Advanced Machine Learning
2: Backpropagation

Giorgio Zannini, Giulia Scikibu Maravalli, Egon Ferri

MARCH 2020

2 Backpropagation

a)

We have to verify the gradient w.r.t. $z^{(3)}$ of the loss function:

$$\tilde{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left[\frac{e^{z^{(3)}_{y_i}}}{\sum_j e^{z^{(3)}_j}}\right]$$

To show the derivation it's easier to start from the simpler case of only one input data.

$$\tilde{J}(\theta)_{z_i} = -\log\left[\frac{e^{z^{(3)}_{y_i}}}{\sum_j e^{z^{(3)}_j}}\right]$$

$$\tilde{J}(\theta)_{z^{(3)}_i} = -\log e^{z^{(3)}_{yi}} + \log\sum_j e^{z^{(3)}_j} = -z^{(3)}_{y_i} + \log\sum_j e^{z^{(3)}_j}$$

$$\frac{\partial J}{\partial z^{(3)}_i} = -\nabla_{z^{(3)}_i} z^{(3)}_{y_i} + \nabla_{z^{(3)}_i}\log\sum_j e^{z^{(3)}_j}$$

$$\frac{\partial J}{\partial z^{(3)}_i} = -\nabla_{z^{(3)}_i} z^{(3)}_{y_i} + \frac{1}{\sum_j e^{z^{(3)}_j}}\nabla_{z^{(3)}_i}\sum_j e^{z^{(3)}_j}$$

$$\frac{\partial J}{\partial z^{(3)}_i} = -\nabla_{z^{(3)}_i} z^{(3)}_{y_i} + \frac{e^{z^{(3)}_i}}{\sum_j e^{z^{(3)}_j}}$$

$$\frac{\partial J}{\partial z^{(3)}_i} = \psi(z^{(3)}_i) - \delta$$

Where $\delta \in R^N$ with:

$$\delta_i = \begin{cases} 1, & \text{if } i = y_i \\ 0, & \text{otherwise} \end{cases}$$

This result can be easily generalized to the loss that refers to the whole dataset:

$$\frac{\partial J}{\partial z^{(3)}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{z_i^{(3)}} \left(\log \sum_j e^{z_j^{(3)}} - z_{y_i}^{(3)} \right)$$

$$\frac{\partial J}{\partial z^{(3)}} = \frac{1}{N} \nabla_{z_i^{(3)}} \left(\log \sum_j e^{z_j^{(3)}} - z_{y_1}^{(3)} + \log \sum_j e^{z_j^{(3)}} - z_{y_2}^{(3)} + ... + \log \sum_j e^{z_j^{(3)}} - z_{y_N}^{(3)} \right)$$

$$\frac{\partial J}{\partial z^{(3)}} = \frac{1}{N} \left(\psi(z_1^{(3)}) - \delta_1 + \psi(z_2^{(3)}) - \delta_2 + ... + \psi(z_N^{(3)}) - \delta_N \right)$$

This can now be rewritten in matrix form as:

$$\frac{\partial J}{\partial z^{(3)}} \left(\{x_i, y_i\}_{i=1}^N \right) = \frac{1}{N} \left(\psi \left(z^{(3)} \right) - \Delta \right)$$

Where Δ is a matrix $N \times K$ dimensional with:

$$\Delta_{ij} = \begin{cases} 1, & \text{if } y_i = j \\ 0, & \text{otherwise} \end{cases}$$

b)

We now have to verify that

$$\begin{aligned} \frac{\partial J}{\partial W^{(2)}} \left(\{x_i, y_i\}_{i=1}^N \right) =& \frac{\partial J}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial W^{(2)}} \\ &= \frac{1}{N} \left(\psi \left(z^{(3)} \right) - \Delta \right) a^{(2)} \end{aligned}$$

and

$$\frac{\partial \tilde{J}}{\partial W^{(2)}} = \frac{1}{N} \left(\psi \left(z^{(3)} \right) - \Delta \right) a^{(2)'} + 2\lambda W^{(2)}$$

The first is easily done by seeing that

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = \frac{\partial \left(W^{(2)} a^{(2)} + b^{(2)} \right)}{\partial W^{(2)}} = a^{(2)}$$

Then:

$$\frac{\partial J}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial W^{(2)}} = \frac{1}{N} \left(\psi \left(z^{(3)} \right) - \Delta \right) a^{(2)'}$$

For the regularized loss it can be seen that

$$\tilde{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left[\frac{e^{z_{y_i}^{(3)}}}{\sum_j e^{z_j^{(3)}}}\right] + \lambda\left(\left\|W^{(1)}\right\|_2^2 + \left\|W^{(2)}\right\|_2^2\right)$$

can be re-written as

$$\tilde{J}(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left[\frac{e^{z_{y_i}^{(3)}}}{\sum_j e^{z_j^{(3)}}}\right] + \lambda\left(W^{(1)\prime}W^{(1)} + W^{(2)\prime}W^{(2)}\right)$$

Then its derivative is just:

$$\frac{\partial \tilde{J}}{\partial W^{(2)}} = \frac{\partial J}{\partial W^{(2)}} + 2\lambda W^{(2)} = \frac{1}{N}\left(\psi\left(z^{(3)}\right) - \Delta\right)a^{(2)\prime} + 2\lambda W^{(2)}$$

c)

Now we derive the regularised loss with respect to the other parameters

$$\left(W^{(1)}, b^{(1)}, b^{(2)}\right)$$

- $b^{(2)}$

$$\frac{\partial J}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial b^{(2)}} = \frac{1}{N}\left(\psi\left(z^{(3)}\right) - \Delta\right)\mathbf{1}'$$

- $W^{(1)}$

$$\frac{\partial \tilde{J}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W^{(1)}} = \frac{1}{N}\left(\psi\left(z^{(3)}\right) - \Delta\right)W^{(2)}\mathbf{I}\left[z^{(2)} > 0\right]' a^{(1)\prime} + 2\lambda W^{(1)}$$

where $\mathbf{I}\left[x > 0\right]$ is 1 when $x > 0$ and 0 when $x < 0$.

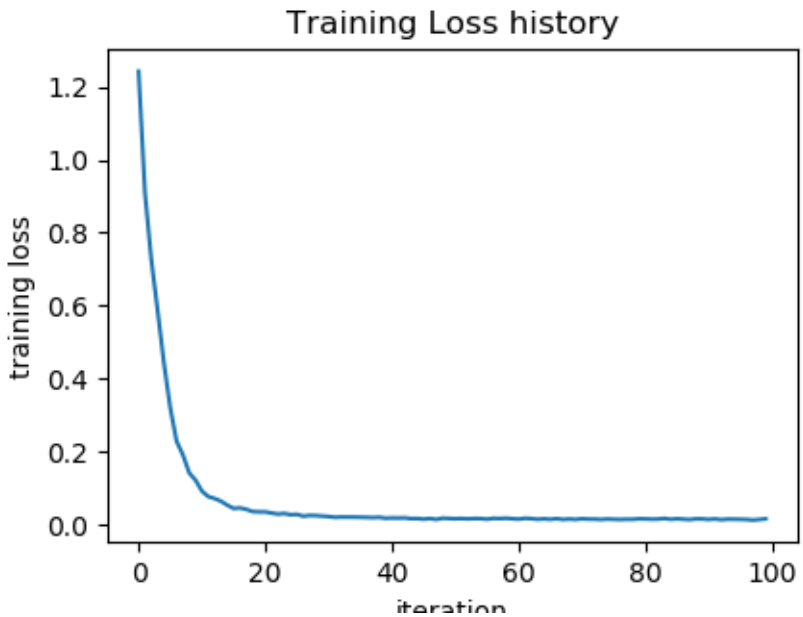
- $b^{(1)}$

$$\frac{\partial \tilde{J}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(1)}} = \frac{1}{N}\left(\psi\left(z^{(3)}\right) - \Delta\right)W^{(2)}\mathbf{I}\left[z^{(2)} > 0\right]'\mathbf{1}'$$

3 Stochastic gradient descent training

a)

After deriving all the gradients and the implementation of the backward part of our net, we implemented the stochastic gradient descent algorithm and run the training on the toy data. Our model obtained exactly the expected $loss = 0.02$, reaching it with the expected behaviour.

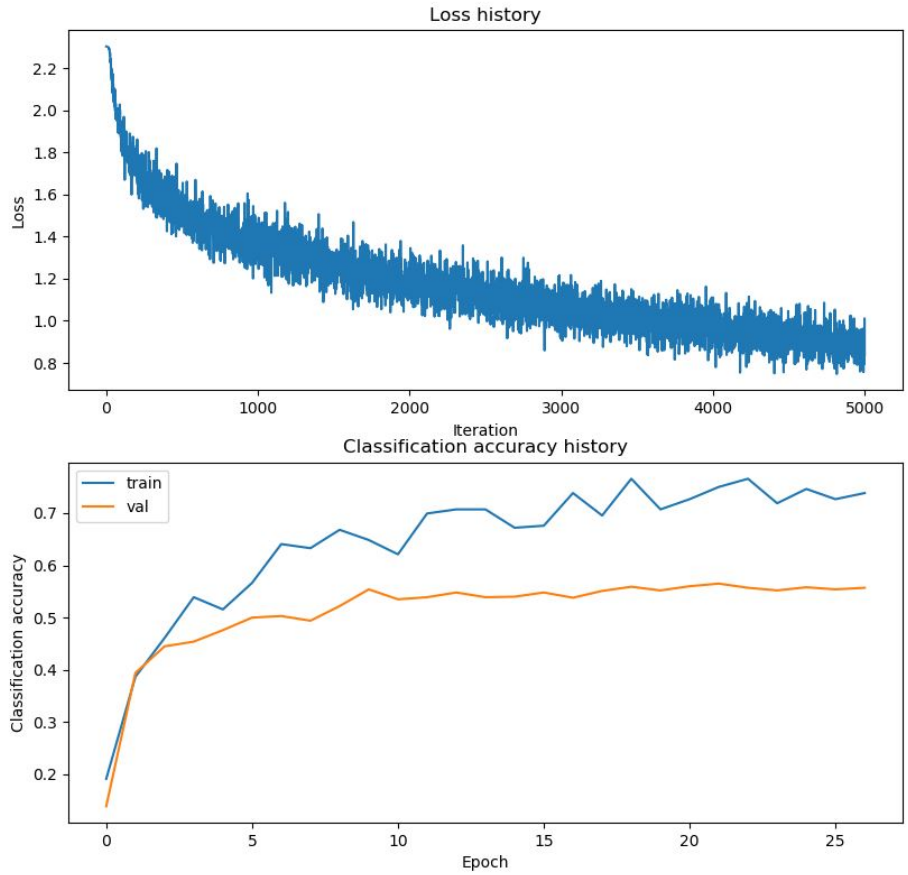


b)

From the given pointers it was clear that the network was not capable enough to handle the data, so our first idea was to increase the number of hidden neurons. Next, since the loss was not properly converging with the given epochs we also decided to have the network run for more epochs with a larger batch size. The learning rate and regularization term have instead gone down. After this consideration obtained by trial and error, we decided to set a proper grid search to find the best set of parameters within a subset decided in the exploration that we did by eye, and these are the results:

train accuracy	val accuracy	hidden size	iterations	batch size	learning rate	regularization
0.711367	0.563	250	5000	256	0.0010	0.0010
0.677694	0.553	250	4000	256	0.0010	0.0010
0.650571	0.553	250	5000	512	0.0010	0.0001
0.676857	0.551	250	4000	256	0.0010	0.0001
0.648612	0.548	250	5000	512	0.0010	0.0010
0.637531	0.546	250	4000	512	0.0010	0.0010
0.665939	0.544	200	4000	256	0.0010	0.0001
0.638959	0.543	250	4000	512	0.0010	0.0001
0.643163	0.542	200	5000	512	0.0010	0.0001
0.710776	0.540	250	5000	256	0.0010	0.0001
0.643184	0.540	200	5000	512	0.0010	0.0010
0.633082	0.536	200	4000	512	0.0010	0.0001
0.632755	0.534	200	4000	512	0.0010	0.0010
0.692286	0.531	200	5000	256	0.0010	0.0010
0.664857	0.530	200	4000	256	0.0010	0.0010
0.694714	0.523	200	5000	256	0.0010	0.0001
0.426939	0.434	250	5000	256	0.0001	0.0010
0.426816	0.434	250	5000	256	0.0001	0.0001
0.425163	0.431	200	5000	256	0.0001	0.0001
0.425061	0.431	200	5000	256	0.0001	0.0010
0.415510	0.422	250	4000	256	0.0001	0.0010
0.415245	0.421	250	4000	256	0.0001	0.0001
0.412286	0.420	200	4000	256	0.0001	0.0010
0.412306	0.420	200	4000	256	0.0001	0.0001
0.378755	0.383	250	5000	512	0.0001	0.0001
0.378776	0.383	250	5000	512	0.0001	0.0010
0.377000	0.379	200	5000	512	0.0001	0.0001
0.376980	0.379	200	5000	512	0.0001	0.0010
0.373714	0.375	250	4000	512	0.0001	0.0010
0.373714	0.375	250	4000	512	0.0001	0.0001
0.372020	0.373	200	4000	512	0.0001	0.0010
0.371959	0.373	200	4000	512	0.0001	0.0001

From this table it is clear that setting the learning rate as 10^{-3} is far better than setting it at 10^{-4}. For the others parameter we don't have strong preferences. Bigger hidden size and number of iteration seems to improve the performance (obviously at the cost of time and computational expense). We also can see a significant better performance over the train dataset with respect to the validation one, that can be analyzed better looking at the historical plot of the two accuracy (this plot refers to the best combination of parameters) :



We can see that soon the accuracy over the train set starts to increase more than then the one over the validation set, but this is in some sense OK because the validation accuracy continues to grow slow and smoothly.

We decided to take as our "chosen" best net the one with the best validation accuracy according to this grid. The best parameter obtained are:

- hidden size = 250
- number of iterations = 5000
- batch size = 256
- learning rate = 0.0010
- regularization coefficient = 0.0001

Although this is a good starting point, we know that just one simulation it should not be enough to decree this parameters as the global best, since this process is partly random.

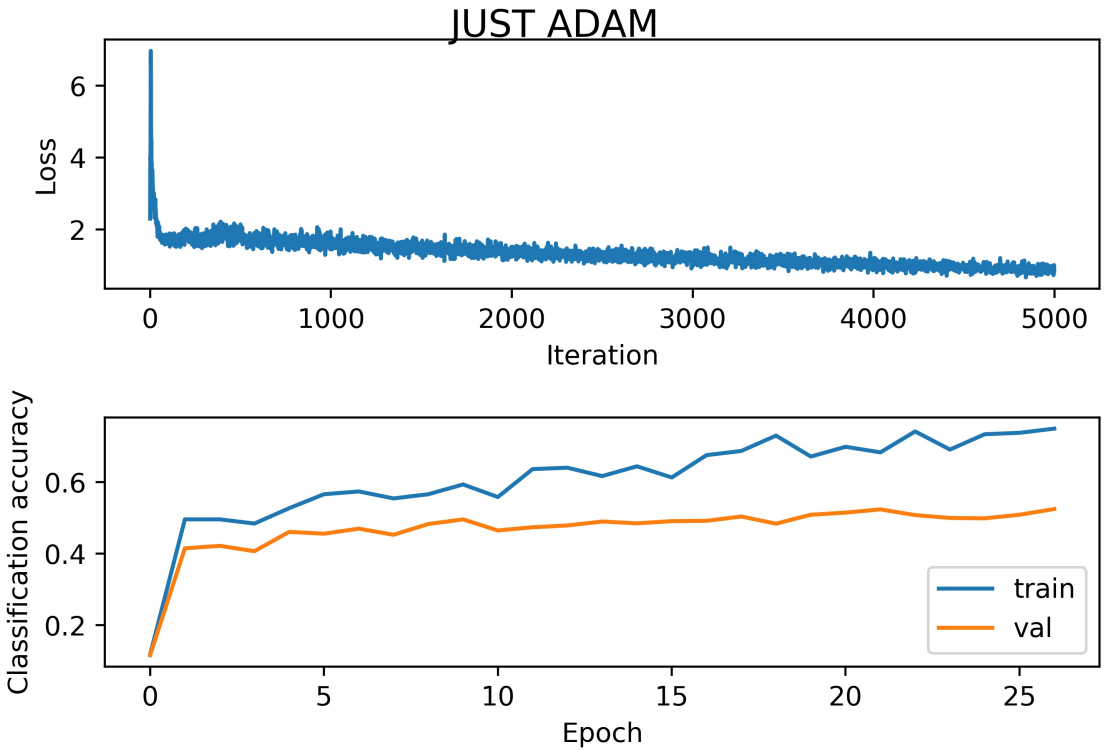
The best thing to do would be to take a lot of simulations for each combination of parameters and than take an average validation accuracy, but since this is not strictly the scope of this exercise we will avoid that to save time and computational power. For the sake of completeness we can at least show a table of 15 different runs of the combination with the best parameters to explore it's consistency:

Train accuracy	Validation accuracy
0.707224	0.545
0.708612	0.557
0.705041	0.552
0.704857	0.546
0.702816	0.557
0.709653	0.551
0.704959	0.562
0.708469	0.555
0.709265	0.541
0.707490	0.524
0.708286	0.540
0.705633	0.533
0.702776	0.552
0.704327	0.541
0.710714	0.563

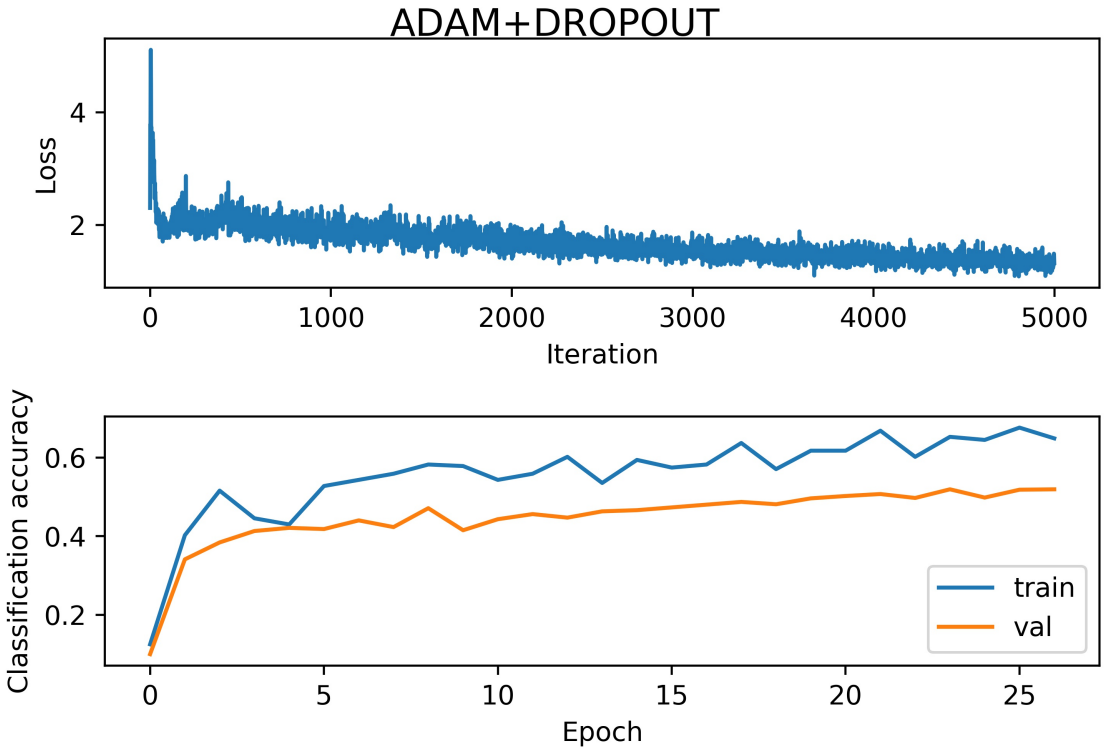
From the table above we see that, even in the worst case, our *best net* performs relatively good. Every one of these runs took roughly 130 seconds.

Going deeper

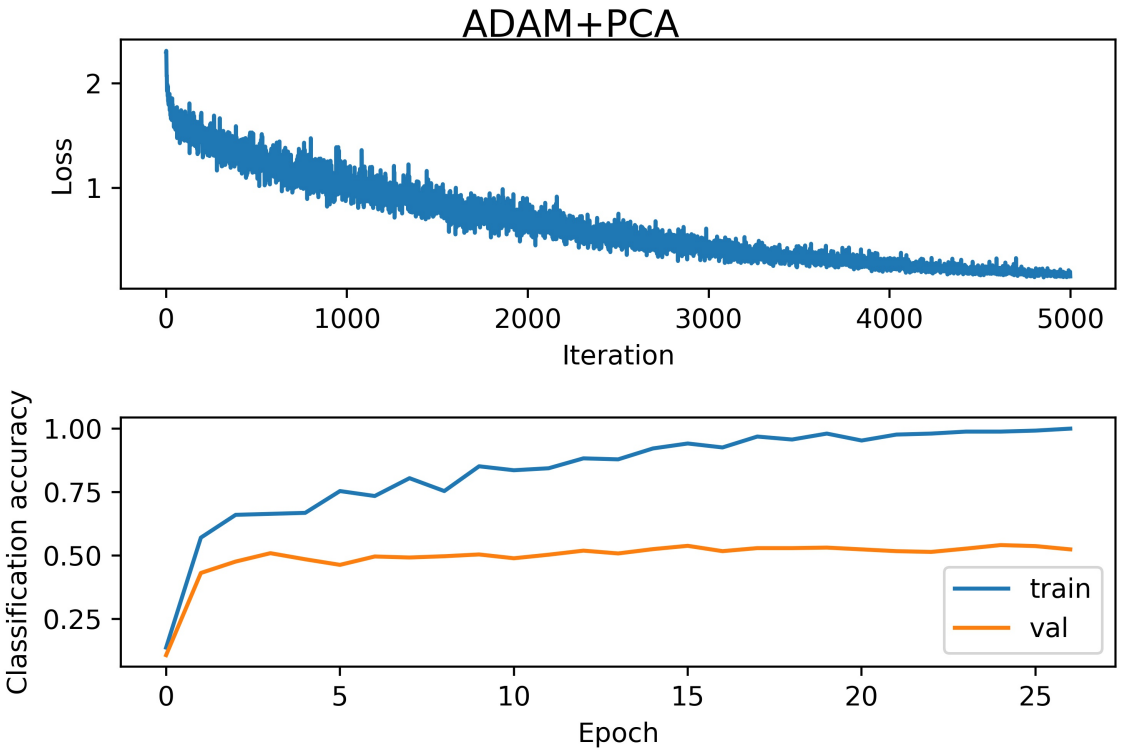
We reached a very good results, but we decided to create an advanced neural network that could help us to experiment more deeply with other techniques. The new class can be found in the script `two_layernet_advanced.py`,and is tested in the script `test.py`.
First, we implemented the PCA (with 400 saved components) to reduce the number of dimensions and speed up the computation, then we decided to implement a better optimizer using Adam which is widely regarded as a good default choice. The chosen parameter for Adam are: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-7$.
Wondering if more regularization was necessary we finally also added dropout to our network. These are the obtained results:



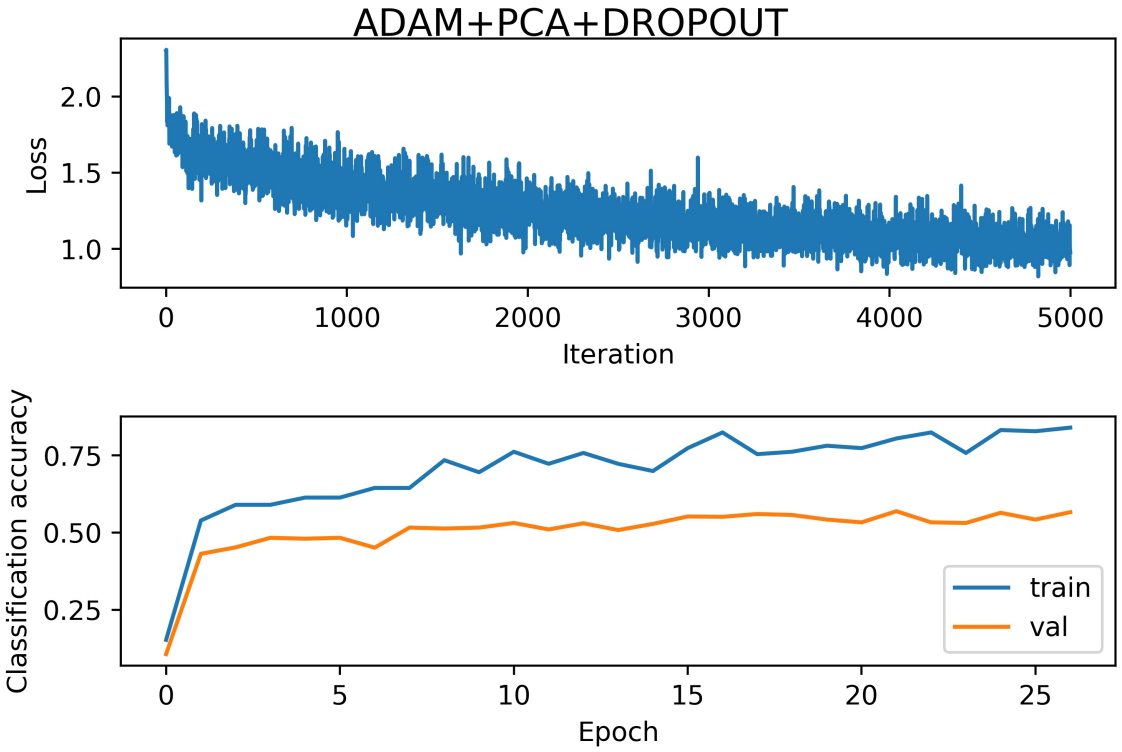
time in seconds: 252 validation accuracy: 0.532



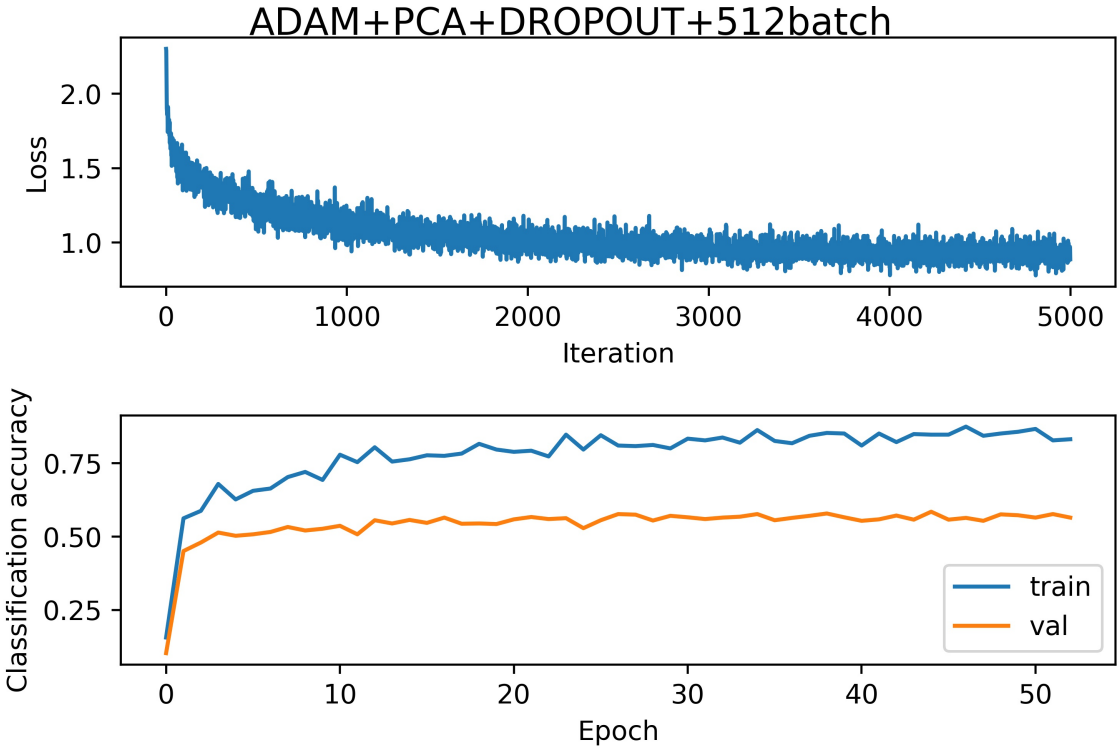
time in seconds: 255 validation, accuracy: 0.504



time in seconds: 37 validation, accuracy: 0.534



time in seconds: 34 validation, accuracy: 0.535



time in seconds: 59 validation, accuracy: 0.575

As we have seen with the previous result, it would be an error to be driven by the validation accuracy of just one run, since we have seen that it is highly influenced by the randomness of the algorithm (we have seen that a different seed can result in strong fluctuation of the train accuracy, even $\pm$ 0.02). The point of this last paragraph is not to find the best absolute net but to analyze different combinations of decisions to gain some insights.

Our plots show some interesting information.

From this experiment Adam doesn't show better performance *perse* (at least with the chosen parameters); since it doesn't improve results while being slower (almost $\times 0.5$ speed). PCA and dropout instead seems to have strong positve effects.

PCA, especially, is very powerful, speeding up our training by cutting the time needed to do each iteration to less than a quarter, while keeping good performance, and is clearly helped a lot by the dropout regularization that avoids that the algorithm collapses to a 0 loss, i.e. overfits too much.

4 Implement multi-layer perceptron using PyTorch library

a)

Two-layer Network

As reported in previous sections, it is possible to build a multi-layer perceptron (with forward and backward computations) using matrix multiplication implemented through *Numpy* library. However, there are several libraries expressly focused on the design of neural networks, as Pytorch. Pytorch is an open-source library based on Torch, developed by Facebook in order to provide a Python interface for fast tensor computing and deep neural network building (1).
In this section the two-layer network previously implemented has been re-created taking advantage of Pytorch, the code is available at `ex2_pytorch.py` file. First of all, the model architecture was defined in a class, basically the multi-layer perceptron consists of:

- Linear(in_features = 3072, out_features = 50, bias=True)
- ReLU()
- Linear(in_features = 50, out_features = 10, bias=True)

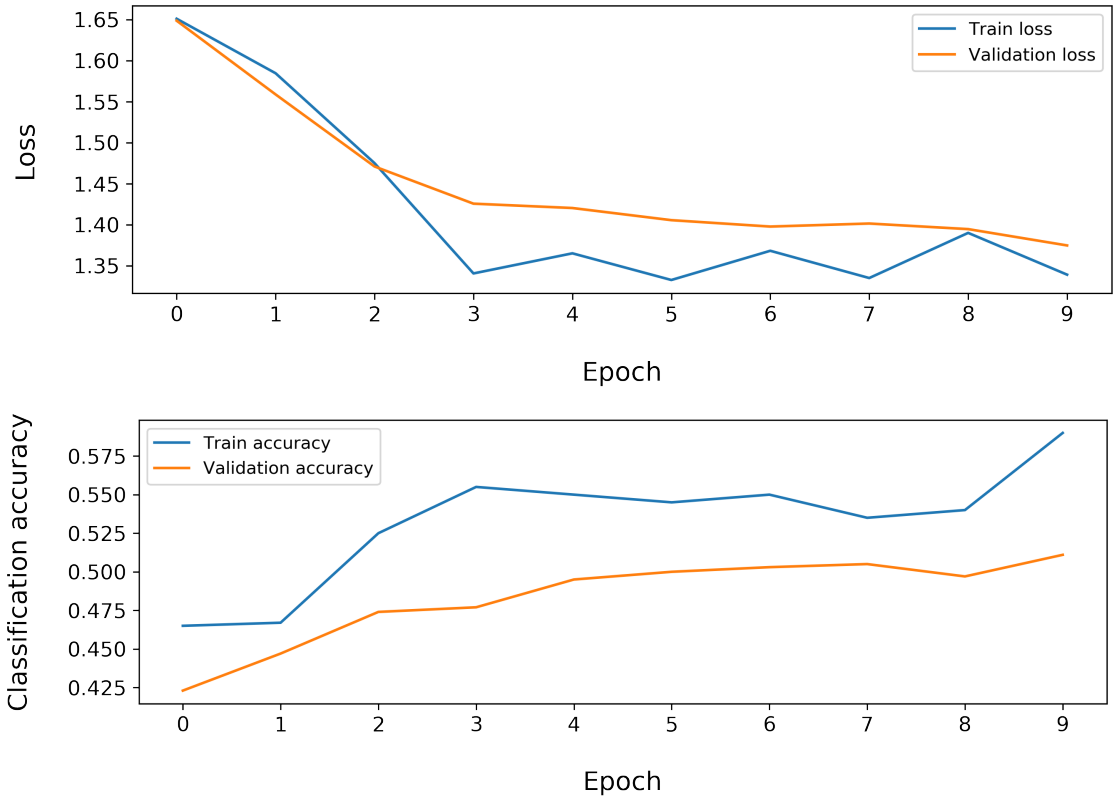
Indeed when defining a network is important to give the right tensor input and output shape for each layer. For instance, the input layer is expected to receive a tensor of shape 3072, since our images data have shape of $3 \times 32 \times 32$, they need to be reshaped before entering the network. The last layer outputs a tensor of dimension 10, since 10 is the number of the possible classification labels. Unlike the Numpy implementation, the *softmax* function does not need to be added at the end of the network, considering that the Pytorch Cross Entropy loss function already implements it while computing the loss.
Thereafter, the model has been initialized with random numbers from a Gaussian distribution with zero mean and 1e-03 standard deviation for the weights and 0 bias. Among the hyperparameters chosen, a batch size of 200 has been set as well as 10 as number of epochs. As specified before, Cross Entropy has been selected as loss function (a common choice for multi-class classification tasks), it is computed in Pytorch as in Equation 1 (note: it includes *softmax* function).

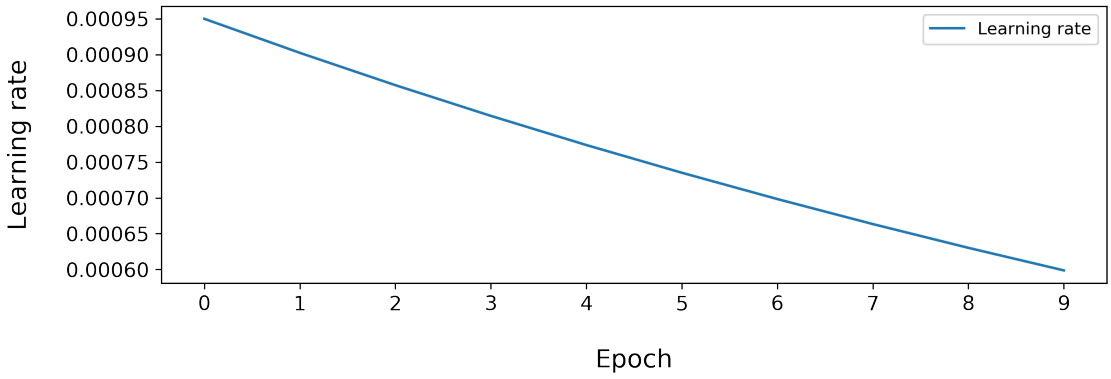
$$\text{loss}(x,\ \text{class}\ ) = -\log\left(\frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])}\right) = -x[\text{class}] + \log\left(\sum_j \exp(x[j])\right) \tag{1}$$

Instead, Adam algorithm has been chosen as method for stochastic optimization. This algorithm take advantage of some features of previous optimization methods, it exploit momentum by using moving average of the gradient (while SGD with momentum uses the

gradient itself) and takes advantage of the squared gradients to scale the learning rate like RMSprop, for more information about this algorithm, see Kingma and Ba (2). Adam relies on an adaptive learning rate, its value is initialized as 1e-03 and then updated after each epoch with a decay of 0.95. The weight decay (L2 penalty) instead is setted at 1e-03.
Then we proceed to train the network, compute the loss and optimize with back computation (i.e. zero gradients, perform a backward pass, and update the weights) in Pytorch this can be easily implemented with the few commands: `optimizer.zero_grad()`, `loss.backward()`, `optimizer.step()`. After the model has been trained for an epoch, the validation data are given in input to compute the validation accuracy reached so far. Once all epochs have been executed, the model is considered trained and optionally the test data can be passed to it to check test accuracy results. The results for train and validation of our implementation of the two-layer network are represented in Figure 1. The loss as well as the accuracy decrease at an higher rate in the first four epochs and this can be expected considering also learning rate values, in the last epochs the loss as the accuracies values are in a smaller range with more fluctuations. However looking at the plots, it can be noticed that we are not in presence neither of underfit nor of overfit. As result, we ended with train accuracy of 59%, a validation accuracy of 51.1% and a test accuracy of 50.2%.

Figure 1: Two-layer network. Loss and accuracy results for train and test, as well as learing rate decrease.





TensorBoard

To visualize information about a given neural network, a very useful tool can be Tensorboard. However, TensorBoard is a toolkit natively created for *TensorFlow* library, not for Pytorch. In order to create *logs* to input in TensorBoard we wrote a script `Tensorboard.py` in `Pytorch_improvements` folder, that using tensorboardX and torchbearer libraries make possible to create visualizations in TensorBoard interface. Just run *Tensorboard.py* and open the resulting link to access the TensorBoard panel with our results for a simplified version of the two-layer network.

c)

Improve MLP performance

In order to improve the performance of the Pytorch two-layer network created in Section a, many adjustments have been made. The code for this section has been implemented in `ex2_pytorch_grid_search.py`, part of `Pytorch_improvements` folder. First, we increase the number of epoch from 10 to 20 (the batch size remains unchanged) and add an **early stopping** mechanism. Basically in order to reduce training time of the model and achieve a better optimization, if the validation accuracy (plus a delta, fixed at 1e-04) value does not improve for 5 consecutive epochs (patience), the training is stopped and the weights associated to the higher validation accuracy value are saved. Another aspect that has been changed is the **number of neurons** per layer. In the two-layer version the hidden layer has 50 neurons. Even if there is not a predefined rule to choose the number of neurons, selecting few neurons could result in underfit while adding too many neurons could result in overfit. As consequence we decided to increase the number of neurons defining a function that taking account of the input size and the number of layers allocate a certain number of neurons to each layer. For instance, for 5 layers the first hidden layer has 250 neurons and the last hidden layer 50, while for 3 layers the fist hidden layer has 150 neurons and the last hidden layer 50, for a better understanding look at `n_neurons` function defined in `ex2_pytorch_grid_search.py`.

To decide which value assign to some hyperparameters we decided to implement a grid search: several values are given to each investigated hyperparameter, then for each combination of them the model was trained and the train and validation result are recorded. The hyperparameter searched are:

- **number of layers**. Increase network depth could be a way to improve the performance. Values considered are: $2, 3, 4, 5, 10$.

- **batch normalization**. The batch normalization is a technique implemented to reduce the internal covariate shift (the amount by what the hidden unit values shift). It should slightly reduce overfitting, allows to use higher learning rate and make the network more robust respect to the weight initialization used (3). For all these reasons using batch normalization (we would place it before each activation function) can be useful to improve the performance of the network.
  Values considered are: True, False.

- **weight initialization/activation function**. In the simple two-layer version the weights are initialized as random numbers from a Gaussian with mean 0 and standard deviation 1e-03. This initialization is fine for small networks but for deep network layers all the activations will tend to 0 (i.e. no learning). In order to avoid this problem we adopted two different weight initializations: the Xavier (4) and the Kaiming (5). In the Xavier initialization the weights are chosen from a random uniform or normal (we chose the normal) distribution that is bounded:

  $$w = \frac{np.random.rand(D_{in}, D_{out})}{\sqrt{D_{in}}} \tag{2}$$

  The Xavier initialization is a good initialization when the activation function is symmetric as the sigmoid or hyperbolic tangent. Instead for asymmetric activation function as the rectified linear unit (ReLU), this kind of initialization will make the activations collapse to zero. Indeed when ReLU and its variants are used, another weight initializaion is usually employed, the Kaiming:

  $$w = np.random.rand(D_{in}, D_{out}) \times \sqrt{\frac{2}{D_{in}}} \tag{3}$$

  In this way activations are scaled for every layer.
  Values considered are: Xavier & Tanh, Kaiming & ReLU, Kaiming & LeakyReLU (a variant of ReLU, solve the *dying ReLU problem*, basically the slope of ReLU function is changed to cause a *leak* and extend its range, in this way small positive gradients are allowed when the unit is not active).

- **dropout**. The dropout is a form of regularization. At each forward pass, randomly selected neurons are dropped with a certain probability (we would set it to 0.3). Dropout would be implemented just in training phase.
  Values considered are: True, False.

- **optimizer**. In the Pytorch two-layer network, Adam was used as optimizer, for more information about Adam look in Section a. We are going to compare the performance of Adam with the one of the very basic SGD (without momentum). Indeed the SGD should present problems in case of presence of saddle points or local minima in the loss function. We repeated here the comparison between Adam and SGD already done while chosing hyperparameters for the Numpy two-layer network, checking if in this situation the results are different.

In total the grid search investigate for 120 combinations. For visualization reasons we do not report in a table the results for all the combinations, Table 1 shows the best 20 combination for validation accuracy and avoiding overfitting. It is possible to visualize train and accuracy values for all the combinations looking at `MLP_grid_search.csv` file.
However, to make easier to access and understand the results `Check_combinations.py` file was created. Running it, it will ask the user to enter some hyperparameter values and then prints out train and validation loss/accuracy plots for the inserted combination (it is possible to try all the 120 combinations).
Table 1 has been produced by sorting results of grid search for validation accuracy values and filtering out all the combinations for which the difference between train and validation accuracy was too wide (a symptom for overfitting), then we show the first 20 results. First of all, the Adam optimizer has marked predominance respect to the SGD, that does never appear in the first positions. This result is different from the one obtained in the Numpy two-layer version, however it should be due to the fact that the two optimizer have been initialized with parameters changed respect to the Numpy version and that here the network is deeper (majority of combination with more than two layers). The dropout can lead to a good performance for same particular combinations, but generally speaking it is not present in the majority of the best results. Instead the presence of Batch Normalization seems to bring to the best train/validation results in our table. About the activation functions ReLU, LeakyReLU as well as the hyperbolic tangent lead for same combination to good accuracy values. The depth of the network instead does not seem to be crucial to achieve best results, since there are combination for which a two-layered network performs better than deeper ones. However we did not try here very deep network, maybe with many more layers the result could be different.

Table 1: Best 20 hyperparameter combinations with corresponding training and validation accuracies. The original dataframe has been sorted according to validation accuracy and keeping just values for which train and validation accuracy values are relatively close (to avoid overfitting).

	N. layers	Batch Norm.	Activation	Dropout	Optimizer	Train acc.	Val. acc.
1	10	True	LeakyReLU	False	Adam	66.0	56.1
2	5	True	LeakyReLU	True	Adam	58.0	55.7
3	4	True	ReLU	False	Adam	66.0	55.4
4	4	True	LeakyReLU	False	Adam	63.5	55.1
5	4	True	Tanh	False	Adam	61.0	54.3

	N. layers	Batch Norm.	Activation	Dropout	Optimizer	Train acc.	Val. acc.
6	5	True	Tanh	False	Adam	62.5	54.2
7	5	False	ReLU	True	Adam	54.0	53.6
8	3	True	Tanh	False	Adam	61.0	53.4
9	3	False	ReLU	False	Adam	59.0	52.8
10	10	True	Tanh	False	Adam	59.5	52.1
11	10	False	Tanh	False	Adam	63.0	52.1
12	2	False	ReLU	False	Adam	56.5	52.0
13	2	False	LeakyReLU	False	Adam	63.0	51.9
14	2	True	ReLU	False	Adam	56.5	51.5
15	2	False	ReLU	True	Adam	53.5	51.1
16	2	True	LeakyReLU	False	Adam	59.0	50.9
17	4	False	Tanh	False	Adam	58.5	50.3
18	3	False	Tanh	False	Adam	60.5	49.9
19	4	False	Tanh	True	Adam	50.0	48.8
20	2	True	Tanh	False	Adam	53.0	48.0

Unlike the combination we have described so far, there are others that lead to overfitting or poor accuracy results. For example the combination of 5 layers, LeakyReLU without batch normalization and dropout, has a good validation accuracy value, i.e. 57% but a train accuracy of 78% with more than 20 percentage points of difference. These values have not been considered as good candidates for hyperparameters choice since there is the possibility of overfitting. SGD as specified before in general leads to poor performances in confront of Adam, especially when it is combined with a deeper network, indeed some combinations of SGD and 10 layers bring to train and accuracy values both under 20%.

Finally, we decided that the **best combinations of hyperparameter** is:

- N. layers: 5.
- Batch norm.: True.
- Initalization & Activation: Kaiming & ReLU.
- Dropout: True.
- Optimizer: Adam.

This choice was made considering validation and train accuracy values and it was preferred to the 1st result of Table 1 because the combination chosen presents a smaller difference between train and validation accuracy as well a 5 layers instead of 10, meaning a shorter training time. However, as already said while searching best hyperparameters for the two-layer network implemented with *Numpy*, identifying best hyperparameters with a single simulation does not guarantee that the chosen ones are really the best possible combination

among the others implemented. Indeed, this process is in part random (i.e. the results are affected for example by to the random seed selected).

Considering the optimized weights for this combination, we run the evaluation on the test set, obtaining a test accuracy of 54.4% (to check it execute `test_accuracy.py` in `Pytorch_improvements` folder).

References

[1] A. Paszke *et al.*, *Automatic differentiation in PyTorch*, 2017, accessed on 03.03.2020. [Online]. Available: https://openreview.net/pdf?id=BJJsrmfCZ

[2] D. P. Kingma and J. L. Ba, *Adam: A method for stochastic optimization*, 2014, accessed on 03.03.2020. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf

[3] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015, accessed on 07.03.2020. [Online]. Available: https://arxiv.org/pdf/1502.03167v3.pdf

[4] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, 2010, accessed on 07.03.2020. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[5] K. He *et al.*, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015, accessed on 07.03.2020. [Online]. Available: https://arxiv.org/pdf/1502.01852.pdf