CHAPTER 4

OPTIMIZATION

In this chapter, we will present variants of the **Gradient Descent** optimization strategy and show how they can be used to optimize neural network parameters.

Let us start with the basic Gradient Descent algorithm and its limitations.

Algorithm 1 (Gradient Descent)

Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $(x_i, y_i) \in \mathcal{D}$
        1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
        2. Compute gradient $\nabla_\theta \mathcal{L}_i$
    2. Compute overall gradient $\nabla_\theta \mathcal{L} = \frac{1}{n} \sum_i \nabla_\theta \mathcal{L}_i$
    3. Update parameters θ based on $\nabla_\theta \mathcal{L}$

The typical update rule for the parameters θ at iteration t is

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \rho \nabla_\theta \mathcal{L}$$

where ρ is an important hyper-parameter of the method, called the learning rate. Basically, gradient descent updates θ in the direction of steepest decrease of the loss $\mathcal{L}$.

As one can see in the previous algorithm, when performing gradient descent, model parameters are updated once per epoch, which means a full pass over the whole dataset is required before the update can occur. When dealing with large datasets, this is a strong limitation, which motivates the use of stochastic variants.

4.1 Stochastic Gradient Descent (SGD)

The idea behind the Stochastic Gradient Descent algorithm is to get cheap estimates for the quantity

$$\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta) = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{D}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

where $\mathcal{D}$ is the whole training set. To do so, one draws subsets of data, called *minibatches*, and

$$\nabla_\theta \mathcal{L}(\mathcal{B}; m_\theta) = \frac{1}{b} \sum_{(x_i, y_i) \in \mathcal{B}} \nabla_\theta \mathcal{L}(x_i, y_i; m_\theta)$$

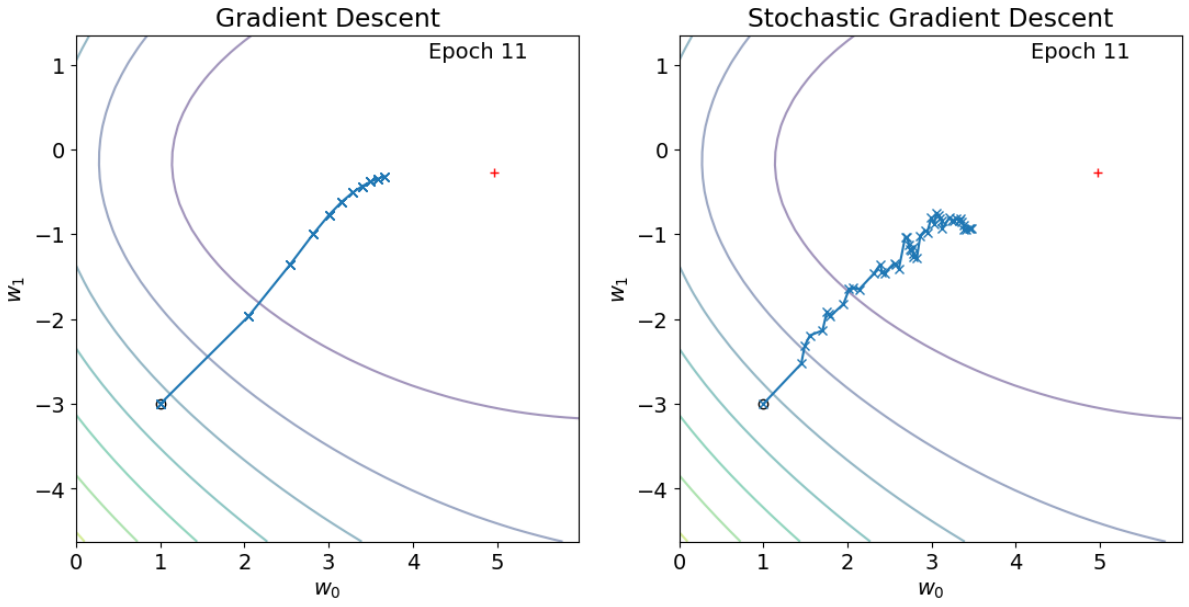
is used as an estimator for $\nabla_\theta \mathcal{L}(\mathcal{D}; m_\theta)$. This results in the following algorithm in which, interestingly, parameter updates occur after each minibatch, which is multiple times per epoch.

Algorithm 2 (Stochastic Gradient Descent)

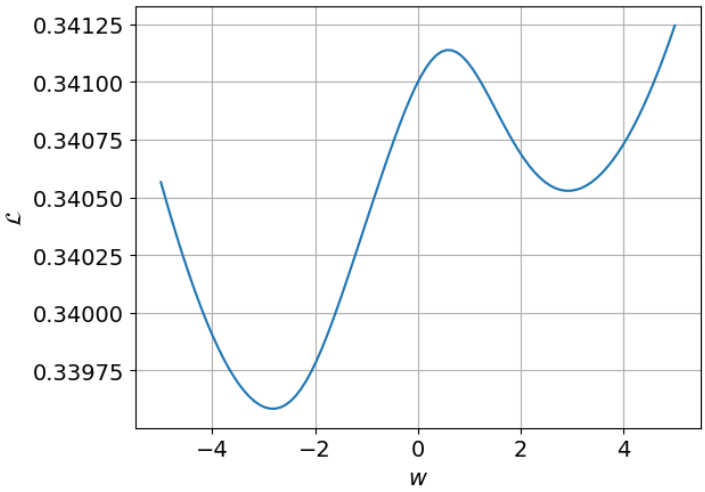
Input: A dataset $\mathcal{D} = (X, y)$

1. Initialize model parameters θ
2. for $e = 1..E$
    1. for $t = 1..n_{\text{minibatches}}$
        1. Draw minibatch $\mathcal{B}$ as a random sample of size b from $\mathcal{D}$
        2. for $(x_i, y_i) \in \mathcal{B}$
            1. Compute prediction $\hat{y}_i = m_\theta(x_i)$
            2. Compute gradient $\nabla_\theta \mathcal{L}_i$
        3. Compute minibatch-level gradient $\nabla_\theta \mathcal{L}_\mathcal{B} = \frac{1}{b} \sum_i \nabla_\theta \mathcal{L}_i$
        4. Update parameters θ based on $\nabla_\theta \mathcal{L}_\mathcal{B}$

As a consequence, when using SGD, parameter updates are more frequent, but they are "noisy" since they are based on an minibatch estimation of the gradient instead of relying on the true gradient, as illustrated below:



Apart from implying more frequent parameter updates, SGD has an extra benefit in terms of optimization, which is key for neural networks. Indeed, as one can see below, contrary to what we had in the Perceptron case, the MSE loss (and the same applies for the logistic loss) is no longer convex in the model parameters as soon as the model has at least one hidden layer:



Gradient Descent is known to suffer from local optima, and such loss landscapes are a serious problem for GD. On the other hand, Stochastic Gradient Descent is likely to benefit from noisy gradient estimations to escape local minima.

4.2 A note on Adam

Adam [Kingma and Ba, 2015] is a variant of the Stochastic Gradient Descent method. It differs in the definition of the steps to be performed at each parameter update.

First, it uses what is called momentum, which basically consists in relying on past gradient updates to smooth out the trajectory in parameter space during optimization. An interactive illustration of momentum can be found in [Goh, 2017].

The resulting plugin replacement for the gradient is:

$$\mathbf{m}^{(t+1)} \leftarrow \frac{1}{1-\beta_1^t}\left[\beta_1 \mathbf{m}^{(t)} + (1-\beta_1)\nabla_\theta \mathcal{L}\right]$$

When β_1 is zero, we have $\mathbf{m}^{(t+1)} = \nabla_\theta \mathcal{L}$ and for $\beta_1 \in]0,1[$, $\mathbf{m}^{(t+1)}$ balances the current gradient estimate with information about past estimates, stored in $\mathbf{m}^{(t)}$.

Another important difference between SGD and the Adam variant consists in using an adaptive learning rate. In other words, instead of using the same learning rate ρ for all model parameters, the learning rate for a given parameter θ_i is defined as:

$$\hat{\rho}^{(t+1)}(\theta_i) = \frac{\rho}{\sqrt{s^{(t+1)}(\theta_i) + \epsilon}}$$

where ϵ is a small constant and

$$s^{(t+1)}(\theta_i) = \frac{1}{1-\beta_2^t}\left[\beta_2 s^{(t)}(\theta_i) + (1-\beta_2)\left(\nabla_{\theta_i}\mathcal{L}\right)^2\right]$$

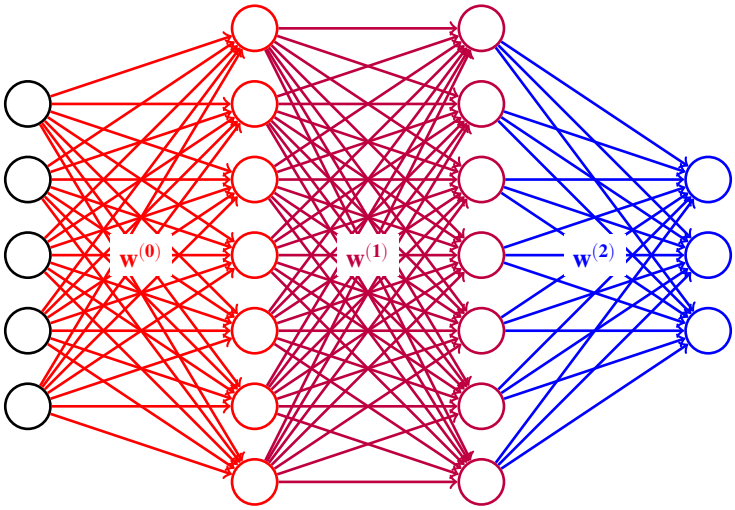
Here also, the s term uses momentum. As a result, the learning rate will be lowered for parameters which have suffered large updates in the past iterations.

Overall, the Adam update rule is:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \hat{\rho}^{(t+1)}(\theta)\mathbf{m}^{(t+1)}$$

4.3 The curse of depth

Let us consider the following neural network:



and let us recall that, at a given layer (ℓ), the layer output is computed as

$$a^{(\ell)} = \varphi(o^{(\ell)}) = \varphi(w^{(\ell-1)}a^{(\ell-1)})$$

where φ is the activation function for the given layer (we ignore the bias terms in this simplified example).

In order to perform (stochastic) gradient descent, gradients of the loss with respect to model parameters need to be computed.

By using the chain rule, these gradients can be expressed as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial w^{(2)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial w^{(1)}} \\
\frac{\partial \mathcal{L}}{\partial w^{(0)}} &= \frac{\partial \mathcal{L}}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial o^{(2)}} \frac{\partial o^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial o^{(1)}} \frac{\partial o^{(1)}}{\partial w^{(0)}}
\end{aligned}$$

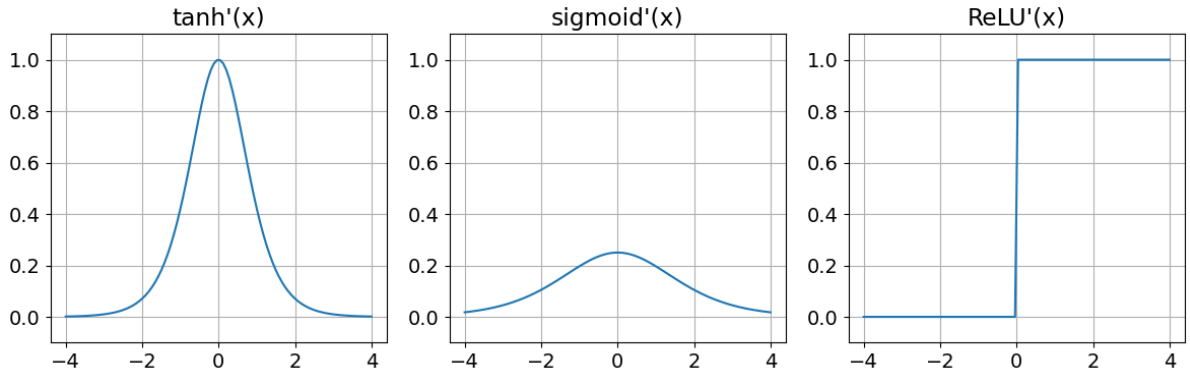
There are important insights to grasp here.

First, one should notice that weights that are further from the output of the model inherit gradient rules made of more terms. As a consequence, when some of these terms get smaller and smaller, there is a higher risk for those weights that their gradients collapse to 0, this is called the **vanishing gradient** effect, which is a very common phenomenon in deep neural networks (*i.e.* those networks made of many layers).

Second, some terms are repeated in these formulas, and in general, terms of the form $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ and $\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}}$ are present in several places. These terms can be further developed as:

$$\begin{aligned}
\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}} &= \varphi'(o^{(\ell)}) \\
\frac{\partial o^{(\ell)}}{\partial a^{(\ell-1)}} &= w^{(\ell-1)}
\end{aligned}$$

Let us inspect what the derivatives of standard activation functions look like:



One can see that the derivative of ReLU has a wider range of input values for which it is non-zero (typically the whole range of positive input values) than its competitors, which makes it a very attractive candidate activation function for deep neural networks, as we have seen that the $\frac{\partial a^{(\ell)}}{\partial o^{(\ell)}}$ term appears repeatedly in chain rule derivations.

4.4 Wrapping things up in `keras`

In `keras`, loss and optimizer information are passed at compile time:

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential

model = Sequential([
    InputLayer(input_shape=(10, )),
    Dense(units=20, activation="relu"),
    Dense(units=3, activation="softmax")
])

model.summary()
```

```
Using TensorFlow backend
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                220

 dense_1 (Dense)             (None, 3)                 63

=================================================================
Total params: 283 (1.11 KB)
Trainable params: 283 (1.11 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(loss="categorical_crossentropy", optimizer="adam")
```

In terms of losses:

- `"mse"` is the mean squared error loss,

- "binary_crossentropy" is the logistic loss for binary classification,
- "categorical_crossentropy" is the logistic loss for multi-class classification.

The optimizers defined in this section are available as "sgd" and "adam". In order to get control over optimizer hyper-parameters, one can alternatively use the following syntax:

```
from keras.optimizers import Adam, SGD

# Not a very good idea to tune beta_1
# and beta_2 parameters in Adam
adam_opt = Adam(learning_rate=0.001,
                beta_1=0.9, beta_2=0.9)

# In order to use SGD with a custom learning rate:
# sgd_opt = SGD(learning_rate=0.001)

model.compile(loss="categorical_crossentropy", optimizer=adam_opt)
```

4.5 Data preprocessing

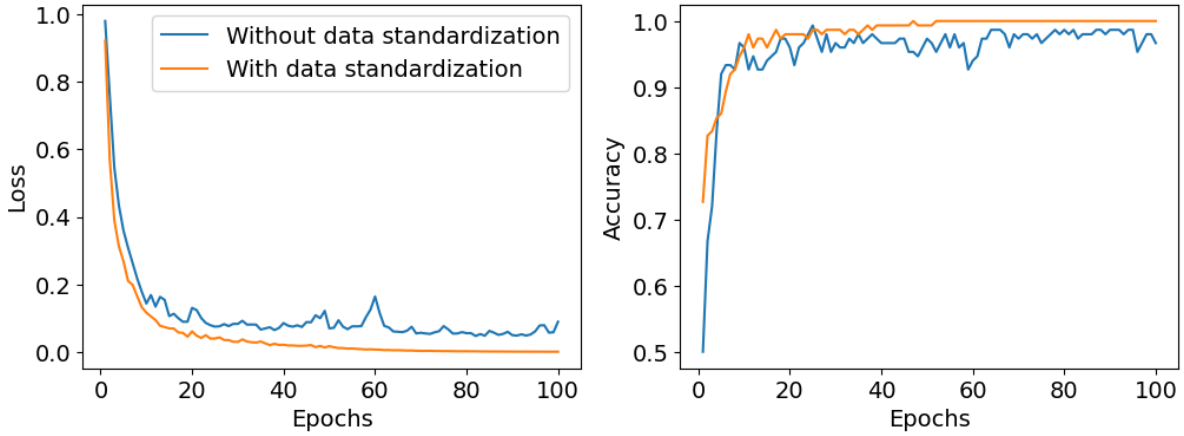
In practice, for the model fitting phase to behave well, it is important to scale the input features. In the following example, we will compare two trainings of the same model, with similar initialization and the only difference between both will be whether input data is center-reduced or left as-is.

```
import pandas as pd
from keras.utils import to_categorical

iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
```

```
from keras.layers import Dense, InputLayer
from keras.models import Sequential
from keras.utils import set_random_seed


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```

Let us now standardize our data and compare performance:

```
X -= X.mean(axis=0)
X /= X.std(axis=0)


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h_standardized = model.fit(X, y, epochs=n_epochs, batch_size=30, verbose=0)
```



CHAPTER 5

REGULARIZATION

As discussed in previous chapters, one of the strengths of the neural networks is that they can approximate any continuous functions when a sufficient number of parameters is used. When using universal approximators in machine learning settings, an important related risk is that of overfitting the training data. More formally, given a training dataset $\mathcal{D}_t$ drawn from an unknown distribution $\mathcal{D}$, model parameters are optimized so as to minimize the empirical risk:

$$\mathcal{R}_e(\theta) = \frac{1}{|\mathcal{D}_t|} \sum_{(x_i, y_i) \in \mathcal{D}_t} \mathcal{L}(x_i, y_i; m_\theta)$$

whereas the real objective is to minimize the "true" risk:

$$\mathcal{R}(\theta) = \mathbb{E}_{x,y \sim \mathcal{D}} \mathcal{L}(x, y; m_\theta)$$

and both objectives do not have the same minimizer.

To avoid this pitfall, one should use regularization techniques, such as the ones presented in the following.

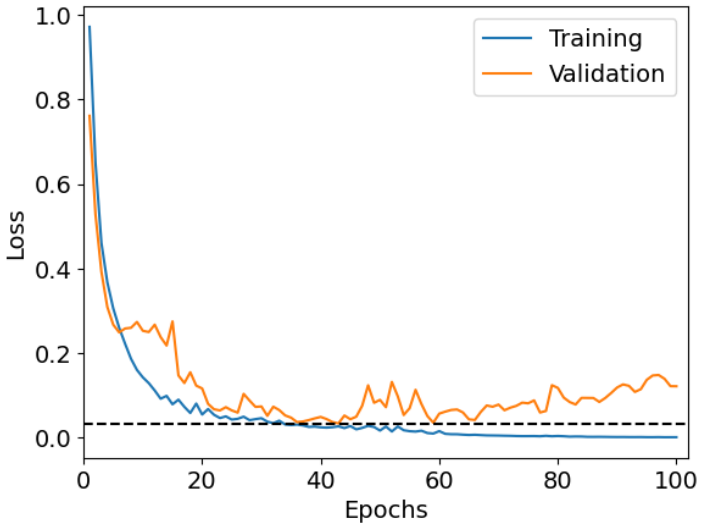
5.1 Early Stopping

As illustrated below, it can be observed that training a neural network for a too large number of epochs can lead to overfitting. Note that here, the true risk is estimated through the use of a validation set that is not seen during training.

```
Using TensorFlow backend
```

```
iris = pd.read_csv("../data/iris.csv", index_col=0)
iris = iris.sample(frac=1)
y = to_categorical(iris["target"])
X = iris.drop(columns=["target"])
X -= X.mean(axis=0)
X /= X.std(axis=0)
```

```
import keras_core as keras
from keras.layers import Dense, InputLayer
from keras.models import Sequential
from keras.utils import set_random_seed


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Here, the best model (in terms of generalization capabilities) seems to be the model at epoch 43. In other words, if we had stopped the learning process after epoch 43, we would have gotten a better model than if we use the model trained during 70 epochs.

This is the whole idea behind the "early stopping" strategy, which consists in stopping the learning process as soon as the validation loss stops improving. As can be seen in the visualization above, however, the validation loss tends to oscillate, and one often waits for several epochs before assuming that the loss is unlikely to improve in the future. The number of epochs to wait is called the *patience* parameter.

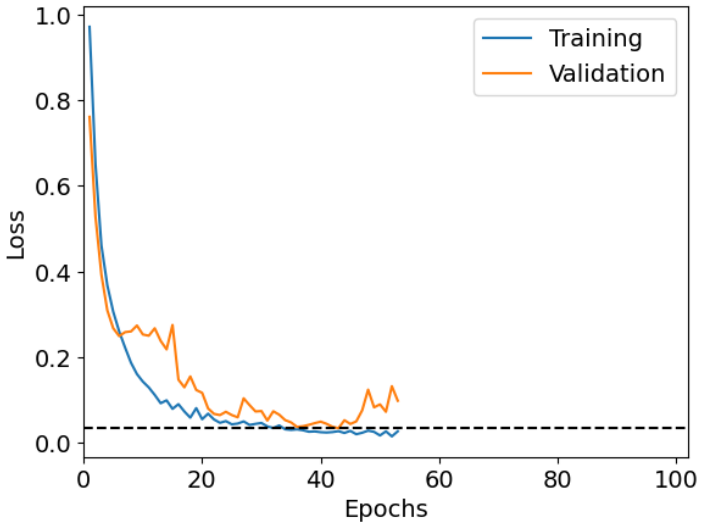
In `keras`, early stopping can be set up via a callback, as in the following example:

```
from keras.callbacks import EarlyStopping


set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu"),
    Dense(units=256, activation="relu"),
```

(continues on next page)

(continued from previous page)

```
    Dense(units=256, activation="relu"),
    Dense(units=3, activation="softmax")
])

cb_es = EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True)

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y,
              validation_split=0.3, epochs=n_epochs, batch_size=30,
              verbose=0, callbacks=[cb_es])
```



And now, even is the model was scheduled to be trained for 70 epochs, training is stopped as soon as it reaches 10 consecutive epochs without improving on the validation loss, and the model parameters are restored as the parameters of the model at epoch 43.

5.2 Loss penalization

Another important way to enforce regularization in neural networks is through loss penalization. A typical instance of this regularization strategy is the L2 regularization. If we denote by $\mathcal{L}_r$ the L2-regularized loss, it can be expressed as:

$$\mathcal{L}_r(\mathcal{D}; m_\theta) = \mathcal{L}(\mathcal{D}; m_\theta) + \lambda \sum_\ell \|\theta^{(\ell)}\|_2^2$$

where $\theta^{(\ell)}$ is the weight matrix of layer ℓ.

This regularization tends to shrink large parameter values during the learning process, which is known to help improve generalization.

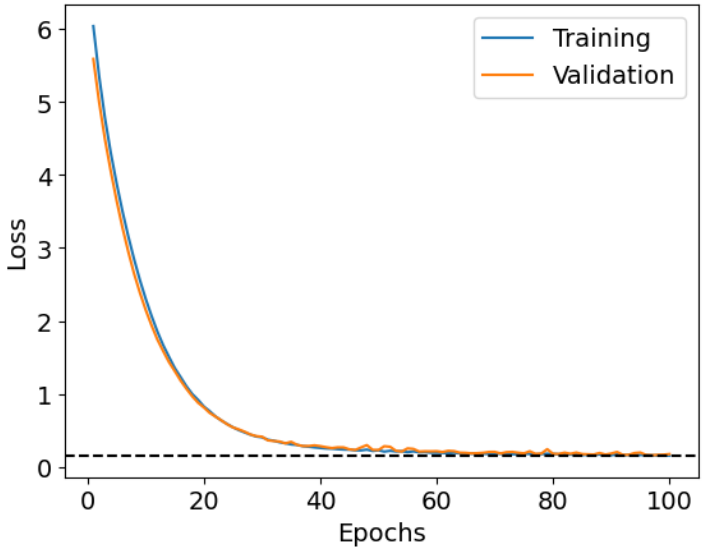
In `keras`, this is implemented as:

```
from keras.regularizers import L2

λ = 0.01
```

(continues on next page)

(continued from previous page)

```
set_random_seed(0)
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=256, activation="relu", kernel_regularizer=L2(λ)),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



5.3 DropOut

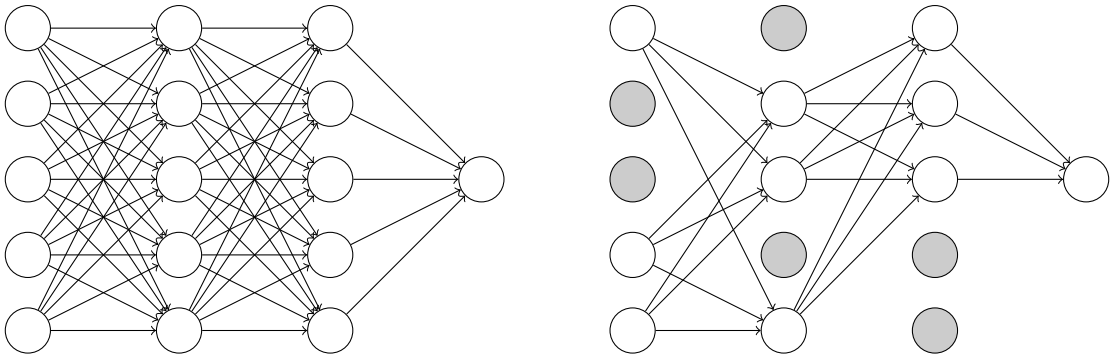


Fig. 5.1: Illustration of the DropOut mechanism. In order to train a given model (left), at each mini-batch, a given proportion of neurons is picked at random to be "switched off" and the subsequent sub-network is used for the current optimization step (*cf.* right-hand side figure, in which 40% of the neurons – coloured in gray – are switched off).

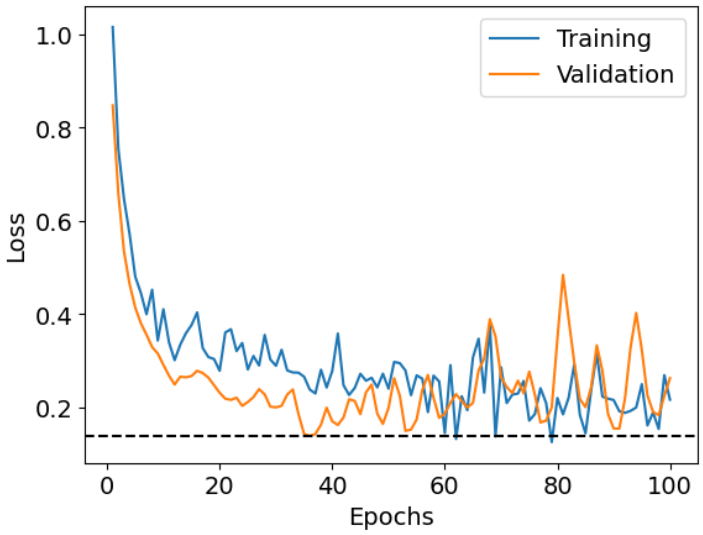
In this section, we present the DropOut strategy, which was introduced in [Srivastava *et al.*, 2014]. The idea behind DropOut is to *switch off* some of the neurons during training. The switched off neurons change at each mini-batch such

that, overall, all neurons are trained during the whole process.

The concept is very similar in spirit to a strategy that is used for training random forest, which consists in randomly selecting candidate variables for each tree split inside a forest, which is known to lead to better generalization performance for random forests. The main difference here is that one can not only switch off *input neurons* but also *hidden-layer ones* during training.

In `keras`, this is implemented as a layer, which acts by switching off neurons from the previous layer in the network:

```
from keras.layers import Dropout

set_random_seed(0)
switchoff_proba = 0.3
model = Sequential([
    InputLayer(input_shape=(4, )),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=256, activation="relu"),
    Dropout(rate=switchoff_proba),
    Dense(units=3, activation="softmax")
])

n_epochs = 100
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
h = model.fit(X, y, validation_split=0.3, epochs=n_epochs, batch_size=30, verbose=0)
```



Exercise #1

When observing the loss values in the figure above, can you explain why the validation loss is almost consistently lower than the training one?

Solution

In fact, the training loss is computed as the average loss over all training mini-batches during an epoch. Now, if we recall that during training, at each minibatch, 30% of the neurons are switched-off, one can see that only a subpart of the full

model is used when evaluating the training loss while the full model is retrieved when predicting on the validation set, which explains why the measured validation loss is lower than the training one.

CHAPTER 6

CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (*aka* ConvNets) are designed to take advantage of the structure in the data. In this chapter, we will discuss two flavours of ConvNets: we will start with the monodimensional case and see how ConvNets with 1D convolutions can be helpful to process time series and we will then introduce the 2D case that is especially useful to process image data.

6.1 ConvNets for time series

Convolutional neural networks for time series rely on the 1D convolution operator that, given a time series $\mathbf{x}$ and a filter $\mathbf{f}$, computes an activation map as:

$$(\mathbf{x} * \mathbf{f})(t) = \sum_{k=-L}^{L} f_k x_{t+k} \tag{6.1}$$

where the filter $\mathbf{f}$ is of length $(2L+1)$.

The following code illustrates this notion using a Gaussian filter:

Convolutional neural networks are made of convolution blocks whose parameters are the coefficients of the filters they embed (hence filters are not fixed *a priori* as in the example above but rather learned). These convolution blocks are translation equivariant, which means that a (temporal) shift in their input results in the same temporal shift in the output:

```
/tmp/ipykernel_11148/368849627.py:32: UserWarning: This figure includes Axes that␣
↪are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
/tmp/ipykernel_11148/368849627.py:23: MatplotlibDeprecationWarning: Auto-removal␣
↪of overlapping axes is deprecated since 3.6 and will be removed two minor␣
↪releases later; explicitly call ax.remove() as needed.
  fig2 = plt.subplot(2, 1, 2)
/tmp/ipykernel_11148/368849627.py:32: UserWarning: The figure layout has changed␣
↪to tight
  plt.tight_layout()
```

```
<IPython.core.display.HTML object>
```

Convolutional models are known to perform very well in computer vision applications, using moderate amounts of parameters compared to fully connected ones (of course, counter-examples exist, and the term "moderate" is especially vague).

Most standard time series architectures that rely on convolutional blocks are straight-forward adaptations of models from the computer vision community ([Le Guennec *et al.*, 2016] relies on an old-fashioned alternance between convolution and pooling layers, while more recent works rely on residual connections and inception modules [Fawaz *et al.*, 2020]). These basic blocks (convolution, pooling, residual layers) are discussed in more details in the next Section.

These time series classification models (and more) are presented and benchmarked in [Fawaz *et al.*, 2019] that we advise the interested reader to refer to for more details.

6.2 Convolutional neural networks for images

We now turn our focus to the 2D case, in which our convolution filters will not slide on a single axis as in the time series case but rather on the two dimensions (width and height) of an image.

6.2.1 Images and convolutions

As seen below, an image is a pixel grid, and each pixel has an intensity value in each of the image channels. Color images are typically made of 3 channels (Red, Green and Blue here).

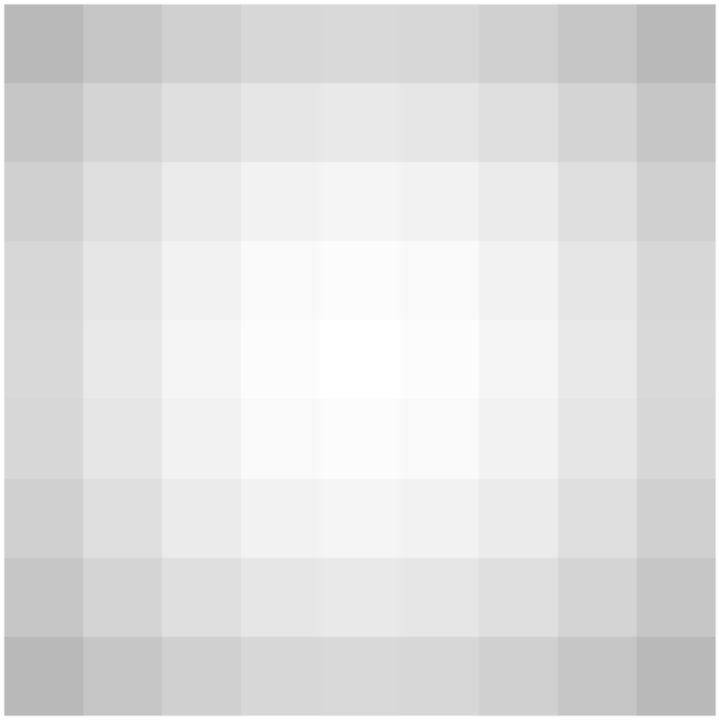
Fig. 6.1: An image and its 3 channels (Red, Green and Blue intensity, from left to right).

The output of a convolution on an image $\mathbf{x}$ is a new image, whose pixel values can be computed as:

$$(\mathbf{x} * \mathbf{f})(i,j) = \sum_{k=-K}^{K} \sum_{l=-L}^{L} \sum_{c=1}^{3} f_{k,l,c} x_{i+k,j+l,c}. \tag{6.2}$$

In other words, the output image pixels are computed as the dot product between a convolution filter (which is a tensor of shape $(2K+1, 2L+1, c)$) and the image patch centered at the given position.

Let us, for example, consider the following 9x9 convolution filter:



Then the output of the convolution of the cat image above with this filter is the following greyscale (*ie.* single channel) image:

One might notice that this image is a blurred version of the original image. This is because we used a Gaussian filter in the process. As for time series, when using convolution operations in neural networks, the contents of the filters will be learnt, rather than set *a priori*.

6.2.2 CNNs *à la* LeNet

In [LeCun *et al.*, 1998], a stack of convolution, pooling and fully connected layers is introduced for an image classification task, more specifically a digit recognition application. The resulting neural network, called LeNet, is depicted below:

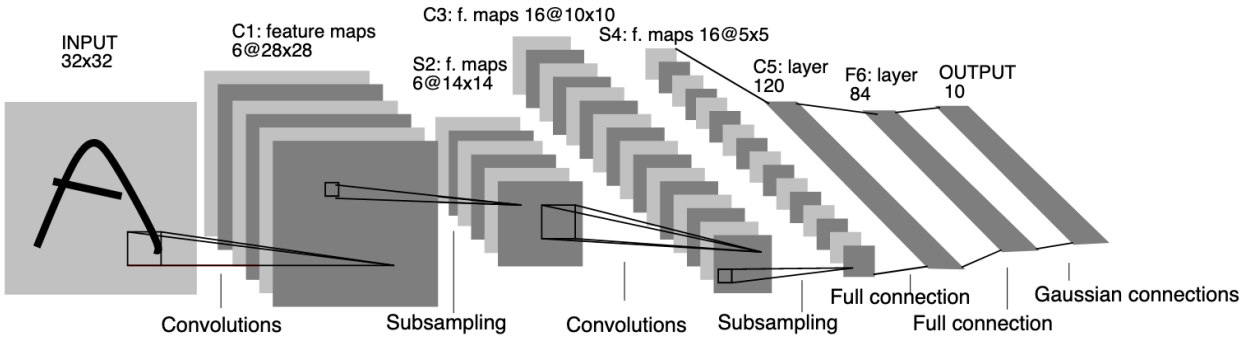


Fig. 6.2: LeNet-5 model

Convolution layers

A convolution layer is made of several convolution filters (also called *kernels*) that operate in parallel on the same input image. Each convolution filter generates an output activation map and all these maps are stacked (in the channel dimension) to form the output of the convolution layer. All filters in a layer share the same width and height. A bias term and an activation function can be used in convolution layers, as in other neural network layers. All in all, the output of a convolution filter is computed as:

$$(\mathbf{x} * \mathbf{f})(i,j,c) = \varphi\left(\sum_{k=-K}^{K}\sum_{l=-L}^{L}\sum_{c'} f^{c}_{k,l,c'} x_{i+k,j+l,c'} + b_c\right) \tag{6.3}$$

where c denotes the output channel (note that each output channel is associated with a filter f^c), b_c is its associated bias term and φ is the activation function to be used.

Tip: In `keras`, such a layer is implemented using the `Conv2D` class:

```
import keras_core as keras
from keras.layers import Conv2D

layer = Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu")
```

Padding

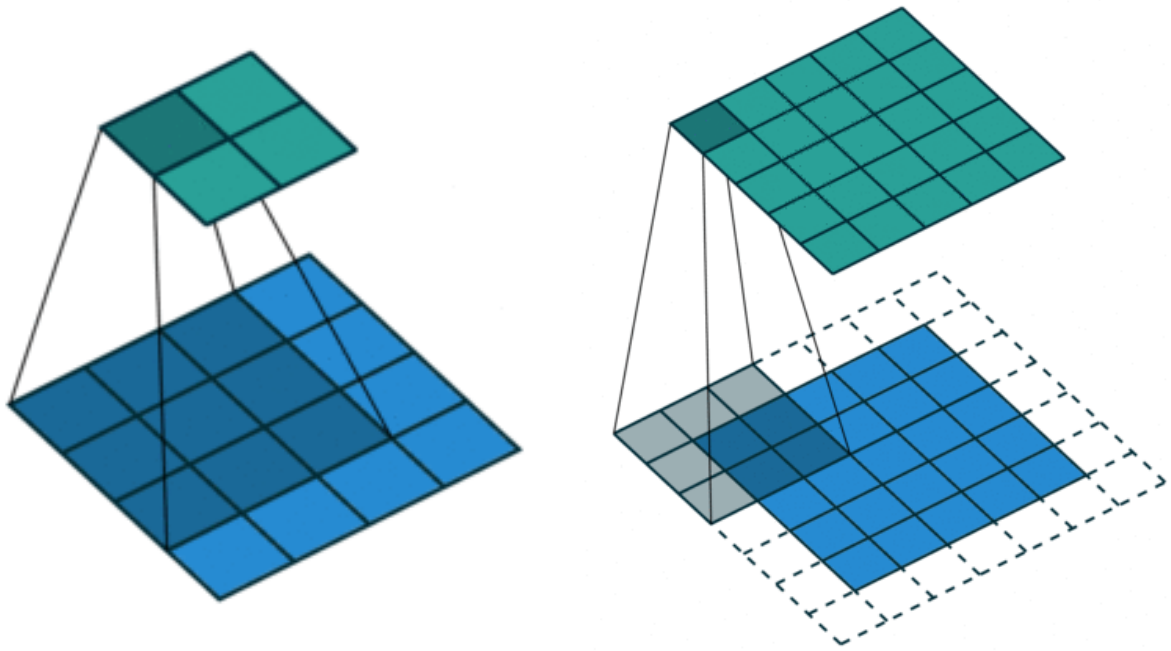


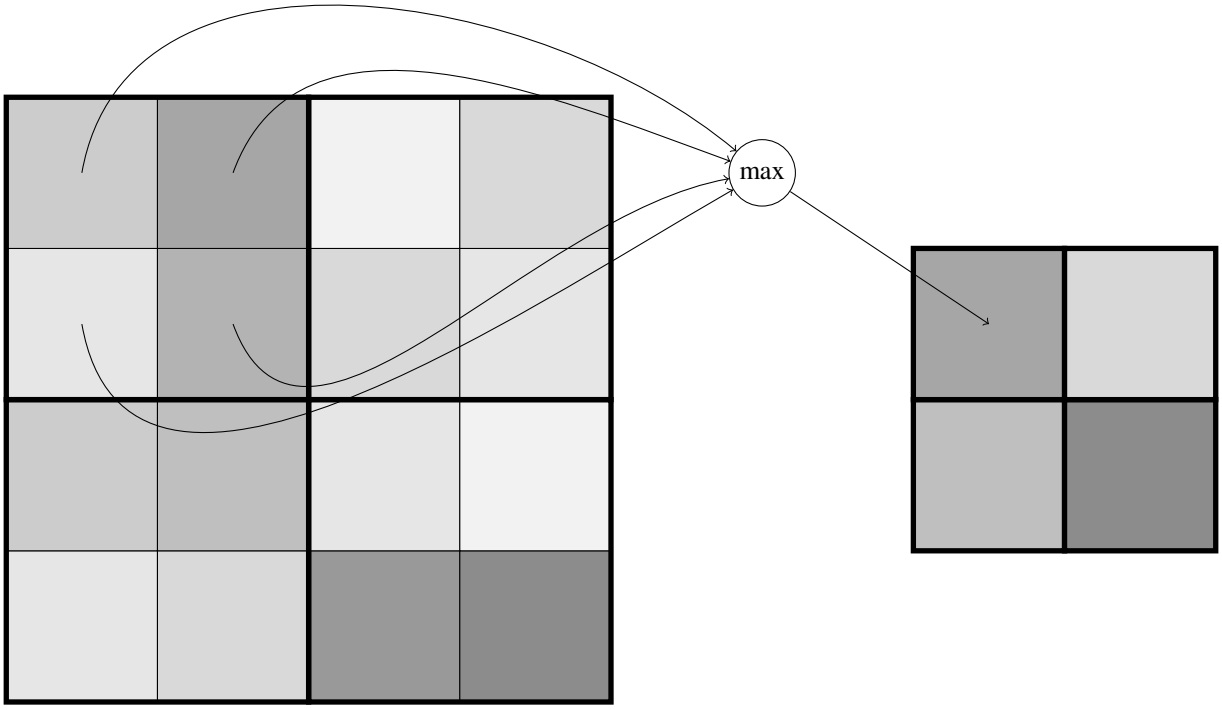
Fig. 6.3: A visual explanation of padding (source: V. Dumoulin, F. Visin - A guide to convolution arithmetic for deep learning). Left: Without padding, right: With padding.

When processing an input image, it can be useful to ensure that the output feature map has the same width and height as the input image. This can be achieved by padding the input image with surrounding zeros, as illustrated in Fig. 6.3 in which the padding area is represented in white.

Pooling layers

Pooling layers perform a subsampling operation that somehow summarizes the information contained in feature maps in lower resolution maps.

The idea is to compute, for each image patch, an output feature that computes an aggregate of the pixels in the patch. Typical aggregation operators are average (in this case the corresponding layer is called an average pooling layer) or maximum (for max pooling layers) operators. In order to reduce the resolution of the output maps, these aggregates are typically computed on sliding windows that do not overlap, as illustrated below, for a max pooling with a pool size of 2x2:



Such layers were widely used in the early years of convolutional models and are now less and less used as the available amount of computational power grows.

Tip: In `keras`, pooling layers are implemented through the `MaxPool2D` and `AvgPool2D` classes:

```
from keras.layers import MaxPool2D, AvgPool2D

max_pooling_layer = MaxPool2D(pool_size=2)
average_pooling_layer = AvgPool2D(pool_size=2)
```

Plugging fully-connected layers at the output

A stack of convolution and pooling layers outputs a structured activation map (that takes the form of 2d grid with an additional channel dimension). When image classification is targeted, the goal is to output the most probable class for the input image, which is usually performed by a classification head that consists in fully-connected layers.

In order for the classification head to be able to process an activation map, information from this map needs to be transformed into a vector. This operation is called flattening in `keras`, and the model corresponding to Fig. 6.2 can be implemented as:

```
from keras.models import Sequential
from keras.layers import InputLayer, Conv2D, MaxPool2D, Flatten, Dense

model = Sequential([
    InputLayer(input_shape=(32, 32, 1)),
    Conv2D(filters=6, kernel_size=5, padding="valid", activation="relu"),
```

(continues on next page)

(continued from previous page)

```
    MaxPool2D(pool_size=2),
    Conv2D(filters=16, kernel_size=5, padding="valid", activation="relu"),
    MaxPool2D(pool_size=2),
    Flatten(),
    Dense(120, activation="relu"),
    Dense(84, activation="relu"),
    Dense(10, activation="softmax")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```