Aprendizagem Profunda

Homework 1

INSTITUTO SUPERIOR TÉCNICO

Ricardo de Jesus Vicente Tavares, 113368

QUESTION 1

1. a) Perceptron

```
Training took 0 minutes and 47 seconds
Final test acc: 0.3743
```

Fig. 1. Final test accuracy for Perceptron

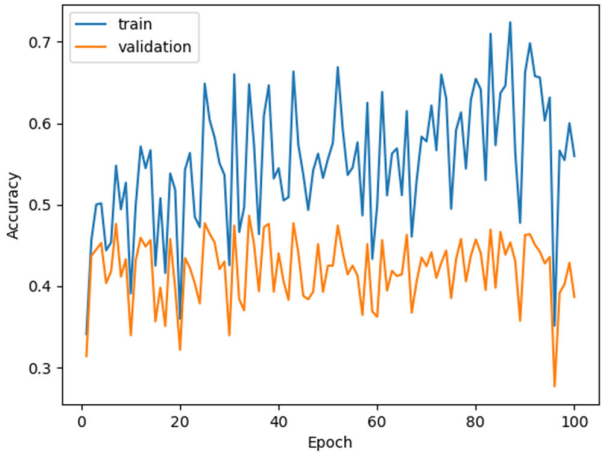

Fig. 2. Accuracy for train and validation set for Perceptron

2. a) Logistic regression classifier with SGD

```
Training took 1 minutes and 53 seconds
Final test acc: 0.4597
```

Fig. 3. Final test accuracy for LRC with SGD

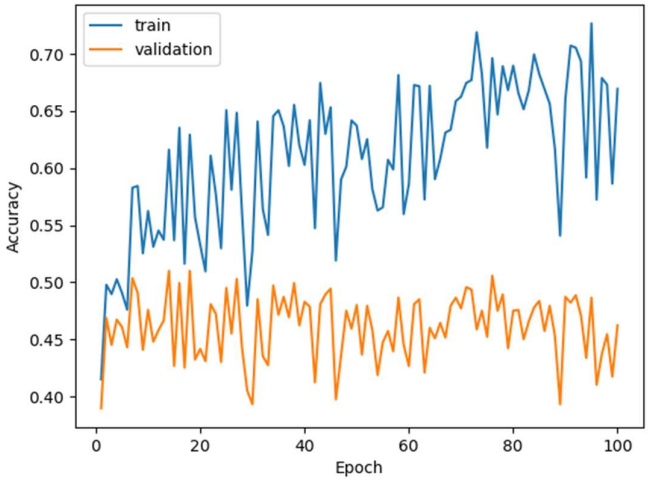

Fig. 4. Accuracy for train and validation set for LRC with SGD (learning rate = 0.001)

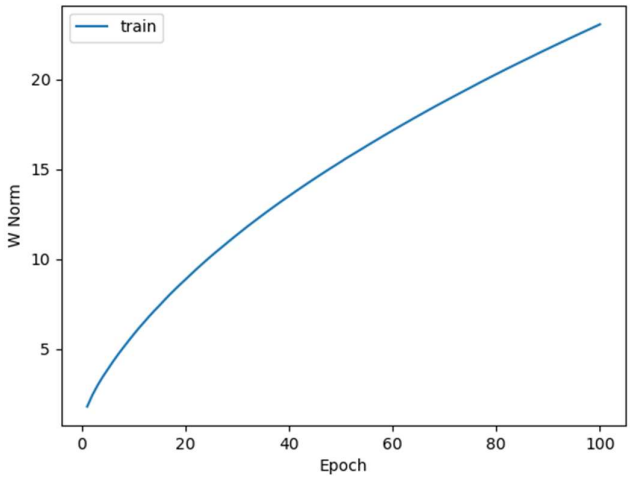

Fig. 5. W norm for LRC with SGD (learning rate = 0.001)

2. b) Logistic regression classifier with SGD and L2 regularization

```
Training took 2 minutes and 27 seconds
Final test acc: 0.4663
```

Fig. 6. Final test accuracy for LRC with SGD and L2 regularization

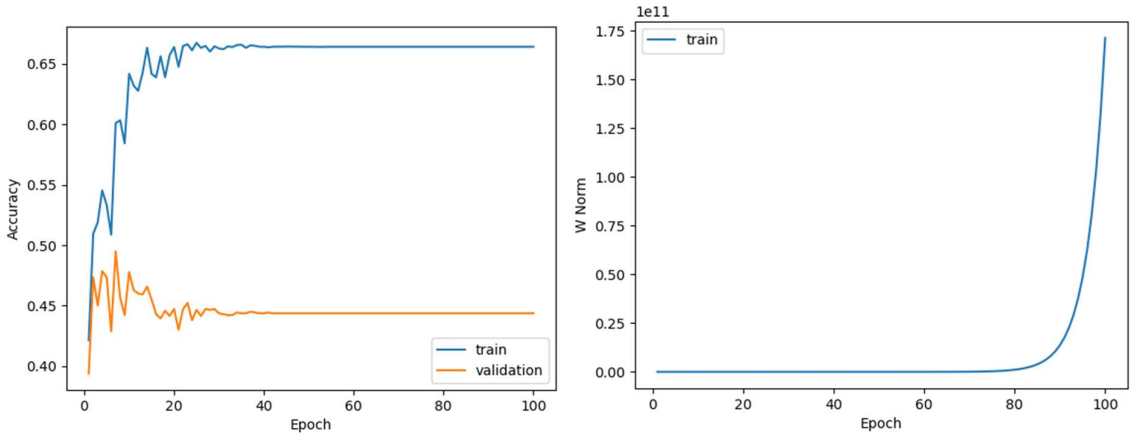

Fig. 7 and 8. Accuracy for train and validation set, and W norm for LRC with SGD (learning rate = 0.001 and $\lambda^{L2} = 0.01$)

The L2 regularization did not improve the end model's accuracy much (Fig. 3 and 6). However, regarding the accuracy of the train and validation set, a faster stabilization of the scores was observed, which happened around epoch 30 (Fig. 7).

The regularization can help the model converge more quickly to a stable solution, as it prevents the weights from becoming too large and the model from excessively 'drifting' during training.

This occurs because L2 regularization helps avoid large updates to the weights, which could lead to fluctuations in the results. As a result, the model can converge more smoothly, avoiding spikes and valleys in the optimization process (Fig. 4), which leads to a faster stabilization of metrics such as accuracy.

2. c) Logistic regression classifier: W norm behaviour

Based on the provided data, we observe the following behaviour for the weight norm:

1. **Without L2 regularization**: The weight norm increases smoothly until it reaches 22.5 around epoch 100. This indicates that, without regularization, the weights gradually increase as the model continues to fit the data, with no restriction on the size of the weights. This is expected in a logistic regression model without regularization, as the model tries to fit the data as much as possible, which can cause the weights to grow steadily.

2. **With L2 regularization**: The weight norm remains low and stable until epoch 80, at which point it begins to rise exponentially, reaching values as high as 100 million. The initial behaviour of low and stable weight norms is due to L2 regularization, which penalizes large weight values, preventing them from growing uncontrollably and helping the model keep the weights small, favouring generalization. However, the exponential growth of the weight norm from epoch 80 onward suggests that, although L2 regularization controlled the growth of the weights up until that point, it was unable to prevent uncontrolled growth after a certain number of epochs. This can be explained by a few factors:

1. *Too high learning rate*: The learning rate of 0.001 might be too high for λ=0.01, which could have caused the model to struggle to find a stable balance between minimizing the cost function and applying regularization. Over time, this could have led to very rapid growth in the weights, especially from epoch 80 onward.

2. *Inadequate λ value*: The regularization parameter λ=0.01 might be too small for effective regularization. When λ is low, the penalty on the weights is not strong enough to control their growth after a certain point in training. This could explain the exponential growth of the weights after epoch 80.

In summary, while L2 regularization helps keep the weights controlled in the early epochs, its effectiveness diminishes as training progresses, especially when the learning rate is too high or the λ parameter is not appropriate, leading to exponential growth in the weight norm.

2. d) Logistic regression classifier with L1 regularization

If we were to use L1 regularization instead of L2 regularization, here's how the values of the weights and the metrics might differ:

1. **Training and Validation Set Accuracy**: L1 regularization might lead to higher variance in accuracy between the train and validation sets, especially if it leads to too many features being eliminated. This can potentially hurt the model's performance on the training set, but it may also improve generalization to the validation set if the regularization eliminates irrelevant features.

2. **Weights (W norm)**: The weight values will likely become sparser, meaning that many of the weights will be driven to zero. This leads to a lower W norm since fewer weights are non-zero. In contrast, L2 regularization shrinks the weights more smoothly but does not make them exactly zero, so the W norm is typically higher compared to L1.

In summary, L1 regularization tends to result in a sparser model with lower W norm, which could either improve or hurt accuracy depending on how much it eliminates useful features. L2 regularization leads to less sparse models and typically provides more consistent train and validation accuracy while maintaining a moderate W norm.

3. a) Multi-layer Perceptron, with a single hidden layer

```
Training took 18 minutes and 50 seconds
Final test acc: 0.5280
```

Fig. 9. Final test accuracy for MLP

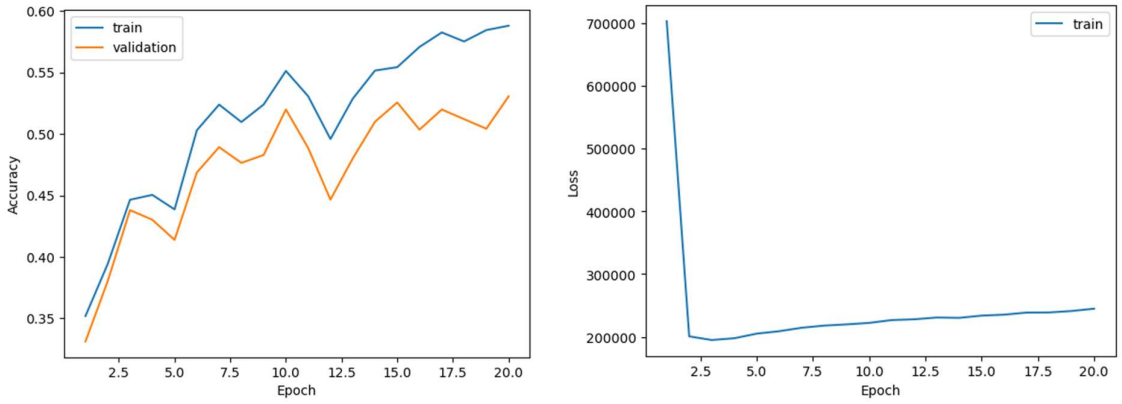

Fig. 10. Train and Validation Accuracy, and Train Loss with MLP

QUESTION 2

1. Logistic regression with PyTorch

```
Training took 0 minutes and 46 seconds
Final validation acc: 0.4060
Final test acc: 0.4233
```

Fig. 11. Final test accuracy for LRC with PyTorch and lr=0.00001

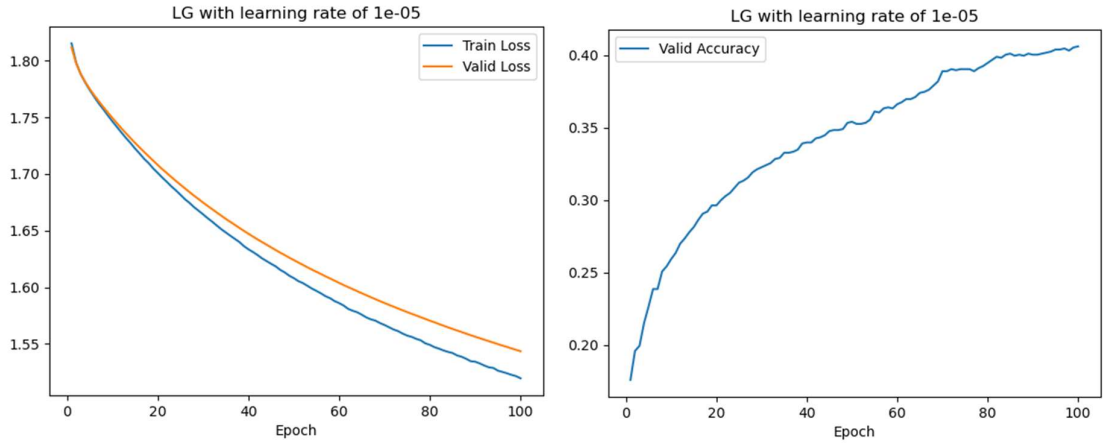

Fig. 12 and 13. Train and Validation Loss, and Validation Accurracy with lr=0.00001

```
Training took 0 minutes and 50 seconds
Final validation acc: 0.5235
Final test acc: 0.5253
```

Fig. 14. Final test accuracy for LRC with PyTorch and lr=0.001

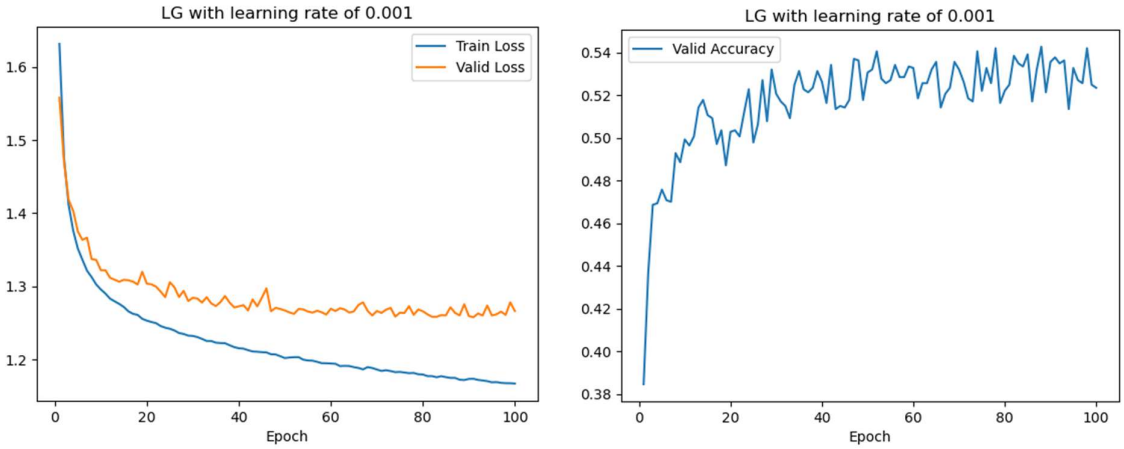

Fig. 15 and 16. Train and Validation Loss, and Validation Accurracy with lr=0.001

```
Training took 0 minutes and 51 seconds
Final validation acc: 0.2892
Final test acc: 0.2867
```

Fig. 14. Final test accuracy for LRC with PyTorch and lr=0.1

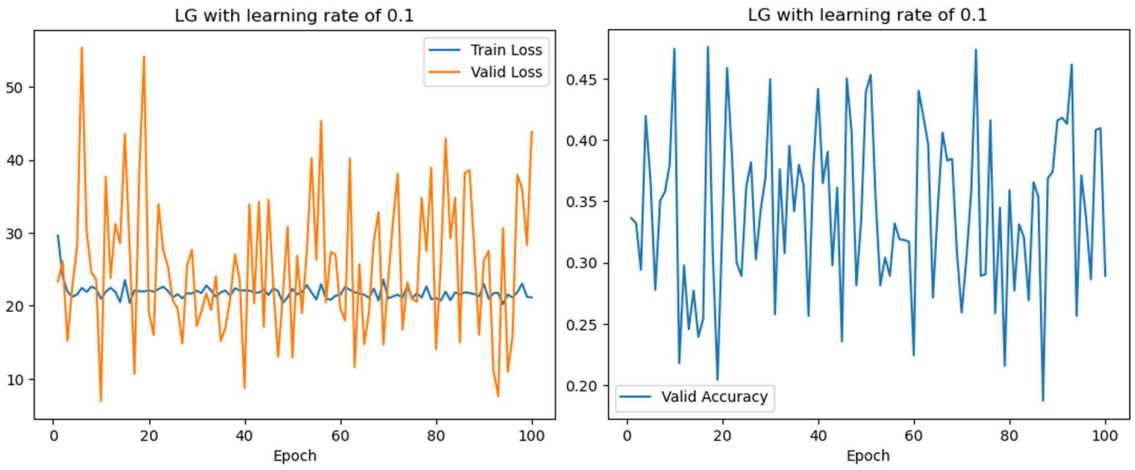

Fig. 15 and 16. Train and Validation Loss, and Validation Accurracy with lr=0.1

It is observed that the highest accuracy (validation: 52.35%; test: 52.53%) occurs with a learning rate of 0.01, compared to 0.1 and 0.00001.

Indeed, when the learning rate is too small (such as 0.00001), the model may struggle to learn efficiently because the weight adjustments during training are too small. This can cause the model to take much longer to converge or even get stuck in a local minimum without significantly improving performance.

On the other hand, a very high learning rate (such as 0.1) can cause the model to take very large steps during training, which may lead to instability, observable in the graph of Fig. 16. The model may overshoot the global minimum or fail to converge, resulting in worse performance.

With a learning rate of 0.001, the model achieves an effective balance between learning speed and stability, enabling it to learn more efficiently and perform well. Moreover, with this balanced learning rate, the loss reduction in both the training and validation sets occurs more quickly.

2. Multilayer Perceptron with PyTorch

2. a) Multilayer Perceptron with PyTorch, batch size

```
Training took 2 minutes and 19 seconds
Final validation acc: 0.5962
Final test acc: 0.5973
```

Fig. 17. Final test accuracy for MLP with PyTorch and batch_size=64

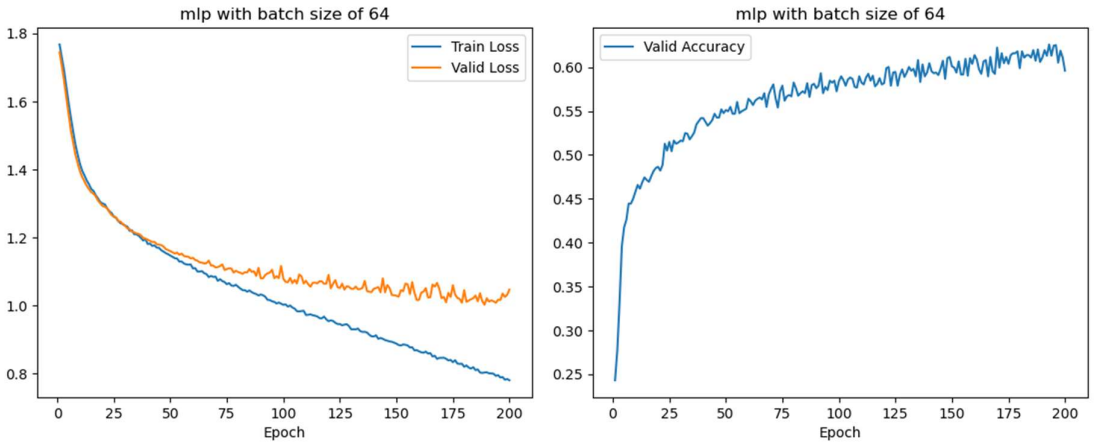

Fig. 18 and 19. Train and Validation Loss, and Validation Accurracy with batch_size=64

```
Training took 0 minutes and 48 seconds
Final validation acc: 0.5043
Final test acc: 0.5213
```

Fig. 20. Final test accuracy for MLP with PyTorch and batch_size=512

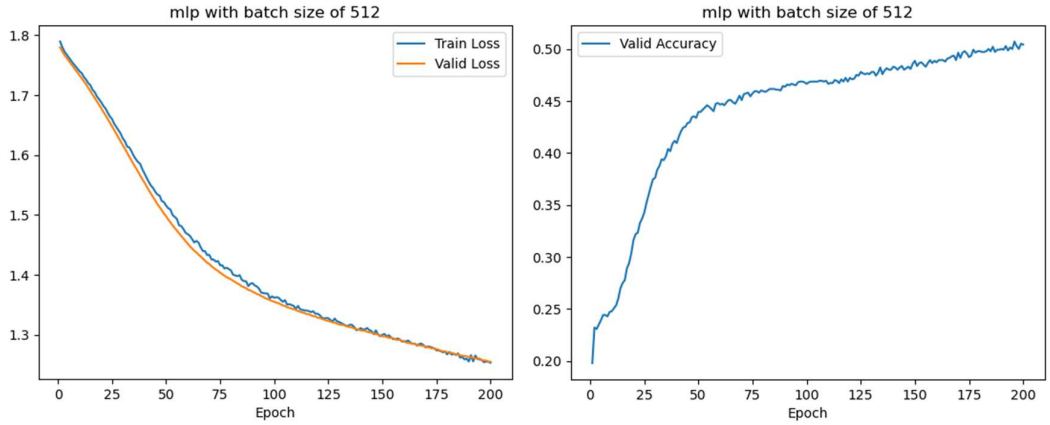

Fig. 21 and 22. Train and Validation Loss, and Validation Accurracy with batch_size=512

Using a smaller batch size (64) results in longer training time (2:19) for the MLP, but higher accuracy (validation: 59.62%; test: 59.73%). With a larger batch size, training is faster (0:48), but accuracy is lower (validation: 50.43%; test: 52.13%). The loss curve decreases more quickly, although the validation loss does not reach such low values as in the case of batch_size=512.

Using a **smaller batch size** results in longer training time for the MLP but higher accuracy. This happens because smaller batches lead to more frequent updates of the model's weights, allowing it to quickly adapt to variations in the data. These frequent updates help the model avoid local minima and overfitting, which improves its generalization ability. As a result, the accuracy tends to be higher. However, the downside is that more updates per epoch increase the total number of calculations, making the training process slower.

In contrast, using a **larger batch size** makes training faster but results in lower accuracy. Larger batches provide a more stable estimate of the data distribution, leading to smoother and more consistent updates of the model’s weights. This stability reduces the computational overhead, allowing the model to converge more quickly. However, the

lack of frequent updates makes the model less flexible in capturing small patterns in the data. Consequently, it may get stuck in local minima or fail to generalize well, leading to lower accuracy.

Thus, it is common to seek a **trade-off between training time and accuracy**. A batch size that is too small can increase computational time, while a batch size that is too large can reduce the model's generalization ability. An intermediate batch size often provides a good balance between these factors.

2. b) Multilayer Perceptron with PyTorch, dropout

```
Training took 2 minutes and 54 seconds
Final validation acc: 0.5819
Final test acc: 0.5817
```

Fig. 23. Final test accuracy for MLP with PyTorch and dropout=0.01

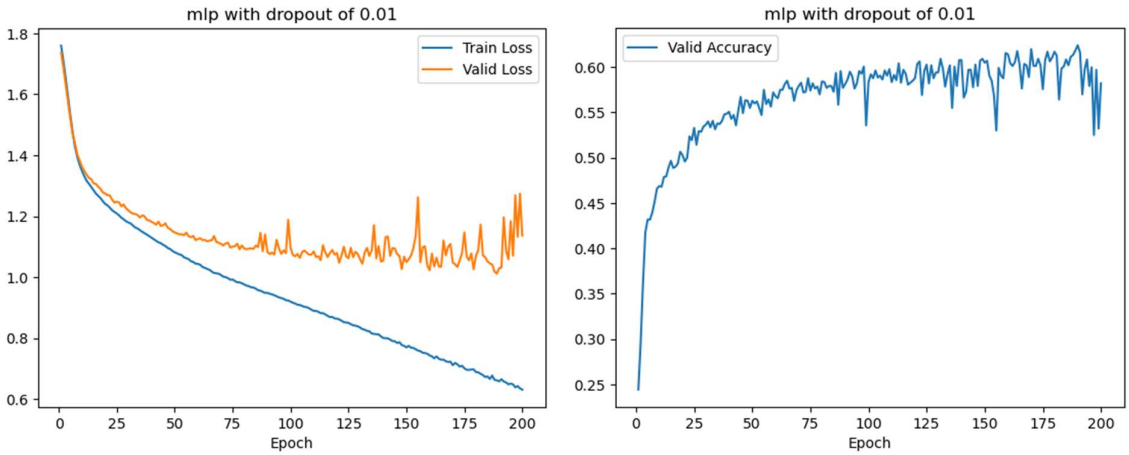

Fig. 24 and 25. Train and Validation Loss, and Validation Accurracy with dropout=0.01

```
Training took 2 minutes and 53 seconds
Final validation acc: 0.5990
Final test acc: 0.5963
```

Fig. 26. Final test accuracy for MLP with PyTorch and dropout=0.25

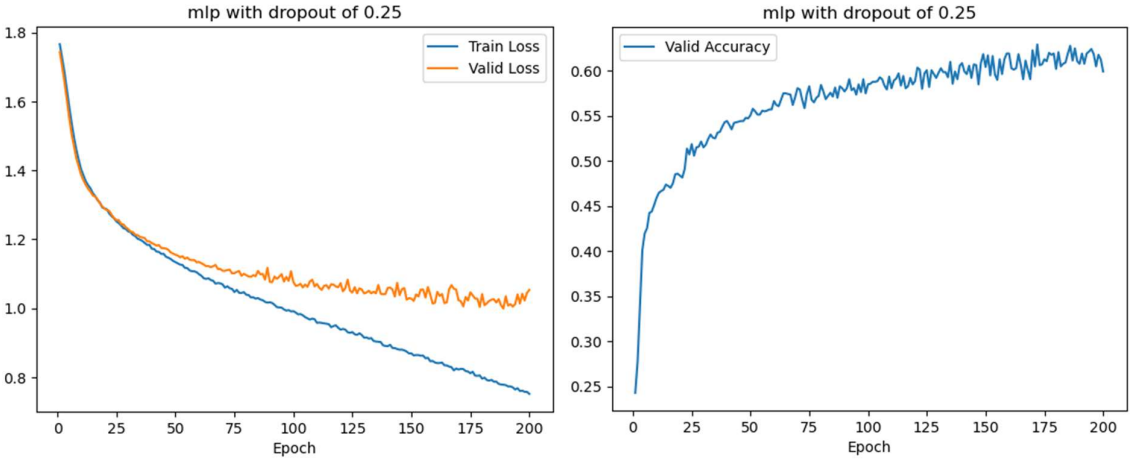

Fig. 27 and 28. Train and Validation Loss, and Validation Accurracy with dropout=0.25

```
Training took 2 minutes and 27 seconds
Final validation acc: 0.5919
Final test acc: 0.5880
```

Fig. 29. Final test accuracy for MLP with PyTorch and dropout=0.5

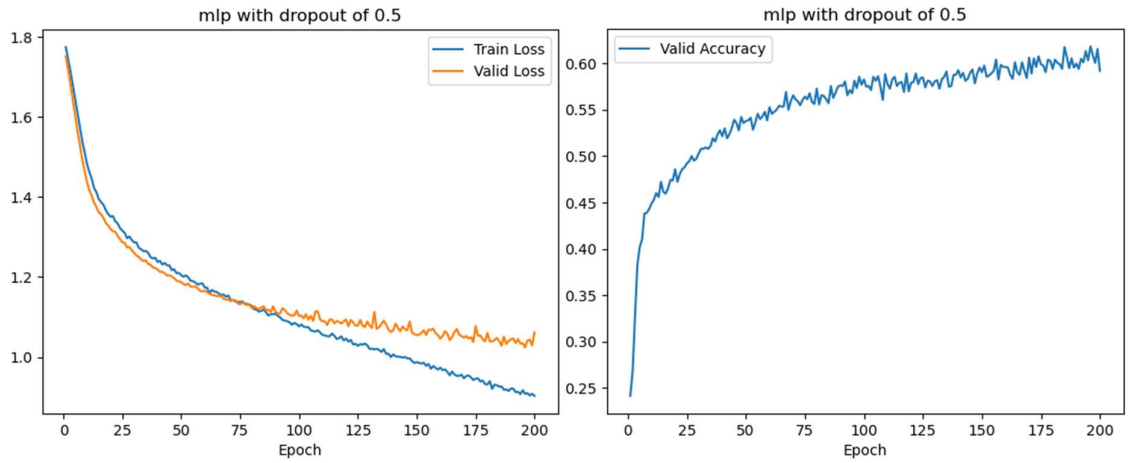

Fig. 30 and 31. Train and Validation Loss, and Validation Accurracy with dropout=0.5

As the dropout value increases, the **validation loss curve** progressively falls. Dropout acts as a regularizer by preventing overfitting. With a higher dropout rate, the network becomes less likely to memorize specific patterns in the training data. Instead, it learns more generalizable features that perform better on unseen data. As a result, the validation loss decreases because the model generalizes better to the validation set.

The performance of different dropout values depends on balancing the prevention of **overfitting** with maintaining the model's **learning** capacity. In our experiment, a dropout value of 0.25 resulted in better accuracy than 0.01 and 0.5 due to the following reasons:

1. With **very low** dropout (0.01), only 1% of the connections are deactivated during training. This value is too low to prevent overfitting, as the model keeps nearly all connections active, potentially leading to excessive dependence on the training data. As a result, the model may overfit the training data, performing worse on the validation set.
2. With a **moderate** dropout (0.25), 25% of the connections are randomly deactivated in each iteration. This value strikes a good balance: it reduces the model's dependency on specific connections, helping to prevent overfitting without overly compromising its learning capacity. Consequently, better generalization and higher accuracy are observed on the validation set.
3. With **high** dropout (0.5), 50% of the connections are deactivated in each iteration. While this effectively reduces overfitting, it can also hinder the model's ability to learn complex patterns in the training data. The model may struggle to capture important relationships between features, resulting in suboptimal learning. Consequently, performance decreases due to the reduced learning capacity.

In conclusion, a dropout value of 0.25 proved to be the best as it balances generalization and learning capacity. This value allowed the model to learn relevant patterns without overfitting the training data. The optimal dropout value varies depending on the dataset's complexity and size. In this case, 0.25 appears to have been ideal for the noise level and complexity of the data.

2. c) Multilayer Perceptron with PyTorch, momentum

```
Training took 0 minutes and 49 seconds
Final validation acc: 0.4615
Final test acc: 0.4767
```

Fig. 32. Final test accuracy for MLP with PyTorch and momentum=0.0

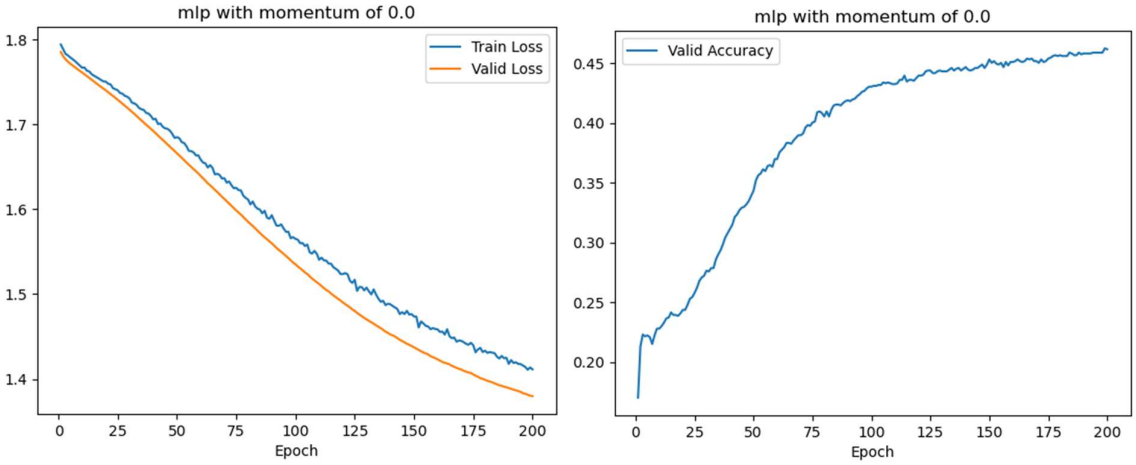

Fig. 33 and 34. Train and Validation Loss, and Validation Accurracy with momentum=0.0

```
Training took 0 minutes and 50 seconds
Final validation acc: 0.5805
Final test acc: 0.5870
```

Fig. 35. Final test accuracy for MLP with PyTorch and momentum=0.9

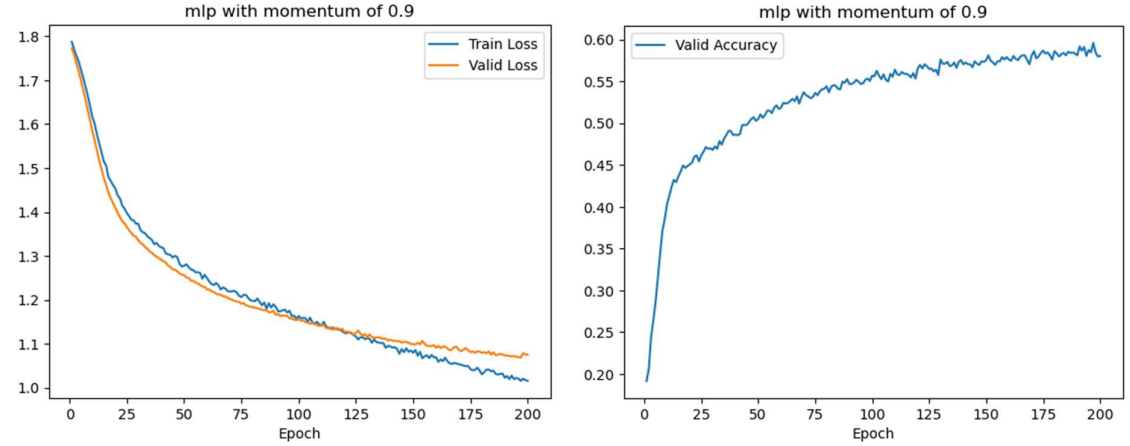

Fig. 31 and 32. Train and Validation Loss, and Validation Accurracy with momentum=0.9

The train and validation loss decrease much faster, and the validation accuracy increases more quickly when momentum=0.9. Additionally, the final values of train and validation loss are lower with momentum=0.9, while the final validation accuracy reaches a higher value (58.05%).

1. **Without momentum** (0.0), the model uses standard gradient descent, meaning weights are updated solely based on the gradient of the current loss. This causes training to progress in small steps, especially in regions with small gradients (plateaus or ravines in the loss function). The lack of inertia requires more iterations for the model to converge, resulting in a slow decrease in the train and validation

loss curves. Validation accuracy increases more gradually, as the model slowly adjusts the weights and learns relevant patterns.

2. With **momentum set to 0.9**, the model gains "inertia" in weight updates. Updates are no longer based solely on the current gradient but also on the accumulation of gradients from previous iterations. Consequently, updates become faster and more directed, particularly in regions of small gradients, such as plateaus, where standard gradient descent would be much slower. This explains the rapid decline in the train and validation loss curves, as the model adjusts more efficiently at the beginning of training. Validation accuracy experiences a significant initial boost because the model quickly learns the most basic and generalizable patterns in the training data. This directly translates to an initial improvement in validation performance.

In conclusion, momentum=0.9 accelerates convergence, reducing train and validation loss faster and boosting validation accuracy significantly early on. Final metrics show improved generalization, with lower losses and higher validation accuracy (58.05%), highlighting the benefits of momentum in training efficiency and performance.

QUESTION 3

1.

a. Expanding the Activation Function

The activation function $g(z) = z(1 - z)$ is equivalent to:

$$g(z) = z - z^2.$$

If the hidden layer is:

$$z = Wx + b,$$

then:

$$h = g(Wx + b) = (Wx + b) - (Wx + b)^2.$$

Thus, the internal representation becomes:

$$h = (Wx + b) - \left((Wx)^2 + 2(Wx)b + b^2\right).$$

b. Defining the Feature Transformation $\phi(x)$

The feature transformation $\phi(x)$ is a vector of features constructed from the original input vector x. In this case, we want $\phi(x)$ to capture both the linear and quadratic terms in x, since the activation function is quadratic.

We can map the feature transformation $\phi(x)$ independent of Θ as:

$$\phi(x) = \left[1, x_1, x_2, \ldots, x_D, x_1^2, x_1x_2, \ldots, x_D^2\right]^\top.$$

This includes the following terms:

- Constant term: 1, which corresponds to the first feature in $\phi(x)$ (bias),
- Linear terms: $x_1, x_2, \ldots, x_D$, which are the individual components of the input vector x,
- Quadratic terms: $x_1^2, x_1x_2, \ldots, x_D^2$, which are the pairwise products of the input components.

The total sum of terms in the vector $\phi(x)$ is given by the sum of 1 constant term (1), D linear terms $(x_1, x_2, \ldots, x_D)$, $\frac{D(D+1)}{2}$ quadratic terms, which include the squares of the features (D) and the interactions between pairs of features $(\frac{D(D-1)}{2})$:

$$1 + D + \frac{D(D+1)}{2} = \frac{(D+1)(D+2)}{2}$$

c. Defining the Linear Transformation A_Θ

We aim to express h as:

$$h = A_\Theta \phi(x).$$

The role of A_Θ is to select linear combinations of the elements in $\phi(x)$, using the parameters W and b.

The structure of A_Θ is generically as follows:

$$A_\Theta = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,M} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K,1} & a_{K,2} & \cdots & a_{K,M} \end{bmatrix},$$

where $M = \frac{(D+1)(D+2)}{2}$ is the size of $\phi(x)$, and $a_{k,m}$ is a combination of W and b, defining how the terms in $\phi(x)$ contribute to h_k, the output of the k-th hidden neuron.

Therefore, A_Θ is a $K \times M$ matrix constructed from the weights W and biases b.

2.

a. Linearity of transformation $\hat{y}$ w.r.t $\phi(x)$

The vector $c_\Theta \in \mathbb{R}^{\frac{(D+1)(D+2)}{2}}$ corresponds to the mapping from the feature vector $\phi(x)$ to the output $\hat{y}$.

Since $g(Wx+b)$ can be viewed as a nonlinear transformation of x, we can express $\hat{y}$ as:

$$\hat{y} = v^\top g(Wx + b) + v_0$$

Given that $g(Wx + b)$ is nonlinear but can be written as a function of the vector $\phi(x)$ (which captures both the linear and quadratic terms of x), we can rewrite $\hat{y}$ as:

$$\hat{y} = c_\Theta^\top \phi(x).$$

The key observation here is that $\hat{y}$ is now expressed as a linear combination of the components of $\phi(x)$. Since $\phi(x)$ contains the necessary nonlinear terms (such as quadratic terms), the nonlinearities in the original model are captured in $\phi(x)$, and $\hat{y}$ is linear in $\phi(x)$.

b. Determining c_Θ

The vector c_Θ will be formed by combining the weights from the hidden layer W, the biases b, the output weights v, and the output bias v_0.

The vector c_Θ is then a vector of length $\frac{(D+1)(D+2)}{2}$, and it depends on the matrix A_Θ, which incorporates both the weights W and the activations. The vector c_Θ is a transformation of the parameters W, b, v, and v_0, encapsulating the mapping from $\phi(x)$ to $\hat{y}$. It can be written as:

$$c_\Theta = \begin{bmatrix} W \\ b \\ v \\ v_0 \end{bmatrix}$$

This representation captures the linear transformation from $\phi(x)$ to $\hat{y}$ and depends on all of the original parameters $\Theta = (W, b, v, v_0)$.

c. Models's nonlinearity

The linearity of the transformation does not mean that the model is linear in terms of the original parameters Θ. Indeed, $\hat{y}$ is expressed as:

$$\hat{y} = c_\Theta^\top \phi(x)$$

where $\phi(x)$ is a transformed feature vector that includes both the original input features x and their higher-order (nonlinear) transformations, such as quadratic terms. The vector c_Θ encapsulates the parameters of the model, including the weights W, biases b, output weights v, and output bias v_0.

The key to this model's nonlinearity is the function $g(Wx + b)$, which introduces a nonlinear transformation of the input x. The output $\hat{y}$ is expressed as a linear combination of these transformed features in $\phi(x)$, but $\phi(x)$ itself is nonlinear because it includes higher-order terms (e.g., quadratic terms). In fact, while c_Θ represents a linear transformation of the feature vector $\phi(x)$, the relationship between the input x and the output $\hat{y}$ through the parameters Θ is still nonlinear because the activation function $g(z) = z(1 - z)$ introduces nonlinearity into the feature map $\phi(x)$.

3.

This question explores the relationship between the parametrization of a model using a parameter vector $\mathbf{c}$ and the original model parameters $\Theta = (W, b, v, v_0)$. The objective is to prove that, for any $\mathbf{c}$ and a sufficiently small precision $\varepsilon > 0$, the parameters Θ can be chosen to approximate $\mathbf{c}$ within the given precision. This implies a series of goals, such as:

- Representing complex models with a smaller set of parameters without significant loss of precision.
- Finding good initializations for the parameters of a model.
- Ensuring that small adjustments to the parameters do not cause model failures.
- Gaining a better understanding of how the parameters of a model contribute to the output.
- Transferring models between different architectures.
- Constructing models that respect specific constraints.
- Solving problems where direct optimization is challenging.
- Reducing redundancy in models.

a. Representation of $\mathbf{c}_\Theta$ in terms of the model

The model uses $\phi(x)$, a feature transformation that maps inputs into a higher-dimensional polynomial space. We assume $\phi(x)$ has $M = \frac{(D+1)(D+2)}{2}$ terms.

The parameters W, b define the hidden layer outputs $h_k(x)$, and v, v_0 linearly combine $h_k(x)$ to produce the final output. Hence, $\mathbf{c}_\Theta$ is related to Θ via:

$$\mathbf{c}_\Theta = A_\Theta \cdot \phi(x),$$

where A_Θ is a $K \times M$ matrix constructed from W, b, and $\phi(x)$ is the polynomial basis vector.

Furthermore, assuming that $K \geq D$ is important to guarantee that the model has sufficient capacity to represent the parameters $\mathbf{c}$ or approximate them within a precision ε.

b. QR decomposition of $\phi(x)$

If a matrix Φ is close to singular, it can be perturbed by a small amount (denoted ϵ) to make it non-singular, i.e., full rank. Mathematically, if Φ is close to singular, there exists a small perturbation Δ such that:

$$\|\Delta\| < \epsilon \quad \text{and} \quad \Phi + \Delta \text{ is non-singular.}$$

This is known as the *matrix perturbation theorem*. It tells us that for most practical matrices (those not too close to being singular), small perturbations can make them non-singular (full rank).

So, we assume Φ, the matrix formed by evaluating $\phi(x)$ over the dataset, has full rank or is ϵ-close to a matrix with full rank, and perform QR decomposition on Φ:

$$\Phi = QR,$$

where Q is $M \times M$ orthogonal ($Q^\top Q = I$) and R is $M \times M$ upper triangular.

Now, we represent $\mathbf{c}$, any real vector, in terms of Q and R:

$$\mathbf{c} = QR\mathbf{c}',$$

where $\mathbf{c}'$ is a transformed version of $\mathbf{c}$.

c. Construction of A_Θ

Using the QR decomposition of Φ, we write:

$$A_\Theta = QRA'_\Theta,$$

where A'_Θ is adjusted such that:

$$c_\Theta = QRA'_\Theta \approx c.$$

The previous equation indicates that c_Θ is approximately equal to some vector c. This means that the true coefficients c_Θ are approximately equivalent to the vector c, which might represent the ideal solution or the desired weights.

In practice, the approximation $\approx c$ suggests that after performing the QR decomposition and adjusting the transformation matrix A'_Θ, the resulting vector c_Θ is very close to c, which could be an optimal set of weights that fits the data well.

4.

a. Expressing the loss function

Assuming that $\hat{y}(x_n; c_\Theta) = c_\Theta^T \phi(x_n)$ and $\phi(x_n)$ is the feature vector corresponding to the input x_n, we can rewrite the squared loss function as:

$$L(c_\Theta; D) = \frac{1}{2} \sum_{n=1}^{N} \left(c_\Theta^T \phi(x_n) - y_n \right)^2$$

This loss function is essentially a quadratic function of c_Θ. We can write it in matrix form for convenience.

If $X \in \mathbb{R}^{N \times \frac{(D+1)(D+2)}{2}}$ is the matrix whose rows are the feature vectors $\phi(x_n)^T$, and $y \in \mathbb{R}^N$ the vector of target values y_n, the predicted output for all training points can then be written as:

$$\hat{y} = X c_\Theta$$

and the loss function becomes:

$$L(c_\Theta; D) = \frac{1}{2} \left\| X c_\Theta - y \right\|^2$$

a. Minimizing the loss function

We need to take the derivative of $L(c_\Theta; D)$ with respect to c_Θ and set it equal to zero:

$$\nabla_{c_\Theta} L(c_\Theta; D) = X^T (X c_\Theta - y) = 0$$

Solving for c_Θ, we get:

$$X^T X c_\Theta = X^T y$$

Assuming that $X^T X$ is invertible, we can solve for c_Θ:

$$c_\Theta = (X^T X)^{-1} X^T y$$

Thus, the closed-form solution for c_Θ is:

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

In conclusion, we can find a closed-form solution for c_Θ, provided that the matrix $X^T X$ is invertible.

d. Global minimization

Global minimization in feedforward neural networks is, in general, challenging due to the non-convexity of the loss function, high dimensionality, and the non-linear nature of the model. The loss surface has multiple local minima, making it computationally expensive to find the global minimum. High-dimensional parameter spaces and non-linear activations further complicate optimization, increasing the difficulty of reaching an optimal solution.

In the specific case we're working with, there may be aspects that make the problem special:

- **Reparameterization of the Activation Function:** The activation function $g(Wx + b)$ is a nonlinear transformation of x, but it can be viewed as a linear transformation of the feature vector $\phi(x)$. This reparameterization simplifies the optimization, making the problem more tractable.
- **Closed-Form Solution for c_Θ:** If a closed-form solution for c_Θ exists, it avoids the need for iterative optimization, bypassing the usual challenges of local minima and simplifying the problem.
- **Simplified Model Structure:** If the model has fewer layers or constraints, the loss function may have fewer local minima, making the global minimum easier to find. This reduces the complexity compared to typical neural networks.