Assignment 1. MLPs and Backpropagation

University of Amsterdam – Deep Learning Course

October 28, 2024

The deadline for this assignment is November 13$^{\text{th}}$ at 23:59.

In this assignment you will learn how to implement and train the basic MLP neural architecture for a classification task. You will make use of modern deep learning libraries that come with sophisticated functionalities like abstracted layer classes, automatic differentiation, optimizers, etc. However, to gain an in-depth understanding of the training procedure of a MLP, we will first focus on the mathematics behind backpropagation and derive the necessary equations from the ground up.

- Section 1 and 2 provide a background on the mathematics behind training a neural network.
- In section 3, you will derive the equations necessary for backpropagation and implement these equations manually in Python using NumPy.
- In section 4 you will implement an MLP in PyTorch and analyze its performance.

The goal of the first sections in this assignment is to refresh your memory of index notation as it is used in linear algebra. In section 3, vector calculus will be applied to an MLP in order to derive the equations of backpropagation for the basic modules in a vanilla neural network. We will need a good understanding of index manipulation in order to handle calculus with objects of arbitrary rank. The rank of an array refers to the dimensionality of its inherent structure: a scalar s has rank 0, a vector $\mathbf{v}$ has rank 1 (v_i), a matrix $\mathbf{M}$ has a rank of 2 (M_{ij}). Note the number of independent indices. An array of higher rank is often referred to as a *tensor*[1]. As such, the object $\mathbf{T}$ with elements T_{ijk} could be referred to as a 3-rank tensor. As will become clear early on, the most important takeaway of working with tensors is to keep good algebraic hygiene throughout your calculations.

1 Index Gymnastics: Notation

The key to performing calculus with objects from linear algebra is to remember that the algebra in index representation is always the same, no matter how you define the shapes of the gradients. As conventions can change from textbook to textbook and paper to paper, it is a good skill to be able to understand how these equations look at the element-level. We will stick to performing calculations with indices from the start, and resort to the luxury of aesthetics only in the end. Remember that these results need to be coded up, so our priority should go to ease of implementation.

Let us begin with some basic notation. One of the most important objects in our arsenal is the *Kronecker delta*[2], which has the power to encode if-statements into our mathematical equations:

[1] Note that this is not the same object as referred to by physicists and mathematicians. In that sense, a tensor is a multi-linear map and should obey certain transformation laws in order to ensure basis-independence. For example, the Christoffel symbols form an array Γ, but this is not a tensor in the physical sense. Our notion of a tensor as a multidimensional array is not as restrictive.

[2] **Leopold Kronecker** (1823-1891) was a German mathematician and an avid number theory fanatic. He has been quoted as saying "God made the integers, all else is the work of man".

$$\delta_{ij} \coloneqq \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{1}$$

When used in a sum, this object has the useful property of selecting or *sifting* the terms that satisfy the equality of its indices. For example, let $\mathbf{a} \in \mathbb{R}^n$ be an arbitrary vector, then

$$\sum_{i=1}^{n} a_i \delta_{ik} = a_1\delta_{1k} + \ldots + a_k\delta_{kk} + \ldots + a_n\delta_{nk} = a_k.$$

In code:

```
a = [1, 3, 7, 4] # our vector

result = 0 # initialize
for i in range(1, n):
    if i == k:
        delta = 1
        result += a[i]*delta
    else:
        delta == 0
        result += a[i]*delta
```

Note that i is a *dummy index*: It can be renamed without any consequences to the truthfulness of the equation. The other index k, is a *free index* and needs to be present on both sides of the equation. It cannot simply disappear! As calculations become more involved, one needs to carefully keep track of which indices are free and which are summed over. Another crucial observation is the following:

$$\frac{\partial x_i}{\partial x_j} = \delta_{ij}.$$

Hint: If this observation feels like a jump, we encourage you to interactively discuss/search/ask to find what it means. Think about what it means for two *components* of a vector to change relative to each other, which is what

$$\frac{\partial x_i}{\partial x_j}$$

is.

This introduces calculus into our set of operations. Note that even though x_3 and x_7 are both elements of a vector called $\mathbf{x}$, the derivative of one with respect to the other is still 0 (because how one part of the vector changes, doesn't affect another part of the vector (we are assuming they are independent variables here)). They are independent variables which happen to have been collected into the same array.

Another subtle trick is quite an obvious one: *indexing*. Given a complicated looking object, analyzing it element by element will prove to be very efficient. For example, given matrix $\mathbf{M}$ its elements can be accessed by indexing using square brackets as follows: $[\mathbf{M}]_{ij} = M_{ij}$. In pythonic syntax, M[i][j]. Observe the relationship between the identity matrix $\mathbf{I}$ and the Kronecker delta: $[\mathbf{I}]_{ij} = \delta_{ij}$. Hint: where in a matrix are i and j the same? What does the identity matrix look like?

Useful notation: Yet another useful piece of notation is that for the *trace* of a square matrix $\mathbf{S} \in \mathbb{R}^{m\times m}$, i.e. $\mathrm{tr}(S) \coloneqq \sum_i S_{ii}$. Sometimes it will be useful to introduce the *ones-vector* $\mathbf{1}$, which simply has all components equal to unity $[\mathbf{1}]_i = 1$. The *Hadamard product*[3] or element-wise product between two matrices of identical size is given by $\mathbf{A} \circ \mathbf{B}$. The elements of the result are $[\mathbf{A} \circ \mathbf{B}]_{ij} = A_{ij}B_{ij}$.

[3] **Jacques Salomon Hadamard** (1865-1963) was a French mathematician and foreign member of both the Royal Society of London and the Royal Netherlands Academy of Arts and Sciences. His work was mainly in differential geometry and partial differential equations. He also wrote a book on the psychology of doing mathematics.

2 Index Gymnastics: Examples

Consider the following matrix equation: $\mathbf{A} = \mathbf{BC}$. Given the standard definition of matrix multiplication, we can index the whole equation as follows:

$$[\mathbf{A}]_{ij} = [\mathbf{BC}]_{ij}$$
$$A_{ij} = \sum_p B_{ip} C_{pj}.$$

Note the introduction of the dummy index p. Also, since the elements are simply numbers, they commute. Now on to some examples involving calculus.

Example 1

Question: Let $r = \mathbf{x} \cdot \mathbf{a} \in \mathbb{R}$ for vectors $\mathbf{x}, \mathbf{a} \in \mathbb{R}^n$. What is $\frac{\partial r}{\partial \mathbf{x}}$?

Solution: We start off by indexing the object under investigation with i and expanding.

$$\left[\frac{\partial r}{\partial \mathbf{x}}\right]_i = \frac{\partial r}{\partial x_i} = \frac{\partial \mathbf{x}^\top \mathbf{a}}{\partial x_i} = \frac{\partial}{\partial x_i} \sum_k x_k a_k = \sum_k \frac{\partial x_k}{\partial x_i} a_k = \sum_k \delta_{ki} a_k = a_i = [\mathbf{a}]_i.$$

After writing out the dot product explicitly, we leverage the linearity of the differential operator. Informally put, we can swap the order of the differential operator and the summation symbol. With the Kronecker delta, we note that the only non-zero term in the sum is the one in which k equals i. Without having to predetermine whether gradients should be represented by column or row vectors, we have the unambiguous result: $\frac{\partial r}{\partial x_i} = a_i$.

Let us pick a shape for our gradient. If we decide to let gradients be column vectors, the result has the pretty form $\frac{\partial r}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}^\top \mathbf{a}}{\partial \mathbf{x}} = \mathbf{a}$.

In the example above, we could have picked the gradient to be a row vector, in that case: $\frac{\partial r}{\partial \mathbf{x}} = \mathbf{a}^\top$. The only difference between the column and row vector gradients is a transpose operation. This seems quite harmless, but don't be fooled. These choices become increasingly more important as the objects increase in rank. The main takeaway is to pick a reasonable convention and stick with it. Here, the default will be the column vector representation, unless stated otherwise. (In an assignment or exam you are usually told which one to use. Always read the instructions carefully!)

Example 2

Question: Consider the scalar $s = \mathbf{b}^\top \mathbf{X} \mathbf{c}$, where $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{c} \in \mathbb{R}^n$, and $\mathbf{X} \in \mathbb{R}^{m \times n}$. Find $\frac{\partial s}{\partial \mathbf{X}}$.

Solution: Again, choosing a way to index the object and expanding gives the following.

$$\left[\frac{\partial s}{\partial \mathbf{X}}\right]_{ij} = \frac{\partial s}{\partial X_{ij}} = \frac{\partial \mathbf{b}^\top \mathbf{X} \mathbf{c}}{\partial X_{ij}} = \frac{\partial}{\partial X_{ij}} \sum_{p,q} b_p X_{pq} c_q = \sum_{p,q} b_p \frac{\partial X_{pq}}{\partial X_{ij}} c_q = \sum_{p,q} \delta_{pi} \delta_{qj} b_p c_q$$
$$= \sum_p \delta_{pi} \delta_{jj} b_p c_j = b_i c_j = [\mathbf{b}\mathbf{c}^\top]_{ij}$$

Note that this object requires two indices in order to define a single element. We also required two Kronecker deltas in order to codify the condition for the derivative to equal unity. (The derivative of X_{31} with respect to X_{31} is 1, so the indices need to match in top-bottom fashion.) First we sum over q and we are left with the terms in which q equal j. Then we sum over p and obtain the final result,

$$\frac{\partial s}{\partial \mathbf{X}} = \frac{\partial \mathbf{b}^\top \mathbf{X} \mathbf{c}}{\partial \mathbf{X}} = \mathbf{b}\mathbf{c}^\top,$$

which can be rewritten in terms of an outer product of the two constant vectors. (Write the final step out in terms of the elements of a matrix in order to convince yourself that this is so.)

Depending on how you choose to approach a problem, you might have to make a choice when casting the elements back into matrix notation. Use the dimensions of the matrices to guide you along the way. In fact, you should always keep a mental note of what type of object you are manipulating throughout the steps of an equation. It helps to write out the dimensions of the different tensors to keep track of what the sizes are of the various terms. This will prove to be helpful when coding everything up. Remember: You can always print the result of `numpy.shape` or `torch.size` in order to check that the dimensions of your arrays are what you expect. With this in mind, try the following exercise.

Exercise 1

Question: Given a vector $\mathbf{x} \in \mathbb{R}^n$ and square matrix $\mathbf{B} \in \mathbb{R}^{n \times n}$, evaluate $\frac{\partial \mathbf{x}^\top \mathbf{B}\mathbf{x}}{\partial \mathbf{x}}$.
Answer: $(\mathbf{B} + \mathbf{B}^\top)\mathbf{x}$.

Example 3

Question: Find an expression for $\frac{\partial \mathbf{Q}^\top \mathbf{Q}}{\partial \mathbf{Q}}$, where $\mathbf{Q} \in \mathbb{R}^{p \times q}$.

Solution: This is a derivative of a matrix with respect to another matrix. It might help to rename the product such that $\mathbf{R} \coloneqq \mathbf{Q}^\top \mathbf{Q}$, then the task at hand is simply to evaluate $\frac{\partial \mathbf{R}}{\partial \mathbf{Q}}$. The object as a whole has four indices, i.e. it is a 4-rank tensor. (We now have a $q \times q$ size matrix in the "numerator", and a $p \times q$ size matrix in the "denominator". So there are four free indices and the object has pq^3 entries.) Down to business:

$$\begin{aligned}
\frac{\partial R_{ij}}{\partial Q_{mn}} &= \frac{\partial [\mathbf{Q}^\top \mathbf{Q}]_{ij}}{\partial Q_{mn}} = \frac{\partial}{\partial Q_{mn}} \sum_k Q^\top_{ik} Q_{kj} = \sum_k \frac{\partial}{\partial Q_{mn}} (Q_{ki} Q_{kj}) \\
&= \sum_k \frac{\partial Q_{ki}}{\partial Q_{mn}} Q_{kj} + \sum_k Q_{ki} \frac{\partial Q_{kj}}{\partial Q_{mn}} = \sum_k \delta_{km} \delta_{in} Q_{kj} + \sum_k Q_{ki} \delta_{km} \delta_{jn} \\
&= \delta_{in} Q_{mj} + \delta_{jn} Q_{mi}.
\end{aligned}$$

That's it! Comparing the left hand side with every step in the calculation, you will observe that there is a conservation of free indices. In other words, if someone asks you for entry $\frac{\partial R_{13}}{\partial Q_{21}}$, it can be readily evaluated using the result: $\frac{\partial R_{13}}{\partial Q_{21}} = \delta_{11} Q_{23} + \delta_{31} Q_{21} = Q_{23}$.

You might have noticed that the previous result was not written in closed-form, but was left in index notation. Closed-form is useful in deep learning because it allows us to "vectorize" our algorithms and use our GPUs to their full potential. So when it is sensible, you should opt for a vectorized expression in your algorithm in order to allow for large batch sizes. In other words: less loops, more speed!

Exercise 2

Question: You are given matrices $\mathbf{V}$ and $\mathbf{W}$. Find an expression for $\frac{\partial \operatorname{tr}(\mathbf{VXW})}{\partial \mathbf{X}}$.
Answer: $\mathbf{V}^\top \mathbf{W}^\top$
First we develop the scalar function $\operatorname{tr}(\mathbf{VXW})$ into:

$$\operatorname{tr}(\mathbf{VXW}) = \sum_i [\mathbf{VXW}]_{ii} = \sum_i \sum_j \mathbf{V_{ij}} [\mathbf{XW}]_{\mathbf{ji}} = \sum_i \sum_j \sum_k V_{ij} X_{jk} W_{ki}$$

Then, calculate the partial derivative $\frac{\partial \operatorname{tr}(\mathbf{VXW})}{\partial \mathbf{X}_{nm}}$ (component-wise notation):

$$\frac{\partial \operatorname{tr}(\mathbf{VXW})}{\partial \mathbf{X}_{nm}} = \frac{\partial \sum_i \sum_j \sum_k V_{ij} X_{jk} W_{ki}}{\partial \mathbf{X}_{nm}} = \sum_i \sum_j \sum_k V_{ij} \delta_{nj} \delta_{mk} W_{ki} = \sum_i V_{in} W_{mi}$$

Generalised solution in matrix notation:

$$\frac{\partial \operatorname{tr}(\mathbf{VXW})}{\partial \mathbf{X}} = \mathbf{V}^\top \mathbf{W}^\top$$

Exercise 3

Question: For a vector $\mathbf{w} \in \mathbb{R}^n$ and its Euclidean norm $\|\mathbf{w}\| := \sqrt{\mathbf{w}^\top \mathbf{w}}$, calculate $\frac{\partial \|\mathbf{w}\|}{\partial \mathbf{w}}$.
Answer: $\frac{\mathbf{w}}{\|\mathbf{w}\|}$.
We can right the norm as:

$$f(g(\mathbf{w})) = \|\mathbf{w}\| = \sqrt{g(\mathbf{w})}$$

We can then apply the chain rule:

$$\frac{df}{dg}\frac{dg}{d\mathbf{w}} = f(g(\mathbf{w}))^{-1}\mathbf{w} = \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

Exercise 4

Question: Let $\mathbf{S}$ be a square matrix, find an expression for $\frac{\partial \operatorname{tr}(\mathbf{S})}{\partial \mathbf{S}}$.
Answer: $\mathbf{I}$.

3 MLP Backpropagation

We will look at backpropagation from a modular perspective. In other words, it will be easier to think of a neural network as a series of functions (with or without adjustable parameters θ) rather than as a network with neurons as nodes. In a traditional sketch of a neural network, it is not as easy to see that within each node, an activation function is being applied to the result of the linear transformation. By making each of these operations a separate module, it will become clear how backpropagation works in the general setting. A simple example of such a modular representation is shown in Figure 1. Note that in the forward pass, certain modules require not only features from the previous layer, but also a set of parameters that are constantly being updated during training. *Backpropagation is an algorithm that allows us to update these parameters using gradient descent in order to decrease the loss.*

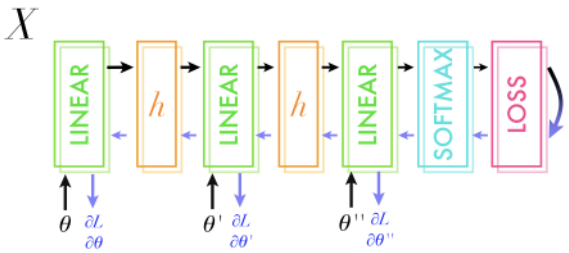


Figure 1. Example of an MLP represented using modules

3.1 Evaluating the Gradients

In the forward pass, some input data is injected into a neural network. The features flow through the neural network blissfully, changing dimensionality along the way. The number of features in a hidden

layer corresponds to the number of neurons in the corresponding layer. In the final layer, some sort of output is generated. In the example of a classification problem, one might consider using softmax in the output layer (as in Figure 1). For training, we will require a *loss function* L, a measure of how poorly the neural network has performed. The lower the loss, the better the performance of our model on that data. Note that our model has parameters θ. In a traditional *linear layer*, the parameters are the weights and biases of a linear transformation. Also, note that a conventional activation function (e.g. ReLU) has no parameters that need to be optimized in this fashion.

In general, we do not want to send in one data point at a time, but rather multiple in a *batch*. Let the number of samples in a batch be represented by S and the number of features (or dimensions) in each sample by M. Concatenating all the samples in a single batch as row-vectors, we obtain the feature matrix $\mathbf{X} \in \mathbb{R}^{S \times M}$.

In a simple linear module, the number of features per data point will usually vary. For example, in a linear transformation from a layer with M neurons to the next layer with N neurons, the number of features goes from M to N. In other words, an input to this linear transformation has M elements, and the output has N, which is just like an ordinary matrix multiplication! For one data point $\mathbf{z} \in \mathbb{R}^M$ being transformed into $\mathbf{v} \in \mathbb{R}^N$ (i.e. batch size of 1) the linear transformation looks like $\mathbf{v} = \mathbf{W}\mathbf{z} + \mathbf{p}$, where $\mathbf{W} \in \mathbb{R}^{N \times M}$ and $\mathbf{p} \in \mathbb{R}^N$. If we transpose this whole equation we get $\mathbf{v}^\top = \mathbf{z}^\top \mathbf{W}^\top + \mathbf{p}^\top$. Note that in programming, the most fundamental array is a list, which is best represented by a row vector. Instead we rewrite the equation with row-vectors $\mathbf{y} = \mathbf{v}^\top, \mathbf{x} = \mathbf{z}^\top, \mathbf{b} = \mathbf{p}^\top$ and we obtain the much nicer looking: $\mathbf{y} = \mathbf{x}\mathbf{W}^\top + \mathbf{b}$. Now we can handle multiple data points at once with input feature matrix $\mathbf{X} \in \mathbb{R}^{S \times M}$, the output features are then given by $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \mathbf{B} \in \mathbb{R}^{S \times N}$. The weight matrix is $\mathbf{W}$ and the bias row-vector $\mathbf{b} \in \mathbb{R}^{1 \times N}$ is tiled S times into $\mathbf{B} \in \mathbb{R}^{S \times N}$. (Note that $B_{ij} = b_j$.)

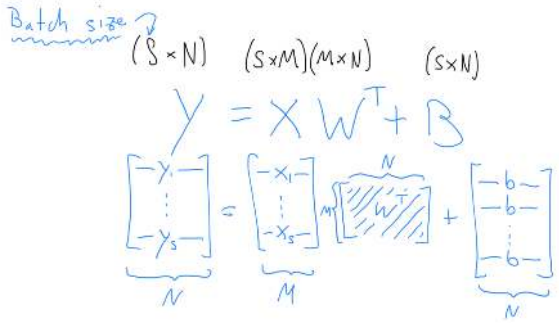
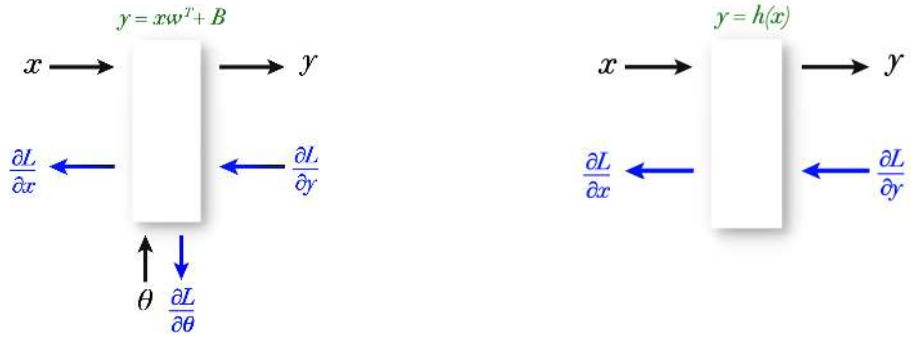


Figure 2. The programming convention used in DL assumes the batch dimension comes first.

For a linear module that receives input features $\mathbf{X}$ and has weight and biases given by $\mathbf{W}$ and $\mathbf{b}$ the forward pass is given by $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \mathbf{B}$. In the backward pass (backpropagation), the gradient of the loss with respect to the output $\mathbf{Y}$ will be supplied to this module by the subsequent module.



(a) A linear module

(b) An activation function module

Figure 3. Forward and backward passes in the basic modules.

Note that performing the chain rule over a matrix requires to sum over all its elements. Let there be a matrix $\mathbf{M}$ with some dependence on a scalar variable t. Then, for some well-defined and continuous function $g : \mathbb{R}^{m\times n} \to \mathbb{R}$, we have:

$$\frac{\partial g(\mathbf{M})}{\partial t} = \sum_{i,j} \frac{\partial g(\mathbf{M})}{\partial M_{ij}} \frac{\partial M_{ij}}{\partial t}.$$

In your further reading, you might encounter the notion of the *Einstein summation convention*[4]. Simply put, this alleviates the need to write the summation sign at the front of an expression. The key to working with this convention is to look for repeated indices, which indicates that the index is a dummy index (and it is therefore being summed over). We will not use it in this course as it will not provide additional clarity in solving the problems. *You are expected to keep summation signs in all expressions in your work.* In NumPy, however, there is a handy implementation of `einsum` which could be useful for removing loops from your calculations.

Question 1 a, b, c) Linear Module (9 points)

Consider a linear module as described above. The input and output features are labeled as $\mathbf{X}$ and $\mathbf{Y}$, respectively. Find closed form expressions for

$$\text{a)} \frac{\partial L}{\partial \mathbf{W}}$$
$$\text{b)} \frac{\partial L}{\partial \mathbf{b}}$$
$$\text{c)} \frac{\partial L}{\partial \mathbf{X}}$$

in terms of the gradients of the loss with respect to the output features $\frac{\partial L}{\partial \mathbf{Y}}$ provided by the next module during backpropagation. Assume the gradients have the same shape as the object with respect to which is being differentiated. E.g. $\frac{\partial L}{\partial \mathbf{W}}$ should have the same shape as $\mathbf{W}$, $\frac{\partial L}{\partial \mathbf{b}}$ should be a row-vector just like $\mathbf{b}$ etc.

Question 1 d) Activation Module (3 points)

Consider an *element-wise* activation function h. The activation module has input and output features labelled by $\mathbf{X}$ and $\mathbf{Y}$, respectively. I.e. $\mathbf{Y} = h(\mathbf{X}) \Rightarrow Y_{ij} = h(X_{ij})$. Find a closed-form expression for

$$\frac{\partial L}{\partial \mathbf{X}}$$

in terms of the gradient of the loss with respect to the output features $\frac{\partial L}{\partial \mathbf{Y}}$ provided by the next module. Assume the gradient has the same shape as $\mathbf{X}$.

The final module before the loss evaluation is responsible for turning the jumbled-up data into predictions for C categories. *Softmax* takes an ordered set of numbers (e.g list or vector) as an input and returns the same sized set with a corresponding "probability" for each element. Therefore, one must have already ensured that this module receives data with a number of features equal to the number of categories C. We would like to generalize this to a batch of many such ordered lists (row vectors). The softmax module is defined for feature matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{S\times C}$ as follows:

[4] **Albert Einstein** (1879-1955) was a German theoretical physicist. In 1921 he was disappointed to hear he was awarded a Nobel Prize "for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect". He would have rather received it for his formulation of general relativity.

$$Y_{ij} = [\text{softmax}(\mathbf{X})]_{ij} := \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}}.$$

Finally, we must specify a loss function for training in order to compare the outputs from our final module (e.g. softmax) to our *targets* $\mathbf{T} \in \mathbb{R}^{S\times C}$, also referred to as *labels*. The rows are the target row-vectors $\mathbf{t} \in \mathbb{R}^{1\times C}$ and are usually one-hot, meaning that all elements are 0 except for the one corresponding to the correct label, which is set to unity. This can be generalized even further such that $\sum_j t_j = \sum_j T_{kj} = 1$, for all samples k. Let us pick the *categorical cross entropy*. The loss of a sample i in the batch is then given by:

$$L_i := -\sum_k T_{ik} \log(Y_{ik})$$

The final loss is the mean over all the samples in the batch. Therefore, $L = \frac{1}{S}\sum_i L_i$.

Question 1 e) Softmax and Loss Modules (3 points)

Let $\mathbf{Z} \in \mathbb{R}^{S\times C}$ be a feature matrix with S samples at the end of a deep neural network. Consider a softmax layer $Y_{ij} = \frac{e^{Z_{ij}}}{\sum_k e^{Z_{ik}}}$ followed by a categorical cross-entropy loss. The final scalar loss L is the arithmetic mean of $L_i = -\sum_k T_{ik} \log(Y_{ik})$ over all samples i in the batch. Targets are collected in $\mathbf{T} \in \mathbb{R}^{S\times C}$ and the elements of each row sum to 1 . It can be shown that the gradients of these modules have the following closed form:

$$\frac{\partial L}{\partial \mathbf{Z}} = \mathbf{Y} \circ \left(\frac{\partial L}{\partial \mathbf{Y}} - \left(\frac{\partial L}{\partial \mathbf{Y}} \circ \mathbf{Y} \right) \mathbf{1}\mathbf{1}^\top \right)$$
$$\frac{\partial L}{\partial \mathbf{Y}} = -\frac{1}{S}\frac{\mathbf{T}}{\mathbf{Y}}.$$

The Hadamard product is defined by $[\mathbf{A} \circ \mathbf{B}]_{ij} = A_{ij}B_{ij}$ and the division of the two matrices is also element-wise. The ones vector is denoted by $\mathbf{1}$ and its size is such that the matrix multiplication in the expression above is well-defined.

All gradients of the loss have the shape of the object with respect to which is being differentiated. One can combine these into a single module with the following gradient:

$$\frac{\partial L}{\partial \mathbf{Z}} = \alpha \mathbf{M}$$

Find expressions for the positive scalar $\alpha \in \mathbb{R}^+$and the matrix $\mathbf{M} \in \mathbb{R}^{S\times C}$ in terms of $\mathbf{Y}, \mathbf{T}$, and S.

3.2 NumPy implementation

After discussing the theory, it is time to get some experience by implementing your own neural network with the equations above. For those who are not familiar with Python and NumPy it is highly recommended to going through the NumPy tutorial.

Question 2 (20 points)

Implement a multi-layer perceptron using purely NumPy routines. The network should consist of a series of linear layers with ELU activation functions followed by a final linear layer and softmax activation. As a loss function, use the common cross-entropy loss for classification tasks. To optimize your network you will use the mini-batch stochastic gradient descent algorithm. Implement all modules in the files `modules.py` and `mlp_numpy.py` by carefully

checking the instructions in the files. You can use the provided `unittests.py` to check your implementation of the modules for bugs.

Part of the success of neural networks is the high efficiency on graphical processing units (GPUs) through matrix multiplications. Therefore, all of your code should make use of matrix multiplications rather than iterating over samples in the batch or weight rows/columns. Implementing multiplications by iteration will result in a penalty.

Implement training and testing scripts for the MLP inside `train_mlp_numpy.py`. Using the default parameters provided in this file, you should get an accuracy of around $47 - 48\%$ using ELU activation function for the entire *test* set for an MLP with one hidden layer of 128 units. Finally, **provide the achieved test accuracy and training loss curve for the training on ans-delft** for the default values of parameters (one layer, 128 hidden units, 10 epochs, learning rate 0.1, seed 42).

4 PyTorch MLP

The main goal of this part is to make you familiar with PyTorch. PyTorch is a deep learning framework for fast, flexible experimentation. It provides two high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autodiff system

You can also reuse your favorite python packages such as NumPy, SciPy and Cython to extend PyTorch when needed. Check out Tutorial 2 for an introduction to PyTorch.

Question 3 (20 points)

Implement the MLP in `mlp_pytorch.py` file by following the instructions inside the file. The interface is similar to `mlp_numpy.py`. Implement training and testing procedures for your model in `train_mlp_pytorch.py` by following the instructions inside the file. Using the same parameters as in the NumPy implementation, you should get similar accuracy on the test set. Again, **provide the achieved test accuracy and training loss curve for the training on ans-delft** for the default values of parameters (one layer, 128 hidden units, 10 epochs, no batch normalization, learning rate 0.1, seed 42).

4.1 Optimization

Consider point x_p where $\nabla_{\mathbf{x}} f(\mathbf{x}_p) = \mathbf{0}$, we call this point a critical or stationary point (the p is to represent the critical point in x). If a critical point is not a local maximum or minimum, it will be classified as a saddle point. To determine if a critical point in a higher dimension is a local minimum or maximum, we can use the Hessian matrix check. Applying the Hessian matrix to a critical point $H(x_p)$ captures how the function curves around the critical point in a higher dimension, similar to how the derivative captures how a quadratic function curves around the critical point in 2 dimensions.

For continuously differentiable function f and real non-singular (invertible) Hessian matrix H at point x_p, if H is positive definite we have a strictly local minimum, and if it is negative definite we have a strictly local maximum.

Question 4 (8 points)

a) Show that the eigenvalues for the Hessian matrix in a strictly local minimum are all positive.

b) If some of the eigenvalues of the Hessian matrix at point p are positive and some are negative, this point would be a saddle point; intuitively explain why the number of saddle points is exponentially larger than the number of local minima for higher dimensions?

Hint: Think of the eigenvalue sign as flipping a coin with probability (1/2) for a head coming up (positive sign).

c) By using the update formula of gradient descent around saddle point p, show why saddle points can be harmful to training.

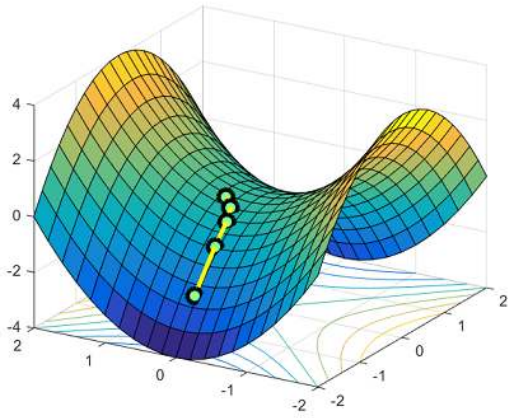


Figure 4. Gradient when approaching saddle point. *source: off the convex path [6]*

4.2 Normalization

During training, as the weights update, the distribution of input to each layer changes. Constantly adapting to new input distributions ('internal covariate shifts') can slow down training. By normalizing inputs to each neuron the network does not have to adapt to changing input distributions. Normalizing also ensures gradients are in a stable range and don't become too small or too large, preventing vanishing and exploding gradients, and ensures gradients don't depend on the scale of parameters. In contexts with these challenges, batch normalization can speed up convergence.

Batch normalization normalizes the previous layer's outputs (i.e., makes the mean across the outputs 0 and variance 1), and scales and shifts those values using learnable parameters γ_i and β_i respectively. These parameters are learned for each output, denoted by the subscript i.

Question 5 (14 points)

a) Adding batch normalization layers causes changes to the backpropagation steps, because we also want to optimize the learnable β_i and γ_i parameters. Assume that we have already backpropagated up to the output of the batch norm node, and therefore we have each $\frac{\partial L}{\partial y_i}$, where $y_i = \gamma_i x_i + \beta_i$. Write the derivatives of L loss with respect to the two parameters β_i and γ_i *in terms of* $\frac{\partial L}{\partial y_i}$.

b) Consider applying batch normalization to a fully connected layer with an input size of 20 and an output size of 40. How many training parameters does this layer have, including batch normalization parameters?

c) During training, batch normalization normalizes inputs using the mean and variance of the current mini-batch. Explain why it would be problematic to normalize inputs the same way during inference (test time), and how batch normalization addresses this problem.

d) Experimental analysis showed that a high percentage of neurons are dead in networks with ReLU activation functions (you can refer to tutorial 3 for more information). Explain the concept of a dead neuron, when it occurs when using ReLU, and how it harms training.

e) How does batch normalization prevent neurons from dying?

f) Previously, you trained your PyTorch MLP using the default values of parameters (one layer, 128 hidden units, 10 epochs, no batch normalization, learning rate 0.1, seed 42). Now, retrain the model with the same parameters, but this time include batch normalization after the hidden layer. Compare the resulting accuracies with those obtained in the original setting, and motivate why these observations make sense.

Submission

Create ZIP archive containing all Python code. Please preserve the directory structure as provided in the Github repository for this assignment. Give the ZIP file the following name: `studentID_assignment1.zip` where you insert your student ID. Please submit your deliverable through Canvas.

References

1. *Matrix Cookbook*: https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf (This contains all the identities you will ever need, and then some!)
2. Useful list of identities on *Wikipedia*: https://en.wikipedia.org/wiki/Matrix_calculus (Note how the results can be cast differently depending on the convention used.)
3. *Mathematics for Machine Learning* textbook: https://mml-book.github.io/book/mml-book.pdf (A nice refresher. Go to section 5.5 for a list of some important identities.)
4. Using `einsum` in NumPy: https://numpy.org/doc/stable/reference/generated/numpy.einsum.html
5. A guide to `numpy.einsum`: https://ajcr.net/Basic-guide-to-einsum/
6. Saddle point figure from `off the convex path`: https://www.offconvex.org/2016/03/22/saddlepoints/