Assignment 1. MLPs and Backpropagation

BGU IEM Introduction to Deep Learning - 2023

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 the final section, 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*. 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 be very careful with the indices.

1 From Indices to Matrices

1.1 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*[1], which has the power to encode if-statements into our mathematical equations:

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

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

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

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:

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

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. They are independent variables which happen to have been collected into the same array.

[1] **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".

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: $[\mathbf{M}]_{ij} = M_{ij}$. Observe the relationship between the identity matrix $\mathbf{I}$ and the Kronecker delta: $[\mathbf{I}]_{ij} = \delta_{ij}$.

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. $\operatorname{tr}(\mathbf{S}) \coloneqq \sum_i S_{ii}$. 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*[2] or element-wise product between two matrices of identical size is given by $\mathbf{A} \circ \mathbf{B}$. The elements of the result are $[\mathbf{A} \circ \mathbf{B}]_{ij} = A_{ij}B_{ij}$. Similarly, define the element-wise division between two matrices as $[\mathbf{A} \oslash \mathbf{B}]_{ij} = A_{ij}/B_{ij}$.

1.2 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:

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

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!)

[2] **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.

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.

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

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, and both indices need to match.) First we summed over q and we were left with the terms in which q equal j. Then we summed over p and obtained 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 we rewrote 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.

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 $\mathbf{R} := \mathbf{Q}^\top \mathbf{Q}$, 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:

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

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!

Question 1

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

b) Given matrices $\mathbf{V} \in \mathbb{R}^{n\times m}, \mathbf{X} \in \mathbb{R}^{m\times p}$ and $\mathbf{W} \in \mathbb{R}^{p\times n}$, find an expression for $\frac{\partial \operatorname{tr}(\mathbf{VXW})}{\partial \mathbf{X}}$.

c) For a vector $\mathbf{w} \in \mathbb{R}^n$ and its Euclidean norm $\|\mathbf{w}\| := \sqrt{\mathbf{w}^\top \mathbf{w}}$, calculate $\frac{\partial \|\mathbf{w}\|}{\partial \mathbf{w}}$.

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

You should express all the answers using vectors and matrices.

1.3 Additional Tools

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:

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

2 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.

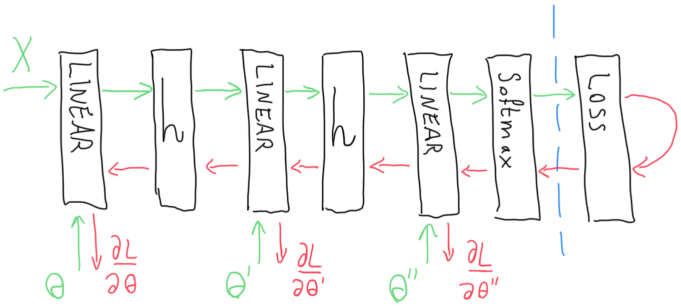


Figure 1. Example of an MLP represented using modules

2.1 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 $B_{ij} = b_j$.)

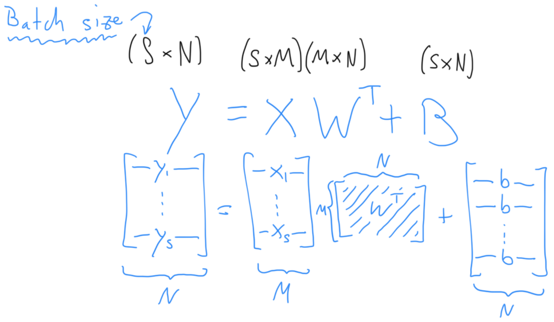
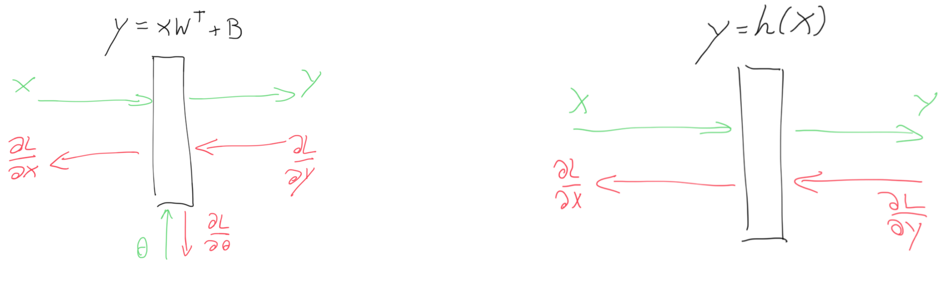


Figure 2. The programming convention used in DL assumes the batch dimension comes first.

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.

Question 2.a Linear Module

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

$$\frac{\partial L}{\partial \mathbf{W}}, \frac{\partial L}{\partial \mathbf{b}}, \frac{\partial L}{\partial \mathbf{X}}$$

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 2.b Activation Module

Consider an *element-wise* activation function h. The activation module has input and output features labeled 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 $\mathbf{X}$. Provide answers both for i) a generic activation function h and ii) the ReLU case $h(x) = \max(0, x)$. *[Hint: You might need to write your answer in terms of a Hadamard product.]*

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:

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

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 2.c Softmax and Loss Modules

i) Consider a softmax module such that $Y_{ij} = [\text{softmax}(\mathbf{X})]_{ij}$, where $\mathbf{X}$ is the input and $\mathbf{Y}$ is the output of the module. Find a closed-form expression for $\frac{\partial L}{\partial \mathbf{X}}$ in terms of $\frac{\partial L}{\partial \mathbf{Y}}$.
[Hint: The answer might require using an all-ones matrix.]

ii) The gradient that kicks the whole backpropagation algorithm off is the one for the loss module itself. The loss module for the categorical cross entropy takes as input $\mathbf{X}$ and returns $L = \frac{1}{S}\sum_i L_i = -\frac{1}{S}\sum_{ik} T_{ik}\log(X_{ik})$. Find a closed form expression for $\frac{\partial L}{\partial \mathbf{X}}$. *[Hint: You may use element-wise operations.]*

Note: You should write the answers to 2.a), 2.b) and 2.c) in terms of matrix operations.

3 NumPy implementation

After discussing the theory, it is time to get some experience by implementing your own neural network with the equations above.

Question 3

Implement a multi-layer perceptron using purely NumPy routines. The network should consist of a series of linear layers with ReLU 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 the code in the files

- `train_mlp_numpy.py`
- `modules.py`
- `mlp_numpy.py`

by carefully checking the instructions in the files.

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 use matrix multiplications rather than iterating over samples in the batch or weight rows/columns. Implementing multiplications by iteration will result in a penalty.

Finally, **provide the achieved test accuracy and loss curve** for the for the default values of parameters (one hidden layer, 128 hidden units, 10 epochs, learning rate 0.1).

4 PyTorch MLP

Question 4.a

Implement the same MLP in `pytorch` by following the instructions inside the file:

- `train_mlp_pytorch.py`
- `mlp_pytorch.py`

The interface is similar to `mlp_numpy.py`. Implement training and testing procedures for your model by following the instructions inside the file.
Using the same parameters as in Question 3, you should get similar accuracy on the test set. Again, **provide the achieved test accuracy and loss curve** for the default values of parameters (one layer, 128 hidden units, 10 epochs, no batch normalization, learning rate 0.1).

4.1 Learning rate

The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate

Question 4.b

i) Why the learning rate is the most important hyperparameter in your deep neural network? What happens(loss, accuracy, training time) if we set the learning rate too high or too low?

ii) What is the appropriate schedule to adjust the learning rate during training? A figure (epoch for the x-axis and learning rate for the y-axis) could be useful to show the schedule. Please make an explanation for your schedule.

iii) To investigate the effect of the learning rate for deep neural networks, train the model with different learning rates using your built MLP network with stochastic gradient descent. Use 9 different learning rates, from 0.000001 to 100 at equal logarithmic intervals. Find the best learning rate for your model and explain why it is better than other options.

iv) Plot two figures:

- Best validation accuracy as function of the learning rates
- Loss curves with different learning rates over time (iteration or epoch).

5 Initialization

When we studied Kaiming initialization, we used a property that you will prove now.

Question 5

Consider a real random variable f with a probability density symmetric around 0, i.e., $p(f) = p(-f)$, and variance σ^2. Prove that if we pass this variable through the ReLU function, $b = ReLU[f] = \max(f, 0)$ then the second moment of the transformed variable is $E[b^2] = \frac{\sigma^2}{2}$.

References

1. *Mathematics for Machine Learning* textbook: `https://mml-book.github.io/book/mml-book.pdf` (See Chapter 5, in particular Section 5.5.)