RDFIA - Introduction to neural networks

1 - Theoretical foundation

1) What are the train, val and test sets used for?

The training, validation and test sets have very different roles.

- The training set is simply here for the neural network to learn the relation between the features $x^{(i)}$ and the targets $y^{(i)}$.
- The test set should only be used tocontrol the coherence of its result with the training set and therefore detect under/ overfitting.
- On the other hand, validation set is used to tune the hyperparameters of the model. If the seperation between validation and testing is not respected, we will, in this case, over fit our model to this dataset. The model could lead to surprising result once in production.

2) What is the influence of the number of examples N?

N represents the number of samples of the relation between features $x^{(i)}$ and targets $y^{(i)}$. When N increases, we, therefore, obtain a better sampling of this relation. This extra information ultimately helps estimating the underlying distribution.

On the other hand, from a more practical point of view, a low N will imply a restricted number of examples. This limitation to sample the underlying distribution makes it hard to predict the behavior from unseen example. In this context, it is said that the model strugles to generalize.

3) Why is it important to add activation functions between linear transformations?

Even tough linear transformations are easy to compute, their capacity to learn complex behavior is limited. Indeed, any combinations of linear transforation can be summed up as a single linear transformation. Activation function is one solution to bring non-linearity in neural networks, and therefore, learn non-linear behavior.

4) What are the sizes n_x, n_h, n_y in the figure 1? In practice, how are these sizes chosen?

In figure 1,

- $n_x = 2$, as the input layer contains 2 neurons.
- $n_h = 4$, there are 4 neurons on the hidden layer.
- $n_y = 2$, as the ouput layer is composed of 2 neurons.

The n_x and n_y sizes are constrained by the dataset. Indeed, n_x must be equal to the number of features, while n_y should match the number of targets.

Lastly, n_h can be chosen arbitraly. More often than not, hidden layers have smaller width than the input layer. This shrinking architecture forces the network to learn a compressed representation of the original input.

This process takes its inspiration from how our brain work. The photoreceptor neurons (the equivalent input pixel of our brain) in our retina are 100 times more numerous than the axons that make up the optic nerve. Said in other terms, the hidden neurons of our retina compress the information by computing features, like edges or contours. Only those features are conveyed by the optic nerve to the brain for the rest of the computation.

5) What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?

y and $\hat{y}$ plays drastically different roles. y reprents the Ground Truth, i.e. the correct target associated to a set of features. On the other hand, $\hat{y}$ is the prediction of our models for those same features.

y and $\hat{y}$ also differs in the way they are obtain. $\hat{y}$ is computed by a forward pass on the whole neural network. While y is manualy set, in the context of supervised learning, or computed in some way in the self-supervised setting.

6) Why use a SoftMax function as the output activation function?

The goal of this pratical is to study the role of neural network in the single-label classification task (i.e. to find the unique label associated to some input features). This unique label target is encoded as one-hot vector (i.e. a vector with a single 1 at the index of the correct label and 0 elsewhere). We therefore need an `argmax` function to transition from an outputed vector of "scores" (with variable range of values) to a probability distribution matching this one-hot encoding (which range is $[0, 1]$).

The classical `argmax` function cannot be used for this purpose, as it is non-differentiable. This property would hinder most common optimization algorithms, based on the gradient.

In the context of binary or multi-label classification, the `sigmoid` function can be used as a great continuous alternative to `argmax`. But our study is focused on mutually exclusive classes, therefore the `sigmoid` function won't be helpful. On the other hand, Softmax is more appropriate to find this probability distribution close to one-hot encoding.

7) Write the mathematical equations allowing to perform the forward pass of the neural network, i.e. allowing to successively produce $\tilde{h}$, h, $\tilde{y}$ and $\hat{y}$ starting at x.

$\tilde{h}$, h, $\tilde{y}$ and $\hat{y}$ are computed the following way:

$$\tilde{h} = xW^{h^T} + b^h h = tanh(\tilde{h})\tilde{y} = hW^{y^T} + b^y y = SoftMax(\tilde{y})$$

with W^h and W^y the weight matrix, respectively, before the second and third layers. And with b^h and b^y the bias vector, respectively, of the second and third layers.

8) During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$?

Cross-entropy As a reminder, the cross-entropy loss is defined as follows:

$$\mathcal{L}(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$$

In our context, y_i is encoded as a one-hot vector. That is all its indices are set to 0 except the one corresponding to the class which is set to 1. This loss, therefore, only penalises prediction $\hat{y}_i$ of the corresponding correct class (i.e. the class set to 1 in the one-hot encoding). To minimize the cross-entropy loss, $\hat{y}_i$ should be equal to 1, with i corresponding to the ground truth class. All other predictions $\hat{y}_{j \neq i}$ do not matter.

MSE The Mean Squared Error (MSE) is defined as follows:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

On the opposite of cross-entropy loss, each $\hat{y}_i$ matters to minimize the MSE. More precisely, every y_i should equal $\hat{y}_i$ to minimize it.

9) How are these functions better suited to classification or regression tasks?

There are several underlying reasons to justify that cross-entropy and the MSE loss are best suited, respectively, for the classification and regression task.

- The inherent likelihood model of a classification task is a logistic model (because of the sigmoid function used to estimate the class probabilities). Computing the maximum of likelihood of a logistic model is equivalent to minimising the cross-entropy. On the other hand, inherent likelihood model

of a regression task follows a gaussian distribution. Then, computing its maximum of likelihood is equivalent to minimising the MSE loss. Therefore, when using a MSE loss for a classification task, you make the implicit assumption of a gaussian distribution, while $y_i \in \{0, 1\}$.

- A second argument in favor of the cross-entropy loss for the classification task is regarding the difficulty to use the MSE to optimize a sigmoid function. Indeed, the application of a MSE loss on a sigmoid returns a non-convex function, which is therefore harder to optimize. But the sigmoid function is precisely used in the context of classification.
- The cross-entropy loss return more significant errors in the classification task. And, more important errors help the gradient descent to converge faster. Indeed, because of the one-hot encoding, the maximum of $\hat{y}_i - y_i$ can not exceed 1. Therefore, the MSE returns an error between $[0, 1]$, which is quite small. On the other hand, $y_i \times \log(\hat{y}_i)$ returns a range of value within $[-\infty, +\infty]$
- Lastly, the MSE loss is a distance function for real vectors, while cross-entropy is used to compare probability distribution. In the classification task, you want to maximise the probability to belong to the correct class. On the other hand, the regression task's goal is to minimize the distance between your predictions and their real target values.

10) What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case?

The gradient descent algorithm comes in three different variants: classic batch gradient descent, stochastic gradient descent and mini-batch gradient descent. Appart from the convergence to the optimum, which they all have under correct hyperparameters, there are actually some significant differences between those three approaches.

First, from a practical point of view, the time to converge, measured in second, is the major criteria to compare those algorithms. The time to convergence can be decompose into two aspects:

- Number of steps to converge
- Computationnal time needed per step

Therefore, a great gradient descent algorithm must combines the two efficiently. The classic batch gradient descent algorithm most precisely estimates the gradient. Those stable steps therefore helps to minimize the number of steps required to converge. But this comes at the cost processing the whole datasets and therefore becomes very time expensive for large datasets.

The online SGD on the other hand uses a single sample to compute its gradient. Each step's computational time is therefore ridiculous. But this makes it also sensible to the noise of each sample. This results in a zig-zag effect in the convergence toward the global minimum (we can also describe this as a higher variance over training epochs). These zig-zags slow down the convergence.

It is also worth noting that SGD is the simplest algorithm to implement, which makes it handy for beginners. It provides rapid feedback on the learning process. And its noisy aspect can, in fact, enable it to skip some local minima.

Finaly, the mini-batch gradient descent is a good trade-off between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. Each of its step are quite fast to compute and the estimation of the gradient is also quite effective. In the end, this approach seems to be the fastest.

The mini-batch algorithm also has another significant advantage compare to the classic batch gradient descent on the memory perspective. Indeed, the latter needs to hold its whole content in memory. This is not always possible, especially for large datasets. The mini-batch algorithm solves this problem.

To concluse, because of the advantage discussed above, the most generic and convenient gradient descent algorithm is clearly the mini-batch SGD. Those advantages makes it the most common implementation of gradient descent used in the field of deep learning.

11) What is the influence of the learning rate η on learning?

In it most basic configuration, the gradient descent algorithm updates the weights w of the neural network with the following formula:

$$w = w - \eta \frac{\partial \mathcal{L}(X, Y)}{\partial w}$$

Intuitively, the learning rate tuning controls the speed at which the weights are updated, and how quickly the model is adapted to the problem. A larger η implies larger step in the direction of the optimal solution and requires fewer training epochs.

Most of the time, larger step will lead to a faster convergence. But this fact is not always true. A too large η value might encourage steps that go beyond the optimal solution. If those missed steps are repeated several time, it can lead to an oscillating or diverging learning. This hard-to tune specificities makes the learning rate η one of the most important hyperparameters of a neural network.

On the other hand, small η can also hamper the model's performance. Beyond the increase of required training epochs, small η can stuck the gradient descent in local minima, and therefore prevent the algorithm from finding an optimal solution.

In order to prevent long training process that could get stuck or unstable training process, the learning rate is often varied during training. The state-of-the-art proposes two approaches to tweaks η during training:

- Learning rate schedule (with decay and momentum)
- Adaptive learning rate

In both approach, the intuition is the same. The first few steps are undergone with large η to quickly converge toward the optimum. The steps are then progressively reduced to allow more fine-grain steps in the neighborhood of the solution.

12) Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss with respect to the parameters, using the naive approach and the backprop algorithm.

Let n be the number of layers of a neural network. We ignore the bias weight, as they will only multiply the complexity by a constant factor (i.e. 2). In the naive calculation of the gradients, each layer computes the gradient of its successive layer before computing its own. Therefore, a layer $i \in 1 \dots n$ would need to compute the gradient of $n - i$ layer.s. This is an arithmetic sequence. The total number of layers' gradient computed is equal to $\frac{n(n-1)}{2}$. Which makes the complexity of the naive approach in $O(n^2)$.

On the other hand, the backpropagation algorithm only computes the derivatives of each layer once. This is done by computing the gradient of the last layer and progressively computing the gradient of the preceding layers by applying the chain rule on the last layer computed. This clever use of the chain rule brings down the complexity of the computation to $O(n)$.

13) What criteria must the network architecture meet to allow such an optimization procedure?

The descent gradient algorithm is based on the gradient, as its name suggests. The network architecture therefore needs to be differentiable to apply this algorithm. Nevertheless, point-wise non-differentiality can be solve. For example, the famous "*Rectified Linear Unit*" (ReLU) activation function is non-differentiable at 0. This differentiation issue at 0 is solved with sub-derivatives computation.

14) The function SoftMax and the loss of cross-entropy are often used together and their gradient is very simple. Show that the loss can be simplified by:

$$l = -\sum_i y_i \tilde{y_i} + \log\left(\sum_i e^{\tilde{y_i}}\right)$$

Let y, $\hat{y}$, and $\tilde{y}$ be, respectively, the target, the predicted target, and the predicted target without the softmax activation. The cross-entropy loss l and the predicted target $\hat{y}$ are defined as follows:

$$\begin{cases} l = -\sum_i y_i \log(\hat{y}_i) \\ \hat{y} = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} \end{cases}$$

If we replace $\hat{y}$ in l, we get:

$$l = -\sum_i y_i \log(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}})$$

$$l = -\sum_i y_i \left(\tilde{y}_i - \log(\sum_j e^{\tilde{y}_j}) \right)$$

$$l = -\sum_i y_i \tilde{y}_i + \sum_i y_i \log(\sum_j e^{\tilde{y}_j})$$

Finally, y_i is a one-hot vector. i.e. $y_i = [0 \dots 010 \dots 0]$. We can therefore keep the only non-nul element of the sum from the right-term. This is equal to:

$$l = -\sum_i y_i \tilde{y}_i + \log \left(\sum_j e^{\tilde{y}_j} \right)$$

15) Write the gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}$.

From the previous question,

$$l = -\sum_i y_i \tilde{y}_i + \log \left(\sum_j e^{\tilde{y}_j} \right)$$

By computing its gradient relative to $\tilde{y}_i$, and by applying the chain rule, we get:

$$\frac{\partial l}{\partial \tilde{y}_i} = - \frac{\partial \left(\sum_j y_j \tilde{y}_j \right)}{\partial \tilde{y}_i} + \frac{\partial log}{\partial \tilde{y}_i} \left(\sum_j e^{\tilde{y}_j} \right) \frac{\partial \left(\sum_j e^{\tilde{y}_j} \right)}{\partial \tilde{y}_i}$$

$$\frac{\partial l}{\partial \tilde{y}_i} = - y_i + \frac{1}{\sum_j e^{\tilde{y}_j}} e^{\tilde{y}_i}$$

Finally, we can identify the SoftMax definition in the right term:

$$\frac{\partial l}{\partial \tilde{y}_i} = - y_i + SoftMax(\tilde{y})_i = \hat{y}_i - y_i$$

And,

$$\nabla_{\tilde{y}} l = \hat{y} - y$$

16) Using the backpropagation, write the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$. Note that writing this gradient uses $\nabla_{\tilde{y}} l$. Do the same for $\nabla_{b_y} l$.

For $\nabla_{W_y} l$, By applying the chain rule, we get:

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial W_{y,ij}}$$

We can then replaced $\tilde{y_k}$ by its definition:

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \left(\sum_{m=1}^{n_h} W_{y,km} h_m + b_{y,k}\right)}{\partial W_{y,ij}}$$

For the remaining partial derivatives, we can remove $b_{y,k}$ and the $W_{y,km}$ with $m \neq j$, as they are independent of $W_{y,ij}$:

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial W_{y,kj} h_j}{\partial W_{y,ij}}$$

With the same reasoning:

$$\frac{\partial l}{\partial W_{y,ij}} = \frac{\partial l}{\partial \tilde{y_k}} h_j$$

And,

$$\nabla_{W_y} l = \nabla_{\tilde{y}} l . h$$

For $\nabla_{b_y} l$, similarly, By applying the chain rule, we get:

$$\frac{\partial l}{\partial b_{y,j}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial b_{y,j}}$$

We can then replaced $\frac{\partial l}{\partial \tilde{y_k}}$ and $\tilde{y_k}$ by their definition:

$$\frac{\partial l}{\partial b_{y,j}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \left(\sum_{m=1}^{n_h} W_{y,km} h_m + b_{y,k}\right)}{\partial b_{y,j}}$$

For the remaining partial derivatives, we can remove the $W_{y,km}$, as they are independent of $b_{y,j}$:

$$\frac{\partial l}{\partial b_{y,j}} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial b_{y,k}}{\partial b_{y,j}}$$

And then, we can remove the $b_{y,k}$ with $k \neq j$, as they are independent of $b_{y,j}$:

$$\frac{\partial l}{\partial b_{y,j}} = \frac{\partial l}{\partial \tilde{y_j}}$$

And,

$$\nabla_{b_y} l = \nabla_{\tilde{y}} l$$

17) Compute other gradients: $\nabla_{\tilde{h}} l$, $\nabla_{W_h} l$, and $\nabla_{b_h} l$.

In order to compute $\frac{\partial l}{\partial \tilde{h_j}}$, we first need $\frac{\partial l}{\partial h_j}$. Let's compute it to.

For $\frac{\partial l}{\partial h_j}$, Similarly to the previous question,

$$\frac{\partial l}{\partial h_j} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial h_j} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \left(\sum_{m=1}^{n_h} W_{y,km} h_m + b_{y,k}\right)}{\partial h_j}$$

$$\frac{\partial l}{\partial h_j} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial W_{y,kj} h_j}{\partial h_j} = \sum_k \frac{\partial l}{\partial \tilde{y_k}} W_{y,kj}$$

Therefore,

$$\nabla_h l = W^{yT} \nabla_{\tilde{y}} l$$

For $\frac{\partial l}{\partial \tilde{h}_j}$,

$$\frac{\partial l}{\partial \tilde{h_j}} = \frac{\partial l}{\partial h_j} \frac{\partial h_j}{\partial \tilde{h}_j} = \frac{\partial l}{\partial h_j} \frac{\partial \tanh(\tilde{h}_j)}{\partial \tilde{h}_j} = \frac{\partial l}{\partial h_j} \left(1 - \tanh^2(\tilde{h}_j)\right) = \left(1 - h_j^2\right) \frac{\partial l}{\partial h_j}$$

Therefore, with $\odot$ the hadamard element-wise product,

$$\nabla_{\tilde{h}} l = (1 - h^2) \odot \nabla_h l$$

For $\frac{\partial l}{\partial W_{h,ij}}$,

$$\frac{\partial l}{\partial W_{h,ij}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \left(\sum_m W_{h,km} x_m + b_{h,k}\right)}{\partial W_{h,ij}} = \frac{\partial l}{\partial \tilde{h}_i} x_j$$

Therefore,

$$\nabla_{W_h} l = \nabla_{\tilde{h}} l . x^T$$

For $\frac{\partial l}{\partial b_{h,j}}$,

$$\frac{\partial l}{\partial b_{h,j}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,j}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \left(\sum_m W_{h,km} x_m + b_{h,k}\right)}{\partial b_{h,j}} = \frac{\partial l}{\partial \tilde{h}_j}$$

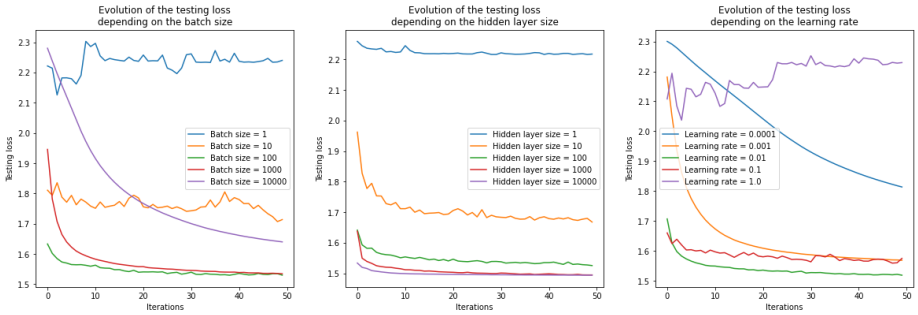
Therefore,

$$\nabla_{b_h} l = \nabla_{\tilde{h}} l$$

2 - Implementation

1) Discuss and analyze your experiments following the implementation. Provide pertinent figures showing the evolution of the loss; effects of different batch size / learning rate, etc

In our experimentations, we have played with three different parameters: the batch size, the hidden layer size and the learning rate. Here is a plot summarizing our work.



As discussed earlier (in question 10), the batch-size influences the steps stability (i.e. good estimation of the gradient), the computationnal cost of an iteration, and its memory usage.

Our experimentation shows that it is necessary to find a sweet spot for the tuning of the batch size. In the problem that concerns us, the batch size that provided the fastest convergence seems to be around 100. Larger or smaller batch size slows down the learning or can even prevent it from learning.

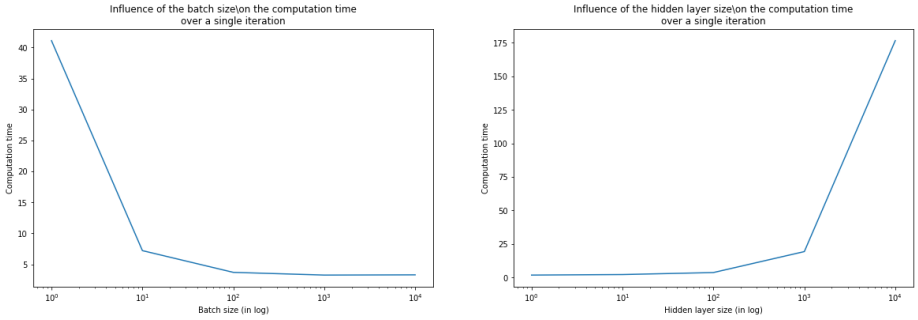
Regarding the hidden layer size, larger layer seems to improve both the convergence and the solution found. The latter is showned by the fact that the hidden layer of size 1 and 10 seems to have converged toward a non-optimal solution. Their smaller size seems to hamper the learning of more complex scheme.

Lastly, this experimentation shows us interesting lessons about the learning rate. We know that larger η implies larger step in the direction of the optimal solution which enables faster convergence. This first phenomenon can remains true until we reach a learning rate of 0.01. Beyond this point, the learning rate starts to increase. The gradient descent with the learning rate of 1 is even clearly

diverging. The diverging behavior might also explain the poorer performance of the gradient descent with the learning rate of 0.1.

This experimentation also shown the impact of low learning rate on the quality of the solution found. Indeed the learning rate of 0.001 seems to have converge in a local optima. We would have expected the same behavior from the learning rate of 0.0001 if we had pushed the experience with more training steps. In the end, the most suited learning rate for our problem seems to be around 0.01.

It is also worth the interest to give a closer a look to the computation time hidden behind those numbers. We have studied the computational time behind the experimented batch size and hidden layer size. We choose to not included the learning rate into this experiment as this parameter has clearly no influence on the computationnal time of one iteration.



First, we can observe that smaller batch size have higher computational cost. Our optimal batch size of 100 seems to also match an optimal computational cost. Secondly, even tough, we found that larger hidden layer improves performance, this comes at the cost of a significantly higher computational time.

Convolutional Neural Networks

1 - Introduction to convolutional networks

1a) Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size xÖyÖz what will be the output size?

The output size of such an operation would be $x' \times y' \times 1$, with:

$$\begin{cases} x' = \left\lceil \frac{x+2p-(k-1)}{s} \right\rceil \\ y' = \left\lceil \frac{y+2p-(k-1)}{s} \right\rceil \end{cases}$$

1b) How much weight is there to learn?

This convolution operation learns a unique convolution per depth layer (with z the depth). Each convolution has $k \times k$ parameters. This operation therefore

needs to learn this number of parameter:

$$k \times k \times z$$

1c) How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size?

The number of weights of a fully-connected layer with n input neurons and m output neurons is $n \times m$.

This operation would have then needed this number of weights:

$$(x \times y \times z) \times (y' \times x')$$

To give an order of magnitude, a classic convolutional operation with a 3×3 kernel, a stride of 1 and a padding of 1 on a 512×512 RGB image would require $3 \times 3 \times 3 = 27$ parameters for the convolution layer. While this same operation with a fully-connecter layer would require this number of parameters

$$(512 \times 512 \times 3) \times (512 \times 512) \approx 2.10^{11}$$

2) What are the advantages of convolution over fully-connected layers? What is its main limit?

Convolutional layers are becoming mainstream in the state-of-the-art architecture for most visual task. Their main distinctive aspect lies in their narrower range of features. In fully-connected layers, each neuron is connected to every neurons from the previous layer. Therefore, any change of a single neuron from a layer will impact every neuron of the next layer. On the other hand, convolutional layers have narrower range of features. Each of its neuron is only connected to "nearby" neurons from the preceding layer within the width of the convolutional kernel. In this context, the activation of any one neuron is insensitive to the activation of most of the neurons from the previous layer.

This restricted range of feature enable convolution filters to learn local information (like lines or curved edges). Those local information are also extracted on the whole image with way fewer effort. As it is only needed to learn a few dozens parameters to achieve so.

The narrower range of features also helps to reduce the number of parameters of the model. This limited number of parameters eases the learning process, while still being able to interpret complex scheme. This has a significant impact on high-dimensional inputs, like images.

Even though, the convolutional layers have lot of advantages compared to fully-connected layers, they still have weak points. The main limits of convolutional layers is their struggle to learn long distance relationship, and to handle rotation and scale-invariance without explicit data augmentation.

Beyond this, CNN architecture requires higher computational capabilities, due to the convolution operation. Those computation typically need a GPU to train it in a reasonable amout of time. CNN architecture also requires more data to be trained.

3) Why do we use spatial pooling?

Convolutional layers can successfully extract maps of features in an input image. But these output feature maps suffer from a location-sensitivity. This means that small movements in the position of the feature in the input image will result in a different feature map. The sensitivity may hamper latter tasks, like classification.

One approach to address this sensitivity is to down sample these feature maps. The down sampling process improves the robustness of the feature map against location variation. This is said to improve the "local translation invariance".

The pooling operation also plays an important role in the dimensionality reduction necessary to obtain a low-dimensionnality classification.

Though, most recent approaches, like Kaiming et al.[1], prefer other alternatives like 1×1 convolution layer or strides to achieve this dimensionality reduction.

4) Suppose we try to compute the output of a classical convolutional network (for example the one in Figure 2) for an input image larger than the initially planned size (224×224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image?

In the architecture specified in figure 2, the three fully-connected layers expect to process a uni-dimensionnal vector. Unfortunately, a larger image won't fit this constraint, as the outputed shape will be a three-dimensional tensor. It is, therefore, impossible to keep all part of the neural network to process a larger image of this kind.

On the other hand, it is indeed possible to keep the convolutional and pooling layers. Those are defined or learned on kernels, which makes them suitable for any images size.

5) Show that we can analyze fully-connected layers as particular convolutions.

If we are allowed to modify the shape of the image, there exist a simple way to interpret a fully-connected layer as a convolutional one. Let d be number of neurons on the output of the fully-connected layer. First we need to flatten the image into a $1 \times 1 \times n$ tensor (with $n = I_{width} \times I_{height} \times I_{nb\_channels}$). Then, d

[1]Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016

convolution filters with a kernel of size 1×1 will process the image in the same way than a fully-connected layer would have.

Another approach to mimic a fully-connected layer can be performed with d convolutional filters with a kernel size of $I_{width} \times I_{height}$.

6) Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest?

By replacing the fully-connected layers by one of its convolutional equivalent, we can now overpass the previous constraint. Thanks to the ability of convolutional layer to process any image size, we can now keep the full network architecture and process any image with it.

We can easily compute its output shape. The successive pooling layers, in the proposed architecture, divide the input image by 224 along its width and height. Therefore, the output shape of such a neural network, on an image of size $n \times m \times d$, will be $\frac{n}{224} \times \frac{m}{224} \times 1000$.

Beyond, the free handling of larger image, this approach has other points of interest. It pushes the boundary of the domain "beyond ImageNet". This approach allows to re-use a model trained on ImageNet to handle not only classification, but also object detection. This transition is important. It allows interpretation of more complex scene, like those represented in the datasets "MS COCO" and "VOC 2012".

7) We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers? Can you imagine what happens to the deeper layers? How to interpret it?

In the general context, if no pooling have been performed so far, the first and second convolutional layers have respectively a receptive fields of size $k_1 \times k_1$, and $(k_1 + k_2 - 1) \times (k_1 + k_2 - 1)$, with k_1, and k_2 the kernel size of those two layers. In the context of VGG16 in our example, the two first layers have both a kernel size of 3. Therefore, the receptive field of the first layer is 3. And the receptive field of the second layer is 5.

When diving deeper into the layers, as the computed feature maps becomes more and more high-level, the receptive fields also increase. Those two phenomena enable deeper layers to process complex and broad signals. Intuitively, by perceiving large set of pixels from the input image that have been process to contain high-level information (like feathers or eyes), we can now recognize, in such an image, complex object (like a bird). This is at the root of complex task like object recognition.

2 - Training from scratch of the model

8) For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed?

In theory, we could try to solve the following equation with $x = 32$, $y = 32$, and $k = 5$.

$$\begin{cases} x = x' = \left\lceil \frac{x+2p-(k-1)}{s} \right\rceil \\ y = y' = \left\lceil \frac{y+2p-(k-1)}{s} \right\rceil \end{cases}$$

But we will find multiple solutions as the system is under-constrained. In fact, among those solutions, only one does not use aberating padding values (to compensate bigger strides). This solution is the most obvious one:

$$\begin{cases} p = 2 \\ s = 1 \end{cases}$$

9) For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed?

In the same way, with $k = 2$, the obvious solution come up naturally:

$$\begin{cases} p = 0 \\ s = 2 \end{cases}$$

10) For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

The number of weights of a convolution layer W_{conv} and of a fully connected layer W_{fc} simply follows these formula:

$$\begin{cases} W_{conv}(i) = C_i * k_i * k_i * C_{i+1} + C_{i+1} \\ W_{fc}(i) = N_i * N_{i+1} + N_i \end{cases}$$

with C_i the number of channels or filters of a given layer i, N_i the number of neurons of a fully-connected layer and k_i the kernel size of a convolutional layer.

Layer	Output size	Number of weights + biais
conv1	$32 \times 32 \times 32$	$2\,400 + 32$
pool1	$16 \times 16 \times 32$	0
conv2	$16 \times 16 \times 64$	$51\,200 + 64$
pool2	$8 \times 8 \times 64$	0
conv3	$8 \times 8 \times 64$	$102\,400 + 64$

Layer	Output size	Number of weights + biais
pool3	$4 \times 4 \times 64$	0
fc4	$1 \times 1 \times 1000$	$1\,024\,000 + 1\,000$
fc5	$1 \times 1 \times 10$	$10\,000 + 10$

We can observe that by diving deeper into the neural networks, the feature maps are smaller and smaller. They, therefore, contain more compressed and high-level information. In the same time, as the need for high-level extraction is needed, the number of weights also increases to match this need.

11) What is the total number of weights to learn? Compare that to the number of examples.

The total number of weights of this architecture amounts to 1 191 170. The CIFAR10 dataset has about 50k images, which is relatively small relative to the number of model parameters. To counter this problem, we can use data augmentation technics. Data augmentation artificially increases the number of available examples, by applying random transformation to them. This can be crucial when the training set is small, like for CIFAR10.

12) Compare the number of parameters to learn with that of the BoW and SVM approach.

From the previous practical 1.c, we have seen that the best performing SVM model was a one-versus-one strategy. In one-versus-one strategy, we use $\frac{n(n-1)}{2}$ classifiers with n the number of class (which was 15 in the previous practical). Each classifiers have $m + 1$ learnable parameters (because it learns an offset (scalar) and an hyperplane defined by its normal vector (m dimensional vector)), with m the size of the word dictionnary (with $m = 1001$ in the previous practical). Therefore, this SVM approach has this number of parameters:

$$nb\_classifiers \times nb\_parameters = \frac{15 \times 14}{2} \times 1002 = 105\,210$$

The is, of course significantly less compared to the 1 million parameters of our AlexNet-like architecture. Though, it is worth noting that the latter architecture will also produces better results. The general state-of-the-art is heading in the direction that higher number of parameters enable better performances. This is especially noticeable in the Natural Language Processing field with modern Large Language Models that are trained with over a trillion parameters.

14) In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data)?

The function `epoch` is called differently to compute the training and testing loss/ accuracy. While testing, no optimizer is provided. The `epoch` function, then, ran the model differently. More specificaly, some layers behave differently in testing or training mode. Some layers like dropout are not use at all during testing. This particularity can significantly change the accuracy result.

16) What are the effects of the learning rate and of the batch-size?

From a theorical point of view, as discussed above in question 10) of part 1, the batch-size is a balance between steps stability (i.e. good estimation of the gradient), its computationnal cost, and its memory usage.

As discussed in question 11) of part 1, above, the learning rate controls the step size of the gradient descent. It, intuitively, controls the speed at which the weights are updated, and how quickly the model is adapted to the problem. A larger η implies larger step in the direction of the optimal solution and requires fewer training epochs.

Most of the time, larger step will lead to a faster convergence. But this fact is not always true. A too large η value might encourage steps that go beyond the optimal solution. If those missed steps are repeated several time, it can lead to an oscillating or diverging learning. This hard-to tune specificities makes the learning rate η one of the most important hyperparameters of a neural network.

On the other hand, small η can also hamper the model's performance. Beyond the increase of required training epochs, small η can stuck the gradient descent in local minima, and therefore prevent the algorithm from finding an optimal solution.

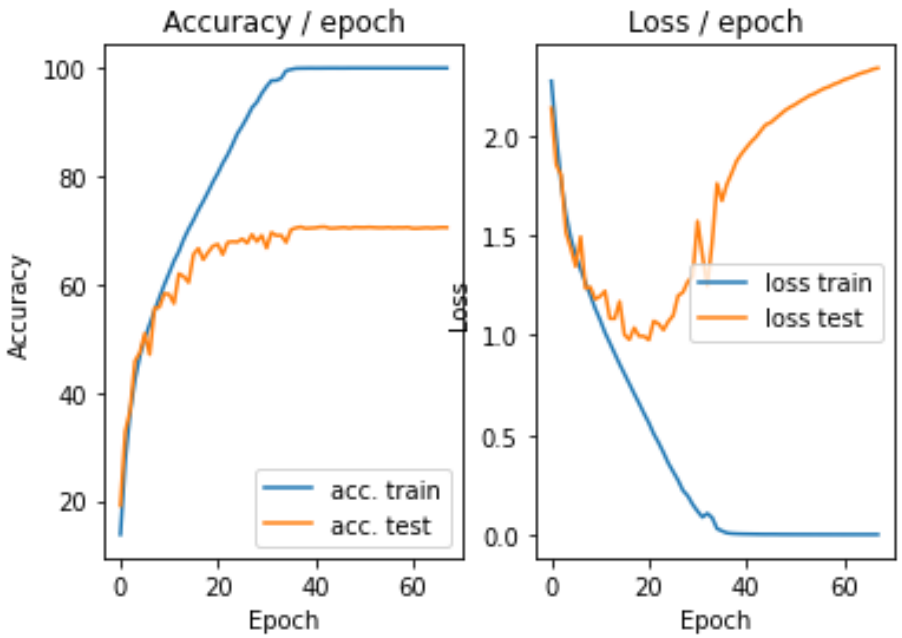
As discussed in part "2 - Implementation" of practical 2ab, above, we can observe, the same conclusion in a practical manner in the context of this problem.

17) What is the error at the start of the first epoch, in train and test? How can you interpret this?

At the start of the first epoch, the training and testing accuracy are, respectively 10.2% (i.e. random performance on 10 classes) and 25.0%. This phenomenon hold the name of under-fitting. It designates the use of a model that is too simple or badly fitted. Under-fitting then struggles to correctly approximate the target function. As the training has not yet started, an under-fitting model is to be expected.

18) Interpret the results. What's wrong? What is this phenomenon?

Here is what the training looks like after a few iteration:

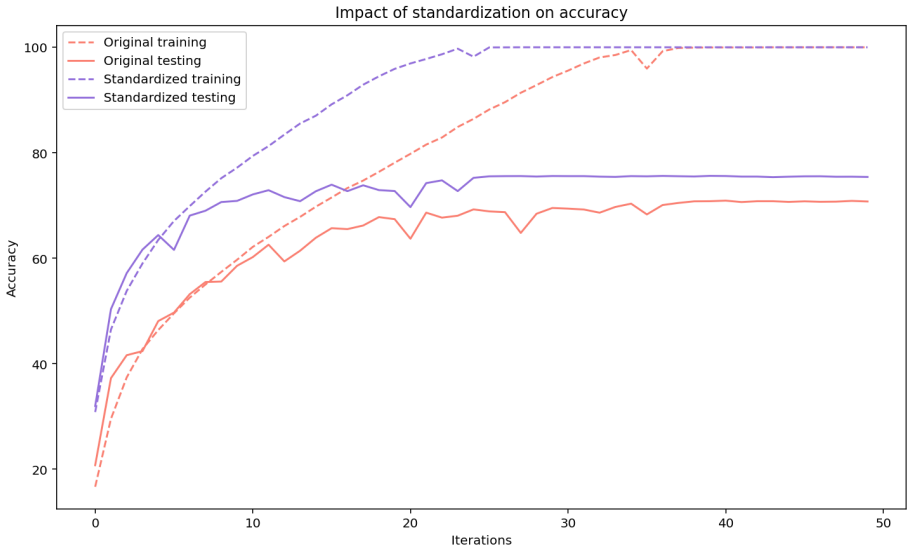


We can clearly see the testing loss starting to increase after about twenty iterations, while the training loss is still decreasing until its accuracy reached 100%. Our model, therefore, performs way better on the training set than the testing set. It has learned the noise of the training data, by heart. This behavior is a clear sign of over-fitting.

The over-fitting highlights a struggle to generalize. It therefore hampers the learning. Indeed, in our example, the testing accuracy is blocked around 65% for the last couple iterations. These results can surely be improved by exploring generalization tools.

3 - Results improvements

19) Standarization: Describe your experimental results.



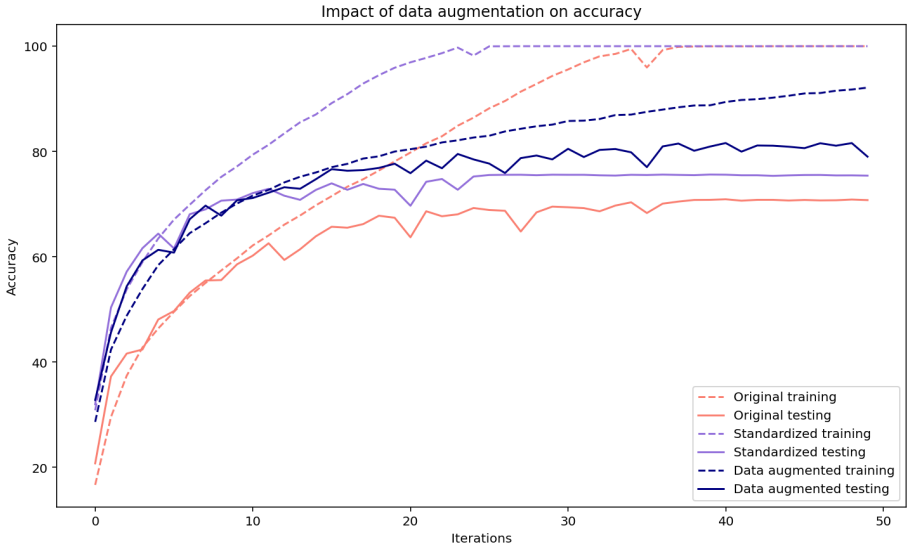
From this graphic, the performance gain is clear and constant. From the first iteration of both the training and testing to the 50th, the standardized approach is benefic. At the end, the standardized approach has a testing accuracy of 78.1%, against a 72.7% accuracy for the original model. This is a 5.4% absolute improvement. It justifies the use of standardization.

20) Why only calculate the average image on the training examples and normalize the validation examples with the same image?

It is necessary to keep the same average image for the standardization of both the testing and training datasets. Let's prove it by reductio ad absurdum. Suppose we are computing an average image for both the training and testing datasets, and that those two average images differ. We can then use these two average images to shift all the images of respectively the training and testing set. But the the images of those two sets would have been shifted to different domains. In this scenario, a model trained on the training domain would then struggles to generalize to another domain. This issue justifies to use a unique average image to standardize both the training and testing set.

It is also important to compute the average image only on the training dataset to respect the testing/ training dataset seperation.

22) Data augmentation: Describe your experimental results and compare them to previous results.



The "Data augmentation" approach is based on the previous approach (i.e. with the standardization) and add a data augmentation layer.

With the addition of this regularization, the testing accuracy is not improved in the few first iterations, but only after about 12 iterations. We can imagine that the diversity of samples added only starts to take effect at this stage. Before this point, the training has not yet seen enough time each sample to make a difference. After 50 iterations, the testing accuracy is improved by an absolute score of 7.8% (78.1% for the standardized approach vs 85.9% for the data augmentation). We will see later that this is actually the regularization approach that provided the best absolute and relative improvements.

Interestingly enough, the training accuracy actually decreases when adding data augmentation. This is due to the increased difficulty to learn by heart the enhanced training dataset with its learger number of sample.

23) Does this horizontal symmetry approach seems usable on all types of images? In what cases can it be or not be?

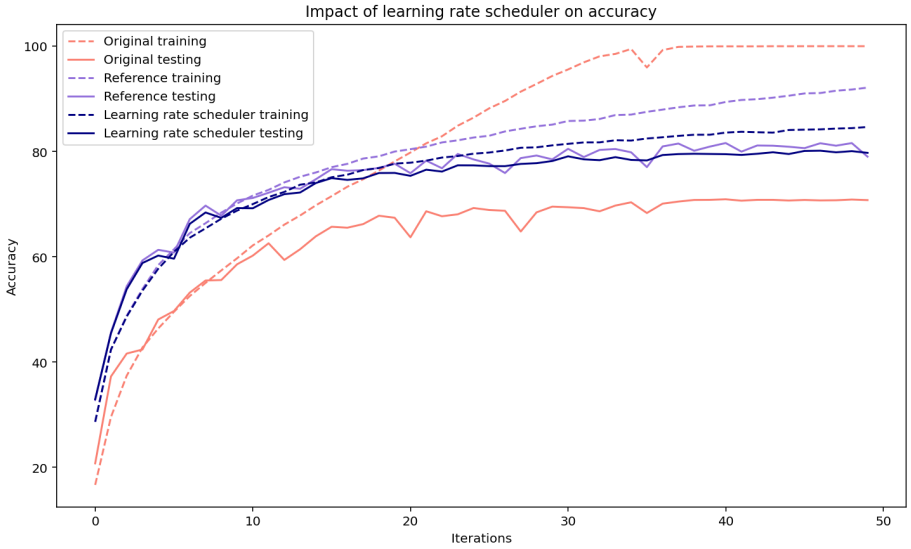
The data augmentation technics are specific to the problem targeted. It is important that the data augmentation technics do no change the label of an image. Indeed, a horizontal symmetry cannot be used in every case, especially when the horizontal information matters. An example in which a horizontal symmetry used as data augmentation won't help is in the context of a classifier that tries to distinguish right hands from left hands.

24) What limits do you see in this type of data increase by transformation of the dataset ?

Data augmentation have some limits, even if they provide some performance improvements. First, augmented datasets will carry the biases of the existing datasets. It is, therefore, necessary to study which data augmentation approaches can be used in order to prevent the replication or amplification of such biais.

To the extrem, with advanced data augmentation technics incorporating GAN, it is necessary to assess the quality of the enhanced datasets and make sure, the augmented image still match their label.

26) Learning rate scheduler: Describe your experimental results and compare them to previous results, including learning stability.



The reference in the plot corresponds to the accumulation of all the previous regularization methods (i.e. data augmentation and standardization). The "learning rate scheduler" approach is based on the reference approach and add a learning rate scheduler.

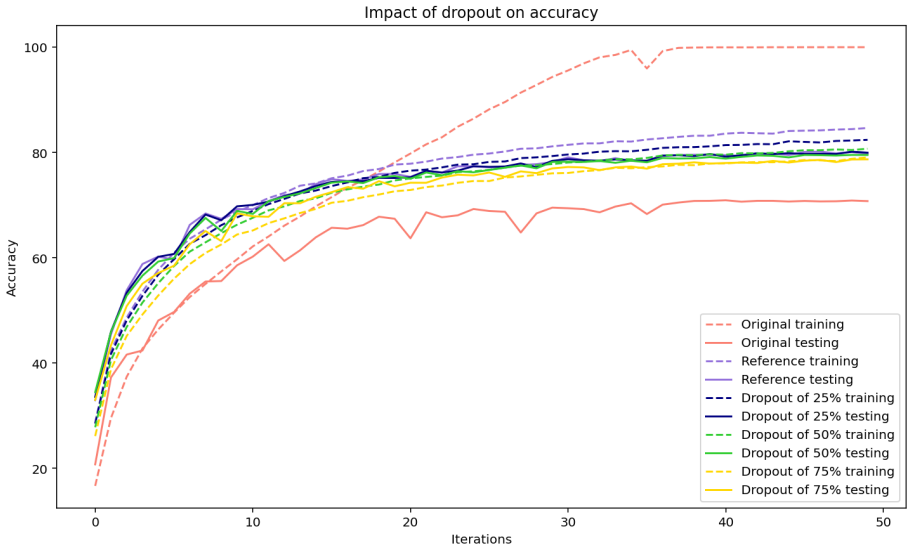
For the new model with a learning rate scheduler, both the training and testing accuracy (84.44% and 80.06% respectively) are below the previous reference model (81.60% and 91.77%). On the other hand, the learning stability has greatly improved. The accuracy curve is less shaky. By continuing the experimentation with a larger number of iterations, the learning rate scheduler accuracy eventually overcomes the reference one.

27) Why does this method improve learning?

Previously, in the question 11 of part 1, we have seen that it can be quite hard to set a good learning rate. A learning rate is specific to a targeted problem and no rule of thumbs exist to choose a fixed learning rate. A badly chosen learning rate can hamper performance of the model in several ways including slow convergence, diverging gradient descent or gradient descent getting stuck in local optimum. Even in best scenarios, the gradient descent hovers around a minimum at an average distance proportional to the learning rate. It is, therefore, impossible to combine precise optimum and fast convergence with a fixed learning rate.

The learning rate scheduler solve elegantly those issue.

29) Dropout: Describe your experimental results and compare them to previous results.



The reference in the plot corresponds to the accumulation of all the previous regularization methods (i.e. learning rate scheduler, data augmentation and standardization). The "dropout" approach is based on the reference approach and add a dropout layer. Three different dropout threasholds are tried: 25%, 50%, 75%.

From this experimentation, it seems that the dropout layer did not improve the accuracy of the model, no matter the dropout rate chosen. In fact, all testing accuracy between the reference and the dropout approaches are more or less the same. The only approach that stand out from the crowd is the dropout layer with a 75% dropout rate. The latter approach provides a too severe regularization, which actually hampers the performance. This lack of improvement might be explained by a redundance of its effect with the other regularization approaches

implemented.

We will not keep the dropout layer in latter experiment.

30) What is regularization in general?

The high number of parameters of deep neural networks enable them to learn complex schemes as well as the noise of the training dataset. This issue is specially under a limiter number of samples, and is called over-fitting. Another common issue of machine learning model is the under-fitting, i.e. training a model that is too simple to fit the data. In the middle of those two extrem, there is a well-fitted model that learned and only learned the important information of the training data. This well-fitted model will then greatly performs on any unseen data. Finding such a model is the goal of any regularization approach.

31) Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it?

With significant computing capabilities, an effective way to "regularize" a model is to average the predictions of several trained neural networks[2]. This approach is called "ensemble of neural networks". In practice, this approach is naturally quickly limited by the need to fit and store multiple models, especially with large neural networks.

Though, a single model can be used to simulate the ensemble of a large number of different networks with different architectures in parallel. It can be achieved by randomly dropping out nodes during training. This is called "dropout" and offers a very computationally cheap and remarkably effective regularization method. It both reduces the overfitting and improves the generalization error in deep neural networks of all kinds.

Without dropout, neurons in a neural network may learn to co-adaptation with other neurons. This pattern seems to reinforce overfitting as these co-adaptations do not generalize well on unseen data.

But dropout layers have the effect of making the training process noisy. By randomly dropping neurons, neurons's co-adaptation are harder to emerge, which in turn makes the model more robust.

32) What is the influence of the hyperparameter of this layer?

The main hyperparameter of the dropout layer is called the "dropout rate". It indicates the frequency at which the input units shoud be randomly set to 0. The other input units are then scaled up by $\frac{1}{1-rate}$ to keep a constant sum over all units.

[2]Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

By increasing the number of nodes that are droped out, we artificially augment the noise during training. As evocated above, this noise breaks co-adaptations and improve the robustness of the model.

33) What is the difference in behavior of the dropout layer between training and test?

The dropout layer has indeed a different behavior in training and testing. The dropout layer take effect only during training. On the other hand, during the testing period, the dropout layer is fully desactivated. This means that no values are dropped during the inference.

Generalization conclusion

Methods	Explanation
Standarization	Shifts traning and testing samples by an average sample to better condition the gradient descent.
Data augmentation	Randomly tweaks samples to artificially increase dataset's diversity in order to reduce over-learning of samples during training and help generalization to new datasets.
Learning rate scheduler	Adapts the learning rate during learning in order to transition from quickly converging large steps to fine-grain steps in the neighborhood of the optimum.
Dropout	Randomly nullify neurons to mimic an ensemble of different neural networks and, therefore, prevent the learning of the noise at computationally cheap cost.
Batch normalization	Normalizes layers' input batch-wise inside the neural network to solve the internal covariate shift.