SAIF

Shanghai Advanced Institute of Finance
上海高级金融学院

量化俱乐部-深度学习-TrainingDNN

2019-11-03

Menu

1. Vanishing/Exploding Gradients Problems
   a. Xavier and He Initialization
   b. Nonsaturating Activation Functions
   c. Batch Normalization
   d. Gradient Clipping
2. Reusing Pretrained Layers
3. Faster Optimizers
   a. Momentum optimization
   b. Nesterov Accelerated Gradient
   c. AdaGrad
   d. RMSPprop
   e. Adam Optimization
   f. Learning Rate Scheduling
3. Avoiding Overfitting Through Regularization
4. Practical Guidelines

自我介绍

张骁喆，高金FMBA 2017。10年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017高金开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化策略研究。
量化投资策略咨询/人工智能咨询培训/金融科技项目管理产品管理。

Intro

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:

1. 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.
2. Second, with such a large network, training would be extremely slow.
3. 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.

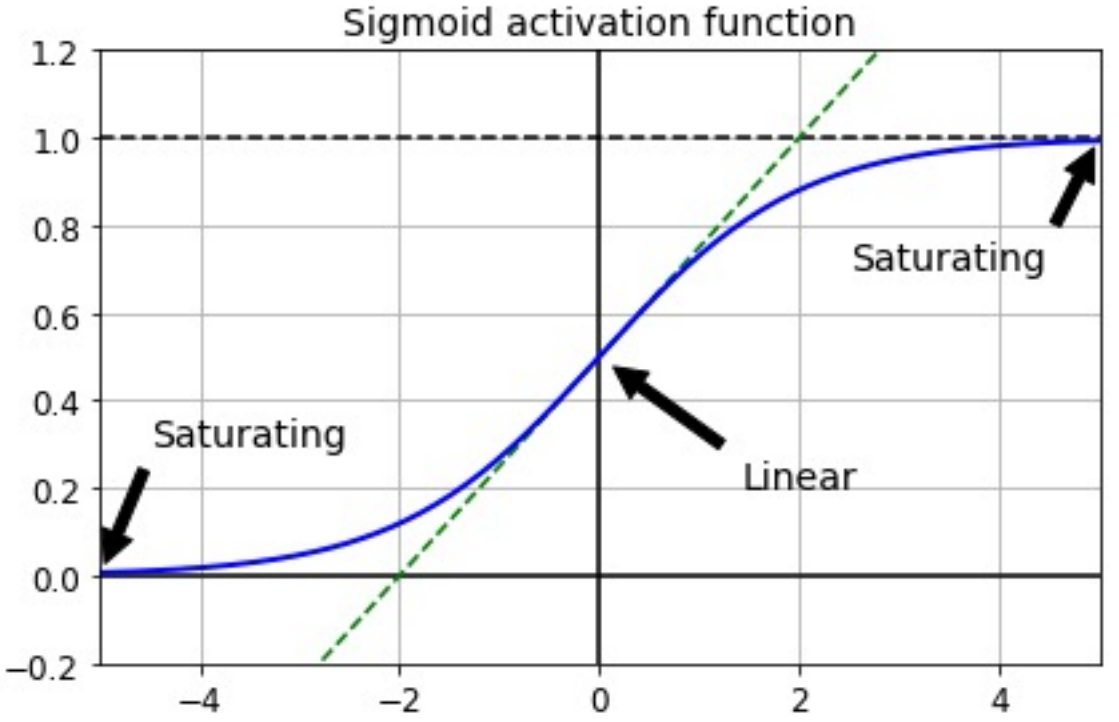
Vanishing/Exploding Gradients Problems

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.

Vanishing/Exploding Gradients Problem

Looking at the logistic activation function, 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.



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

they proposed a good compromise that has proven to work very well in practice: the connection weights must be initialized randomly as described below, 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.*

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

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

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

Xavier and He Initialization

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. The initialization strategy for the ReLUactivation function (and its variants, including the ELU activation described shortly) is sometimes called *He initialization* (after the last name of its author).

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)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Xavier and He Initialization

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

In [6]:
```
he_init = tf.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                          kernel_initializer=he_init, name="hidden1")
```

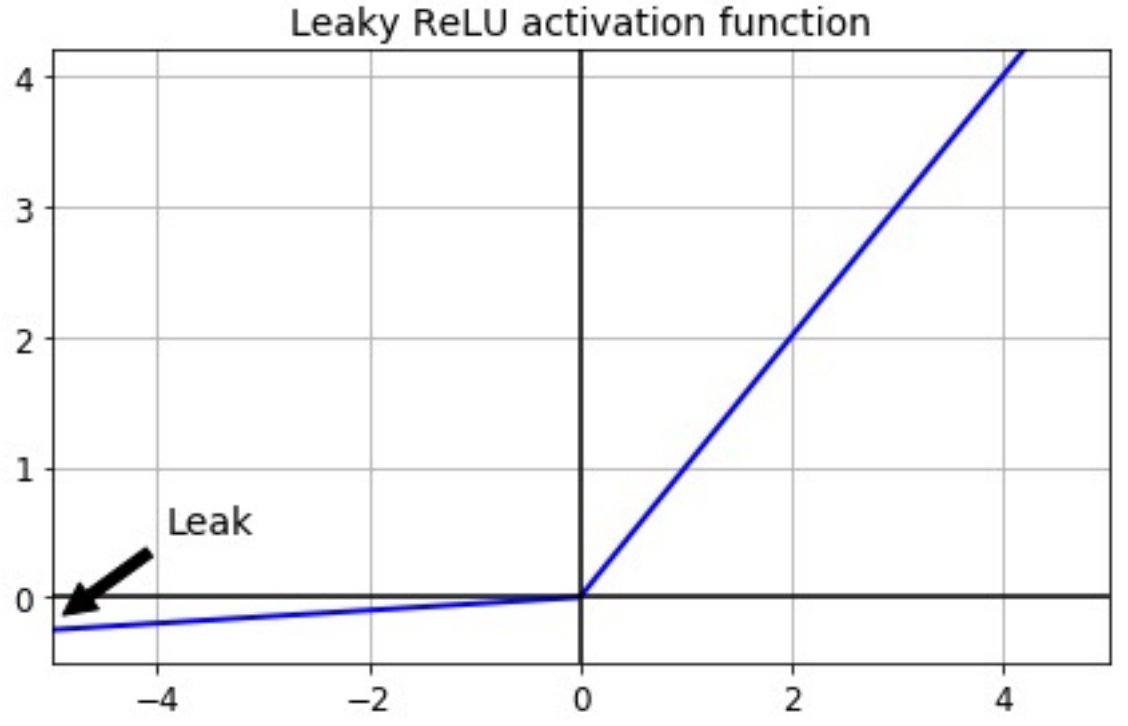
```
WARNING:tensorflow:From <ipython-input-6-da109dac52d3>:3: dense (from tensorflow.python.layers.core) is deprecated an
d will be removed in a future version.
Instructions for updating:
Use keras.layers.dense instead.
```

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. 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). 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

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.

Nonsaturating Activation Functions

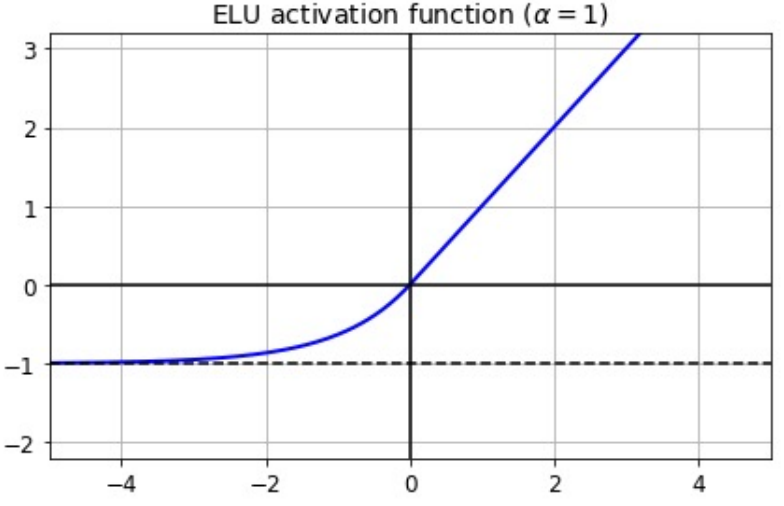


Nonsaturating Activation Functions

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 below:

Equation 11-2. ELU activation function

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

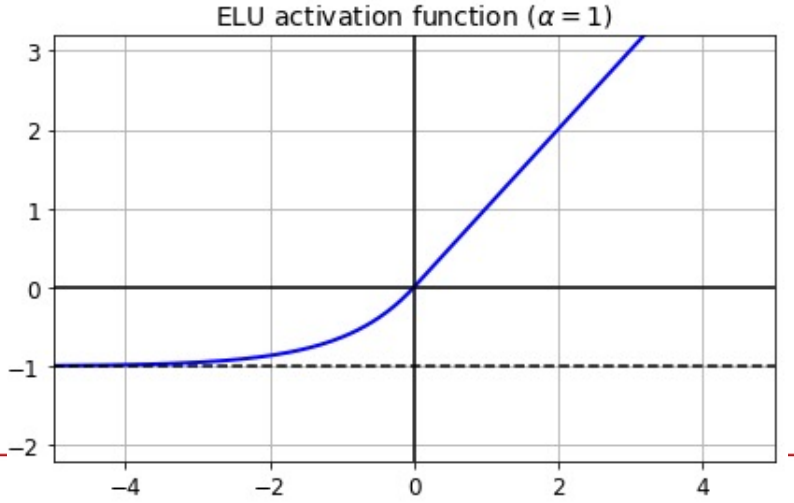


Nonsaturating Activation Functions

***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.



Nonsaturating Activation Functions

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:

```
In [24]: hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

Batch Normalization

In a 2015 paper, Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems

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.

Batch Normalization

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

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

Batch Normalization

- 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.
- $\hat{x}^{(i)}$ 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).

Batch Normalization

The authors demonstrated that this technique considerably improved all the deep neural networks they experimented with:

1. 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.
2. They were able to use much larger learning rates, significantly speeding up the learning process
3. 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)

However, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer.

Batch Normalization

Implementing Batch Normalization with TensorFlow

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)
```

Batch Normalization

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

Batch Normalization

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)
```

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 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.

Gradient Clipping

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:

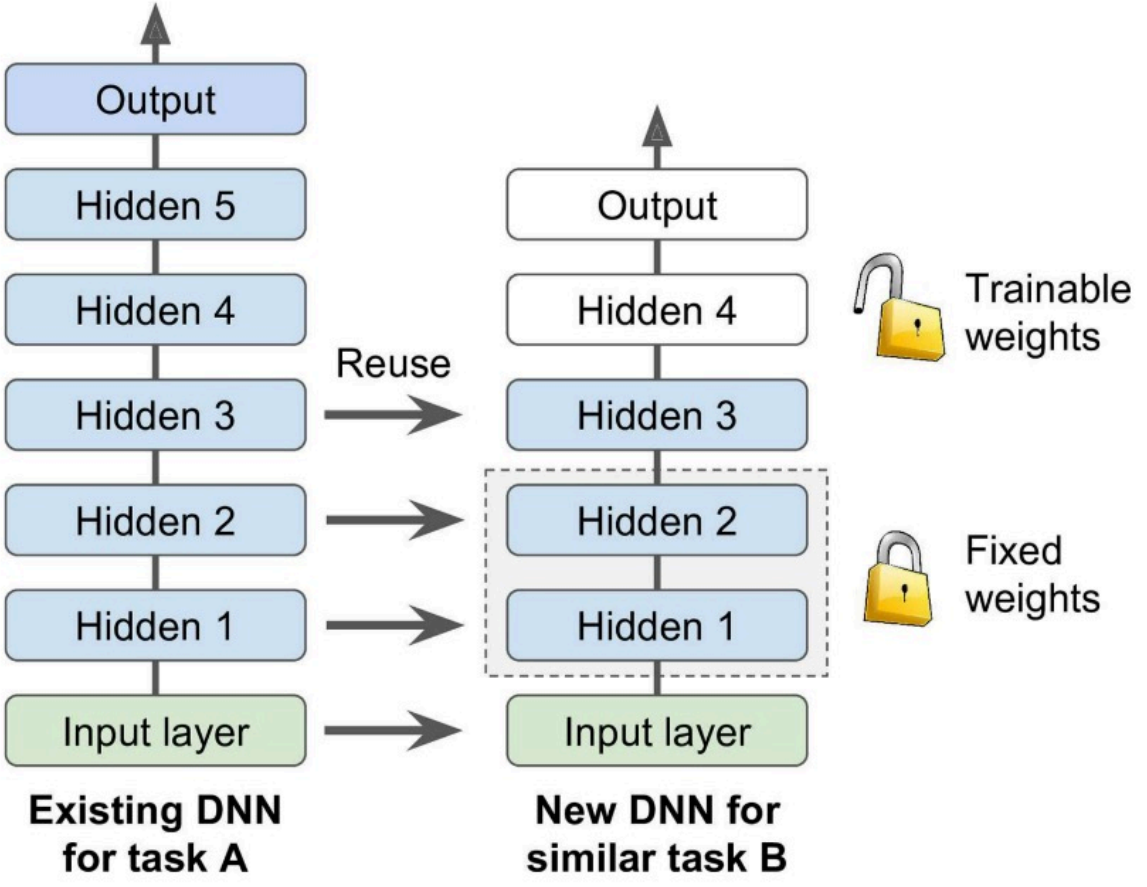
You would then run this training_op at every training step, as usual. It will compute the gradients, clip them between –1.0 and 1.0, and apply them. The threshold is a hyperparameter you can tune.

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.

Reusing Pretrained Layers

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.



Reusing a TensorFlow Model

If the original model was trained using TensorFlow, you can simply restore it and train it on the new task:

In [66]:
```
with tf.Session() as sess:
    saver.restore(sess, "./tf_logs/GradientClipping/my_model_final.ckpt")
    # continue training the model...
```

```
INFO:tensorflow:Restoring parameters from ./tf_logs/GradientClipping/my_model_final.ckpt
```

Reusing a TensorFlow Model

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:

In [75]:
```
reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]") # regular expression
restore_saver = tf.train.Saver(reuse_vars) # to restore layers 1-3

init = tf.global_variables_initializer()
saver = tf.train.Saver()

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

    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
        print(epoch, "Validation accuracy:", accuracy_val)

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

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:

```
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 variables of layer hidden1
with tf.variable_scope("", default_name="", reuse=True):  # root scope
    hidden1_weights = tf.get_variable("hidden1/kernel")
    hidden1_biases = tf.get_variable("hidden1/bias")

# Create dedicated placeholders and assignment nodes
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
    print(hidden1.eval(feed_dict={X: [[10.0, 11.0]]}))
```

```
[[ 61.  83. 105.]]
```

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:

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 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*.

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

Momentum optimization

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.

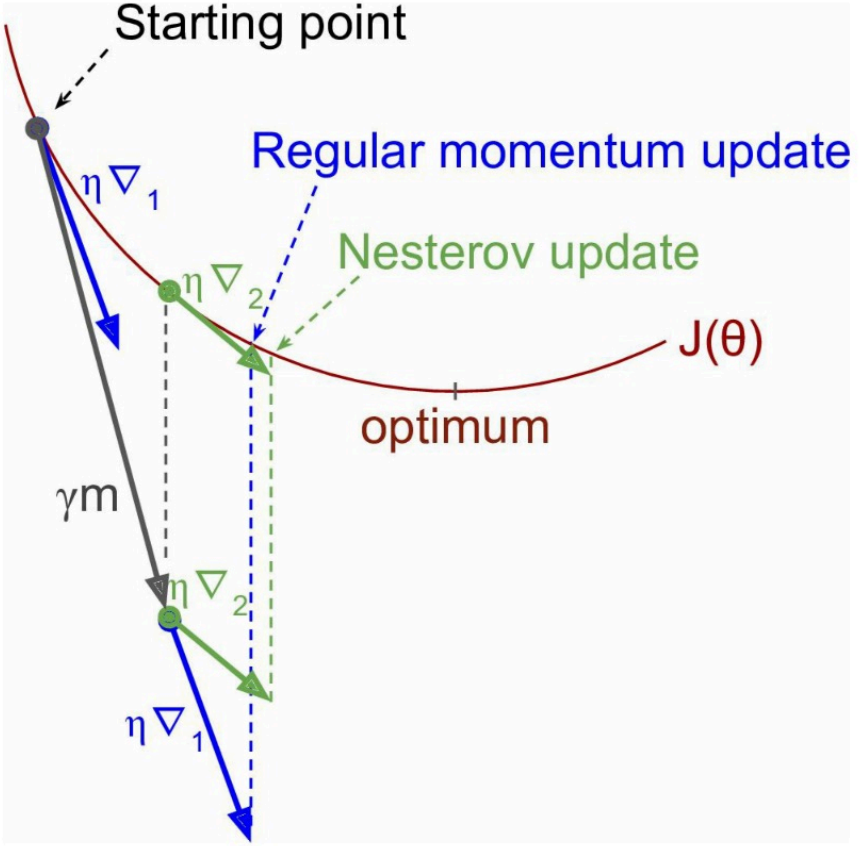
Momentum optimization cares a great deal about what previous gradients were: at each iteration, it adds the local gradient to the *momentum vector* **m** (multiplied by the learning rate η), and it updates the weights by simply subtracting this momentum vector.

```
In [98]: # Momentum optimization
         optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                                momentum=0.9)
```

Nesterov Accelerated Gradient

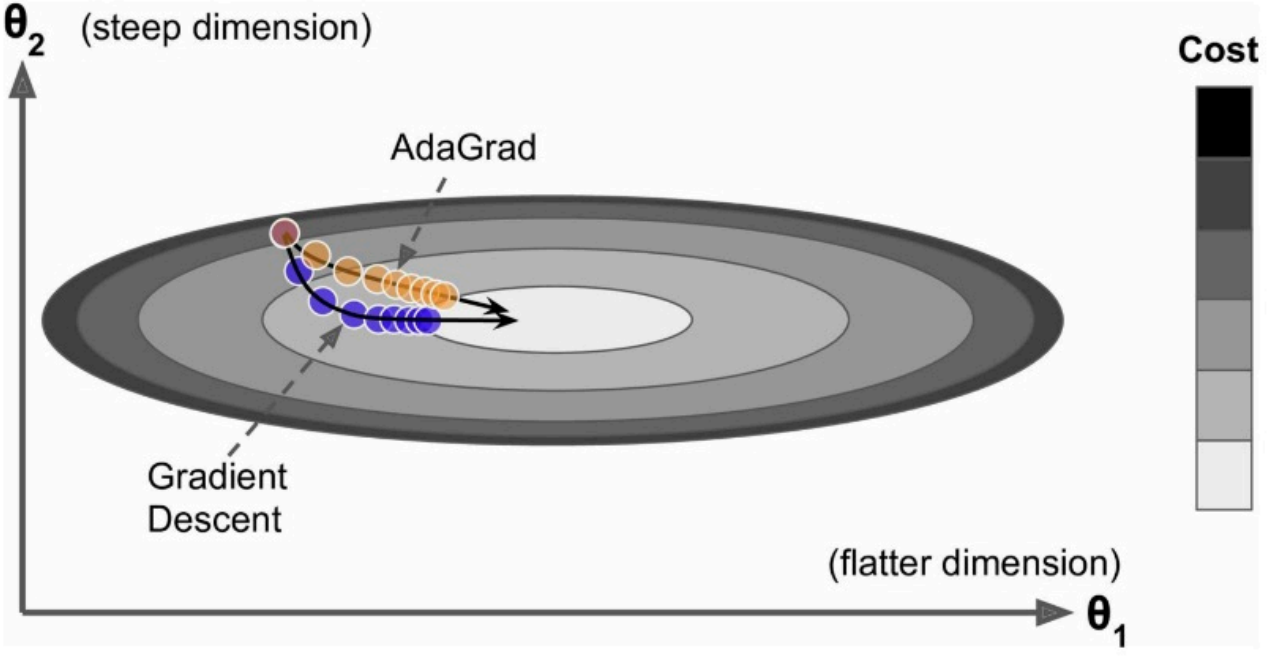
One small variant to Momentum optimization, proposed by Yurii Nesterov in 1983,is almost always faster than vanilla Momentum optimization. The idea of *Nesterov Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position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

```
In [99]: # Nesterov Accelerated Gradient
         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.




```
In [100]: # AdaGrad
          optimizer = tf.train.AdagradOptimizer(learning_rate=learning_rate)
```

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm 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.

In [101]:
```
# RMSProp
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

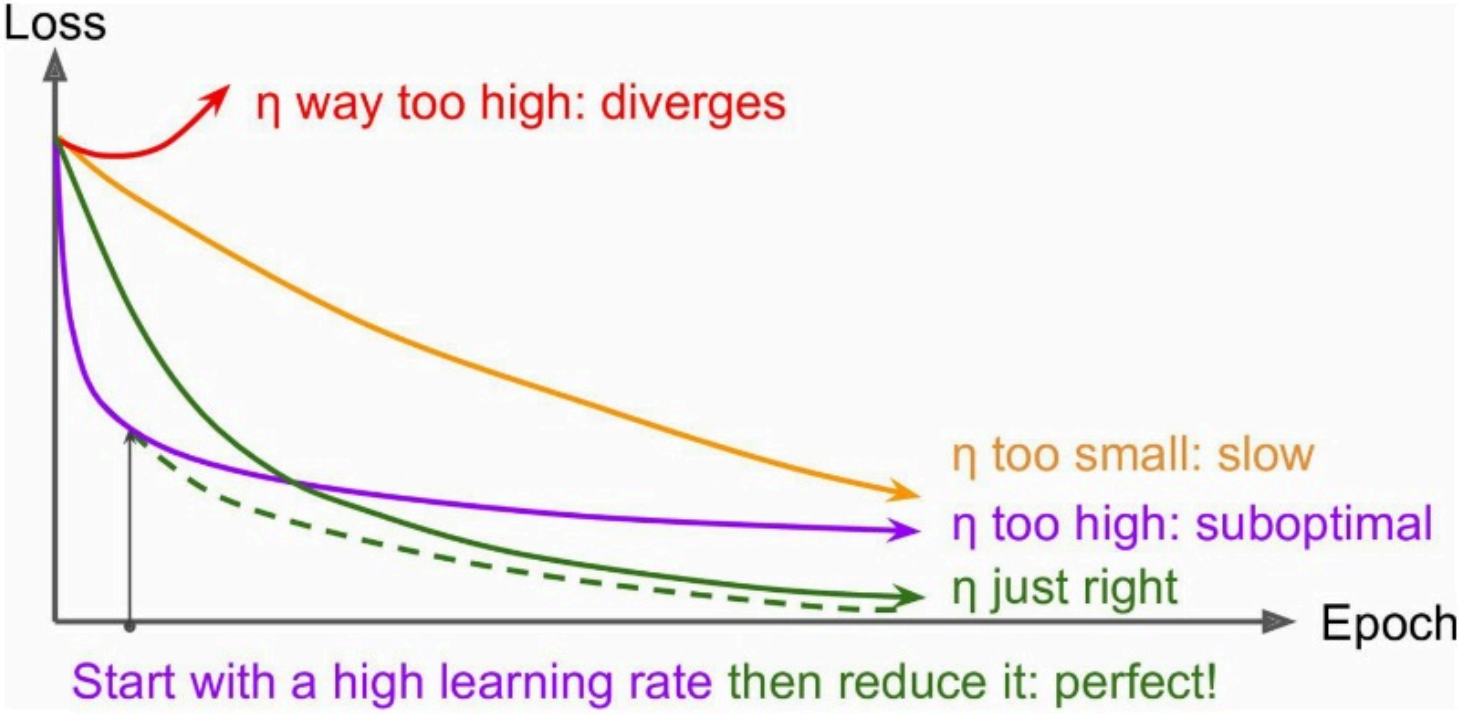
Adam Optimization

Adam, 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

In [102]:
```
# Adam Optimization
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge. 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.



Learning Rate Scheduling

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.

Learning Rate Scheduling

A 2013 paper 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.

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

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.

ℓ1 and ℓ2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ1 and ℓ2 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 ℓ1 regularization like this:

In [108]:
```
W1 = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("outputs/kernel:0")

scale = 0.001 # l1 regularization hyperparameter

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
    reg_losses = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
    loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

$\ell 1$ and $\ell 2$ Regularization

Alternatively, we will use Python's partial() function to avoid repeating the same arguments over and over again.

```
In [113]: my_dense_layer = partial(
              tf.layers.dense, activation=tf.nn.relu,
              kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

          with tf.name_scope("dnn"):
              hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
              hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
              logits = my_dense_layer(hidden2, n_outputs, activation=None,
                                      name="outputs")
```

You just need to add these regularization losses to your overall loss, like this:

```
In [114]: with tf.name_scope("loss"):
              xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                  labels=y, logits=logits)
              base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
              reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
              loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

Dropout

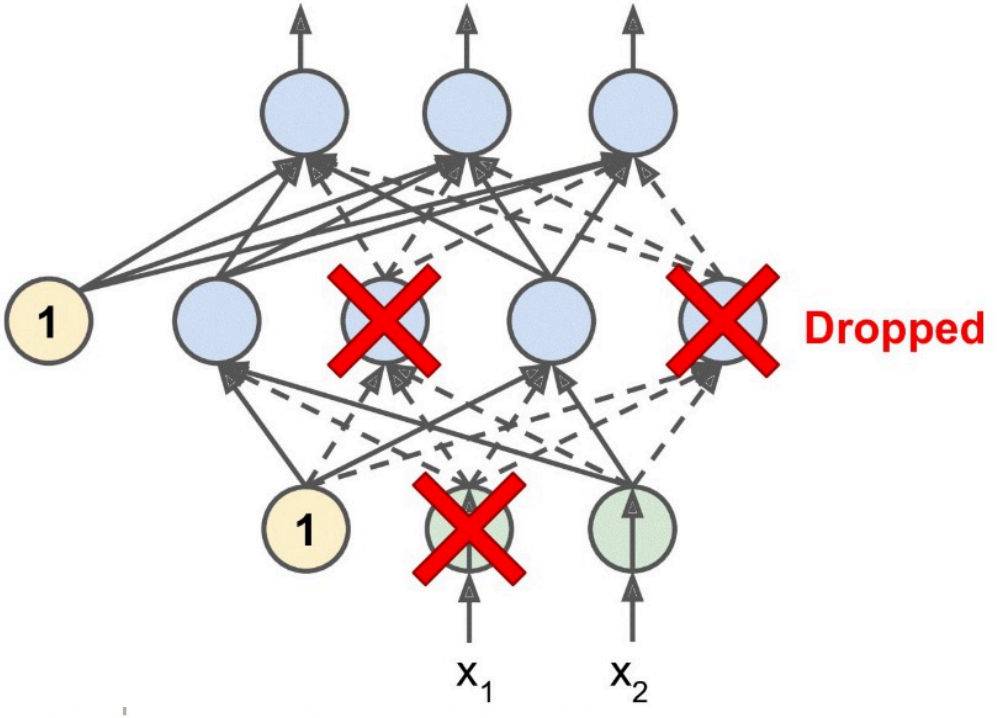
The most popular regularization technique for deep neural networks is arguably *dropout*.

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).

Dropout

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.

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, which can be seen as an averaging ensemble of all these smaller neural networks.



Dropout

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

```
In [118]: training = tf.placeholder_with_default(False, shape=(), name='training')

          dropout_rate = 0.5  # == 1 - keep_prob
          X_drop = tf.layers.dropout(X, dropout_rate, training=training)

          with tf.name_scope("dnn"):
              hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                                        name="hidden1")
              hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
              hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                                        name="hidden2")
              hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
              logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")
```

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 ( ). 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).

Max-Norm Regularization

A cleaner solution is to create a max_norm_regularizer() function and use it just like the earlier l1_regularizer() function:

In [106]:
```
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)
        return None # there is no regularization loss term
    return max_norm
```

Max-Norm Regularization

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

In [108]:
```
max_norm_reg = max_norm_regularizer(threshold=1.0)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Max-Norm Regularization

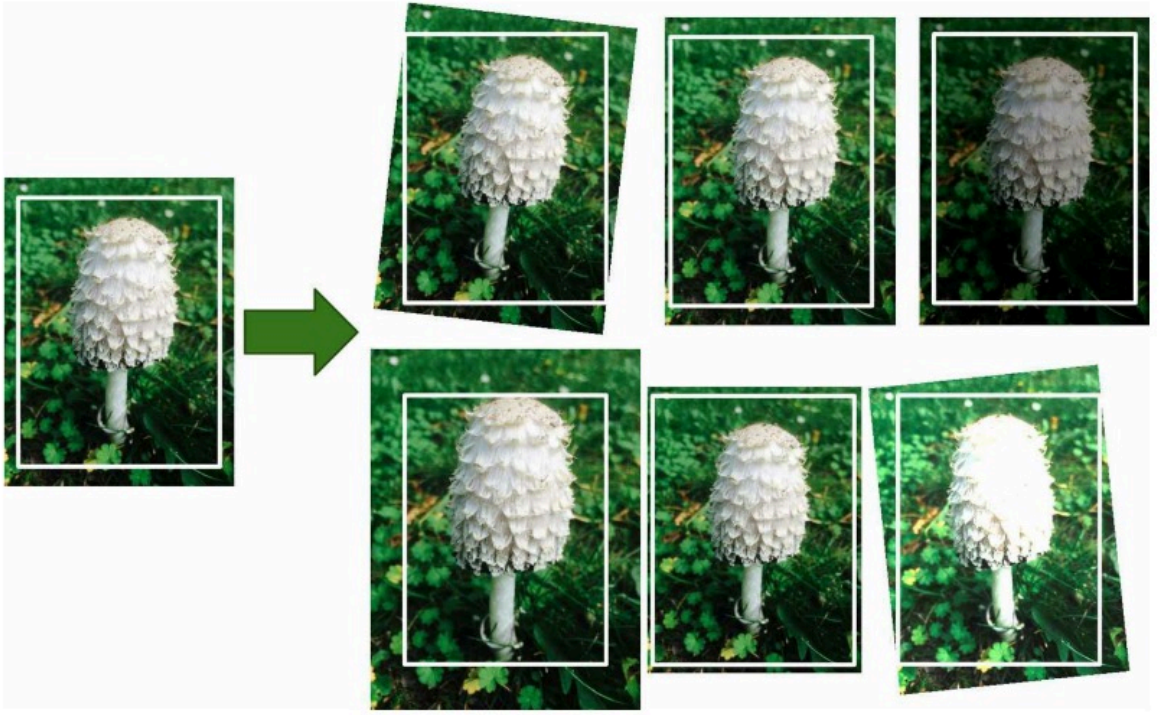
Note that max-norm regularization does not require adding a regularization loss term to your overall loss function, so the max_norm() function returnsNone. But you still need to be able to run the clip_weights operation after each training step, so you need to be able to get a handle on it.

```
In [111]: clip_all_weights = tf.get_collection("max_norm")

          with tf.Session() as sess:
              init.run()
              for epoch in range(n_epochs):
                  for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
                      sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
                      sess.run(clip_all_weights)
                  acc_valid = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
                  print(epoch, "Validation accuracy:", acc_valid)

              save_path = saver.save(sess, "./tf_logs/MaxNorm/my_model_final.ckpt")
```

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.

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.



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 below will work fine in most cases.

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

Practical Guidelines

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, 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 ℓ1 regularization to the mix. If you need an even sparser model, you can try using FTRL instead of Adam optimization, along with ℓ1 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.

SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院