THE UNIVERSITY OF CHICAGO

Machine Learning and Predictive Analytics, MSCA-31009-4

Lecture 6

Igor Yakushin

May 4, 2021

Overview I

1. Training Deep Neural Networks
   - The Vanishing/Exploding Gradients Problems
     - Weight initialization
     - Nonsaturating Activation Functions
     - Batch Normalization
     - Gradient Clipping
   - Reusing Pretrained Layers
   - Unsupervised Pretraining
   - Pretraining on an Auxiliary Task
   - Self-supervised learning
   - Faster Optimizers
     - Momentum Optimization
     - Nesterov Accelerated Gradient
     - AdaGrad
     - RMSProp
     - Adam and Nadam
   - Learning Rate Scheduling
   - Avoiding Overfitting Through Regularization

Overview II

- l_1, l_2 regularization
- Dropout
- Monte Carlo (MC) Dropout
- Max-Norm Regularization

Training Deep Neural Networks

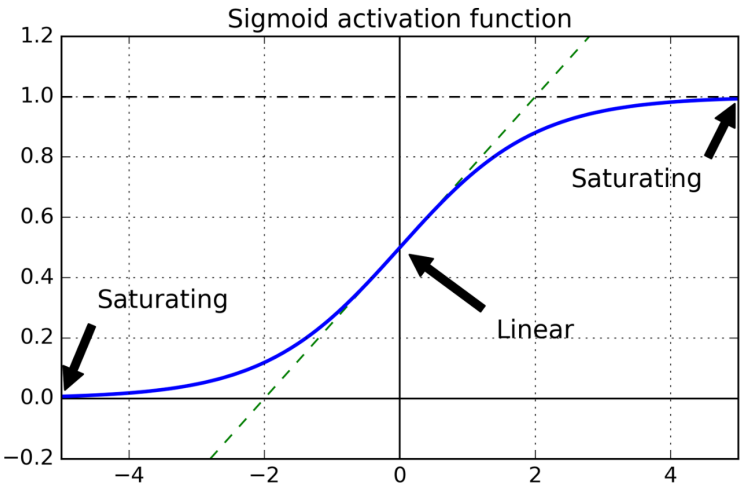
Training Deep Neural Networks

- So far we have been training networks with just 1-3 hidden layers
- Training deeper networks consisting of 10s or 100s layers runs into the following problems:
  - Vanishing gradient problem - gradients get smaller and smaller from layer to layer
  - Exploding gradient problem - gradients get larger and larger
  - Training might be extremely slow
  - You might not have enough data to train such a big network
  - The model with millions parameters would severely overfit

The Vanishing/Exploding Gradients Problems

- The backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient along the way.
- Once the gradients are computed, all the weights of the model are adjusted
- Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, Gradient Descent leaves the weights on the lower layers practically unchanged and training never converges to a good solution - vanishing gradients problem
- 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 - exploding gradients problem

The Vanishing/Exploding Gradients Problems



- One reason for vanishing gradients is using a sigmoid-like activation function in the hidden layers
- When inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0
- Using something like ReLU, helps to solve this issue.
- Another issue is weights initialization. It was shown that simple random initialization from a normal distribution contributes to the problem of vanishing/exploding gradients
- Several initialization procedures were demonstrated to help

Weight initialization

Glorot or Xavier initialization:

- fan_{in} - number of inputs of the layer
- fan_{out} - number of outputs of the layer
- $fan_{avg} = \frac{fan_{in}+fan_{out}}{2}$
- The weights of a layer are initialized either from a normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$ or from a uniform distribution between $-r$ and $+r$, where $r = \sqrt{\frac{3}{fan_{avg}}}$

LeCun initialization, suggested 10 years earlier, is very similar to Glorot but is using fan_{in} instead of fan_{avg}: the weights for a layer are taken from a normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{in}}$

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

- Different authors also originally proposed different activation functions

Weight 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")
```

- Cells 4-6, l6_n1.ipynb

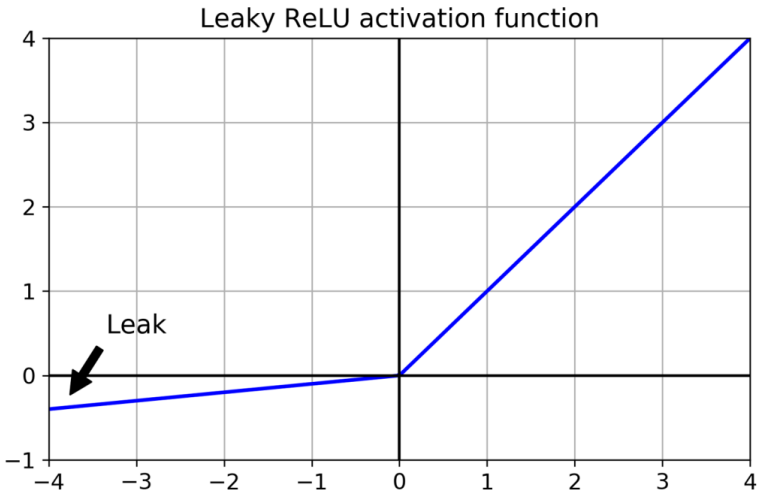
Nonsaturating Activation Functions

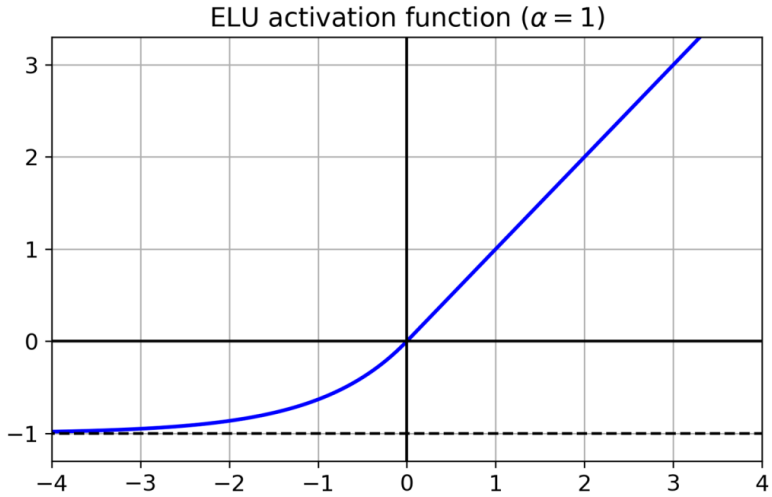
- ReLU helps to avoid saturation and is one of the most popular activation functions but it has its own problems
- It suffers from a problem known as the dying ReLUs: during training, some neurons effectively "die": they stop outputting anything other than 0.
- A neuron dies when its weights get tweaked 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
- Several variants of ReLU were proposed to fight with this problem

Nonsaturating Activation Functions

- LeakyReLU: $LeakyReLU_\alpha(z) = \max(\alpha z, z)$
- Randomized leaky ReLU (RReLU): α is picked randomly in a given range during training and is fixed to an average value during testing
- Parametric leaky ReLU (PReLU): α is a parameter to be learned during training
- Exponential linear unit (ELU):

$$ELU_\alpha = \begin{cases} \alpha(\exp(z) - 1), & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases} \tag{1}$$





Nonsaturating Activation Functions

- Scaled ELU (SELU):

$$ELU_\alpha = \lambda \begin{cases} \alpha(\exp(z) - 1), & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases} \tag{2}$$

- 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.
- So, which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general SELU $>$ ELU $>$ leaky ReLU (and its variants) $>$ ReLU $>$ tanh $>$ logistic.

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 PReLU, replace LeakyRelu(alpha=0.2) with PReLU() .
- There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own
- 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")
```

- Cells 7-34, l6_n1.ipynb

Batch Normalization

- Although using He (or Glorot) 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 (BN) is another approach to vanishing/exploding gradients
- BN is added either before or after applying activation function.
- BN 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. 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

Batch Normalization

- 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")

1. $\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$ - a vector of input means evaluated over the whole mini-batch (one mean per input), m_B - mini-batch size
2. $\boldsymbol{\sigma}_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\boldsymbol{\mu}_B - \mathbf{x}^{(i)})^2$ - a vector of mean stds
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$ - a vector of zero-centered and normalized inputs for instance i; ϵ - a small number to avoid division by 0
4. $\boldsymbol{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}$ - rescaled and shifted version of inputs; $\boldsymbol{\gamma}$ - output scale parameter vector for the layer; $\otimes$ - elementwise multiplication (each input is multiplied by the corresponding scale factor); $\boldsymbol{\beta}$ - shift parameter vector for the layer

Batch Normalization

- How does it work for predictions? There are no mini-batches there.
- During training, BN collects statistics to compute mean and std on the whole dataset. Those means and std are used during predictions.
- To sum up, four parameter vectors are learned in each batch-normalized layer:
  - γ - the output scale vector - and β - the output offset vector - are learned through regular backpropagation,
  - μ - the final input mean vector - and σ - the final input standard deviation vector - are estimated using an exponential moving average.
- Note that μ and σ are estimated during training, but they are used only after training to replace the batch input means and standard deviations
- The experiments show that BN helps with vanishing/exploding gradient problem, makes training less sensitive to weight initialization, typically improves the convergence and also acts as a regularization
- The downside is: it requires more computations but it might be compensated by faster convergence

Batch Normalization

In Keras:

```
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

If you want to have BN before the activation function, add activation funciton as a layer:

```
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")
])
```

Since BN trains an offset β, no need to use bias

Batch Normalization

BN's hyperparameters:

- momentum is used to compute moving average of the vector $\boldsymbol{v}$ (mean and std):

$$\hat{\boldsymbol{v}} \leftarrow \hat{\boldsymbol{v}} \times momentum + \boldsymbol{v} \times (1 - momentum) \tag{3}$$

  here $\hat{\boldsymbol{v}}$ - moving average, $\boldsymbol{v}$ - value for the current batch. Good values are close to 1: 0.9, 0.99, etc.
- axis - which axis should be normalized. Default: -1. That means that the statistics are computed over all the axes except for the last one.
  - When the input batch is 2D (i.e., the batch shape is [batch size, features]), mean and std are computed over batch instances
  - When the input batch is 3D - [batch size, height, width], by default statistics will be computed over batch instances and rows. If you want to compute statistics only over batch instances, axis=[1,2]
- Cells 35-42, l6_n1.ipynb

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 - gradient clipping
- This technique is most often used in recurrent neural networks, as Batch Normalization is tricky to use in RNNs
- In Keras, implementing Gradient Clipping is just a matter of setting the clipvalue or clipnorm argument when creating an optimizer, like this:

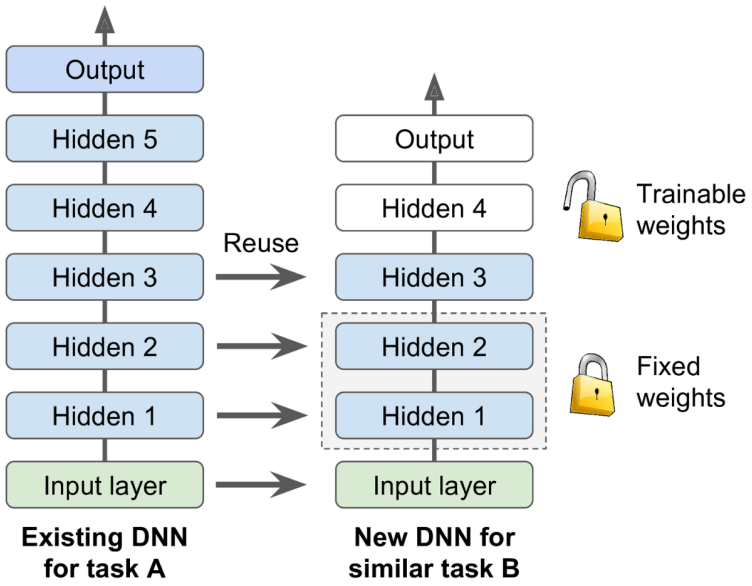
```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- This optimizer will clip every component of the gradient vector to a value between -1.0 and 1.0
- However, that might significantly change the orientation of the gradient
- If you want to keep the orientation and clip the length, use clipnorm instead. Cells 43-44, l6_n1.ipynb.

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 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
- 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.

Reusing Pretrained Layers



- 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.
- The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, try keeping all the hidden layers and just replacing the output layer.
- Try freezing all the reused layers first (i.e., make their weights non-trainable 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.

Reusing Pretrained Layers

- 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.
- Cells 45-65, l6_n1.ipynb

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.
- You may still be able to perform unsupervised pretraining
- If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder or a generative adversarial network
- 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 finetune the final network using supervised learning. We shall talk about that in more details in a few weeks.

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.
- 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. 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

Self-supervised learning

- Self-supervised learning is when you automatically generate the labels from the data itself, then you train a model on the resulting "labeled" dataset using supervised learning techniques. Since this approach requires no human labeling whatsoever, it is best classified as a form of unsupervised learning.
- For some tasks labeling data automatically is easy. Several years ago I worked on a gravitational lensing project. Solving the forward problem was easy: there is a simulator that, given the mass distribution, star location, can compute how the observations would look like on Earth. The inverse problem is hard: given the observation, figure out if there is lensing in the observation, where is the true location of the corresponding star, what mass distribution caused lensing. So one can simply generate randomly a lot of initial conditions for the forward simulation, run those, obtain the corresponding labels and then train ML model.

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
  - 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: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam and Nadam optimization.
- Cells 66-72, l6_n1.ipynb

Momentum Optimization

- When a ball rolls down the hill, where it goes at the next moment is determined not only by the local gradient (that would only be true if its velocity were 0) but also but how fast it goes, by the previous momentum that it already has
- The faster the ball goes, the more its next step will be defined by the previous momentum rather than by the local gradient only.

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{m} \end{aligned} \tag{4}$$

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

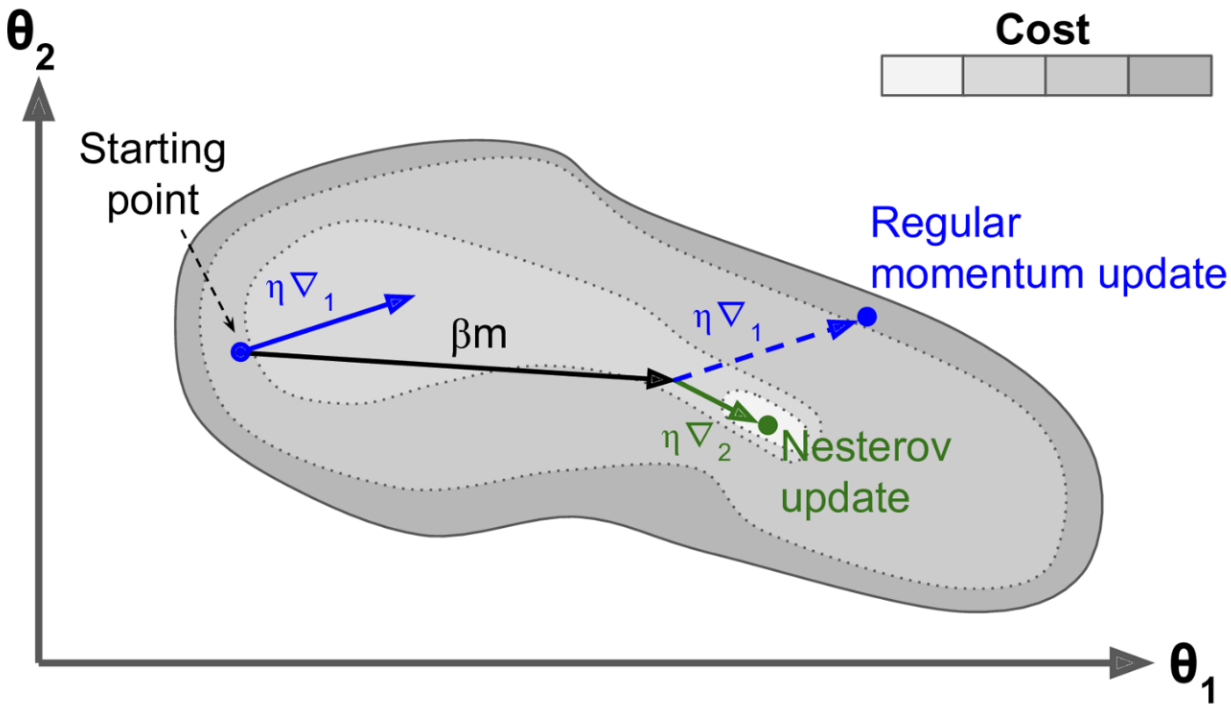
Nesterov Accelerated Gradient

- One small variant to momentum optimization 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 momentum $\boldsymbol{\theta} + \beta \boldsymbol{m}$

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{m} \end{aligned} \tag{5}$$

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

Nesterov Accelerated Gradient



AdaGrad

- Consider 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 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

$$\begin{aligned} \boldsymbol{s} &\leftarrow \boldsymbol{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \epsilon} \end{aligned} \tag{6}$$

- $\otimes$, $\oslash$ - element-wise multiplication/division
- $\boldsymbol{s}$ accumulates square of the gradient

AdaGrad

- AdaGrad 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.
- One additional benefit is that it requires much less tuning of the learning rate hyperparameter
- AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks
- AdaGrad is useful for understanding adaptive learning rate but in practice there are better 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 16 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

$$\begin{aligned} \boldsymbol{s} &\leftarrow \beta \boldsymbol{s} + (1-\beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \epsilon} \end{aligned} \tag{7}$$

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

- rho corresponds to β
- RMSProp almost always outperforms AdaGrad

Adam and Nadam

- Adam - 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;
  - just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients

$$
\begin{aligned}
\boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
\boldsymbol{s} &\leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
\hat{\boldsymbol{m}} &\leftarrow \frac{\boldsymbol{m}}{1 - \beta_1^t} \\
\hat{\boldsymbol{s}} &\leftarrow \frac{\boldsymbol{s}}{1 - \beta_2^t} \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \eta \hat{\boldsymbol{m}} \oslash \sqrt{\boldsymbol{s} + \epsilon}
\end{aligned}
\tag{8}
$$

Adam and Nadam

- t - iteration number (starting at 1).
- m and s are initialized at 0 and therefore will be biased toward 0 at the beginning of training, rescaling them helps to boost m and s at the beginning of training

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

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

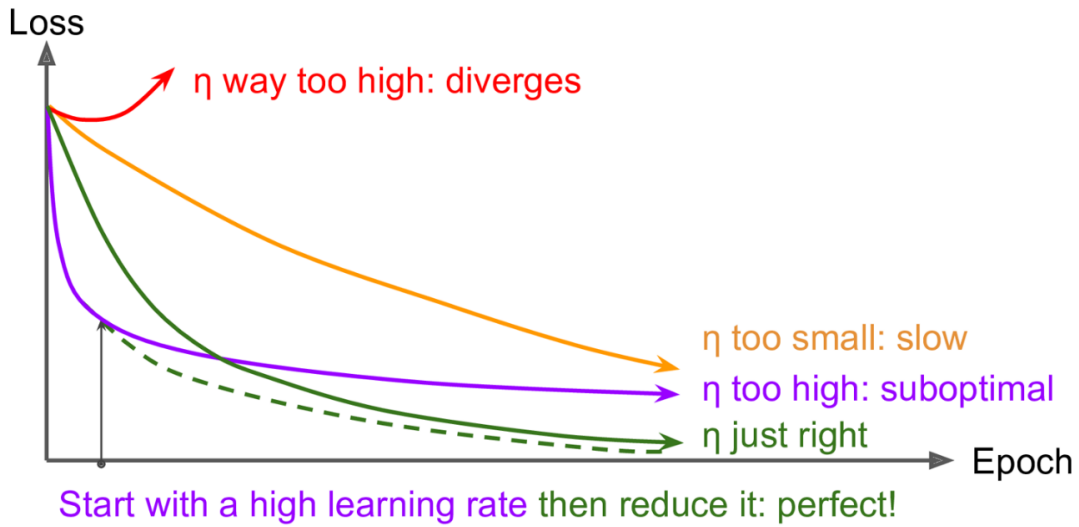
Adam and Nadam

- While Adam scales down the parameter updates by the l_2 norm of the time-decayed gradients. AdaMax uses $\max(\beta_2 \boldsymbol{s}, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))$ instead
- This can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better
- Nadam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam.

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

Learning Rate Scheduling

- Finding a good learning rate is very important.
- If you set it much too high, training may 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 really settling down



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.
- 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.
- The strategies to change learning rate during the training are called learning schedules

Learning Rate Scheduling

- Power scheduling: $\eta(t) = \frac{\eta_0}{(1+\frac{t}{s})^c}$. Here η_0, s, c are hyperparameters. Typically $c = 1$. This schedule first drops quickly, then more and more slowly.
- Exponential scheduling: $\eta(t) = \eta_0 0.1^{\frac{t}{s}}$. η will gradually drop by a factor of 10 every s steps.
- Piecewise constant scheduling: Use a constant η for a number of epoch, then - a smaller one for a number of epochs, etc.
- Performance scheduling: Measure the validation error every N steps, and reduce the learning rate by a factor of λ when the error stops dropping
- 1cycle scheduling: Starts by increasing the initial learning rate η_0 , growing linearly up to η_1 halfway through training. Then it decreases η linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude. η_1 is chosen the same way as for the optimal constant rate and η_0 is 10 times lower.

Learning Rate Scheduling

- Power scheduling in Keras:

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

  Here $decay = \frac{1}{s}$, $c = 1$
- 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

- 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

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

- For performance scheduling, pass the ReduceLROnPlateau callback to the fit():

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

- For 1cycle: just create a custom callback that modifies the learning rate at each iteration. You can update the optimizer's learning rate by changing self.model.optimizer.lr

---

- To sum up: exponential decay, performance scheduling, and 1cycle can considerably speed up convergence
- Cells 73-99, l6_n1.ipynb

Avoiding Overfitting Through Regularization

- 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 have already seen several regularization techniques that can be used with neural networks: Early Stopping, Batch Normalization
- Now we shall consider ℓ_1, ℓ_2, Dropout, and Max-Norm regularization.

ℓ_1, ℓ_2 regularization

- Just like we used ℓ_1 and ℓ_2 regularization for simple linear models, we can use them for neural networks.
- Those regularizations are applied to the weights of each layer separately:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=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. This is then added to the final loss.
- Similarly one can use l1() or l1_l2()
- Since you will typically want to apply the same regularizer to all layers in your network, as well as using the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments
- You can simplify this process with partial wrapper that allows you to set the default arguments

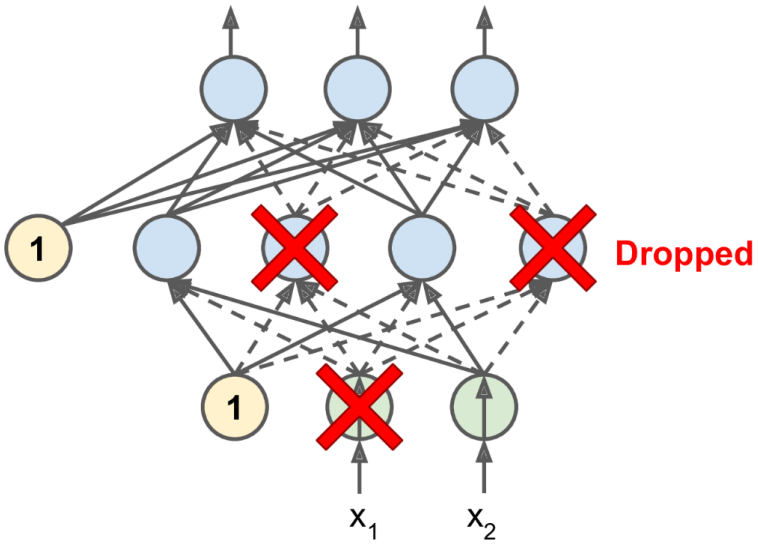
ℓ_1, ℓ_2 regularization

```
from functools import partial
RegularizedDense =
    partial(keras.layers.Dense,
            activation="elu",
            kernel_initializer="he_normal",
            kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
          kernel_initializer="glorot_uniform")
])
```

Dropout

- Dropout is one of the most popular regularization techniques for deep neural networks.
- 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

- 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.
- Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks. 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.
- In practice, you can usually apply dropout only to the neurons in the top one to three layers, excluding the output layer

```
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.
- If you want to regularize a self-normalizing network based on the SELU activation function, you should use alpha dropout: this is a variant of dropout that preserves the mean and standard deviation of its inputs

Dropout

- 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

Monte Carlo (MC) Dropout

- MC Dropout can boost the performance of any trained dropout model without having to retrain it or even modify it at all, provides a much better measure of the model's uncertainty, and is also amazingly simple to implement.

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

- We just make 100 predictions over the test set, setting training=True to ensure that the Dropout layer is active, and stack the predictions. Since dropout is active, all the predictions will be different and one can average over them and compute std
- Cells 109-123, l6_n1.ipynb

Monte Carlo (MC) Dropout

- If your model contains other layers that behave in a special way during training (such as BatchNormalization layers), then you should not force training mode like we just did. Instead, you should replace the Dropout layers with the following MCDropout class, to enable training only in dropout layers:

```
class MCDropout(keras.layers.Dropout):
  def call(self, inputs):
    return super().call(inputs, training=True)
```

- If you are creating a model from scratch, it's just a matter of using MCDropout rather than Dropout .
- But if you have a model that was already trained using Dropout, you need to create a new model that's identical to the existing model except that it replaces the Dropout layers with MCDropout, then copy the existing model's weights to your new model.

Max-Norm Regularization

- Another regularization technique that is popular for neural networks is called max-norm regularization: for each neuron, it constrains the weights $\boldsymbol{w}$ of the incoming connections such that $\|\boldsymbol{w}\|_2 < r$, where r is the max-norm hyperparameter
- Cells 100-126, l6_n1.ipynb