster is it now?

    4. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?

    5. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?

10. Pretraining on an auxiliary task.

    1. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little training data. Start by building two DNNs (let's call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add a single output layer on top of both DNNs. You should use TensorFlow's `concat()` function with `axis=1` to concatenate the outputs of both DNNs along the horizontal axis, then feed the result to the output layer. This output layer should contain a single neuron using the logistic activation function.

    2. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.

    3. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.

    4. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in Appendix A.

"Understanding the Difficulty of Training Deep Feedforward Neural Networks," X. Glorot, Y Bengio (2010).

Here's an analogy: if you set a microphone amplifier's knob too close to zero, people won't hear your voice, but if you set it too close to the max, your voice will be saturated and people won't understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

This simplified strategy was actually already proposed much earlier—for example, in the 1998 book *Neural Networks: Tricks of the Trade* by Genevieve Orr and Klaus-Robert Müller (Springer).

Such as “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” K. He et al. (2015).

“Empirical Evaluation of Rectified Activations in Convolution Network,” B. Xu et al. (2015).

“Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” D. Clevert, T. Unterthiner, S. Hochreiter (2015).

“Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” S. Ioffe and C. Szegedy (2015).

“On the difficulty of training recurrent neural networks,” R. Pascanu et al. (2013).

Another option is to come up with a supervised task for which you can easily gather a lot of labeled training data, then use transfer learning, as explained earlier. For example, if you want to train a model to identify your friends in pictures, you could download millions of faces on the internet and train a classifier to detect whether two faces are identical or not, then use this classifier to compare a new picture with each picture of your friends.

[10] At least for now: research is moving fast, especially in the field of optimization. Be sure to take a look at the latest and greatest optimizers every time a new version of TensorFlow is released.

[11] “Some methods of speeding up the convergence of iteration methods,” B. Polyak (1964).

[12] “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$,” Yurii Nesterov (1983).

[13] “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” J. Duchi et al. (2011).

[14] This algorithm was created by Tijmen Tieleman and Geoffrey Hinton in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *http://goo.gl/RsQeis*; video: *https://goo.gl/XUbIyJ*). Amusingly, since the authors have not written a paper to describe it, researchers often cite “slide 29 in lecture 6” in their papers.

[15] “Adam: A Method for Stochastic Optimization,” D. Kingma, J. Ba (2015).

[16] These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

[17] “Primal-Dual Subgradient Methods for Convex Problems,” Yurii Nesterov (2005).

[18] “Ad Click Prediction: a View from the Trenches,” H. McMahan et al. (2013).

[19] “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition,” A. Senior et al. (2013).

[20] “Improving neural networks by preventing co-adaptation of feature detectors,” G. Hinton et al. (2012).

[21] “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” N. Srivastava et al. (2014).