Neural networks: how to monitor the learning process

LECTURE 2, 19/12/2023 (1H)

Abstract. We will start with a small recap from the previous lecture and we will then dig into how to prepare the data, initialize weights and run the network. We will discuss the batch normalization and will present tips and tricks that reduce the risks of overfitting and improve the network performance, like regularization, L2 dropout and data augmentation. We will then introduce one example, Resnet, which is often used in meteorological applications.

Keywords: Preprocessing, weight initialization, batch normalization, regularization and L2 dropout, loss functions, data augmentation. Overview of some checks to perform for monitoring the CNN algorithm, using one example. Resnet, fine-tuning, transfer learning.

Contents:

Credits for the content in this lecture:

Almost all the material from this lecture is a collection of concepts, explanations, images and visualizations from other more technical computer engineering lectures, done with the aim of extracting and offering to the students of the master program of climate sciences only the minimal necessary information to achieve an beginner level of understanding of machine learning methods applied to computer vision. We extracted learning material from many currently available resources present on the web that we want to fully acknowledge here. All what presented here can be found in a more extended, technical and detailed way in the original sources listed here:

- Stanford course on convolutional neural network for computer vision
- Article1 and article2 of Ketan Doshi on batch normalization
- Article on techniques to avoid overfitting by David Chuan-En Li
- article on transfer learning and fine tuning and references therein from Victor Chaba

We deeply thank all the people that contributed to share this knowledge and make it available online, so that other people in the world can benefit from it.

2.0 Neural networks

Until now, we have been discussing a linear classifier that computes the scores associated to each input image x as s = f(x,W) = Wx, with W the matrix of weights. To build a neural network, it is enough to start with this system and introduce a non-linearity. One example could be:

$$s = W_2 max(0, W_1 x)$$

W1 can be a matrix for of [50x3073] transforming the input image in a vector of dimension 50, on which then the non-linearity operates elementwise. Then, W2 can be another matrix, of size [10x50] providing the final numbers that we interpret as class scores. Both W1 and W2 can be learned with stochastic gradient descent via backpropagation. Between the output of W1 and the input of W2, the nonlinearity (not represented in the figure) operates removing all negative values from the result of W1x. If, as in the example above, the output of a gate (W1x) becomes the input of another gate (W2), we are dealing with a neural network and gates normally are called neurons. It is necessary that neurons are connected not in a cyclic form, i.e. the output of W2 cannot become again the input of W1.
In regular networks the neurons are normally organized in layers, and one of the most common types of layer is the fully connected layer, where neurons of adiacent layers are connected fully pairwise but there are no connections among neurons of the same layer.

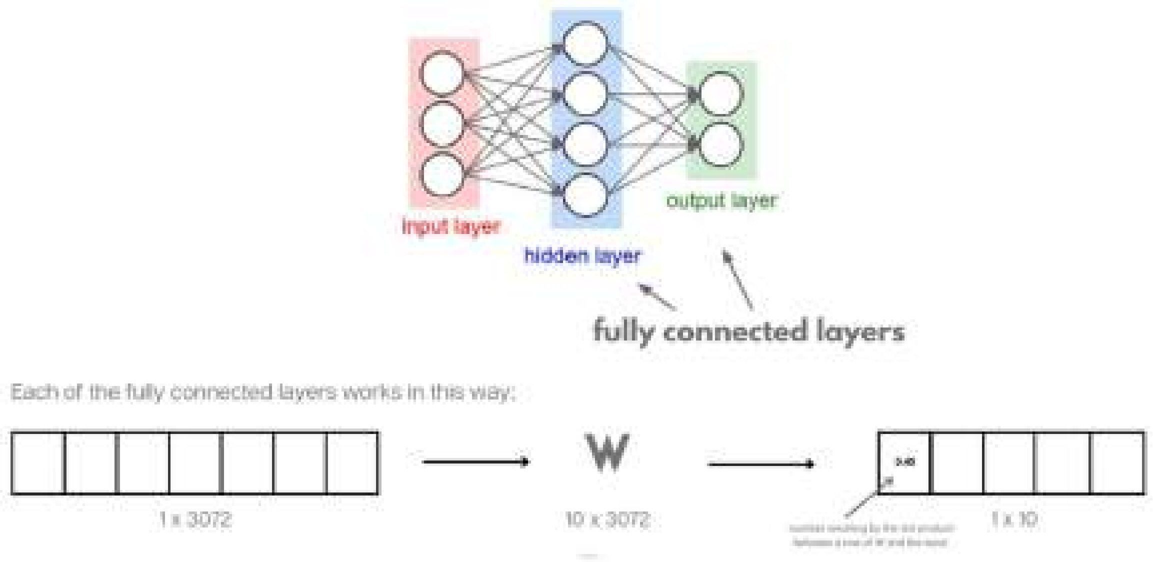

Figure 1.16: 2 layer fully connected neural network. Figure re-elaborated based on the material of the lecture series of the Stanford University's CS231n course.

Activation functions. The non-linearity introduced is often called "activation function". These functions take the input number and apply some type of mathematical operations on it. Various possibilities exist for its functional form (see figure 1.17):

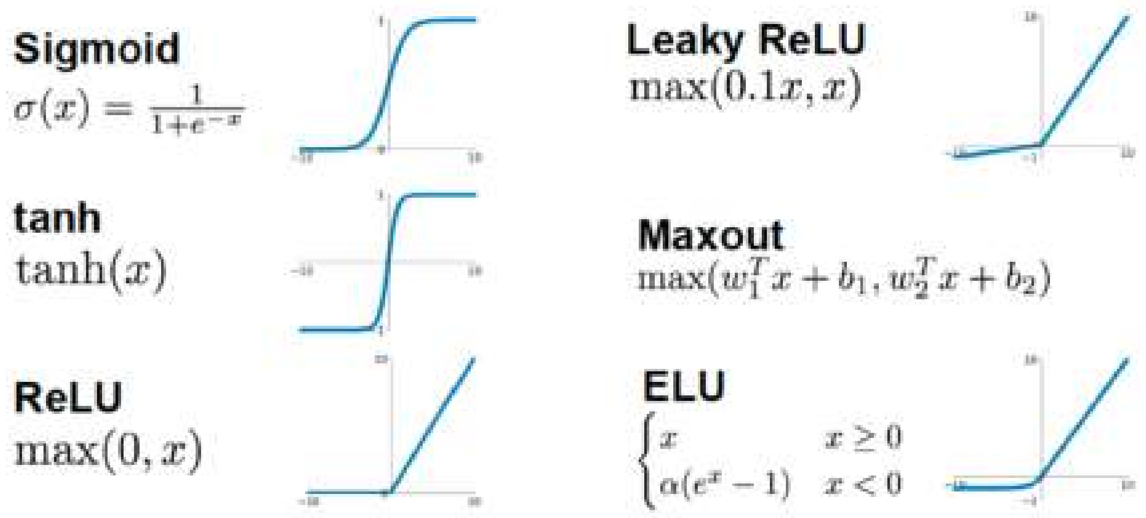

Figure 1.16: Representation of the most commonly used activation functions, image taken from the lecture series of the Stanford University's CS231n course.

It is important to notice that the gradient of the activation function will play a role in backpropagation, because it will be multiplied by the local gradient. Let's briefly look at each of them:

- **SIGMOID**: this function pushes all the input values in the range [0,1], with large negative values becoming 0 and large positive ones becoming 1. Currently this function is not very used because it tends to kills the large gradients by making them very small, and not letting the signal go through. Moreover, its output is not centered around zero and this introduces a zig-zag behaviour during backpropagation because the gradients will become either all positive or all negative.
- **TANH**: Tanh pushes values between -1 and 1. Also in this case the activation tends to reduce the gradients and cause saturation, but the output is zero centered causing less issues in backprogation.
- **RELU**: RELU stands for Rectified Linear Unit and it is one of the most used activation functions. It returns the maximum between 0 and the input value. Krizhevsky et al. showed that it greatly accelerates convergence and it is a simpler operation (thresholding) compared to sigmoid/tanh, However, it can happen that it causes an update in such a way that the gate will never activate again during training. A good setting of the learning rate can greatly reduce the problem.
- **LEAKYRELU**: It is a modification of the ReLU attempting to solve the problem of the dying gates by setting the function for values smaller than zero not to zero but to a small positive slope.
- **MAXOUT**: Maxout was introduced by Goodfellow et al., and computes the output of the function

$$W_1^T x + W_2^T x + b_2$$

ReLU and LeakyRelu are special cases of this function and maxout has all the pros of the relus and no drawbacks.

Overall, we recommend to use the ReLU and train with attention the learning rate, don't use sigmoid.

Neural network architectures. A regular neural network receives an input and transforms it through a series of hidden layers made of neurons. Each neuron is fully connected to the neurons of the previous layer, but it does not share any connection with the other neurons of his layer. The last connected layer is called the output layer and when the network's goal is to perform a classification, it contains simply the class scores.

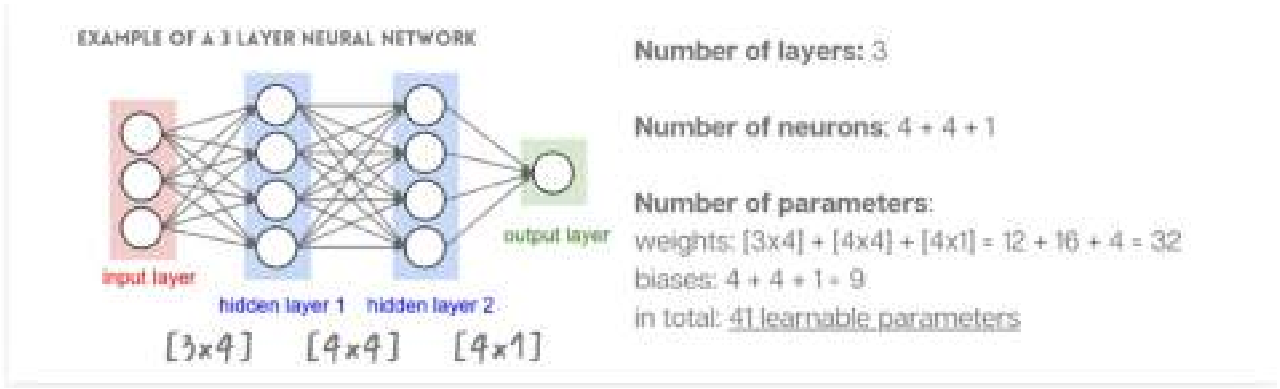
EXAMPLE OF A 3 LAYER NEURAL NETWORK
input layer
hidden layer 1
hidden layer 2
output layer
[3x4]
[4x4]
[4x1]
Number of layers: 3
Number of neurons: 4 + 4 + 1
Number of parameters:
weights: [3x4] + [4x4] + [4x1] = 12 + 16 + 4 = 32
biases: 4 + 4 + 1 = 9
in total: 41 learnable parameters

Figure 1.17: Example of a 3 fully connected neural network, re-elaborated based on the notes from the lecture series of the Stanford University's CS231n course.

A N layer neural network does not include the input layer and its output layer does not have an activation function (unlike the other layers in the network). The size of the neural network is given by the number of neurons in the layers of the network (as said before, excluding the input layer) or alternatively by the number of parameters. A network with a single layer can represent any function (see more on this in chapter 6 of the book "Deep learning" from Goodfellow et al.) but experience shows that adding more layers get better performances. For this reason, people built networks with more than just one layer. Adding more layers gives the capacity to represent more functions, but it makes it easier to overfit the training data. Generally, larger networks work better than smaller ones thanks to their higher model capacity, but methods like regularization have to be adopted to deal with the problem of overfitting.

Based on what we discussed, you can easily foresee that regular neural networks do not scale well with images. In our example we considered a very small image of 32x32x3=3072 elements. A single fully connected layer for this input would have the size of 1x3072 and 3072 weights. However, typically images have much larger sizes, like at least 200x200x3 = 120000 elements and this number would be a huge number of weights to manage for the network, only in the first hidden layer, and weights will add up quickly if you add more layers. In the next lecture, we will introduce convolutional neural networks that can deal better with the image dimensionality.

2.1 Considerations on the activation functions

In the previous lecture, we saw that activation functions are included in neural networks for learning complex patterns in the data. These functions take an input from the previous neuron and decide, what input can be transmitted to the next neuron and what needs to be killed.

There are **two main reasons** why we need them in artificial neural networks:
1) They help in keeping the output of a neuron within a certain range we decide based on our needs, avoiding computational issues caused by numbers growing to extremely large values in the network.
2) They add non-linearity to the network. If the model needs to learn non-linear patterns, like for example in classification tasks, then specific non-linear layers need to be added to the network.

We also saw that neural networks are usually trained using gradient descent, via backpropagation of the gradients with the chain rule. During backpropagation, gradients get multiplied with the activation functions. If the activation functions re-scale the input into a range of values between 0 and 1, this means that values of the gradients get strongly reduced. In general, gradients tend to vanish because of the depth of the network and this problem goes under the name of "vanishing gradient problem".

2.2 Normalization of the input data

Typically, inputs of neural networks are normalized to zero mean and standard deviation equal to 1. Why we do this? normalization brings all the features of the input on the same scale. If we don't do this, what will happen? Imagine that we have in particula two features having very different scales. Also the weights associated to the features, since the network output is the linear combination of the feature vectors, will differ very much in scale, simply not to drown the smallest feature.
If we then look at what happens in the backward propagation, we can see that during gradient descent, the network will have large updates on the directions with the largest weights, and much smaller updates on the direction of the smallest feature, resulting in a gradient that oscillates a lot and needing more steps to converge to the minimum . If, instead the features are normalized, the contributions to the gradient during gradient descent will be of the same order, generating a more homogeneous descent (Figure 2.1). Moreover, not to make the gradients shift in a particular direction, we need to get an output of the activation function that is symmetrical around zero. In order for this to happen, we need some preliminary pre-processing of the input data.

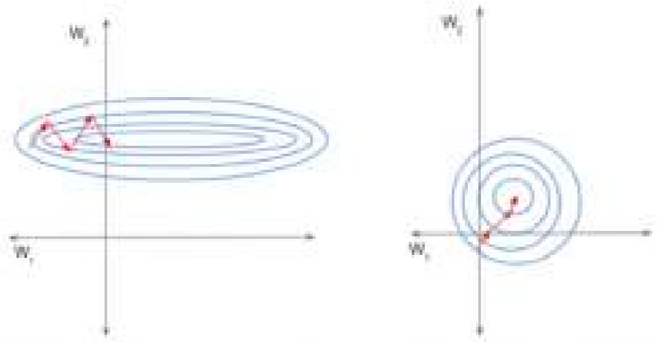

Figure 2.1: a) Example of gradient descent for two un-normalized features and b) gradient descent when the features w1 and w2 are normalized. The graphics are taken from the wonderful article on the batch normalization from Ketan Doshi

2.3 How to pre-process your data in practice

If we now consider the data matrix of input as [NxD], where each data example is a row, we can think of 3 main types of pre-processing:
1) **Zero-centering**. It is the operation of subtracting the mean from each of the individual features of the data. It concides with the operation of centering the data around the origin. With images, it is common to subtract a single mean value from all pixels of the image (as done in AlexNet), or to do this operation for each of the RGB channels (as done in VGGnet).
2) **Normalization**. This is the process that brings data dimensions all on the same scale, approximately. It is obtained by first zero-centering the data, and then by dividing each dimension by its standard deviation. In case of images, this type of pre-processing is not really needed because pixels scales are relatively equal (in the range 0-255).
3) **PCA and whitening**. This is another form of pre-processing. Data are centered as described above, but in this case we then calculate the covariance matrix that will give information about the correlation structure of the data. The next step is to apply the singular value decomposition, that is a factorization of the matrix that allows us to obtain the eigenvectors and array of singular values. The transformation of data covariance into the identity matrix corresponds to squeeze the data in an isotropic bubble (Figure 2.2). We can then re-project the data on the obtained eigenbasis to decorrelate them and in this way, we can reduce the dimensionality of the data by selecting only the first eigenvectors and discarding the other dimensions where data has less or no variance. This is also called the Principal component analysis dimensionality reduction and with this operation, we will obtain a reduced dataset that can sometimes be used for training the classifier on a reduced datasets. Finally, the last step is the whitening, which means simply to divide every dimension in the eigenbasis by the eigenvalue and its goal is to normalize the scale. One drawback of this transformation is that it can greatly amplify the noise in the data because it stretches all the dimensions (including the non relevant ones, where the variance is basically only noise) to be equal to the size input. This might be a problem when we add small constants (es 10^-5) to prevent divisions by zero values in the network.

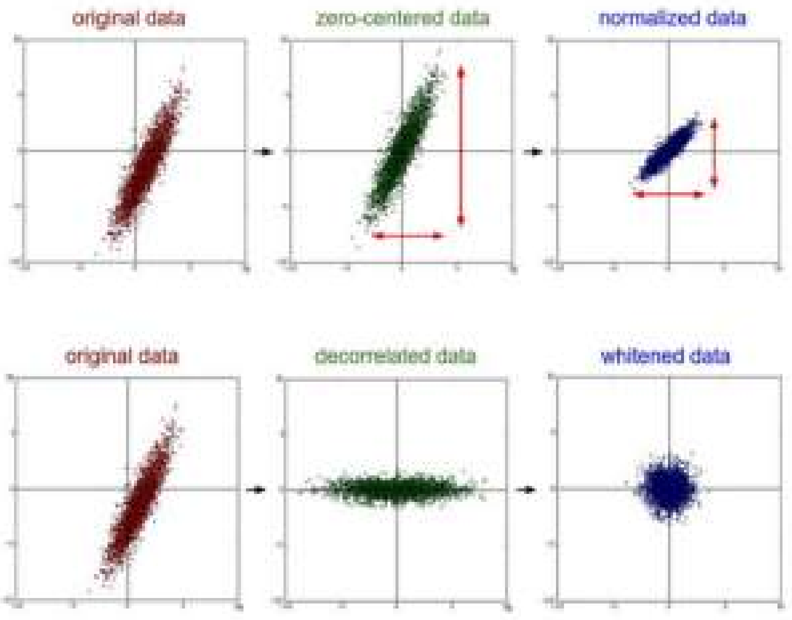

Figure 2.2: The pre-processing methods (figure from the slides of the lecture series of the Stanford course on Computer vision with CNN.

Data preprocessing can solve some of the problems mentioned above (the vanishing or the shift of the gradient) only in the first layer, but the mean will not be zero for the subsequent layers. Typically, pre-processing is more efficient and easier to do at the beginning, on the entire dataset, instead of applying it to smaller parts of it (batches).

2.4 Weight initialization

Before starting the training of the neural network, it is necessary to initialize the weights. One first guess idea that might come into your mind is to initialize all the weights to zero, but you would realize that with this choice, the network will not learn because there is no simmetry breaking: all neurons will do the same thing, and they will all give the same gradient, so they will be updated in the same way.
Another possibility could be to give as initialization a set of small random numbers. In this case the simmetry would be broken, but the network might not work for deep architectures. Figure 2.3 shows the distribution of the activations (inputs to the layers) and we can see that only the first layer has zero mean and a variance larger that 0.2, then the subsequent layers inputs display a variance shrinking to zero and collapsing to all zeros in the end. All activations become zero, and gradients proportional to the values of the weights, will be attenuated.

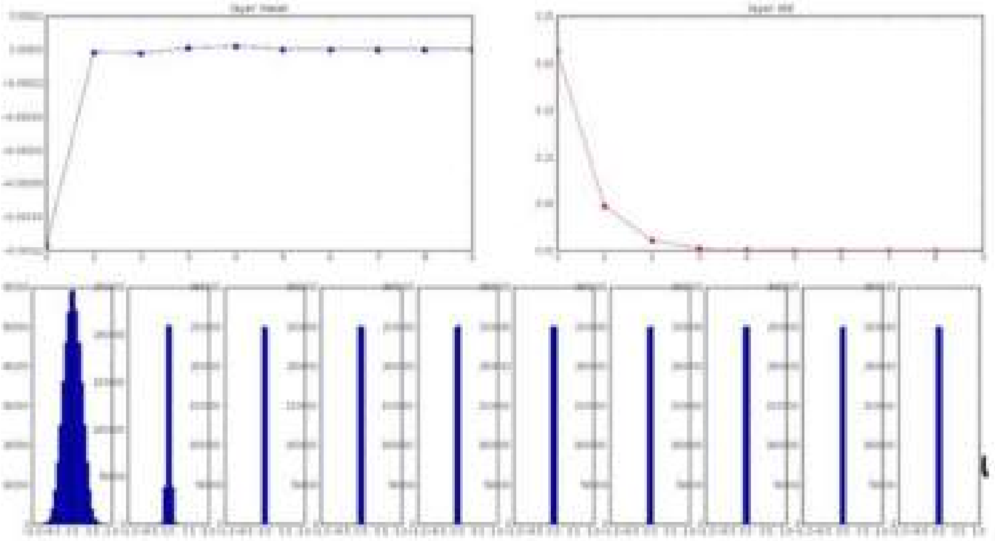

Figure 2.3: Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero.

We can state the problem in a more general way, that is the following: **weights are selected from random distributions and scaled so that the variance of the input and output layer is maintained constant (**Narkhede et al., 2021).
Various authors followed this variance scaling based initialization and provided scaling factors for the variance, so that the variances in input and output could stay constant.

The first answer to such a question came from a paper: Glorot and Bengio adopted a value of the variance v = 1/N, where N is the number of nodes feeding into the layer and assumes that activation functions are linear. This initialization is called "**Xavier initialization**" and achieves faster convergence and better accuracy. However, when used with nonlinear ReLU activation functions, it kills half of the neurons and the variance gets smoothed (Figure 2.4).

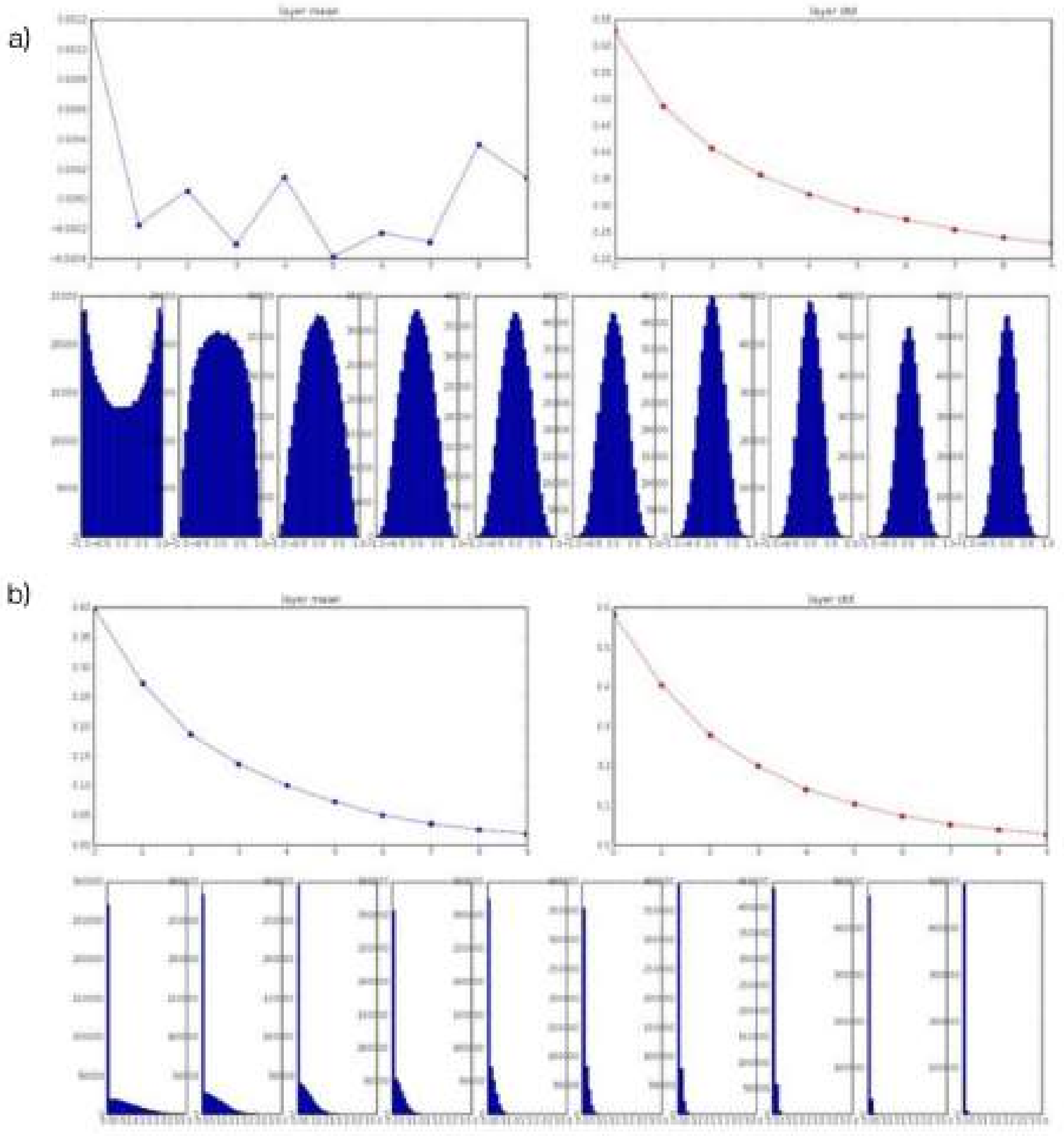

Figure 2.4: Example of the behavior of the mean (top left), variance (top right) and distributions (bottom) for Xavier initialization on a) a linear activation and b) a ReLU non linear activation.

To account for non linearities, He et al. modified the scaling factor from Glorot and Bengio to v = 2/N, for making it work well with Relu activation functions. This initialization is known as "**He initialization**". However, weight initialization is an active field of research and different approaches can be found in the comprehensive review from Narkhede et al., 2021.

2.3 Batch normalization

Batch normalization became an essential part of deep neural networks because it speeds up training and improve accuracy of results.
Batch normalization comes in the network as an **additional layer** that is usually added to other blocks of the architecture, like the convolutional or the fully connected layer.
Let's now look at how the batch normalization acts on the network layers. Imagine to have a network with a series of hidden layers. Each activation from the previous layer in such a network becomes the input of the next layer, so for each layer we can identify an activation (input) coming in and an activation output coming out of the selected layer.
What batch normalization does is to **normalize the activations from each previous layer**, so that when the activation becomes the input of the next layer, it is normalized.
We can think of this operation as an additional layer that is put between two hidden layers, taking the output of the first hidden layer, normalizing it and passing it to the next hidden layer (Figure 2.5)

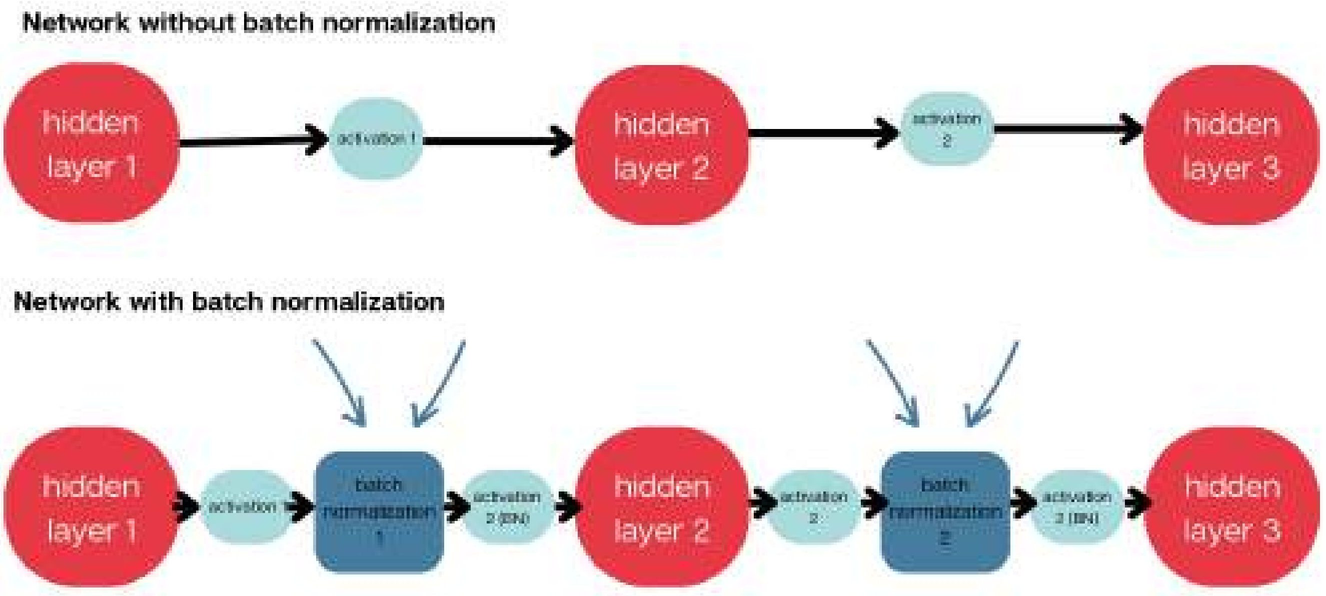

Figure 2.5: Example of network without and with batch normalization. The batch layer normalizes the activations from the previous layer before they reach the subsequent layer.

Let's now see how the batch normalization does the normalization, by first considering the parameters of the batch normalization layer:

- two <u>*learnable*</u> parameters (**beta** and **gamma**)
- two <u>*non-learnable*</u> parameters (**mean moving average** and **variance moving average**)

During training feedforward phase, we provide as input a mini-batch of data, i.e. a subset of the whole input dataset, for example imagine we take M samples of the N features we have, and for each feature we have an activation vector.

In the layer the operations the batch does to prepare the output are:

1. calculating the mean and standard deviation of the activation arrays.
2. normalization of the activation arrays with the mean and the standard deviations that have been calculated
3. scaling and shifting of the normalized activation arrays with the beta and gamma parameters

$$\mu_B = \frac{1}{M}\sum_{1}^{M}(A_i)$$

$$\sigma_B = \sqrt{\frac{1}{M}\sum_{1}^{M}(A_i - \mu_B)^2}$$

$$\hat{A}_i = \frac{A_i - mu_B}{\sigma_B}$$

$$BN_i = \gamma \cdot \hat{A}_i + \beta$$

In addition to these, used to generate the output, the layer also stores a running count of the exponential moving average of the mean and variance, obtaining an EMA at the end of the training, that we will then use in the inference phase (Figure 2.6)

$$\mu_{mov} = \alpha\mu_{mov} + (1-\alpha)\mu_i$$

$$\sigma_{mov} = \alpha\sigma_{mov} + (1-\alpha)\sigma_i$$

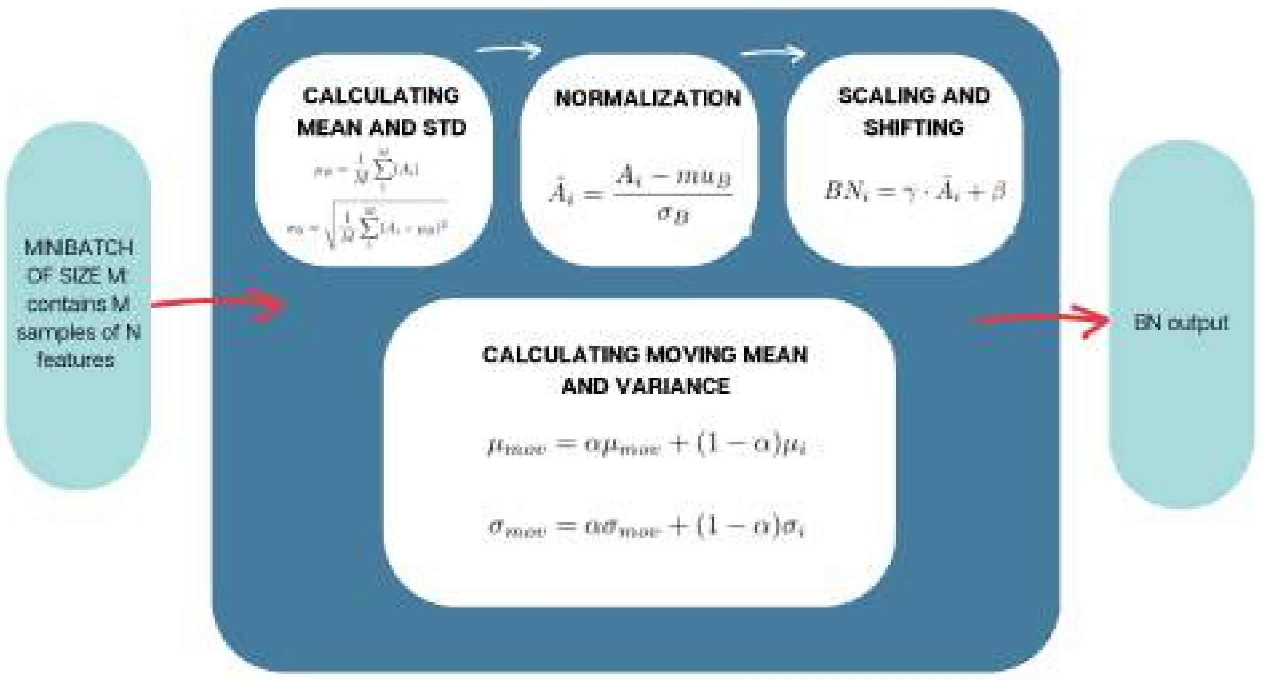
MINIBATCH OF SIZE M: contains M samples of N features
CALCULATING MEAN AND STD
$\mu_B = \frac{1}{M}\sum_{1}^{M}(A_i)$
$\sigma_B = \sqrt{\frac{1}{M}\sum_{1}^{M}(A_i - \mu_B)^2}$
NORMALIZATION
$\hat{A}_i = \frac{A_i - mu_B}{\sigma_B}$
SCALING AND SHIFTING
$BN_i = \gamma \cdot \hat{A}_i + \beta$
CALCULATING MOVING MEAN AND VARIANCE
$\mu_{mov} = \alpha\mu_{mov} + (1-\alpha)\mu_i$
$\sigma_{mov} = \alpha\sigma_{mov} + (1-\alpha)\sigma_i$
BN output

Figure 2.5: Representation of the operations done in the batch layer. The figure is based on the figure presented in the article on Batch normalization explained from Ketan Doshi.

It is important to note that the scaling and shifting is what allows the batch to shift the output to a different mean and standard deviation, unlike what can be done with the input, where all input layers need to have zero mean and unit variance. Moreover, the parameters beta and gamma are learned in the training process, like all weights and the batch is optimizing them during training to fit the values to those giving the best predictions.

During inference, after the training, the activations flow in the same architecture. In this case, in the batch layer, the normalization is done using the two moving average parameters, that have been calculated and stored during the training. While ideally we could have calculated and saved the mean and variance for all the data in training, this operation would have been very expensive. Moving average is a good proxy and it is more efficient because the calculation is incremental.

Why does batch normalization help in training neural networks? There are two main explanations for this:

1) **It reduces the internal covariate shift.** The internal covariate shift is the change in the distribution of network activations due to the change in network parameters during training. The problem of covariate shift occurs when the model is trained with data having a very different distribution with respect to the data which are used for inference. To work on the new data the model needs to re-learn some of the output target functions, slowing down the process because each layer tries to learn from a constantly shifting input, taking longer to converge. Batch normalization stabilizes the shifting input by optimizing the beta and gamma parameters, thus speeding up the training (see Figure 2.6)

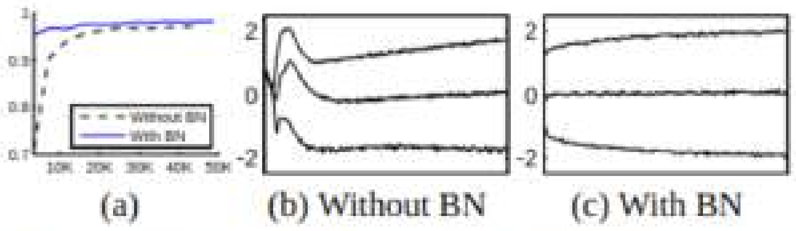

Figure 2.6: from the batch optimization paper (Ioffe and Szegedy, 2015): (a) The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy. (b, c) The evolution of input distributions to a typical sigmoid, over the course of training, shown as {15, 50, 85}the percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift

2) **It smooths the loss function and the gradient.** Batch normalization smoothens the loss landscape by changing the distribution of the weights of the networks. In this way, steps of gradient descent can be larger in a given direction and learning rate can be higher. If you want to know more about this aspect, check the paper from Li et al., 2018.

2.4 Methods to avoid overfitting

Overfitting happens in machine learning algorithms when the model that we develop performs extremely well on the training data, but fails to generalize to a different, never seen before, dataset.
Various approaches have been presented in literature to try to reduce or solve the problem of overfitting. We already talked about cross-validation, which is indeed a method to reduce overfitting. Other methods are:

Data augmentation
One very simple idea to fight overfitting is to increase the dataset in input. In classification tasks, this can be easily done by manipulating the input images using some transformations like rotation, rescaling or shifting. We saw an example of such transformations in figure 1.2.

L1/L2 regularization
Regularization is a way to prevent the model from learning a too complex model. Regularization typically penalizes the cost function. Depending on the type of regularization, with some peaky weight vectors or by generating sparse weight vectors. In L1 regularization, we add to the cost function a term

$$\lambda|w|$$

which allows all the weights to decay to zero. It penalizes the sum of the absolute values of the weights, and it is robust to outliers.
L2 regularization instead we add the term

$$\frac{1}{2}\lambda w^2$$

which penalizes the sum of the square values of the weights (the peaky weight vectors) and preferring the diffuse ones, but it is less solid to outliers. Usuallt the two normalization can also be combined in the Elastic net regularization (Zou and Hastie, 2004)

Max norm constraints
Max norm is a constrain that is trying to avoid overfitting by limiting the values of the weights in the model so that they modulus is less than a fixed threshold. This thresholding also prevents from obtaining exploding gradients during backpropagation.
Typically, after parameter update, the vector of weights is forced to satisfy

$$||\vec{w}||_{L_2} < C$$

Data dropout

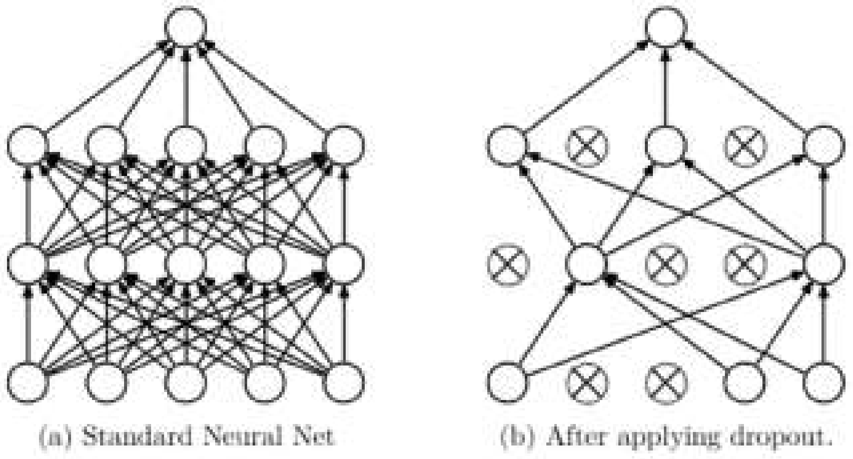

Figure 2.7: Figure from the paper of Srivastava et al., on data dropout.

With this form of regularization, the idea is to reduce the independent learning units of the network, diminishing the complexity of the model, by ignoring some sets of neuron units of the model. It is **implemented in the training phase**, by keeping a neuron active with a given probability p, and set to zero otherwise, becoming p another hyperparameter of the model. "Dropout can be interpreted as sampling a Neural Network within the full Neural Network, and only updating the parameters of the sampled network based on the input data. ... During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks (more about ensembles in the next section). " (from the Stanford course CS231n on computer vision"

2.3 Monitoring the learning process

To monitor the learning process of the network, one should look at how some parameters evolves as the epochs of the iterations progress. In particular, It is useful to plot as a function of epochs:

- **the loss function**: from the shape of the loss function as a function of the epochs we can get some information on the correctness of the learning rate value we assigned. The loss is evaluated during the forward pass of the individual batches. Noise in the obtained curve might depend on the size of the batch used in input.

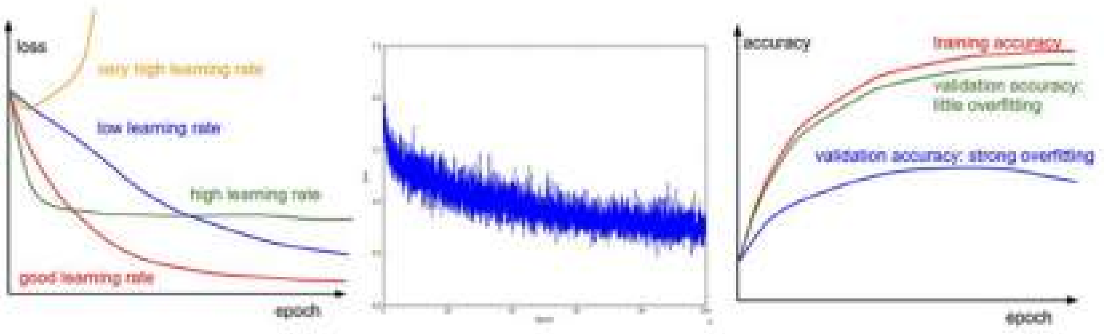

Figure 2.8: Left: Loss function drawings indicating the shapes that reveal high, low and good learning rate. Middle; we can see the noise in the loss function when the batch normalization size is low, the example is from the CIFAR10 dataset. Right: the training and validation accuracy for the case of strong and little overfitting. The figure created by composing two figures from the CS231n Stanford course in computer vision.

- **the training and validation accuracy**: these two quantities can give indications on the amount of overfitting of the model. The two possible behaviours one can obtain are represented in the figure 2.8.

- **ratio of the weights:** it is another quantity giving indications on the learning rate. It is calculated by taking the ratio of the update values to the magnitude values of the weights and the reference value indicating a good learning rate is 1.3.

- **activation and gradient distributions per each layer:** a useful tool is the visualization of the distributions of the activations or the gradients at each layer. We already used this tool for evaluating the impact of the weight initialization we were using.

- **visualization of the first layer**: when you work with images, it can be useful to visualize the features (weights) of the first layer. Noisy features could reveal unconvergence in the network, wrong learning rate, or low regularization penalty (see Figure 2.9)

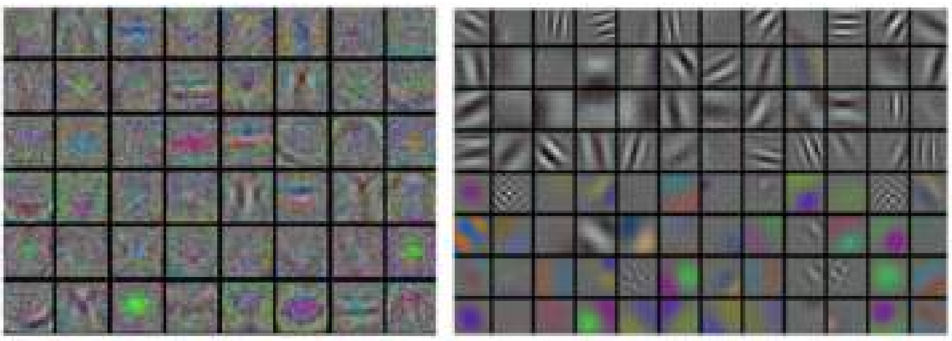

Figure 2.9: Left: noisy weights for the first layer of the neural network, right: smooth features that indicate the training is going fine, from the CS231n Stanford course in computer vision.

2.4 Parameter updates and optimization of the hyperparameter

After computing the gradients, during backpropagation, the calculated gradients are used to update the parameters. There are various ways to update the parameters starting from the gradients:

- **Vanilla update**: the update is done along the direction of the negative gradient. The update, given the learning rate as hyperparameter, is:

$$x = x - \text{learning-rate} * dx$$

- **Momentum update**: based on interpreting the loss as a potential energy function, it sets the initial parameters like to put a particle in an initial position with zero velocity. Then, if we imagine to apply a force to the particle, this force is then exactly the negative gradient of the loss function. In this case the gradient impacts the velocity, and then the velocity impacting on the position. There's a new hyperparameter, that can be associated in the physical meaning to the role of friction that dampens the velocity and reduces the kinetic energy of the system.

$$v = \mu v - \text{ learning-rate } * dx$$
$$x = x + v$$

- **Nesterov momentum**: this is an approach similar to the one of the momentum update, but here we threat the future approximate position as a "look ahead".

$$x_{ahead} = x + \mu * v$$
$$v = \mu v - \text{ learning-rate} * dx_{ahead}$$
$$x = x + v$$

- **Newton's method**: it is a second order method that iterates an update dependent on the Hessian matrix, i.e.. a matrix of the second order partial derivatives of the function. The gradient vector is the same seen in the gradient descent. With the local curvature given by the Hessian, updates are more efficient.

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

In addition to the parameter updates, one good practice in machine learning is to anneal the learning rate over time. You can imagine the learning rate as a sort of level of kinetic energy available in the system. When it is too high, particles bounces randomly around and cannot reach the minima. However, we need to be careful on how we make the learning rate decay, because too slow decay can make the system converge too quickly, without finding the best position.
There are three common types of implementation for the learning decay, i.e. the step decay, the exponential decay and the 1/t decay.

Optimization of hyperparameters

In general in a model there are two types of parameters:

- **hyperparameters**: all the parameters arbitrarily set before starting the training. They define how the model is structured in the very beginning.
- **model parameters**: parameters learned during the training (weights in neural networks, for example). They establish how to use input data for obtaining the output.

We saw the amount of hyperparameters that characterize a neural network. Some of the most common are the initial learning rate, the decay constant for the learning rate, the strength of the regularization or of the dropout. We usually want to find the right combination that creates the best model, or, in terms of a loss function, that minimizes the loss function or maximizes the accuracy.

Optimization of the hyperparameters can be done with:

- **manual search**: with this method, we choose some values for the hyperparameters based on our experience. then we do the training, we evaluate the accuracy of the model, and we start again until we get to a decent accuracy.
- **random search**: Based on a grid of hyperparameters, we follow the same procedure but we pick the values as random combinations of the values of the grid.
- **grid search**: In this case we test the model on all the possible combinations of the grid. For choosing which parameters to put in the grid, we can apply first the random search.
- **automated hyperparameter tuning**: the parameters are obtained using either Bayesian optimization, Gradient descent or evolutionary algorithms.
- **artificial neural network tuning**: it is possible to apply grid search or random search using deep learning

For more on this topic, check the article from Pier Paolo Ippolito on Medium.

2.5 Transfer learning and fine tuning

IIt is quite difficult to have a dataset large enough for training a convolutional neural network from scratch using random initialization. Transfer learning and fine-tuning are two popular techniques born to take advantage of the knowledge acquired in pre-existing models.

Transfer learning: it is a technique based on:

1. Pretrain a convnet on a large dataset, and load it into memory
2. Remove the last fully connected layer that would give the class scores for the different tasks, and then use the rest of the convnet as a feature extractor for a new dataset. Without the fully connected layer, from the CNN we would get an array of 4096 features for each input image (called CNN codes).
3. Freeze the parameters of the convnet so as to avoid losing any information they contain during future training rounds.
4. Adding as new layer a linear classifier.
5. Training the new layer on another dataset, smaller, keeping the weights frozen.

Fine-tuning: it is a similar technique compared to transfer learning, but the difference in this case is that when the new training on the new dataset is done, the weights of the classifier are unfrozen, and we allow them to be updated during this new training. Now, since these parameters are already quite well tuned, for this second training we can potentially use a smaller learning rate, for example 1/10 of the original value can be a good starting point. In this way, **fine-tuning enables the model to learn task-specific features** but still preserving the general knowledge obtained from the large initial dataset.

Let's focus on the main differences between the two processes:

- *Training approach*: in transfer learning all pre-trained layers are frozen, while in fine tuning we unfreeze some of them.
- *Domain similarity*: Fine tuning works when the new dataset is quite large and related to the original one, while transfer learning works when the new task is similar to the one the original model was trained on.
- *Computational resources*: Transfer learning needs less resources because only the new layers get trained while for fine tuning the usage of the computational resources is quite similar to the original one.
- *Training time*: Transfer learning needs less training time because it has less parameters to train compared to fine-tuning
- *Dataset size*: transfer learning works for small datasets because it exploits the knowledge from the large dataset