DEEC
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
TÉCNICO LISBOA

Master in Electrical and Computer Engineering

Deep Learning Course

1st Homework

2nd Period 2024/2025

Authors:
João Gonçalves – ist199995
jrazevedogoncalves@tecnico.ulisboa.pt
Teresa Nogueira – ist1100029
maria.teresa.ramos.nogueira@tecnico.ulisboa.pt

Department of Electrical and Computer Engineering
Instituto Superior Técnico, Lisbon, Portugal

Team Contribution Disclaimer

The elaboration of this report was a collaborative effort, with both students contributing equally to its completion. However, we attribute Question 1 to Teresa Nogueira, Question 2 to João Gonçalves, and Question 3 was jointly developed by both members.

Question 1 – Linear Classifiers and Neural Networks

In this section, we implement various machine learning models from scratch (without any toolkit) for landscape classification using the Intel Image Classification dataset, consisting of six categories of 48×48 RGB images. Starting with a perceptron algorithm, we compare logistic regression with and without regularization, and a multiple linear regression (MLP) with a single hidden layer.

Question 1.1 - Perceptron Algorithm

A) Performance of perceptron algorithm: The train and validation accuracy as a function of epochs as well as their respective values after 100 epochs are displayed below.

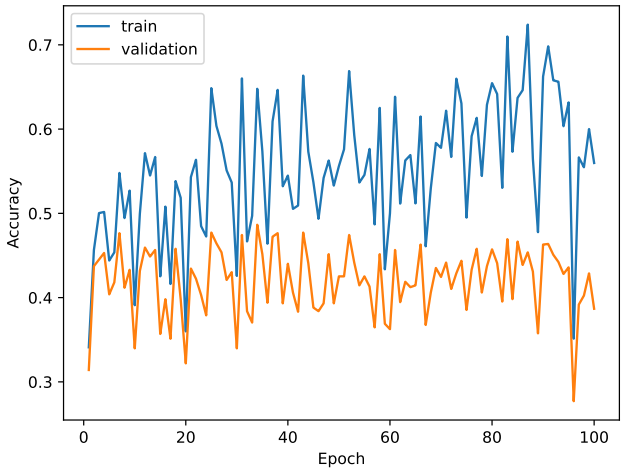


Fig. 1.1. Train and validation accuracy as a function of the epoch number for the perceptron algorithm.

Train Acc	Validation Acc	Test Acc
0.5598	0.3868	0.3743

Tab. 1.1. Percepton algorithm accuracy after 100 epochs.

The perceptron algorithm fails to achieve consistent improvements in both training and validation accuracy over 100 epochs. The erratic fluctuations in training and validation accuracy indicate that the perceptron struggles to capture the underlying relationships in the dataset.

Question 1.2 - Logistic Regression Classifier

A) Logistic Regression without Regularization: For a logistic regression classifier using stochastic gradient descent (SGD), the weight update step is given by

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} + \eta_k \left(\mathbf{e}_y \phi(\mathbf{x})^\top - \sum_{y'} P_W(y'|\mathbf{x}) \mathbf{e}_{y'} \phi(\mathbf{x})^\top \right).$$

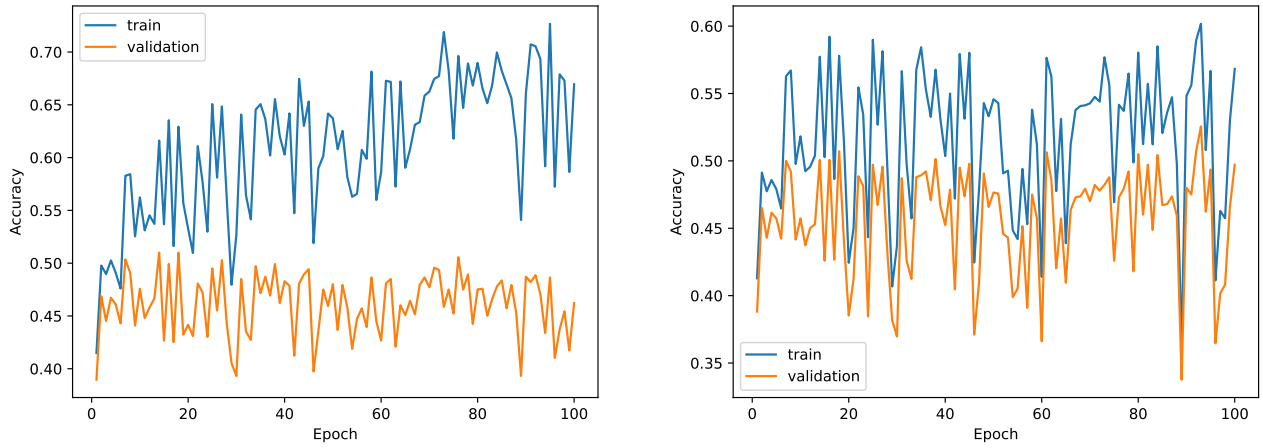
Here, $\mathbf{W} \in \mathbb{R}^{C \times F}$ represent the weight matrix, where C is the number of classes and F is the number of features. The term η_k denotes the learning rate. The vector $\mathbf{e}_y \in \mathbb{R}^C$ is a one-hot encoding of the true class label y, and $\phi(\mathbf{x}) \in \mathbb{R}^F$ represents the feature vector of the current example $\mathbf{x}$. The probability $P_W(y'|\mathbf{x})$ is computed for each class y' using the current weights $\mathbf{W}$ and the softmax function.

B) Logistic Regression with ℓ_2 Penalty: For a logistic regression classifier using SGD and ℓ_2-regularization, the weight update step is given by

$$\mathbf{W}^{(k+1)} = (1 - \eta_k \lambda) \mathbf{W}^{(k)} + \eta_k \left(e_y \phi(\mathbf{x})^\top - \sum_{y'} P_W(y'|\mathbf{x}) \mathbf{e}_{y'} \phi(\mathbf{x})^\top \right),$$

where λ is the regularization coefficient.

The term $(1 - \eta_k \lambda) < 1$ introduces weight decay in each iteration, which penalizes the magnitude of the weights. A model trained using (stochastic) gradient descent without regularization often learns large weights as we can see in Figure 1.3 (a), overfitting in the training data. This overfitting results in higher training accuracy but poor performance on held-out test data as is confirmed in Figure 1.2 (a) and in Table 1.2, where test accuracy is only 0.4597.



(a) No regularization. **(b)** ℓ_2-regularization.

Fig. 1.2. Training and validation accuracy over epochs for two versions of logistic regression.

Regularization	Train Acc	Validation Acc	Test Acc
None	0.6694	0.4623	0.4597
ℓ_2-regularization	0.5683	0.4658	0.5053

Tab. 1.2. Logistic regression accuracy with and without ℓ_2 regularization after 100 epochs.

Despite the application of ℓ_2-regularization, the model's generalization ability does not show noticeable improvement. Now, both training and validation accuracies remain stagnant and erratic, as seen in Figure 1.2 (b) and the train accuracy is also noticibly smaller. This suggests that the logistic regression model, lacks the representational power needed to capture the relationships in the dataset. While regularization mitigates overfitting by penalizing large weights, it cannot overcome the model's inherent limitations. This is further evidenced by the test accuracies for both regularized and non-regularized models, which are nearly identical, with the regularized model offering only a marginal improvement.

C) Impact of ℓ_2 Regularization on Weight Norms: As already explained, regularization effectively limits the growth of the weights through weight decay, but is unable to compensate for the inherent limitations of the model. The early convergence of the weight norm suggests this.

Taking the formula for the update step in a logistic regression classifier using SGD

$$\mathbf{W}^{(k+1)} = (1 - \eta_k \lambda)\mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right)$$

For $\|\mathbf{W}\|_F$ to stabilize, $\mathbf{W}^{(k+1)} - \mathbf{W}^{(k)} \to 0$, therefore

$$\begin{aligned}(1 - \eta_k \lambda)\mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right) - \mathbf{W}^{(k)} &= 0 \\ -\eta_k \lambda \mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right) &= 0\end{aligned}$$

At equilibrium, weight decay balances the loss gradient,

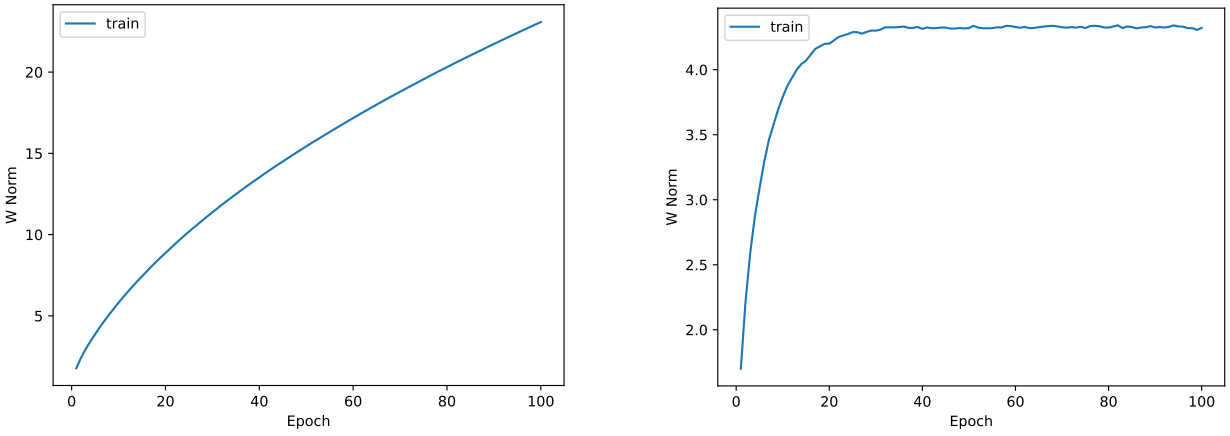
$$\lambda \mathbf{W}^{(k)} = \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right)$$

The model finds a balance where the loss stabilizes. This stabilization indicates that the model has reached its best performance under the current configuration. Further training will not improve the model significantly.

For the non-regularized case ($\lambda = 0$), for $\|\mathbf{W}\|_F$ to stabilize:

$$\begin{aligned}\mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right) - \mathbf{W}^{(k)} &= 0 \\ \eta_k \nabla_{\mathbf{W}} L\left(\mathbf{W}^{(k)}; (x_t, y_t)\right) &= 0\end{aligned}$$

This means that, without regularization, the weights are driven only by the gradient of the loss, the weights will continue to grow indefinitely as the model tries to minimize the loss.



(a) No regularization. **(b)** ℓ_2-regularization.

Fig. 1.3. Comparison of weight norm behavior between two versions of a logistic regression models.

Both figures mirror the discussion above. For the regularized version, the weight norm reaches equilibrium after approximately 40 epochs, with no significant improvement observed afterward. In contrast, the non-regularized version shows the weight norm growing indefinitely throughout training. Additionally it is important to note that the weight norm stabilizes at 5 for the regularized version, while the non-regularized version reaches a weight of 23 at epoch 100 demonstrating how ℓ_2-regularization limits the weight growth.

D) Expected Differences Between ℓ_1 and ℓ_2 Regularization Effects: With ℓ_1-regularization we want to find

$$\arg\min_{\mathbf{W}} \sum_{t=1}^{N} L(\mathbf{W}; (x_t, y_t)) + \lambda \|\mathbf{W}\|_1,$$

where $\|\mathbf{W}\|_1$ is the ℓ_1-norm of the weight matrix,

$$\|\mathbf{W}\|_1 = \sum_y \|\mathbf{w}_y\|_1 = \sum_y \sum_i |w_{y,i}|,$$

which is the sum of the absolute values of all weight components $w_{y,i}$ where y indexes over the classes and i indexes over the features.

The gradient of this norm does not exist at 0, therefore we must rely on the subgradient

$$\frac{\partial |w_{y,i}|}{\partial w_{y,i}} = \begin{cases} +1 & \text{if } w_{y,i} > 0, \\ -1 & \text{if } w_{y,i} < 0, \\ 0 & \text{if } w_{y,i} = 0. \end{cases}$$

Which, in summary, means

$$\frac{\partial \|\mathbf{W}\|_1}{\partial \mathbf{W}} = \operatorname{sgn}(\mathbf{W}).$$

Consequentially, for a logistic regression classifier using SGD and ℓ_1-regularization, the weight update step is given by

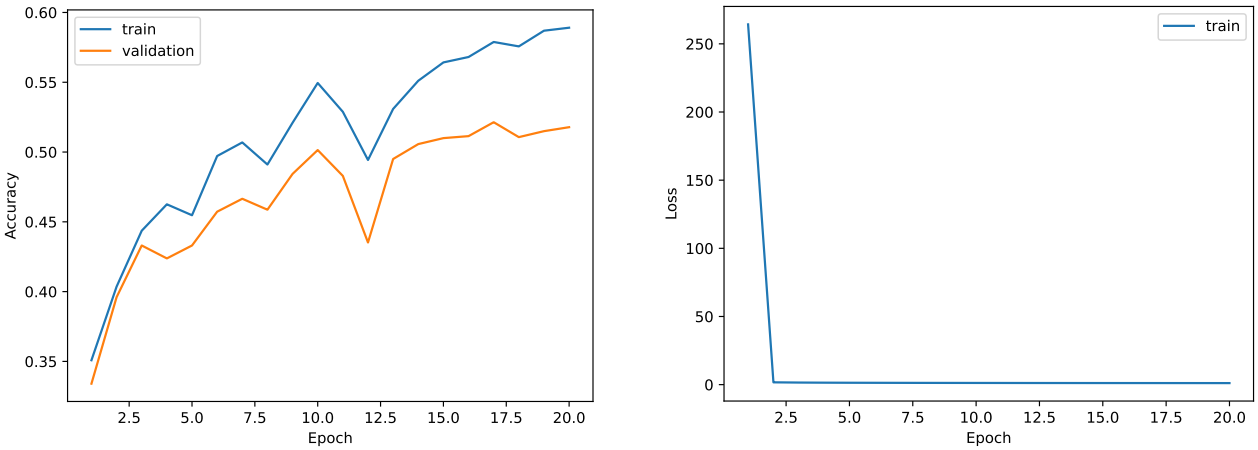
$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta_k \lambda \operatorname{sgn}\left(\mathbf{W}^{(k)}\right) + \eta_k \left(e_y \phi(x)^\top - \sum_{y'} P_W(y'|x) e_{y'} \phi(x)^\top \right).$$

The weight update now includes a shrinkage term proportional to the sign of the weight, rather than the magnitude of the weight itself, as seen with the ℓ_2-norm. This shrinkage force is constant and independent of the weight's magnitude, which is why ℓ_1-regularization can drive some weights exactly to zero, resulting in sparser weight matrixes. While ℓ_2-regularization also moves weights toward zero, it rarely makes them exactly zero since the shrinkage force vanishes as the weight gets smaller.

Both ℓ_1- and ℓ_2-regularization reduce the weight norm, but ℓ_1-regularization creates a simpler model by zeroing out many weights, while ℓ_2-regularization shrinks all weights proportionally, retaining all features.

Question 1.3 - Multi-Layer Perceptron

For this question, we implemented a multi-layer perceptron with a single hidden layer with 100 hidden units and a relu activation function for the hidden layers, and a multinomial logistic loss in the output layer. The obtained results are shown below:



(a) Accuracy. **(b)** Loss.

Fig. 1.4. Train and validation accuracy (left) and loss (right) as a function of the epoch number.

Loss	Train Acc	Validation Acc	Test Acc
1.1279	0.5891	0.5178	0.5320

Tab. 1.3. MLP validation, train accuracy and loss after 20 epochs.

The model's accuracy significantly improves compared to the previous approaches, which means the use of non-linear activation functions in an MLP enables better input representations for accurate label prediction. Initially, the loss is much larger due to the randomly initialized weights but decreases rapidly before stabilizing as training progresses.

Note: To compute softmax and cross-entropy while avoiding numerical instability due to underflow or overflow, the following techniques are used:

The softmax function is calculated as

$$\text{softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}.$$

This formulation ensures numerical stability by subtracting $\max(x)$, preventing the exponential terms from becoming too large. On the other hand, cross-entropy, is given by

$$H(p, q) = -\log(\text{softmax}(\hat{y})_y).$$

where q is the predicted probability vector, computed as $\text{softmax}(\hat{y})$ and p is the one-hot encoded true label vector, assigning 100% probability to the correct class y. Directly computing $\log(\text{softmax}(\hat{y})_y)$ can lead to numerical issues with $-\log(0)$. To address this, we reformulate

$$\log\left(\text{softmax}(x)_i\right) = x_i - \max(\mathbf{x}) - \log\left(\sum_j e^{x_j - \max(\mathbf{x})}\right).$$

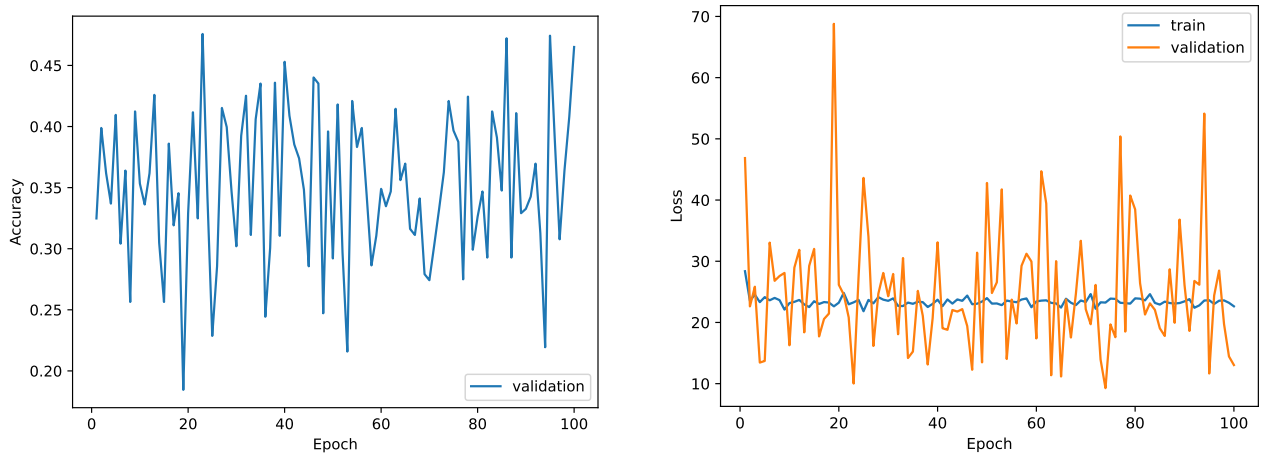
Question 2 – Automatic Differentiation Toolkit

In this section, we implement landscape classification using an automatic differentiation toolkit, building on the manually implemented models from Question 1. We use the deep learning framework `PyTorch` to automate gradient computation and backpropagation.

Question 2.1 – Learning Rate Impact on Accuracy

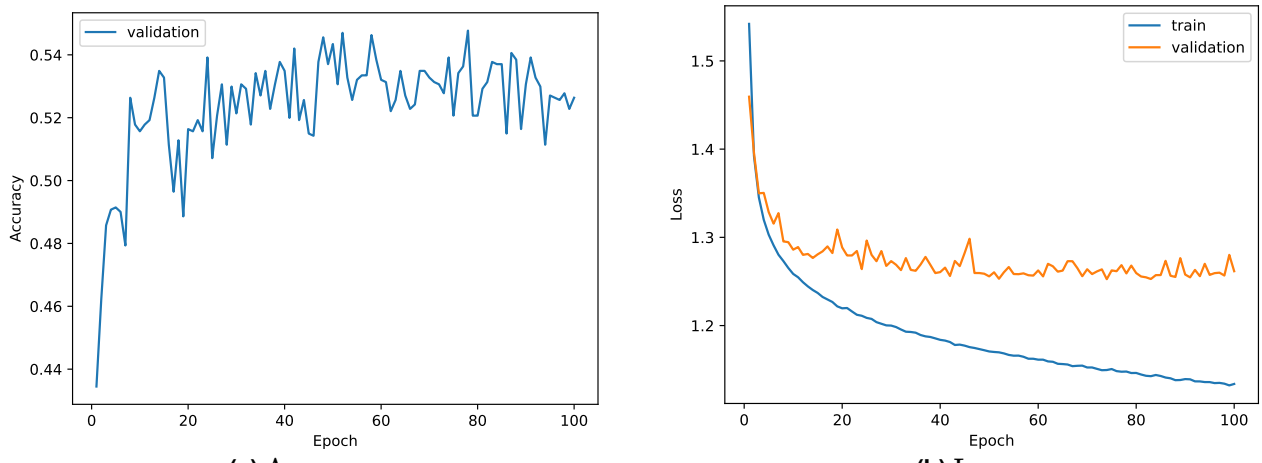
An optimal learning rate should be low enough for accurate convergence while high enough for reasonable training time. Smaller rates require more update steps, potentially yielding better final weights, whereas larger rates can cause large updates which lead to divergent behaviour.

The plots of accuracy and loss for varying values of learning rate in a linear model using logistic regression and stochastic gradient are shown below in Figures 2.1, 2.2 and 2.3. Lastly, Table 2.1 summarizes the results after 100 epochs.



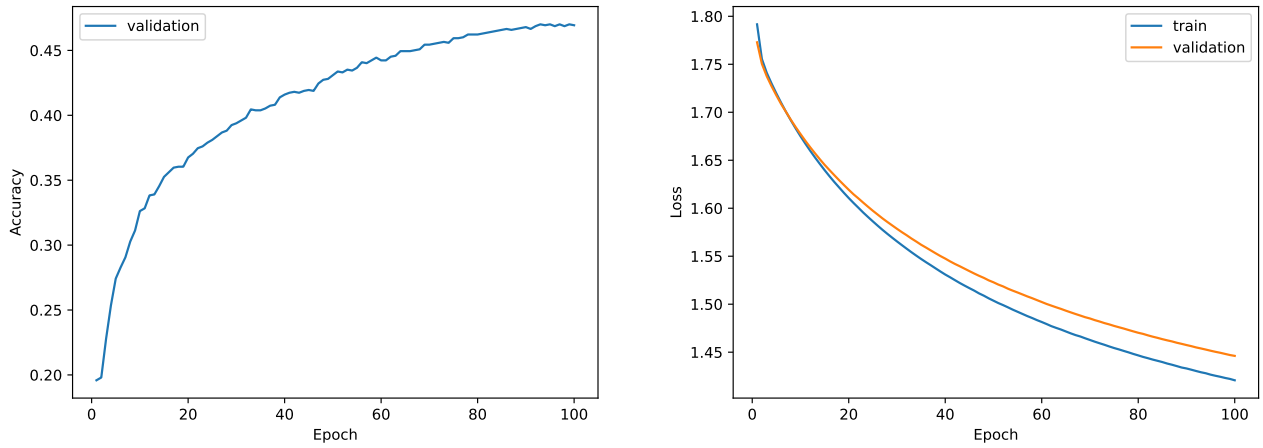
(a) Accuracy. **(b)** Loss.

Fig. 2.1. Validation accuracy (right) and comparison of training and validation loss (right) across epochs for a learning rate of 0.1.



(a) Accuracy. **(b)** Loss.

Fig. 2.2. Validation accuracy (right) and comparison of training and validation loss (right) across epochs for a learning rate of 0.001.



(a) Accuracy. **(b)** Loss.

Fig. 2.3. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for a learning rate of 0.00001.

Learning Rate	Train Loss	Validation Loss	Validation Acc	Test Acc
0.1	22.6370	13.0418	0.4651	0.4707
0.001	1.1338	1.2616	0.5264	0.5247
0.00001	1.4207	1.4462	0.4694	0.4623

Tab. 2.1. Linear model using logistic regression accuracy and loss after 100 epochs.

The learning rate that achieved the highest validation accuracy was 0.001.

As for the evolution of accuracy and loss with respect to learning rate, it follows the expected pattern: as the learning rate decreases, the plots become progressively smoother with reduced erratic behavior.

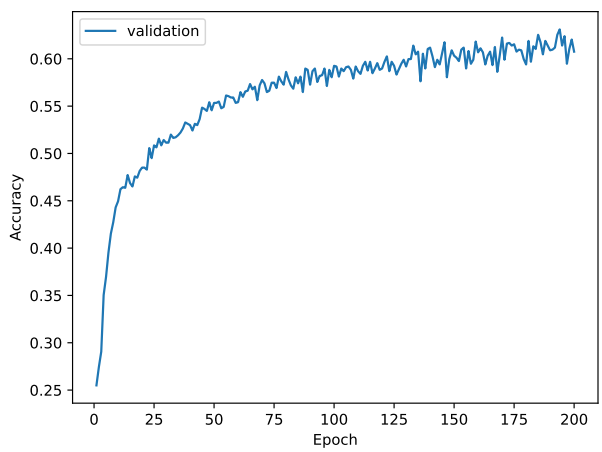
- At the highest learning rate of 0.1, there is no discernible trend in either accuracy or loss, indicating divergence and poor performance, as reflected in the test accuracy, which is only 0.4707 and the very large loss, 22.6370.
- The learning rate of 0.001 shows noticeable improvement, with clear trends in both plots and only slight jitter, achieving the best performance overall with a test accuracy of 0.5247 and a loss of 1.1338.
- Finally, while the smallest learning rate of 0.00001 produces smooth, well-defined curves with minimal jitter, it is far too slow. This results in very small updates to the model, requiring significantly more iterations to reach the performance achieved by the previous learning rate. It's test accuracy is marginally worse than that of learning rate 0.1, at 0.4623 and its loss is 1.4207 which is slightly larger than of learning rate 0.001.

Question 2.2 – Feed-Forward Neural Network

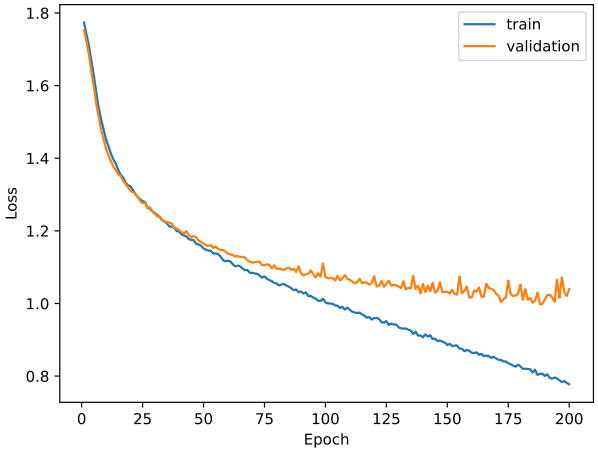
For this section we implemented a feed-forward neural network while varying it's parameter, which will be discussed below.

A) Batch Size Impact on Performance and Time of Execution: Batch size significantly impacts the training dynamics and generalization of machine learning models. Smaller batch sizes result in noisier gradient updates, which can help escape local minima but lead to more unstable optimization paths. Larger batch sizes, while providing more stable gradients, may degrade the model's generalization performance due to overfitting to the training data.

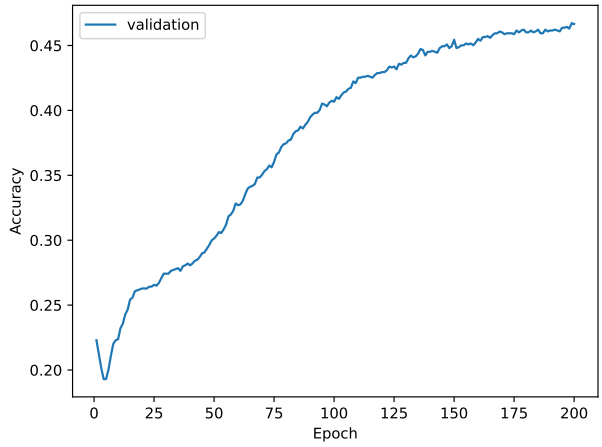
The plots of accuracy and loss for varying values of batch size are shown below in Figure 2.4. Lastly, Table 2.2 summarizes the results after 200 epochs.



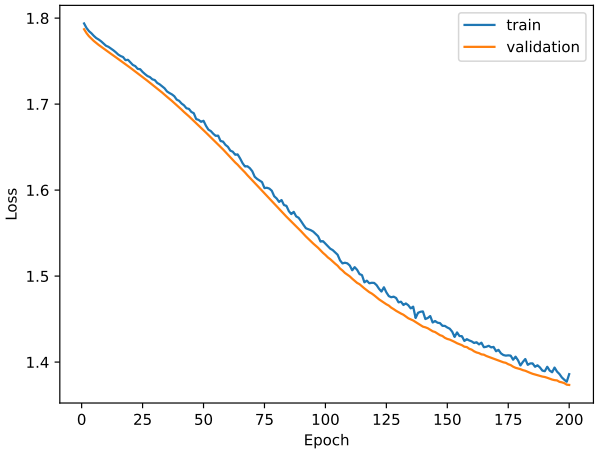
(a) Accuracy for a batch size of 64.



(b) Loss for a batch size of 64.



(c) Accuracy for a batch size of 512.



(d) Loss for a batch size of 512.

Fig. 2.4. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for two different batch size settings in MLP.

Batch Size	Train Loss	Validation Loss	Validation Acc	Test Acc
64	0.7776	1.0399	0.6075	0.5963
512	1.2704	1.2718	0.5021	0.5147

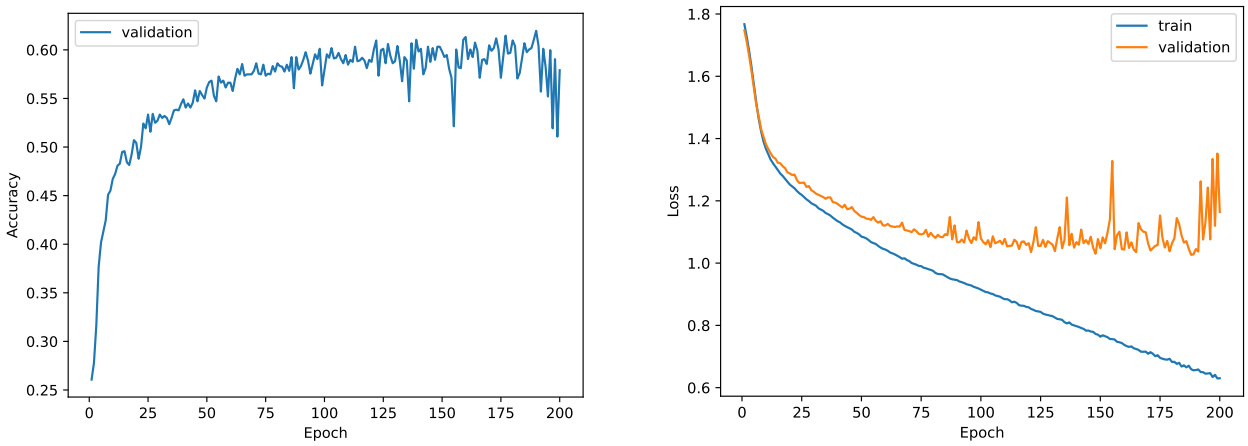
Tab. 2.2. MLP accuracy and loss after 200 epochs for different values of batch size.

With the increase of the batch size, the results align with the expected patterns: larger batch sizes improve computational efficiency but often lead to reduced generalization.

- For the batch size of 64 the model performs more updates per epoch, which slows down computation (in our case 7 minutes and 58 seconds) but results in better convergence and generalization. This happens because the training process benefits from noisier gradients, which prevents overfitting.
- For larger batch sizes like 512, the model performs fewer updates per epoch, resulting in faster computation (in or case 4 minutes and 43 seconds) but lower overall performance for the same number of epochs since the model converges slower.

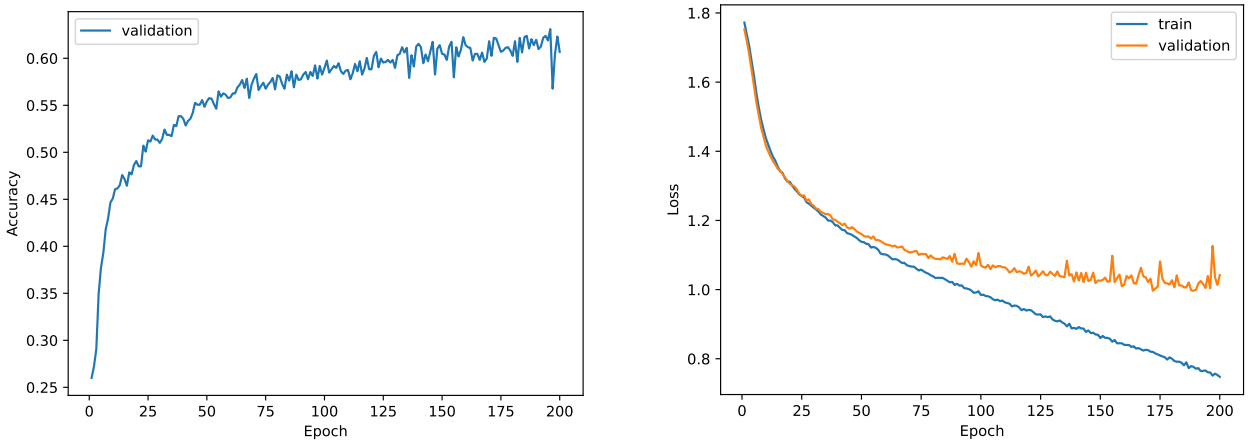
B) Dropout Impact on Accuracy and Loss: Dropout is a regularization technique. Dropout works by probabilistically removing, or *dropping out*, inputs to a layer which in turn decreases the flexibility of the model which can prevent overfitting and improve performance.

The plots of accuracy and loss for varying values of dropout are shown below in Figures 2.5, 2.6 and 2.7. Lastly, Table 2.3 summarizes the results after 200 epochs.



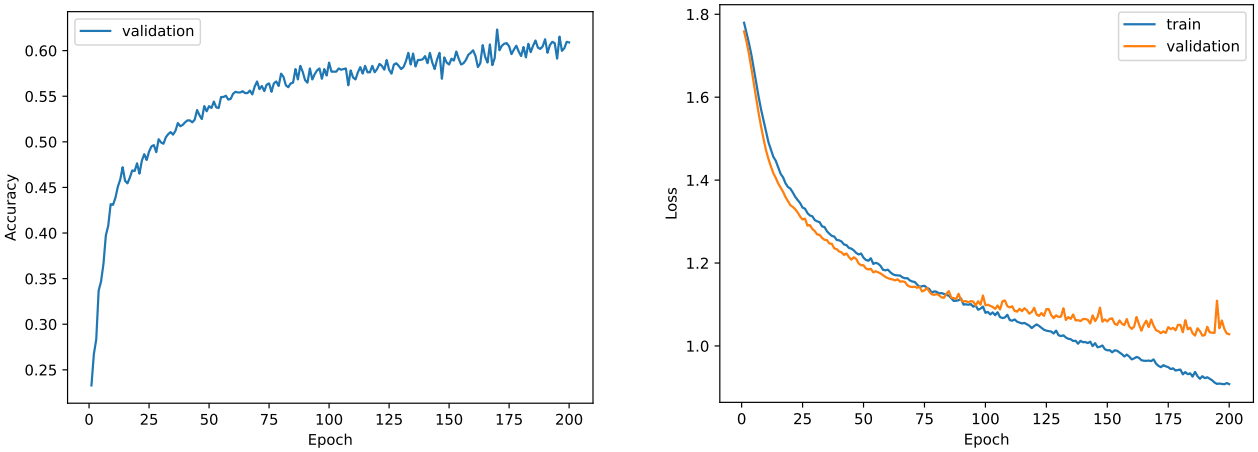
(a) Accuracy for a dropout of 0.01. **(b)** Loss for a dropout of 0.01.

Fig. 2.5. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for a dropout of 0.01 in MLP.



(a) Accuracy. **(b)** Loss.

Fig. 2.6. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for a dropout of 0.25 in MLP.



(a) Accuracy. **(b)** Loss.

Fig. 2.7. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for a dropout of 0.50 in MLP.

Droput	Train Loss	Validation Loss	Validation Acc	Test Acc
0.01	0.6302	1.1641	0.5791	0.5783
0.25	0.7477	1.0422	0.6068	0.6007
0.50	0.9079	1.0283	0.6090	0.6027

Tab. 2.3. MLP accuracy and loss after 200 epochs for different values of dropout.

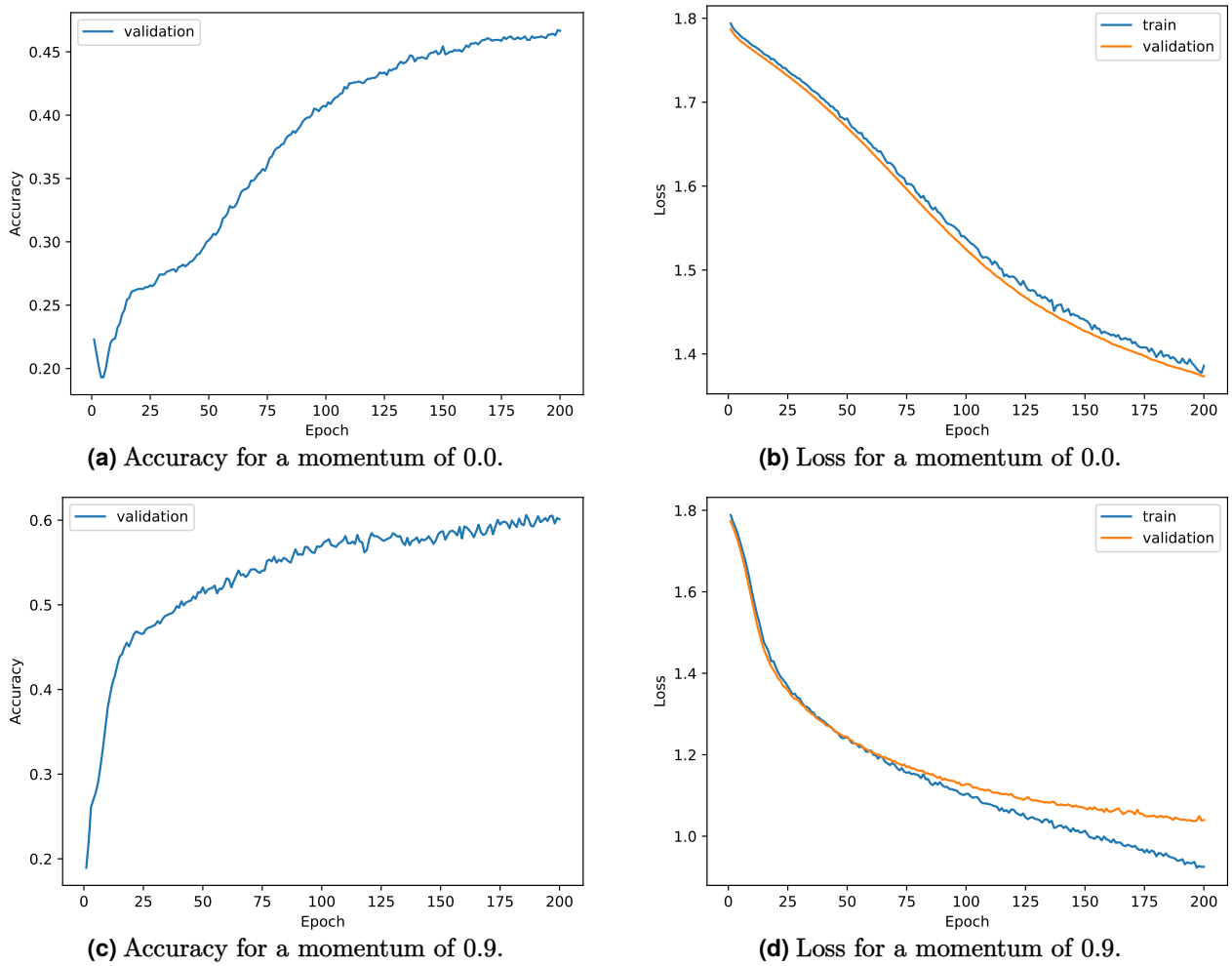
As seen in the table, increasing the dropout rate leads to a higher training loss but results in better generalization, as indicated by the improved validation and test accuracies. The dropout that achieved the highest validation and test accuracy was 0.5.

- At a dropout rate of 0.01, the training loss is the lowest at 0.6302, but the validation accuracy is 0.5791 and test accuracy is 0.5783, which are suboptimal.
- As dropout increases to 0.25, the model generalizes better with a slight increase in training loss (0.7477) but higher validation accuracy (0.6068) and test accuracy (0.6007).
- Finally, at a dropout of 0.50, while the training loss increases further to 0.9079, the validation and test accuracies continue to improve slightly to 0.6090 and 0.6027, respectively.

This behavior aligns with the expected outcome when using dropout: By changing the network itself, it acts as a regularizer (or generalizer) that prevents overfitting by making the network more robust. Training loss increases because fewer neurons contribute during training, but the model performs better on unseen data.

C) Momentum Impact on Accuracy and Loss: Momentum helps accelerate SGD, especially in regions where the gradients fluctuate, by smoothing the updates and avoiding sharp directional changes. It is designed to help the optimizer *remember* the previous gradients and prevent it from getting stuck in poor local minima or oscillating too much.

The plots of accuracy and loss for varying values of momentum are shown below in Figure 2.8. Lastly, Table 2.4 summarizes the results after 200 epochs.



(a) Accuracy for a momentum of 0.0.

(b) Loss for a momentum of 0.0.

(c) Accuracy for a momentum of 0.9.

(d) Loss for a momentum of 0.9.

Fig. 2.8. Validation accuracy (left) and comparison of training and validation loss (right) across epochs for three different momentum settings in MLP: (a, b) 0.0, (c, d) 0.9.

Momentum	Train Loss	Validation Loss	Validation Acc	Test Acc
0.0	1.3860	1.3734	0.4665	0.4850
0.9	0.9250	1.0398	0.6011	0.6053

Tab. 2.4. MLP accuracy and loss after 200 epochs for different values of momentum.

As for the evolution of accuracy and loss with respect to momentum, the results follow the expected pattern: Momentum leads to faster convergence (lower loss in fewer epochs) and improved generalization.

- At a momentum value of 0.0, the model converges gradually, with a steady increase in validation accuracy, reaching approximately 0.4665 . The loss decreases consistently for both training and validation sets, demonstrating a slower convergence compared to the example with momentum.
- At a momentum value of 0.9, the model benefits from accelerated convergence (the trends are much steeper than in the first case, both in accuracy and in loss) in the early epochs. This acceleration results in a higher final validation and test accuracy, 0.6011 and 0.6053, respectively. The training loss decreases faster, while the validation loss shows a slower decline toward convergence, suggesting that momentum helps the optimizer move more efficiently through the loss surface.

Momentum accelerates training by using the accumulated gradients from previous steps, allowing for faster convergence while maintaining higher accuracy. Instead of relying solely on the gradient at the current step θ_n, the momentum trick sums up gradients from all previous steps $\theta_{n-1}, \theta_{n-2}, \ldots$ with exponentially decreasing contributions.

Additionally, for large batch sizes (1024), where gradients are inherently more stable but the overall performance is worse for a small amount of epochs as discussed before, the momentum efficiently helps the optimizer move towards a better minimum faster.

Question 3 – Multi-Layer Perceptron

In this section, we examine a single-hidden-layer neural network designed for univariate regression, where the activation function is specified as $g(z) = z(1-z)$. The predicted output of the network, $\hat{y} \in \mathbb{R}$, is given by $\hat{y} = \mathbf{v}^\top \mathbf{h} + v_0$, with the hidden layer activations $\mathbf{h} \in \mathbb{R}^K$ defined as $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$. Here, $\mathbf{x} \in \mathbb{R}^D$ represents the input, and the model parameters are collectively denoted by $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$, where $\mathbf{W} \in \mathbb{R}^{K \times D}$, $\mathbf{b} \in \mathbb{R}^K$, $\mathbf{v} \in \mathbb{R}^K$, and $v_0 \in \mathbb{R}$.

The analysis centers on demonstrating that the specific choice of activation function $g(z)$ enables a reparameterization of the model, simplifying it to a linear formulation under some assumptions.

Question 3.1 – Feature Mapping and Hidden Layer Linear Transformation

Our goal is to show that the layer output $\mathbf{h} \in \mathbb{R}^K$ can be written as a linear transformation of a certain feature mapping, i.e.,

$$\mathbf{h} = \mathbf{A}_\Theta \phi(\mathbf{x}), \text{ with } \begin{cases} \phi : \mathbb{R}^D \to \mathbb{R}^{\frac{(D+1)(D+2)}{2}} \\ \mathbf{A}_\Theta \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}} \end{cases}.$$

Here, ϕ is the feature mapping, and $\mathbf{A}_\Theta$ is a matrix determined by the model parameters Θ. To achieve this, we explicitly define the feature mapping ϕ and derive the corresponding matrix $\mathbf{A}_\Theta$. Starting from the definition of the hidden layer output

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

where $\mathbf{W} \in \mathbb{R}^{K \times D}$ is the weight matrix, $\mathbf{b} \in \mathbb{R}^K$ is the bias vector, and g is the activation function applied elementwise, we analyze the components of each neuron $k \in \{1, \ldots, K\}$:

- The pre-activation value is given by $z_k = \mathbf{w}_k^\top \mathbf{x} + b_k$, where $\mathbf{w}_k^\top$ is the k-th row of matrix $\mathbf{W}$;
- Applying the activation function g to the pre-activation value, we can expand output h_k as

$$\begin{aligned} h_k = g(z_k) &= (\mathbf{w}_k^\top \mathbf{x} + b_k)(1 - (\mathbf{w}_k^\top \mathbf{x} + b_k)) \\ &= b_k - b_k^2 + (1 - 2b_k)(\mathbf{w}_k^\top \mathbf{x}) - (\mathbf{x}^\top \mathbf{w}_k)(\mathbf{w}_k^\top \mathbf{x}); \end{aligned}$$

- Thus, we can readily see that h_k may be written as a linear transformation, i.e.,

$$h_k = \mathbf{a}_k^\top \phi(\mathbf{x}),$$

  if we define the mapping of the input $\mathbf{x} \in \mathbb{R}^D$ as

$$\phi(\mathbf{x}) = [1, x_1, x_2, \ldots, x_D, x_1^2, x_1 x_2, \ldots, x_D^2]^\top,$$

  and the parameter vector

$$\mathbf{a}_k = [b_k - b_k^2, (1 - 2b_k)w_{k,1}, (1 - 2b_k)w_{k,2}, \ldots, (1 - 2b_k)w_{k,D}, -w_{k,1}^2, -w_{k,1}w_{k,2}, \ldots, -w_{k,D}^2]^\top.$$

Stacking the outputs h_k for all $k \in \{1, \ldots, K\}$, we obtain the full hidden layer output

$$\mathbf{h} = [h_1, \ldots, h_K]^\top = \mathbf{A}_\Theta \phi(\mathbf{x}),$$

where $\mathbf{A}_\Theta$ is constructed by stacking the row vectors $\mathbf{a}_k^\top$ (deduced above).

Note that the feature mapping $\phi(\mathbf{x})$ is independent of the model parameters, making it a universal representation for any input $\mathbf{x} \in \mathbb{R}^D$.

We also add that

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

which makes

$$\dim(\mathbf{A}_\Theta) = K \times \frac{(D+1)(D+2)}{2}.$$

Question 3.2 – Linear Model Representation of the Output

Based on the previous results, we know that the hidden layer output $\mathbf{h} \in \mathbb{R}^K$ can be written as

$$\mathbf{h} = \mathbf{A}_\Theta \phi(\mathbf{x}),$$

where $\phi : \mathbb{R}^D \to \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ is the feature mapping of the input $\mathbf{x} \in \mathbb{R}^D$, and $\mathbf{A}_\Theta \in \mathbb{R}^{K \times \frac{(D+1)(D+2)}{2}}$ is the transformation matrix determined by the model parameters.

The predicted output of the model, $\hat{y} \in \mathbb{R}$, is given by

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

where $\mathbf{v} \in \mathbb{R}^K$ is the weight vector connecting the hidden layer to the output, and $v_0 \in \mathbb{R}$ is the output bias. Substituting $\mathbf{h} = \mathbf{A}_\Theta \phi(\mathbf{x})$ into the expression for $\hat{y}$, we get

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

The product $\mathbf{v}^\top \mathbf{A}_\Theta$ results in a row vector with $\frac{(D+1)(D+2)}{2}$ elements, which we'll denote as $\mathbf{k}_\Theta^\top$.

Next, we note that the feature mapping $\phi(\mathbf{x})$ includes a constant feature $\phi_1(\mathbf{x}) = 1$ in its first position. This allows the bias v_0 to be absorbed into the first element of $\mathbf{k}_\Theta^\top$. We define

$$\mathbf{c}_\Theta^\top = \mathbf{k}_\Theta^\top + [v_0, 0, \ldots, 0],$$

where v_0 is added to the first element of $\mathbf{k}_\Theta^\top$, and the remaining elements remain unchanged. Substituting this into the expression for $\hat{y}$, we have

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

where $\mathbf{c}_\Theta \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ combines the effects of $\mathbf{v}^\top \mathbf{A}_\Theta$ and the bias v_0. This formulation demonstrates that the output of the model $\hat{y}$ is a linear function of the feature mapping $\phi(\mathbf{x})$, with the parameter vector $\mathbf{c}_\Theta$ encapsulating both the transformation and the bias.

It is important to note that while the model $\hat{y} = \mathbf{c}_\Theta^\top \phi(\mathbf{x})$ is linear in terms of the feature mapping $\phi(\mathbf{x})$, it is not linear with respect to the original parameters $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$. This is because the parameter vector $\mathbf{c}_\Theta$ depends on the original parameters in a nonlinear way. Specifically, $\mathbf{c}_\Theta$ is derived from $\mathbf{v}^\top \mathbf{A}_\Theta + [v_0, 0, \ldots, 0]$, where $\mathbf{A}_\Theta$ itself involves quadratic and interaction terms of the weights $\mathbf{W}$ and biases $\mathbf{b}$ through the hidden layer activations. Consequently, while $\hat{y}$ is a linear function of the transformed input $\phi(\mathbf{x})$, the overall model remains nonlinear in terms of the parameters Θ.

Question 3.3 – Parameterization via Orthogonal Decomposition

We now show that for any real vector $\mathbf{c} \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ and any tolerance $\varepsilon > 0$, there exists a choice of network parameters $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$ such that the induced parameter vector $\mathbf{c}_\Theta$ satisfies

$$\|\mathbf{c}_\Theta - \mathbf{c}\| < \varepsilon.$$

The vector $\mathbf{c}$ represents the target coefficients for the constant, linear, and quadratic components of the feature mapping $\phi(\mathbf{x})$. From earlier results, the induced parameter vector $\mathbf{c}_\Theta$ corresponds to the decomposition of the predicted output

$$\mathbf{c}_\Theta^\top \phi(\mathbf{x}) = v_0 + \sum_{k=1}^{K} v_k(b_k - b_k^2) + \left(\sum_{k=1}^{K} v_k(1 - 2b_k)\mathbf{w}_k \right)^\top \mathbf{x} - \mathbf{x}^\top \left(\sum_{k=1}^{K} v_k \mathbf{w}_k \mathbf{w}_k^\top \right) \mathbf{x}.$$

Here, the constant term is $v_0 + \sum_{k=1}^{K} v_k(b_k - b_k^2)$, the linear coefficients are encoded by $\mathbf{W}^\top \operatorname{diag}(\mathbf{v})(1 - 2\mathbf{b})$, and the quadratic part corresponds to the symmetric matrix $\mathbf{C} = \mathbf{W}^\top \operatorname{diag}(\mathbf{v})\mathbf{W}$.

To match $\mathbf{c}$, we construct the network parameters as follows. We start with the quadratic part $\mathbf{C}$, which encodes the second-order terms. By the orthogonal decomposition theorem, any symmetric matrix $\mathbf{C} \in \mathbb{R}^{D \times D}$ can be written as

$$\mathbf{C} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top,$$

where $\mathbf{Q} \in \mathbb{R}^{D \times D}$ is an orthonormal matrix and $\mathbf{\Lambda} \in \mathbb{R}^{D \times D}$ is diagonal with eigenvalues $\lambda_1, \ldots, \lambda_D$. Since $K \geqslant D$, we can define the weight matrix $\mathbf{W}$ and vector $\mathbf{v}$ as

$$\mathbf{W} = \begin{bmatrix} \mathbf{Q}^\top \\ \mathbf{0}_{(K-D)\times D} \end{bmatrix}, \quad \mathbf{v} = \left[\lambda_1, \lambda_2, \ldots, \lambda_D, 0, \ldots, 0\right]^\top.$$

This construction ensures that the symmetric matrix $\mathbf{C} = \mathbf{W}^\top \operatorname{diag}(\mathbf{v})\mathbf{W}$ matches the quadratic coefficients in $\mathbf{c}$.

If $\mathbf{C}$ is singular or rank-deficient, we perturb it by adding a small diagonal matrix $\delta\mathbf{I}$, where $\delta > 0$ is chosen such that

$$\|\mathbf{C} + \delta\mathbf{I} - \mathbf{C}\| < \varepsilon.$$

This ensures that $\mathbf{C} + \delta\mathbf{I}$ becomes non-singular and the approximation error remains within the prescribed tolerance ε. Under this perturbation, the modified quadratic part still satisfies the required accuracy.

Next, we match the linear coefficients $\mathbf{c}_{\text{lin}}$. Given that the linear part of $\mathbf{c}_\Theta$ is expressed as

$$\mathbf{c}_{\text{lin}} = \mathbf{W}^\top \operatorname{diag}(\mathbf{v})(1 - 2\mathbf{b}),$$

we solve for the bias vector $\mathbf{b}$ such that the equation holds. Since $\mathbf{W}^\top \operatorname{diag}(\mathbf{v})$ is full rank under $K \geqslant D$, a solution for $\mathbf{b}$ always exists.

Finally, we match the constant term c_{const} by setting

$$v_0 = c_{\text{const}} - \sum_{k=1}^{K} v_k(b_k - b_k^2),$$

where c_{const} is the first entry of $\mathbf{c}$.

Combining these steps, we have constructed parameters $\Theta = (\mathbf{W}, \mathbf{b}, \mathbf{v}, v_0)$ such that the induced vector $\mathbf{c}_\Theta$ satisfies:

$$\|\mathbf{c}_\Theta - \mathbf{c}\| < \varepsilon.$$

Question 3.4 – Closed-Form Solution and Global Optimization

Given the training data $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$, where $N > \frac{(D+1)(D+2)}{2}$, we aim to minimize the squared loss function

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

Substituting the expression for the predicted output $\hat{y} = \mathbf{c}_\Theta^\top \phi(\mathbf{x})$, where $\phi(\mathbf{x}) \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ is the feature mapping of the input, we can rewrite the loss as

$$L(\mathbf{c}_\Theta; \mathcal{D}) = \frac{1}{2} \|\mathbf{X}\mathbf{c}_\Theta - \mathbf{y}\|_2^2,$$

where $\mathbf{X} \in \mathbb{R}^{N \times \frac{(D+1)(D+2)}{2}}$ is the feature matrix whose rows are the feature vectors $\phi(\mathbf{x}_n)^\top$, and $\mathbf{y} \in \mathbb{R}^N$ is the vector of targets y_n.

To find the optimal solution $\hat{\mathbf{c}}_\Theta$ that minimizes the loss, we compute the gradient of L with respect to $\mathbf{c}_\Theta$ and set it to zero

$$\nabla_{\mathbf{c}_\Theta} L(\mathbf{c}_\Theta; \mathcal{D}) = \mathbf{X}^\top (\mathbf{X}\mathbf{c}_\Theta - \mathbf{y}) = 0.$$

Rearranging the equation, we obtain the closed-form solution for the optimal weights

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

The invertibility of the Gram matrix $\mathbf{X}^\top \mathbf{X}$ relies on the assumption that $\mathbf{X}$ has full column rank and that $N > \frac{(D+1)(D+2)}{2}$. Specifically, $\mathbf{X}$ is said to have full column rank if its columns are linearly independent, which ensures that the null space of $\mathbf{X}$,

$$\mathcal{N}(\mathbf{X}) = \{\mathbf{u} \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}} \mid \mathbf{X}\mathbf{u} = 0\},$$

contains only the zero vector. To confirm that $\mathbf{X}^\top \mathbf{X}$ is invertible, we proceed by contradiction. Suppose there exists a nonzero vector $\mathbf{u}$ such that

$$\mathbf{X}^\top \mathbf{X}\mathbf{u} = 0.$$

Multiplying both sides by $\mathbf{u}^\top$, we find

$$\mathbf{u}^\top \mathbf{X}^\top \mathbf{X}\mathbf{u} = \|\mathbf{X}\mathbf{u}\|_2^2 = 0.$$

Since the squared norm $\|\mathbf{X}\mathbf{u}\|_2^2$ is zero, it follows that $\mathbf{X}\mathbf{u} = 0$. However, the assumption that $\mathbf{X}$ has full column rank implies that $\mathcal{N}(\mathbf{X}) = \{0\}$. Therefore, $\mathbf{u} = 0$, contradicting the assumption that $\mathbf{u}$ is nonzero.

This establishes that the null space of $\mathbf{X}^\top \mathbf{X}$ is trivial, and thus $\mathbf{X}^\top \mathbf{X}$ is positive definite and invertible. The condition $N > \frac{(D+1)(D+2)}{2}$ ensures that the number of data points exceeds the dimension of the feature space, which means we have enough data to uniquely determine all parameters, guaranteeing that $\mathbf{X}$ cannot be rank-deficient. [1]

What makes this problem special is the structure of the network and the choice of the activation function $g(z) = z(1 - z)$. Unlike general feed-forward neural networks, where the loss function is non-convex due to nonlinearities in the parameters, this particular architecture allows the hidden layer output $\mathbf{h}$ to be written as a linear transformation of the feature mapping $\phi(\mathbf{x})$. As a result, the predicted output $\hat{y}$ becomes a linear function of $\phi(\mathbf{x})$, and the squared loss function is convex. This convexity ensures that the minimization of the loss reduces to a standard linear least-squares problem, which admits a unique global minimizer given by the closed-form solution above.

References

[1] G. Strang, *Linear Algebra and its Applications*, 4th ed. Thomson, Brooks/Cole, 2006.
[2] N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," 2017.
[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.