TÉCNICO LISBOA

Deep Learning

MEIC-A

Homework 1 [EN]

Authors:

Catarina Bento (93230) catarina.c.bento@tecnico.ulisboa.pt
David Belchior (95550) davidbelchior@tecnico.ulisboa.pt

Group 32

2022/2023 – 1st Semester, Q2

Contents

Question 1

1.1.

a)

In a multi-class perceptron, the prediction stage happens as follows:

$$\hat{y}_i = argmax(W^T x_i) \tag{1}$$

With x_i being a given training sample and y_i its correspondent true class.
If $\hat{y}_i \neq y_i$, W must be updated according to the following rule:

$$\begin{aligned} W_{y_i} &\leftarrow W_{y_i} + \eta x_i \\ W_{\hat{y}_i} &\leftarrow W_{\hat{y}_i} - \eta x_i \end{aligned} \tag{2}$$

In this context, W_{y_i} and $W_{\hat{y}_i}$ represent the ${y_i}^{th}$ and ${\hat{y}_i}^{th}$ columns of W, respectively, and η represents the update rule's learning rate, which defines how quickly each of the matrix's columns update for each training input/label pair.

An epoch is only finished once every pair is used to train the perceptron.

In our project, the update function is then computed as follows:

Listing 1: Update stage of the weight matrix for a simple perceptron.

```
def update_weight(self, x_i, y_i, **kwargs):
        """
        x_i (n_features): a single training example
        y_i (scalar): the gold label for that example
        other arguments are ignored
        """
        eta = kwargs.get("learning_rate", 1)
        y_i_hat = np.argmax(self.W.dot(x_i))
        if y_i_hat != y_i:
            self.W[y_i, :] += eta * x_i
            self.W[y_i_hat, :] -= eta * x_i
```

As for the accuracy on both the training and testing sets, the results are as follows:

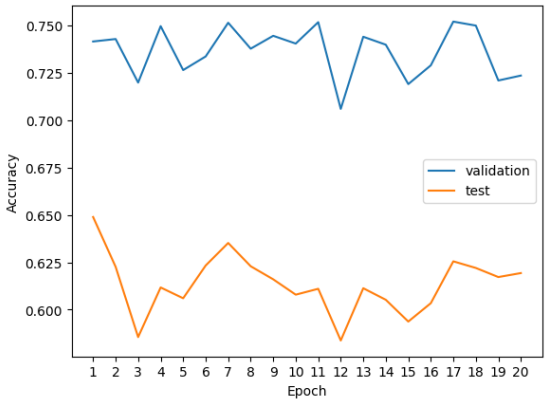

Figure 1: Accuracy on the training and testing sets with the multi-class perceptron.

b)

Logistic regression employs a similar technique to the former exercise, this time using the $softmax$ function to smooth the previously calculated probability for each label, instead of outputting a single output label corresponding to the one with highest probability. Since the outcome will be a probability, the feature's outputs are between 0 and 1.

The $softmax$ function for an entry x_i of a vector $x \in \mathbb{R}^n$ is defined as follows:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} \tag{3}$$

Our update function is then defined as follows:

$$W_{y_i} \leftarrow W_{y_i} + \eta(y'_i - \hat{y}_i) \cdot x_i^T \tag{4}$$

y'_i is a one-hot vector, where $y'_{ij} = 1$ iff $argmax(y_i) = j$, and 0 otherwise.
The following code snippet shows how it can be computed:

Listing 2: Update stage of the weight matrix for a logistic regression problem.

```
def update_weight(self, x_i, y_i, learning_rate=0.001):
    """
    x_i (n_features): a single training example
    y_i: the gold label for that example
    learning_rate (float): keep it at the default value for your plots
    """
    y_hat_i = softmax(self.W.dot(x_i))
    y_i_one_hot = np.zeros(y_hat_i.shape)
    y_i_one_hot[y_i] = 1
    self.W += learning_rate * (y_i_one_hot - y_hat_i)[:, None].dot(x_i[:,
        None].T)
```

As for the accuracy on both the training and testing sets, the results are as follows:

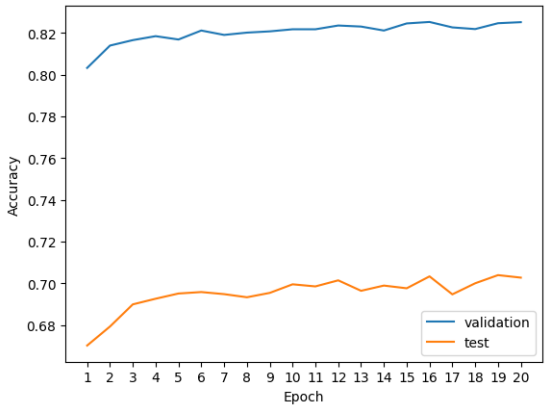

Figure 2: Accuracy on the training and testing sets using logistic regression.

1.2.

a)

The perceptron is only able to solve linearly separable problems, that is, ones that can be separated by a line or hyperplane (example on Figure 3). It applies the idea of placing a straight line boundary in the feature space and, when given a new point, the model checks the side of the boundary where it falls. However, perceptrons cannot model nonlinear functions (such as the logical XOR operation).

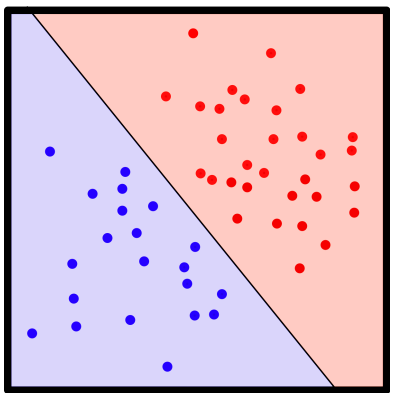

Figure 3: Linearly Separable Data

That limitation of the perceptron does not apply to feed-forward networks with intermediate or "hidden" non-linear units, also called multi-layer perceptrons (MLP). An MLP is a neural network where the mapping between inputs and output is non-linear. As such, MLP is more expressive in a sense that it can represent non-linear functions, as long as its units use non-linear activation functions. However, if the activation function of the multi-layer perceptron is linear (multiple layers of cascade linear units), it will only be able to represent linear functions.

In conclusion, the MLP only overcomes the perceptron's limitation of only representing linear functions, if its units use non-linear activation functions. [1]

b)

Before we start, it's worth mentioning that, whenever a 1-D operation is applied on a vector $x \in \mathbb{R}^n$, it's equivalent to calling the operation for each entry of that vector, i.e.:

$$f(x) = [f(x_i)], i = 1, \cdots, n \tag{5}$$

The first step in an epoch of an MLP training process is to do the forward propagation. In this step, we will propagate data from the first to the last layer in a similar fashion to the perceptron. However, in this case, we will have one hidden layer, in which each node is a neuron that uses an activation function, which adds non-linearity to the network. In the output layer, we will simply apply $softmax$.

First, we need to define the activation function used for the hidden layer, the $ReLU$:

$$\begin{aligned} ReLU(x) &= max(0, x) \\ \frac{d}{dx} ReLU(x) &= \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases} \end{aligned} \tag{6}$$

Note that $softmax$ has already been defined in (3).

Then, the forward-propagation begins. For the first layer, we will calculate the preactivation ($z^{[1]}$), which is the weighted sum of the inputs, plus the bias vector.

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]} = W^{[1]} x + b^{[1]} \tag{7.1}$$

Then, $z^{[1]}$ is passed to the activation function ($ReLU$), and we get $a^{[1]}$, which represents the activated nodes from the hidden layer.

$$a^{[1]} = ReLU(z^{[1]}) \tag{7.2}$$

For the output layer, we will repeat the preactivation ($z^{[2]}$) and then send to the $softmax$ function; the output will represent the probability of each label.

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \tag{7.3}$$

$$a^{[2]} = softmax(z^{[2]}) \tag{7.4}$$

The next step is the backward-propagation. In this step, we will be propagating the data from the last layer, this time backwards. We will be computing the chosen loss function, the cross-entropy, and updating the weights and biases in order to minimise it.

The cross-entropy loss function with $softmax$ can be defined as follows:

$$\begin{aligned} E &= -\sum_{k=1}^{n}[y_k ln(a_k^{[2]})] \\ &= -\sum_{k=1}^{n}[y_k ln(softmax(z_k^{[2]}))] \\ &= -\sum_{k=1}^{n}[y_k ln(\frac{e^{z_k^{[2]}}}{\sum_{j=1}^{n} e^{z_j^{[2]}}})] \\ &= -\sum_{k=1}^{n}[y_k ln(e^{z_k^{[2]}} - \sum_{j=1}^{n} e^{z_j^{[2]}})] \\ &= \sum_{k=1}^{n}[y_k ln(\sum_{j=1}^{n} e^{z_j^{[2]}})] - \sum_{k=1}^{n}[y_k z_k] \end{aligned} \tag{8.1}$$

Given that y is a one-hot vector, $\sum_{k=1}^{n} y_k = 1$. Additionally, $ln(\sum_{j=1}^{n} e^{z_j^{[2]}})$ is a constant term for all $k = 1, \cdots, n$, yielding the following equation:

$$E = ln(\sum_{j=1}^{n} e^{z_j^{[2]}}) - y^T z^{[2]} \tag{8.2}$$

For a full back-propagation, we need to compute, for all layers ($L = 1, 2$), $\frac{\partial E}{\partial W^{[L]}}$ and $\frac{\partial E}{\partial b^{[L]}}$, which, after applying the chain result, are shown to depend on $\frac{\partial E}{\partial z^{[L]}}$. As such, let us compute them first.

$\frac{\partial E}{\partial z^{[2]}}$ requires an additional step, which we will do now:

$$\frac{\partial}{\partial z_i^{[2]}}[ln(\sum_{j=1}^{n} e^{z_j^{[2]}})] = \frac{e^{z_i^{[2]}}}{\sum_{j=1}^{n} e^{z_j^{[2]}}} = softmax(z_i^{[2]}) \tag{9}$$

Now, we can procede with the main computations.

$$\begin{aligned} \frac{\partial E}{\partial z^{[2]}} &= softmax(z^{[2]}) - y \\ \frac{\partial E}{\partial z^{[1]}} &= \frac{\partial E}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \\ &= [W^{[2]^T} \frac{\partial E}{\partial z^{[2]}}] \odot \frac{d}{dx} ReLU(z^{[1]}) \end{aligned} \tag{10}$$

$$\begin{aligned} \frac{\partial E}{\partial W^{[L]}} &= \frac{\partial E}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial W^{[L]}} = \frac{\partial E}{\partial z^{[L]}} a^{[L-1]^T} \\ \frac{\partial E}{\partial b^{[L]}} &= \frac{\partial E}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial b^{[L]}} = \frac{\partial E}{\partial z^{[L]}} \end{aligned} \quad , \text{L} = 1, 2 \tag{11}$$

As for the accuracy and loss on the MLP, the results are as follows:

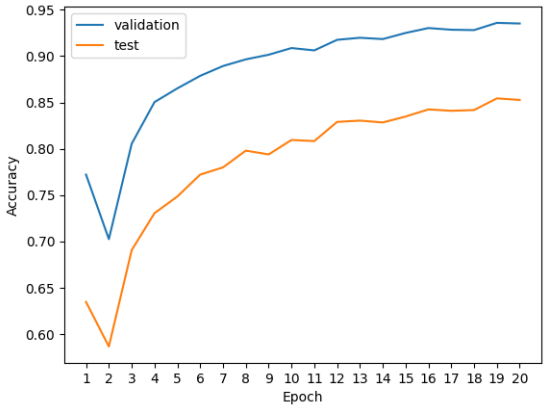

Figure 4: Accuracy on the training and testing sets, with the MLP

Question 2

2.1.

This exercise is an application of 1.1.b) using PyTorch, a Python library dedicated to machine learning. For further details on the exact calculations done by the module, please refer to that section or to the module's documentation [2].

Listing 3: Initialisation of the linear regression model.

```
def __init__(self, n_classes,
    n_features, **kwargs):
        """
        n_classes (int)
        n_features (int)
        """
        super().__init__()
        self.model =
            nn.Linear(n_features,
            n_classes)
```

Listing 4: Forward step of the linear regression model.

```
def forward(self, x, **kwargs):
        """
        x (batch_size x n_features):
            a batch of training
            examples
        """
        return self.model(x)
```

By experimenting with three different learning rates (0.001, 0.01 and 0.1), we found that the best configuration was the logistic regression with learning rate = 0.001, with a final validation accuracy of 70.19%.

Table 1: Final validation accuracies per learning rate with logistic regression.

Learning rate	**Validation accuracy**
0.001	70.19%
0.01	68.06%
0.1	61.28%

As for the training loss and the validation accuracy, as a function of the epoch number and with learning rate = 0.001, the results are as follows:

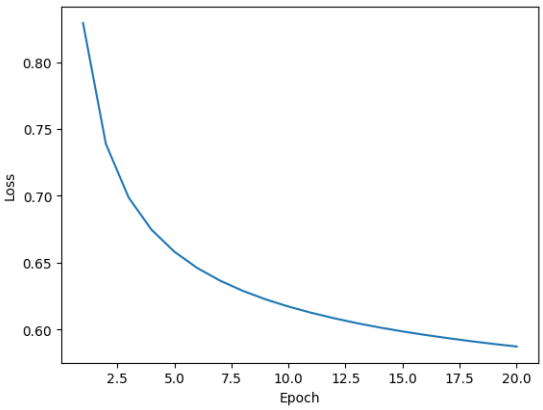

Figure 5: Training loss per epoch

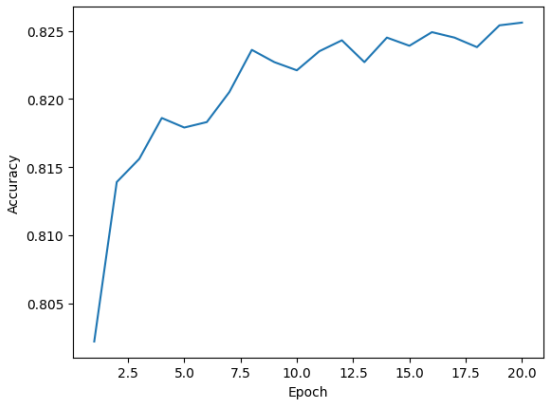

Figure 6: Validation accuracy per epoch

2.2.

This exercise is similar to 1.2.b): the exceptions involve the usage of *tanh* or *ReLU* as the activation function for all layers, instead of *ReLU* for the hidden layers (or, in our case, layer) and $softmax$ for the output layer, and a dropout, which specifies a probability for each of the elements of the hidden layers (after the activation function) to be set to 0 as a regularisation technique.

The initialisation and the forward step of our model are defined as follows:

Listing 5: Initialisation of the MLP model.

```
def __init__(self, n_classes, n_features, hidden_size, layers,
        activation_type, dropout, **kwargs):
    """
    n_classes (int)
    n_features (int)
    hidden_size (int)
    layers (int)
    activation_type (str)
    dropout (float): dropout probability
    """
    super().__init__()
    if activation_type == "tanh":
        activation_type = nn.Tanh
    elif activation_type == "relu":
        activation_type = nn.ReLU
    else:
        raise ValueError("Activation type must be either 'tanh' or 'relu'")
    dropout = nn.Dropout(dropout)
    params = [nn.Linear(n_features, hidden_size), activation_type(), dropout]
    for _ in range(layers-1):
        params.append(nn.Linear(hidden_size, hidden_size), activation_type(),
            dropout)
    params.append(nn.Linear(hidden_size, n_classes))
    self.model = nn.Sequential(*params)
```

Listing 6: Forward step of the MLP model.

```
def forward(self, x, **kwargs):
    """
    x (batch_size x n_features): a batch of training examples
    """
    return self.model(x)
```

To find the best configuration, we tuned (while leaving the other values as default) the learning rate, the hidden size, the dropout probability, or the activation function, having the following results:

Table 2: Final testing accuracies per learning rate with the MLP.

Learning rate	**Validation accuracy**
0.001	74.50%
0.01	85.99%
0.1	87.27%

Table 3: Final testing accuracies per hidden size with the MLP.

Hidden size	**Validation accuracy**
100	85.99%
200	88.17%

Table 4: Final testing accuracies per dropout probability with the MLP.

Dropout probability	**Validation accuracy**
0.3	85.99%
0.5	84.09%

As such, the best configuration found was the one with 200 hidden units, while leaving the other values as default. The training loss and the validation accuracy, as a function of the epoch number and with hidden units = 200 while the other values stay their default value, are as follows:

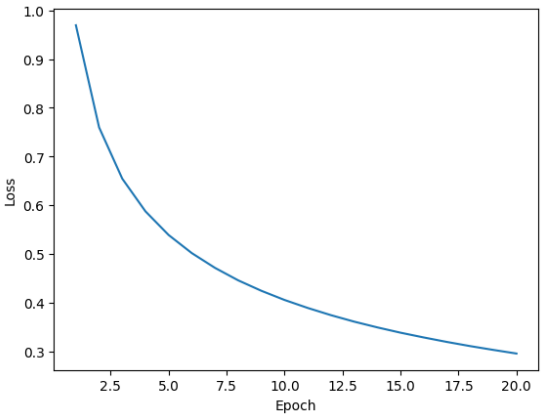

Figure 7: Training loss per epoch

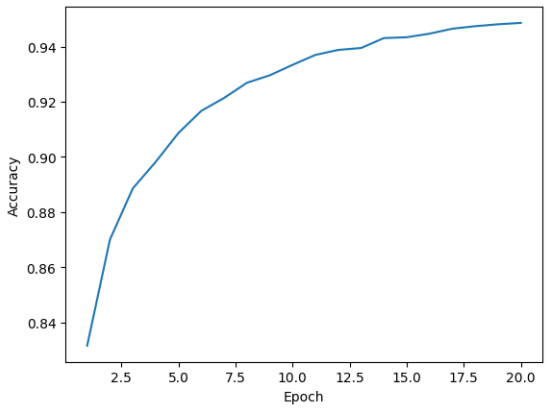

Figure 8: Validation accuracy per epoch

Table 5: Final testing accuracies per activation function with the MLP.

Activation function	**Validation accuracy**
ReLU	85.99%
tanh	82.57%

2.3.

This time, we tuned the number of hidden layers while keeping the remaining parameters as default, with the following results:

Table 6: Final testing accuracies per number of hidden layers with the MLP.

Hidden layers	**Validation accuracy**
1	85.99%
2	31.08%
3	39.60%

The best of the three configurations remained the default. The training loss and the validation accuracy for that configuration are as follows:

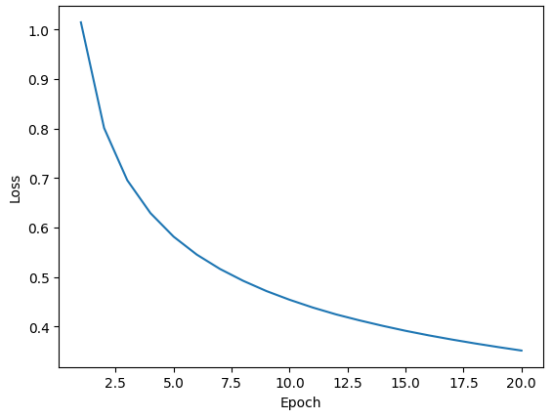

Figure 9: Training loss per epoch

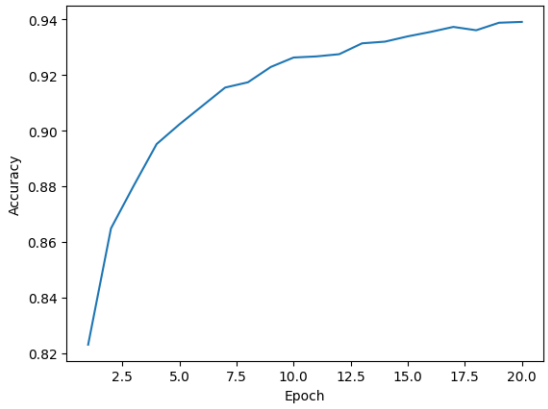

Figure 10: Validation accuracy per epoch

Question 3

3.1.

First of all, let us compute each entry of h, $h_i, i = 1, \cdots, K$:

$$\begin{aligned} h_i &= (W_i \cdot x)^2 \\ &= (\sum_{j=1}^{D} [W_{ij} x_j])^2 \\ &= (W_{i1}x_1 + W_{i2}x_2 + \cdots + W_{iD}x_D)^2 \\ &= \sum_{j=1}^{D}\sum_{k=1}^{D} [W_{ij}W_{ik}x_j x_k] \\ &= W_{i1}^2 x_{i1}^2 + 2W_{i1}W_{i2}x_{i1}x_{i2} + \cdots + 2W_{i1}W_{iD}x_{i1}x_{iD} \ (D \text{ terms}) \\ &+ W_{i2}^2 x_{i3}^2 + 2W_{i2}W_{i3}x_{i2}x_{i3} + \cdots + 2W_{i2}W_{iD}x_{i2}x_{iD} \ (D-1 \text{ terms}) \\ &+ \cdots \\ &+ W_{iD}^2 x_{iD}^2 \ (1 \text{ term}) \end{aligned} \tag{12}$$

This gives us a total of $\sum_{j=1}^{D} j = \frac{D(D+1)}{2}$ distinct terms, corresponding to the desired dimensionality of $\phi(x)$. This allows us to define $\phi(x)$ and A_θ as follows:

$$\phi(x) = \begin{bmatrix} x_{i1}^2 & x_{i1}x_{i2} & \cdots & x_{i1}x_{iD} & x_{i2}^2 & x_{i2}x_{i3} & \cdots & x_{i2}x_{iD} & \cdots & x_{iD}^2 \end{bmatrix}^T \tag{13}$$

$$A_\theta = \begin{bmatrix} A_{\theta 1} \\ A_{\theta 2} \\ \vdots \\ A_{\theta K} \end{bmatrix} \quad A_{\theta i} = \begin{bmatrix} W_{i1}^2 & 2W_{i1}W_{i2} & \cdots & 2W_{i1}W_{iD} & W_{i2}^2 & 2W_{i2}W_{i3} & \cdots & 2W_{i2}W_{iD} & \cdots & W_{iD}^2 \end{bmatrix} \tag{14}$$

Using these parameters, we can define $h = A_\theta \phi(x)$ as a linear transformation of $\phi(x)$, *Q.E.D.*

3.2.

Firstly, we know that:

$$\hat{y} = v^T h = v^T A_\theta \phi(x) \tag{15}$$

Since A_θ is a linear transformation of $\phi(x)$, we can conclude that $\hat{y}$ is also a linear transformation of $\hat{y}$.

c_θ can then be defined as:

$$c_\theta^T = v^T A_\theta \Leftrightarrow c_\theta = A_\theta^T v \tag{16}$$

Each $c_{\theta j}, j = 1, \ldots, \frac{D(D+1)}{2}$ is, then:

$$c_{\theta j} = \sum_{i=1}^{K} [A_{\theta ij} v_i] \tag{17}$$

This, however, does not mean that $\hat{y}$ is a linear model in terms of the original parameters $\theta(W, v)$, as both A_θ and $\phi(x)$ were obtained using non-linear transformations of W and x, respectively.

3.3.

Let us begin by developing $\hat{y}$:

$$\begin{aligned} \hat{y} &= v^T h \\ &= v^T (Wx)^2 \\ &= v^T (Wx) \odot (Wx) \end{aligned} \tag{18}$$

For the next step, we need to prove that $v^T(Wx) \odot (Wx) = (Wx)^T v'(Wx)$, where v' is a diagonal matrix containing the entries of the vector v. This proof will be shown for $v \in \mathbb{R}^3, W \in \mathbb{R}^{3x2}, x \in \mathbb{R}^3$, but it can be extended to other dimensions:

$$\begin{gathered} W = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} x = \begin{bmatrix} g \\ h \end{bmatrix} v = \begin{bmatrix} i \\ j \\ k \end{bmatrix} \Rightarrow v' = diag(i, j, k) \\ Wx = \begin{bmatrix} ag + bh \\ cg + dh \\ eg + fh \end{bmatrix} \\ (Wx)^T v'(Wx) = (ag + bh)^2 i + (cg + dh)^2 j + (eg + fh)^2 k \\ g(Wx) = (Wx)^2 = (Wx) \odot (Wx) = \begin{bmatrix} (ag + bh)^2 \\ (cg + dh)^2 \\ (eg + fh)^2 \end{bmatrix} \\ v^T g(Wx) = (ag + bh)^2 i + (cg + dh)^2 j + (eg + fh)^2 k \end{gathered} \tag{19}$$

Thus, $v^T g(Wx) = (Wx)^T v'(Wx), Q.E.D.$

As such, we can then define the following equation:

$$\begin{aligned} \hat{y} &= v^T (Wx) \odot (Wx) \\ &= (Wx)^T v'(Wx) \\ &= x^T W^T v' W x \\ &= x^T S x \end{aligned} \tag{20}$$

In this context, given that $W^T W$ is, by definition, a symmetrical matrix, and that v' is a diagonal matrix, then $S = W^T v' W$ is also a symmetrical matrix with dimensions DxD. As such, it only has $\frac{D(D+1)}{2}$ distinct entries.

Considering that $x^T S x = c_\theta \phi(x)$, to prove that $\forall c \in \mathbb{R}^{\frac{D(D+1)}{2}}, \exists \theta : c_\theta = c$, we can equivalently prove that $\forall S \in \mathbb{S}^{DxD}, \exists W, v : S = W^T v' W$, with v' being the already mentioned diagonal

matrix containing the entries of the vector v. As such, we must ensure that $span(W) = \mathbb{R}^D$, i.e., $rank(W) = D$.

Given that, for $W \in \mathbb{R}^{K\text{x}D}$, $rank(W) \leq min(K, D)$:

- If $K \geq D, \exists W : rank(W) = D$, so our model can be parametrised with c_θ instead of θ;
- If $K < D, \forall W : rank(W) \leq K < D \Rightarrow \nexists W : rank(W) = D$, so an equivalent parametrisation is not guaranteed.

This means that, if existent, we're dealing with a linear model w.r.t. c_θ.

3.4.

This problem is distinct from usual feed-forward neural networks for a simple reason: in the previous subsections, we were able to transform our parameters to create a linear model w.r.t. $c_\theta = A_\theta^T v$. In this context, X comes as a simple extension of the problem to N samples, so the previous statements remain true.

Minimising $L(c_\theta, D)$ has a well-known closed-form solution. Let us deduce it, but first, it's useful to convert our current notation to matrix-vector notation:

$$\frac{1}{2}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2 = \frac{1}{2}\sum_{n=1}^{N}(y_n - \hat{y_n})^2 = \frac{1}{2}\|y - Xc_\theta\|^2 \tag{21}$$

Now, we can obtain our closed-form function by equating its gradient to 0:

$$\begin{aligned}
&\frac{1}{2}\nabla_{c_\theta}\|y - Xc_\theta\|^2 = 0 \Leftrightarrow \\
\Leftrightarrow &\frac{1}{2}\nabla_{c_\theta}(c_\theta^T X^T X c_\theta - 2c_\theta X^T y + \|y\|^2) = 0 \\
\Leftrightarrow &X^T X c_\theta - X^T y = 0 \\
\Leftrightarrow &X^T X c_\theta = X^T y \\
\Leftrightarrow &c_\theta = (X^T X)^{-1} X^T y
\end{aligned} \tag{22}$$

This holds true if X has full column rank, since, in that case, $(X^T X)^{-1}$ exists.

Contributions

All exercises, apart from 1.2.a) and 3.1. were done as a pair.
1.2.a) was concluded solely by Catarina, while 3.1. was done by David.

References

[1] A. Wichert, *Machine Learning - A Journey to Deep Learning*. World Scientific Publishing, 2021.

[2] "PyTorch Documentation." https://pytorch.org/docs/stable/index.html. Accessed: 2022/12/22.