Instituto Superior Técnico

Deep Learning

1st Semester - 2021/2022

Homework assignment 1

Group:
Tomás Antunes - 102112
Raquel Félix - 90497
Vasco Pearson - 97015

Professor:
André Martins
Francisco Melo
Ben Peters

12th January, 2022

Contents

1 Question 1

1.1 Question 1.1)

The sigmoid activation function is $\sigma(z) = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}, z \in \mathbb{R}$, therefore its derivative is

$$\sigma'(z) = -(1+e^{-z})^{-2}(-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}}$$

Since $1-\sigma(z) = \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} = \frac{e^{-z}}{1+e^{-z}}$ we have the desired result: $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

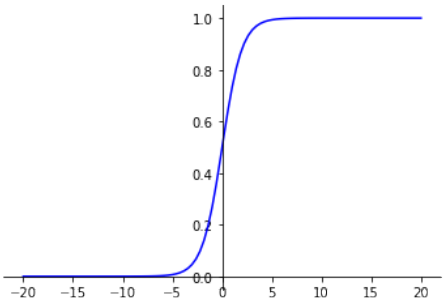


Figure 1: Sigmoid Function

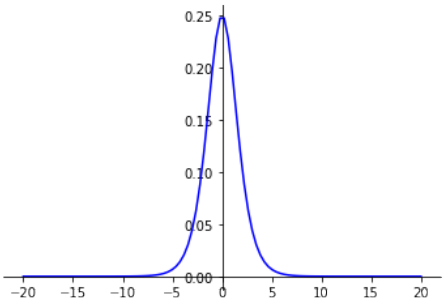


Figure 2: Sigmoid derivative

1.2 Question 1.2)

In this question we are considering a binary classification problem with $y \in \pm 1$. A binary logistic regression model defines $P(y=+1|x;w,b) = \sigma(z)$ with $z = w^T\phi(x) + b$. The binary logistic loss is

$$L(z;y) = \begin{cases} -\log\sigma(z), & \text{if } y=+1 \\ -\log(1-\sigma(z)), & \text{if } y=-1 \end{cases} = -\frac{1+y}{2}\log\sigma(z) - \frac{1-y}{2}\log(1-\sigma(z)),$$

where y is the gold label. Assuming $y=+1$ the first and second derivatives of the loss function with respect to z are computed bellow.

$$L'(z;y=+1) = -\frac{\sigma'(z)}{\sigma(z)}$$

From the previous question we know that $\sigma'(z) = \sigma(z)(1-\sigma(z))$, therefore we have

$$L'(z;y=+1) = -\frac{\sigma(z)(1-\sigma(z))}{\sigma(z)} = \sigma(z) - 1$$

$$L''(z;y=+1) = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

The second derivative of the binary logistic loss function is the same as the derivative of the sigmoid function (see the plot in Figure 2).

A twice-differentiable function of a single variable is convex if and only if its second derivative is nonnegative on its entire domain. So in order to check if the binary logistic loss is convex as a function of z we need to show that $L''(z; y = +1) = \sigma(z)(1 - \sigma(z))$ is nonnegative on its entire domain.

Since $\sigma(z) = \frac{1}{1+e^{-z}}$ is the quociente of two nonnegative functions (the exponencial function is never negative) it is always nonnegative. Furthermore, $1 - \sigma(z) = \frac{e^{-z}}{1+e^{-z}}$ is also the quocient of two nonnegative functions. Evidently, we can conclude that the second derivative of the binary logistic loss is also nonnegative because it is the product of two nonnegative functions.

1.3 Question 1.3)

The softmax transformation is a function from $\mathbb{R}^K$ to $\mathbb{R}^K$ defined as

$$[\text{softmax}(\mathbf{z})]_j = \frac{\exp(z_j)}{\sum_{i=1}^{K} \exp(z_i)}, \mathbf{z} \in \mathbb{R}^K, j = 1, ..., K$$

In this case we are dealing with a multiclass problem, where $y \in \{1, ..., K\}$, with $K > 2$. The Jacobian matrix of the softmax transformation on point $\mathbf{z}$ is the K-by-K matrix whose (j, k)-th entry is $\frac{\partial[\text{softmax}(\mathbf{z})]_j}{\partial z_k}$. So in order to compute this matrix we need to compute $\frac{\partial[\text{softmax}(\mathbf{z})]_j}{\partial z_k}$ for each pair (j, k)

$$J_{softmax} = \begin{bmatrix} \frac{\partial s_1}{\partial z_1} & \frac{\partial s_1}{\partial z_2} & \cdots & \frac{\partial s_1}{\partial z_K} \\ \frac{\partial s_2}{\partial z_1} & \frac{\partial s_2}{\partial z_2} & \cdots & \frac{\partial s_2}{\partial z_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s_K}{\partial z_1} & \frac{\partial s_K}{\partial z_2} & \cdots & \frac{\partial s_K}{\partial z_K} \end{bmatrix}, \text{ where } s_j = [\text{softmax}(\mathbf{z})]_j = \frac{\exp(z_j)}{\sum_{i=1}^{K} \exp(z_i)}$$

Since the exponencial function is strictly positive, the outputs of the softmax fuction will also be strictly positive. Knowing this, we can make the following derivation much shorter by taking the partial derivative of the log of the output

$$\frac{\partial \log s_j}{\partial z_k} = \frac{1}{s_j} \cdot \frac{\partial s_j}{\partial z_k}$$

and arranging it so that on the left we have the partial derivative we want to compute:

$$\frac{\partial s_j}{\partial z_k} = s_j \cdot \frac{\partial \log s_j}{\partial z_k}$$

To compute de derivative we start by simplifying the expression of the logarithm:

$$\log s_j = \log\left(\frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)}\right) = z_j - \log\left(\sum_{i=1}^{K}\exp(z_i)\right)$$

The partial derivative of the expression obtained above is:

$$\frac{\partial}{\partial z_k}\left(z_j - \log\left(\sum_{i=1}^{K}\exp(z_i)\right)\right) = \begin{cases} 1 - \frac{1}{\sum_{i=1}^{K}\exp(z_i)} \cdot \frac{\partial}{\partial z_k}\sum_{i=1}^{K}\exp(z_i), & \text{if } j = k \\ -\frac{1}{\sum_{i=1}^{K}\exp(z_i)} \cdot \frac{\partial}{\partial z_k}\sum_{i=1}^{K}\exp(z_i), & \text{if } j \neq k \end{cases} =$$

$$= \begin{cases} 1 - \frac{\exp(z_k)}{\sum_{i=1}^{K}\exp(z_i)}, & \text{if } j = k \\ -\frac{\exp(z_k)}{\sum_{k=i}^{K}\exp(z_i)}, & \text{if } j \neq k \end{cases} = \begin{cases} 1 - s_k, & \text{if } j = k \\ -s_k, & \text{if } j \neq k \end{cases}$$

Multiplying the result by s_j we get the result we wanted:

$$\frac{\partial s_j}{\partial z_k} = s_j \cdot \frac{\partial \log s_j}{\partial z_k} = \begin{cases} s_j \cdot (1 - s_k), & \text{if } j = k \\ -s_j \cdot s_k, & \text{if } j \neq k \end{cases}$$

Now, since we have obtained a formula for all the values of the Jacobian matrix, we can compute the matrix:

$$J_{softmax} = \begin{bmatrix} s_1 \cdot (1 - s_1) & -s_1 \cdot s_2 & \dots & -s_1 \cdot s_K \\ -s_2 \cdot s_1 & s_2 \cdot (1 - s_2) & \dots & -s_2 \cdot s_K \\ \vdots & \vdots & \ddots & \vdots \\ -s_K \cdot s_1 & -s_K \cdot s_2 & \dots & s_K \cdot (1 - s_K) \end{bmatrix}$$

where $s_j = [\text{softmax}(\mathbf{z})]_j = \frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)}$

1.4 Question 1.4)

The multinomial logistic loss is defined as $L(\mathbf{z}; y = j) = -\log[\text{softmax}(\mathbf{z})]_j$. In this question we will compute the gradient and the Hessian matrix of this loss with respect to $\mathbf{z}$. If the Hessian matrix is positive semi-definite then we can conclude that the multinomial logistic loss is convex.

We start by computing the gradient:

$$\nabla L(\mathbf{z}; y = j) = \begin{bmatrix} \frac{\partial L(\mathbf{z}; y=j)}{\partial z_1} \\ \frac{\partial L(\mathbf{z}; y=j)}{\partial z_2} \\ \frac{\partial L(\mathbf{z}; y=j)}{\partial z_3} \\ \vdots \\ \frac{\partial L(\mathbf{z}; y=j)}{\partial z_K} \end{bmatrix}$$

Knowing that $-\log(\text{softmax}(\mathbf{z})_j) = -\log\left(\frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)}\right) = \log\left(\sum_{i=1}^{K}\exp(z_i)\right) - z_j$, we get:

$$\begin{cases} \frac{\partial L(\mathbf{z};y=j)}{\partial z_k} = \frac{\exp(z_k)}{\sum_{i=1}^{K}\exp(z_i)} - 1, & \text{if } j = k \\ \frac{\partial L(\mathbf{z};y=j)}{\partial z_k} = \frac{\exp(z_k)}{\sum_{k=i}^{K}\exp(z_i)}, & \text{if } j \neq k \end{cases} = \begin{cases} s_k - 1, & \text{if } j = k \\ s_k, & \text{if } j \neq k \end{cases}$$

Where $s_j = [\text{softmax}(\mathbf{z})]_j = \frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)}$. The partial derivative expression above was computed in question 1.3, we only had to multiply it by -1. We can now compute the gradient:

$$\nabla L(\mathbf{z}; y = j) = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_{j-1} \\ s_j - 1 \\ s_{j+1} \\ \vdots \\ s_K \end{bmatrix}$$

It's important to note that the matrix above is a generalization. If $j = 1$ we would have $s_1 - 1$ in the first line of the matrix.

We now want to compute the Hessian matrix of the loss function, in order to show that the multinomial logistic loss is convex. To do so we need to compute the following second order partial derivatives:

$$\frac{\partial^2 L(\mathbf{z}; y = j)}{\partial z_k \partial z_l} = \begin{cases} \frac{\partial(s_k)}{\partial z_l}, & \text{if } j \neq k \\ \frac{\partial(s_k - 1)}{\partial z_l}, & \text{if } j = k \end{cases}$$

Taking into consideration the fact that $s_k = [\text{softmax}(\mathbf{z})]_k = \frac{\exp(z_k)}{\sum_{i=1}^{K}\exp(z_i)}$ and that $\frac{\partial s_k}{\partial z_l} = \begin{cases} s_k \cdot (1 - s_l), & \text{if } k = l \\ -s_k \cdot s_l, & \text{if } k \neq l \end{cases}$ (as we saw in question 1.3), we can easily compute these partial derivatives:

$$\begin{cases} \frac{\partial(s_k)}{\partial z_l}, & \text{if } j \neq k \\ \frac{\partial(s_k - 1)}{\partial z_l}, & \text{if } j = k \end{cases} = \begin{cases} s_k \cdot (1 - s_l), & \text{if } j \neq k \wedge k = l \\ -s_k \cdot s_l, & \text{if } j \neq k \wedge k \neq l \\ s_k \cdot (1 - s_l), & \text{if } j = k \wedge k = l \\ -s_k \cdot s_l & \text{if } j = k \wedge k \neq l \end{cases}$$

Now we can obtain the Hessian Matrix, as shown bellow.

$$\nabla^2 L(\mathbf{z}; y=j) = \begin{bmatrix} \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_1 \partial z_1} & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_1 \partial z_2} & \cdots & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_1 \partial z_K} \\ \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_2 \partial z_1} & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_2 \partial z_2} & \cdots & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_2 \partial z_K} \\ \vdots & & & \\ \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_K \partial z_1} & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_K \partial z_2} & \cdots & \frac{\partial^2 L(\mathbf{z};y=j)}{\partial z_K \partial z_K} \end{bmatrix} =$$

$$= \begin{bmatrix} s_1 \cdot (1 - s_1) & -s_1 \cdot s_2 & ... & -s_1 \cdot s_K \\ -s_2 \cdot s_1 & s_2 \cdot (1 - s_2) & ... & -s_2 \cdot s_K \\ \vdots & \vdots & \ddots & \vdots \\ -s_K \cdot s_1 & -s_K \cdot s_2 & ... & s_K \cdot (1 - s_K) \end{bmatrix}$$

To finish, we only need to show that the Hessian matrix of the loss function is positive semi-definite and conclude that the multinomial logistic loss is convex. A symmetric matrix M with real entries is positive semi-definite if the real number $z^T M z$ is nonnegative for every nonzero real column vector z.

Let $S(x)$ be defined as

$$S(x) = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_K \end{bmatrix} = \begin{bmatrix} \frac{\exp(z_1)}{\sum_{i=1}^K \exp(z_i)} \\ \frac{\exp(z_2)}{\sum_{i=1}^K \exp(z_i)} \\ \vdots \\ \frac{\exp(z_K)}{\sum_{i=1}^K \exp(z_i)} \end{bmatrix}$$

We can define the Hessian matrix of the loss function as $\text{diag}(S(x)) - S(x)S(x)^T$. We need to show that if $z \in \mathbb{R}^n$ then $z^T M z \geq 0$. Notice that

$z^T(\text{diag}(S(x)) - S(x)S(x)^T)z \geq 0$

$\iff z^T \text{diag}(S(x))z \geq z^T S(x)S(x)^T z$

$\iff \sum_{i=1}^K s_i z_i^2 \geq (S(x)^T z)^2$

$\iff \sum_{i=1}^K s_i z_i^2 \geq (\sum_{i=1}^K z_i s_i)^2$

$\iff \sum_{i=1}^K s_i z_i^2 \geq (\sum_{i=1}^K z_i s_i)^2$

$\iff (\sum_{i=1}^K e^{x_i} z_i^2)(\sum_{i=1}^K e^{x_i}) \geq (\sum_{i=1}^K z_i e^{x_i})^2$

This last inequality is true, as can be seen by applying the Cauchy-Schwarz inequality $(a^T a)(b^T b) \geq (a^T b)^2$ to the vectors

$$a = \begin{bmatrix} \sqrt{e^{x_1}} \\ \sqrt{e^{x_2}} \\ \vdots \\ \sqrt{e^{x_K}} \end{bmatrix}, b = \begin{bmatrix} z_1\sqrt{e^{x_1}} \\ z_2\sqrt{e^{x_2}} \\ \vdots \\ z_K\sqrt{e^{x_K}} \end{bmatrix}$$

We have proven that the Hessian matrix of the loss function is positive semi-definite, therefore we can conclude that the multinomial logistic loss is convex.

1.5 Question 1.5)

In this question we want to show that in a linear model where $z = W\phi(x) + b$, the multinomial logistic loss is also convex with respect to the model parameters (W, b), and therefore a local minimum is also a global minimum.

We start by showing that affine functions are convex. Let $f(x) = Ax + b$, where A is an $m \times n$ matrix and b is an $m \times 1$ vector. It follows that:

$$\begin{aligned} f(\lambda x + (1-\lambda)y) &= A(\lambda x + (1-\lambda)y) + b && (1.1) \\ &= \lambda Ax + (1-\lambda)Ay + \lambda b + (1-\lambda)b && (1.2) \\ &= \lambda f(x) + (1-\lambda)f(x) && (1.3) \end{aligned}$$

Let $g : E^m \to E^1$ be a convex function (in our case g is the multinomial logistic loss defined as $L(\mathbf{z}; y = j) = -\log[\text{softmax}(\mathbf{z})]_j$) and let $h : E^n \to E^m$ be an affine function of the form $h(x) = Ax + b$, where A is an $m \times n$ matrix and b is an $m \times 1$ vector. We want to show that the composite function $f : E^n \to E^1$ defined as $f(x) = g(h(x))$ is a convex function.

Let $0 < \theta < 1$ and $x_1, x_2 \in E^n$. Note that since h is an affine function we have that $h(\theta x_1 + (1-\theta)x_2)) = \theta h(x_1) + (1-\theta)h(x_2)$. It follows that

$$\begin{aligned} f(\theta x_1 + (1-\theta)x_2) &= g(h(\theta x_1 + (1-\theta)x_2)) && (1.4) \\ &= g(\theta h(x_1) + (1-\theta)h(x_2)) && (1.5) \\ &\leq \theta g(h(x_1)) + (1-\theta)g(h(x_2)) && (1.6) \\ &= \theta f(x_1) + (1-\theta)f(x_2) && (1.7) \end{aligned}$$

so f is convex.

We have proven that the composition of an affine map $h : \mathbb{R}^n \to \mathbb{R}^m$ with a convex

function $g : \mathbb{R}^m \to \mathbb{R}$ is convex. Therefore, in a linear model where $z = W\phi(x) + b$, the multinomial logistic loss is also convex with respect to the model parameters (W, b), and therefore a local minimum is a also global minimum (because a local minimizer of a convex function is also a global minimizer).

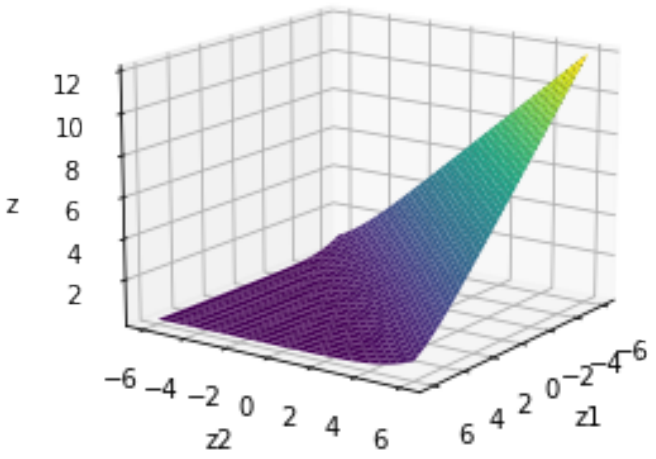


Figure 3: Multinomial logistic loss function when $z \in R^2$

If $\mathbf{z}$ is not a linear function of the model parameters this is not always true. For example, take $z_i = sin(x_i)$. In this case $L(\mathbf{z}; y = j) = -\log[\text{softmax}(\mathbf{z})]_j$ is not convex, as seen in the plot below, where $x \in \mathbb{R}^2$.

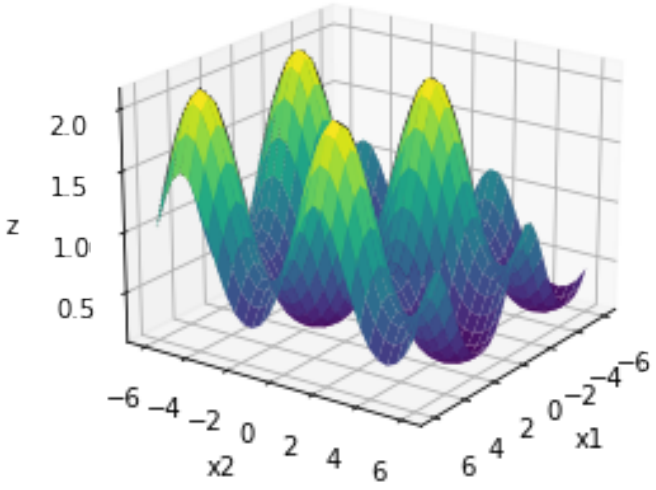


Figure 4:

2 Question 2

2.1 Question 2.1)

We want to show that the squared error loss

$$L(z, y) = \frac{1}{2}\|z - y\|_2^2 = \frac{1}{2}(z - y)^T(z - y)$$

where $z = W^T\phi(x) + b$, is convex with respect to (W, b).

We saw in the previous question that the composition of an affine map $h : \mathbb{R}^n \to \mathbb{R}^m$ with a convex function $g : \mathbb{R}^m \to \mathbb{R}$ is convex. Since z is obtained from an affine map, we only need to prove that $L(z, y) = \frac{1}{2}(z-y)^T(z-y)$ is convex with respect to z.

We can write the loss as $L(z, y) = \frac{1}{2}\sum_{i=0}^{m}(z_i - y_i)^2$. The second order partial derivatives $\frac{\partial^2 L}{\partial z_i z_j}$ are zero if $i \neq j$ and equal 2 if $i = j$. So we can conclude that the hessian matriz will be a diagonal matrix with 2 on the diagonal, so it is positive definite (since all its eigenvalues are positive). Therefore $L(z, y) = \frac{1}{2}(z - y)^T(z - y)$ is convex with respect to z and consequently it is also convex with respect to the model parameters.

2.2 Question 2.2)

In this question we will use the Ames housing dataset to predict the price of the different properties using the features already in the provided data. In section *a*) we will implement a linear regression model and in section *b*) we will implement a neural network model for the same regression problem.

2.2.1 a)

We started by implementing the `update_weights` method of the `LinearRegression` class in `hw1-q2.py`. The first step was to get the predicted value $\hat{y}$ from the model using the predict method. After this, in order to update the weights, we computed the partial derivatives that will be needed later.

$$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2, \text{ where } \hat{y} = w\phi(x)$$

$$\frac{\partial L(\hat{y}, y)}{\partial \hat{y}} = \hat{y} - y$$

Above we have the loss function and its derivative with respect to $\hat{y}_i$. In our task we are doing stochastic gradient descent, meaning that we are only using one training observation to update the weights. Now we only need the derivatives of $\hat{y}$ with respect to the model parameters.

$$\hat{y} = \sum_{i=0}^{n} (x_i w_i + b_i), \text{where } n \text{ is the number of features}$$

$$\frac{\partial \hat{y}}{\partial b_i} = 1$$

$$\frac{\partial \hat{y}}{\partial w_i} = x_i$$

It's important to note that in our problem the bias vector b is included in the weight matrix w and the input matrix X has an extra column of one's to multiply by the bias. Therefore we update the last row of the weight matrix when we want to update the bias.

After implementing the `update_weights` method, we trained the model for 150 epochs using stochastic gradient descent with a learning rate of 0.001. The results are shown bellow.

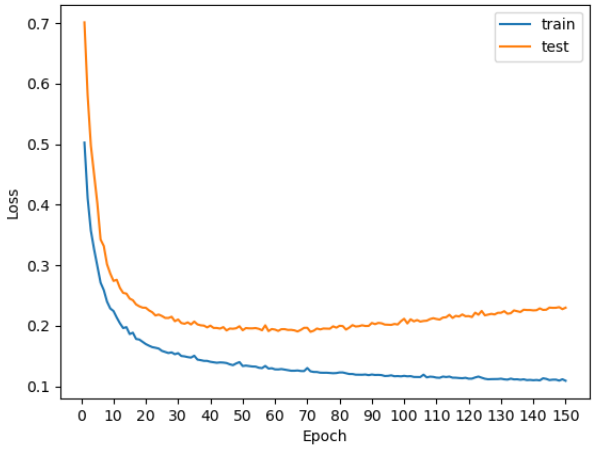


Figure 5: Training the linear regressor

We can see above that the model is starting to overfit after around 80 epochs, because the value of the loss function on the training set keeps decreasing while the loss on the test set starts to increase. The ideal time to stop the training would be when the loss is no longer decreasing on the test set, so in our case, around 60 epochs would be enough to train the model well without overfitting. If we trained the model for less than 40 epochs the model would underfit the data, since a better value for the loss function could still be obtained on the test set and the training set, if we trained the model for more epochs.

In order to compare our weight vector and the weight vector computed analytically we plotted the distance between these values as a function of the epoch number.

This can be seen in Figure 6, bellow. To be able to do this we had to compute the analytic solution by implementing the `solve_analytically` function. We saw in the theoretical classes that the analytic solution was given by the following formula.

$$w = (X^T X)^{-1} X^T y$$

However, it so happens that in our case $X^T X$ is a singular matrix, and therefore it cannot be inverted to obtain the analytic solution. We solved this problem by adding a small value λ (we used $\lambda = 1 \times 10^{-4}$) to the diagonal of $X^T X$ before inverting as shown in the formula bellow. This is equivalent to ridge regression.

$$w = (X^T X + \lambda I)^{-1} X^T y$$

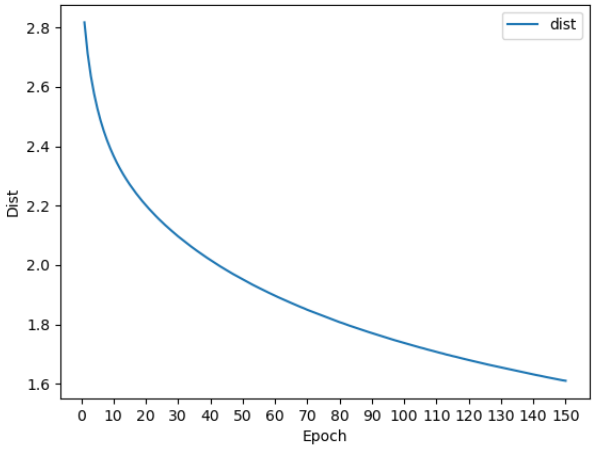


Figure 6: Distance between our weight vector and the weight vector computed analytically

We can see in the plot above that the distance decreases when the number of epochs increases, as was expected. When training the model for an even higher number of epochs the distance keeps decreasing but never seems to reach zero. This happens because in our problem $X^T X$ is not invertible, therefore there will be an infinite number of solutions w that achieve the same minimal square error on the training data. The stochastic gradient descent algorithm will get close to one of those solutions, but probably not the one we are considering as the analytical solution.

2.2.2 b)

Here we implemented a feed-forward network with a single hidden layer (with 150 hidden units and a ReLU activation function) to solve the regression problem above, including the gradient back-propagation algorithm which is needed to train the

model. To do this we implemented the `__innit__()`, `update_weight` and `predict` methods of the `NeuralRegression` class.

In the `__innit__()` class we defined the weights and biases of the neural regression model. The first weight matrix is a $hidden \times n\_features$ matrix, where $hidden$ corresponds to the number of hidden units in the next layer (150) and $n\_features$ corresponds to the number of features of our data. The second weight matrix will also have $n\_features$ columns, but will only have 1 row because there will only be one unit in the output layer. The biases will be matrices with one column. The first bias will have $hidden$ (150) number of rows, corresponding to the 150 hidden units in the second layer, and the second bias will only have one row, so it will only have one entry since the output layer only has one unit. The biases where initialized with zero vectors, while the values in the weight matrices where initialized with $w_{ij} \sim \mathcal{N}(\mu, \sigma^2)$, where $\mu = 0.1$ and $\sigma^2 = 0.1^2$.

In the `predict` method we run a forward pass for each data observation in X and append the corresponding prediction to a vector $\hat{y}$ that we shall return. A forward pass for one observation x is done in the following way:

$$z_1 = w_1 x + b_1$$

$$h_1 = ReLU(z_1)$$

$$\hat{y} = w_2 h_1 + b_2$$

Finally, to implement the `update_weight` method we began by running a forward pass for a single training example (because we are applying stochastic gradient descent) just as we showed above. This is followed by the backpropagation algorithm, where it was necessary to compute several gradients that are shown bellow.

Gradient of the output layer:

$$\frac{\partial L(\hat{y}, y)}{\partial z_L} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} = \frac{\partial \frac{1}{2}(\hat{y} - y)^2}{\partial \hat{y}} = \hat{y} - y$$

Gradient of hidden layer parameters:

$$\frac{\partial L(\hat{y}, y)}{\partial W_l} = \frac{\partial L(\hat{y}, y)}{\partial z_l} h_{l-1}^T$$

$$\frac{\partial L(\hat{y}, y)}{\partial b_l} = \frac{\partial L(\hat{y}, y)}{\partial z_l}$$

Gradients of hidden layer below:

$$\frac{\partial L(\hat{y}, y)}{\partial h_l} = W_{l+1}^T \frac{\partial L(\hat{y}, y)}{\partial z_{l+1}}$$

Gradients of hidden layer below (before activation $h_l = g(z_l)$):

$$\frac{\partial L(\hat{y}, y)}{\partial z_l} = \frac{\partial L(\hat{y}, y)}{\partial h_l} \cdot g'(z_l) = \frac{\partial L(\hat{y}, y)}{\partial h_l} \cdot \begin{cases} 0, & \text{if } z_l \leq 0 \\ 1, & \text{if } z_l > 0 \end{cases}$$

After implementing the backpropagation algorithm we update the weights and biases in the following way (η is the learning rate):

$$W_l = W_l - \eta \frac{\partial L(\hat{y}, y)}{\partial W_l}$$

$$b_l = b_l - \eta \frac{\partial L(\hat{y}, y)}{\partial b_l}$$

This finishes our implementation. We preceeded to train the model for 150 epochs using stochastic gradient descent with a learning rate of 0.001. The results for the training set and the test set are shown bellow.

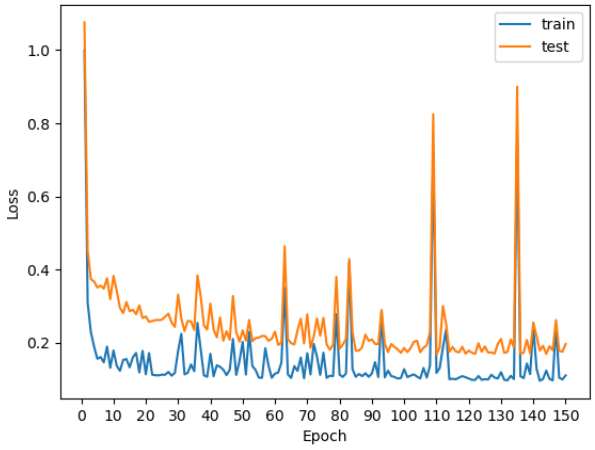


Figure 7: Training the neural regressor

In the figure above we can see that in the training set, the value of the loss function ends up at around 0.1, just as it did with the linear regressor. The major difference this time is that the loss of the test set keeps decreasing along with the loss of the training set, meaning that this time no overfitting occurs and our model is able to generalize. Another noticeable difference is the variation in the values of the loss function. This most likely happens due to the learning rate being to big. If the learning rate was slightly smaller the convergence would take more time but there would be less variation in the values of the loss function.

3 Question 3

In this question, we will implement several classifiers for a simple image classification problem, using the Fashion-MNIST dataset.

3.1 Question 3.1

We will start by implementing a perceptron and a logistic regression model and compare there performance in this classification task.

3.1.1 a)

In this exercise we were asked to implement the `update_weights` method of the `Perceptron` class, in order to solve an image classification problem. A Perceptron is a linear classifier composed by the input values, weights and bias, a net sum, and an activation function, allowing the computation of an output.

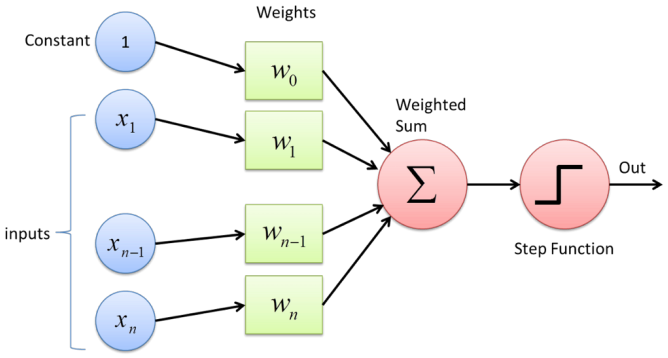


Figure 8: Structure of Perceptron, [2]

A Perceptron can do binary and multi-class classification, the latter is usualy solved by reduction to binary classification or by tackling the multiple classes directly. The perceptron architecture above is one we would use for a binary classification problem. Our problem is a multi-class classification problem, therefore we have a 2D weight matrix (with *n_classes* rows and *n_inputs* columns) and an output node for each class.

To update the weights and train the model, we check if the predicted label equals the gold label. If it does, the weights are not updated, but if it does not equal the gold label, the weights are updated as follows:

$$w_{y_i}^{k+1} = w_{y_i}^{k} + \phi(x_i)$$
$$w_{\hat{y}_i}^{k+1} = w_{\hat{y}_i}^{k} - \phi(x_i)$$

Where $\phi(x)$ is the feature vector, w is the weight matrix and k is the number of

mistakes. This means we increase the weight of the gold class and decrease the weight of the incorrect class.

After computing the `update_weights` method, the accuracy plot was performed for the validation and test set over 20 epochs. The results are shown below.

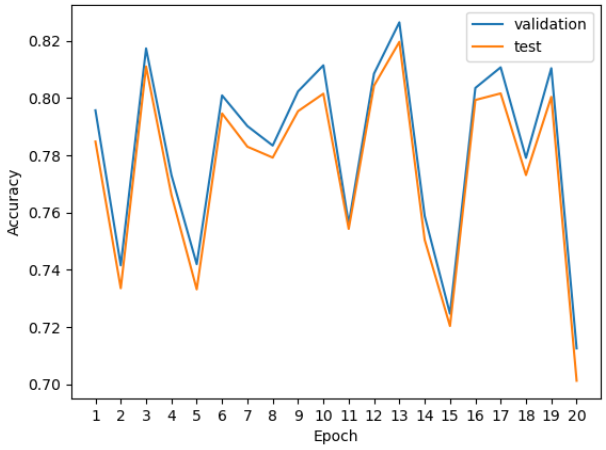


Figure 9: Accuracy plot for validation and test set as function of the number of epochs for the Perceptron

We can see that the accuracy is oscillating over epochs, which is not ideal, however it is still much better than random chance, since the accuracy normally lies between 0.72 and 0.82.

3.1.2 b)

This exercise is identical to the previous one, using Logistic Regression instead. For training, stochastic gradient descent and a learning weight of 0.001 were considered.

To implement the `update_weight` method of the `LogisticRegression` class we had to use one hot encoding on variable y_i in order to compare it with our label probabilities. These probabilities were obtained by applying a softmax tranformation to the label scores ($label\_scores = Wx_i$, where W is the weight matrix and x_i is a single training example). The weight matrix is then updated in the following way:

$$W = W + learning\_rate \times (y\_one\_hot - label\_probabilities)x_i$$

After implementing the necessary code, we trained our model for 20 epochs. Below it is possible to see the accuracy plot for the validation and test set over these 20 epochs. We can see that there is constant improvement in the accuracy as the number of epochs increases and we can conclude that this model is better for this classification task than the perceptron algorithm implemented above, since the accuracy on the test set ends up at about 0.84.

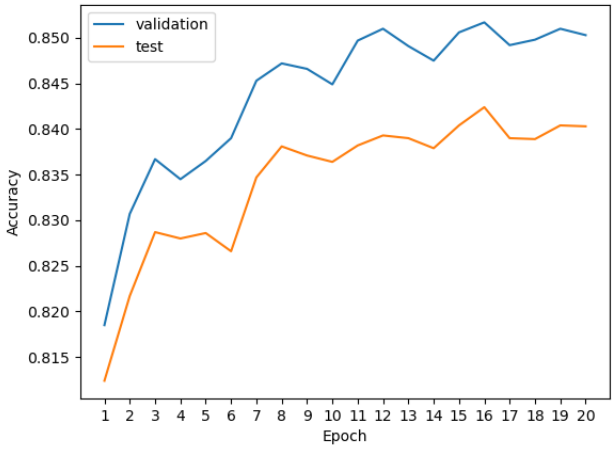


Figure 10: Accuracy plot for validation and test set as function of the number of epochs for Logistic Regression

3.2 Question 3.2

Now, we will implement a multi-layer perceptron (a feed-forward neural network) to solve the same classification problem, and compare its results with the previous model.

3.2.1 a)

As previously mentioned, the Perceptron is a linear classifier, meaning it finds a hyperplane defined by (W, b), W is a weight matrix and b is the bias, that splits the feature space into two spaces. The decision boundary is then defined by the intersection of half spaces, since we are dealing with a multi-class classification problem. For the Perceptron to perform properly it is then necessary for the dataset to be linearly separable. To put this in other words, it has to be possible to split the feature space into regions delimited by hyperplanes. However, sometimes this is not the case and the Perceptron fails to converge. In those cases, a Multi-Layer Perceptron (MLP) is a better approach, as it adds intermediate layers between the input and output and so, each of the units of each layer computes representations of the input, increasing the expressive power of the networks and allowing more complex and non-linear problems to be solved. For this exercise a single hidden layer is used, and the output is as follows:

$$f(x) = o(W_2 h + b_2) = o(W_2(g(W_1 x + b_1)) + b_2)$$

where h is the vector of hidden units, o the softmax output activation function, W matrix of weights, b vector of biases and g the hidden layer ReLU activation function, a non-linear function. This proves that the output is non-linear dependent

of W and b. However, if the activation function was linear instead, this would be equivalent to a single linear layer and no expressive power increase by using MLP.

3.2.2 b)

In this exercise we were asked to implement a Multi-layer Perceptron with a single hidden layer with the gradient backpropagation algorithm to train the model. We will use 200 hidden units, a ReLU activation function for the hidden layers, and a multinomial logistic loss. The backpropagation algorithm was already discussed in Question 2, the only difference is the loss function, which is now the multinomial logistic loss (also called cross-entropy). Bellow we show the expression for the cross-entropy loss and compute its gradient.

$$L(y, p) = -\sum_{k=1}^{d} y_k \log p_k$$

$$\frac{\partial L(y,p)}{\partial z^{[L]}} = -\frac{\partial \sum_{k=1}^{d} y_k \log p_k}{\partial z^{[L]}} = p - y$$

After implementing the `__init__`, `predict` and `train_epoch` methods, we trained our model for 20 epochs. Below it is possible to see the accuracy plot for the validation and test set over these 20 epochs.

Figure 11: Accuracy plot for validation and test set as function of the number of epochs for Multi-Layer Perceptron

Similar to what happened before with the logistic regression model, the accuracy is constantly improving with the number of epochs. However, in this model we end up with an even higher accuracy of around 0.86/0.87 in the test set and more than 0.87 in the validation set, meaning that this is the best model for this classification problem, out of the ones we tried.

4 Question 4

In this question we will implement a model for the same classification problem as before but this time using a deep learning framework (Pytorch) with automatic diferentiation. This means that we no longer need to write backpropagation by hand.

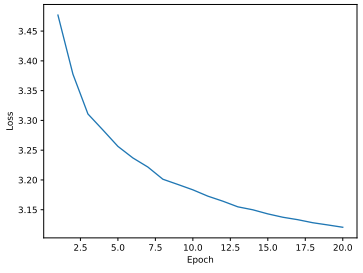
4.1 Question 4.1

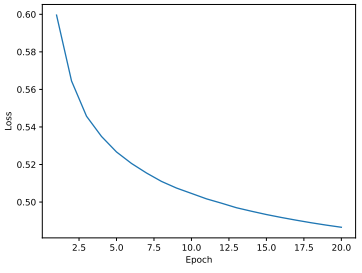
Here we will implement a linear model with logistic regression, using stochastic gradient descent as our training algorithm. For this we implemented the `train_batch()` method and the `__init__()` and `forward()` methods from the `LogisticRegression` class.

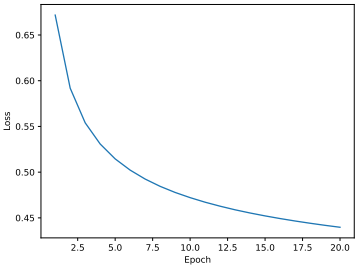
In the `__init__()` method we defined the weights and biases using the *nn.Linear* class from pytorch. The weight matrix will be a $n\_classes \times n\_features$ matrix and the bias is set to true by default. In the `forward()` method we only need to compute the forward pass of the model. We do this by applying our weights and biases to the batch of training examples given as argument.

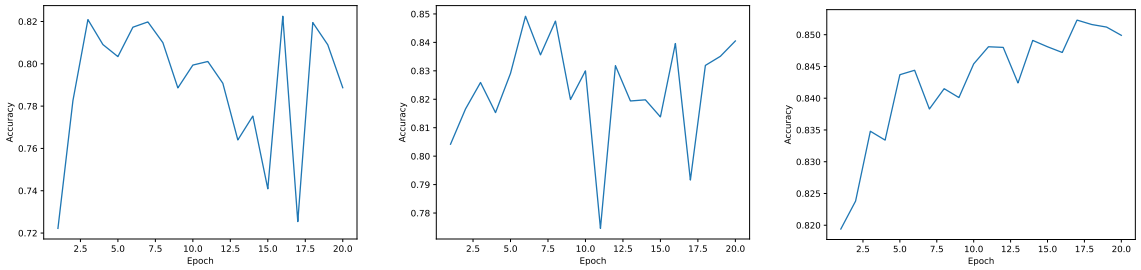
To implement the `train_batch()` method we start by setting the gradients to zero. After this we compute a forward pass and compute the value of the loss. The gradient of the given tensor with respect to the weights/bias is computed using the *loss.backward*() method and the weights and biases are updated using the *optimizer.step*() method.

After implementing the necessary code, we trained our model with different learning rates. The results are bellow, where we use learning rates of 0.1, 0.01 and 0.001, from left to right. The first three plots correspond to the training loss and the last three correspond to the accuracy on the validation set.









We can see that in all three cases the loss decreases with the number of epochs, however with a lower learning rate (0.001) the loss is lower than in the other two cases (note also that the loss is much higher for a learning rate of 0.1). The accuracy plots on the other hand are very distinct and are more helpful in choosing the best learning rate. While the first two plots jump up and down with no improvement, the last plot, corresponding to a learning rate of 0.001, shows overall improvement of the acuracy values on the validation set as the number of epochs gets higher. The final accuracy on the test set for this case is 0.8388.

4.2 Question 4.2

Now we will implement a feed-forward neural network with a single layer, using dropout regularization. To do this we only had to implement the `__init__()` and `forward()` methods of the `FeedforwardNetwork` class, since the `train_batch` method was implemented in the previous question.

In the `__init__()` method we started by defining the weights and biases using the *nn.Linear* class from pytorch. Since we have one hidden layer we will have two instances of the *nn.Linear* class. The first weight matrix will be a $hidden\_size \times n\_features$ matrix and the second weight matrix will be a $n\_classes \times hidden\_size$. If we have 2 hidden layers there will be an extra weight matrix of $hidden\_size \times hidden\_size$ dimensions, and if we have 3 hidden layers there will be two of these weight matrices. This will be needed for Question 4.3. Again, the bias is set to true by default. After this, we define a dropout layer using *nn.Dropout* that can have a probability of 0.3 or 0.5 of setting an element from the input tensor to zero. Finally we define the activation function as *ReLu* or *Tanh* depending on the value of `activation_type`.

After this we did some hyperparameter tuning in order to achieve the best accuracy on the test set. The following table shows the values we experimented with:

Number of Epochs	20
Learning Rate	{0.1, 0.01, 0.001}
Hidden Size	{100, 200}
Dropout	{0.3, 0.5}
Batch Size	1
Activation	{ReLU, Tanh}
Optimizer	{SGD, Adam}

After trying various combinations, the model with the highest values of accuracy on the validation set (above 0.88 and almost reaching 0.89 on some epochs) was the one with the parameters described bellow. This model had a final accuracy on the test set of 0.8795.

Number of Epochs	20
Learning Rate	0.001
Hidden Size	200
Dropout	0.3
Batch Size	1
Activation	ReLU
Optimizer	SGD

The following plots show the value of the loss function on the training data and the accuracy on the validation data as a function os the number of epochs, during training.

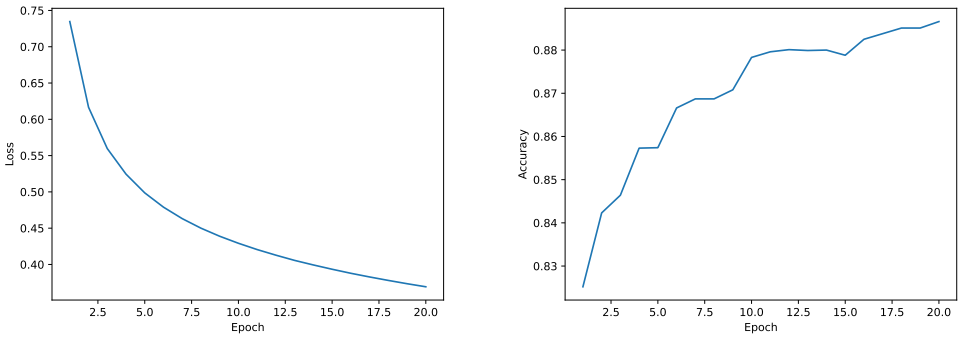


Figure 12: Training loss (left) and validation accuracy (right) as a function of the number of epochs

4.3 Question 4.3

Using the hyperparameters described in the table bellow, we now increase the previous model to have 2 and 3 layers.

Number of Epochs	20
Learning Rate	0.01
Hidden Size	200
Dropout	0.3
Batch Size	1
Activation	ReLU
Optimizer	SGD

The model with highest values of accuracy on the validation set was the one with two layers. These values where above 0.87 for several epochs at the end of training, as shown in figure 13, with the highest value (0.8775) being achieved in the final epoch. The final test accuracy for the model with 2 layers was 0.8753. The acuracy values on the validation set for the model with 3 layers were slightly lower, and so was the final test accuracy, which was 0.8608.

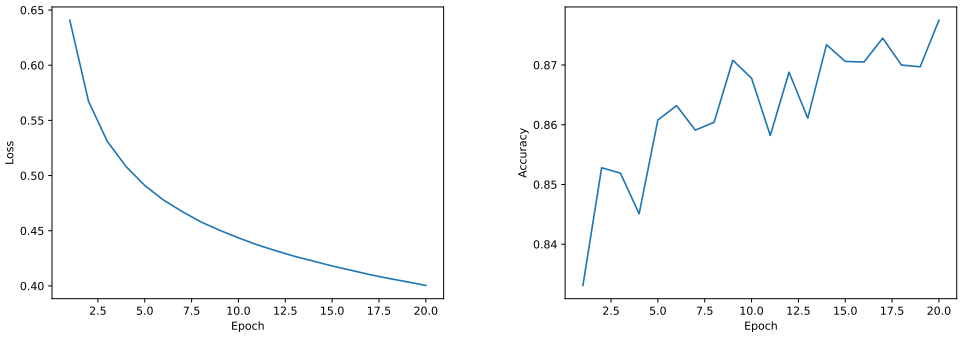


Figure 13: Training loss (left) and validation accuracy (right) as a function of the number of epochs, for the model with 2 hidden layers

5 Bibliography

- Convex Optimization, by Stephen Boyd and Lieven Vandenberghe, [1]
- Perceptron? : https://datascience.eu/pt/aprendizado-de-maquina/perceptron/ (accessed in 11/01/2022), [2]
- MARTINS, ANDRÉ; MELO, FRANCISCO; FIGUEIREDO, MÁRIO. (2021). Deep Learning Course: All lectures [PowerPoint presentation], [3]
- Artificial Intelligence Engines: A Tutorial Introduction to the Mathematics of Deep Learning: James Stone 2019 Sebtel Press, [4]
- Deep Learning with Python: François Chollet 2017 Manning Publications, [5]