EE-411: Fundamentals of inference and learning **2021-2022**

Lecture 7: Neural networks

Lecturer: Florent Krzakala *Scribes: Seyedkhashayar Najafi, Soroush Mehdi, Gojko Cutura*

7.1 Intro

7.2 Not Intro

2-Layer N.N.

As we are introduced with mapping in the previous lectures, we work with mapping in the neural networks, but instead of guessing the mapping we have to learn it.

$$\hat{y} = \sigma^{(2)}\left(\vec{W^{(2)}} \cdot \sigma^{(1)}\left(W^{(1)}\vec{x}\right)\right) \tag{7.1}$$

$\hat{y}$: Scalar

$\sigma^{(1)}$: Function,Component-wise

$\sigma^{(2)}$: Function

$\vec{W^{(2)}} \in \mathbb{R}^n$

$W^{(1)} \in \mathbb{R}^{n\times d}$

In the predict function (7.1), there are two multiplicands $W^{(1)}$ and $\vec{W^{(2)}}$ that we have to learn.

What is the reason that we like N.N?

- Because, in the same way as kernel methods, with N.N.s we are going to use feature maps. However, here we are going to *learn* them:
$$\vec{x} \to \Phi = \sigma^{(1)}(W^{(1)}\vec{x})$$
- Universal approximation theorem for a general two layer neural net
$$\hat{y} = \sigma^{(2)}\left(\vec{W^{(2)}} \cdot \sigma^{(1)}\left(W^{(1)}\vec{x} + \mathbf{b}^{(1)}\right) + b^{(2)}\right) \tag{7.2}$$
In the linear format we can write it as:
$$\hat{y} = \sigma^{(2)}\left(\sum_{i=1}^{n} W_i^{(2)}\sigma^{(1)}\left(\sum_{j=1}^{D} W_{ij}^{(1)}x_j + b_i^{(1)}\right) + b^{(2)}\right) \tag{7.3}$$
If (1) n is large enough and (2) σ is not a polynomial, we can approximate any continuous function. For instance:
$$\sigma(z) = \begin{cases} \dfrac{1}{1+e^{-z}} \to: \textit{Sigmoid} \\ max(0,z) \to: \textit{Relu} \\ tanh(z) \end{cases}$$

Example 1 :

A very instructive example is: use $\sigma^{(1)}(z) = cos(z)$ and $\sigma^{(2)}(z) = z$ using $\mathbf{b}^{(1)} = \begin{cases} \vdots \\ -\frac{\pi}{2} \\ \frac{\pi}{2} \\ \vdots \end{cases}$, and then we are going to have:

$$\hat{y} = \sum_{i=1}^{\infty} W_i^{(2)} \left(cos(\mathbf{W_i^{(1)}x}) + sin(\tilde{\mathbf{W_i^{(1)}}}\mathbf{x}) \right) \rightarrow \text{Fourier expansion}$$

We already know from Fourier analysis can this can approximate well continuous functions, so the universal approximation theorem should not come as a surprise!

Example 2 :

If we choose $W^{(1)}$ a random matrix and $W^{(2)} = \frac{1}{\sqrt{D}}$ and $\sigma_2(z) = f(z)$ then the N.N will be a Random Features model, and there when n is very large it leads to a kernel method. Again we know such method can fit anything.

Note:

We can use the predict function as before for different problems.

$$[y_i, \vec{x_i}]_n \Longrightarrow \begin{cases} Regression \rightarrow \underset{W_1,W_2,b_1,b_2}{argmin} \left(\sum_{i=1}^{\infty} (y - \hat{y}(\vec{x}))^2 \right) \\ \\ Classification \rightarrow \begin{cases} \underset{W_1,W_2,b_1,b_2}{argmin}\ Hinge(y, \hat{y}(\vec{x})) \\ \underset{W_1,W_2,b_1,b_2}{argmin}\ Logistic(y, \hat{y}(\vec{x})) \end{cases} \end{cases} \tag{7.4}$$

7.3 Back Propagation

Example

For n=1, we would like to minimize the Loss:

$$\mathcal{L} = \frac{\left(\widehat{y}\left(x\right) - y\right)^2}{2} \tag{7.5}$$

So we need to find the expressions below:

$$\frac{\partial Loss}{\partial W_{ab}^{(1)}}, \frac{\partial Loss}{\partial W_a^{(2)}}, \frac{\partial Loss}{\partial b_a^{(1)}}, \frac{\partial Loss}{\partial b^{(2)}} \tag{7.6}$$

To do this, we need to just remember chain rule:

$$\frac{d}{dx} f\left(g\left(x\right)\right) = f'\left(g\left(x\right)\right) g'\left(x\right)$$

Now we start finding the derivations:

$$\frac{\partial Loss}{\partial W_a^{(2)}} = \left(\widehat{y}\left(x\right) - y\right) \sigma^{2'} \left(\vec{W}^{(2)}\sigma^{(1)}\left(W^{(1)}\vec{x} + \vec{b^{(1)}}\right) + b^{(2)}\right)\left[\sigma^{(1)}\left(W^{(1)}\vec{x} + \vec{b^{(1)}}\right)\right]_a \tag{7.7}$$

Now we introduce the notations below:

$$\begin{aligned} \vec{h}^{(1)} &\triangleq W^{(1)}\vec{x} + \vec{b^{(1)}} \\ \vec{X}^{(1)} &\triangleq \sigma_1(\vec{h}^{(1)}) \\ h^{(2)} &\triangleq \vec{W}^{(2)} \cdot \vec{X}^{(1)} + b^{(2)} \end{aligned} \tag{7.8}$$

So now we've got: $\widehat{y} = \sigma^{(2)}(h^{(2)})$. And if we replace the terms 7.8 in equation 7.7, we get:

$$\frac{\partial Loss}{\partial W_a^{(2)}} = \underbrace{\left(\widehat{y} - y\right) \sigma^{(2)'}\left(h^{(2)}\right)}_{e^{(2)}} X_a^{(1)} \tag{7.9}$$

So by taking $e^{(2)} = \left(\widehat{y} - y\right) \sigma^{(2)'}\left(h^{(2)}\right)$, we have our answer as:

$$\frac{\partial Loss}{\partial W_a^{(2)}} = e^{(2)} X_a^{(1)} \tag{7.10}$$

Using the same method for b_2 we get:

$$\frac{\partial Loss}{\partial b^{(2)}} = e^{(2)} \tag{7.11}$$

Now we go to the inner expressions and performing those derivations we find:

$$\frac{\partial Loss}{\partial W_{ab}^{(1)}} = \underbrace{(\underbrace{\left(\widehat{y} - y\right) \sigma^{(2)'}\left(h^{(2)}\right)}_{e^{(2)}} W_a^{(2)} \sigma^{(1)'}\left(h_a^{(1)}\right)}_{e_a^{(1)}} x^b \tag{7.12}$$

From expression above we define $\mathbf{e}^{(1)}$ as:

$$\mathbf{e}^{(1)} = \begin{cases} e^{(2)} W_1^{(2)} \sigma^{(1)'}\left(h_1^{(1)}\right) \\ \vdots \\ e^{(2)} W_n^{(2)} \sigma^{(1)'}\left(h_n^{(1)}\right) \end{cases}$$

Finally we have

$$\frac{\partial Loss}{\partial W_{ab}^{(1)}} = e_a^{(1)} X_b^{(1)} \tag{7.13}$$

and similarly for $\mathbf{b}^{(1)}$ we get:

$$\frac{\partial Loss}{\partial b^{(1)}} = e_a^{(1)} \tag{7.14}$$

So now we have every derivation we needed. As can be seen from the flow of the calculations, we start from the outmost expressions and go inwards (easiest to hardest), hence the name "BACK-PROPAGATION". This example was for small number of layers, but fortunately if we increase the number of layers, the expressions can be calculated in the same fashion.

In full generality for a multi-layer neural networks where

$$\hat{y} = \sigma^{(L)}(W^{(L)}\sigma^{(L-1)}(W^{(L-1)} \dots \sigma^{(1)}(W^{(1)}\mathbf{x}_0 + \mathbf{b}^{(1)})) + b^{(L)}) \tag{7.15}$$

we denote

$$\mathbf{x}^{(i)} = \sigma^{(i)}(W^{(i)}x^{(i-1)} + \mathbf{b}^{i}) = \sigma^{(i)}(\mathbf{h}^{i}) \tag{7.16}$$
$$\mathbf{h}^{(i)} = W^{(i)}x^{(i-1)} + \mathbf{b}^{(i)} \tag{7.17}$$

and write:

R **Feed Forward:** We give X to the system and calculate the h and the xs
Back Propagation: We calculate the error e from h

$$e^{(L)} = \sigma^{(L)'}(\hat{y} - y) \tag{7.18}$$
$$\dots \tag{7.19}$$
$$\mathbf{e}_j^{(l)} = \sigma^{(l)'}(\mathbf{h}_j^{(l)}) \sum_i W_{ij}^{(l+1)}\mathbf{e}_i^{(l+1)} \tag{7.20}$$
$$\dots \tag{7.21}$$
$$\mathbf{e}_j^{(1)} = \sigma^{(1)'}(\mathbf{h}_j^{(l)}) \sum_i W_{ij}^{(2)}\mathbf{e}_i^{(2)} \tag{7.22}$$

We can write all derivatives as

$$\frac{\partial \mathcal{L}}{\partial W_{ab}^{(l)}} = \mathbf{x}_b^{(l-1)}\mathbf{e}_a^{(l)} \tag{7.23}$$
$$\frac{\partial \mathcal{L}}{\partial b_a^{(l)}} = \mathbf{e}_a^{(l)} \tag{7.24}$$

7.4 Deep Learning - Introduction

Now that we are familiar with 2-layer neural networks, we're ready to move on to more modern approaches, summed up in two words as "Deep Learning". In its essence, deep learning is conceptually not much different from these simple neural networks. What do we mean by that? Well, as you will see in the following subsections, we apply exactly the same rules as we've done so far (same feed-forward methods through each of the layers, Loss computation, and the same way of doing back-propagation of the error through the network). However, even with that in mind, deep learning has definitely sparked a little "revolution" in the engineering world, by introducing much more complex and powerful neural networks, along with methods (and tricks) of efficiently training them.

Why the need for Deep Learning?

- With the same number of nodes as their shallow counterparts, deep networks can represent more complex functions
- Many sources of data are hierarchical by nature, as are deep networks (e.g. classification of animal species)
- There was a strong inspiration to mimic the human brain, e.g. the visual cortex (convolutional networks)
- Finally, there's something about these networks that prevents overfitting... still not completely comprehended by mankind :)

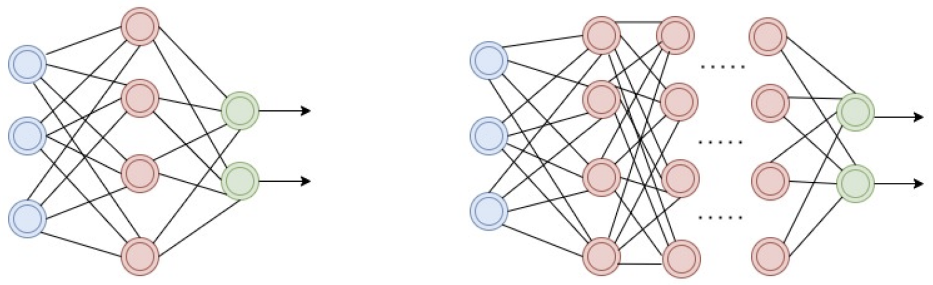

Figure 7.1: Illustration of shallow and deep networks

7.4.1 Minimizing the empirical risk - Gradient descent

Yet again, we opt for the well-established gradient descent method for minimizing the empirical risk of the network. As mentioned earlier, the methods remain the same, but with much more layers in the network, some general computation rules should be derived.

The feed-forward pass of a deep neural network could be represented as:

$$\vec{x_0} \rightarrow \vec{x_1} = g_1\left(W_1\vec{x_0}\right) \rightarrow ... \rightarrow \vec{x}_n = g_n\left(W_n\vec{x}_{n-1}\right) \rightarrow ... \rightarrow \hat{y} = g_L\left(W_L\vec{x}_{L-1}\right)$$

where the variables $\vec{x}_i$ represent the inputs to the corresponding layers, matrices W_i are those layers' weights, and the functions g_i are their activation functions. To make the notation much more pleasant, we introduce variables $\vec{h}_i$ as $\vec{h}_i = W_i\vec{x}_{i-1}$. The feed-forward pass can now be rewritten as:

$$\vec{x_0} \rightarrow \vec{x_1} = g_1\left(\vec{h}_1\right) \rightarrow ... \rightarrow \vec{x}_n = g_n\left(\vec{h}_n\right) \rightarrow ... \rightarrow \hat{y} = g_L\left(\vec{h}_L\right)$$

The back-propragation of errors can be represented as:

$$e_j^1 = g_1'\left(h_j^1\right)\sum_i W_{ij}^2 e_i^2 \leftarrow ... \leftarrow e_j^n = g_n'\left(h_j^n\right)\sum_i W_{ij}^{n+1} e_i^{n+1} \leftarrow ... \leftarrow e^L = g_L'\left(h^L\right)\left(\hat{y} - y\right)$$

When the output $\hat{y}$ is obtained, the loss can be computed as:

$$L = \frac{\left(\hat{y} - y\right)^2}{2}$$

The main thing that remains is to compute the derivatives of the loss with respect to the weights:

$$
\begin{aligned}
\frac{\partial L}{\partial W_{ab}^{(l)}} &= (\hat{y} - y)\, g_L' \left(h^L\right) \sum_k W_k^{(L)} \frac{\partial x_k^{(L-1)}}{\partial W_{ab}^{(l)}} = \sum_k W_k^{(L)} \frac{\partial x_k^{(L-1)}}{\partial W_{ab}^{(l)}} e^L \\
&= \sum_k W_k^{(l)} \left(\frac{\partial}{\partial W_{ab}^{(l)}} g^{(L-1)} \left(\sum_{k'} W_{kk'}^{(L-1)} x_{k'}^{(L-2)} \right) \right) e^L \\
&= \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial W_{ab}^{(l)}} \sum_k W_{kk'}^{(L-1)} W_k^{(L)} \left(g^{(L-1)\prime} \left(h_k^{L-1} \right) \right) e^L \\
&= \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial W_{ab}^{(l)}} \sum_k W_{kk'}^{(L-1)} e_k^{L-1} \\
&= \dots = \sum_k \frac{\partial x_k^{(n-2)}}{\partial W_{ab}^{(l)}} \sum_i W_{ik}^{(n-1)} e_i^{n-1} = \dots \\
&= \sum_k \frac{\partial x_k^{(l)}}{\partial W_{ab}^{(l)}} \sum_i W_{ik}^{(l+1)} e_i^{l+1} = x_b^{(l-1)} e_a^{(l)} \\
&\Rightarrow \frac{\partial L}{\partial W_{ab}^{(l)}} = x_b^{(l-1)} e_a^{(l)}
\end{aligned}
$$

7.4.1.1 Batch gradient descent

The initial approach to performing gradient descent in deep learning was to compute the gradient of the risk as the sum (or average) of the gradient of the loss for **all** of the elements in the dataset. This can be formulated as:

$$\theta^{t+1} = \theta^t - \eta \nabla \mathcal{R} \left(\theta^t, \{\mathbf{x}\}_{i=1,\dots,n}, \{y\}_{i=1,\dots,n}\right)$$

$$\nabla \mathcal{R} \left(\theta^t, \{\mathbf{x}\}_{i=1,\dots,n}, \{y\}_{i=1,\dots,n}\right) = \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L} \left(\mathbf{x}_i, y_i, \theta^t\right)$$

However, with datasets getting larger and larger, the flaws of this method were obvious right away: it's extremely inefficient - it takes a lot of time to compute the gradient of the risk for the whole dataset, just to perform one parameter update step.

7.4.1.2 Mini-batch gradient descent

The mini-batch gradient descent method came about as the solution to the previously stated problem, as an easy to implement (yet game-changing) trick. Without it, it would be basically impossible to train neural networks nowadays, even though it adds a bit of noise. The method says that you can just split the whole dataset into smaller chunks of data (i.e. mini-batches), and then for each of them compute the gradient of the risk and perform an update step for the parameters.

$$\theta^{t+\frac{1}{n_b}} = \theta^t - \eta \nabla \mathcal{R} \left(\theta^t, \{\mathbf{x}\}_{i=n_1,\dots,n_2}, \{y\}_{i=n_1,\dots,n_2}\right)$$

$$\nabla\mathcal{R}\left(\theta^t, \{\mathbf{x}\}_{i=n_1,\ldots,n_2}, \{y\}_{i=1,\ldots,n}\right) = \frac{1}{n_2 - n_1}\sum_{i=n_1}^{n_2}\nabla\mathcal{L}\left(\mathbf{x}_i, y_i, \theta^t\right)$$

The improvement is obvious: let's assume that it takes an amount of time T to compute the loss gradients for the whole dataset; in the case of batch gradient descent, we would have only one update step during time T, whereas in the case of mini-batches, we would have N_b updates in the same time span T (where N_b is the number of samples in a mini-batch).

7.4.1.3 Stochastic gradient descent

Stochastic gradient descent is a specific (extreme) case of mini-batch gradient descent, where mini-batches are of size one, $N_b = 1$. This method adds even more noise to the learning process, but it still seems to work very well.

$$\theta^{t+\frac{1}{n}} = \theta^t - \eta\nabla\mathcal{R}\left(\theta^t, \{\mathbf{x}\}_i, \{y\}_i\right)$$

$$\nabla\mathcal{R}\left(\theta^t, \{\mathbf{x}\}_{i=n_1,\ldots,n_2}, \{y\}_{i=1,\ldots,n}\right) = \nabla\mathcal{L}\left(\mathbf{x}_i, y_i, \theta^t\right)$$

Why Mini-batch gradient descent?

- The parameter updates are much more frequent compared to batch gradient descent - that is much faster and memory-efficient
- The effective noise in the dynamics helps optimization and regularization (not quite proven yet but there is a justification behind it)
- In practice: basically the only way to use neural networks with gradient descent and large modern datasets
- Usually it all boils down to the same point, meaning that the added noise shouldn't be too big of a worry

7.5 Deep Learning - tips and tricks

7.5.1 Gradient descent

A physics analogy - we can think of the parameters as the "movement", and the gradient could be represented as the "speed". In that setting, speed changes with the gradient of the force (much like the speed-acceleration relation). We formulate the "speed" and the "movement" as:

$$\mathbf{v}^{t+1} = \eta\nabla f\left(\theta^t\right)$$

$$\theta^{t+1} = \theta^t - \mathbf{v}^{t+1}$$

- Momentum

  This method proposes that the current speed is not only dependent on the current value of the gradient, but also on a fraction of the speed in the previous time step.

$$\mathbf{v}^{t+1} = \gamma \mathbf{v}^t + \eta \nabla f\left(\theta^t\right)$$
$$\theta^{t+1} = \theta^t - \mathbf{v}^{t+1}$$

  Extending on the physics analogy, this could be interpreted as

$$v_1 = v_0 + a\Delta t$$

  A good example to think of would be keeping the ball rolling in the same direction. Also, it's effectively averaging all the previous directions of the ball.

- Nesterov acceleration

  Nesterov acceleration extends on the previous method. It also adds momentum, but it evaluates the gradient as if it were in the step before!

$$\mathbf{v}^{t+1} = \gamma \mathbf{v}^t + \eta \nabla f\left(\theta^t - \gamma \mathbf{v}^t\right)$$
$$\theta^{t+1} = \theta^t - \mathbf{v}^{t+1}$$

7.5.1.1 Adaptive learning rates

One parameter that we haven't addressed at all so far is the learning rate η. The main idea is choosing a larger value in the beginning and decreasing it as time goes by and we're (hopefully) approaching a minima. Here is a few methods of choosing η in such a fashion, but more wisely, i.e. adapting it to the current state of the learning algorithm:

- Adagrad

  The Adagrad method scales η for each parameter (i.e. it has a per-parameter learning rate), according to the history of gradients (previous step).

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f\left(\theta^t\right)$$

  Here, G is a diagonal matrix that contains the sum of all squared gradients so far:

$$G_{t+1} = G_t + (\nabla f)^2$$

  Thus, when the gradient is very large, the learning rate gets reduced, and vice-versa. However, the main problem of Adagrad is that the values of G are always increasing, leading to all gradient updates going to zero!

- RMSprop

  The RMSprop method (Root Mean Square prop) is very similar to Adagrad, with one small (yet game-changing) tweak. Namely, in RMSprop, the G term is calculated by taking the exponentially decaying moving average, instead of the sum of squared gradients. This can be interpreted as a "memory" term perhaps (how far behind you look).

$$G_{t+1} = \gamma G_t + (1 - \gamma)(\nabla f)^2$$

- ADAM

  The ADAM (Adaptive Moment Estimation) method is by far the most utilized optimizer nowadays. The main concept of ADAM is that it keeps the exponentially decaying average of both the gradients (boiling down to the momentum) and the square of the gradients (the G term from before). That can be formulated in the following way:

$$G_t = \beta_2 G_{t-1} + (1-\beta_2)\left(\nabla f\right)^2$$
$$M_t = \beta_1 M_{t-1} + (1-\beta_1)\left(\nabla f\right)$$

  After normalizing these terms by doing: $\hat{M}_t = \frac{M_t}{1-\beta_1^t}, \hat{G}_t = \frac{G_t}{1-\beta_2^t}$, we arrive to the parameter-updating rule of ADAM:

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{\hat{G}_t + \epsilon}}\hat{M}_t$$

- NADAM

  This method is a combination of ADAM and Nesterov acceleration. It can be found as an optimizer in PyTorch.

7.5.2 Initialization of the weights

A very important question in training neural networks was how to initialize the weights... Intuitively, yet naively, one could just initialize them to zeros. However, that causes a problem of the outputs being zero, and with the derivative of the activation (ReLU in particular) at that value also being zero, the parameters would never be updated!

So, what should we do? On one hand, the weights should be small enough to try to stimulate symmetric activation functions around their linear regime (tanh, sigmoid), causing the gradients to be larger and the training process to be faster. On the other hand, the weights should be also large enough, otherwise the signal is too weak for backpropagation...

What works in practice?

- Xavier initialization

$$W_{init} \sim \mathcal{N}\left(0, \frac{2}{N_{in} + N_{out}}\right)$$

  where W_{init} is the initialized learnable weight matrix of shape (N_{in}, N_{out})

- Kaiming-He initialization

$$W_{init} \sim \mathcal{N}\left(0, \frac{2}{N_{in}}\right)$$

  It's good to note that this initialization method does in fact work better that Xavier. There are some solid justifications for this, but (as usual) it is largely empirical.

  NOTE: the numerator 2 in the previous expressions isn't always used! It actually depends on the activation function that is used. Luckily for us, this is also pre-implemented in PyTorch and is computed by default.

Does it actually matter? It certainly does... as the case with many aspects of deep learning, it's not backed up by some strong theory, but the empirical evidence is undeniable.

7.5.3 Data pre-processing

Data pre-processing is a very important step leading up to the training part itself. This does not only apply for deep learning, but is true in the general case of dealing with any type of data.

Firstly, the scale of the data is very important. Let's assume that we, for example, have a dataset of people and some of their characteristics. In the dataset, the height feature is given in meters, so the range is approximately $[0, 2]$. Alongside that, we have the age feature, with an approximate range of $[0, 100]$. If we were to fit a model to this data, it is very likely that it would be biased towards the features of a bigger scale! A very easy thing to do in this case is to just convert the height to centimeters, ultimately arriving to a similar scale. Of course, that is an extremely case-specific advice, so what should we do in general?

One of the approaches is to transform the data such that all the features have a mean of zero (i.e. **centering**), and that they're standard deviation is $O(1)$ (i.e. **normalizing**). This is easily formulated as:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

7.5.3.1 Batch Normalization

Now, there is a very popular application of the previous scaling technique (centering + normalizing), where it is performed across each mini-batches of data during the training process of a neural network. This is called Batch Normalization, and we'll briefly explain how it works.

Let's say that we are given a mini-batch of samples $\mathcal{B} = x_1, ..., x_m$. The steps we take in order to perform Batch Normalization are the following:

- Step 1 - calculate the mean and variance of the samples across the mini-batch $\mathcal{B}$

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i, \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

- Step 2 - normalize the input data by the statistics from step 1

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

- Step 3 - Compute the output for the Batch Normalization layer, by scaling and shifting the normalized data from step 2

$$y_i = BN_{\gamma,\beta}(x_i) = \gamma \hat{x}_i + \beta$$

  where γ and β are **learnable** parameters!

NOTE 1: It's still not completely clear why this works as well as it does (it's largely empirical), but it is a great way to prevent overfitting and speed up the learning process.

NOTE 2 - how to apply this to unseen, test data? Well, we keep track of the average statistics (μ, σ) during the training phase, and then we use them for all the batches in the test phase.

7.5.4 Data augmentation

What do we do when we are lacking data for training? One option would be to go out in the wild again and collect more data, but in most cases that is not feasible. Instead, we do some simple tricks in forms of transforming the data we already have, to make the new dataset more diverse.

Some of the transformations that are used are: zooming in, flipping, rotating, or cropping the images, playing with the contrast and the colors, etc.

It would be useful to point out that, even in cases when there is enough data (or when we think that it's enough, because there's no such thing as enough data), it's still good to perform this type of data augmentation! An example would be a face recognition algorithm that runs on smartphones - even if it had a bunch of portraits of people for training, the algorithm should probably be robust to common use-cases, such as holding the phone at an angle while taking the photo, or the camera lens being dirty, or the hand that's holding the phone being shaky, ... all of this can be covered by adding a pre-processing step as we described!

7.5.5 GPU-accelerated computation

In the beginning of the deep learning research, one of the biggest problems for, if not the biggest, was that it was very computationally heavy and slow to train. As we have learned by now, there are numerous matrix multiplications going on all the time, and that was very hard to handle for CPUs (Central Processing Units). Even the modern CPUs have a very hard time with deep learning optimizations.

The game-changing discovery in the field was that GPUs (Graphics Processing Units) are super efficient in these types of computations! That allowed for much faster training, and even processing bigger batches at once since GPUs work very well with very big matrices. For example, with GPUs having bigger and bigger RAM memories, we can take a whole mini-batch of e.g. 128 images, concatenate it into one huge matrix, and then use it to compute the output.

However, with gamers being more and more busy, and cryptocurrency mining being very common at the moment, the prices of GPUs have sky-rocketed. So, make sure that you find a cheap, or even free way of using these processing units (for example Google Colab, Kaggle, Amazon Web Service, ...).

7.6 Regularization

7.6.1 Early stopping

One of the simplest popular methods that is used to prevent the model from overfitting is called early stopping. It consists of splitting the training data into a new training set and a validation set. Once that is done, and we begin the training phase, we monitor the performance of the model over time, evaluated on the validation set. When the error on the validation set starts to increase, that most likely means that the model is starting to overfit on the training data, thus being less able to generalize to unseen samples. So, at that point, we interrupt the training and keep the model parameters as they were at that point.

It sometimes happens that we intentionally let the model degrade for a while and then stop the training. This is controlled by a so-called ”patience term”.

7.6.2 Weight decay (l_2 penalty)

Weight decay is actually a method that we're already familiar with, but maybe not using that exact name - it is actually the same as l_2 regularization (also known as Ridge or Tikhonov). Let's revise what weight decay is. A new loss is introduced, that needs to be minimized:

$$\mathcal{L}'(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2$$

The corresponding gradient now becomes $\frac{\partial \mathcal{L}'}{\partial w} = \frac{\partial \mathcal{L}}{\partial w} + \lambda w$, resulting in the following update rule:

$$w^{t+1} = w^t - \eta \frac{\partial \mathcal{L}'}{\partial w} = w^t - \eta \left(\frac{\partial \mathcal{L}}{\partial w} + \lambda w \right) = (1 - \eta\lambda) w^t - \eta \frac{\partial \mathcal{L}}{\partial w}$$
$$\Rightarrow w^{t+1} = (1 - \eta\lambda) w^t - \eta \frac{\partial \mathcal{L}}{\partial w}$$

To conclude, this method is called weight decay due to the factor $(1 - \eta\lambda)$ in the update rule, ultimately making the weights smaller and smaller with time.

7.6.3 Dropout

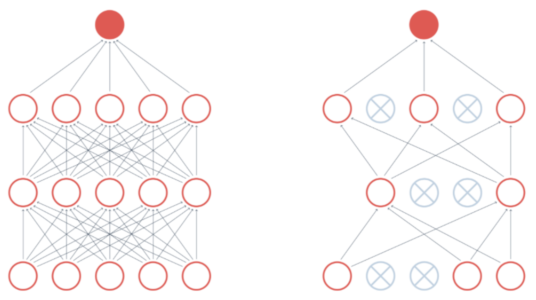

Figure 7.2: Illustration of the dropout operation

One of the most frequently used methods for regularization (preventing overfitting) nowadays is definitely Dropout. The way that it works is the following: at each iteration of the training (i.e. at each forward pass of a mini-batch) we randomly sample a part of the neurons in a layer. Those neurons will practically get disabled (not used at all) in that iteration, and an illustration is given in figure 7.2. That is reflected by putting the corresponding entries of the weight matrix to zero for that iteration, and then getting them back to what they were once the iteration is over.

Now, this is sort of becoming a pattern in these notes, but there is no real theoretical proof that this works... However, there is a lot of empirical proof that it works wonderfully when trying to prevent the model from overfitting. One common justification for this is that this way we prevent some dominant neurons from gathering all the information (i.e. overfitting), and instead, by disabling them in some iterations, we make some of the other neurons learn as well.

A sports analogy could be made here - if a team has a star player and the whole gameplan depends on him, that team might be bad at playing against different opponents. However, if that star player sits out a game or two from time to time, that would prompt some of his teammates to step up and take over the responsibility, which leads to more game-ready players.

7.7 Special layers

7.7.1 Convolutional layer

Convolutional layers (and networks) were first introduced in the 1980s, by one of the most important deep learning researchers, Yan LeCun. It introduced a completely new way of computing the feature maps, compared to the fully-connected layers that we've covered so far. Instead of each element in the input of the layer being connected to each of the elements in the output (i.e. fully-connected), only a certain neighborhood of the entries in the input is weighted (and aggregated) and connected to a given entry in the output. We are going to formulate the convolutional operator for 1- and 2-dimensional cases, and things will hopefully be clear after that (the illustrations may be helpful as well).

- 1-D convolutional layer

  In case of dealing with 1-dimensional data, the convolution operator can be defined as follows:

$$g(x) = f(x) \star w = \sum_{dx=-a}^{a} w(dx) f(x + dx)$$

  where g is the output array, f is the original array and w is the **convolutional kernel**. Every element of the kernel is considered by $-a \leq dx \leq a$, and is **learnable**. So, the kernel "traverses" through the input data, at each step calculating the weighted sum of the neighborhood that it covers at the time, and finally assigning the resulting value to the corresponding entry in the output data.

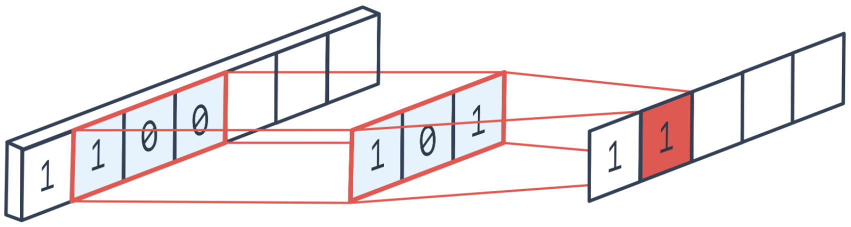

Figure 7.3: Illustration of 1-D convolution

- 2-D convolutional layer

  In case of 2-dimensional data (such as images), we extend the 1-dimensional convolution operator to the y axis as well. Namely, the convolutional kernel now also becomes a small matrix, and the feature computation part is completely analogous.

$$g(x, y) = f(x, y) \star w = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} w(dx, dy) f(x + dx, y + dy)$$

where g is the output (filtered) image, f is the original image, w is the previously defined convolutional kernel. Again, every element of the kernel is considered by $-a \leq dx \leq a$ and $-b \leq dy \leq b$.

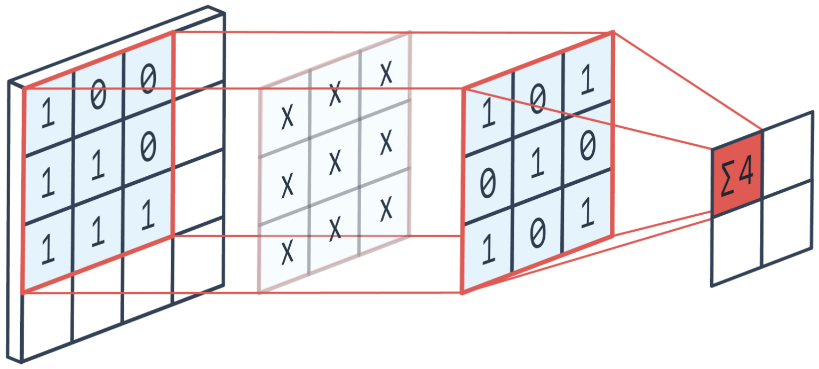

Figure 7.4: Illustration of 2-D convolution

The two key take-aways about convolutional layers are **1)** they are much more efficient since there is a lot less computations to be done (time efficiency), and it also uses a single set of weights (kernel) to compute the whole input (memory efficiency), and **2)** it imposes the notion of locality by only considering a certain neighborhood in the input when computing corresponding values for the output.

7.7.2 Max-Pooling layer

Pooling layers are a way of down-sampling the data, which is (together with convolutional layers) a building block of every convolutional neural network. As we'll see below, this is a very simple operator, that acts as a "sliding window" through the input data (similar to the convolution kernel, but without overlaps), and assigns the maximum entry inside of the input window to the corresponding entry in the output of the layer. Again, let's define the pooling operator in 1- and 2-dimensional cases.

- 1-D max-pooling layer

$$g(x) = max\{f(x + dx)\}, \text{ where } dx \in [-a, a]$$

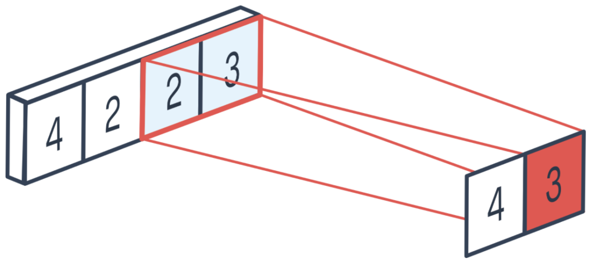

Figure 7.5: Illustration of 1-D max-pooling

- 2-D max-pooling layer

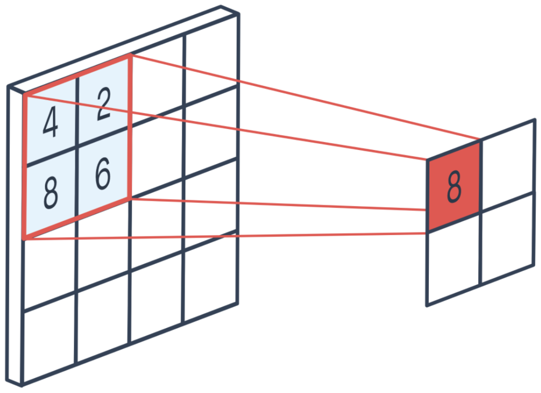

Figure 7.6: Illustration of 2-D max-pooling

$$g(x, y) = max\{f(x + dx, y + dy)\}, \text{ where } dx \in [-a, a], dy \in [-b, b]$$

Another important upside of the max-pooling layer is that it implicitly implements **translation invariance**, i.e. invariance of what the second layer sees, with respect to translations in the first layer. This is very important in computer vision, since (for example) we want to identify an object in the image regardless of its relative position in the image (a car is a car, whether it's on the far left or on the far right of the frame)!

7.7.3 Flatten layer

The third and final layer that we are going to cover is the Flatten layer. It is used to transform 2-dimensional images into 1-dimensional arrays, and the illustration of this method can be found in figure 7.7.

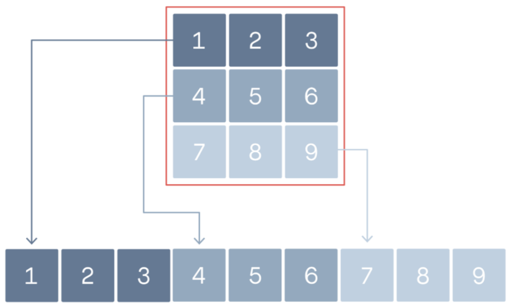

Figure 7.7: Illustration of the flattening operation

7.8 Convolutional Neural Networks

Finally, after introducing its core elements (convolution, pooling and flatten layers), we can assemble them and talk about wath we've been going towards - the Convolutional Neural Network (CNN). CNNs are basically a big sequence of stacked convolutional and pooling layers, followed by an activation, where higher stages of the network compute more global, more invairant features. If the problem at hand is of the classification nature, then we often have a classification layer at the end (perhaps a fully-connected layer followed by the softmax operation to be able to interpret the outputs as class probabilities).

Nowadays it is needless to say that CNNs achieved unbelievable results on image processing (computer vision) problems, and have further sparked the deep learning revolution.

7.8.1 Popular CNNs

Some of the popular CNN architectures include LeNet, AlexNet, VGG-16 and many others.

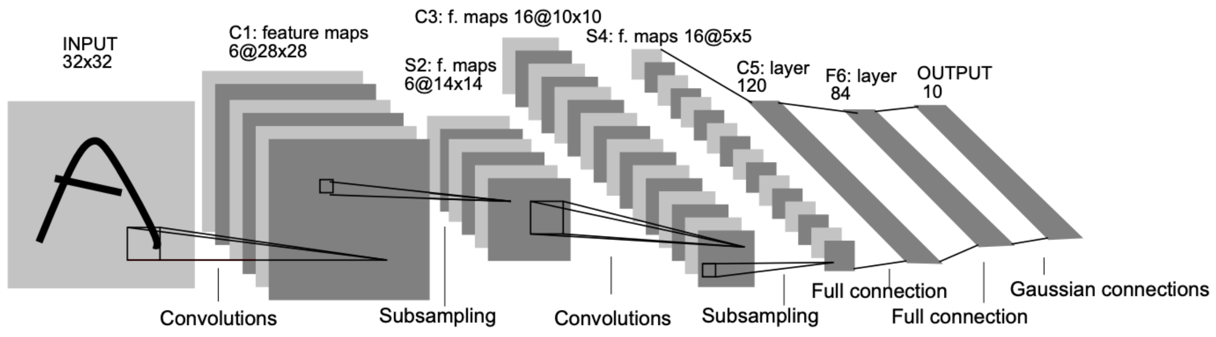

Figure 7.8: Illustration of the LeNet architecture

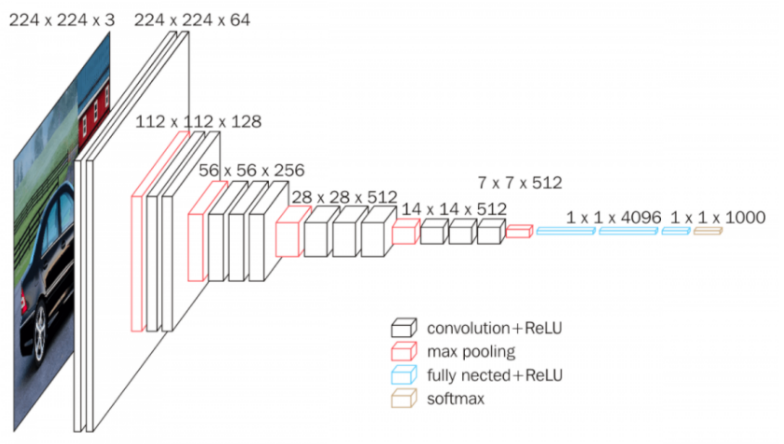

Figure 7.9: Illustration of the VGG-16 architecture

At this point, as deeper and deeper the networks became, the more promising the results were. However, that quickly reached a saturation point, after which it was useless to add more layers, as it could, not just have no effect, but also easily degrade the performances of the network.

7.8.2 ResNets and skip-connections

Luckily for the deep learning community, in the year of 2015, researchers from Microsoft have introduced the ResNet. ResNets bring with them another learning trick that was an absolute game-changer when it comes to allowing deep networks go deeper and deeper and overcoming the saturation point mentioned in the previous subsection.

The focal point of the ResNet paper were the **skip-connections**. They represent a link between an output of a given layer and an output of a certain layer, later in the network. This link could be implemented in different ways, but the most simple one would be basic addition of those matrices/tensors.

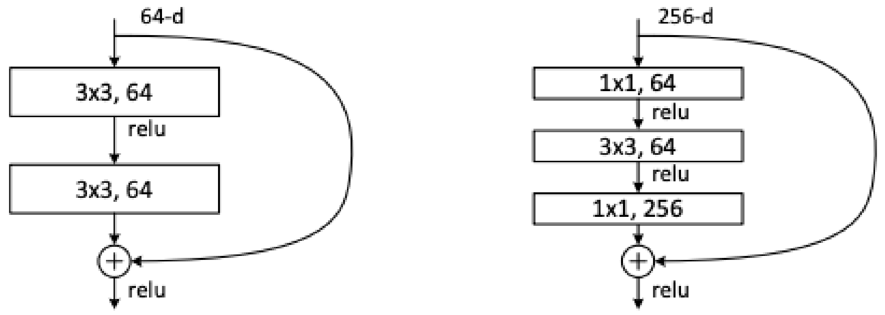

Figure 7.10: Examples of ResNet building blocks; left - building block of ResNet-34; right - "bottleneck" building block of ResNet-50/101/152

Why skip-connections?

As we already mentioned, skip-connections are a tool to help us train deeper and deeper networks. They do this by tackling the problem of vanishing gradients (as the gradients get propagated back through the network layers, each time they get multiplied by the corresponding layer matrices, leading to them getting smaller and smaller). If we look at figure 7.10, we can see that, by adding skip-connections, the gradient in the backward pass will "go" down two different paths - the first one is the one that we had before, i.e. through the layers, and the second one is via the skip connection! This allows the previous layers to receive a gradient that was not passed through these layers in between, so it has not decreased and is much more meaningful for training!

7.8.3 Overparameterizing

After ResNets gained traction and those methods were included in many other research papers, a weird thing started to happen, that's a bit counter-intuitive given what we've learned so far - as the number of parameters per model continued to grow further and further, there was surprisingly very little overfitting! Moreover, these deep networks seemed to have some strange ability to prevent themselves from overfitting (together with the ones that we covered here - Dropout and Batch Normalization).

This remains an open question in the deep learning field, but here is some intuition. So, the functions that we are optimizing in modern deep learning are highly non-convex, thus they have many different local minima in the search space. To support the bias-variance trade-off concept, most of these local minima are undesirable in terms of their ability to generalize to unseen data. However, some of the minima appear to be very good at it. Here, we arrive to an interesting conclusion - gradient descent seems to have an implicit bias towards those ”good” local minima! How? We don’t know yet, but let’s stick around for a couple more years and find out. :)