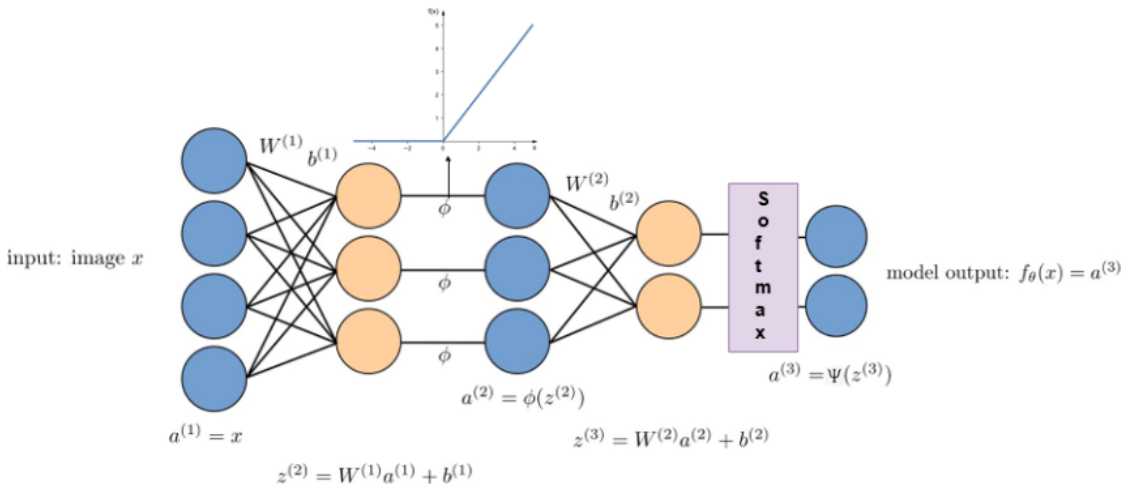
Alfano Caterina Berardi Angelo Cappelli Dario Fratocchi Emanuele

November 2020


input: image x
$W^{(1)}$
$b^{(1)}$
ϕ
ϕ
ϕ
$W^{(2)}$
$b^{(2)}$
Softmax
model output: $f_\theta(x) = a^{(3)}$
$a^{(3)} = \Psi(z^{(3)})$
$a^{(2)} = \phi(z^{(2)})$
$a^{(1)} = x$
$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$
$z^{(2)} = W^{(1)}a^{(1)} + b^{(1)}$

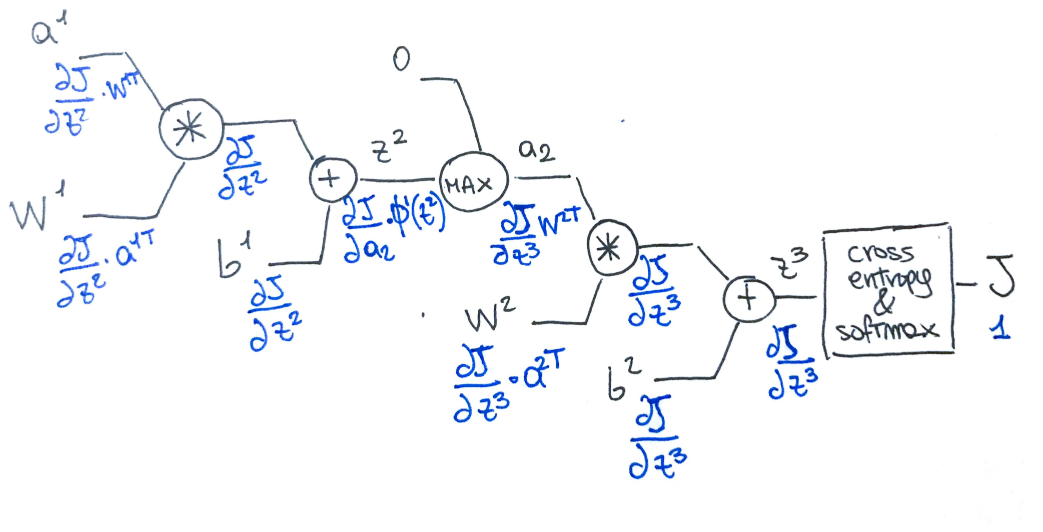
This was smoothly done thanks to the Numpy library which allowed us to compute vector and matrix multiplications and additions in a fast way.
After the two layers we obtained a K-dimensional vector for each of the N input images, which associated a probability score to all the K classes for that image. The rounded difference between our scores and the correct scores is: 2.9e-08.
We also had to compute the loss of each image, which expresses how wrong our scores were with respect to the correct label of the image (we used cross entropy loss). The rounded difference between our loss and the correct loss is 1.7e-13.

Question 2

2.a

To complete our neural network model we had to implement the backward part, which consists in the use of backpropagation as a way to obtain the gradients of the loss function w.r.t the weights of the network to know how to update the weights and optimize our model.
To understand the formulas used for the backpropagation we have to look at the computational graph below and keep in mind some rules that are valid within the chain rule:

- First of all, the partial derivative w.r.t. the addends of a sum is the same as the derivative w.r.t. the result of the sum (the upstream).
- Secondly, the partial derivative w.r.t. a factor of a vector-matrix multiplication is obtained by multiplying the upstream gradient and the transpose of the other factor.
- Finally, the general rule is that the partial derivative w.r.t a term is obtained by multiplying the upstream and the derivative of the function of that step applied to the same term.



The first step in the backward part of the neural network is finding the derivative of the loss w.r.t z_i^3. We have to keep in mind that the loss was defined as:

$$J = \frac{1}{N}\sum_{i=1}^{N}\Bigg[-\log\frac{\exp{(z_i^3)_{y_i}}}{\sum_{i=j}^{K}\exp{(z_i^3)_j}}\Bigg] \tag{1}$$

So let's first unwrap this function in a more explicit way (using the logarithm's property):

$$J = \frac{1}{N}\sum_{i=1}^{N}\Bigg[-(z_i^3)_{y_i} + \log\sum_{j=1}^{K}\exp{(z_i^3)_j}\Bigg] \tag{2}$$

Now we can start working on the derivative:

$$\frac{\partial J}{\partial z_i^3} = \frac{\partial}{\partial z_i^3}\frac{1}{N}\sum_{i=1}^{N}\Bigg[-(z_i^3)_{y_i} + \log\sum_{j=1}^{K}\exp{(z_i^3)_j}\Bigg] \tag{3}$$

Since we are working on the derivative w.r.t z_i^3 we can remove the external sum from 1 to N and focus just on the i-th element, as well as move the constant factor outside of the derivative.

$$\frac{\partial J}{\partial z_i^3} = \frac{1}{N}\Bigg[\frac{\partial}{\partial z_i^3}\Bigg(-(z_i^3)_{y_i} + \log\sum_{j=1}^{K}\exp{(z_i^3)_j}\Bigg)\Bigg] \tag{4}$$

Now we can derive the logarithm like this:

$$\frac{\partial J}{\partial z_i^3} = \frac{1}{N}\Bigg[\frac{\partial}{\partial z_i^3}\big(-(z_i^3)_{y_i}\big) + \frac{\frac{\partial}{\partial z_i^3}\sum_{j=1}^{K}\exp{(z_i^3)_j}}{\sum_{j=1}^{K}\exp{(z_i^3)_j}}\Bigg] \tag{5}$$

The derivative above the fraction is computing the derivative on each j element of the vector z_i^3 (obtaining a vector of zeros with just a 1 in the j-th position) and summing them all together, which is the same as taking the derivative of each element of the vector. Moreover we are talking about the derivative of the exponential function, so we can easily say:

$$\frac{\partial J}{\partial z_i^3} = \frac{1}{N}\Bigg[\frac{\partial}{\partial z_i^3}\big(-(z_i^3)_{y_i}\big) + \frac{\exp{(z_i^3)_j}}{\sum_{j=1}^{K}\exp{(z_i^3)_j}}\Bigg] \tag{6}$$

$$\frac{\partial J}{\partial z_i^3} = \frac{1}{N}\Bigg[\frac{\partial}{\partial z_i^3}\big(-(z_i^3)_{y_i}\big) + \psi(z_i^3)_j\Bigg] \tag{7}$$

For this last step we need to reason in the same way that we did when going from (5) to (6).

In fact, taking the derivative of $(z_i^3)_{y_i}$ means creating a vector of zeros with just a 1 in the y_i-th position, which can be represented with Kronecker's delta $(\delta_p)_{y_i}$. In our case we also need to specify that among the many δ tensors (stored in the Δ matrix) we want the one corresponding to (z_i^3) - which will be of the shape $(\delta_p)_{y_i}$.
Finally we have proved that the derivative of J w.r.t to z_i^3 is indeed:

$$\frac{\partial J}{\partial z_i^3} = \frac{1}{N}\left(\psi(z_i^3)_j - \Delta_i\right) \tag{8}$$

2.b and 2.c

We know that if we want to obtain the partial derivative of the loss J with respect to the weight matrix W^2 we can compute it like this:

$$\frac{\partial J}{\partial W^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^3}\frac{\partial z_i^3}{\partial W^2} \tag{9}$$

We also know that z^3 is the result of $xW^2 + b^2$. This means that we can say that the derivative w.r.t b^2 is equal to the derivative w.r.t z^3

$$\frac{\partial J}{\partial b^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^3} = \frac{1}{N}(\psi(z_i^3) - \Delta_i) \tag{10}$$

It also means that we can obtain the derivative w.r.t W^2 expanding the previous formula like this:

$$\frac{\partial J}{\partial W^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^3}\frac{\partial z_i^3}{\partial W^2} = \frac{\partial J}{\partial z_i^3}{a^2}^T = (\frac{1}{N}(\psi(z_i^3) - \Delta_i)){a^2}^T \tag{11}$$

Similarly we can define:

$$\frac{\partial J}{\partial a^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^3}\frac{\partial z_i^3}{\partial a^2} = \frac{\partial J}{\partial z_i^3}{W^2}^T = (\frac{1}{N}(\psi(z_i^3) - \Delta_i)){W^2}^T \tag{12}$$

To get the derivatives w.r.t W^1 and b^1 we first need to compute the derivative w.r.t z^1 which is defined like this:

$$\frac{\partial J}{\partial z^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial a^2}\phi'(z^2) = (\frac{1}{N}(\psi(z_i^3) - \Delta_i)){W^2}^T\phi'(z^2) \tag{13}$$

This means we can finally write the last two formulas, making the same considerations as before regarding the sums and multiplications.

$$\frac{\partial J}{\partial b^1} = \sum_{i=1}^{N} \frac{\partial J}{\partial z^2} = (\frac{1}{N}(\psi(z_i^3) - \Delta_i)){W^2}^T\phi'(z^2) \tag{14}$$

$$\frac{\partial J}{\partial W^1} = \sum_{i=1}^{N} \frac{\partial J}{\partial z^2}{a^1}^T = (\frac{1}{N}(\psi(z_i^3) - \Delta_i)){W^2}^T\phi'(z^2){a^1}^T \tag{15}$$

Finally we have to remember that as far as W^1 and W^2 are concerned we can't ignore the regularization applied after each of the layer. So, referring to equation (6) of the assignment we have to add to the previous partial derivatives another term (the derivative of a sum is the sum of the derivatives). While considering the derivative w.r.t. W^2 we will only consider the regularization of W^2 and vice versa (since the other term will be considered as a constant in the the derivative). This means that:

$$\begin{aligned}\frac{\partial J}{\partial W^2} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^3}\frac{\partial z_i^3}{\partial W^2} + (\lambda(\|W^2\|_2^2))' =\\ \frac{\partial J}{\partial z_i^3} a^{2^T} + (\lambda(\|W^2\|_2^2))' =\\ (\frac{1}{N}(\psi(z_i^3) - \Delta_i))a^{2^T} + \lambda((\|W^2\|_2^2))' =\\ (\frac{1}{N}(\psi(z_i^3) - \Delta_i))a^{2^T} + \lambda 2W^2\end{aligned} \tag{16}$$

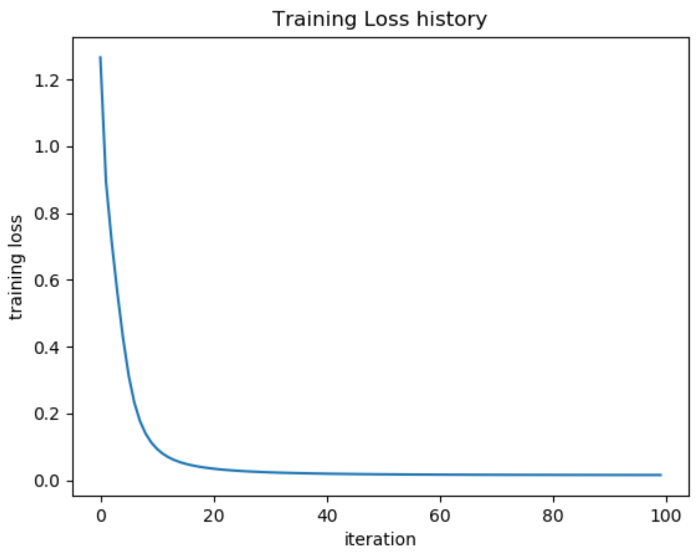
2.d

We tested our backpropagation algorithm using these formulas and obtained the right gradients. In fact, when comparing our gradients to the numerical ones we got:

- W1 max relative error: 3.561318e-09
- W2 max relative error: 3.440708e-09
- b1 max relative error: 2.738422e-09
- b2 max relative error: 4.447667e-11

Question 3

We were asked to use random batches sampled from the input data with given size. We did it by generating a random number in the range of the length of the input and sampling around this number in the amount specified by batch-size. There is also a special case for when the input is already smaller than the default batch size (200); in this situation the whole dataset is considered as a batch. As for the prediction function, we just used the trained model to compute the scores and assign each photo to the class with the highest score. Running the model on the toy data we obtained a loss of 0.015 and the following plot:

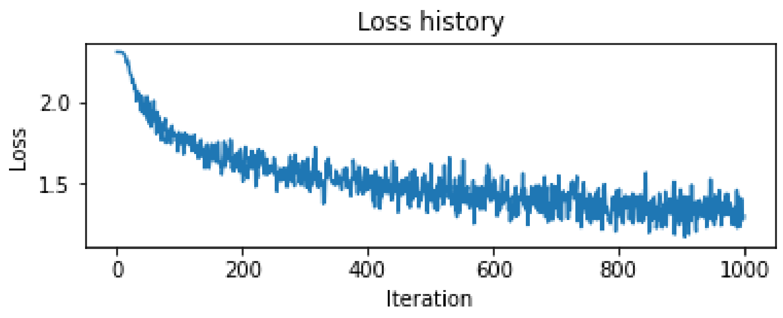


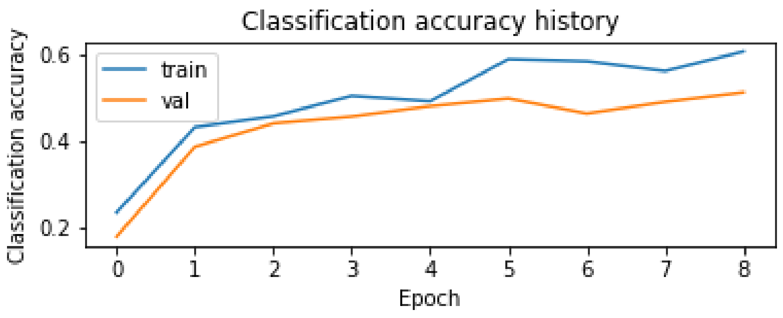
Finally we were ready to test our model on real data, from the Cifar10 dataset. As expected, with the default hyper-parameters the performance was not good, in fact we only reached an accuracy of 0.286. It was easy to spot on the plots and by looking at the results of each epoch that the loss was changing too slowly. Moreover the validation and train accuracy curves overlapped very frequently and there was no real gap between them, so we had to increase the model capacity. We decided to try out some combinations in a for loop, especially to see how much we should increase the learning parameter and the batch-size.

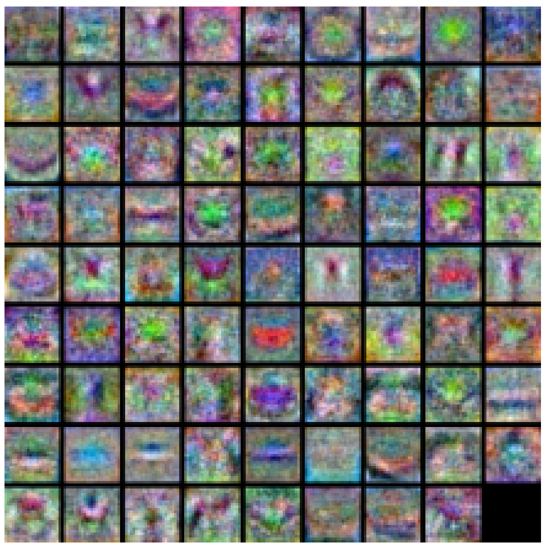
These are the parameters we tried:

- learning rates: 0.001, 0.002, 0.003
- hidden layer size: 20, 40, 60, 80
- regularization: 0.0001, 0.001, 0.003
- batch size: 200, 300, 400, 500

The best combination that resulted from this tests is: learning rate 0.002, hidden layer size 80, batch size of 400 and regularization 0.0001. With these hyper-parameters we were able to be obtain an **accuracy of 0.519** (with a loss of 1.28 in the final step). We also plotted the loss curve to see how it got smaller after each iteration, as long as comparing the train and validation accuracy and the visualization of the updated weights.
Since this was our best model we also tested it on the **test-set** and obtained a **0.50 accuracy**. This is a confirmation that our model wasn't overfitting the train set because it performed almost as good in the test set.



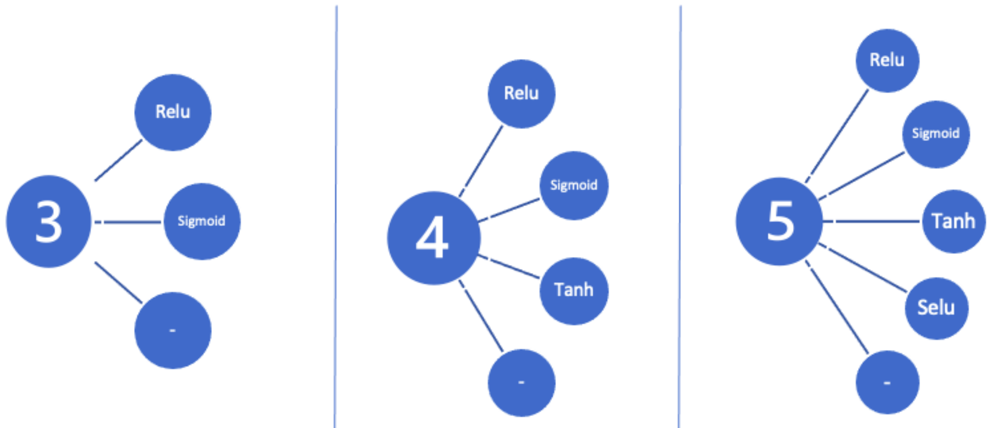




Question 4

In this exercise we completed the given code to realize a two layer neural network by creating the hidden layers of linear operation, by adding the ReLu function in the forward pass and by computing the cross entropy pytorch function (which also includes the softmax function) to compute the losses, and by getting the gradients thanks to the backpropagation module to optimize the weights of the parameters in the backward phase.
Since we already knew, from the previous exercise, which were the best hyper parameters to choose, we used them and trained the model. We were able to obtain a validation accuracy of **53.7** which resulted in a **51.43** accuracy on the test dataset.

Once the two-layers network had obtained a reasonable performance, we tried to increase the network depth to improve the results; in particular, we worked on the combination of several activation functions belonging to different layers. We never added an activation function to the last layer since the results will be passed to the softmax function when computing the loss.
Here are some outcomes of the Validation Accuracy (VA) and Test Accuracy (TA) of the models we tested:



VA	52.9 %	50.5 %	50.9 %
TA	51.55 %	50.03 %	48.46 %

As we can see, the results show that too many layers dont' guarantee a better performance, and can sometimes lead to overfitting.