# 3.
# Faster Optimizers

# Speeding Up Training

- Ways to speed up training a large deep neural network (and reach a better solution):
  - applying a good initialization strategy for the connection weights
  - using a good activation function
  - using batch normalization
  - reusing parts of a pretrained network (possibly built for an auxiliary task or using unsupervised learning).
  - using a faster optimizer than the regular gradient descent optimizer.

# Momentum Optimization

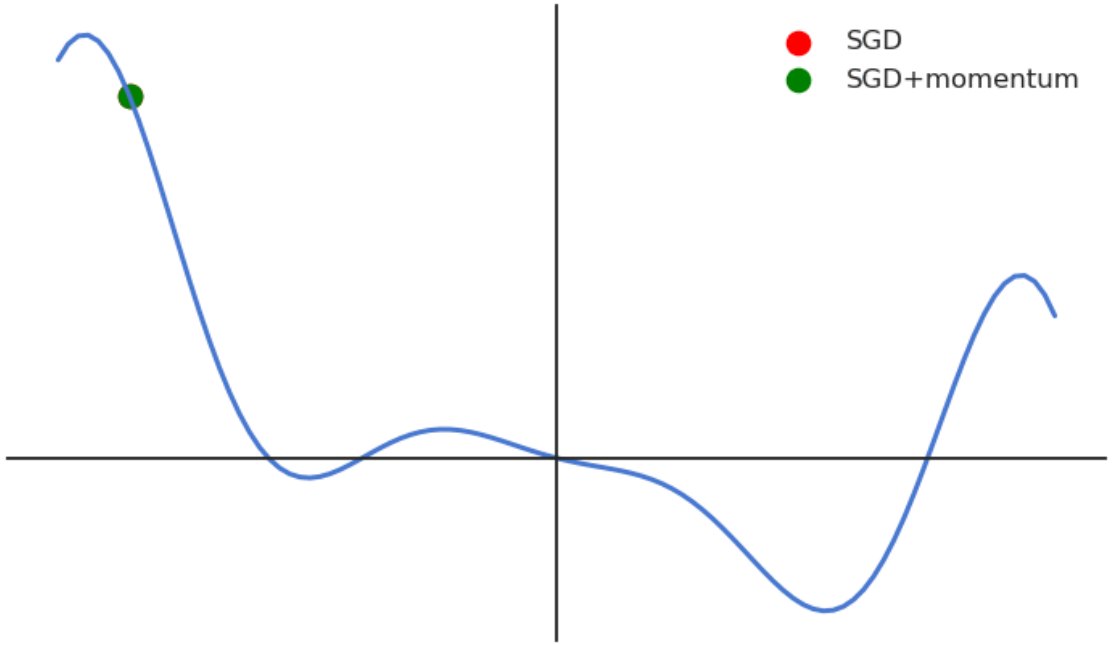
- Regular gradient descent: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.
  - It only relies on the current gradient to update the parameters
- Momentum optimization considers the weighted average of past gradients:

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

  - $\mathbf{m}$: the *momentum vector*
  - $\beta$: the *momentum hyperparameter*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.
- Regular gradient descent is generally much slower to reach the minimum than momentum optimization.

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

# Momentum Optimization



# Nesterov Accelerated Gradient

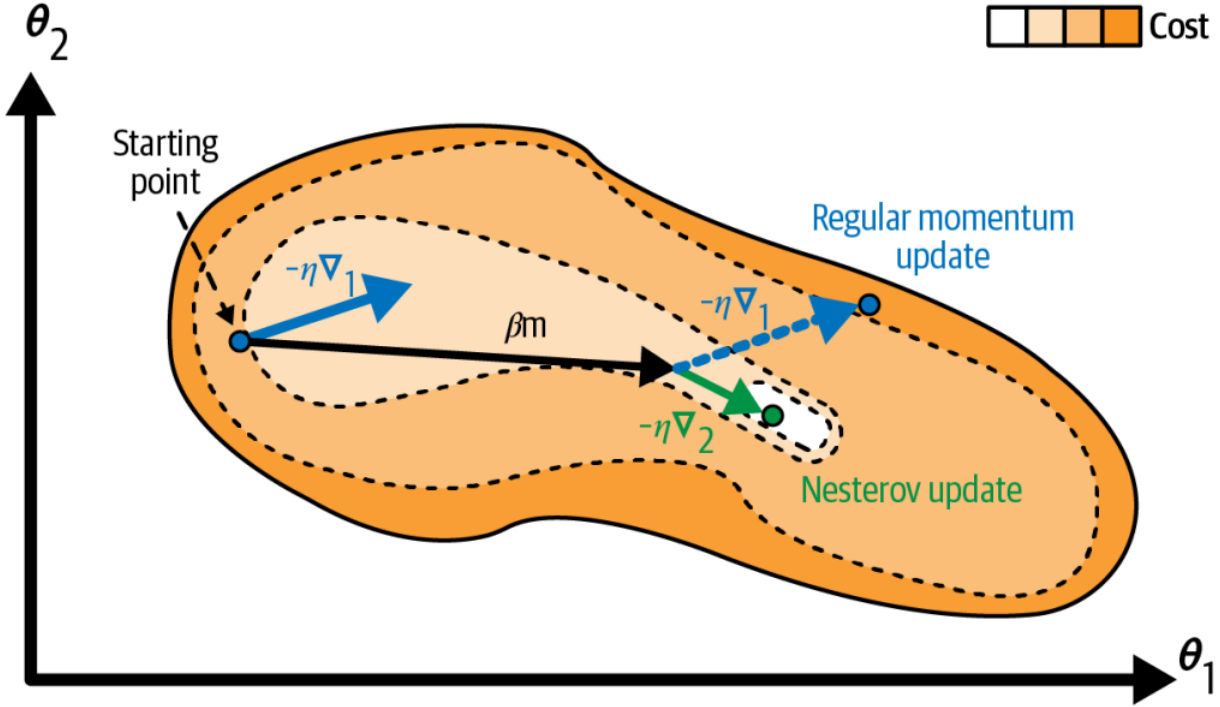
- The *Nesterov accelerated gradient* (NAG) method, aka. *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta \boldsymbol{m}$.

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

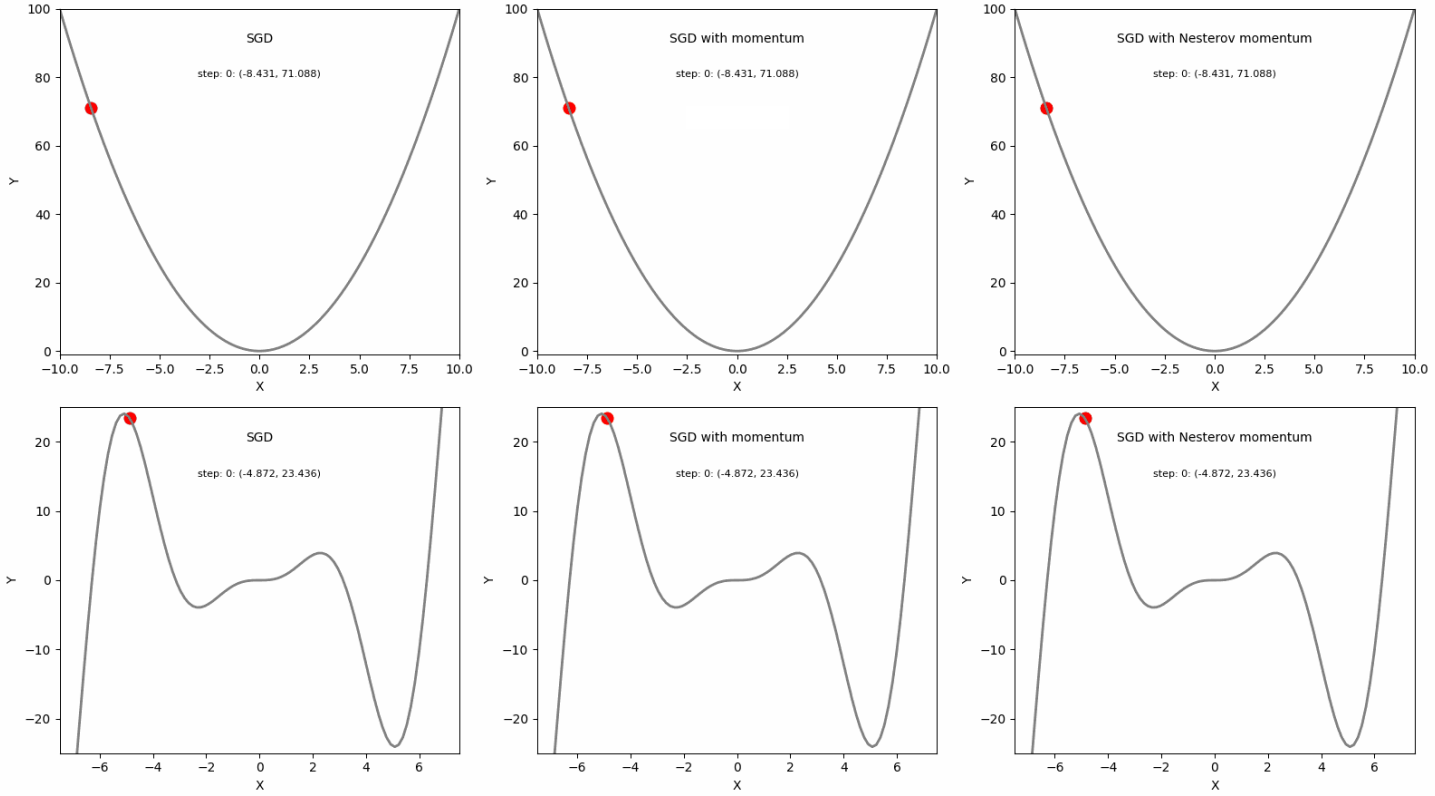
- This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position.

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

# Nesterov Accelerated Gradient

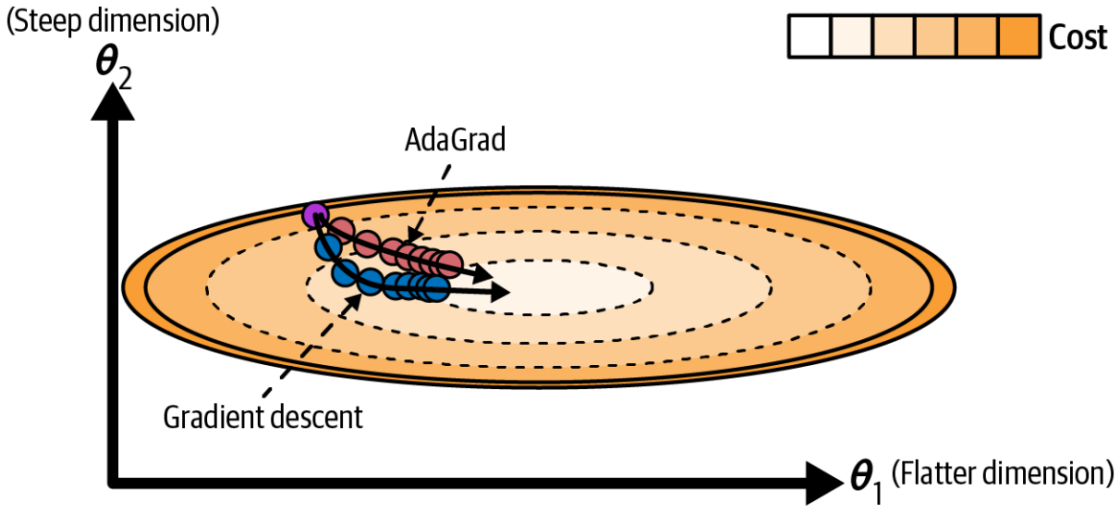


# Nesterov Accelerated Gradient



# AdaGrad

- The elongated bowl problem: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it slowly goes down to the bottom of the valley.
  - We want the algorithm correct its direction earlier to point a bit more toward the global optimum.



# AdaGrad

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

1. $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta})/\partial \theta_i)^2$
2. $\theta_i \leftarrow \theta_i - \eta\, (\partial J(\boldsymbol{\theta})/\partial \theta_i)/\sqrt{s_i + \epsilon}$

- The second step is almost identical to gradient descent, but with one difference: the gradient vector is scaled down by a factor of $\sqrt{\boldsymbol{s} + \varepsilon}$.
- The 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 points the resulting updates more directly toward the global optimum.
  - It also requires much less tuning of the learning rate hyperparameter $\eta$.

# RMSProp

- 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:

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

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

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

# Adam

- *Adam* (*adaptive moment estimation*), combines ideas of momentum optimization and RMSProp:
  - like momentum optimization, it keeps track of an exponentially decaying average of past gradients
  - 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. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

- Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter $\eta$.

# Variants of Adam

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

- AdaMax: Adam scales down the parameter updates by the $\ell_2$ norm of the time-decayed gradients ($\sqrt{\mathbf{s}}$).
  - AdaMax replaces the $\ell_2$ norm with the $\ell_\infty$ norm.
- Nadam is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam.
- AdamW is a variant of Adam that integrates a regularization technique called *weight decay*.
  - Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99.

# Adaptive Optimization Methods

- Adaptive optimization methods (including RMSProp, Adam and its variants) are often great, converging fast to a good solution.
  - A 2017 paper by Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets.
  - If you are disappointed by your model's performance, try using NAG instead: your dataset may just be allergic to adaptive gradients.
- The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*)
  - Unfortunately, these algorithms are very hard to apply to deep neural networks because there are $n^2$ Hessians per output (where $n$ is the number of parameters).

# Training Sparse Models

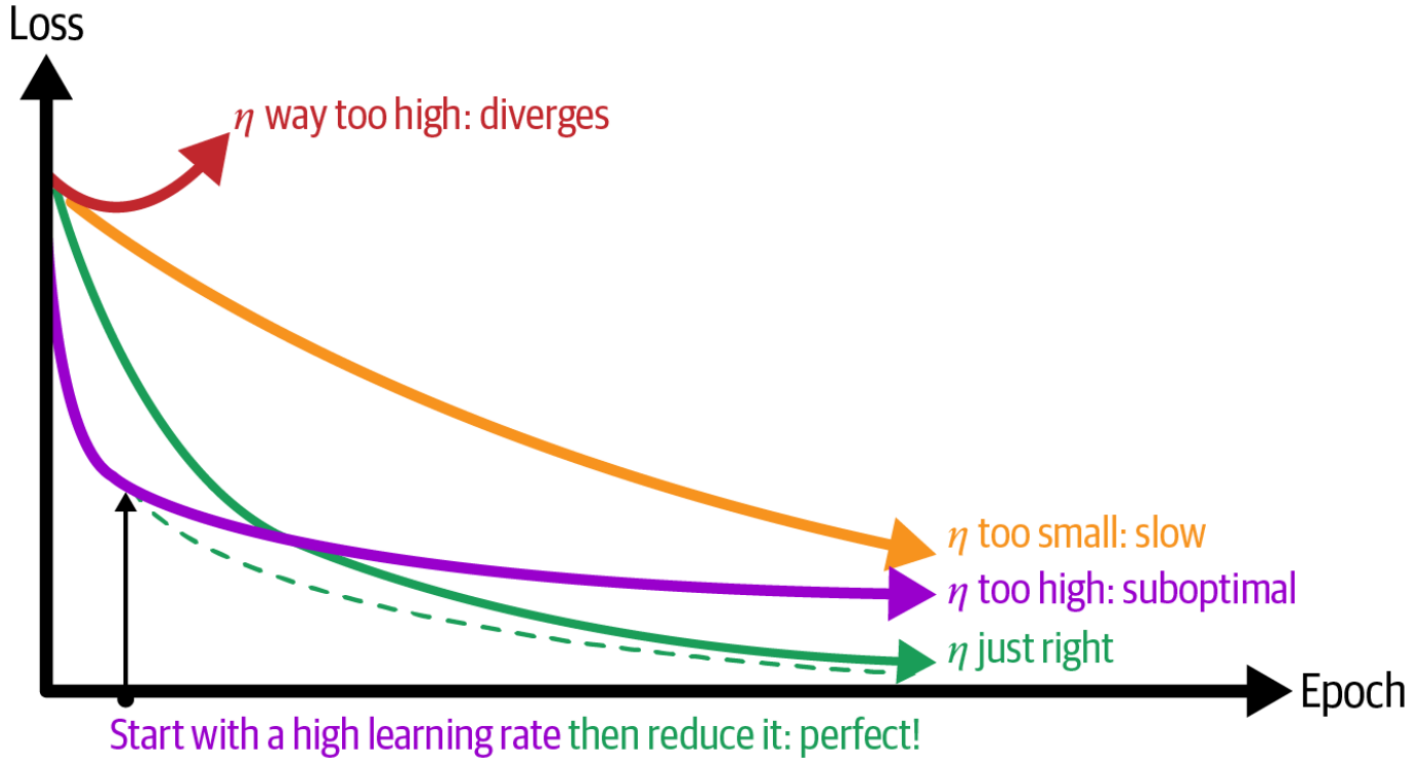
- All the optimization algorithms we just discussed produce dense models, meaning that most parameters will be nonzero.
  - If you need a fast model at runtime, or if you need to take up less memory, you may prefer a sparse model instead.
- One way to achieve sparsity is to train the model as usual, then get rid of the tiny weights (set them to zero).
  - This will typically not lead to a very sparse model, and it may degrade the model's performance.
- A better option is to apply strong $\ell_1$ regularization during training.
- If these techniques remain insufficient, use TF-MOT (TensorFlow Model Optimization Toolkit), which provides a pruning API capable of iteratively removing connections during training based on their magnitude.

# Optimizer Comparison

| Class | Convergence speed | Convergence quality |
|---|---|---|
| `SGD` | * | *** |
| `SGD(momentum=...)` | ** | *** |
| `SGD(momentum=..., nesterov=True)` | ** | *** |
| `Adagrad` | *** | * (stops too early) |
| `RMSprop` | *** | ** or *** |
| `Adam` | *** | ** or *** |
| `AdaMax` | *** | ** or *** |
| `Nadam` | *** | ** or *** |
| `AdamW` | *** | ** or *** |

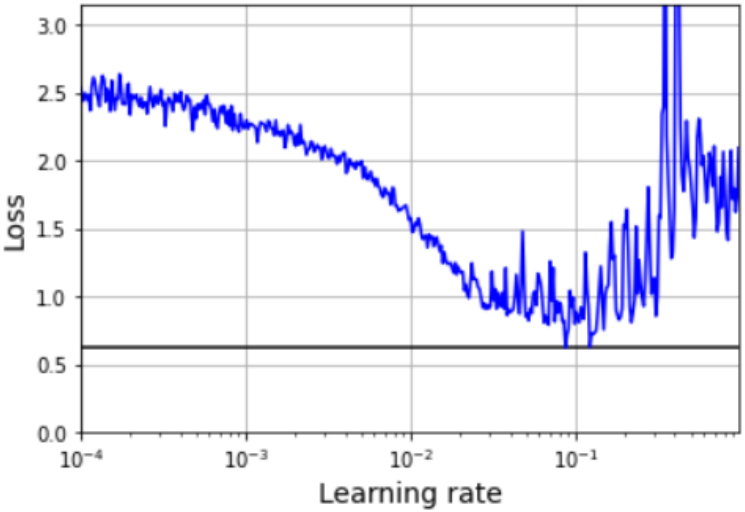
# 4.
# Learning Rate Scheduling

# Finding a good learning rate



# Learning Schedule

- Learning rate tuning: train the model for a few hundred iterations with different rates that are exponentially increasing.
  - You can then reinitialize your model and train it with that learning rate.
- Doing better tha ith a large learning rate and then re fast progress, you can reach a good
  - Or start with a p it again.
- *Learning schedu* arning rate during training.

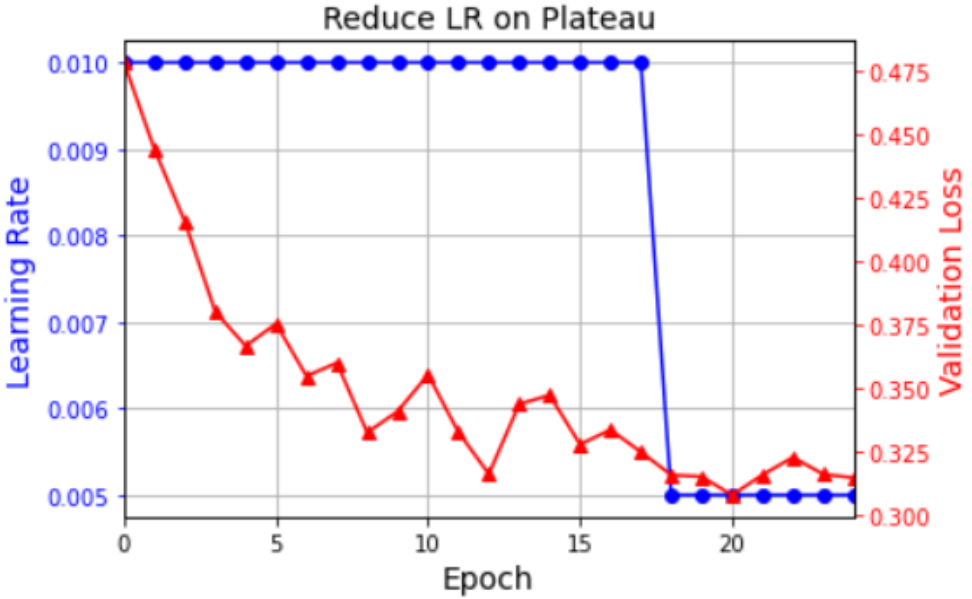


# Learning Schedule

- *Power scheduling:* Set the learning rate to a function of the iteration number $t$: $\eta(t) = \eta_0/(1 + t/s)^c$.
  - The initial learning rate $\eta_0$, the power $c$ (typically set to 1), and the steps $s$ are hyperparameters.
- *Exponential scheduling:* Set the learning rate to $\eta(t) = \eta_0\ (0.1)^{t/s}$.
  - While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every $s$ steps.
- *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 (e.g., $\eta_1$ = 0.001 for 50 epochs), and so on.
  - This solution requires fiddling around to figure out the right sequence of learning rates and number of epochs.

# Learning Schedule

- *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.




- *1cycle sche* …ining rate $\eta_0$, growing line…
  - Then it … gain during the second h…
  - Keras do…
- It is shown t… …g, and 1cycle can consider…

# 5.
# Regularization

# Regularization

- Deep neural networks typically have tens of thousands of parameters, sometimes even millions.
  - This gives them high degree of freedom and means they can fit a huge variety of complex datasets.
  - This great flexibility makes them prone to overfitting the training set.
- Regularization is often needed to prevent this.
  - We used early stopping and batch normalization for regularization.
- There are other techniques:
  - $\ell_1$ and $\ell_2$ regularization
  - Dropout
  - Max-norm regularization.

# $\ell_1$ and $\ell_2$ regularization

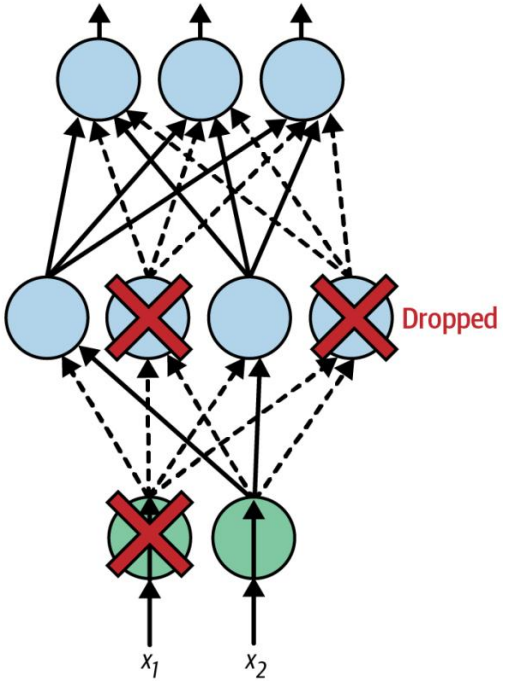
- Use $\ell_2$ regularization to constrain a neural network's connection weights, and/or $\ell_1$ regularization if you want a sparse model.

```
layer = tf.keras.layers.Dense(100, activation="relu",
                              kernel_initializer="he_normal",
                              kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

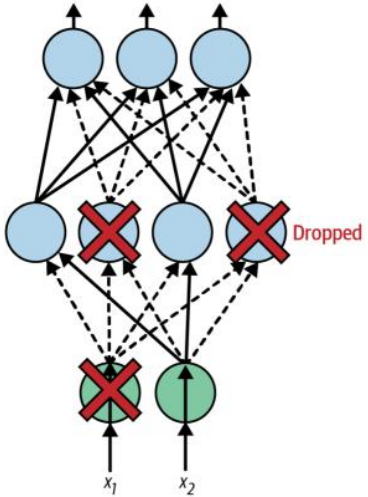
- The `l2()` function returns a regularizer that will be called at each step during training to compute the regularization loss.
- $\ell_2$ regularization is fine when using SGD, momentum optimization, and Nesterov momentum optimization, but not with Adam and its variants.
  - If you want to apply regularization to Adam, use AdamW instead of $\ell_2$ regularization.

# Dropout

- *Dropout* is one of the most popular regularization techniques for deep neural networks and it has proven to be highly successful.
- 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.



# Dropout




- The hyperparameter $p$ is called the *dropout rate*, and it is typically set between 10% and 50%.
  - 20%–30% in recurrent neural nets
  - 40%–50% in convolutional neural networks
- After training, neurons don't get dropped anymore.
- 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

- Suppose $p = 75\%$: on average only 25% of all neurons are active at each step during training, so after training, a neuron would be connected to four times as many input neurons as it would be during training.
  - To compensate for this fact, we need to multiply each neuron's input connection weights by four during training.
  - If we don't, the neural network will not perform well as it will see different data during and after training.
- More generally, we need to divide the connection weights by the *keep probability* $(1-p)$ during training.
- In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).

# Dropout in Keras

- Implement dropout using Keras' `tf.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.

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

# Max-Norm Regularization

- *Max-norm regularization*: for each neuron, it constrains the weights $\boldsymbol{w}$ of the incoming connections such that $\|\boldsymbol{w}\|_2 \leq r$, where $r$ is the max-norm hyperparameter and $\|.\|_2$ is the $\ell_2$ norm.
- Max-norm regularization does not add a regularization loss term to the overall loss function.
  - It is typically implemented by computing $\|\boldsymbol{w}\|_2$ after each training step and rescaling $\boldsymbol{w}$ if needed ($\boldsymbol{w} \leftarrow \boldsymbol{w}r/\|\boldsymbol{w}\|_2$).
  - Reducing $r$ increases the amount of regularization and helps reduce overfitting.

```
dense = tf.keras.layers.Dense(
    100, activation="relu", kernel_initializer="he_normal",
    kernel_constraint=tf.keras.constraints.max_norm(1.))
```