Module 2

Training Deep Neural Networks

Introduction

When training a deep neural network, here are some of the problems you could run into:

- You may be faced with the problem of gradients growing ever smaller or 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.

In this module we will go through each of these problems and present techniques to solve them.

The Vanishing/Exploding Gradients Problem

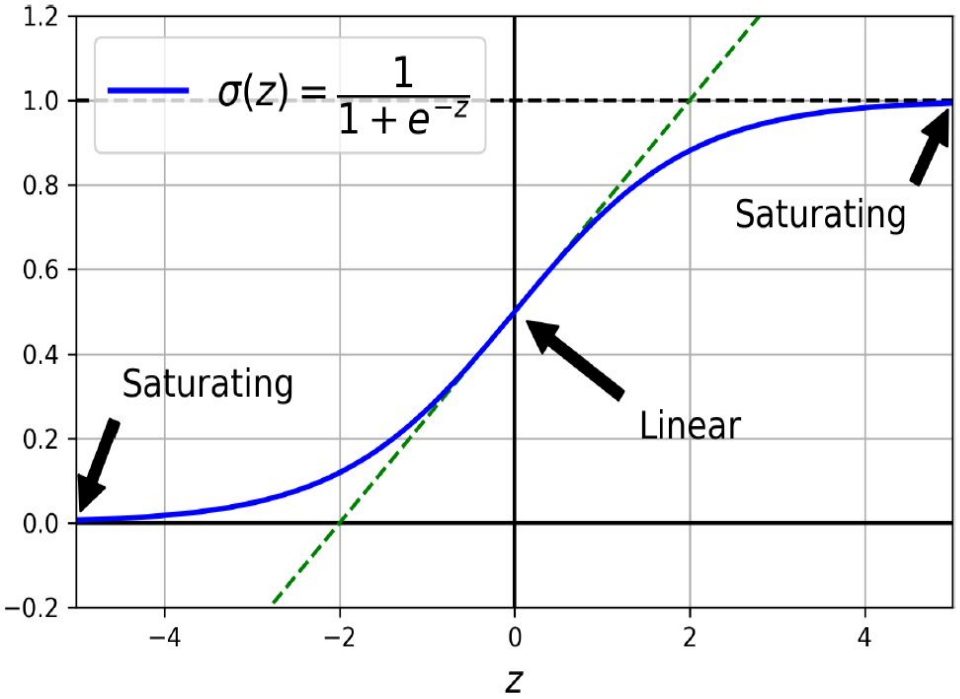
The Vanishing/Exploding Gradients Problem

The backpropagation algorithm uses the gradients to determine optimal parameters. Unfortunately, gradients often get smaller and smaller and as a result, the gradient descent update leaves the weights virtually unchanged, and training never converges to a good solution. This is called the vanishing gradients problem.

In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the exploding gradients problem, which surfaces most often in recurrent neural networks.

The Vanishing/Exploding Gradients Problem

Looking at the sigmoid 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 (i.e., the curve is flat at both extremes). Thus, backpropagation has no gradient to propagate back through the network.



The Vanishing/Exploding Gradients Problem

- Glorot and He Initialization
- Better Activation Functions
- Batch Normalization
- Gradient Clipping

Glorot Initialization

Glorot and Bengio propose a way to significantly alleviate the unstable gradients problem. They point out that 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.

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**. This initialization strategy is called Xavier initialization or Glorot initialization, after the paper's first author.

Using Glorot initialization can speed up training considerably, and it is one of the practices that led to the success of deep learning.

Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a **poor choice of activation function**. Until then most people have only used sigmoid activation functions.
But it turns out that other activation functions behave much better in deep neural networks - in particular, the **ReLU activation function, mostly because it does not saturate for positive values, and also because it is very fast to compute.**

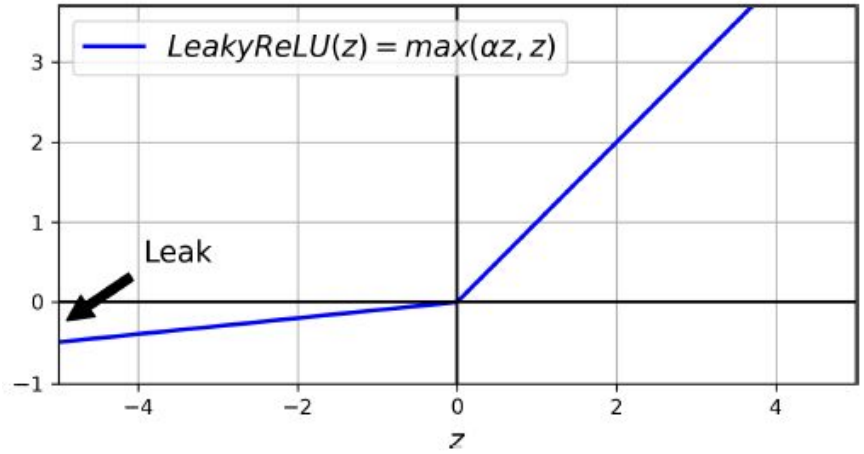
Better Activation Functions

Unfortunately, the ReLU activation function 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. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron’s inputs plus its bias term) is negative for all instances in the training set. To solve this problem, you may want to use a variant of the ReLU function, such as the leaky ReLU.

Leaky ReLU

The leaky ReLU activation function is defined as LeakyReLUα(z) = max(αz, z). The hyperparameter α defines how much the function “leaks”.
Having a slope for z < 0 ensures that leaky ReLUs never die.

In fact, setting α = 0.2 (a huge leak) seemed to result in better performance than α = 0.01 (a small leak).



ReLU Variants

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 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.

ReLU Variants

Keras includes the classes LeakyReLU and PReLU in the tf.keras.layers package.

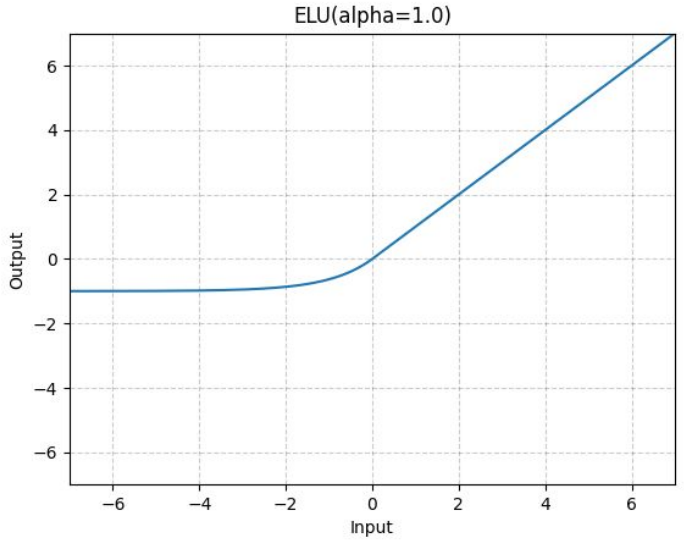
ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their derivatives abruptly change (at z = 0). This sort of discontinuity can slow down convergence.

So we will look at a smooth variant of the ReLU activation function, ELU.

ELU

The ELU activation function looks a lot like the ReLU function, with a few major differences:

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



It takes on negative values when z < 0.

The hyperparameter α is usually set to 1, but you can tweak it like any other hyperparameter. Using ELU with Keras is as easy as setting activation="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).

Activation Function

Which activation function should you use for the hidden layers of your deep neural networks?

Activation Function

Which activation function should you use for the hidden layers of your deep neural networks?

ReLU remains a good default for simple tasks, plus it’s very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations

If you have spare time and computing power, you can use cross-validation to evaluate other activation functions as well.

Batch Normalization

In 2015, Sergey Ioffe and Christian Szegedy proposed a technique called batch normalization (BN). 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.**

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

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

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

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

In this algorithm:

- μB is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
- mB is the number of instances in the mini-batch.
- σB is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- x(i) is the vector of zero-centered and normalized inputs for instance i.
- ε is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically 10–5). This is called a smoothing term.
- γ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- ⊗ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- z(i) is the output of the BN operation. It is a rescaled and shifted version of the inputs.

Batch Normalization

During training, BN standardizes its inputs, then rescales and offsets them.

To sum up, four vectors are involved in each batch-normalized layer:
γ (the output scale vector) and β (the output offset vector) are learned through regular backpropagation, and μ (the final input mean vector) and σ (the final input standard deviation vector) are estimated using an exponential moving average.

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

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

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

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

Batch Normalization

During training, BN standardizes its inputs, then rescales and offsets them.

What about at test time?

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.

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

Batch Normalization

Ioffe and Szegedy demonstrated that batch normalization considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task.

The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the sigmoid activation function.

The networks were also much less sensitive to the weight initialization.

Able to use much larger learning rates, significantly speeding up the learning process.

Acts like a regularizer, reducing the need for other regularization techniques

Batch Normalization

You may find that training is rather slow, because each epoch takes much more time when you use batch normalization.

This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance.

Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold.

In Keras, implementing gradient clipping is done by setting the clipvalue when creating an optimizer:

```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile([...], optimizer=optimizer)
```

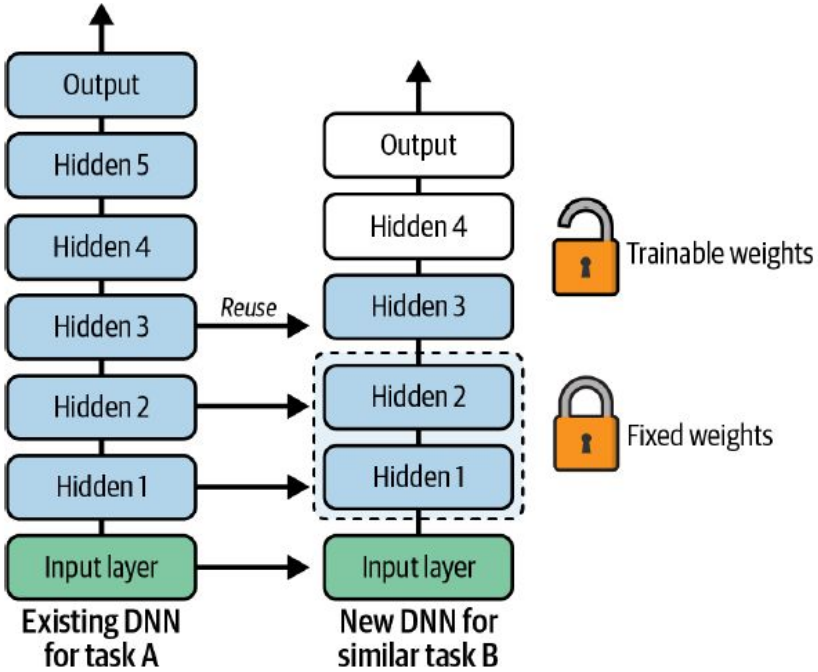
This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0.
The threshold is a hyperparameter you can tune.

Reusing Pretrained Layers

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle.

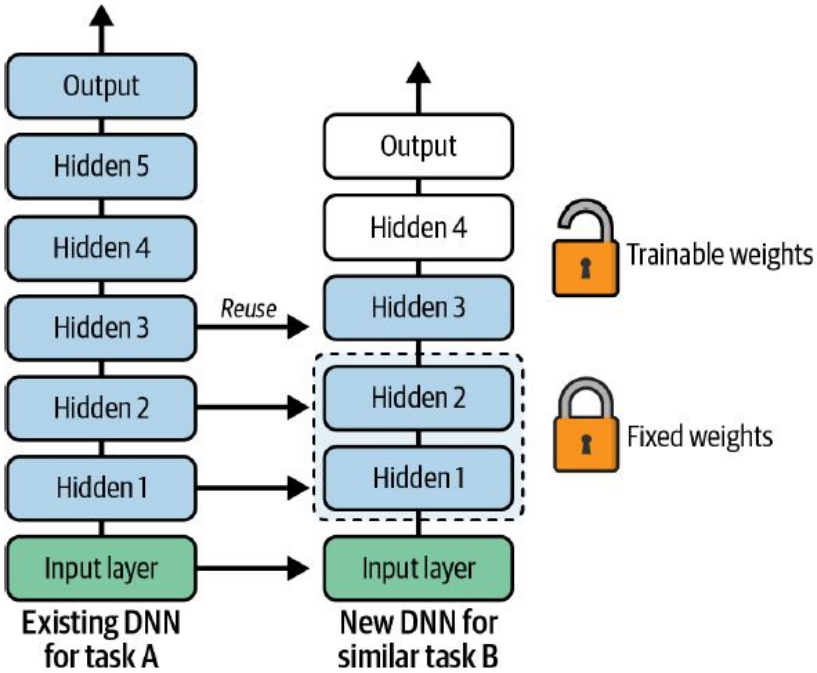
This technique is called transfer learning. It will not only speed up training considerably, but also requires significantly less training data.



Reusing Pretrained Layers

Suppose you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects, and 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.

If the input pictures for your new task don’t have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model.



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 probably will not have the right number of outputs.
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.

The more similar the tasks are, the more layers you will want to reuse (starting with the lower layers).
For very similar tasks, try to keep all the hidden layers and just replace 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 and they will remain fixed), 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.

Unsupervised Pre-Training

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.

First, you should try to gather more labeled training data, but if you can’t, you may still be able to perform unsupervised pre-training.
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, 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 fine-tune the final network using supervised learning (i.e., with the labeled training examples).

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.

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.

Faster Optimizers

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):

1. Applying a good initialization strategy for the connection weights
2. Using a good activation function,
3. Using batch normalization, and
4. Reusing parts of a pretrained network

Another speed boost comes from using a faster optimizer than the regular gradient descent optimizer.

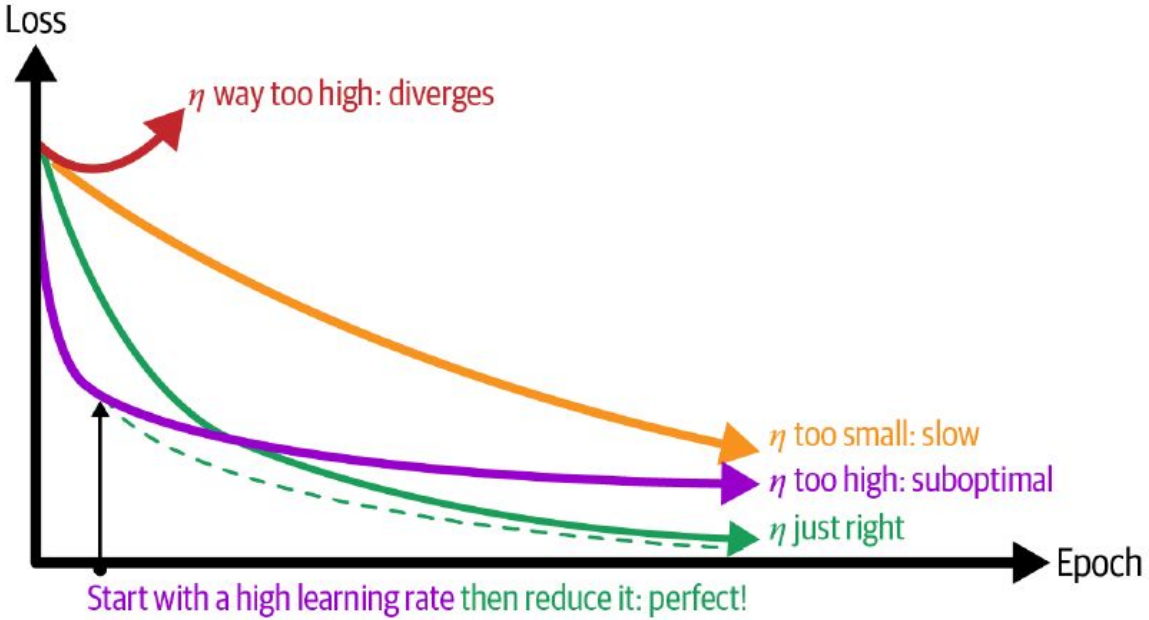
Momentum

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.

This is the core idea behind momentum optimization.

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.



Exponential Scheduling

The learning rate will gradually drop by a factor of 10 every s steps.

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 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.

Avoiding Overfitting Through Regularization

Avoiding Overfitting Through Regularization

Deep neural networks typically have tens of thousands of parameters, sometimes even millions.
This great flexibility makes the network prone to overfitting the training set.

Regularization is often needed to prevent this.

- ℓ1 and ℓ2 regularization
- Dropout

ℓ1 and ℓ2 Regularization

You can use ℓ2 regularization to constrain a neural network's connection weights, and/or ℓ1 regularization if you want a sparse model (with many weights equal to 0).

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

You can just use tf.keras.regularizers.l1() if you want ℓ1 regularization

If you want both ℓ1 and ℓ2 regularization, use tf.keras.regularizers.l1_l2() (specifying both regularization factors).

Dropout

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

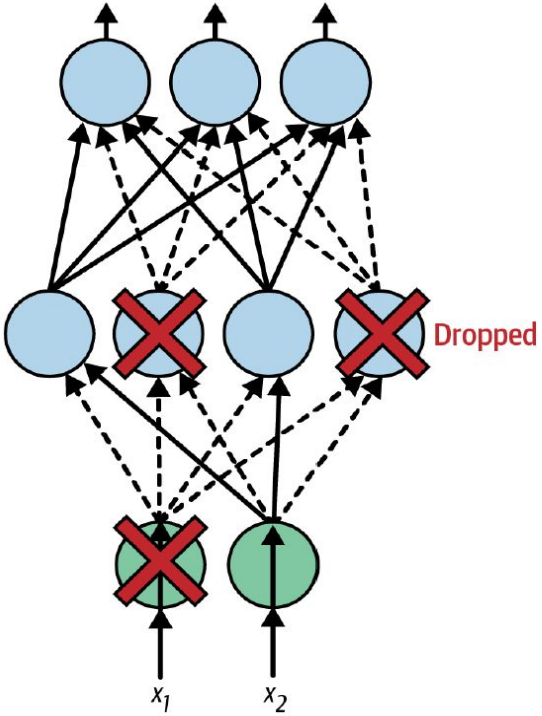
Many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

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. After training, neurons don’t get dropped anymore.

Dropout

At each training iteration, a random subset of neurons in one or more layers - except the output layer - are dropped out. These neurons output 0 at this iteration.



Dropout

Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly.

It is generally well worth the extra time and effort, especially for large models.