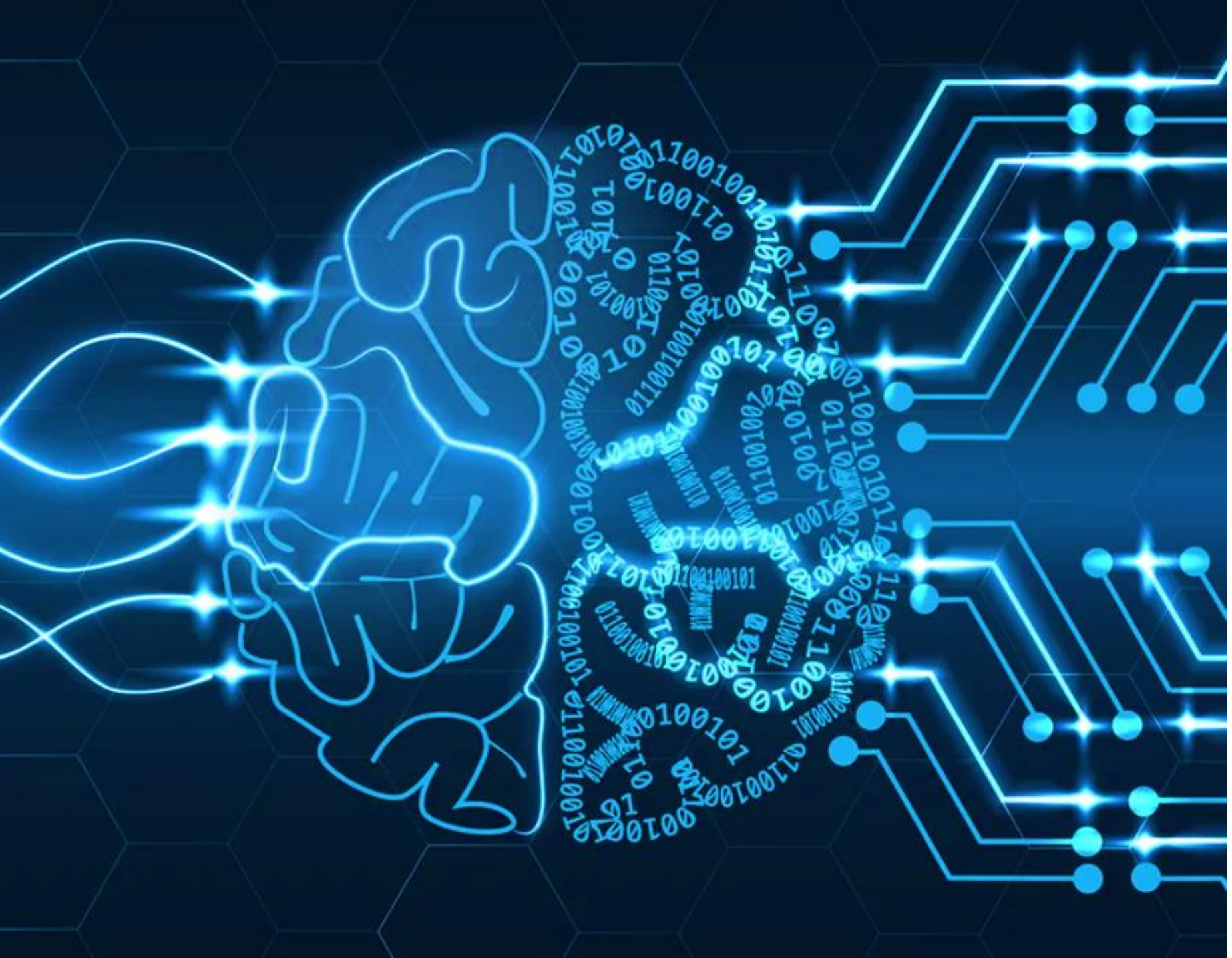


# شبکه‌های عصبی مصنوعی

اردیبهشت ۱۴۰۲


دانشکده مهندسی صنایع

دانشگاه صنعتی شریف

دکتر مهدی شریف‌زاده

سعیدرضا زواشکیانی


بسم الله الرحمن الرحیم

# Training Deep Neural Networks

Training a deep DNN isn't easy. Here are some of the problems you could run into:

- You may be faced with the tricky **vanishing gradients problem** or the related **exploding gradients problem**. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might **not have enough training data** for such a large network, or it might be too **costly to label**.
- Training may be extremely **slow**.
- A model with millions of parameters would severely risk **overfitting** the training set, especially if there are not enough training instances or if they are too noisy.

# The Vanishing/Exploding Gradient Problems

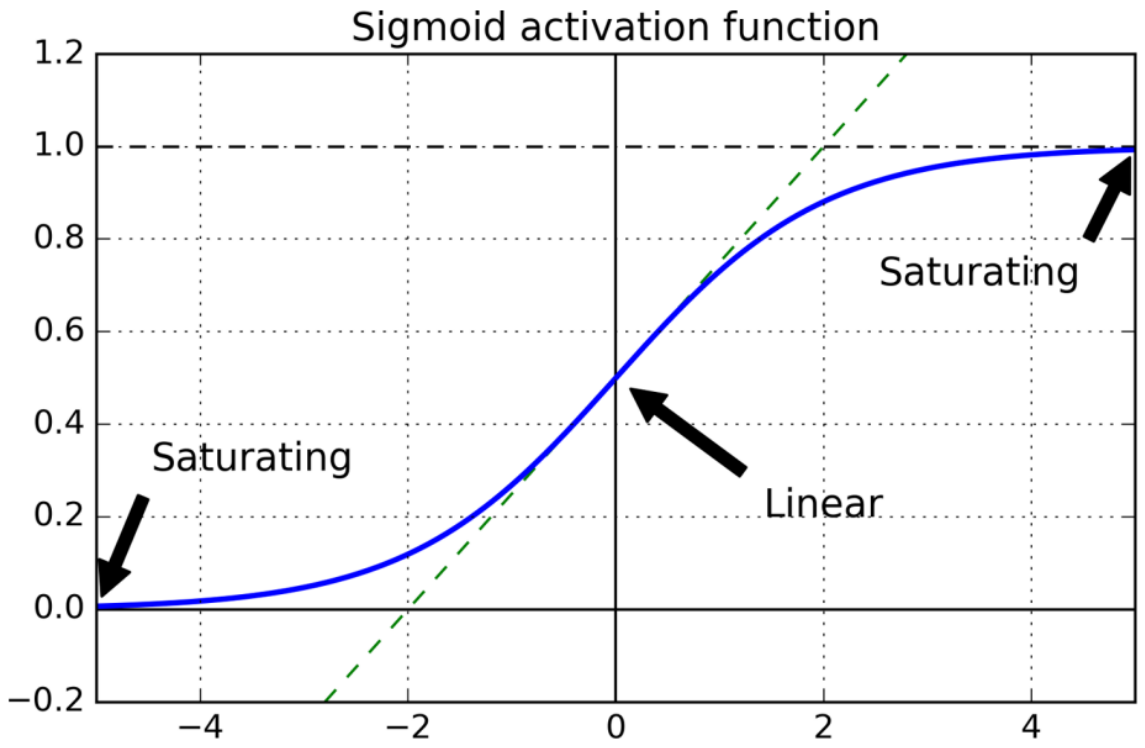
As discussed before, the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each 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 layers' connection weights virtually unchanged, and training never converges to a good solution. We call this the **vanishing gradients problem**.

# The Vanishing/Exploding Gradient Problems

In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the **exploding gradients problem,** which surfaces in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

# The Vanishing/Exploding Gradient Problems

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.



# Glorot and He Initialization

To solve the unstable gradient 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, we need the **variance of the outputs of each layer to be equal to the variance of its inputs**, and we need the **gradients to have equal variance** before and after flowing through a layer in the reverse direction

# Glorot and He Initialization

It is actually not possible to guarantee both **unless the layer has an equal number of inputs and outputs** (these numbers are called the fan-in and fan-out of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in Equation 11-1, where $fan_{avg} = (fan_{in} + fan_{out})/2$. This initialization strategy is called **Xavier initialization** or **Glorot initialization**, after the paper's first author.

**Normal distribution** with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$

Or a **uniform distribution** between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{avg}}}$

# Glorot and He Initialization

Some papers have provided similar strategies for different activation functions. These strategies differ only by the scale of the variance and whether they use $fan_{avg}$ or $fan_{in}$, as shown in Table 11-1 (for the uniform distribution, just compute $r = \sqrt{3\sigma^2}$). The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described shortly) is sometimes called **He initialization**, after the paper's first author. The SELU activation function will be explained later in this chapter. It should be used with **LeCun initialization**.

| Initialization | Activation functions | $\sigma^2$ (Normal) |
|---|---|---|
| Glorot | None, tanh, logistic, softmax | $1 / fan_{avg}$ |
| He | ReLU and variants | $2 / fan_{in}$ |
| LeCun | SELU | $1 / fan_{in}$ |

# Glorot and He Initialization

By default, Keras uses Glorot initialization with a uniform distribution. When creating a layer, you can change this to He initialization by setting kernel_initializer="he_uniform" or kernel_initializer="he_normal" like this:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

If you want He initialization with a uniform distribution but based on $fan_{avg}$ rather than $fan_{in}$ , you can use the VarianceScaling initializer like this:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,
mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid",
kernel_initializer=he_avg_init)
```

# Nonsaturating Activation Functions

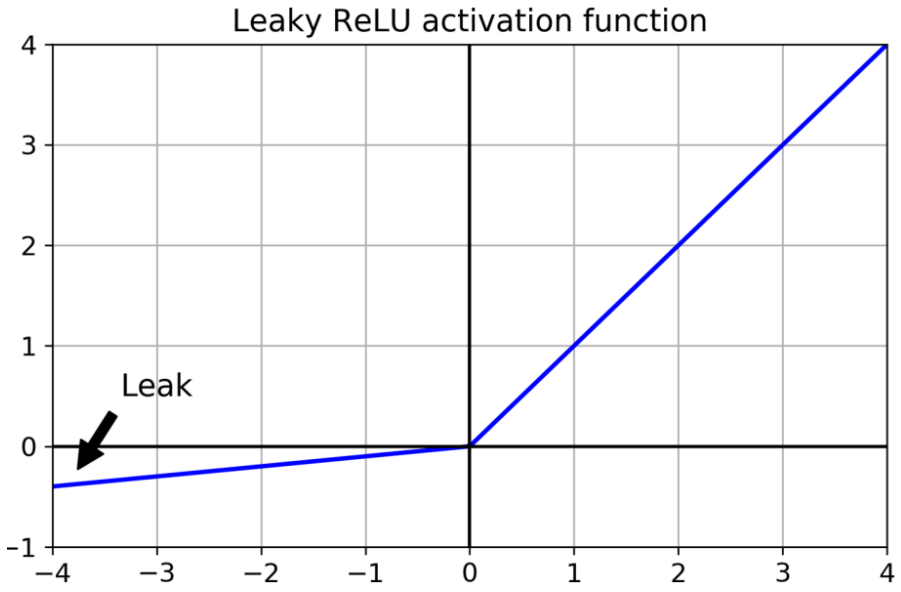
The problems with unstable gradients were in part due to a **poor choice of activation function**. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the **ReLU activation function**, mostly because it does not saturate for positive values (and because it is fast to compute).

# Nonsaturating Activation Functions

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. A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are **negative** for all instances in the training set. When this happens, it just keeps outputting zeros, and Gradient Descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative

# Nonsaturating Activation Functions

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 $LeakyReLU_{\alpha}(z) = \max(\alpha z, z)$. The hyperparameter $\alpha$ 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.



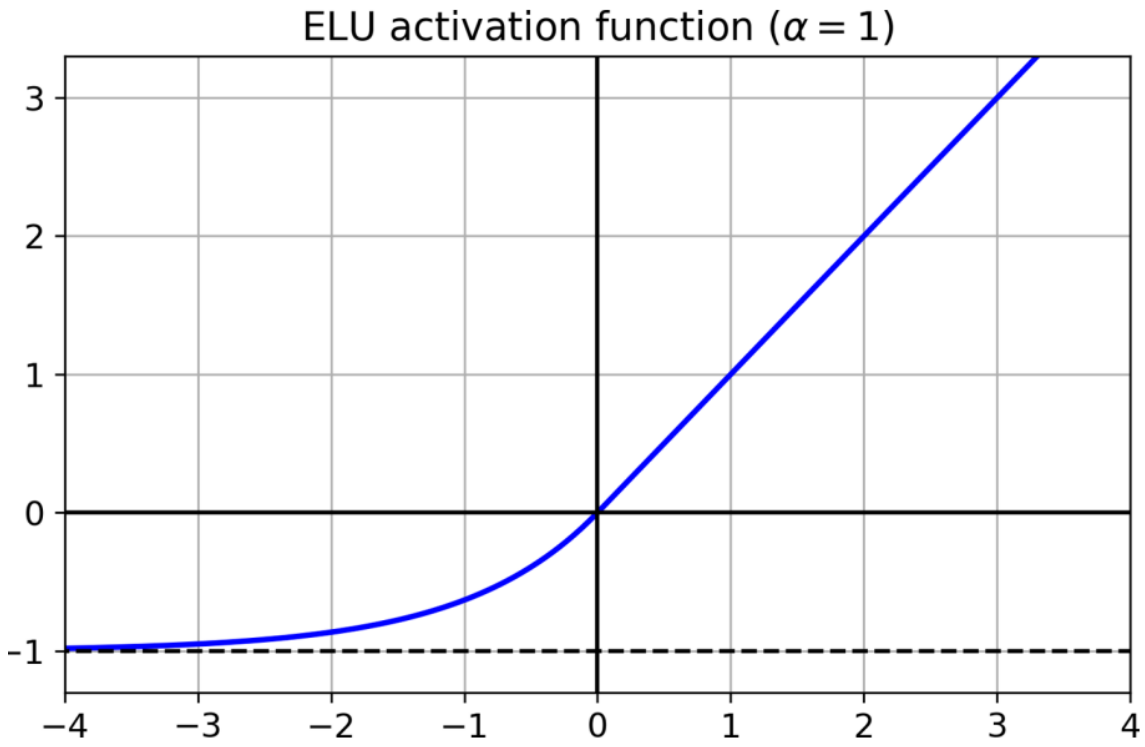
# Nonsaturating Activation Functions

Variants of Leaky ReLU:

- ***Randomized leaky ReLU*** **(RReLU),** where α is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set)
- ***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). PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

# Nonsaturating Activation Functions

***Exponential linear unit*** **(ELU)** is another activation function that outperformed all the ReLU variants: training time was reduced, and the neural network performed better on the test set.




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

# Nonsaturating Activation Functions: ReLU vs ELU

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter $\alpha$ 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.
- It has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
- If $\alpha$ is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps **speed up Gradient Descent** since it does not bounce as much to the left and right of $z = 0$.

# Nonsaturating Activation Functions: ELU

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

# Nonsaturating Activation Functions: SELU

**Scaled ELU (SELU)** activation function is a scaled variant of the ELU activation function. If you build a neural network composed exclusively of a stack of **dense layers**, and if all hidden layers use the SELU activation function, then the network will **self-normalize**: the output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function often **significantly outperforms** other activation functions for such neural nets (especially deep ones).

$$\begin{cases} f(x)=\lambda x & \text{if } x>0 \\ (x)=\lambda\alpha(ex-1) & \text{if } x\leq 0 \end{cases} \qquad \lambda \approx 1.0507 \qquad a \approx 1.6733$$

# Nonsaturating Activation Functions: SELU

There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be **standardized** (mean 0 and standard deviation 1).
- Every hidden layer's weights must be initialized with **LeCun normal initialization**. In Keras, this means setting kernel_initializer="lecun_normal".
- The network's architecture must be **sequential**. Unfortunately, if you try to use SELU in nonsequential architectures, such as recurrent networks (see Chapter 15) or networks with skip connections (i.e., connections that skip layers, such as in Wide & Deep nets), self-normalization will not be guaranteed, so SELU will not necessarily outperform other activation functions.

# Nonsaturating Activation Functions

To use the leaky ReLU activation function, create a LeakyReLU layer and add it to your model just after the layer you want to apply it to:

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

For SELU activation, set activation="selu" and kernel_initializer="lecun_normal" when creating a layer:

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

# Batch Normalization

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

**Batch Normalization** consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply **zero-centers and normalizes each input**, then **scales and shifts** the result using two new parameter vectors per layer: one for scaling, the other for shifting.

# Batch Normalization

In other words, the operation lets the model **learn** the **optimal scale and mean of each of the layer's inputs**. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (e.g., using a StandardScaler); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

# Batch Normalization

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

2. $$\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left( \mathbf{x}^{(i)} - \boldsymbol{\mu}_B \right)^2$$

3. $$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$$

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

# Batch Normalization

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at **test time**? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's mean and standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable.

# Batch Normalization

One solution could be to wait until the end of training, then run the **whole training set** through the neural network and compute the mean and standard deviation of each input of the BN layer. These "final" input means and standard deviations could then be used instead of the batch input means and standard deviations when making predictions. However, most implementations of Batch Normalization estimate these final statistics during training by using **a moving average of the layer's input means and standard deviations.**

$$SMA = \frac{A1 + A2 + \cdots + An}{n}$$

$where$:

$A = Average\ in\ period\ n$

$n = Number\ of\ time\ periods$

# Batch Normalization

This is what Keras does automatically when you use the BatchNormalization layer.

To sum up, four parameter vectors are learned in each batch-normalized layer: $\boldsymbol{\gamma}$ (the output scale vector) and $\boldsymbol{\beta}$ (the output offset vector) are learned through regular backpropagation, and $\boldsymbol{\mu}$ (the final input mean vector) and $\boldsymbol{\sigma}$ (the final input standard deviation vector) are estimated using an **exponential moving average**.

Note that $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are estimated during training, but they are used only after training

$$
\begin{aligned}
y[n] &= \alpha x[n] + (1-\alpha) y[n-1] \\
&= \alpha x[n] + (1-\alpha)\left(\alpha x[n-1] + (1-\alpha) y[n-2]\right) \\
&= \alpha x[n] + (1-\alpha)\left(\alpha x[n-1] + (1-\alpha)\left(\alpha x[n-2] + (1-\alpha) y[n-3]\right)\right) \\
&\cdots \\
&= \alpha \sum_{k=0}^{n} (1-\alpha)^k x[n-k]
\end{aligned}
$$

Note: the weighting factor of previous inputs decreases exponentially.

# Batch Normalization

Batch Normalization considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). 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. The authors were able to use much larger learning rates, significantly speeding up the learning process. Finally, Batch Normalization acts like a regularizer, reducing the need for other regularization techniques.

# Batch Normalization

As with most things with Keras, implementing Batch Normalization is simple and intuitive. Just add a BatchNormalization layer before or after each hidden layer's activation function, and optionally add a BN layer as well as the first layer in your model. For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images)

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu",
kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu",
kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

# Batch Normalization

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

# Batch Normalization

Let’s look at the parameters of the first BN layer. Two are trainable (by backpropagation), and two are not:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

# Batch Normalization

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, as which is preferable seems to depend on the task —you can experiment with this too to see which option works best on your dataset. To add the BN layers before the activation functions, you must remove the activation function from the hidden layers and add them as separate layers after the BN layers. Moreover, since a Batch Normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer (just pass use_bias=False when creating it):

# Batch Normalization

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal",

use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal",
use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

# Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping. This technique is most often used in recurrent neural networks, as Batch Normalization is tricky to use in RNNs. For other types of networks, BN is usually sufficient.

In Keras, implementing Gradient Clipping is just a matter of setting the clipvalue or clipnorm argument when creating an optimizer, like this:

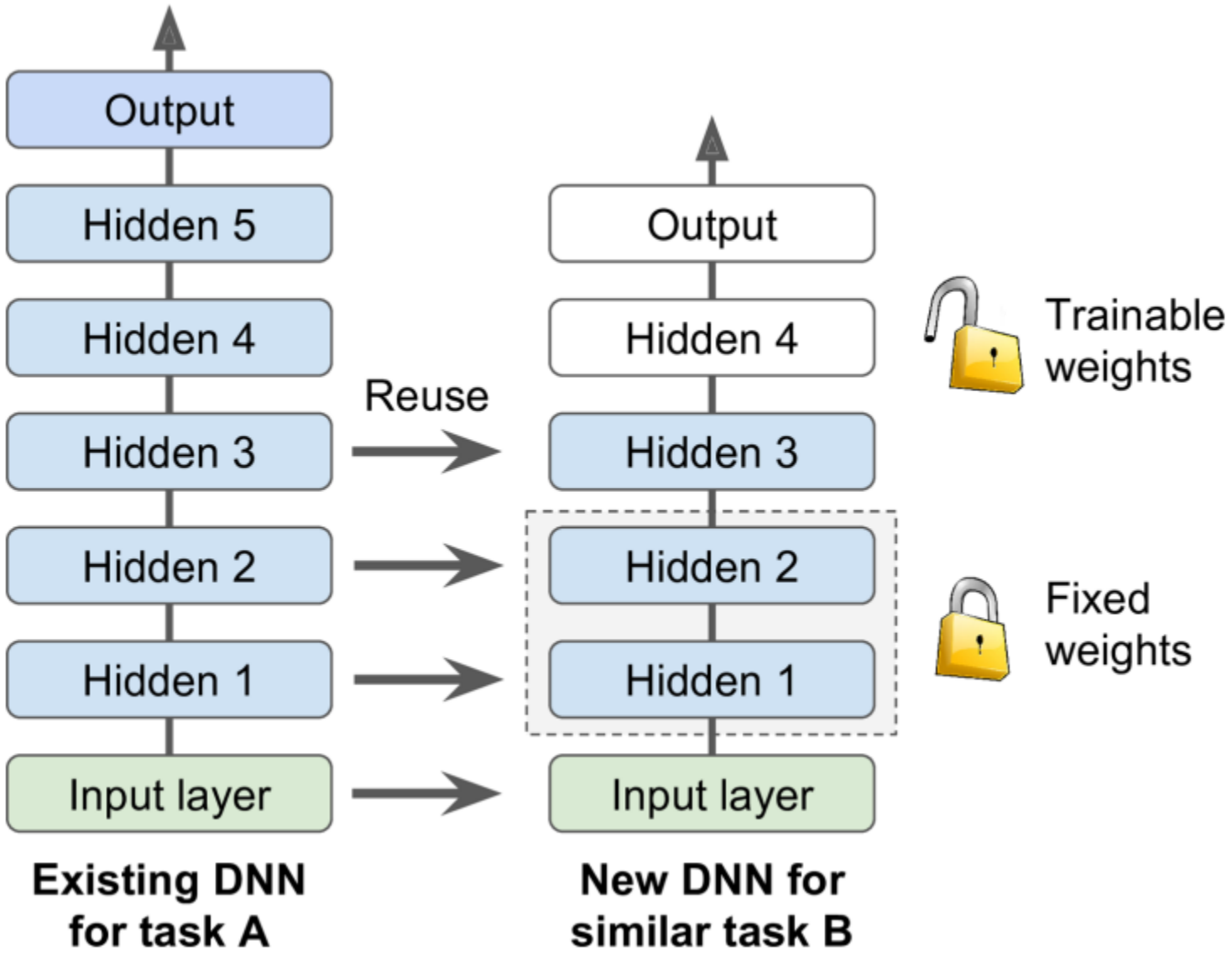
```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

# 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 (we will discuss how to find them in Chapter 14), then reuse the lower layers of this network. This technique is called **transfer learning.** It will not only **speed up training** considerably, but also require significantly **less training data**.

Suppose 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, even partly overlapping, so you should try to reuse parts of the first network

# Reusing Pretrained Layers



# Reusing Pretrained Layers

The output layer of the original model should usually be replaced because 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.

# Reusing Pretrained Layers

Try **freezing all the reused layers** first (i.e., make their weights nontrainable so that Gradient Descent won't modify them), 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. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

# Reusing Pretrained Layers

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the 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 adding more hidden layers.

# Transfer Learning with Keras

Suppose the Fashion MNIST dataset only contained eight classes. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let's call this model A. You now want to tackle a different task: you have images of sandals and shirts, and you want to train a binary classifier (positive=shirt, negative=sandal). Your dataset is quite small; you only have 200 labeled images. When you train a new model for this task (let's call it model B) with the same architecture as model A, it performs reasonably well (97.2% accuracy). But since it's a much easier task (there are just two classes), you were hoping for more. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

# Transfer Learning with Keras

First, you need to load model A and create a new model based on that model's layers. Let's reuse all the layers except for the output layer:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

# Transfer Learning with Keras

Note that model_A and model_B_on_A now share some layers. When you train model_B_on_A, it will also affect model_A. If you want to avoid that, you need to clone model_A before you reuse its layers. To do this, you clone model A's architecture with clone.model(), then copy its weights (since clone_model() does not clone the weights):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

# Transfer Learning with Keras

Now you could train model_B_on_A for task B, but since the new output layer was initialized randomly it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights. To do this, set every layer's trainable attribute to False and compile the model:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])
```

# Transfer Learning with Keras

Now you can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights:

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

# Transfer Learning with Keras

So, what's the final verdict? Well, this model's test accuracy is 99.25%, which means that transfer learning reduced the error rate from 2.8% down to almost 0.7%! That's a factor of four!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

# Transfer Learning with Keras

Are you convinced? You shouldn't be: I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses." When a paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results (which may be due to sheer luck), without mentioning how many failures they encountered on the way. Most of the time, this is not malicious at all, but it is part of the reason so many results in science can never be reproduced.

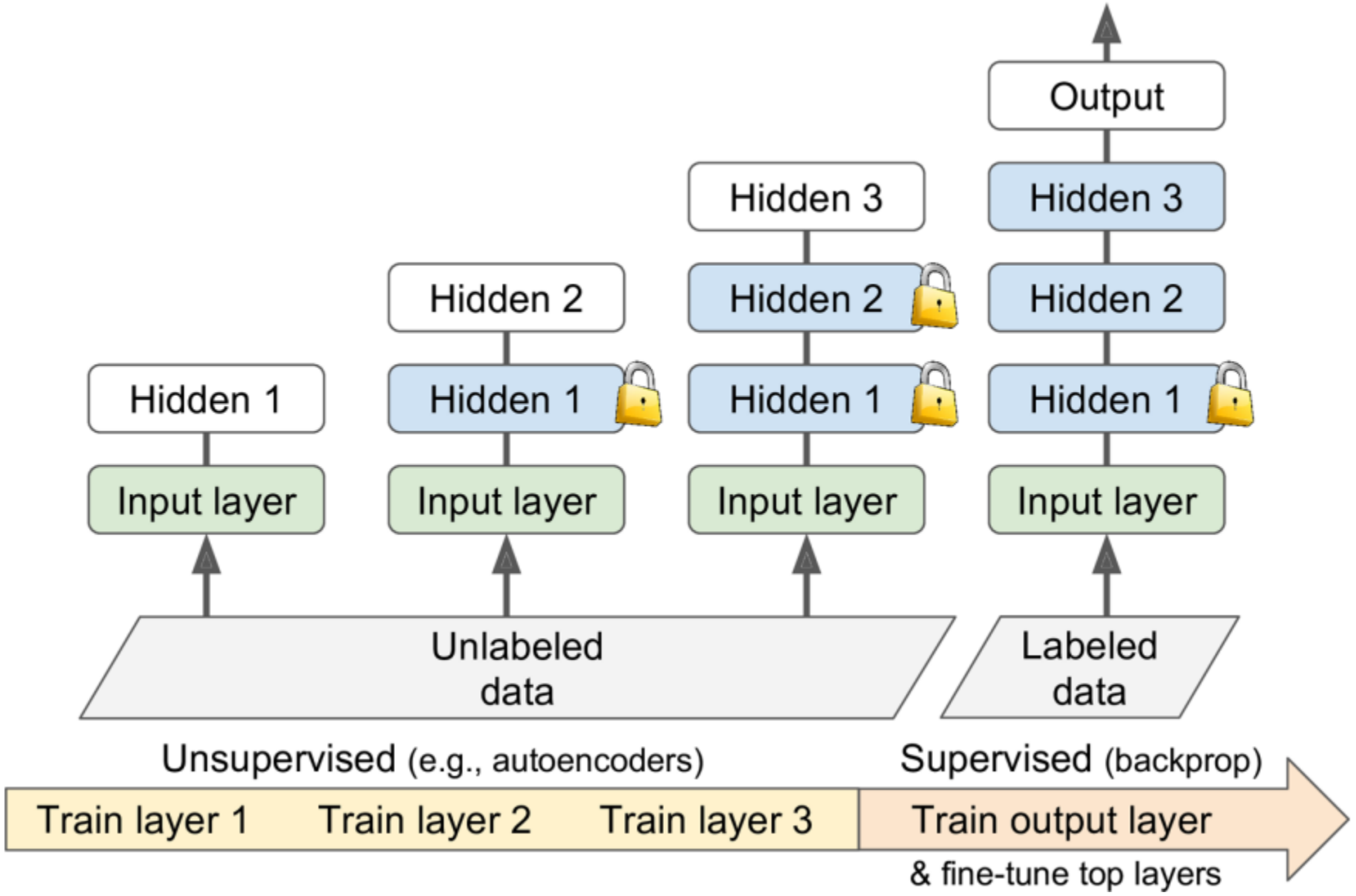
# Transfer Learning with Keras

Why did I cheat? It turns out that transfer learning **does not work** very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful in other tasks. Transfer learning works best with deep convolutional neural networks, which tend to learn feature detectors that are much more general (especially in the lower layers).

# 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 hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform unsupervised pretraining (see Figure 11-5). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model. Then you can reuse the lower layers of the autoencoder or the lower layers of the GAN's discriminator, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

# Unsupervised Pretraining



# Pretraining on an Auxiliary Task

If you do not have much labeled training data, 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.

# Pretraining on an Auxiliary Task

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. You could, however, gather a lot of pictures of random people on the web 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 that uses little training data.

# Pretraining on an Auxiliary Task

For natural language processing (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What ___ you saying?” is probably “are” or “were”). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data

# 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 (possibly built on an auxiliary task or using unsupervised learning). 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 algorithms: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

# Faster Optimizers: Momentum

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**. In contrast, regular Gradient Descent will simply take small, regular steps down the slope, so the algorithm will take much more time to reach the bottom.

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

# Faster Optimizers: Momentum

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the momentum vector m (multiplied by the learning rate η), and it updates the weights by adding this momentum vector. In other words, the gradient is used for acceleration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the momentum, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

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

# Faster Optimizers: Momentum

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 $\eta$ multiplied by $\frac{1}{1-\beta}$ (ignoring the sign). 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!

# Faster Optimizers: Momentum

This allows momentum optimization to escape from plateaus much faster than Gradient Descent. We saw that when the inputs have very different scales, the cost function will look like an elongated bowl. 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 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.

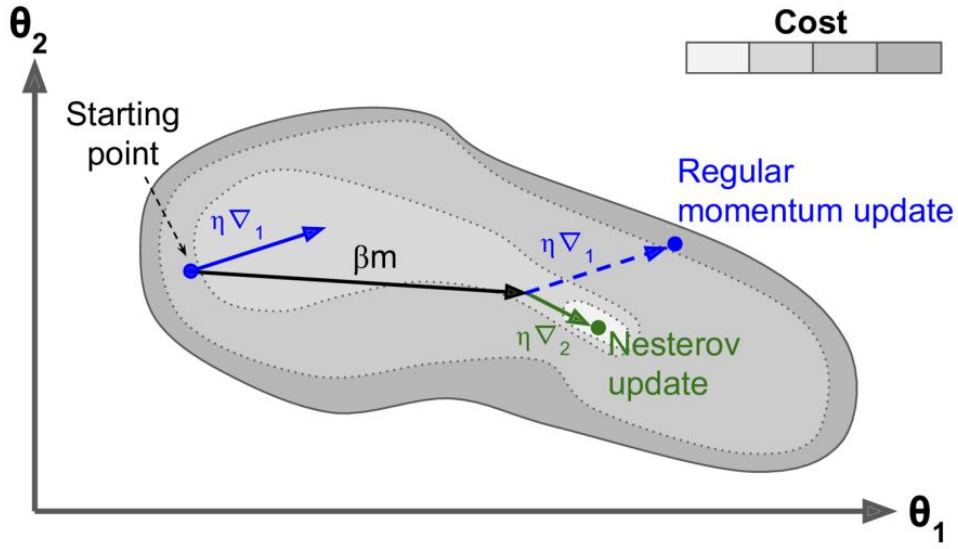
# Faster Optimizers: Nesterov

One small variant to momentum optimization, proposed by Yurii Nesterov in 1983, is almost always faster than vanilla momentum optimization. The Nesterov Accelerated Gradient (NAG) method, also known as Nesterov momentum optimization, measures the gradient of the cost function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta \boldsymbol{m}$

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta} + \beta \mathbf{m}\right)$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

# Faster Optimizers: Nesterov

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 the gradient at the original position, as you can see in Figure 11-6 (where $\nabla_1$ represents the gradient of the cost function measured at the starting point $\boldsymbol{\theta}$, and $\nabla_2$ represents the gradient at the point located at $\boldsymbol{\theta} + \beta \boldsymbol{m}$).



# Faster Optimizers: Nesterov

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, $\nabla_1$ continues to push farther across the valley, while $\nabla_2$ pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

NAG is generally faster than regular momentum optimization. To use it, simply set nesterov=True when creating the SGD optimizer:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

# Faster Optimizers: AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The AdaGrad algorithm achieves this correction by **scaling down** the gradient vector along the steepest dimensions

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

# Faster Optimizers: AdaGrad

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

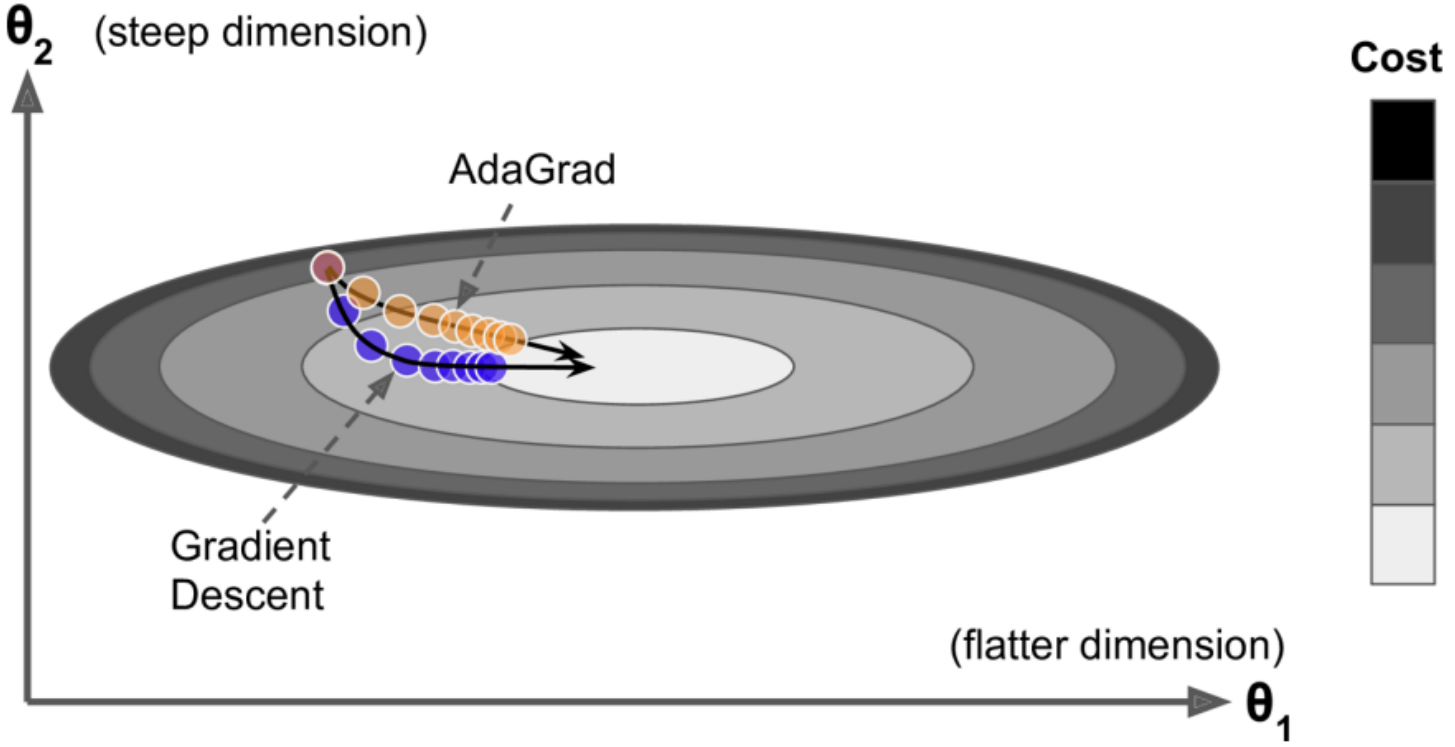
1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

# Faster Optimizers: AdaGrad

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{s + \varepsilon}$ (the $\oslash$ symbol represents the element-wise division, and ε is a smoothing term to avoid division by zero, typically set to $10^{-10}$ ). This vectorized form is equivalent to simultaneously computing $\theta_i \leftarrow \theta_i - \eta \, \partial J(\theta)/\partial\theta_i/\sqrt{s_i + \varepsilon}$ for all parameters $\theta_i$ .

# Faster Optimizers: AdaGrad

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. One additional benefit is that it requires much less tuning of the learning rate hyperparameter $\eta$.



# Faster Optimizers: AdaGrad

AdaGrad frequently performs well for simple quadratic problems, but 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 Keras has an Adagrad optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). Still, understanding AdaGrad is helpful to grasp the other adaptive learning rate optimizers.

# Faster Optimizers: RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and 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.

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.

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

# Faster Optimizers: RMSProp

As you might expect, Keras has an RMSprop optimizer:

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

# Faster Optimizers: Adam & Nadam

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

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - {\beta_1}^{\mathsf{T}}}$
4. $\hat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - {\beta_2}^{\mathsf{T}}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$

# Faster Optimizers: Adam & Nadam

In this equation, 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 $\boldsymbol{m}$ and $\boldsymbol{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost $\boldsymbol{m}$ and $\boldsymbol{s}$ at the beginning of training.

# Faster Optimizers: Adam & Nadam

The momentum decay hyperparameter $\beta_1$ is typically initialized to 0.9, while the scaling decay hyperparameter $\beta_2$ is often initialized to 0.999. As earlier, the smoothing term $\varepsilon$ is usually initialized to a tiny number such as $10^{-7}$ . These are the default values for the Adam class (to be precise, epsilon defaults to None, which tells Keras to use keras.backend.epsilon(), which defaults to $10^{-7}$ ; you can change it using keras.backend.set_epsilon()). Here is how to create an Adam optimizer using Keras:

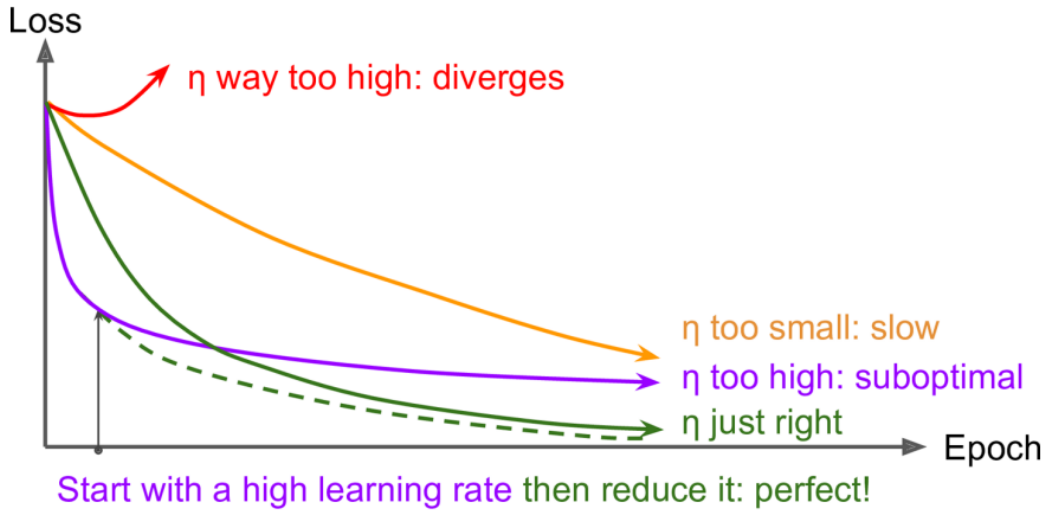
```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

# Faster Optimizers: Adam & Nadam

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

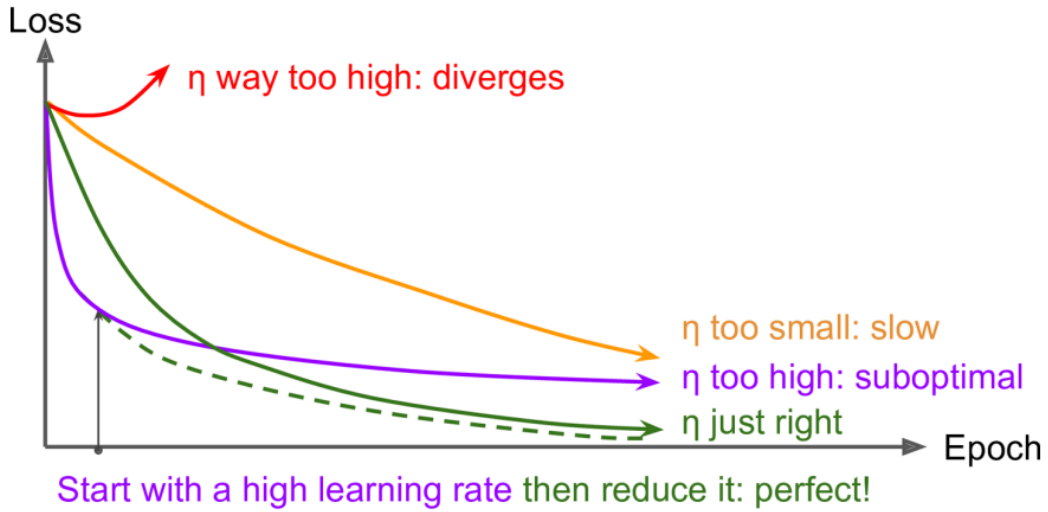
# Learning Rate Scheduling

Finding a good learning rate is very important. If you set it much too high, training may diverge (as we discussed in “Gradient Descent”). 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 really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution



# Learning Rate Scheduling

you can find a good learning rate by training the model for a few hundred iterations, exponentially increasing the learning rate from a very small value to a very large value, and then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up. You can then reinitialize your model and train it with that learning rate.



# Learning Rate Scheduling

But you can do better than a constant learning rate: if you start with a large learning rate and then reduce it once training 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. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called **learning schedules**

# Learning Rate Scheduling

- **Power scheduling**

  Set the learning rate to a function of the iteration number t:

$$\eta(t) = \eta_0 / \left(1 + \frac{t}{s}\right)^c$$

  The initial learning rate $\eta_0$ , the power c (typically set to 1), and the steps $s$ are hyperparameters. The learning rate drops at each step. After s steps, it is down to $\eta_0/2$. After s more steps, it is down to $\eta_0/3$, then it goes down to $\eta_0/4$, then $\eta_0/5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, power scheduling requires tuning $\eta_0$ and $s$ (and possibly $c$).

# Learning Rate Scheduling

- **Exponential scheduling**

  Set the learning rate to $\eta(t) = \eta_0 \, (0.1)^{\frac{t}{s}}$ . The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every s steps.

# Learning Rate Scheduling

- **Piecewise constant scheduling**

  Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fiddling around to figure out the right sequence of learning rates and how long to use each of them.

# Learning Rate Scheduling

- **Performance scheduling**

  Measure the **validation error every N steps** (just like for early stopping), and reduce the learning rate by a factor of $\lambda$ when the error stops dropping.

# Learning Rate Scheduling

Implementing power scheduling in Keras is the easiest option: just set the decay hyperparameter when creating an optimizer:

The decay is the inverse of s (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that c is equal to 1.

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

# Learning Rate Scheduling

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

Next, create a LearningRateScheduler callback, giving it the schedule function, and pass this callback to the fit() method:

```
lr_scheduler =
keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=
[lr_scheduler])
```

# Learning Rate Scheduling

The LearningRateScheduler will update the optimizer's learning_rate attribute at the beginning of each epoch. Updating the learning rate once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you can always write your own callback (see the "Exponential Scheduling" section of the notebook for an example). Updating the learning rate at every step makes sense if there are many steps per epoch. Alternatively, you can use the keras.optimizers.schedules approach, described shortly.

# Learning Rate Scheduling

The schedule function can optionally take the current learning rate as a second argument. For example, the following schedule function multiplies the previous learning rate by $0.1^{\frac{1}{20}}$ , which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1):

This implementation relies on the optimizer's initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

# Learning Rate Scheduling

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. Things are not so simple if your schedule function uses the epoch argument, however: the epoch does not get saved, and it gets reset to 0 every time you call the fit() method. If you were to continue training a model where it left off, this could lead to a very large learning rate, which would likely damage your model's weights. One solution is to manually set the fit() method's initial_epoch argument so the epoch starts at the right value.

# Learning Rate Scheduling

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want; see the “Piecewise Constant Scheduling” section of the notebook for an example), then create a LearningRateScheduler callback with this function and pass it to the fit() method, just like we did for exponential scheduling

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

# Learning Rate Scheduling

For performance scheduling, use the ReduceLROnPlateau callback. For example, if you pass the following callback to the fit() method, it will multiply the learning rate by 0.5 whenever the best validation loss does not improve for five consecutive epochs (other options are available; please check the documentation for more details):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

# Learning Rate Scheduling

Lastly, tf.keras offers an alternative way to implement learning rate scheduling: define the learning rate using one of the schedules available in keras.optimizers.schedules, then pass this learning rate to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as the exponential_decay_fn() function we defined earlier:

```
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size =
32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

# Regularization

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to **overfitting** the training set. We need regularization. We already implemented one of the best regularization techniques in Chapter 10: **early stopping**. Moreover, even though **Batch Normalization** was designed to solve the unstable gradients problems, it also acts like a pretty good regularizer. In this section we will examine other popular regularization techniques for neural networks: $\ell_1$ **and** $\ell_2$ **regularization, dropout**.

# Regularization

Just like you did in Chapter 4 for simple linear models, you can use $\ell_2$ regularization to constrain a neural network's connection weights, and/or $\ell_1$ regularization if you want a sparse model (with many weights equal to 0). Here is how to apply $\ell_2$ regularization to a Keras layer's connection weights, using a regularization factor of 0.01

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",

kernel_regularizer=keras.regularizers.l2(0.01))
```

# Regularization

The l2() function returns a regularizer that will be called at each step during training to compute the regularization loss. This is then added to the final loss. As you might expect, you can just use keras.regularizers.l1() if you want $\ell_1$ regularization; if you want both $\ell_1$ and $\ell_2$ regularization, use keras.regularizers.l1_l2() (specifying both regularization factors).

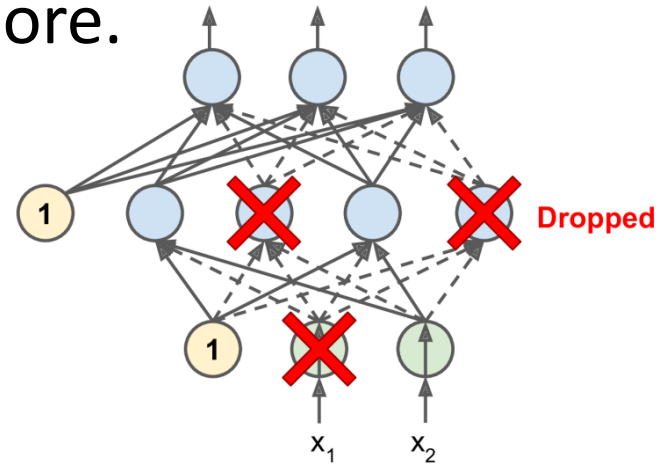
```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",

kernel_regularizer=keras.regularizers.l2(0.01))
```

# Dropout

**Dropout** is one of the most popular regularization techniques for deep neural networks. It has proven to be highly successful: even the state-of-the-art neural networks get 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%).

# Dropout

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always 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. The hyperparameter p is called the dropout rate, and it is typically set between 10% and 50%: closer to **20–30%** in recurrent neural nets and closer to **40–50%** in convolutional neural networks. After training, neurons don't get dropped anymore.



# Dropout

It’s surprising at first that this destructive 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 be forced to adapt its organization; it could not rely on any single person to work 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.

# Dropout

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

# Dropout

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 are 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 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 because 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

# Dropout

There is one small but important technical detail. Suppose p = 50%, in which case during testing a neuron would be connected to twice as many input neurons as it would be (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 will be 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).

# Dropout

To implement dropout using Keras, you can use the keras.layers.Dropout layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every Dense layer, using a dropout rate of 0.2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu",
kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu",
kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

# Dropout

If you observe that the model is **overfitting**, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate 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. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong. 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.