Deep Learning (CS324)
Assignment 1 Report

Fitria Zusni Farida
Student ID: 12112351

April 3, 2024

1 Introduction

Neural networks play an important role to solve problems in Artificial Intelligence. It is able to learn to solve problems by implementing how the human biological neural network in the brain revolutionized the approach to understand data, uncover patterns, and make predictions. Neural networks' ability to model complex and non-linear relationships between inputs and outputs allows them to excel in tasks that are challenging for traditional algorithmic approaches. This adaptability and learning capability make them invaluable in diverse domains, ranging from image recognition, where they interpret visual data, to natural language processing, which involves understanding and generating human language.

Neural network consists of layers, with its layers consisting of neurons or nodes, the basic units. Each neuron receives input, processes it, and passes its output to the next layer. It is also able to learn from the data by updating the weights through backpropagation. The processing involves weighted sums of the inputs plus a bias term, followed by the application of an activation function. A neural network with only one layer is called a single layer perceptron. Whereas, a neural network with one or more hidden layers in addition to input and output layers is called multi-layer perceptron. MLPs are said to be more powerful because they can approximate virtually any continuous function and solve problems that are not linearly separable.

2 Motivation

The primary motivation of this project are as follow:

(1) Evaluate and compare the accuracy of single layer perceptron using different gaussian distributions with different variance.

(2) Evaluate the accuracy of training and test data using multi layer perceptron with default value of the parameters.

(3) Evaluate the multilayer perceptron model using stochastic gradient descent.

(4) Analyze the influence of batch size from 1 to a relatively large number using a multilayer perceptron model using batch gradient descent parameter.

3 Methodology

In this experiment, two different neural network models, single and multi layer perceptrons are implemented.

(1) Perceptron

In this single layer perceptron, the task is binary classification (class 1 and -1) from a given distribution dataset. The perceptron architecture is illustrated in the following figure. A given dataset with M samples and N feature dimension is feeded to the neural network. A perceptron (with a neuron) has weights with dimension N+1 (including bias, which allows the decision boundary to shift). Its weight is initialized to be all zeros.

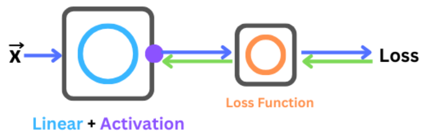


Figure 1: The architecture of the single layer perceptron.

The perceptron decision is based on following formula:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$

where the activation function y(x) is given by:

$$\text{y}(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases}$$

To obtain good perceptron model, the model should be able to learn by training the training dataset and update or adjust the weight based on the following explained steps:

Step 1: Predict the given input using above formulation in the forward() function as above formulation mentioned.

Step 2: From the prediction, there are four possible cases:

predicted label	**true label**
1	1
1	-1
-1	1
-1	-1

Table 1: Predicted vs. True Labels

Then, the loss function $L(w, b)$ can be defined as the sum of the product of the output and the weight vector plus the bias for misclassified points:

$$L(w, b) = \sum_{i=1}^{p} (y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

where p is the number of misclassifications, where the loss is always be negative value.

Step 3: From the above loss function, the derivative of the loss function wrt the parameter can be obtained as follows:

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = \sum_{\mathbf{x}_i \in M} y_i \mathbf{x}_i$$

$$\nabla_b L(\mathbf{w}, b) = \sum_{\mathbf{x}_i \in M} y_i$$

Step 4: These gradients can be used to update the parameters using following updating rules:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}, b)$$

$$b_{\text{new}} = b_{\text{old}} + \eta \nabla_b L(\mathbf{w}, b)$$

where η is the learning rate.

Step 5: Do the same thing for the next epoch until it reaches the intended number of epochs to update the weights.

Step 6:Evaluate the loss and accuracy for each epoch to see in which epoch the loss function converges. In other words, its perceptron model performs better in that epoch.

(2) Multi-Layer Perceptron

In this multilayer perceptron, the task is multi classification with C classes. The given input X is MxN dataset with M samples and N dimension and corresponding label MxC one-hot encoding with M samples and C classes. The network architecture consist of as described following:

1) Linear layer (fully-connected layer): applies a linear transformation to the incoming data, output = x.W + b. This layer is responsible for learning the weighted sum of inputs, which can represent any linear relationship between the input data and the output.

2) ReLU Activation (Rectified Linear Unit): introduces non-linearity into the model, allowing the network to learn complex patterns. ReLU is preferred in hidden layers because it helps alleviate the vanishing

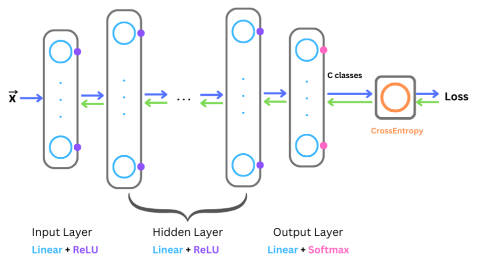


Figure 2: The architecture of the MLP.

gradient problem, is computationally efficient, and often leads to faster convergence.

3) Softmax: for multi-class classification problems, the Softmax function is often applied after the final linear layer to convert the vector of raw (logit) scores to probabilities.

For each linear layer, the number of parameters (weights and biases) depends on how many inputs from the previous layer and the number of neurons or output of the layer. Its parameters are stored inside the array called ‘params‘ and will be updated during backpropagation.

After forward-propagration, the backpropagation step involves forward propagation to find the Loss, then to find the grads of Loss with respect to input (the output of softmax). To avoid computationally expensive and numerically unstable step Jacobian matrix in the Softmax backward pass, the Softmax backward is integrated with the backward pass in the CrossEntropy loss, so that it can be obtained as follow:

$$\frac{\partial L}{\partial x_j} = p_j - y_j$$

In which p is predictions probability and y is true label. During backpropagation, the ReLU activation function update the dout (gradient of Loss wrt to the output of the ReLU layer) by following formulation. The derivative of the ReLU function is given by:

$$\frac{\partial \text{ReLU}}{\partial x_i} = \begin{cases} 1 & \text{if } x_i > 0, \\ 0 & \text{if } x_i \leq 0. \end{cases}$$

The chain rule for the derivative of the loss function with respect to the input of ReLU is:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \text{ReLU}(x_i)} \cdot \frac{\partial \text{ReLU}}{\partial x_i}$$

Combining these, the gradient of the loss function with respect to the input is:

$$\frac{\partial L}{\partial x_i} = \begin{cases} \frac{\partial L}{\partial \text{ReLU}(x_i)} & \text{if } x_i > 0, \\ 0 & \text{if } x_i \leq 0. \end{cases}$$

In the layer which contains parameters where Linear function is performed, the parameters will be updated from the backward pass. The updating rule is as follow:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}, b)$$

$$b_{\text{new}} = b_{\text{old}} - \eta \nabla_b L(\mathbf{w}, b)$$

where η is the learning rate. The ‘grad‘ array in the Linear layer stores the gradient of the Loss wrt its own layer's parameter. This gradient of Loss wrt to parameter can be formulated as follow:

Gradients of the Loss with respect to Weights ($d\mathbf{W}$):

$$\text{self.grads}['\text{weight}'] = \frac{\partial L}{\partial \mathbf{W}}$$

Here, L represents the loss function of the network. During the backward pass, dout is the gradient of the loss function with respect to the output of the current layer, $\frac{\partial L}{\partial \text{out}}$. Given the forward pass equation out $= \mathbf{XW} + \mathbf{b}$, we can find $\frac{\partial L}{\partial \mathbf{W}}$ by applying the chain rule:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial \mathbf{W}}$$

Since out $= \mathbf{XW} + \mathbf{b}$, $\frac{\partial \text{out}}{\partial \mathbf{W}} = \mathbf{X}$. Therefore, the gradient with respect to the weights is computed by:

$$\text{self.grads}['\text{weight}'] = \mathbf{X}^\top \cdot \text{dout}$$

This gives us the rate at which the weights should be adjusted during epoch iteration to minimize the loss.

Gradients of the Loss with respect to Biases ($d\mathbf{b}$):

$$\text{self.grads}['\text{bias}'] = \frac{\partial L}{\partial \mathbf{b}}$$

Similar to weights, we use the chain rule:

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial \mathbf{b}}$$

And since out $= \mathbf{XW} + \mathbf{b}$, $\frac{\partial \text{out}}{\partial \mathbf{b}} = 1$. The gradient with respect to biases is the sum of gradients across the batch:

$$\text{self.grads}['\text{bias}'] = \sum_{\text{samples}} (\text{dout})$$

The `'keepdims=True'` ensures that the output has the same dimensions as the bias.

In this experiment, three modes of updating parameters are implemented.

1) 'batch' mode: This is default mode using batch gradient descent to update the rule. It updates the weights in once all training samples.

2) 'stochastic' mode: It updates the parameter by considering single training sample at a time. Here, the sample will be selected randomly from the given training set.

3) 'mini-batch' mode: It updates the weights once in every batch with specified batch size.

4 Experiments

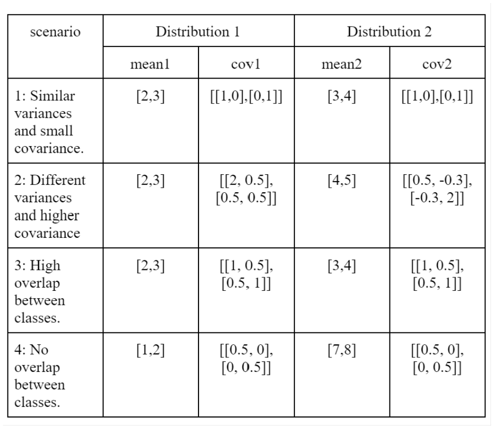
(1) Perceptron

Step 1: Data preparation

Two Gaussian distributions in R^2 are defined. The dataset used in this experiment contains 200 points in total. Its dataset is divided into a training dataset which contains 80 points sampled from each distribution and a test dataset which contains 20 points sampled also from each distribution.

The task is binary classification. Thus, the label used here is represented by -1 and 1 which is created exactly balanced. To ensure robustness, the dataset is also shuffled to avoid the model learning solely based on the features.

The gaussian distributions experimented are divided into following scenarios in Figure 3.



scenario	Distribution 1		Distribution 2	
	mean1	cov1	mean2	cov2
1: Similar variances and small covariance.	[2,3]	[[1,0],[0,1]]	[3,4]	[[1,0],[0,1]]
2: Different variances and higher covariance	[2,3]	[[2, 0.5], [0.5, 0.5]]	[4,5]	[[0.5, -0.3], [-0.3, 2]]
3: High overlap between classes.	[2,3]	[[1, 0.5], [0.5, 1]]	[3,4]	[[1, 0.5], [0.5, 1]]
4: No overlap between classes.	[1,2]	[[0.5, 0], [0, 0.5]]	[7,8]	[[0.5, 0], [0, 0.5]]

Figure 3: Experiment scenarios.

Step 2: Implement the perceptron with adjusted epoch and learning rate. In this experiment, 100 epochs and 0.01 learning rate are implemented.

Step 3: Train the training dataset and test the test set with trained model.

Step 4: Evaluate the training and test accuracy. From the evaluation, it is obtained following result showed in Table 2.

Scenario	train accuracy (in 100th epoch)	test accuracy
1	0.76	0.68
2	0.73	0.75
3	0.51	0.53
4	1.0	1.0

Table 2: Training and test accuracy.

(2) Multi-Layer Perceptron

Step 1: Data preparation Using sci-kit learn and the make moons method, a dataset of 1000 two-dimensional points is generated. The dataset has two classes, represented by class 0 and class 1. Its dataset is then divided into 20% training and 80% testing dataset. Its label is in one-hot encoding with dimension M by C, which M is the number of samples and C is the number of classes.

The experiments are implemented as following scenarios:

Scenario 1: `make_moons(n_samples=1000, shuffle=True, noise=0.2, random_state=42)`

Scenario 2: `make_moons(n_samples=1000, shuffle=True, noise=0.4, random_state=42)`

Scenario 3: `make_moons(n_samples=[750, 250], shuffle=True, noise=0.2, random_state=42)`

Step 2: Train and test the MLP model using following default hyper-parameter:

```
dnn hidden units = '20'
learning rate = 1e-2
max steps = 1500
eval freq = 10
mode = 'batch'
```

Step 3: Train and test the MLP model using above hyper-parameters except for the mode which is set to be mode = 'stochastic'

Step 4: Train and test the MLP model using above hyper-parameters except for mode which is set to be mode = 'mini-batch' and specify the batch size.

Finally, from the testing set, it is obtained following loss and accuracy in Figure 4:

If we take closer look to the mini-batch mode, it can be drawn following relationship between batch size and the accuracy in Figure 5:

mode	batch size	Scenario 1	Scenario 2	Scenario 3
batch	-	88.0	77.0	92.5
stochastic	-	85.0	83.0	86.0
mini-batch	1	99.0	84.4	97.5
mini-batch	2	99.0	83.5	98.0
mini-batch	4	99.0	85.5	97.5
mini-batch	8	98.0	85.0	98.0
mini-batch	16	97.5	85.0	98.5
mini-batch	32	98.5	84.5	98.0
mini-batch	64	98.5	85.5	98.5
mini-batch	128	97.5	84.5	98.5
mini-batch	256	99.0	85.5	97.5
mini-batch	512	98.5	85.0	97.5
mini-batch	1000	87.0	77.0	85.0

Figure 4: MLP accuracy result.

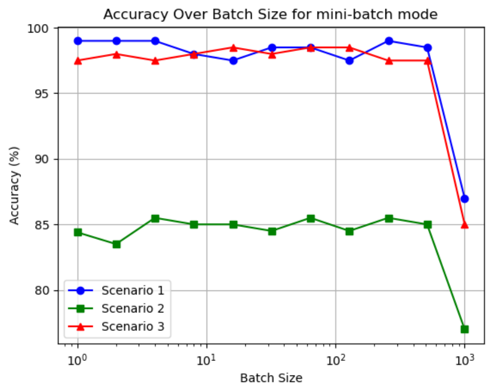


Figure 5: mini batch accuracy result.

5 Discussion and Insights

(1) Perceptron

From the experiment result in Table 2. It can be obtained following analysis:

- Scenario 1

This scenario represents two classes with similar variances and small covariance. The perceptron performs reasonably well but not perfectly with 0.76 training accuracy and 0.68 testing accuracy, which suggests that while the two distributions are somewhat separable, there is still some overlap that the perceptron cannot separate. The test accuracy is lower than training accuracy indicates slight overfitting to the testing data.

- Scenario 2

Despite the higher covariance, which typically implies more overlap between classes, the test accuracy is higher than in Scenario 1 with accuracy 0.75. This could be due to the different variances providing enough separation for the perceptron to generalize better. As the distribution shown in the appendix, the dataset distribution is well separated between two opposite region, but some samples are overlap along the separation line, which is the cause why the model accuracy cannot reach perfect. Compared to Scenario 1, which we can see the points are still distributed along opposite regions, in the Scenario 2, the samples are rarely seen in the opposite region (not as much as Scenario 1), which makes the model learn better when it is implemented in the test dataset.

- Scenario 3

Here, the classes have high overlap, as indicated by the covariance matrix, which is closer to being a diagonal matrix with larger values. The samples are scattered in each other class regions. The perceptron struggles in this scenario, with both training and test accuracies close to 0.5, which is kind of random guessing for a binary classification task. This scenario represents a difficult classification problem for a linear model like the perceptron due to the significant class overlap.

- Scenario 4

This is an ideal scenario where the two classes are linearly separable with no overlap, leading to perfect classification on both training and test sets. The perceptron is able to learn the decision boundary perfectly, which is reflected in the accuracy.

(2) Multi-Layer Perceptron

Based on the result in Table 4, it can be analyzed as following:

- 'batch' mode

In Scenario 1 and 2, this mode shows a relatively lower accuracy compared with and mini-batch mode. Moreover, these two scenarios show that 'batch' and 'mini-batch' with batch size 1000, which take whole dataset size, result in slightly similar accuracy. In Scenario 3, its mode perform better than stochastic mode. This may be due to this mode take the whole samples and try to draw conclusion once from a time. Whereas, the stochastic mode randomly take one sample which the probability taking class 0 is higher since it's the majority class. However, the mini-batch mode perform better compared with batch and stochastic. This may be due to for each batch, the model try to learn the behavior and implement it to the next batches, which more likely to have similar behavior. As it can be seen in the appendix, the loss is reduced significantly in the first few steps.

- 'stochastic' mode

In the Scenario 2, which the separation between classes are more difficult than Scenario 1 and 2, the stochastic mode performs better than batch mode. This may be due to stochastic take one sample randomly, and the probability taking sample from the its own region is higher (as it can be seen in the appendix that the overlapped samples are mostly in the boundary). However, its performance worse in Scenario 1 and 3 compared with the mini-batch. The mini-batch with batch size 1-512 show better accuracy compare this stochastic mode. From the figure in showed in the Appendix, this stochastic mode learn faster (by noticing the computational time). In the first few steps, the loss decrease significantly (because it chooses sample generated largest error from previous epoch), and also it is noticed that along with the epoch or steps, the accuracy fluctuate and may not necessarily decrease.

- 'mini-batch' mode

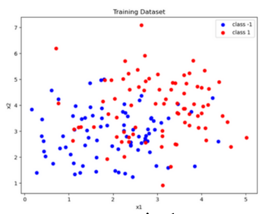
As shown in the Figure 5 the accuracy over batch size, Scenario 1 and 2, the graph depict similar pattern as the number of batch size increase. As shown in the distribution in the appendix, these Scenario 1 and Scenario 2 only differ in the noise. Scenario 3, with imbalance dataset has different pattern except for batch size = 1000, which drop significantly among other smaller batch sizes. Imbalance dataset might be sensitive to the smaller batch size because smaller batches may contain a more representative matrix of mix classes. When the batch size equals the total number of samples, it's possible that the model's performance is negatively affected by the imbalanced nature of the data because it is exposed to the full extent of the imbalance in each training step. Scenario 2 shows lower accuracy compared with Scenario 1 and 3 since its noise is higher, thus, it struggles to separate between the classes. Moreover, larger batch sizes lead to faster computational time, but could negatively impact the model's ability to generalize well, as it can be seen these three scenario's accuracy drop significantly for the number of sample batch size.

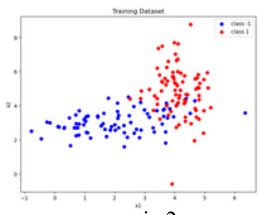
6 Conclusions

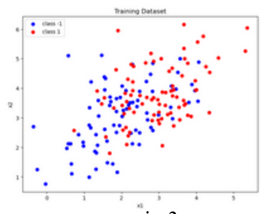
Through a series of experiments with both single and multi-layer perceptron models, it can be obtained valuable insights into the capabilities and limitations of these neural network architectures and the influence of gradient descent methods on their performance. The single-layer perceptron, functioning as a linear classifier, demonstrated varying levels of success depending on the degree of separation between the distributions. It excelled when the distributions were linearly separable with minimal overlap, but its performance deteriorated as the complexity of the class separation increased—particularly when classes exhibited significant covariance and overlap. The multi-layer perceptron (MLP), equipped with the capacity to model non-linear boundaries, showcased a more robust performance against the complexities. The inclusion of hidden layers enabled the MLP to learn more complex patterns. The results reinforce the importance of selecting the appropriate model complexity and optimization strategy tailored to the problem at hand—a single-layer perceptron for simpler, linearly separable tasks, and a multi-layer perceptron with a well-chosen gradient descent method for more complex scenarios.

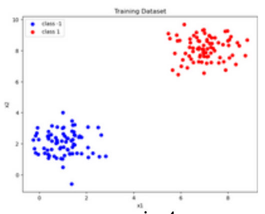
Appendix

1) Perceptron

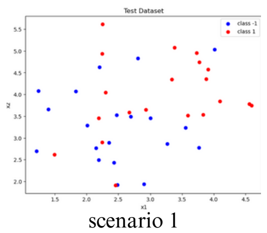

scenario 1

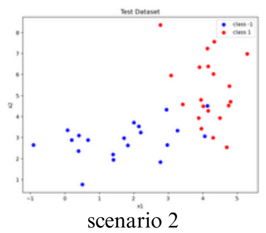

scenario 2

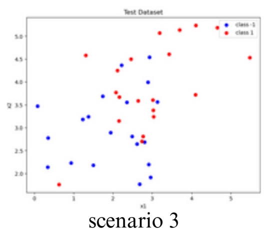

scenario 3

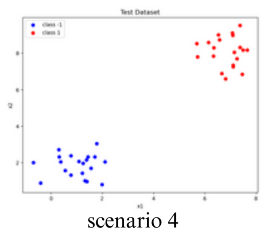

scenario 4

Training dataset distribution

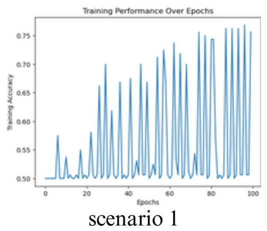

scenario 1

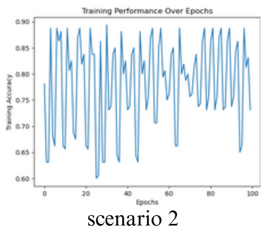

scenario 2

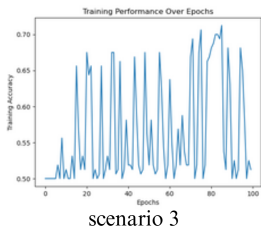

scenario 3

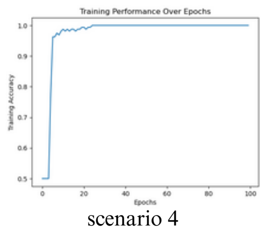

scenario 4

Test dataset distribution


scenario 1

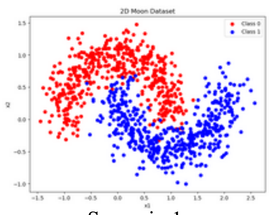

scenario 2

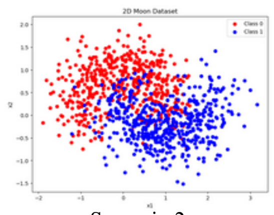

scenario 3

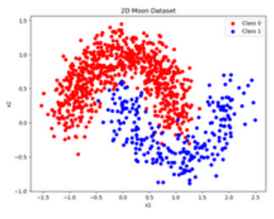

scenario 4

Training accuracy over epochs

2) Multi-Layer Perceptron

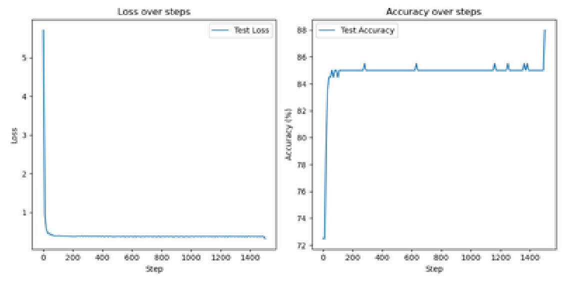

Scenario 1

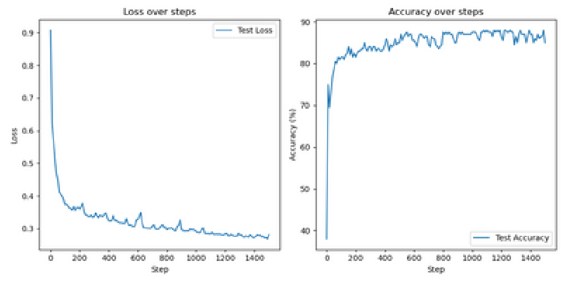

Scenario 2

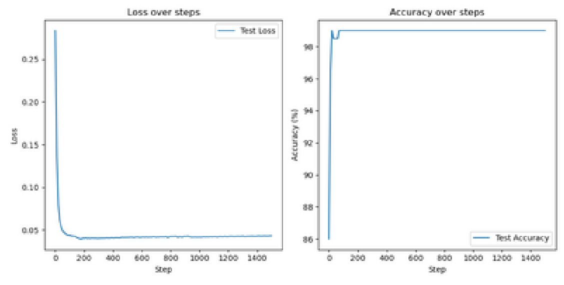

Scenario 3

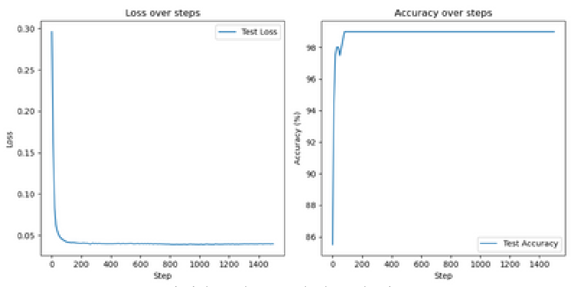
Dataset distribution

- Scenario 1

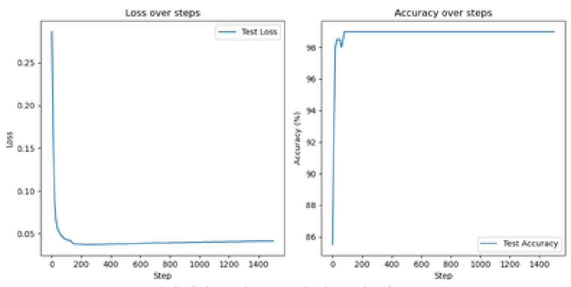

`batch` mode


`stochastic` mode

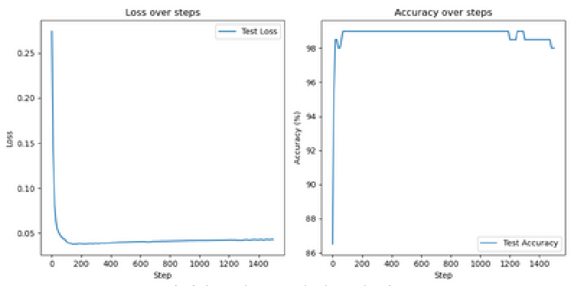

`mini-batch` mode size = 1



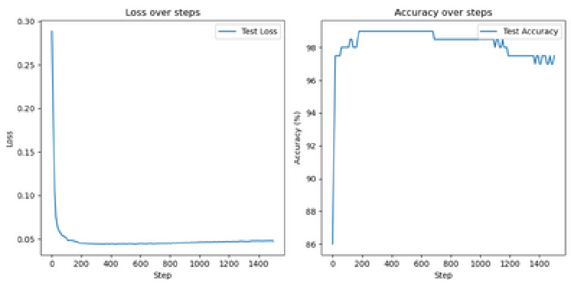
`mini-batch` mode batch size = 2



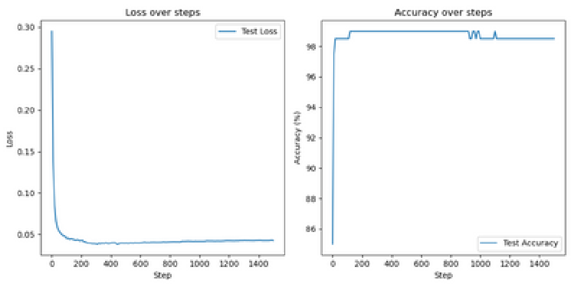
`mini-batch` mode batch size = 4



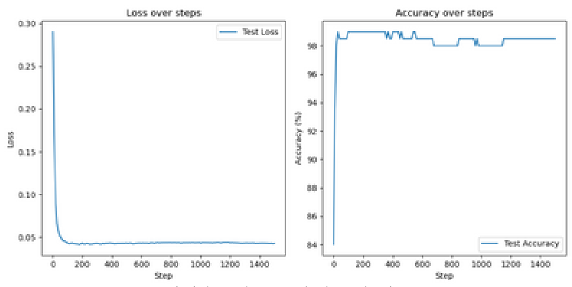
`mini-batch` mode batch size = 8



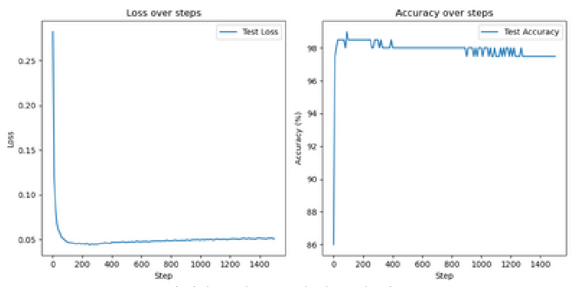
`mini-batch` mode batch size = 16



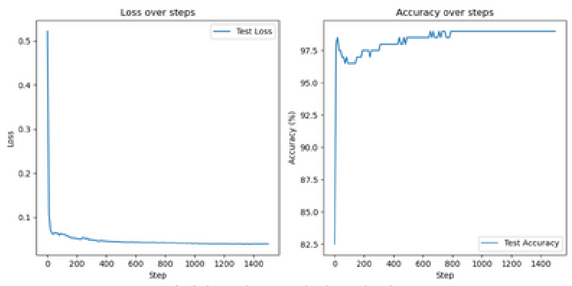
`mini-batch` mode batch size = 32



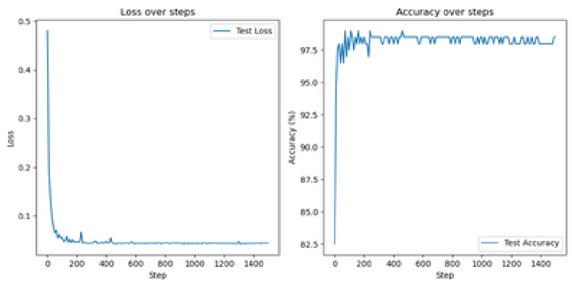
`mini-batch` mode batch size = 64



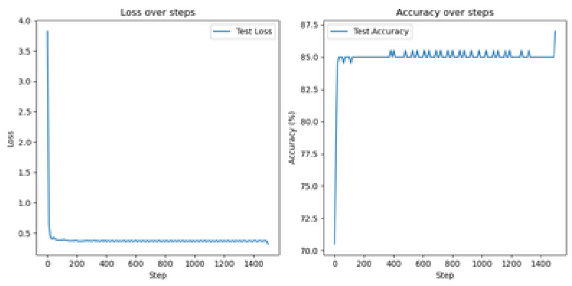
`mini-batch` mode batch size = 128



`mini-batch` mode batch size = 256

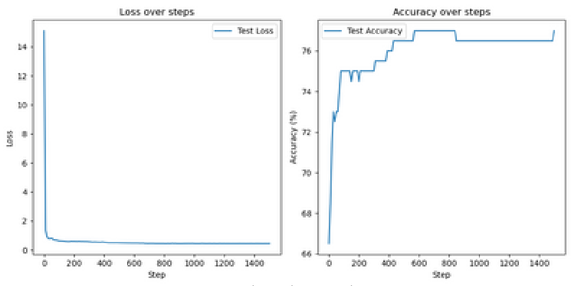


`mini-batch` mode batch size = 512

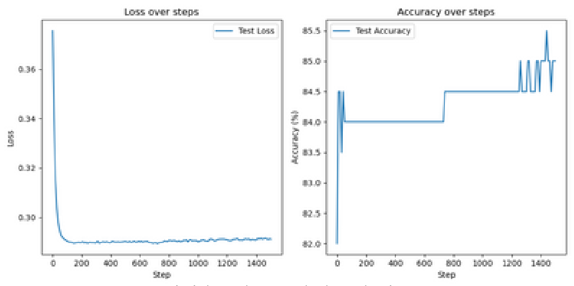


`mini-batch` mode batch size = 1000

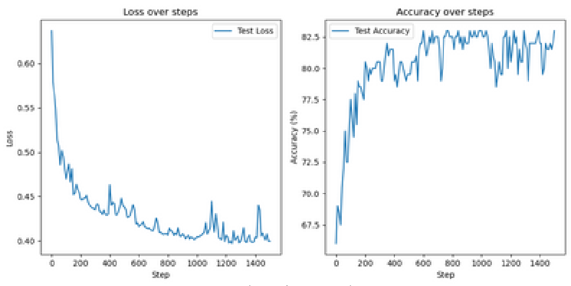
- Scenario 2



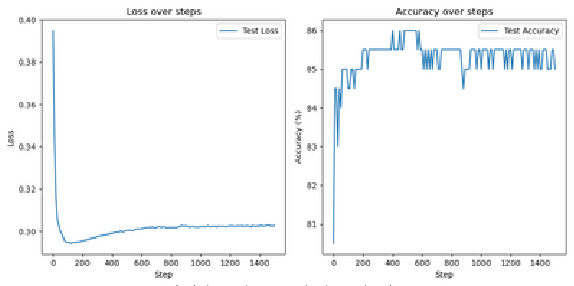
`batch` mode



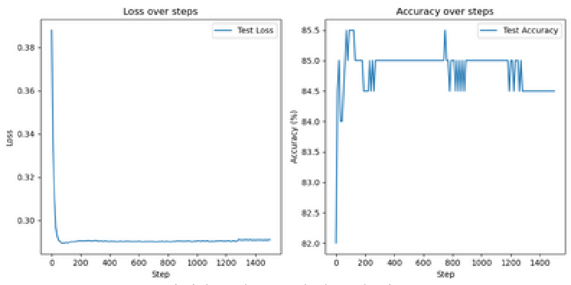
`mini-batch` mode batch size = 8



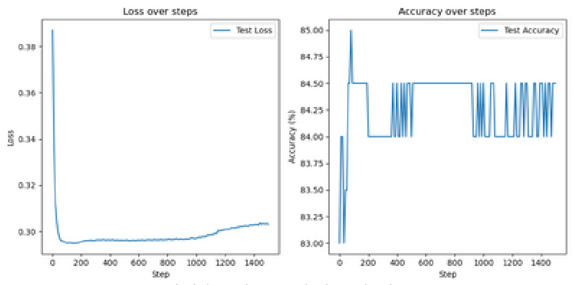
`stochastic` mode



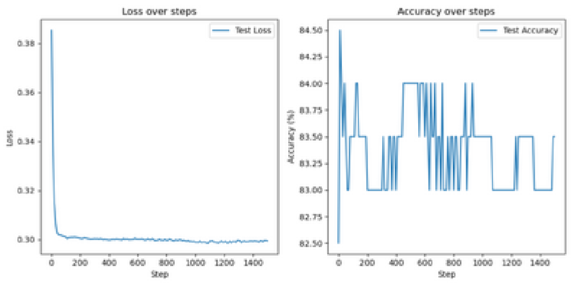
`mini-batch` mode batch size = 16



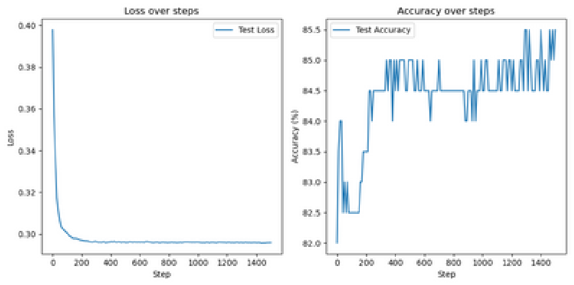
`mini-batch` mode batch size = 1



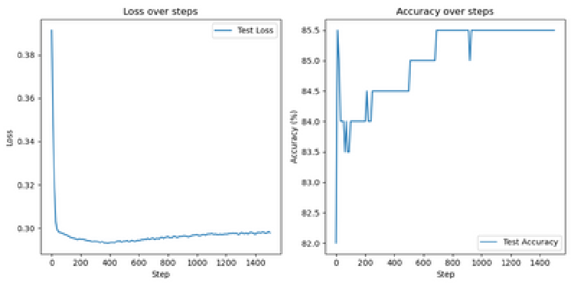
`mini-batch` mode batch size = 32



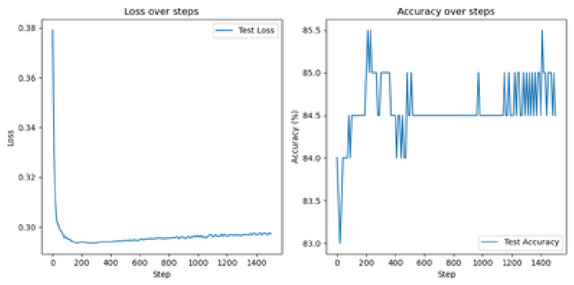
`mini-batch` mode batch size = 2



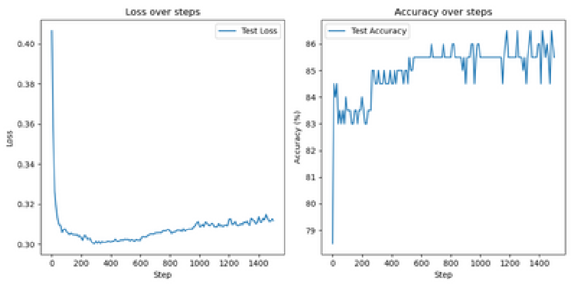
`mini-batch` mode batch size = 64



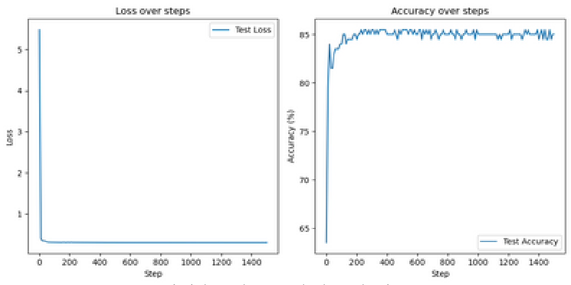
`mini-batch` mode batch size = 4



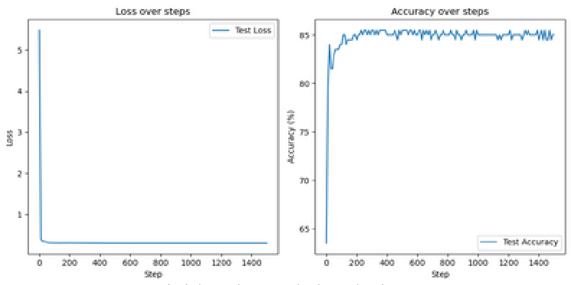
`mini-batch` mode batch size = 128



`mini-batch` mode batch size = 256

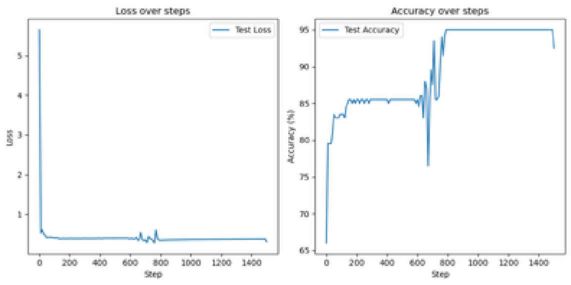


`mini-batch` mode batch size = 512

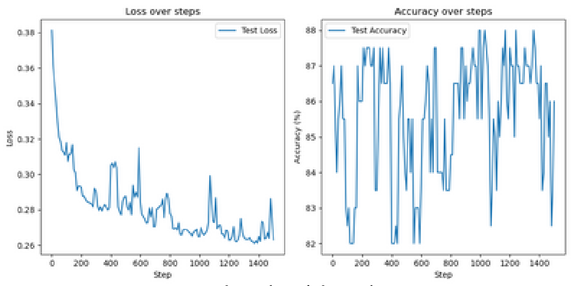


`mini-batch` mode batch size = 1000

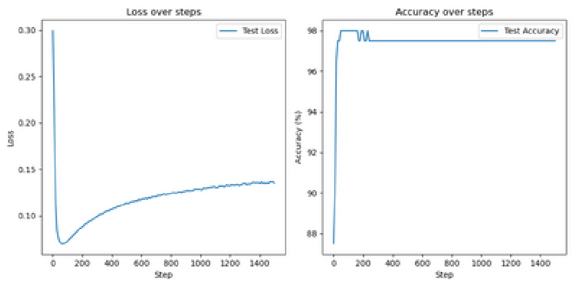
- Scenario 3



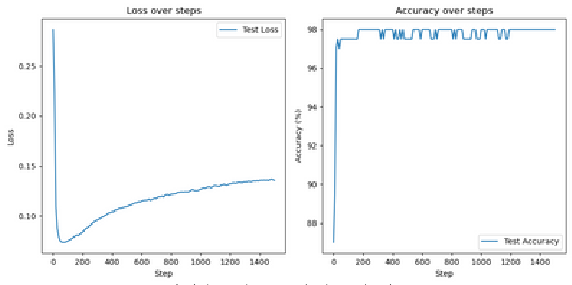
`batch` mode



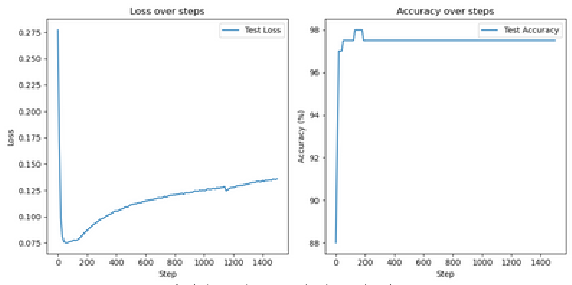
`stochastic` mode



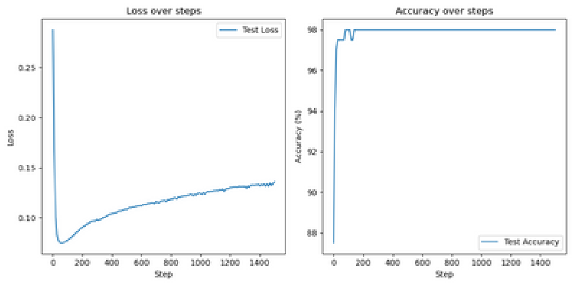
`mini-batch` mode batch size = 1



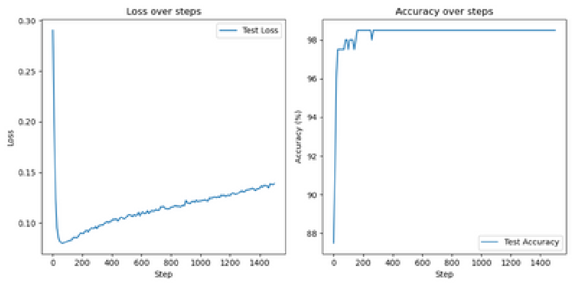
`mini-batch` mode batch size = 2



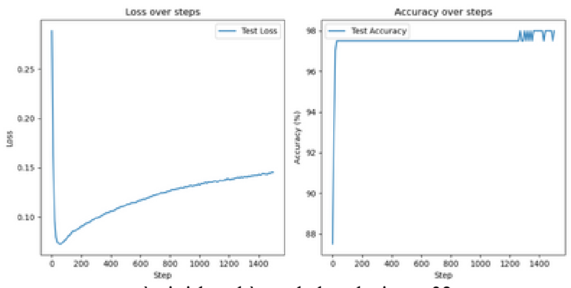
`mini-batch` mode batch size = 4



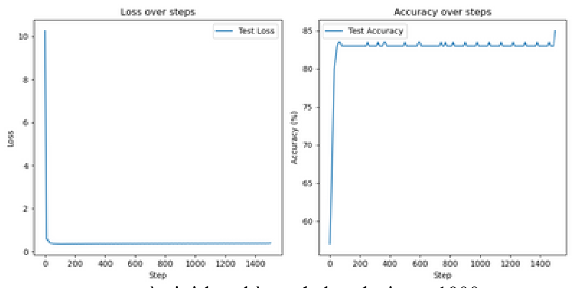
`mini-batch` mode batch size = 8



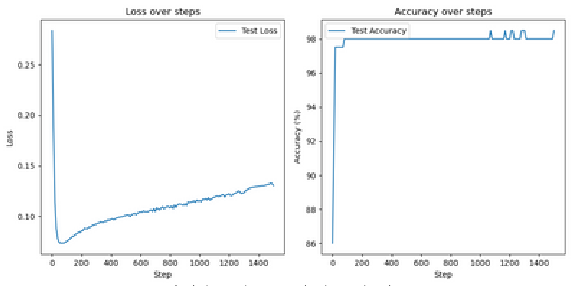
`mini-batch` mode batch size = 16



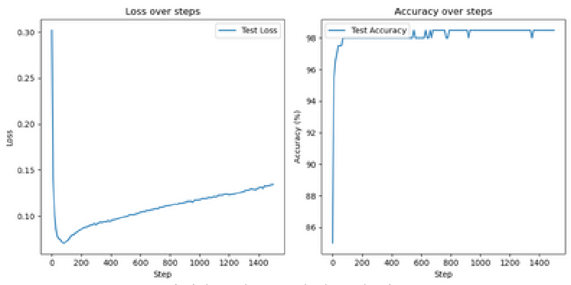
`mini-batch` mode batch size = 32



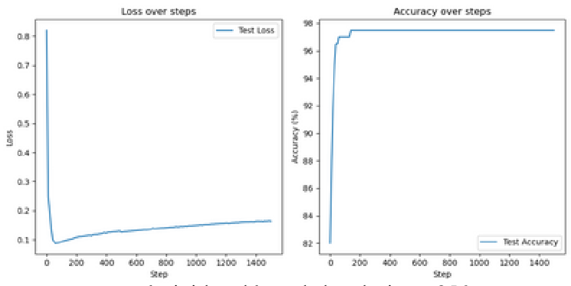
`mini-batch` mode batch size = 1000



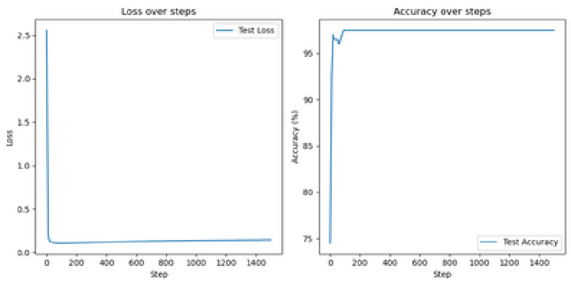
`mini-batch` mode batch size = 64



`mini-batch` mode batch size = 128



`mini-batch` mode batch size = 256



`mini-batch` mode batch size = 512