TÉCNICO LISBOA

Deep Learning

MSc Electrical and Computer Engineering

Homework 1 - Report

Authors:

António Jotta (99893) antonio.jotta@tecnico.ulisboa.pt
António Vasco Morais de Carvalho (102643) antonio.v.morais.c@tecnico.ulisboa.pt

Group 17

Group contribution:
António Jotta: Question 2; Question 3.3, 3.4.
António Morais: Question 1; Question 3.1, 3.2.
Both: Report (repective questions with review on both sides).

2024/2025 – 1st Semester, P2

Contents

Question 1

In this section, a linear classifier is implemented to classify landscape images using a pre-processed version of the Intel Image Classification dataset. The dataset consists of 17.034 RGB images, each with a resolution of 48x48 pixels, categorized into six classes: 0: buildings, 1: forest, 2: glacier, 3: mountain, 4: sea, 5: street.

1

In this first part of the exercise, the Perceptron algorithm was implemented and evaluated.

a)

The `update_weight` method of the `Perceptron` class was implemented, after which the model was trained for 100 `epochs` on the training set, yielding the following accuracy results:

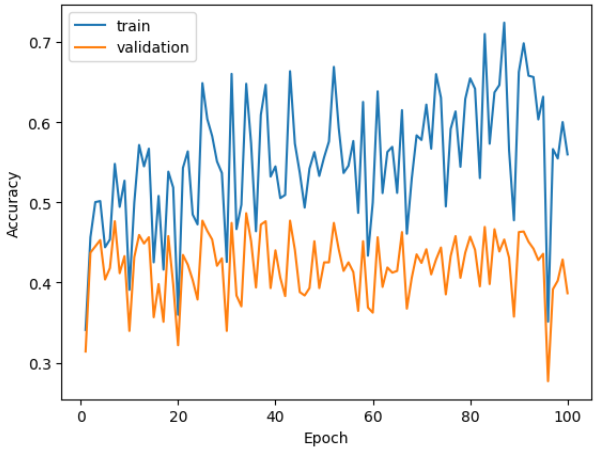


Figure 1: Train and validation accuracies of the Perceptron model as a function of the number of epochs.

Accuracy		
Train	**Validation**	**Test**
0.5598	0.3868	0.3743

Table 1: Train, validation and test accuracies.

From Figure 1, it is evident that the training accuracy fluctuates significantly while showing a general upward trend over time, reaching a peak above 70% by the 90$^{\text{th}}$ epoch. This indicates that the perceptron is able to learn from the training data, although the high variability suggests sensitivity to the data distribution.

In contrast, the validation accuracy consistently remains within the range of 35% to 50%. The gap between training and validation accuracy indicates that the model does not generalize well on the validation data, due to overfitting on the training data.

The limited generalization capability is further highlighted by the test accuracy obtained of 0.3743, suggesting that the Perceptron model may be too simplistic for this classification task.

2

Now, a Logistic Regression (LR) classifier was implemented using stochastic gradient descent (SGD) for training. Specifically, two versions of the classifier were developed: (i) a non-regularized logistic regression model, and (ii) a LR model with ℓ_2-regularization.

a)

Starting with LR without ℓ_2-regularization, the `update_weight` method of the `Logistic Regression` class was implemented. Ignoring the `l2_penalty` argument and training the classifier for 100 `epochs` with a `learning_rate` of 0.001, the plot below was obtained.

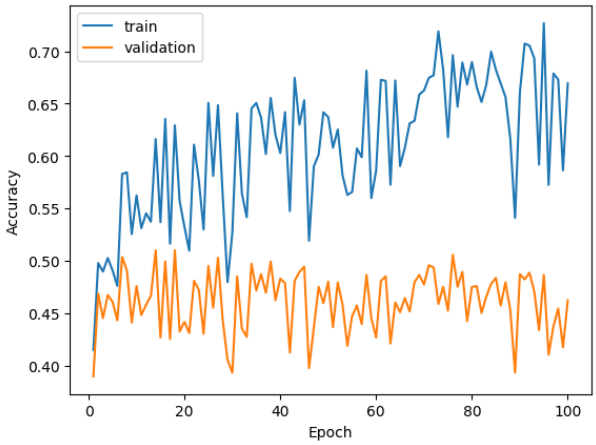


Figure 2: Train and validation accuracies of the LR model as a function of the number of epochs.

Similar to the Perceptron model, the LR model exhibits an upward trend in training accuracy, peaking above 70%. However, it demonstrates poor generalization on the validation data and overfitting to the training data, then evidenced by a test score of 0.4597.

b)

The `update_weight` method was then changed to support ℓ_2-regularization, where the strength of the regularization is defined by the `l2_penalty` argument. The model was trained with a `l2_penalty` of 0.01 over 100 `epochs`.

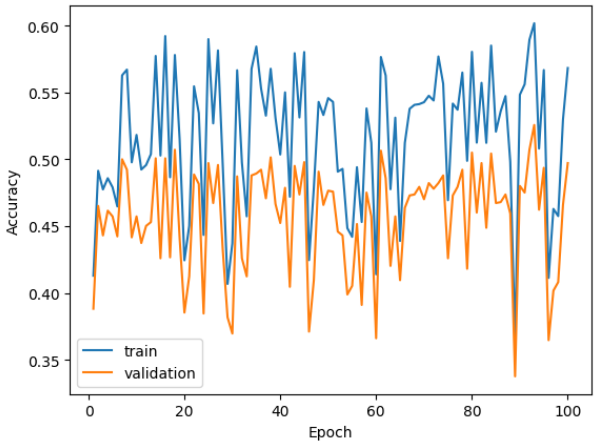


Figure 3: Train and validation accuracies of the LR model, with ℓ_2-regularization, as a function of the number of epochs.

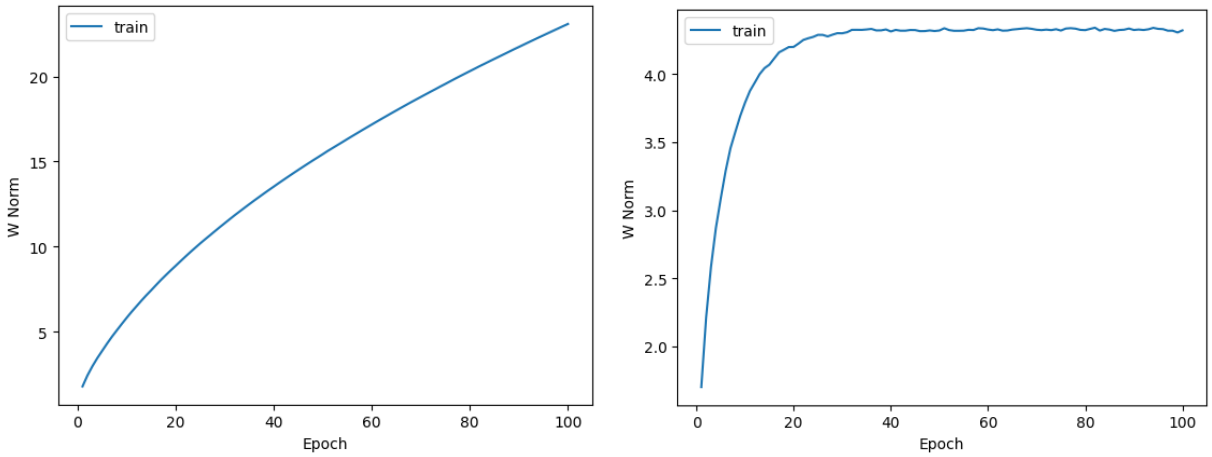
Accuracy		
Train	**Validation**	**Test**
0.5683	0.4972	0.5053

Table 2: Train, validation and test accuracies.

A comparison between Figure 2 and Figure 3 reveals a noticeable reduction in the gap between training and validation accuracies with the inclusion of ℓ_2-regularization. This suggests a reduced tendency for overfitting, as regularization constrains the size of the model's weights, resulting in improved generalization on unseen data, as evidenced by a test score of 0.5053.

c)

In this section a closer look is taken at how regularization impacts the values of the weights of the LR classifier. For that, the ℓ_2-norm of the weights of both the non-regularized and regularized versions of the LR classifier are shown below.



(a) Without ℓ_2-regularization. **(b)** With ℓ_2-regularization.

Figure 4: Norms of the weights of the LR classifier as a function of the number of epochs.

ℓ_2-norm of the weights	
Non-regularized	**Regularized**
23.0806	4.3217

Table 3: ℓ_2-norm of the weights with and without regularization.

In Figure 4a, the norm of the weights appears to increase without restriction over the epochs ending at a value of 23.0806. The absence of regularization allows the weights to grow continuously, leading to overfitting as the model excessively adapts to the training data.

Conversely, Figure 4b illustrates the impact of regularization, where the norm of the weights increases rapidly before stabilizing as training progresses around the value of 4.3217. This stabilization is attributed to the ℓ_2 penalty, which constrains weight growth and helps mitigate overfitting on the training data.

d)

If ℓ_1-regularization were applied instead of ℓ_2, the resulting weights at the end of training would be more sparse. The first one imposes an absolute value penalty on the weights, encouraging many to become exactly zero, thereby performing implicit feature selection by eliminating less significant features. In contrast, ℓ_2-regularization penalizes the squared values of the weights, resulting in weights that are close to zero, distributing importance more evenly across all features.

3

Next, a multi-layer perceptron (MLP) - a feed-forward neural network - is implemented using as input the original feature representation.

a)

The MLP was implemented with a single hidden layer and trained using the backpropagation algorithm (implemented in the `train_epoch` method of the `MLP` class). The hidden layer consists of 100 units, using the ReLU activation function. The output layer employs multinomial logistic loss (also known as cross-entropy) as the loss function. Bias terms were included for the hidden units and initialized as zero vectors. The weight matrices were initialized using a normal distribution $w_{ij} \approx \mathcal{N}(\mu, \sigma^2)$, with $\mu = 0.1$ and $\sigma^2 = 0.1^2$. The model was then trained for 20 `epochs` with SGD, with a `learning_rate` of 0.001.

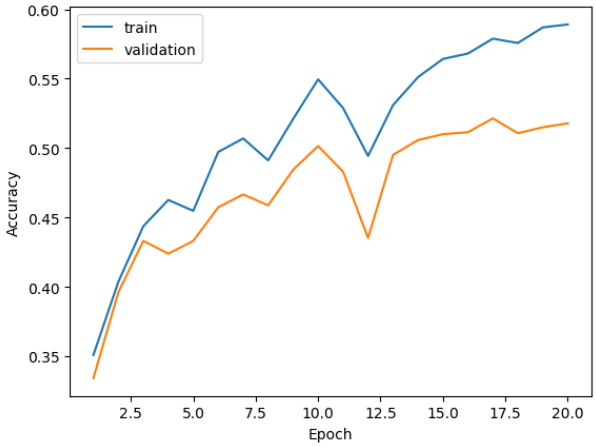


Figure 5: Train and validation accuracies of the MLP model as a function of the number of epochs.

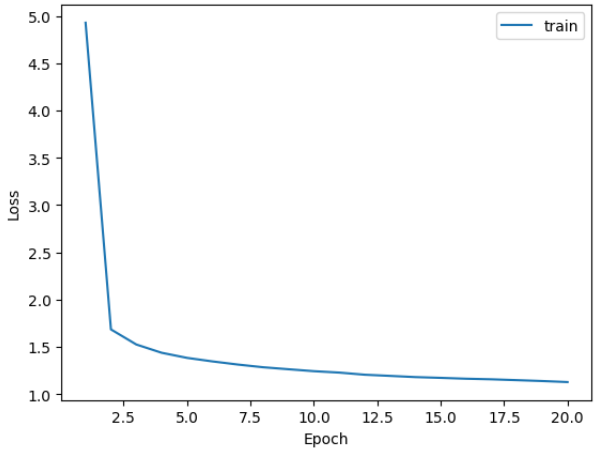


Figure 6: Train loss of the MLP model as a function of the number of epochs.

From Figure 5, the training accuracy steadily increases, indicating that the model is fitting well to the training data. The validation accuracy also shows an upward trend; however, the growing gap between the train and validation curves suggests overfitting, where the model begins to generalize poorly to new data. Despite this, the model achieves a **test accuracy of 0.5320**, the highest yet achieved.

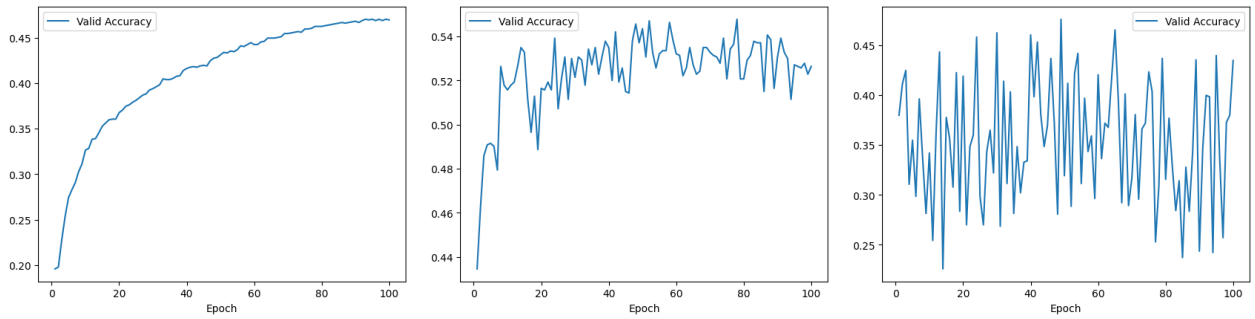
The loss returned by the `train_epoch` method represents the average loss computed across all data points, as recommended by the faculty. Regarding the loss plot - Figure 6 - it illustrates a sharp decrease during the initial epochs, reflecting rapid learning from the training data. The initial high values of the loss may be attributed to the random initialization of the model's weights. Subsequently, the loss curve approaches a plateau, indicating effective generalization on the training data.

Question 2

In Question 1, gradient backpropagation was implemented manually. In this section, the same classifier system is implemented using a deep learning framework, PyTorch, with automatic differentiation.

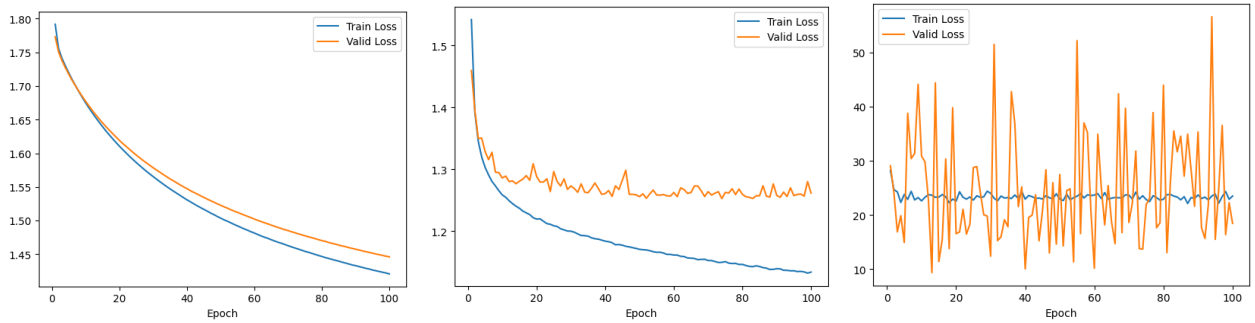
1

A linear model was implemented using LR and SGD for training. The model was trained with a `batch_size` of 32 and an ℓ_2-regularization parameter (`l2_decay`) set to 0.01. The training process was conducted for 100 epochs, and the `learning_rate` (let us denote as η) was tuned over the following values: $\{0.00001, 0.001, 0.1\}$.



(a) $\eta = 0.00001$. **(b)** $\eta = 0.001$. **(c)** $\eta = 0.1$.

Figure 7: Validation accuracy of the LR model with SGD for $\eta = \{0.00001, 0.001, 0.1\}$.



(a) $\eta = 0.00001$. **(b)** $\eta = 0.001$. **(c)** $\eta = 0.1$.

Figure 8: Train and validation loss of the LR model with SGD for $\eta = \{0.00001, 0.001, 0.1\}$.

Learning Rate	Accuracy	
	Validation	Test
0.00001	0.4694	0.4623
0.001	0.5264	0.5247
0.1	0.4345	0.4433

Table 4: Validation and test accuracies for different `learning_rate` values.

As evidenced in Table 4, the learning rate of 0.001 achieved the highest validation accuracy of 0.5264, with a corresponding test accuracy of 0.5247. From the validation loss curves in Figure 8, it is clear that the behavior of the model is strongly influenced by the choice of the learning rate.

For $\eta = 0.00001$ in Figure 8a, the loss decreases slowly for both the train and validation sets. This slow convergence is due to the small learning rate, which results in minimal updates to the model weights during each epoch. Although the losses steadily decrease, the model requires significantly more epochs to reach optimal performance.

In contrast, for $\eta = 0.001$ (Figure 8b), the training loss decreases smoothly, and the validation loss stabilizes without significant fluctuations. This behavior indicates that the learning rate is appropriately balanced, enabling the model to converge effectively while generalizing well to unseen data. However, slight overfitting to the training data is observed, as the gap between the training and validation loss curves increases. Nonetheless, the model achieves the highest accuracies, suggesting that it captures important patterns in the data.

Lastly, with $\eta = 0.1$, the validation loss fluctuates considerably and fails to stabilize, while the training loss shows minor fluctuation (Figure 8c). These fluctuations are caused by the large learning rate, which leads to the model overshooting the optimal solution during weight updates. Consequently, the model fails to converge properly, leading to poor generalization and the lowest validation and test accuracies.

2

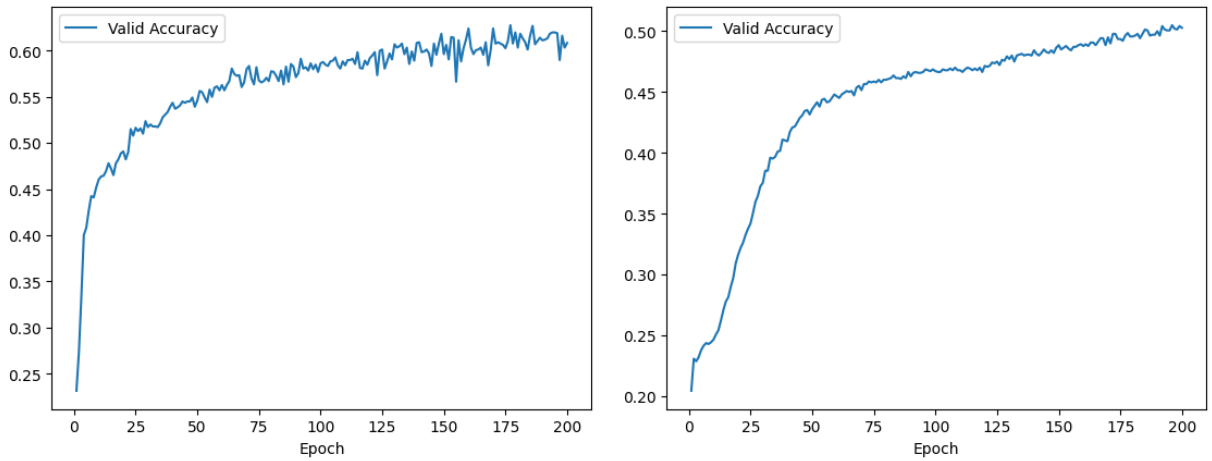
A feed forward neural network was implemented with the hyperparameters shown in the table below used as default:

Number of Epochs	200
Learning Rate	0.002
Hidden Size	200
Number of Layers	2
Dropout	0.3
Batch Size	64
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 5: Default hyperparameters.

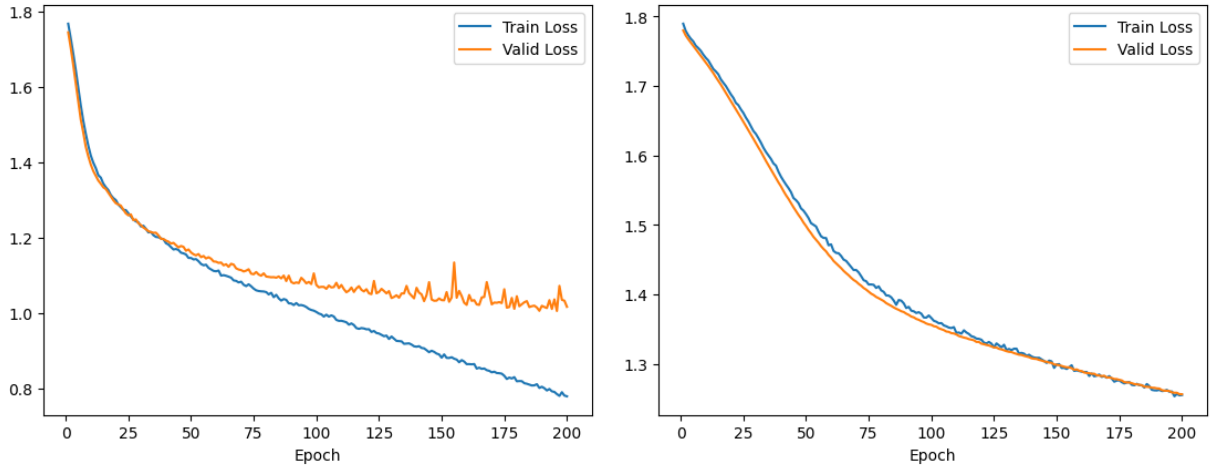
a)

Two models were trained: the first with the default hyperparameters listed in Table 5, while the second used a `batch_size` of 512, keeping all other hyperparameters at their default values.



(a) `batch_size` $= 64$. **(b)** `batch_size` $= 512$.

Figure 9: Validation accuracy of the MLP model for `batch_size` $= \{64, 512\}$.



(a) `batch_size` $= 64$. **(b)** `batch_size` $= 512$.

Figure 10: Train and validation loss of the MLP model for `batch_size` $= \{64, 512\}$.

Batch Size	Accuracy		Training Time
	Validation	Test	
64	0.6083	0.6087	1m45s
512	0.5028	0.5190	1m24s

Table 6: Validation and test accuracies and training time for the different `batch_size` values.

The results in Table 6 highlight differences in both performance and execution time between the two batch sizes.

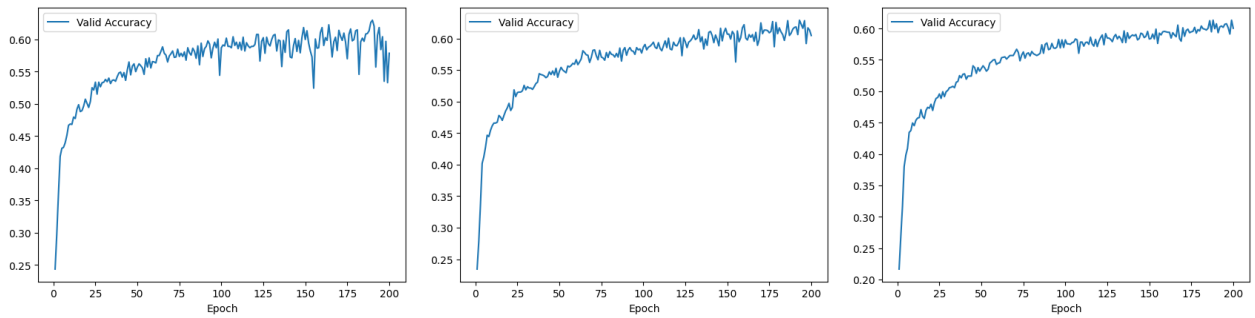
Regarding performance, differences can be attributed to the effect of batch size on the training dynamics. The smaller batch size allows the model to update its weights more frequently (allowing a closer fit to the training data), introducing more stochasticity in the gradient updates. This added noise can help the model converge to better solutions and escape local minima, leading to improved accuracy. However, the frequent weight updates can lead to overfitting, as observed in Figure 10a, where the gap between the training and validation curves increases, and the final train loss is lower than that in Figure 10b.

In contrast, a larger batch size smooths the gradient estimates, reducing the variability in weight updates. While this can occasionally result in more stable training as seen in Figure 10b (exhibits less overfitting, as evidenced by the train and validation loss curves being closer together when compared to Figure 10a), it may also prevent the model from exploring diverse regions of the loss landscape, leading to suboptimal solutions. This effect is evident in the lower accuracies observed with a batch size of 512.

Concerning execution time, the batch size of 64 presents a greater time of execution. A smaller batch size requires more iterations per epoch due to processing fewer samples at a time, resulting in additional computational overhead from more frequent weight updates. On the other hand, the larger batch size processes more samples per iteration, reducing the total number of iterations and resulting in faster training.

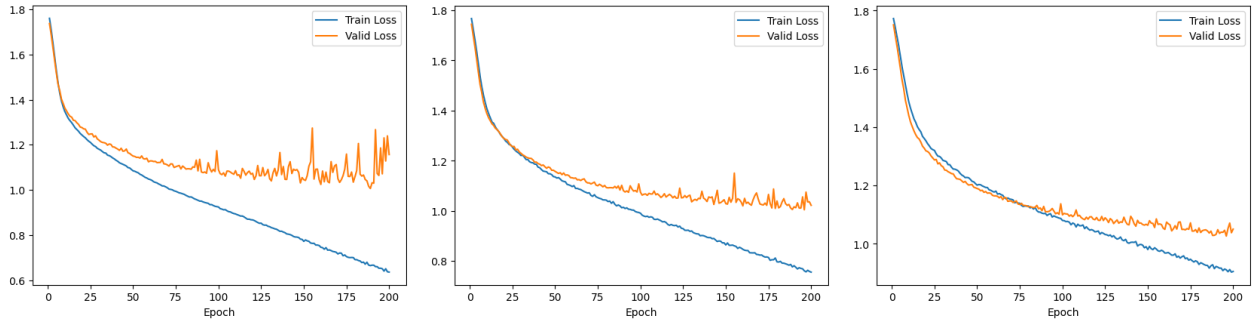
b)

To study the effects of the `dropout` hyperparameter on the model, three MLP models were trained with `dropout` $= \{0.01, 0.25, 0.5\}$, while maintaining all other hyperparameters at their default values.



(a) `dropout` $= 0.01$. **(b)** `dropout` $= 0.25$. **(c)** `dropout` $= 0.5$.

Figure 11: Validation accuracy of the MLP model for `dropout` $= \{0.00001, 0.001, 0.1\}$.



(a) `dropout` $= 0.01$. **(b)** `dropout` $= 0.25$. **(c)** `dropout` $= 0.5$.

Figure 12: Train and validation loss of the MLP model for `dropout` $= \{0.00001, 0.001, 0.1\}$.

Dropout	Accuracy	
	Validation	Test
0.01	0.5783	0.5777
0.25	0.6047	0.6077
0.5	0.6004	0.5967

Table 7: Validation and test accuracies for the different `dropout` values.

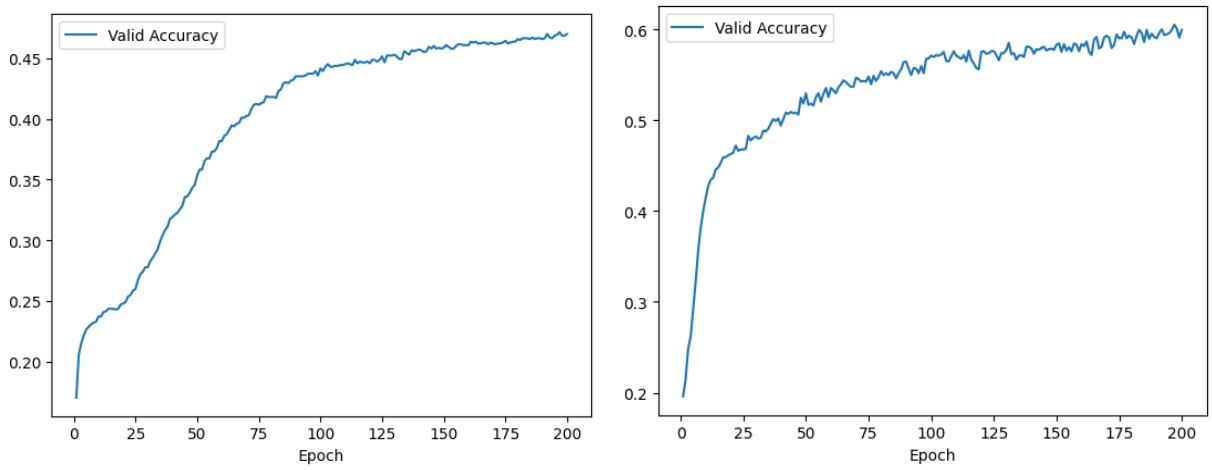
For a dropout rate of 0.01, the training loss decreases steadily (Figure 12a), but the validation loss curve exhibits considerable noise towards the end of training. This suggests that the model is overfitting to the training data, as it fails to generalize well to the validation set. The low dropout rate provides insufficient regularization, allowing the model to memorize the training data, including noise and irrelevant patterns. Consequently, the final validation and test accuracies are the lowest among the three configurations.

With a dropout rate of 0.25, the training and validation loss curves are more closely aligned and decrease smoothly - Figure 12b. The validation loss shows less noise compared to 0.01 of dropout, and the training loss reaches its lowest value here. This indicates that the model is effectively learning from the training data while maintaining good generalization to unseen data. The balance achieved with this dropout rate results in the highest validation and test accuracies among the tested values.

At a dropout rate of 0.5, the validation loss (Figure 12c) displays the least noise among all three settings. However, the training loss is higher than in the 0.25 dropout scenario, indicating that the model is not fitting the training data as closely. The higher dropout rate introduces stronger regularization by randomly dropping half of the neurons during training. While this reduces overfitting (evidenced by the small gap between training and validation losses), it also limits the model's capacity to learn complex patterns in the data. As a result, the validation and test accuracies are slightly lower than those achieved with a dropout rate of 0.25.

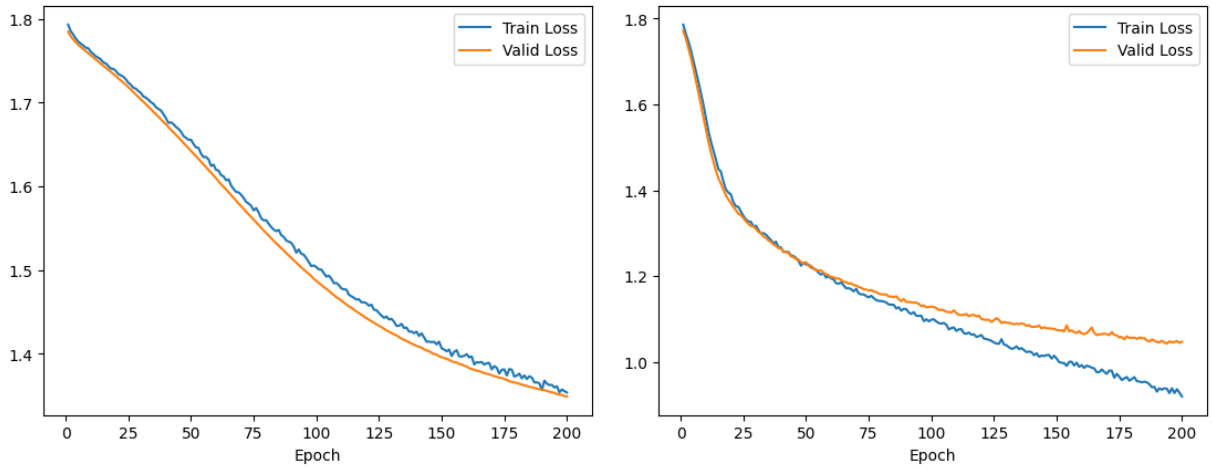
c)

The MLP model was implemented with a `batch_size` of 1024, `momentum` values of 0.0 and 0.9, while keeping all other hyperparameters at their default values.



(a) `momentum` $= 0.0$. **(b)** `momentum` $= 0.9$.

Figure 13: Validation accuracy of the MLP model for: `batch_size` $= 1024$ and `momentum` $= \{0.0, 0.9\}$.



(a) `momentum` $= 0.0$. **(b)** `momentum` $= 0.9$.

Figure 14: Train and validation loss of the MLP model for: `batch_size` $= 1024$ and `momentum` $= \{0.0, 0.9\}$.

Momentum	**Accuracy**	
	Validation	**Test**
0.0	0.4701	0.4887
0.9	0.5997	0.5997

Table 8: Validation and test accuracies for the different `momentum` values.

In Figure 14a, corresponding to a momentum of 0.0, both train and validation loss curves decrease steadily over the epochs, resembling linear trends. The difference between these curves

is minimal throughout the training process, indicating low overfitting and good generalization to unseen data. However, the overall training loss remains relatively high, which leads to low validation and test accuracies.

In contrast, Figure 14b, shows that (for a momentum of 0.9) the train and validation losses decrease more rapidly, following an almost exponential decay. The train loss reaches a lower value when compared to the case with zero momentum. There is a more noticeable gap between the train and validation loss curves, especially after the 75^{th} epoch, where the difference grows to about 0.2 at the end of the training. This indicates increased overfitting to the training data as the number of epochs increases. Despite overfitting, both train and validation losses are lower than those in Figure 14a, resulting in higher validation and test accuracies.

Following this analysis, when momentum is not used, the optimizer (SGD) updates parameters based solely on the current gradient, leading to slow convergence and limited learning, as reflected by higher losses and lower accuracy. Incorporating momentum accelerates convergence by leveraging past gradients, effectively overcoming local minima. While this approach reduces losses and improves accuracy, the increased gap between training and validation losses indicates a higher risk of overfitting.

Question 3

This section explores a feed-forward neural network with a single hidden layer, using the activation function $g(z) = z(1 - z)$. Unlike widely used activation functions such as tanh, sigmoid, or ReLU, this specific choice allows the model to be reinterpreted as a linear model under certain assumptions, thanks to a reparametrization approach.

The network is designed for a univariate regression task, where the predicted output $\hat{y} \in \mathbb{R}$ is computed as $\hat{y} = \mathbf{v}^\top \mathbf{h} + v_0$. Here, $\mathbf{h} \in \mathbb{R}^K$ represents the internal representations obtained from the activation function, expressed as $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$. The input vector is denoted as $\mathbf{x} \in \mathbb{R}^D$, and the model parameters are given by $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0) \in \mathbb{R}^{K\times D} \times \mathbb{R}^K \times \mathbb{R}^K \times \mathbb{R}$.

1

It is intended to demonstrate that the hidden representation

$$\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{1}$$

can be expressed as a linear transformation of a feature transformation $\phi(\mathbf{x})$, and to determine the mapping ϕ and the matrix $\mathbf{A}_\Theta$.

The hidden representation $\mathbf{h} \in \mathbb{R}^K$ is computed element-wise as

$$h_k = g(z_k), \quad \text{where } z_k = \mathbf{w}_k^\top \mathbf{x} + b_k. \tag{2}$$

Using the activation function $g(z) = z(1 - z)$, expression (2) expands to

$$h_k = z_k(1 - z_k) = z_k - z_k^2,$$

and then substituting $z_k = \mathbf{w}_k^\top \mathbf{x} + b_k$, the following expression is obtained:

$$h_k = (\mathbf{w}_k^\top \mathbf{x} + b_k) - (\mathbf{w}_k^\top \mathbf{x} + b_k)^2. \tag{3}$$

From (3), the first term $(\mathbf{w}_k^\top \mathbf{x} + b_k)$, represents a linear combination of the input features. The second term $(\mathbf{w}_k^\top \mathbf{x} + b_k)^2$, can be expanded to include quadratic terms. Expanding the square in the second term gives:

$$\begin{aligned}(\mathbf{w}_k^\top \mathbf{x} + b_k)^2 &= (\mathbf{w}_k^\top \mathbf{x})^2 + 2\mathbf{w}_k^\top \mathbf{x} b_k + b_k^2 = \\ &= \sum_{i=1}^{D} w_{k,i}^2 x_i^2 + 2\sum_{i<j} w_{k,i} w_{k,j} x_i x_j + 2b_k \sum_{i=1}^{D} w_{k,i} x_i + b_k^2.\end{aligned} \tag{4}$$

Combining the linear and the obtained quadratic term in (4), (3) becomes:

$$\begin{aligned}h_k &= (\mathbf{w}_k^\top \mathbf{x} + b_k) - \left(\sum_{i=1}^{D} w_{k,i}^2 x_i^2 + 2\sum_{i<j} w_{k,i} w_{k,j} x_i x_j + 2b_k \sum_{i=1}^{D} w_{k,i} x_i + b_k^2 \right) = \\ &= (b_k - b_k^2) + \sum_{i=1}^{D} (w_{k,i} - 2b_k w_{k,i}) x_i - \sum_{1 \le i \le j \le D} w_{k,i} w_{k,j} x_i x_j.\end{aligned} \tag{5}$$

The hidden representation $\mathbf{h}$ can thus be written as a linear transformation of a feature mapping $\phi(\mathbf{x})$, where $\phi(\mathbf{x})$ consists of constant, linear, and quadratic terms. The feature transformation (mapping) $\phi(\mathbf{x})$ is given as

$$\phi(\mathbf{x}) = \begin{bmatrix} 1, & x_1, & \ldots, & x_D, & x_1^2, & \ldots, & x_D^2, & x_1 x_2, & \ldots, & x_D^2 \end{bmatrix}^\top, \tag{6}$$

where $\phi_1(\mathbf{x}) = 1$ (constant term), $\phi_{i+1}(\mathbf{x}) = x_i$ for $i = 1, \ldots, D$ (linear terms) and $\phi_{ij} = x_i x_j$ for $1 \le i \le j \le D$ (quadratic terms). Moreover, the number of features of $\phi(\mathbf{x})$ is given by its dimension

$$\dim(\phi(\mathbf{x})) = 1 + D + \frac{D(D+1)}{2} = \frac{(D+1)(D+2)}{2}.$$

Matrix $\mathbf{A}_\Theta$ encodes the relationship between $\mathbf{h}$ and $\phi(\mathbf{x})$. The matrix $\mathbf{A}_\Theta \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}}$ is constructed from the coefficients of the terms in $\phi(\mathbf{x})$, which are determined by $\mathbf{W}$ and $\mathbf{b}$. Following from (5), the elements of $\mathbf{A}_\Theta$ are as follows

$$\begin{cases} (\mathbf{A}_\Theta)_{k,1} = b_k - b_k^2 & \rightarrow \text{for } \phi_1(\mathbf{x}) = 1, \\ (\mathbf{A}_\Theta)_{k,i+1} = w_{k,i} - 2b_k w_{k,i}, \quad i = 1, \ldots, D & \rightarrow \text{for } \phi_{i+1}(\mathbf{x}) = x_i, \\ (\mathbf{A}_\Theta)_{k,m} = -w_{k,i} w_{k,j} & \rightarrow \text{for } \phi_{ij}(\mathbf{x}) = x_i x_j \ (m \text{ indexing } x_i x_j,\ 1 \le i \le j \le D). \end{cases}$$

Therefore, the matrix can be represented as

$$\mathbf{A}_\Theta = \begin{bmatrix} b_1 - b_1^2 & w_{1,1} - 2b_1 w_{1,1} & \cdots & w_{1,D} - 2b_1 w_{1,D} & -w_{1,1}^2 & \cdots & -w_{1,1} w_{1,2} & -w_{k,D}^2 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots \\ b_K - b_K^2 & w_{K,1} - 2b_K w_{K,1} & \cdots & w_{K,D} - 2b_K w_{K,D} & -w_{K,1}^2 & \cdots & -w_{K,1} w_{K,2} & -w_{K,D}^2 \end{bmatrix},$$

where each row of $\mathbf{A}_\Theta$ corresponds to the weights and biases of a hidden neuron and combines the contributions of constant, linear and quadratic terms. Thus, the hidden representation $\mathbf{h}$ can be expressed as

$$\mathbf{h} = \mathbf{A}_\Theta \phi(\mathbf{x}). \tag{7}$$

Therefore, showing that $\mathbf{h}$ is a linear transformation of $\phi(\mathbf{x})$.

2

Based on the claim made in the previous question we want to show that $\hat{y}$ is a linear transformation of $\phi(\mathbf{x})$, i.e, $\hat{y}$ can be written as $\hat{y}(\mathbf{x}, \mathbf{c}_\Theta) = \mathbf{c}_\Theta^\top \phi(\mathbf{x})$ for some $\mathbf{c}_\Theta \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ and determine the corresponding parameter vector $\mathbf{c}_\Theta$.

To show that $\hat{y}$ can be written as a linear transformation of $\phi(\mathbf{x})$, start from the model's prediction

$$\hat{y} = \mathbf{v}^\top \mathbf{h} + v_0, \tag{8}$$

where $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$. From the first part of the problem, it has been established that $\mathbf{h}$ can be expressed by (7), where $\mathbf{A}_\Theta \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}}$ is a matrix depending on the parameters $(\mathbf{W}, \mathbf{b})$, and $\phi(\mathbf{x}) \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ is the feature transformation independent of Θ. Substituting this into (8) gives:

$$\begin{aligned} \hat{y} &= \mathbf{v}^\top (\mathbf{A}_\Theta \phi(\mathbf{x})) + v_0 = \\ &= (\mathbf{v}^\top \mathbf{A}_\Theta) \phi(\mathbf{x}) + v_0. \end{aligned} \tag{9}$$

Thus showing that $\hat{y}$ is a linear transformation of $\phi(\mathbf{x})$.

The parameter vector $\mathbf{c}_\Theta$ can now be defined as

$$\mathbf{c}_\Theta = (\mathbf{v}^\top \mathbf{A}_\Theta) + v_0 \mathbf{e}_1, \tag{10}$$

where $\mathbf{e}_1 = [1, 0, 0, \ldots, 0]^\top$ is a vector that ensures the constant bias v_0 is included. Now joining the expression (10) with (9) confirms that

$$\hat{y} = \mathbf{c}_\Theta^\top \phi(\mathbf{x}) \Leftrightarrow \hat{y}(\mathbf{x}, \mathbf{c}_\Theta) = \mathbf{c}_\Theta^\top \phi(\mathbf{x}). \tag{11}$$

Hence, $\hat{y}$ is a linear transformation of the feature vector $\phi(\mathbf{x})$ that can be written as (11). The parameter vector $\mathbf{c}_\Theta$ is then determined as

$$\mathbf{c}_\Theta = \mathbf{v}^\top \mathbf{A}_\Theta + v_0 \mathbf{e}_1.$$

This result indicates that the model behaves as a linear model in the transformed feature space $\phi(\mathbf{x})$. However, it is important to note that the dependence of $\mathbf{c}_\Theta$ on the original parameters $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$ is not linear. Specifically, $\mathbf{A}_\Theta$ is constructed from $(\mathbf{W}, \mathbf{b})$ through nonlinear interactions. Consequently, while the model is linear with respect to $\phi(\mathbf{x})$, it is not a linear model in terms of the original parameters.

3

Disclaimer: ChatGPT was used to help the group figure out which steps were needed to produce the desired proof.

Assuming $K \geq D$, it is needed to show that for any real vector $\mathbf{c} \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ and any $\epsilon > 0$ there is a choice of the original parameters $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$ such that $\|\mathbf{c}_\Theta - \mathbf{c}\| < \epsilon$.

Recall from Question 3.1 that $\mathbf{A}_\Theta$ is a matrix parametrized by $\mathbf{W}$ and $\mathbf{b}$, $\phi(\mathbf{x})$ is the mapping (6) and $\mathbf{h}$ is the linear transforation (7). The vector $\mathbf{c}$ corresponds to the "vech" operation on $\mathbf{A}_\Theta$, where the entries of $\mathbf{c}$ are mapped to the lower triangular elements of $\mathbf{A}_\Theta$:

$$\mathbf{c} = \text{vech}(\mathbf{A}_\Theta).$$

The matrix $\mathbf{A}_\Theta$ is assumed to be non-singular. Using orthogonal decomposition:

$$\mathbf{A}_\Theta = \mathbf{Q}\,\text{diag}(\lambda)\,\mathbf{Q}^\top,$$

where $\mathbf{Q}$ is an orthogonal matrix ($\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$) and $\lambda = [\lambda_1, \ldots, \lambda_K]^\top$ are the non-zero eigenvalues of $\mathbf{A}_\Theta$.

Assuming $\mathbf{c}$ is given, then $\mathbf{A}_\Theta$ can directly be constructed by placing the elements of $\mathbf{c}$ into their corresponding positions in the symmetric matrix:

$$\mathbf{A}_\Theta = \text{vech}^{-1}(\mathbf{c}).$$

The diagonal elements of $\mathbf{A}_\Theta$ correspond to quadratic terms involving $\mathbf{b}$, while the off-diagonal elements correspond to cross-terms involving $\mathbf{W}$. From $\mathbf{A}_\Theta$, $\mathbf{b}$ is extracted as:

$$\mathbf{b} = \sqrt{\text{diag}(\mathbf{A}_\Theta)},$$

where $\text{diag}(\mathbf{A}_\Theta)$ extracts the diagonal elements of $\mathbf{A}_\Theta$, and the square root ensures consistency with the quadratic activation.

The off-diagonal elements of $\mathbf{A}_\Theta$ encode pairwise interactions, which can be directly attributed to the outer product structure of $\mathbf{W}$. Specifically, for each pair (i, j) that denotes row and column respectively:

$$(\mathbf{A}_\Theta)_{i,j} = w_{k,i}w_{k,j}, \quad \text{for the } k\text{-th row of } \mathbf{W}.$$

Solving for $\mathbf{W}$ involves decomposing these terms into the corresponding components.

To ensure $\|\mathbf{c}_\Theta - \mathbf{c}\| < \epsilon$, recall that $\mathbf{A}_\Theta$ is non-singular and uniquely determined by $\mathbf{c}$. By directly constructing $\mathbf{A}_\Theta$ from $\mathbf{c}$, there is no loss in precision, and the relationship holds exactly.

Lastly, the parameters $\mathbf{v}$ and v_0 are chosen to satisfy the regression output (8). Given the relationship in (7), $\mathbf{v}$ can be adjusted to match any desired regression output.

Thus, the parametrization with Θ is equivalent to specifying $\mathbf{c}$, completing the proof.

4

Suppose the training data $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$ with $N > \frac{(D+1)(D+2)}{2}$ is given. Assuming the matrix $\mathbf{X} \in \mathbb{R}^{N \times \frac{(D+1)(D+2)}{2}}$ whose rows are the feature vectors $\{\phi(\mathbf{x}_n)\}_{n=1}^N$ has rank $\frac{(D+1)(D+2)}{2}$. By aiming to minimize the squared loss

$$L(\mathbf{c}_\Theta; \mathcal{D}) = \frac{1}{2} \sum_{n=1}^{N} \left(\hat{y}(\mathbf{x}_n; \mathbf{c}_\Theta) - y_n\right)^2,$$

a closed form solution $\hat{\mathbf{c}}_\Theta$ is sought.

Let $\mathbf{x} \in \mathbb{R}^D$ be the input, $\mathbf{W} \in \mathbb{R}^{K \times D}$, $\mathbf{b} \in \mathbb{R}^K$, and $\mathbf{v} \in \mathbb{R}^K$, $v_0 \in \mathbb{R}$ be the parameters. The hidden layer is given by (1), where the activation function is applied element-wise, and the output is $\hat{y} = \mathbf{v}^\top \mathbf{h} + v_0$. Recall from (7) that $\mathbf{h}$ can be represented as a linear function of $\phi(\mathbf{x})$. As a result, there exists a matrix $\mathbf{U} \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}}$ such that $\mathbf{h} = \mathbf{U}\phi(\mathbf{x})$, and therefore the output can be written as

$$\hat{y}(\mathbf{x}; \Theta) = \mathbf{v}^\top \mathbf{h} + v_0 = \mathbf{v}^\top \mathbf{U} \phi(\mathbf{x}) + v_0.$$

Collecting parameters into a vector $\mathbf{c}_\Theta$, it can be observed that the model is linear in the space of the quadratic features $\phi(\mathbf{x})$, even though it originated from a nonlinear network architecture.

Given the dataset $\mathcal{D}$ and the design matrix $\mathbf{X}$, the squared loss is given by:

$$L(\mathbf{c}_\Theta; \mathcal{D}) = \frac{1}{2} \sum_{n=1}^{N} (\hat{y}(\mathbf{x}_n; \mathbf{c}_\Theta) - y_n)^2 = \frac{1}{2} \|\mathbf{X}\mathbf{c}_\Theta - y\|_2^2. \tag{12}$$

If $\mathbf{X}$ has full column rank (which is said that it has), the solution of minimizing the squared loss (12) is obtained by:

$$\hat{\mathbf{c}}_\Theta = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top y.$$

Thus, a closed-form global minimizer for the reparameterized problem is obtained. This is possible as $g(z) = z(1 - z)$ allows the hidden layer to be expressed as a linear function in a quadratic expansion of the inputs, transforming the originally nonlinear problem into a linear regression problem in a higher-dimensional feature space.