Advanced Machine Learning

Deep Neural Network and Backpropagation

Luca Scofano (1762509) Simone Marretta (1911358)
Nana Teukam Yves Gaetan (1741352) Daniele Trappolini (1710415)

November 2020

1 Question 2.a

1.1 Task and variable explanation

Verify that the loss function defined in Eq. (5) has gradient w.r.t $z_i^{(3)}$:

First of all let's define Eq. (5):

$$\frac{\partial J}{\partial z_i^{(3)}}(\theta, \{x_i, y_i\}_{i=1}^N) = \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i) \tag{1}$$

Where Δ_i is a matrix of K columns and N rows, and can be represented as:

$$(\Delta_i)_j = \begin{cases} 1 & if j = y_i \\ 0 & otherwise \end{cases} \tag{2}$$

Let's start by defining the initial equation of $J(\theta, \{x_i, y_i\}_{i=1}^N)$ as:

$$J(\theta, \{x_i, y_i\}_{i=1}^N) = \frac{1}{N} \sum_{i=1}^{N} -log[\psi(z_i^{(3)})] \tag{3}$$

Where:

$$\psi(z_i^{(3)}) = Softmax\,Transformation = \frac{exp(z_i^{(3)})_{yi}}{\sum_{j=1}^{k} exp(z_i^{(3)})_j} \tag{4}$$

1.2 Considering only one entry:

We only consider one datapoint $i\,\epsilon\,[1, ..., N]$. The Loss function that we are considering is:

$$J(\theta, x_i, y_i) = -log[\psi(z_i^{(3)})] = -log[\frac{exp(z_i^{(3)})_{yi}}{\sum_{j=1}^{k} exp(z_i^{(3)})_j}] \tag{5}$$

And we can rewrite this equation as follows:

$$J(\theta, x_i, y_i) = log \sum_{j=1}^{k} exp(z_j^{(3)}) - log\, exp(z_{yi}^{(3)}) \tag{5.1}$$

that could be simplified as:

$$J(\theta, x_i, y_i) = log \sum_{j=1}^{k} exp(z_j^{(3)}) - z_{yi}^{(3)} \tag{5.2}$$

At this point we can use the ***chain rule*** to ease the computation, but we have to define some variables first:

$$v = \sum_{j=1}^{k} exp(z_j^{(3)})$$
$$u = log(v)$$
$$\frac{\partial u}{\partial v} = \frac{1}{v}$$
$$w = z_{yi}^{(3)}$$

And the chain rule is:

$$\frac{\partial u}{\partial w} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial w} \tag{6}$$

By taking the equation (5.2) and applying what we've just seen we're left with:

$$\frac{\partial J}{\partial z_i^{(3)}}(\theta, x_i, y_i) = \frac{1}{v} \bigtriangledown_{z_i^3} \sum_{j=1}^{k} exp(z_j^{(3)}) = \frac{1}{\sum_{j=1}^{k} exp(z_j^{(3)})} exp(z_i^{(3)}) \tag{7}$$

Why did we get this result? $\bigtriangledown_{z_i^3} \sum_{j=1}^{k} exp(z_j^{(3)})$ is the derivative with respect to z_i, so all of the z_j's that are not z_i will have a derivative equal to *0* and when $z_i = z_j$ the derivative will be $exp(z_i^{(3)})$

We can now finalize our calculation by looking at equation (4), our softmax transformation, and substituting it in our result:

$$\frac{\partial J}{\partial z_i^{(3)}}(\theta, x_i, y_i) = \psi(z_i^{(3)}) - \delta \tag{8}$$

Where δ_i is a vector of K dimensions with:

$$\delta_i = \begin{cases} 1 & if\, i = y_i \\ 0 & otherwise \end{cases} \tag{9}$$

1.3 Considering all N entries:

We can consider the case in which we are evaluating over the whole N datapoints by extending what we've said in the previous subsection, but obviously, we are now dealing with matrices. (***Reminder***, we are now considering equation (3) as our Loss Function)

Our goal is to verify that

$$\frac{\partial J}{\partial z_i^{(3)}}(\theta, \{x_i, y_i\}_{i=1}^{N}) = \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i) \tag{10}$$

As we did before we can use the ***chain rule*** to compute all of the partial derivatives of our final matrix:

$$\begin{aligned} \frac{\partial J}{\partial z_i^{(3)}}(\theta, \{x_i, y_i\}_{i=1}^{N}) &= \frac{1}{N} \bigtriangledown_{z_i^3} \sum_{i=1}^{N} (log \sum_{j=1}^{k} exp(z_j^{(3)}) - z_{yi}^{(3)}) \\ &= \frac{1}{N}(\psi(z_1^{(3)}) - \delta_1 + ... + \psi(z_N^{(3)}) - \delta_N) \end{aligned} \tag{11}$$

Rewriting it in matrix form we get:

$$\frac{\partial J}{\partial z_i^{(3)}}(\theta, \{x_i, y_i\}_{i=1}^N) = \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i) \tag{12}$$

Where Δ_i is equal to equation (2).

2 Question 2.b

2.1 Not Regularized Loss

We have to verify that:

$$\frac{\partial J}{\partial W^{(2)}}(\theta, \{x_i, y_i\}_{i=1}^N) = \sum_{i=1}^{N} \frac{\partial J}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial W^{(2)}} = \sum_{i=1}^{N} \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i) a_i^{(2)^T} \tag{13}$$

As we did previously we should use the Chain Rule (as the task suggests), since the first partial derivative has been computed in sub-task 2.3 (Eq. (12)) we are left with the computation of $\frac{\partial z_i^{(3)}}{\partial W^{(2)}}$.

For the second part of the chain rule we have to cmpute $\frac{\partial z_i^{(3)}}{\partial W^{(2)}}$. Since $z_i^{(3)}$ could be written as $W^{(2)}a^{(2)} + b^{(2)}$, deriving it with respect to $W^{(2)}$ gives us $a^{(2)^T}$, the equation has then been verified.

2.2 Regularized Loss

We have to verify that:

$$\frac{\partial \tilde{J}}{\partial W^{(2)}}(\theta, \{x_i, y_i\}_{i=1}^N) = \sum_{i=1}^{N} \frac{\partial \tilde{J}}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial W^{(2)}} = \sum_{i=1}^{N} \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i) a_i^{(2)^T}) + 2\lambda W^{(2)} \tag{14}$$

The equation in question is:

$$\tilde{J}(\theta, \{x_i, y_i\}_{i=1}^N) = \frac{1}{N} \sum_{i=1}^{N} -log[\psi(z_i^{(3)})] + \lambda \left(\left\| W^{(1)} \right\|_2^2 + \left\| W^{(2)} \right\|_2^2 \right) \tag{15}$$

The regularization part can be rewritten as

$$\lambda\left(\left\|W^{(1)}\right\|_2^2+\left\|W^{(2)}\right\|_2^2\right)=\lambda\left(W^{1^T}W^1+W^{2^T}W^2\right) \tag{16}$$

And the derivative will be:

$$\begin{aligned}\frac{\partial\tilde{J}}{\partial W^{(2)}}(\theta,\{x_i,y_i\}_{i=1}^N)&=\frac{\partial J}{\partial W^{(2)}}+2\lambda W^{(2)}\\&=\sum_{i=1}^N\frac{1}{N}(\psi(z_i^{(3)})-\Delta_i)a_i^{(2)^T})+2\lambda W^{(2)}\end{aligned} \tag{17}$$

3 Question 2.c

At this point we would like to complete the back-propagation process by computing the partial derivatives of the entire parameter set $\theta=(W^{(1)},b^{(1)},b^{(2)})$ with respect to the regularized loss equation $\tilde{J}$ (Eq. 16). We'll use the ***chain rule*** as well for these sub-tasks.

3.1 Derivative of $b^{(2)}$ with respect to J

$$\frac{\partial J}{\partial b^{(2)}}=\frac{\partial J}{\partial z^{(3)}}\cdot\frac{\partial z^{(3)}}{\partial b^{(2)}}=\frac{1}{N}(\mathbf{1}(\psi(z^{(3)})-\Delta_i)) \tag{18}$$

We found the first partial derivative, $\frac{\partial J}{\partial z_i^{(3)}}$ in Eq.(12), and it's easy to see that $\frac{\partial z_i^{(3)}}{\partial b^{(2)}}$ is a vector of **1's** since $z_i^{(3)}=W^{(2)}a^{(2)}+b^{(2)}$

3.2 Derivative of $W^{(1)}$ with respect to $\tilde{J}$

$$\frac{\partial\tilde{J}}{\partial W^{(1)}}=\frac{\partial\tilde{J}}{\partial z^{(3)}}\cdot\frac{\partial z^{(3)}}{\partial a^{(2)}}\cdot\frac{\partial a^{(2)}}{\partial z^{(2)}}\cdot\frac{\partial z^{(2)}}{\partial W^{(1)}} \tag{19}$$

- $\frac{\partial\tilde{J}}{\partial z^{(3)}}=\sum_{i=1}^N\frac{1}{N}(\psi(z_i^{(3)})-\Delta_i)+2\lambda W^{(2)}$
- $\frac{\partial z^{(3)}}{\partial a^{(2)}}=W^{(2)^T}$
- $ReLU\ function\ involved$
  $\frac{\partial a^{(2)}}{\partial z^{(2)}}=\mathbf{I}[z^{(2)}>0]$
- $\frac{\partial z^{(2)}}{\partial W^{(1)}}=\mathbf{a^{(1)^T}}$

Where $\mathbf{I}[z^{(2)} > 0]$ is equal to $\boldsymbol{1}$ when $z^{(2)} > 0$ or $\boldsymbol{0}$ otherwise.

The final result is:

We are going to simplify the notation:

- $Part1 = \frac{1}{N}((\psi(z_i^{(3)}) - \Delta_i)W^{(2)^T} \odot \mathbf{I})$

Part1 is a matrix with dimensions **N x n. of weights**

$$\frac{\partial \tilde{J}}{\partial W^{(1)}} = Part1^T a^{(1)^T} + 2\lambda W^{(2)} \tag{20}$$

3.3 Derivative of $b^{(1)}$ with respect to J

$$\frac{\partial J}{\partial b^{(1)}} = \frac{\partial \tilde{J}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(1)}} \tag{21}$$

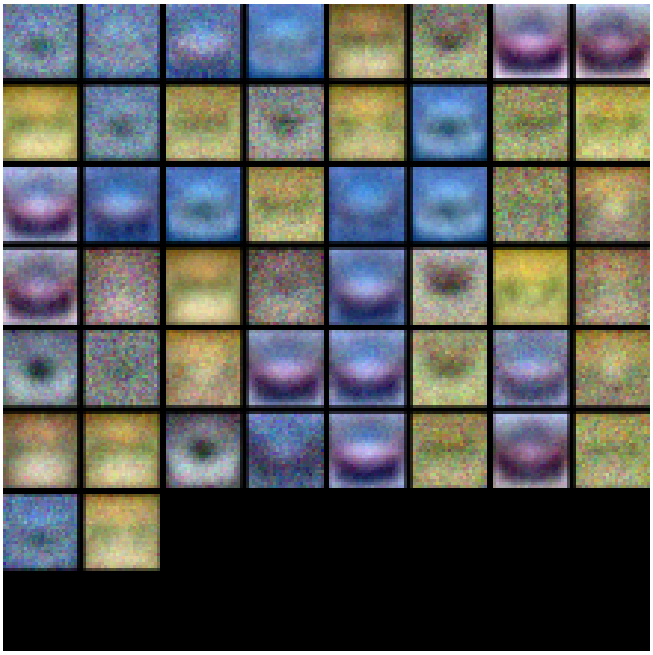
- $\frac{\partial J}{\partial z^{(3)}} = \sum_{i=1}^{N} \frac{1}{N}(\psi(z_i^{(3)}) - \Delta_i)$
- $\frac{\partial z^{(3)}}{\partial a^{(2)}} = W^{(2)^T}$
- $ReLU\ function\ involved$
  $\frac{\partial a^{(2)}}{\partial z^{(2)}} = \mathbf{I}[z^{(2)} > 0]$
- $\frac{\partial z^{(2)}}{\partial b^{(1)}} = \mathbf{1^T}$

The final result is:

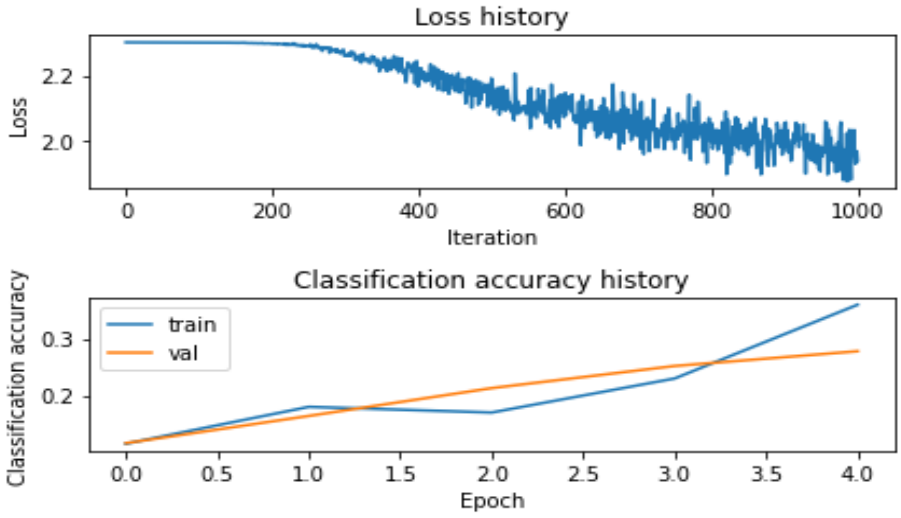
$$\frac{\partial \tilde{J}}{\partial W^{(1)}} = Part1^T \mathbf{1^T} + 2\lambda W^{(2)} \tag{22}$$

4 Question 3.b

Our task was to debug the model training and come up with better hyper-parameters to improve the performance on the validation set (**Fig.1** shows the initial situation that we should improve).



(a) Features



(b) Loss History/Iterations and Accuracy/epochs

Figure 1: Results with random initialization

The hyper-parameters we decided to tune are the following:

1. Batch size
2. Number of iterations
3. Hidden layer size
4. Learning Rate
5. Regularization penalization

4.1 Batch Size

The stochastic gradient descent method operates in a small-batch regime wherein a fraction of the training data (usually 32 to 512 data points) is sampled from the training set to compute an approximation of the gradient. At priori we can say, since it is largely shown, that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize (hence it will overfit the training data), we'll check is it's the case for our data-set as well.

We have three different approaches on how to pick the batch's size:

- Batch gradient descent, $B = |x|$
- Online stochastic gradient descent: $B = 1$
- Mini-batch stochastic gradient descent: $B > 1 \, but \, B < |x|$.

(Note: Where $\boldsymbol{x}$ *is our training set and* $\boldsymbol{B}$ *the batch size)*

What is the optimal size?

Interestingly, stochastic gradient descent (SGD) with small batch sizes appears to locate minima with better generalization properties than large-batch SGD and shown to have faster convergence to acceptable solutions.

On the other end, using a batch equal to the entire dataset guarantees convergence to the global optima of the objective function. However, this is at the cost of slower, empirical convergence to that optima.

Searching on the web[(1)] we found an alternative explanation on why training with larger batch sizes leads to lower test accuracy? One hypothesis might be that the training

samples in the same batch interfere (compete) with each others' gradient. One sample wants to move the weights of the model in one direction while another sample wants to move the weights the opposite direction. Therefore, their gradients tend to cancel and you get a small overall gradients. Perhaps if the samples are split into two batches, then competition is reduced as the model can find weights that will fit both samples well if done in sequence. In other words, sequential optimization of samples is easier than simultaneous optimization in complex, high dimensional parameter spaces.

Optimal batch size: 256

4.2 Number of iterations

The number of iterations could allow us to converge to an optimal solution, but there is a trade-off to keep in mind, *speed*. Most times we need just the right number of iterations, like in our case.

Optimal number of iterations: 3000

4.3 Hidden layer size

So what about size of the hidden layer(number of neurons)? There are some empirically-derived-go-to rules, as for example, we should pick the number of layers between the size of the input and the size of the output layer.

Optimal number of hidden layers: 200

4.4 Learning rate

Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect to the loss. There is a trade-off to consider when we pick a small learning rate, we might not miss the local minima, but on the downside it might take a longer time to converge, especially if we get to a plateau region.

How do we find an optimal learning rate?

In Section 3.3 of "Cyclical Learning Rates for Training Neural Networks."[(2)], Leslie N. Smith argued that you could estimate a good learning rate by training the model

initially with a very low learning rate and increasing it (either linearly or exponentially) at each iteration.

Some works in the optimization literature have shown that increasing the learning rate can compensate for larger batch sizes. We tried that out without having compelling results, we decided to ignore it and stick with our best model.

When our inputs are images, a poorly set learning rate leads to noisy features (Fig 1.) Quite the contrary, smooth, clean and diverse features are the result of a well tuned learning rate (Fig 2.)

Optimal learning rate: 0.001

4.5 Regularization penalization

When we are using Stochastic Gradient Descent (SGD) to fit our network's parameters to the learning problem at hand, we take, at each iteration of the algorithm, a step in the solution space towards the gradient of the loss function $J(\theta, X, y)$ in respect to the network's parameters θ, in order to minimize the function. This method could cause overfitting on our training data since the space of deep neural networks is very rich.

This overfitting may result in significant generalization error and bad performance on unseen data (or test data, if we are creating a model). Fortunately regularization techniques can be used in order to deal with this issue.

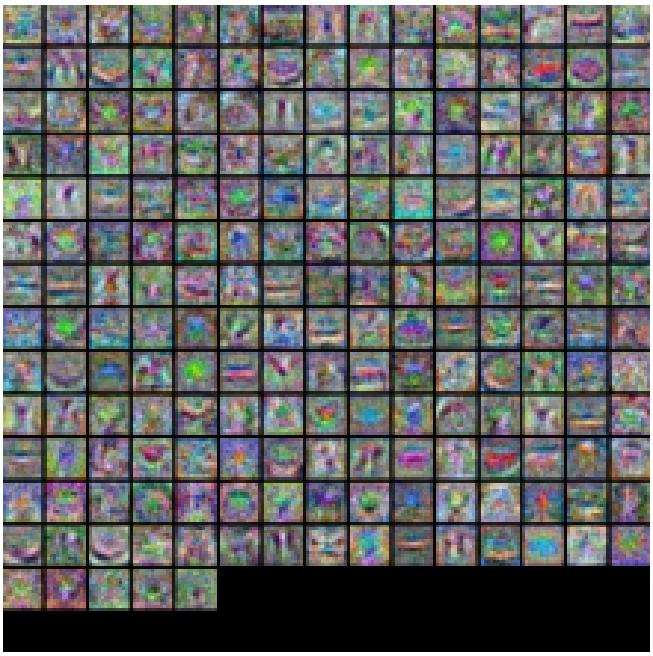
A common method in statistics and machine learning is to add a regularization term to the loss function, meant to incorporate a measure of model complexity into the function to be minimized.

Tim Roughgarden, Professor of Computer Science and member of the Data Science Institute at Columbia University, summarized the entire process saying that we become "biased toward simpler models, on the basis that they are capturing something more fundamental, rather than some artifact of the specific data set". Adding the $\mathbf{L^2}$ term usually results in much smaller weights across the entire model.

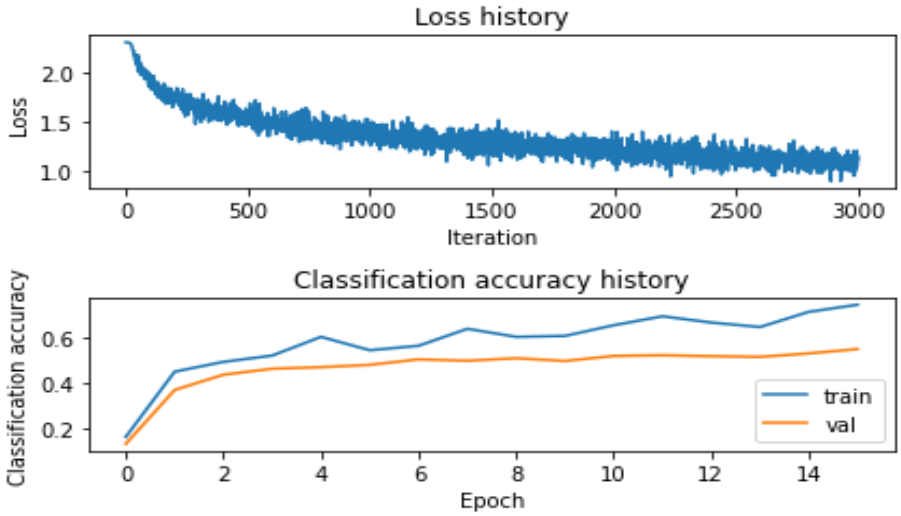
A higher parameter λ will give more importance to the regularization, and this is the hyper-parameter we should set.

Optimal regularization penalization: 0.001

4.6 Final considerations



(a) Features



(b) Loss History/Iterations and Accuracy/epochs

Figure 2: Results with best combination initialization

Reminder, the combination tuple is formed as followed:

- Hidden size
- Number of iterations
- Batch size

- Learning rate
- Regularization parameter

Top 10 results

combinations	train accuracy	val accuracy
(200, 3000, 256, 0.001, 0.001)	0.687408	0.552
(300, 4000, 512, 0.001, 0.001)	0.673837	0.551
(300, 4000, 512, 0.001, 0.01)	0.642102	0.548
(300, 4000, 256, 0.001, 0.01)	0.679082	0.547
(300, 4000, 512, 0.001, 0.0001)	0.644429	0.546
(200, 4000, 256, 0.001, 0.0001)	0.665939	0.544
(100, 4000, 128, 0.001, 0.01)	0.615327	0.544
(300, 3000, 512, 0.001, 0.01)	0.621551	0.543
(300, 3000, 512, 0.001, 0.001)	0.623551	0.543
(200, 3000, 256, 0.001, 0.01)	0.637571	0.542

Bottom 10 results

combinations	train accuracy	val accuracy
(100, 4000, 512, 0.01, 0.0001)	0.100265	0.087
(100, 4000, 512, 0.01, 0.001)	0.100265	0.087
(100, 4000, 512, 0.01, 0.01)	0.100265	0.087
(100, 4000, 256, 0.01, 0.0001)	0.100265	0.087
(100, 4000, 256, 0.01, 0.001)	0.100265	0.087
(100, 4000, 256, 0.01, 0.01)	0.100265	0.087
(100, 4000, 128, 0.01, 0.0001)	0.100265	0.087
(100, 4000, 128, 0.01, 0.001)	0.100265	0.087
(100, 4000, 128, 0.01, 0.01)	0.100265	0.087
(200, 4000, 128, 0.01, 0.01)	0.100265	0.087

As we said previously, the highest score on the validation and test set was the one with the intermediate batch size, since the smallest (128) produced the worst results because the dimension of the initial data-set is quite large, on the other hand the 512 size came in close second. We tried to tune the learning rate (taking a smaller one) of the second combination, trying to improve the score, but it actually produced worst results.

But taking a closer look we can surely say that the first combinations gives us the best balance between speed and results, since it has the smallest number of iterations out of the possible ones we picked and has a smaller size of the number of weights and biases contained in the hidden layer.

4.7 Recap:

Possible combinations are formed by:

- Hidden size = (100, **200**, 300)
- Number of iterations = (**3000**, 4000, 5000)
- Batch size = (128, **256**, 512)
- Learning rate = (0.01, **0.001**, 0.0001)
- Regularization parameter = (0.01, **0.001**, 0.0001)

With our best combination we got an **Accuracy on test set = 0.545**

Question 4: Implement multi-layer perceptron using PyTorch library

In this section we implemented the same two-layer network we did before but this time we used PyTorch. The implementation was quite straightforward using nn.Sequential, nn.Layers, nn.ReLU, etc. After training our network with 10 epochs, we achieved final validation accuracy of 51.3%. Here is a screanshot of the architecture of our model.

```
MultiLayerPerceptron(
  (layers): Sequential(
    (0): Linear(in_features=3072, out_features=50, bias=True)
    (1): ReLU()
    (2): Linear(in_features=50, out_features=10, bias=True)
  )
)
```

Figure 3: two-layer network

In order to improve the performance of our model we tried different combinations of parameters. The parameters we modified are: $Number\ of\ Layers$, $Size\ of\ the\ layers$, $Activation\ function$, and $Batch\ normalization$.

In the initial two layer network, the hidden layer had 50 neurons. It is known that there is not a perfect way of choosing the number of neurons in the hidden layers, since by experience we know that too few neurons can result in underfitting while having too many can result in overfitting. Therefore, we decided to use fixed number neurons in the hidden layer. That's means in the case of a five hidden layers, we always choose (arbitrarily) the set $[300, 250, 100, 50, 25]$ while in case of two hidden layers we chose $[50, 25]$. Moreover, the number of hidden layers chosen are: $2, 3, 4, 5, 6, 7$. To conclude, another reason we decided to play around with the size of the hidden layers, is because increasing the depth of the network can be a way to improve the performance.

Batch normalization is another hyperparameter we decided to play around with. As we know, Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). This technique also allows each layer of a network to learn by itself a little bit more independently of other layers.

The third parameter we changed in order to improve the performance of our model is the Activation function. They are a crucial component of deep learning. Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency of training a model—which can make or break a large scale neural network. Activation functions also have a major effect on the neural network's ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place.

The Activation functions chosen are:

- ReLU (Rectified Linear Unit).
- Leaky ReLU
- Sigmoid / Logistic

Layers	**Hidden layers**	**Batch Norm**	**Act Funct**	Val Acc
5	[250, 100, 50, 25]	True	ReLU	54.9
6	[300, 250, 100, 50, 25]	True	LeakyReLU(negative_slope=0.01)	54.7
7	[500, 300, 250, 100, 50, 25]	True	ReLU	54.4
6	[300, 250, 100, 50, 25]	True	ReLU	54.4
7	[500, 300, 250, 100, 50, 25]	True	LeakyReLU(negative_slope=0.01)	54.3
3	[50, 25]	True	LeakyReLU(negative_slope=0.01)	53.1
5	[250, 100, 50, 25]	True	LeakyReLU(negative_slope=0.01)	53.0
4	[100, 50, 25]	True	LeakyReLU(negative_slope=0.01)	52.7
4	[100, 50, 25]	True	ReLU	52.7
3	[50, 25]	False	ReLU	52.1

Table 1: Best Combinations

Layers	**Hidden layers**	**Batch Norm**	**Act Funct**	Val Acc
6	[300, 250, 100, 50, 25]	False	LeakyReLU(negative_slope=0.01)	10.0
5	[250, 100, 50, 25]	False	LeakyReLU(negative_slope=0.01)	10.0
7	[500, 300, 250, 100, 50, 25]	False	LeakyReLU(negative_slope=0.01)	10.0
7	[500, 300, 250, 100, 50, 25]	False	ReLU	10.0
4	[100, 50, 25]	False	ReLU	9.0
5	[250, 100, 50, 25]	False	Sigmoid	9.0
4	[100, 50, 25]	False	LeakyReLU(negative_slope=0.01)	9.0
6	[300, 250, 100, 50, 25]	False	ReLU	9.0
5	[250, 100, 50, 25]	False	ReLU	9.0
7	[500, 300, 250, 100, 50, 25]	False	Sigmoid	8.6

Table 2: Worse combinations

After grid search we had 36 combinations. Looking at Table 1 / 2 we can notice that Batch Normalization seems to bring to the best train/validation results. Regarding the activation functions ReLU and LeakyReLU are those giving us the best performances. To conclude, the number of layers does not seem to be crucial to achieve our best results, since there are combination for which a two-layered network performs much better than a deeper network.

5 Bibliography:

1. Effect of batch size on training dynamics, Kevin Shen 2018, Medium.com

2. Cyclical Learning Rates for Training Neural Networks, Leslie N. Smith, 2015