Deep Learning Practical Work 1-a and 1-b

William KHIEU
Akli Mohamed Ait-Oumeziane

October 29, 2024

Contents

Section 1 - Theoretical Foundation

1.1 Supervised Dataset

1. What are the train, validation, and test sets used for?

Training set:

The training set is used to train the model. It provides the data that the learning algorithm uses to adjust the model's parameters (e.g., weights in a neural network) to minimize the loss function.

- **Model Learning:** The model learns patterns and relationships between the input features $\mathbf{x}^{(i)}$ and the target outputs $\mathbf{y}^{(i)}$ by iteratively adjusting its parameters to fit the training data.
- **Parameter Optimization:** During training, the model optimizes its internal parameters to reduce errors on the training set.

Validation set:

The validation set is used to fine-tune the model's hyperparameters and to evaluate its performance during the training phase without influencing the training process directly.

- **Hyperparameter Tuning:** Helps in selecting the best set of hyperparameters (e.g., learning rate, number of layers, regularization strength) by evaluating different models on the validation set.
- **Model Selection:** Assists in choosing the best-performing model among multiple models trained with different configurations.
- **Preventing Overfitting:** Monitors the model's performance to detect overfitting. If the model performs well on the training set but poorly on the validation set, it may be overfitting.
- **Early Stopping:** Training can be halted when performance on the validation set stops improving, preventing further overfitting.

Test set:

The test set provides an unbiased evaluation of the final model's performance after training and hyperparameter tuning are complete.

- **Performance Assessment:** Measures how well the model generalizes to new, unseen data.
- **Final Evaluation:** Used only once to report the final accuracy, precision, recall, F1 score, or other relevant metrics.
- **No Influence on Training:** The test set should not be used in any part of the model training or selection process to avoid biased evaluations.

2. What is the influence of the number of examples N?

The number of examples N significantly influences a model's ability to learn, generalize, and perform accurately on new data. The impact of N can be summarized as follows:

- **More Data, Better Learning:** Increasing N provides more diverse examples, helping the model learn a richer representation of the underlying data distribution. This exposure to a wider variety of situations during training improves the model's generalization capabilities.
- **Overfitting and Underfitting Mitigation:** A small Ncan lead to overfitting, where the model learns noise and specific patterns in the training set that do not generalize.A larger N helps reduce overfitting, especially in complex models.
- **Statistical Reliability:**
  - **Variance Reduction:** More data points reduce the variance of the model's predictions, leading to more reliable and stable outputs.
  - **Confidence in Predictions:** With a larger N, statistical estimates like means and variances become more accurate, enhancing the model's confidence levels.
- **Computational Trade-offs:** Increasing N leads to longer training times and higher computational costs.
- **Optimal Data Size:** Aim for a sufficient N that captures the data distribution without unnecessary redundancy.

1.2 Network Architecture (Forward)

3. Why is it important to add activation functions between linear transformations?

Adding activation functions between linear transformations is crucial because, without them, the neural network would effectively be equivalent to a single linear transformation, regardless of the number of layers. This is due to the fact that the composition of linear functions results in another linear function. Activation functions introduce non-linearities into the network, enabling it to model complex, non-linear relationships in the data. This non-linearity is essential for the network to capture and learn intricate patterns that cannot be represented by linear transformations alone.

4. What are the sizes n_x, n_h, n_y in Figure 1? In practice, how are these sizes chosen?

In Figure 1, the sizes are as follows:

- n_x: The size of the input layer, representing the number of features in the input vector x.
- n_h: The number of neurons in the hidden layer h.
- n_y: The size of the output layer, corresponding to the output vector $\hat{y}$.

In practice:

- n_x is determined by the dataset and equals the number of input features.
- n_y depends on the problem's requirements:
  - For regression tasks, it may be 1 (predicting a single value) or more.

  - For classification tasks, it typically equals the number of classes.
- n_h is a hyperparameter chosen based on factors like:
  - The complexity of the problem.
  - The amount of training data.
  - Computational resources.
  - It often requires experimentation and tuning to find the optimal size.

5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?

- $\hat{y}$: This vector represents the predicted output from the neural network for a given input x. It is the network's estimation based on its current weights and biases.
- y: This vector represents the true target output, the actual values or labels that we aim to predict.

Difference between $\hat{y}$ and y:

The key difference is that $\hat{y}$ is the model's prediction, while y is the ground truth. Comparing $\hat{y}$ to y allows us to measure the accuracy of the model's predictions and compute loss functions, which are used to update the model's parameters during training.

6. Why use a SoftMax function as the output activation function?

The SoftMax function is used as the output activation function because it converts the raw output scores (logits) from the network into probabilities that sum to 1. This is essential for multi-class classification problems where we want to interpret the output as a probability distribution over the possible classes. SoftMax ensures that each component of the output vector $\hat{y}$ is between 0 and 1 and that the sum of all components equals 1, allowing us to:

- Interpret the outputs probabilistically.
- Use appropriate loss functions like cross-entropy.
- Make informed decisions based on the highest probability.

7. Write the mathematical equations allowing to perform the forward pass of the neural network, i.e., allowing to successively produce $\tilde{h}, h, \tilde{y}$, and $\hat{y}$ starting at x.

The forward pass of the neural network involves the following mathematical equations:

(a) **Compute the affine transformation for the hidden layer:**

$$\tilde{h} = xW_h^T + b_h$$

where:

- W_h: Weight matrix of size $n_h \times n_x$.
- b_h: Bias vector of size n_h.

(b) **Apply the tanh activation function:**

$$h = \tanh(\tilde{h})$$

where:

- h: Output vector of the hidden layer of size n_h.

(c) **Compute the affine transformation for the output layer:**

$$\tilde{y} = hW_y^T + b_y$$

where:

- W_y: Weight matrix of size $n_y \times n_h$.
- b_y: Bias vector of size n_y.

(d) **Apply the SoftMax activation function:**

$$\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_{j=1}^{n_y} \exp(\tilde{y}_j)} \quad \text{for } i = 1, \ldots, n_y$$

where:

- $\hat{y}$: Final output vector of size n_y representing probabilities.

These equations define the step-by-step computation from the input x to the output $\hat{y}$ through the network layers.

1.3 Loss Function

8. During training, we try to minimize the loss function. For cross-entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$?

(a) **For Cross-Entropy Loss (Classification Tasks):**

y_i represents the true probability distribution of the classes (often one-hot encoded), and $\hat{y}_i$ is the predicted probability distribution from the SoftMax output.

Minimizing Loss:

- **Increase $\hat{y}_i$ for the correct class**: For the class where $y_i = 1$ (the true class), increasing $\hat{y}_i$ (the predicted probability for that class) will decrease $-\log \hat{y}_i$, thus lowering the loss.
- **Decrease $\hat{y}_i$ for incorrect classes**: For classes where $y_i = 0$, their contributions to the loss are zero. However, since the probabilities must sum to 1 (due to SoftMax), decreasing $\hat{y}_i$ for incorrect classes allows increasing $\hat{y}_i$ for the correct class.

Overall Goal: The network adjusts $\hat{y}_i$ to make the predicted probability distribution match the true distribution, with the highest probability assigned to the correct class.

(b) **For Mean Squared Error (MSE) Loss (Regression Tasks):** This loss measures the squared difference between each predicted value $\hat{y}_i$ and the true target y_i.

Minimizing Loss:

- **Reduce the difference $y_i - \hat{y}_i$**: To decrease the loss, the network must adjust $\hat{y}_i$ to be as close as possible to y_i for each output dimension.

- **Symmetric Penalization**: Both overestimations and underestimations are penalized equally due to the squared term.

Overall Goal: The network aims to minimize the average squared error between predictions and targets, effectively bringing the predicted outputs $\hat{y}_i$ closer to the true values y_i.

9. How are these functions better suited to classification or regression tasks?

(a) **Why Cross-Entropy is Suited for Classification Tasks**

- **Nature of Outputs**: In classification, outputs represent class probabilities.
- **SoftMax Compatibility**: Cross-entropy works seamlessly with the SoftMax activation function, which outputs probability distributions over classes.
- **Measuring Distribution Differences**: Cross-entropy quantifies the divergence between the true distribution y and the predicted distribution $\hat{y}$.
- **Penalizing Misclassifications**: Assigning a high probability to the wrong class results in a higher loss.
- **Encouraging Correct Confidence**: The loss decreases when the model is confident and correct.

(b) **Why Mean Squared Error is Suited for Regression Tasks**

- **Nature of Outputs**: Regression tasks involve predicting continuous numerical values.
- **Direct Measurement**: MSE directly measures the average squared difference between predicted and actual values.
- **Quadratic Penalization**: The squared term heavily penalizes larger errors, encouraging the model to avoid significant deviations from the target.

Cross-Entropy is better suited for classification because it effectively handles discrete class labels and probability distributions, penalizing the model based on the likelihood of the correct class.
Mean Squared Error is better suited for regression because it measures the average squared difference between continuous target values and predictions, providing a direct assessment of prediction accuracy in numerical terms.

1.4 Optimization Algorithm

10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic, and online stochastic versions? Which one seems the most reasonable to use in the general case?

In the general case, **Mini-Batch Stochastic Gradient Descent** is often the most reasonable to use because it offers a balance between computational efficiency and convergence stability. It can handle large datasets efficiently without the high variance associated with online SGD and without the computational overhead of full-batch gradient descent.

Table 1: Advantages and Disadvantages of Gradient Descent Variants

Variant	Advantages	Disadvantages
Classic Gradient Descent (Full Batch)	• Deterministic convergence: stable and smooth updates. • Stable updates: loss decreases monotonically. • Possibility of using larger learning rates due to accurate gradients.	• Computationally intensive: slow with large datasets. • Memory constraints: entire dataset must be in memory. • Not scalable: impractical for large-scale tasks.
Mini-Batch Stochastic Gradient Descent	• Computational efficiency: faster updates than full batch. • Scalability: suitable for large datasets. • Efficient hardware utilization: leverages vectorization and parallel computing. • Convergence stability: balances noisy updates and efficiency.	• Noisy gradient estimates: introduces some noise in updates. • Hyperparameter sensitivity: requires careful tuning. • Potential for suboptimal convergence due to stochastic nature.
Online Stochastic Gradient Descent (Per-Example Updates)	• Fast iterations: quick updates processing one example at a time. • Online learning capability: suitable for streaming data. • Minimal memory usage: ideal for extremely large datasets.	• High variance in updates: leads to noisy and unstable updates. • Slow overall convergence: may require more iterations. • Inefficient hardware utilization: doesn't exploit batch operations.

11. What is the influence of the learning rate η on learning?

Effects of Different Learning Rate Values

(a) **Learning Rate Too Large**

- **Overshooting Minima:** A large η can cause the optimizer to take too big steps, potentially overshooting the minimum of the loss function repeatedly without settling down.
- **Divergence:** The loss may start increasing instead of decreasing, leading to divergence where the model fails to learn anything meaningful.
- **Instability:** The training process becomes erratic, with significant fluctuations in the loss value, making it hard to converge to a good solution.
- **Possible Explosions:** In some cases, especially with complex neural networks, weights can grow uncontrollably large, causing numerical instability.

(b) **Learning Rate Too Small**

- **Slow Convergence:** A small η results in tiny updates to the weights, making the learning process very slow. It may take an impractical amount of time to reach a satisfactory performance level.
- **Getting Stuck in Local Minima:** The optimizer might get trapped in local minima or saddle points because the small steps prevent it from escaping shallow regions of the loss surface.

- **Resource Inefficiency:** Consumes more computational resources and time due to the increased number of iterations required to converge.
- **Sensitivity to Noise:** In stochastic methods, small learning rates can make the optimizer more sensitive to the noise inherent in gradient estimates from mini-batches.

Strategies for Managing the Learning Rate

- **Learning Rate Schedules :** Time-Based Decay / Step Decay / Exponential Decay reduce η over time.
- **Adaptive Learning Rates :** Algorithms like AdaGrad, RMSProp, and Adam adjust the learning rate during training for each parameter individually based on the historical gradients.
- **Learning Rate Warm-up and Cool-down**

Practical Considerations

- **Hyperparameter Tuning:** The optimal learning rate often depends on the specific problem, architecture, and dataset. Empirical testing and techniques like grid search or random search can help find a suitable η.
- **Batch Size Interaction:** There's an interplay between batch size and learning rate. Larger batch sizes can often accommodate larger learning rates.

The optimal choice of η depends on the topology of the loss function, the size of the dataset, and the network architecture. A good η allows a fast and stable convergence to a global minimum or a good local minimum.

12. Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss with respect to the parameters, using the naive approach and the backpropagation algorithm.

The comparison of gradient calculation complexity between the naïve approach and the backpropagation algorithm is as follows:

- **Naïve Approach**:
  - Complexity: $O(N \times L)$
  - N: number of parameters in the network
  - L: number of layers
  - Each parameter requires a separate gradient calculation across all layers
- **Backpropagation**:
  - Complexity: $O(N)$
  - Computes gradients in a single backward pass through the network
  - Reuses intermediate calculations, avoiding redundancy

Explanation of the difference:

- The naïve approach recalculates the same gradients multiple times for each parameter.

- Backpropagation leverages the chain structure of the network to efficiently propagate gradients.
- For deep networks (large L), the performance difference is particularly significant.

Backpropagation is therefore much more efficient, especially for deep networks, which explains its universal use in training neural networks.

13. What criteria must the network architecture meet to allow such an optimization procedure?

To enable an efficient optimization process, the network architecture must meet several important criteria:

- **Differentiability**:
  - All functions in the network must be differentiable with respect to their parameters.
  - This enables the calculation of gradients necessary for backpropagation.
  - Exceptions: some non-differentiable functions like ReLU are tolerated because they have well-defined sub-gradients.
- **Continuity**:
  - Functions should be continuous to ensure smooth gradient variation.
  - This facilitates convergence in the optimization process.
- **Gradient Propagation**:
  - The architecture should allow for efficient gradient propagation through all layers.
  - Avoid vanishing or exploding gradient problems.
- **Parametrization**:
  - Each layer should have adjustable parameters (weights, biases).
  - These parameters must directly influence the network's output.
- **Proper Connectivity**:
  - Connections between layers should enable adequate information flow.
  - Avoid bottlenecks that could limit learning capacity.
- **Numerical Stability**:
  - Operations should be numerically stable to avoid computational issues.
  - Techniques like batch normalization can help maintain stability.

These criteria ensure that the network can be effectively optimized using gradient-based methods, such as stochastic gradient descent.

14. The SoftMax function and the cross-entropy loss are often used together, and their gradient is very simple. Show that the loss can be simplified by:

$$\ell = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right)$$

The simplification of the loss function (cross-entropy) combined with the SoftMax function is as follows:

$$\text{Cross-entropy:} \quad \ell = -\sum_{i=1}^{n_y} y_i \log(\hat{y}_i)$$

$$\text{SoftMax:} \quad \hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}}$$

By substituting SoftMax into the cross-entropy:

$$\begin{aligned}
\ell &= -\sum_{i=1}^{n_y} y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}}\right) \\
&= -\sum_{i=1}^{n_y} y_i \left(\log(e^{\tilde{y}_i}) - \log\left(\sum_{j=1}^{n_y} e^{\tilde{y}_j}\right)\right) \\
&= -\sum_{i=1}^{n_y} y_i \tilde{y}_i + \sum_{i=1}^{n_y} y_i \log\left(\sum_{j=1}^{n_y} e^{\tilde{y}_j}\right)
\end{aligned}$$

Since $\sum_{i=1}^{n_y} y_i = 1$ (one-hot encoding), we finally obtain:

$$\ell = -\sum_{i=1}^{n_y} y_i \tilde{y}_i + \log\left(\sum_{j=1}^{n_y} e^{\tilde{y}_j}\right) \tag{1}$$

This simplified form is numerically more stable and more efficient to compute.

15. Write the gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}$.

$$\frac{\partial \ell}{\partial \tilde{y}_i} = \ldots \quad \Rightarrow \quad \nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \ldots$$

The gradient of the loss with respect to $\tilde{y}_i$ is calculated as follows:

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{y}_i} &= \frac{\partial}{\partial \tilde{y}_i}\left[-\sum_{k=1}^{n_y} y_k \tilde{y}_k + \log\left(\sum_{j=1}^{n_y} e^{\tilde{y}_j}\right)\right] \\
&= -y_i + \frac{\partial}{\partial \tilde{y}_i}\log\left(\sum_{j=1}^{n_y} e^{\tilde{y}_j}\right) \\
&= -y_i + \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \\
&= -y_i + \hat{y}_i
\end{aligned}$$

Thus, for all i, we obtain the gradient vector:

$$\nabla_{\tilde{y}} \ell = \hat{y} - y \tag{2}$$

This simple form of the gradient is one of the reasons why the combination of SoftMax and cross-entropy is so popular in deep learning for classification tasks.

16. Using backpropagation, write the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y}\ell$. Note that writing this gradient uses $\nabla_{\tilde{y}}\ell$. Do the same for $\nabla_{b_y}\ell$.

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \dots \quad \Rightarrow \quad \nabla_{W_y}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_y n_h}} \end{bmatrix} = \dots$$

Let's calculate the gradients of the loss with respect to the weights W_y and biases b_y of the output layer:

- **Gradient with respect to W_y**:

$$\begin{aligned}
\frac{\partial \ell}{\partial W_{y,ij}} &= \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} \\
&= \sum_{k=1}^{n_y} (\hat{y}_k - y_k)\frac{\partial}{\partial W_{y,ij}}(W_y h)_k \\
&= (\hat{y}_i - y_i) h_j
\end{aligned}$$

In matrix notation:

$$\nabla_{W_y}\ell = (\hat{y} - y)h^T \tag{3}$$

- **Gradient with respect to b_y**:

$$\begin{aligned}\frac{\partial \ell}{\partial b_{y,i}} &= \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,i}} \\ &= \sum_{k=1}^{n_y} (\hat{y}_k - y_k) \frac{\partial}{\partial b_{y,i}} (W_y h + b_y)_k \\ &= \hat{y}_i - y_i\end{aligned}$$

  In vector notation:

$$\nabla_{b_y} \ell = \hat{y} - y \tag{4}$$

These gradients are used to update the weights and biases of the output layer during gradient descent.

17. Compute other gradients: $\nabla_{\tilde{h}} \ell$, $\nabla_{W_h} \ell$, $\nabla_{b_h} \ell$.

Now, let's calculate the gradients for the hidden layer:

- **Gradient with respect to $\tilde{h}$**:

$$\begin{aligned}\nabla_{\tilde{h}} \ell &= \frac{\partial \ell}{\partial h} \frac{\partial h}{\partial \tilde{h}} \\ &= W_y^T (\hat{y} - y) \odot (1 - h^2)\end{aligned}$$

  where $\odot$ represents the Hadamard product (element-wise product).

- **Gradient with respect to W_h**:

$$\frac{\partial \ell}{\partial W_h} = \nabla_{\tilde{h}} \ell \cdot x^T \tag{5}$$

- **Gradient with respect to b_h**:

$$\frac{\partial \ell}{\partial b_h} = \nabla_{\tilde{h}} \ell \tag{6}$$

Explanation:

- The gradient with respect to $\tilde{h}$ is obtained by applying the chain rule. $W_y^T(\hat{y} - y)$ propagates the error from the output layer back to the hidden layer, and $(1 - h^2)$ is the derivative of the tanh activation function.
- For W_h, we multiply the gradient of $\tilde{h}$ by x^T because $\tilde{h} = W_h x + b_h$.
- The gradient with respect to b_h is directly $\nabla_{\tilde{h}} \ell$ since b_h is added directly to $\tilde{h}$.

These gradients allow us to update all the network parameters during backpropagation.

Section 2 - Implementation

2.1 Forward and Backward Manuals

1. Discuss and analyze your experiments following the implementation. Provide pertinent figures showing the evolution of the loss; effects of different batch size / learning rate, etc.

Experimental Analysis

(a) **Overview of Experiments**

In our experiments, we evaluated the impact of different batch sizes and learning rates on the model's performance and training efficiency. The experiments involved testing combinations of batch sizes of 1, 10, 50, 100, and 200 with learning rates of 0.01, 0.05, 0.1, 0.5, 1, and 5. Here, we analyze the results, discussing both the trends observed in training and the implications on model performance. Relevant figures are referenced to visualize the evolution of the loss and highlight the impact of these hyperparameters.

(b) **Observed Trends in Loss Evolution and Convergence**

From the figures in the results section, we observed that aligning the learning rate with the batch size tended to yield better convergence and stability. Specifically:

i. **Small Batch Sizes with Small Learning Rates**: For batch sizes of 1 and 10, a lower learning rate (0.01 to 0.1) was necessary to achieve smooth convergence. For example, with a batch size of 1 and a learning rate of 0.01, the final train accuracy reached 97%, and the test accuracy was 93.5%, with a stable loss of approximately 0.092 on the training data. In contrast, using a high learning rate (e.g., 5) with a small batch size resulted in instability or, divergence with the loss reaching infinity (see Fig. 6 and 7), indicating an inability of the model to converge.

ii. **Large Batch Sizes with Higher Learning Rates**: As the batch size increased to 50, 100, and 200, larger learning rates (0.5 to 1) produced stable and faster convergence. For example, with a batch size of 100 and a learning rate of 1, we achieved a final train accuracy of 97.5% and test accuracy of 93.5%, with a low final training loss of 0.093 (see Fig. 20). A batch size too high (200) combined with too low of a learning rate (e.g., 0.01) led to slower convergence, as seen by prolonged training times and convergence towards local minima rather than the global minimum.

(c) **Effects of Different Batch Sizes and Learning Rates on Accuracy and Loss**

The data shows that both **training and test accuracies** increased when the batch size and learning rate were appropriately matched. Highlights from specific configurations include:

i. **Batch Size 10 with Learning Rate 0.1**: This combination provided a good balance of accuracy and convergence speed, achieving a final training accuracy of 98% and a test accuracy of 93%, with a training loss of 0.092.

ii. **Batch Size 50 with Learning Rate 0.5**: This setting achieved a high test accuracy of 93.5% with a low final training loss (0.092) and converged efficiently,

indicating that larger batches paired with moderate learning rates generally optimize both accuracy and speed.

(d) **General Observations and Conclusions**

i. **Small Batch Sizes and Learning Rates**: When using smaller batch sizes, a lower learning rate was critical to achieving convergence. High learning rates caused erratic loss behavior, leading to divergence or fluctuating loss values (e.g., learning rate of 5 with batch size 1 caused NaN values in the loss).

ii. **Large Batch Sizes with Moderate to High Learning Rates**: Larger batch sizes with matching higher learning rates allowed the model to converge more quickly and reach lower final loss values, as seen in batch size 200 with learning rate 1. This combination facilitated stable convergence and minimized training time, demonstrating that larger batches benefit from larger step sizes in gradient descent.

iii. **Optimal Range of Learning Rate per Batch Size**: Our results suggest that maintaining a learning rate in a range close to the batch size (e.g., batch size 10 with learning rate 0.1) provides a balanced approach, maximizing accuracy while avoiding divergence or lengthy convergence times.

In conclusion, our experiments show that selecting an appropriate learning rate relative to the batch size significantly impacts model convergence and performance. The relationship between batch size and learning rate should be carefully tuned, as an excessively high or low learning rate for a given batch size can lead to poor model generalization or unstable training behavior.

2. Write the function `init_params(nx, nh, ny)` which initializes the weights of a network from the sizes n_x, n_h, n_y and stores them in a dictionary. All weights will be initialized according to a normal distribution of mean 0 and standard deviation 0.3.

```
def init_params(nx, nh, ny):
    params = {}
    params["Wh"] = torch.randn(nh, nx) / math.sqrt(0.3)
    params["Wy"] = torch.randn(ny, nh) / math.sqrt(0.3)
    params["bh"] = torch.randn(1, nh) / math.sqrt(0.3)
    params["by"] = torch.randn(1, ny) / math.sqrt(0.3)
    return params
```

Listing 1: Implementation of init_params

3. Write the function `forward(params, X)` which calculates the intermediate steps and the output of the network from an input batch `X` of size `nbatch` $\times n_x$ and weights stored in `params` and store them in a dictionary. We return the dictionary of intermediate steps and the output $\hat{Y}$ of the network.

```
def forward(params, X):
    outputs = {}
    outputs["X"] = X
    outputs["htilde"] = torch.mm(X, params["Wh"].t()) + params["bh"]
    outputs["h"] = torch.tanh(outputs["htilde"])
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].t()) + params["by
        "]
    outputs["yhat"] = torch.softmax(outputs["ytilde"], dim=1)
```

```
8       return outputs["yhat"], outputs
```

Listing 2: Implementation of forward

4. Write the function `loss_accuracy(Y_hat, Y)` which computes the cost function and the precision (rate of good predictions) from an output matrix $\hat{Y}$ (output of `forward`) with respect to a ground truth matrix `Y` of the same size, and returns the loss L and the precision `acc`.

Note: Use the function `_, indsY = torch.max(Y, 1)` which returns the index of the predicted class (or to be predicted) for each example.

```
1  def loss_accuracy(Y_hat, Y):
2      L = -torch.mean(torch.sum(Y * torch.log(Yhat), dim=1))
3      acc = torch.mean((torch.argmax(Yhat, dim=1) == torch.argmax(Y, dim=1)).
           float())*100
4      return L, acc
```

Listing 3: Implementation of loss_accuracy

5. Write the function `backward(params, outputs, Y)` which calculates the gradients of the loss with respect to the parameters and stores them in a dictionary.

```
1  def backward(params, outputs, Y):
2      grads = {}
3      bsize = Y.shape[0]
4      delta_y = outputs["yhat"] - Y
5      delta_h = torch.mm(delta_y , params["Wy"]) * (1 - outputs["h"] ** 2)
6      grads["Wy"] = torch.mm(delta_y.t(), outputs["h"]) / bsize
7      grads["Wh"] = torch.mm(delta_h.t(), outputs["X"]) / bsize
8      grads["by"] = torch.mean(delta_y, dim=0, keepdim=True)
9      grads["bh"] = torch.mean(delta_h, dim=0, keepdim=True)
10     return grads
```

Listing 4: Implementation of backward

6. Write the function `sgd(params, grads, eta)` which applies a stochastic gradient descent by mini-batch and updates the network parameters from their gradients and the learning rate η.

```
1  def sgd(params, grads, eta):
2      params["Wh"] -= eta * grads["Wh"]
3      params["Wy"] -= eta * grads["Wy"]
4      params["bh"] -= eta * grads["bh"]
5      params["by"] -= eta * grads["by"]
6      return params
```

Listing 5: Implementation of sgd

7. Write the global learning algorithm using these functions.

```
data = CirclesData()
data.plot_data()
N = data.Xtrain.shape[0]
Nbatch = 10
nx = data.Xtrain.shape[1]
nh = 10
ny = data.Ytrain.shape[1]
eta = 0.03

params = init_params(nx, nh, ny)

curves = [[], [], [], []]

# epoch
for iteration in range(150):

    # permute
    perm = np.random.permutation(N)
    Xtrain = data.Xtrain[perm, :]
    Ytrain = data.Ytrain[perm, :]

    # batches
    for j in range(N // Nbatch):

        indsBatch = range(j * Nbatch, (j + 1) * Nbatch)
        X = Xtrain[indsBatch, :]
        Y = Ytrain[indsBatch, :]

        # forward propagation
        Yhat, outputs = forward(params, X)

        # calculate loss and accuracy
        L, acc = loss_accuracy(Yhat, Y)

        # backward propagation to compute gradients
        grads = backward(params, outputs, Y)

        # update parameters using SGD
        params = sgd(params, grads, eta)

    Yhat_train, _ = forward(params, data.Xtrain)
    Yhat_test, _ = forward(params, data.Xtest)
    Ltrain, acctrain = loss_accuracy(Yhat_train, data.Ytrain)
    Ltest, acctest = loss_accuracy(Yhat_test, data.Ytest)
    Ygrid, _ = forward(params, data.Xgrid)

    title = "Iter {}: Acc train {:.1f}% ({:.2f}), acc test {:.1f}% ({:.2f})".
        format(
        iteration, acctrain, Ltrain, acctest, Ltest
    )
    print(title)
    data.plot_data_with_grid(Ygrid, title)

    # Record the metrics
    curves[0].append(acctrain.item())
    curves[1].append(acctest.item())
    curves[2].append(Ltrain.item())
    curves[3].append(Ltest.item())
```

Listing 6: Global Learning Algorithm

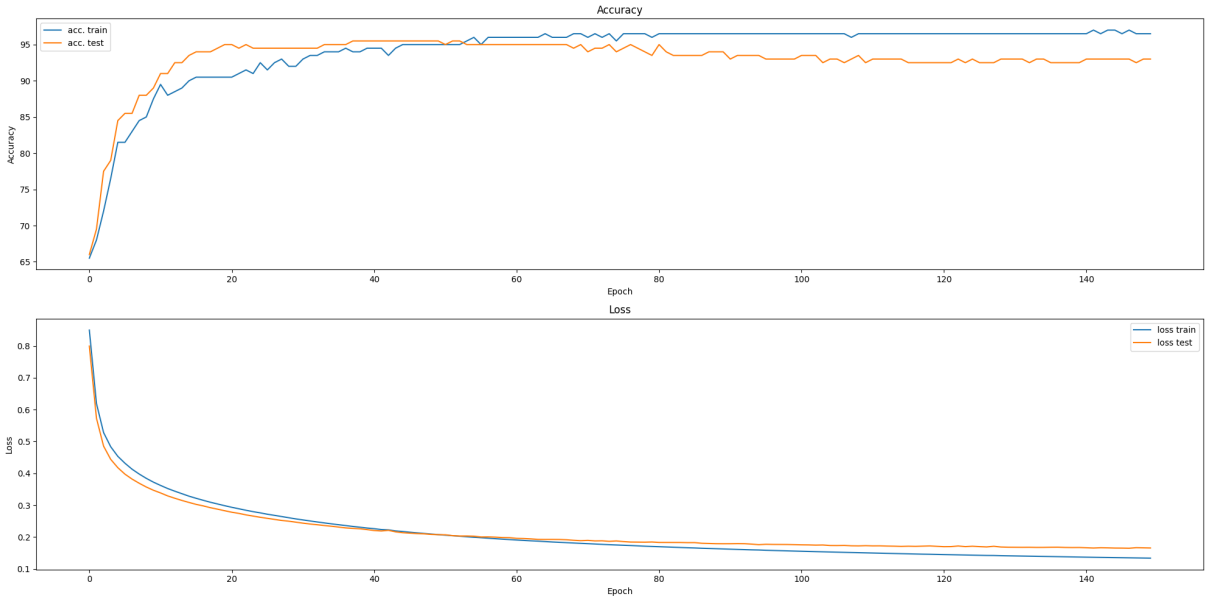

Figure 1: Accuracy and Loss Curve

```python
fig, axs = plt.subplots(2, 1, figsize=(20, 10))

# Plot accuracy curves on the first subplot
axs[0].plot(curves[0], label="acc. train")
axs[0].plot(curves[1], label="acc. test")
axs[0].set_title('Accuracy')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Accuracy')
axs[0].legend()

# Plot loss curves on the second subplot
axs[1].plot(curves[2], label="loss train")
axs[1].plot(curves[3], label="loss test")
axs[1].set_title('Loss')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Loss')
axs[1].legend()

# Show the plot
plt.tight_layout()
plt.show()
```

Listing 7: Loss and Accuracy dynamics

2.2 Figures

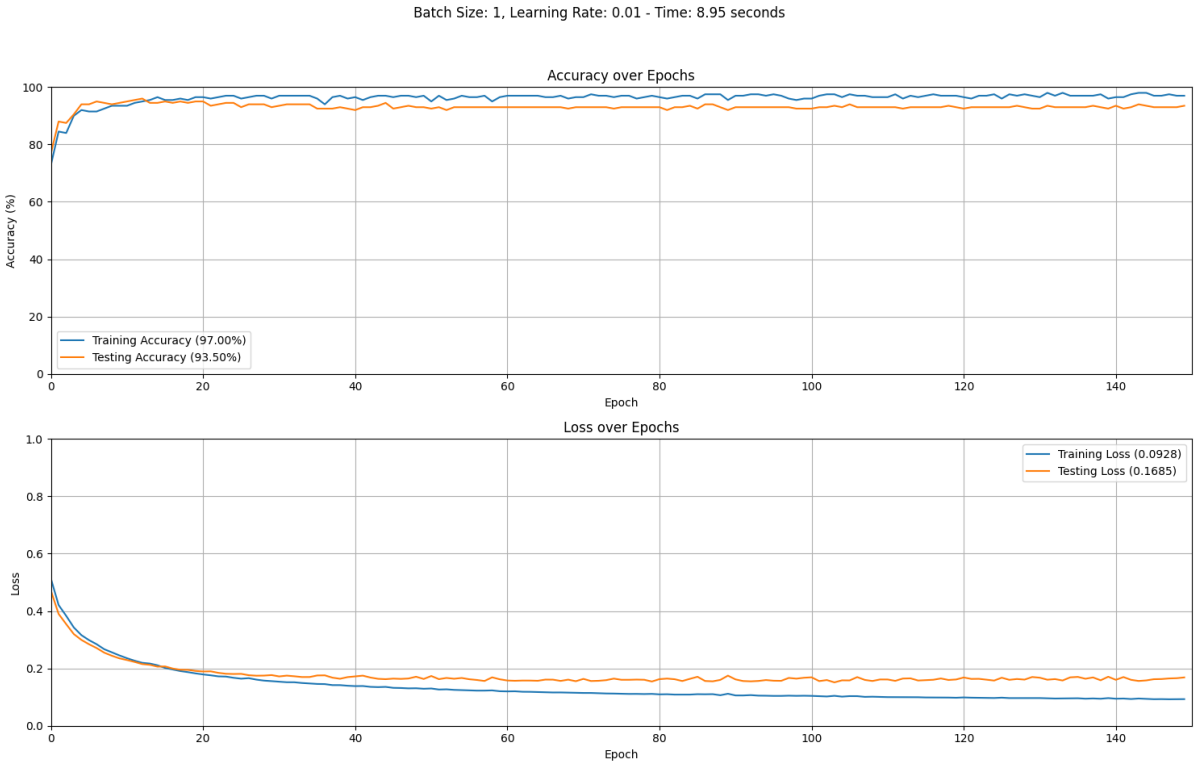

Figure 2: Batch Size: 1, Learning Rate: 0.01

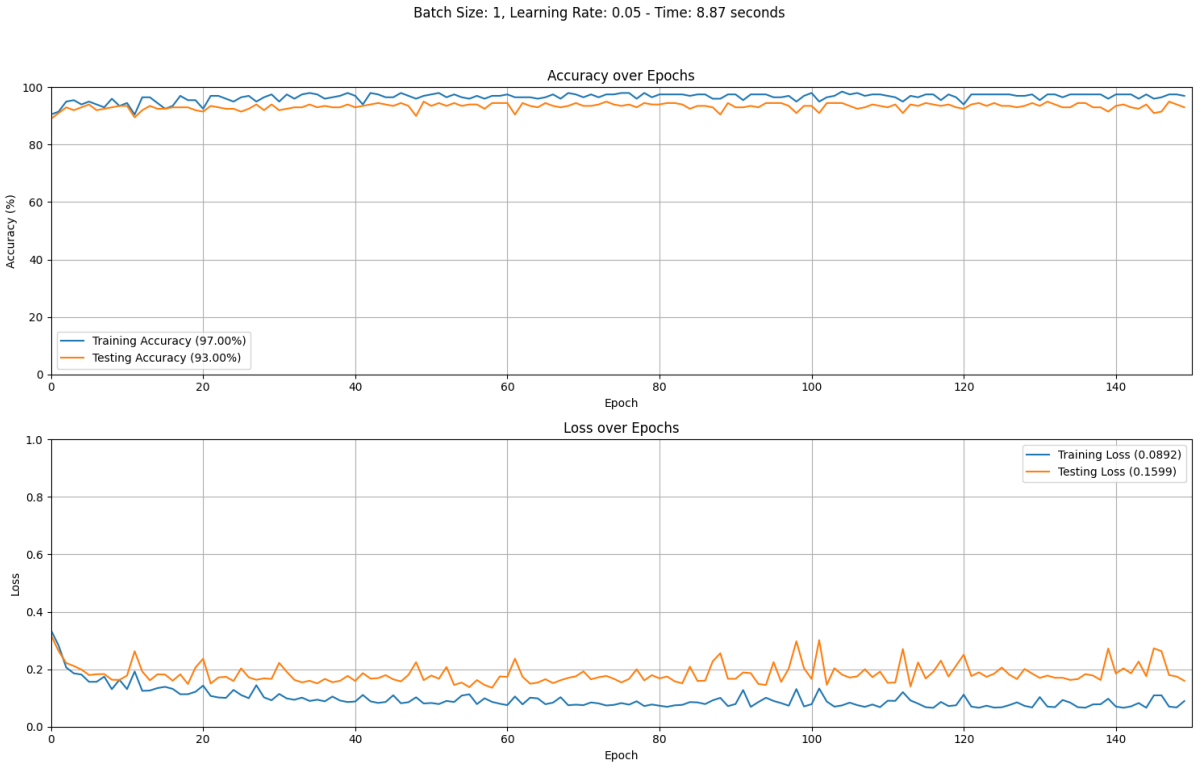

Figure 3: Batch Size: 1, Learning Rate: 0.05

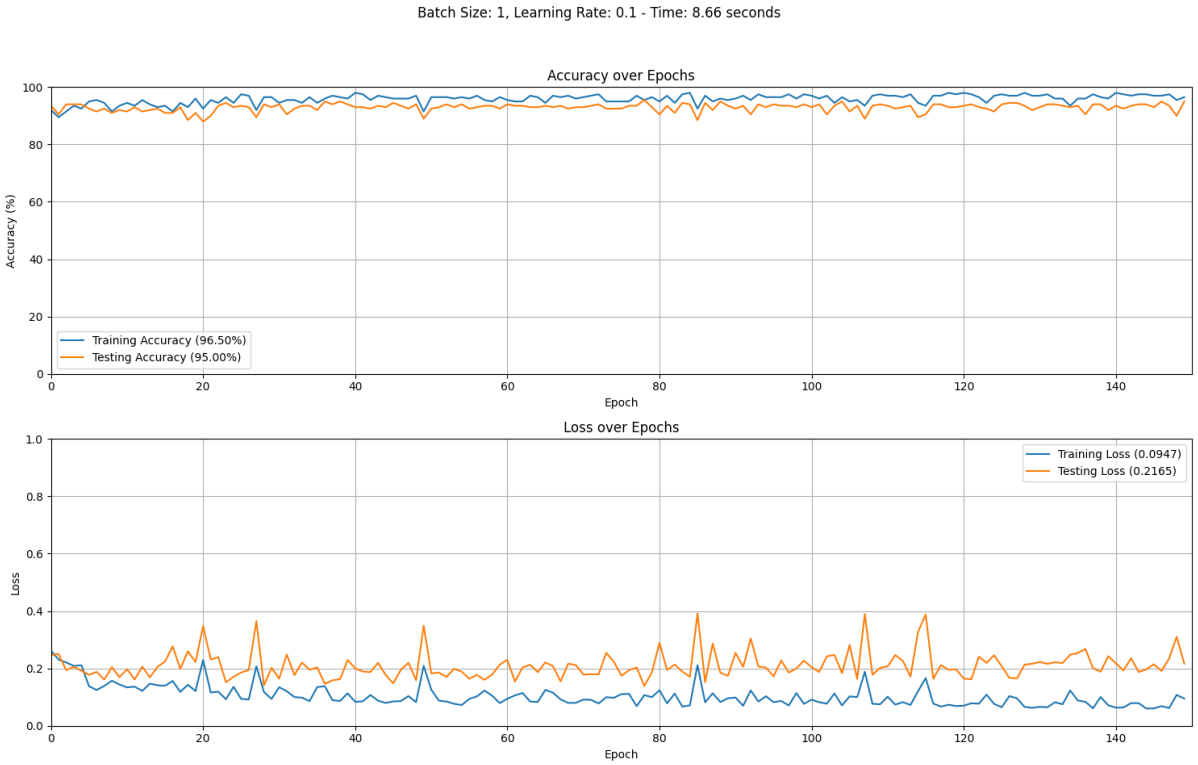

Figure 4: Batch Size: 1, Learning Rate: 0.1

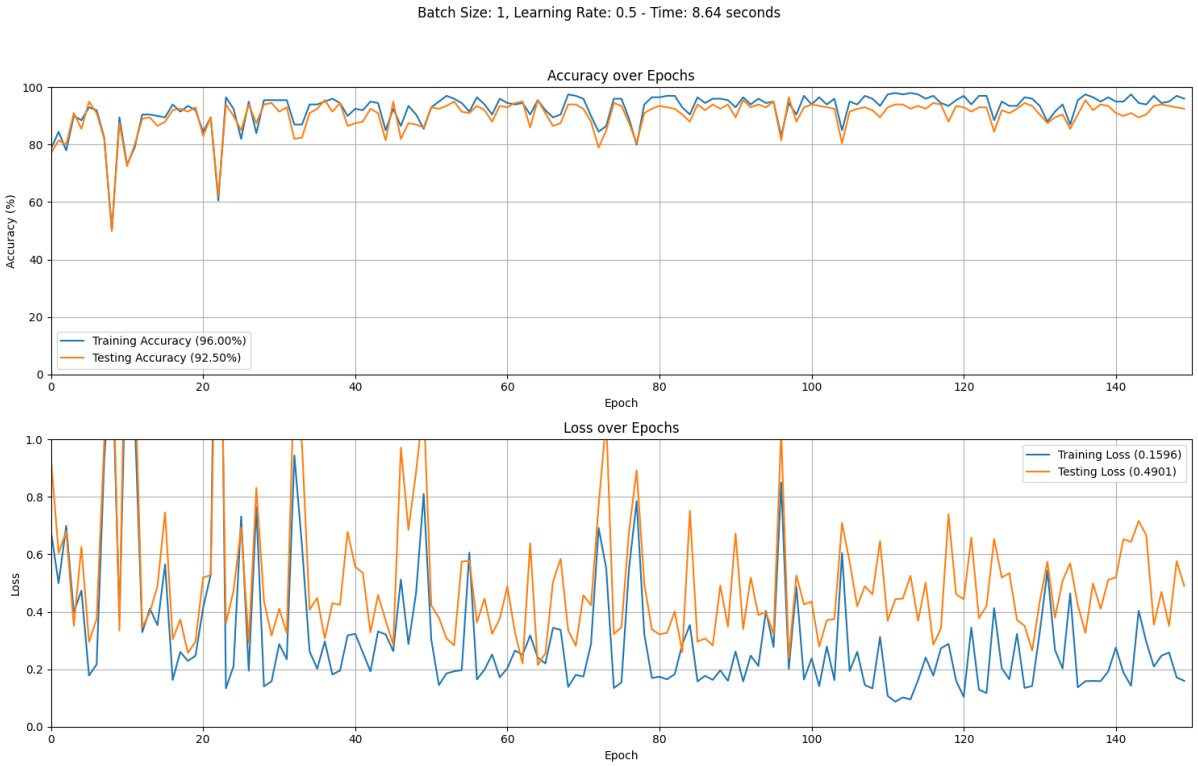

Figure 5: Batch Size: 1, Learning Rate: 0.5

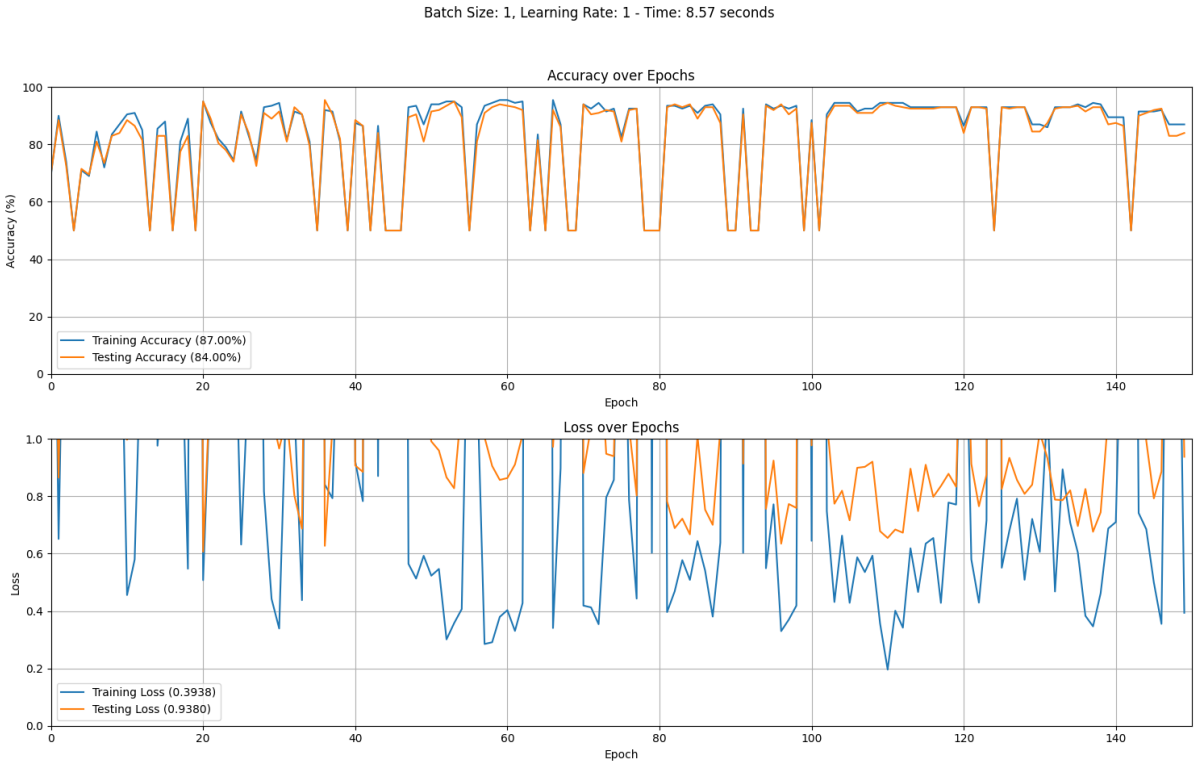

Figure 6: Batch Size: 1, Learning Rate: 1

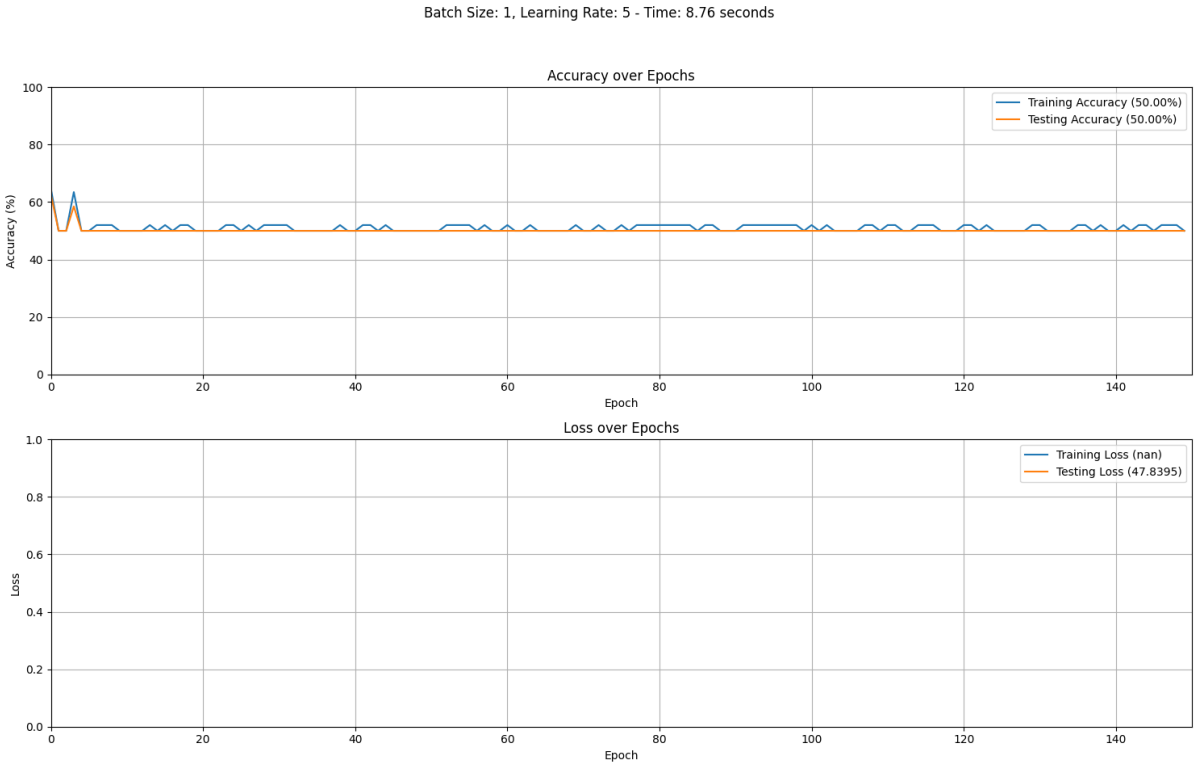

Figure 7: Batch Size: 1, Learning Rate: 5

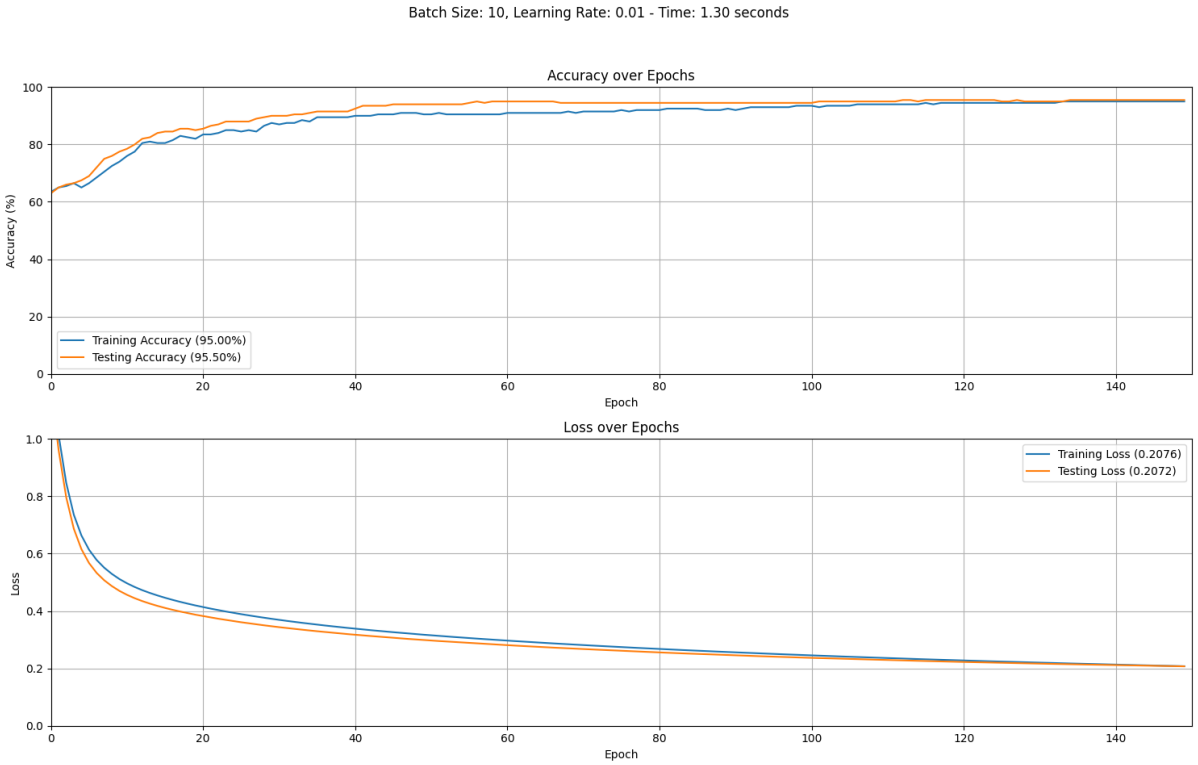

Figure 8: Batch Size: 10, Learning Rate: 0.01

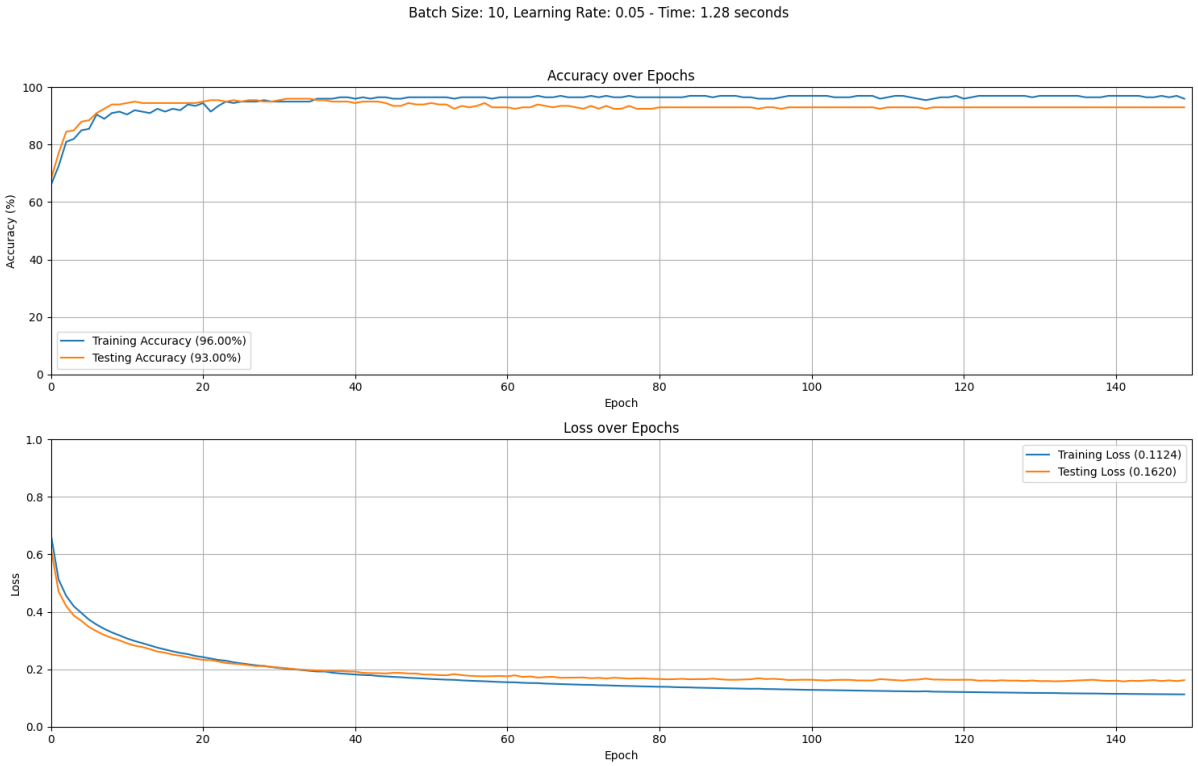

Figure 9: Batch Size: 10, Learning Rate: 0.05

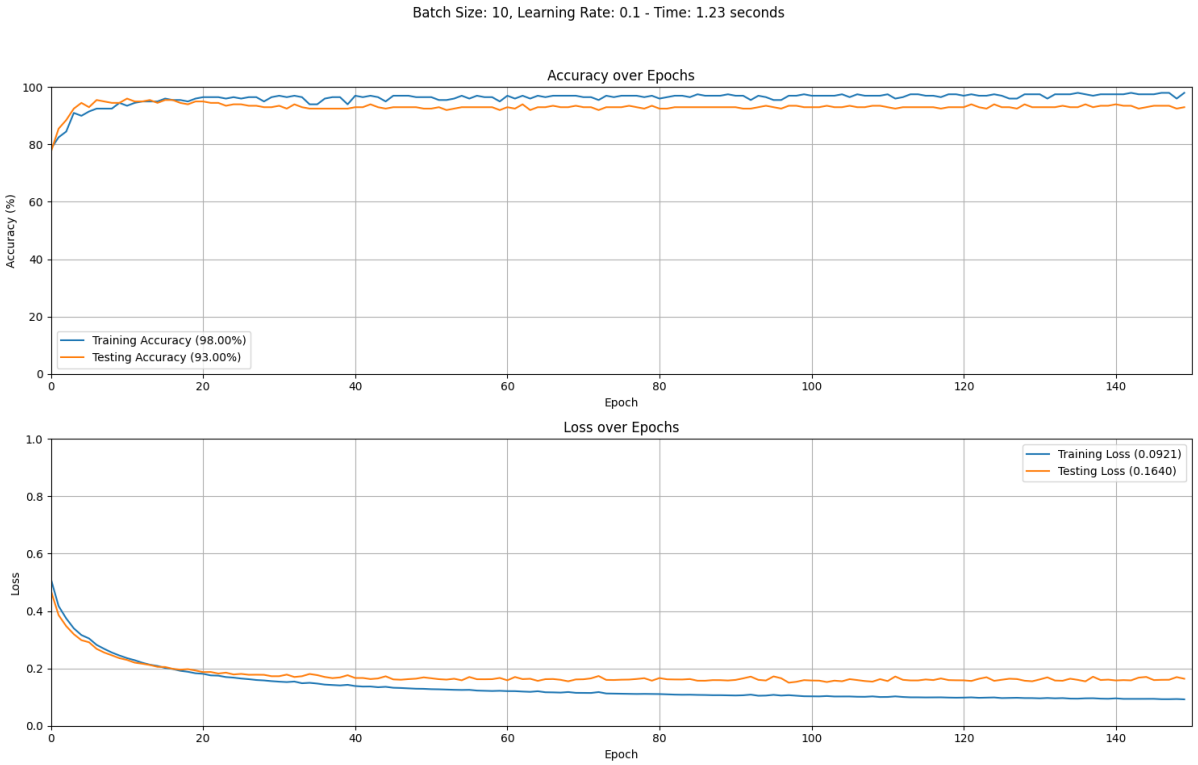

Figure 10: Batch Size: 10, Learning Rate: 0.1

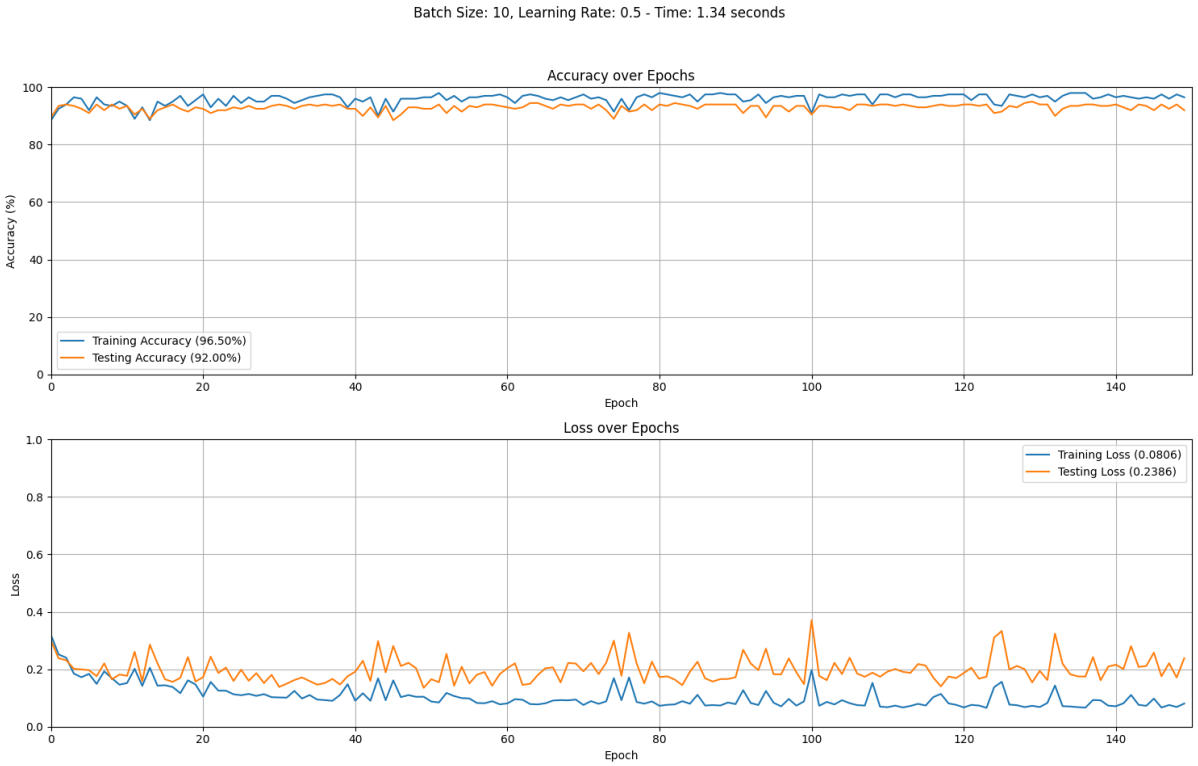

Figure 11: Batch Size: 10, Learning Rate: 0.5

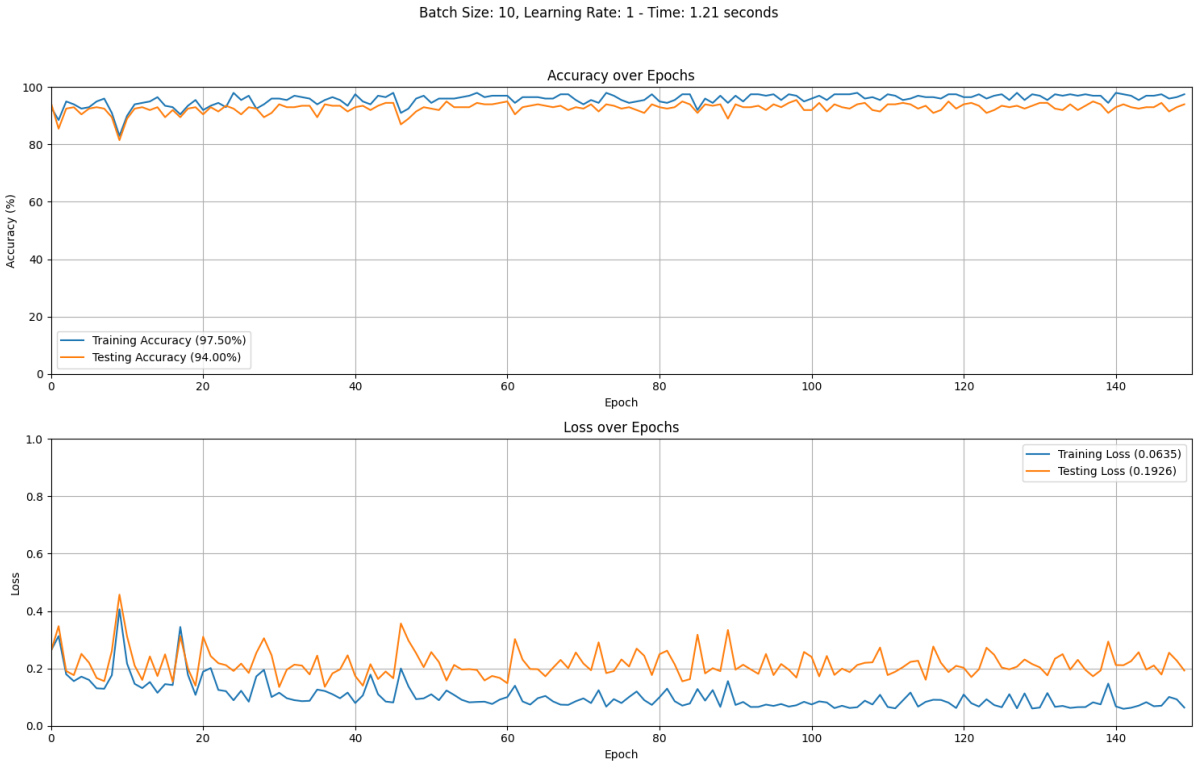

Figure 12: Batch Size: 10, Learning Rate: 1

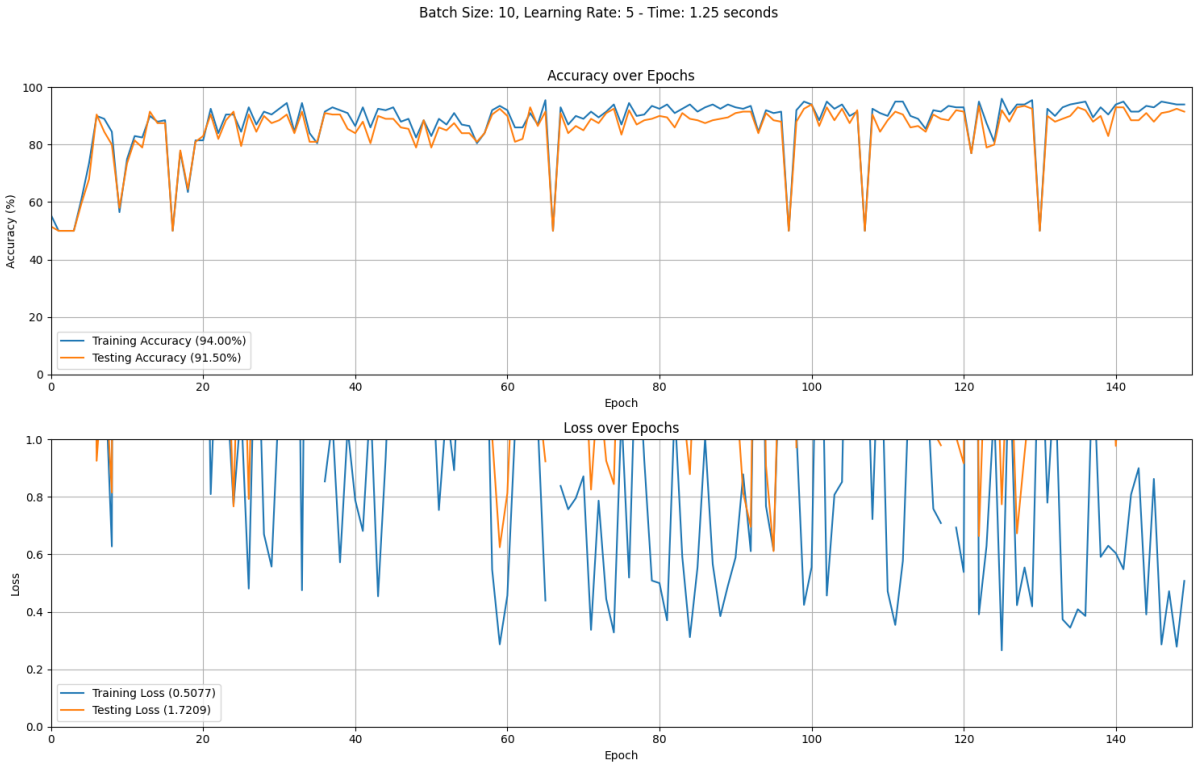

Figure 13: Batch Size: 10, Learning Rate: 5

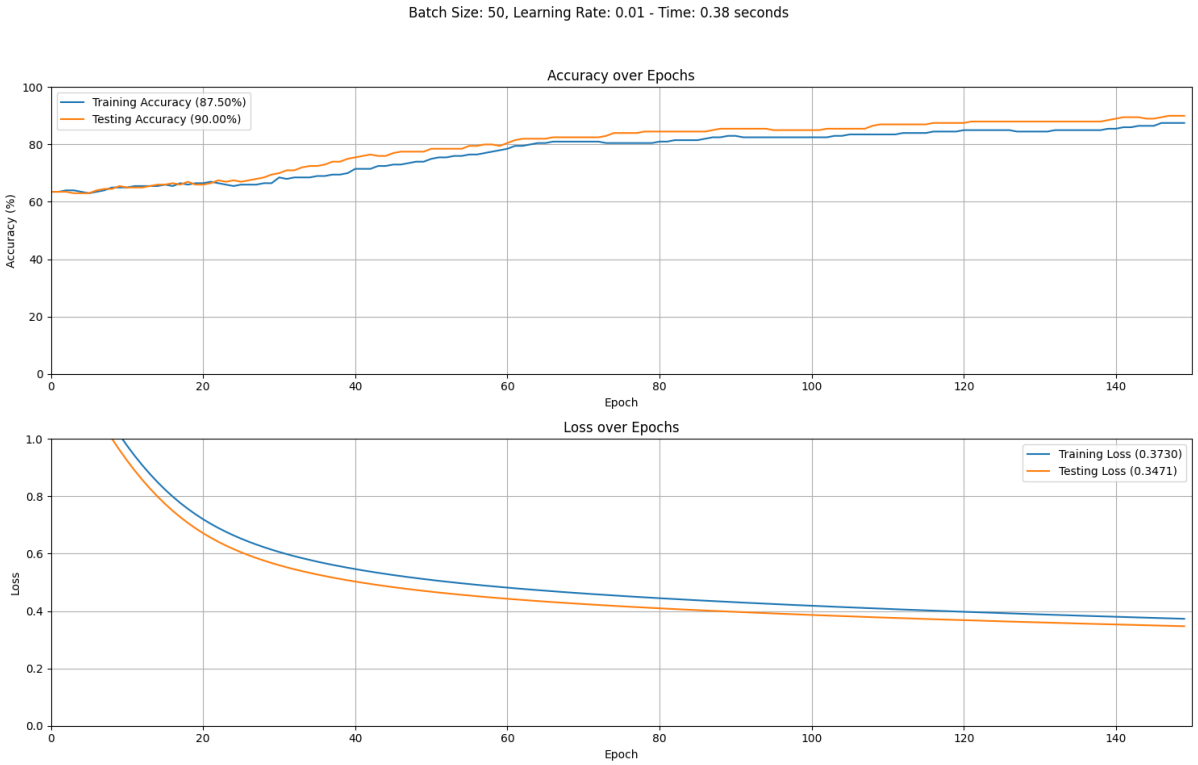

Figure 14: Batch Size: 50, Learning Rate: 0.01

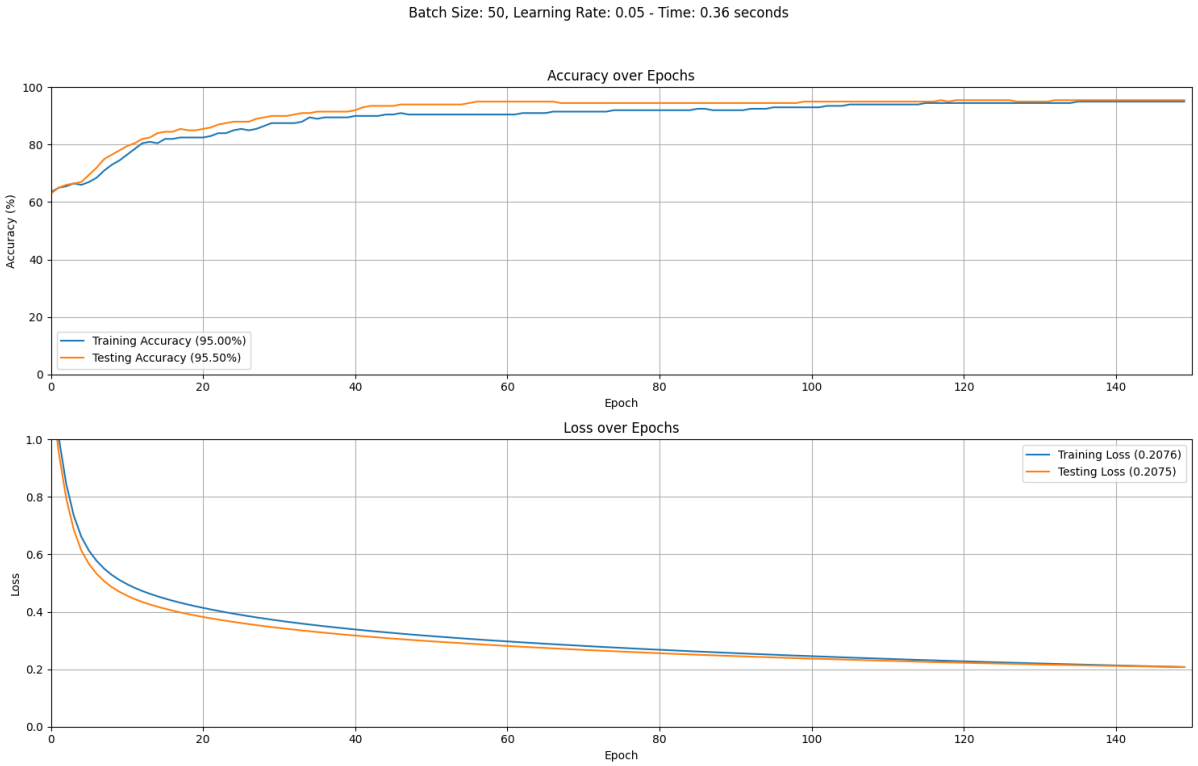

Figure 15: Batch Size: 50, Learning Rate: 0.05

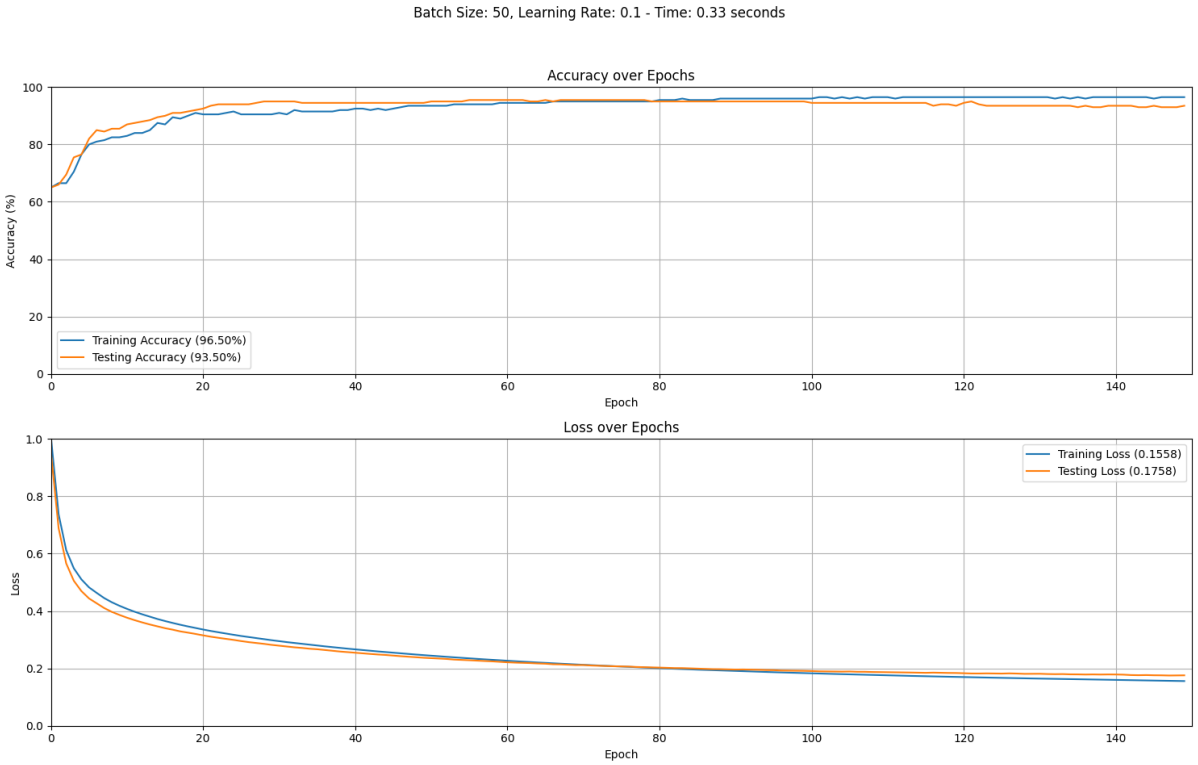

Figure 16: Batch Size: 50, Learning Rate: 0.1

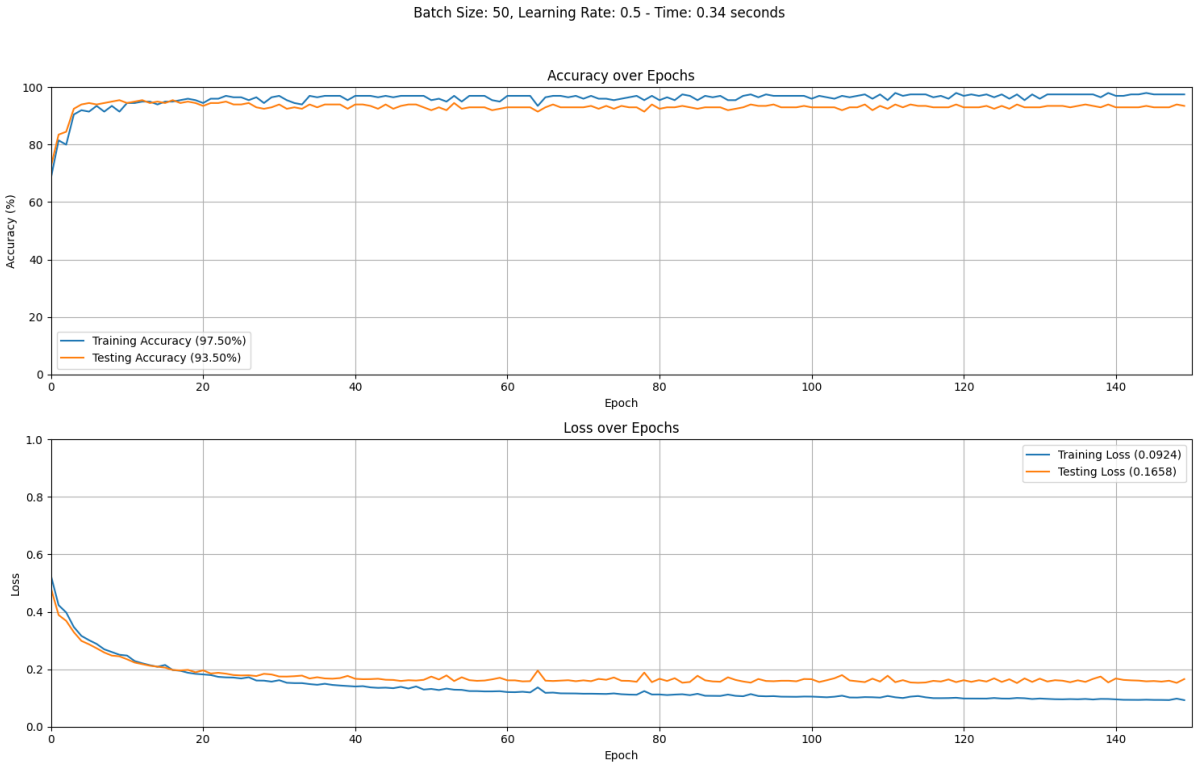

Figure 17: Batch Size: 50, Learning Rate: 0.5

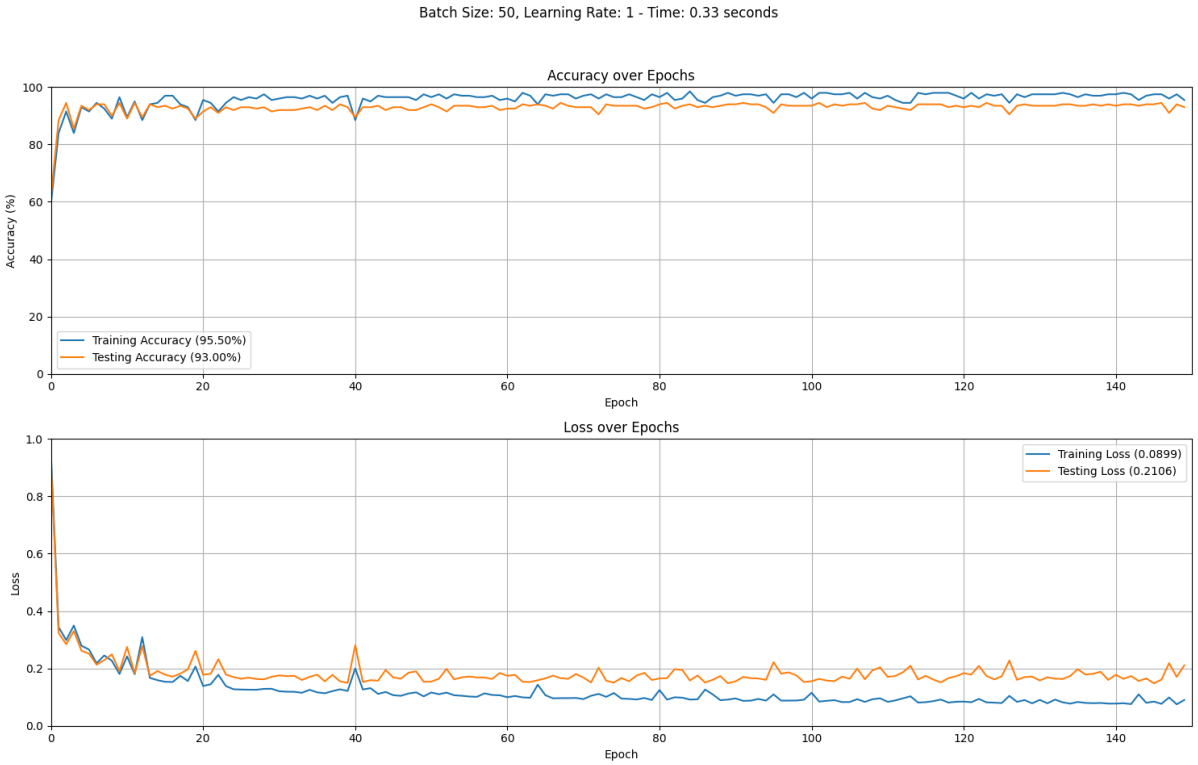

Figure 18: Batch Size: 50, Learning Rate: 1

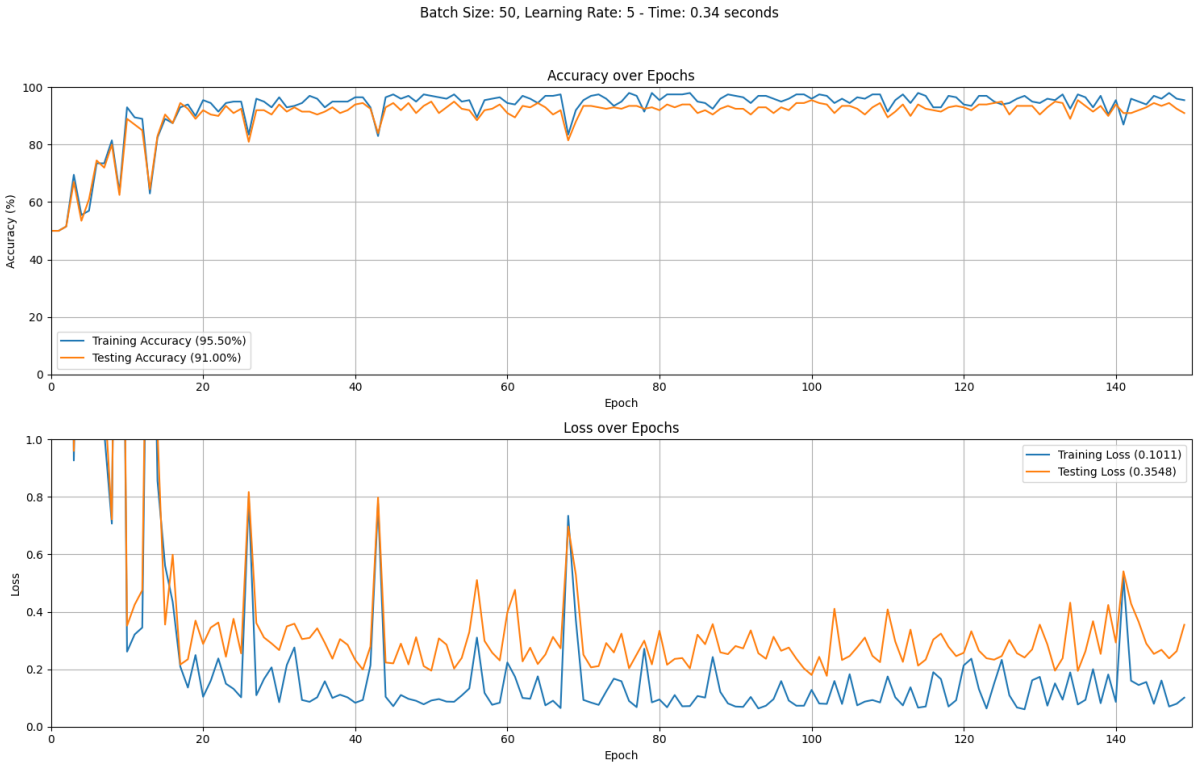

Figure 19: Batch Size: 50, Learning Rate: 5

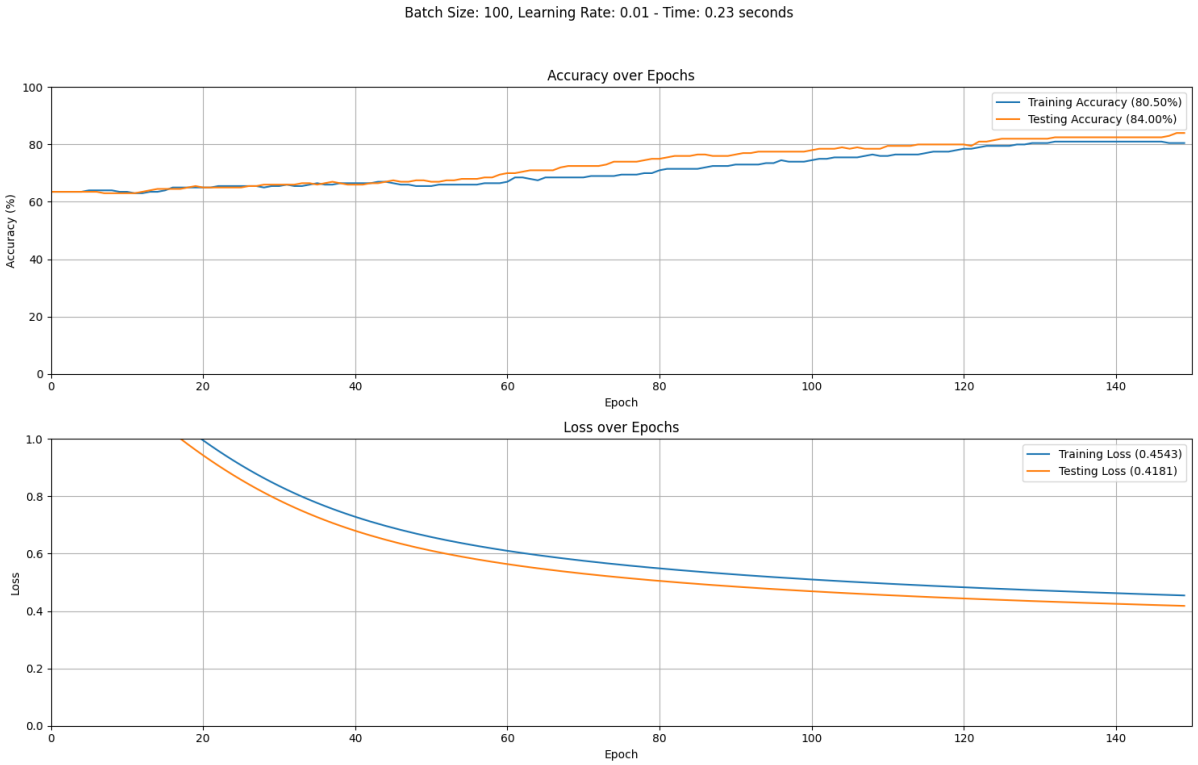

Figure 20: Batch Size: 100, Learning Rate: 0.01

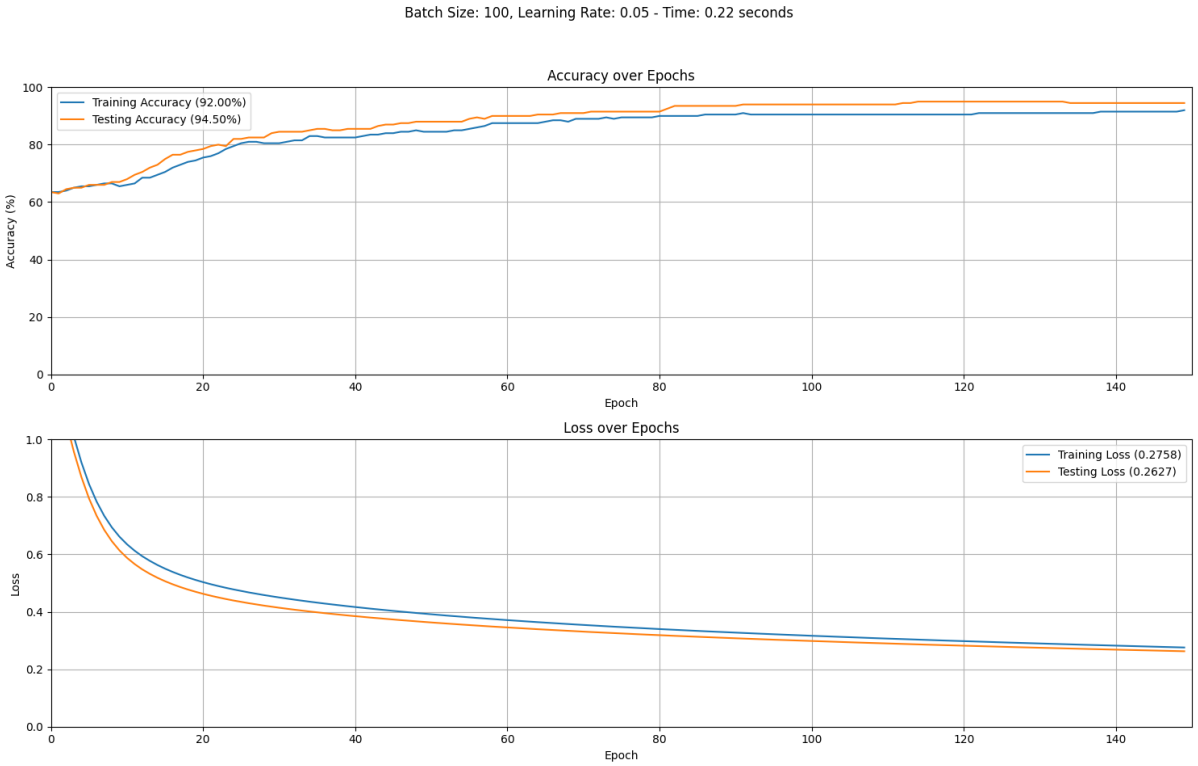

Figure 21: Batch Size: 100, Learning Rate: 0.05

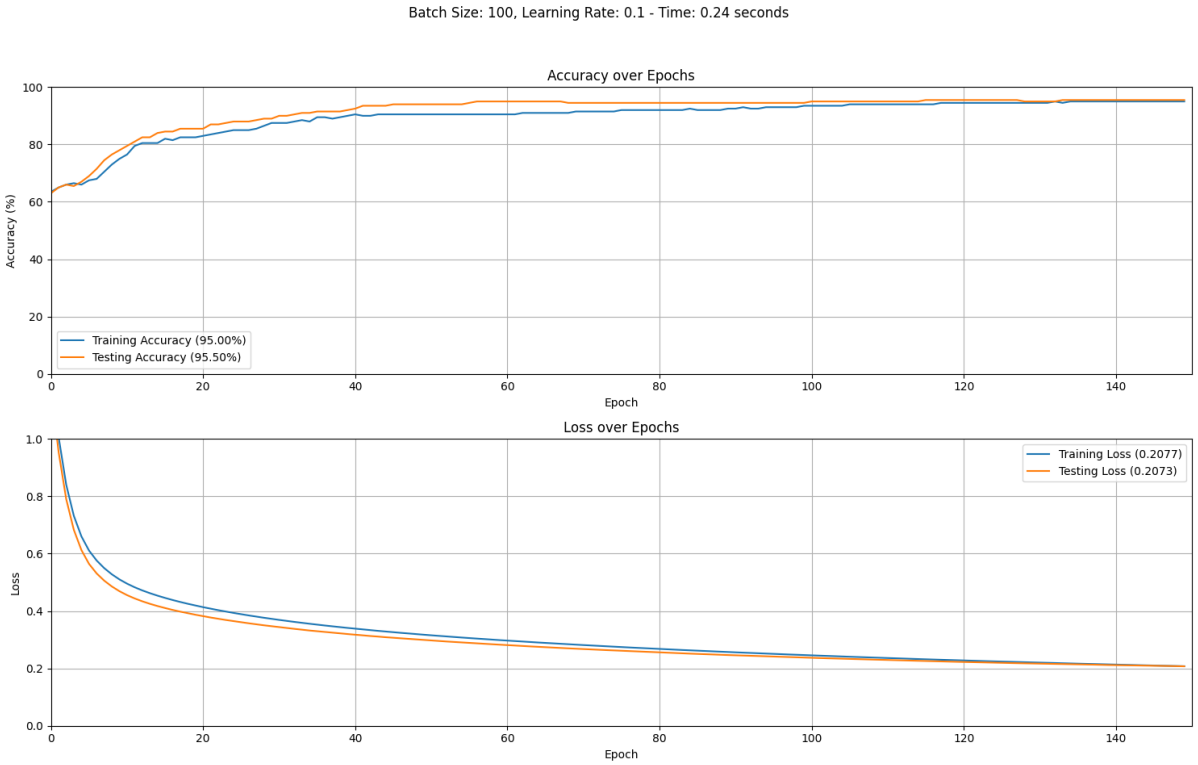

Figure 22: Batch Size: 100, Learning Rate: 0.1

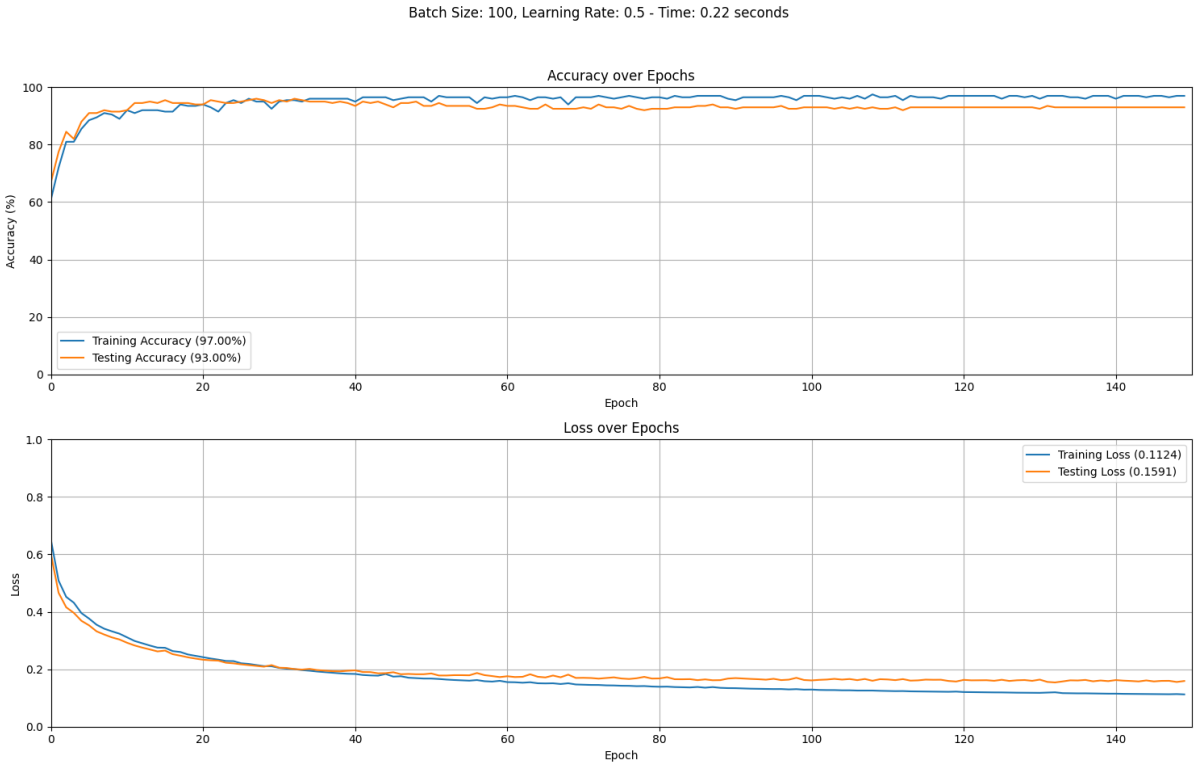

Figure 23: Batch Size: 100, Learning Rate: 0.5

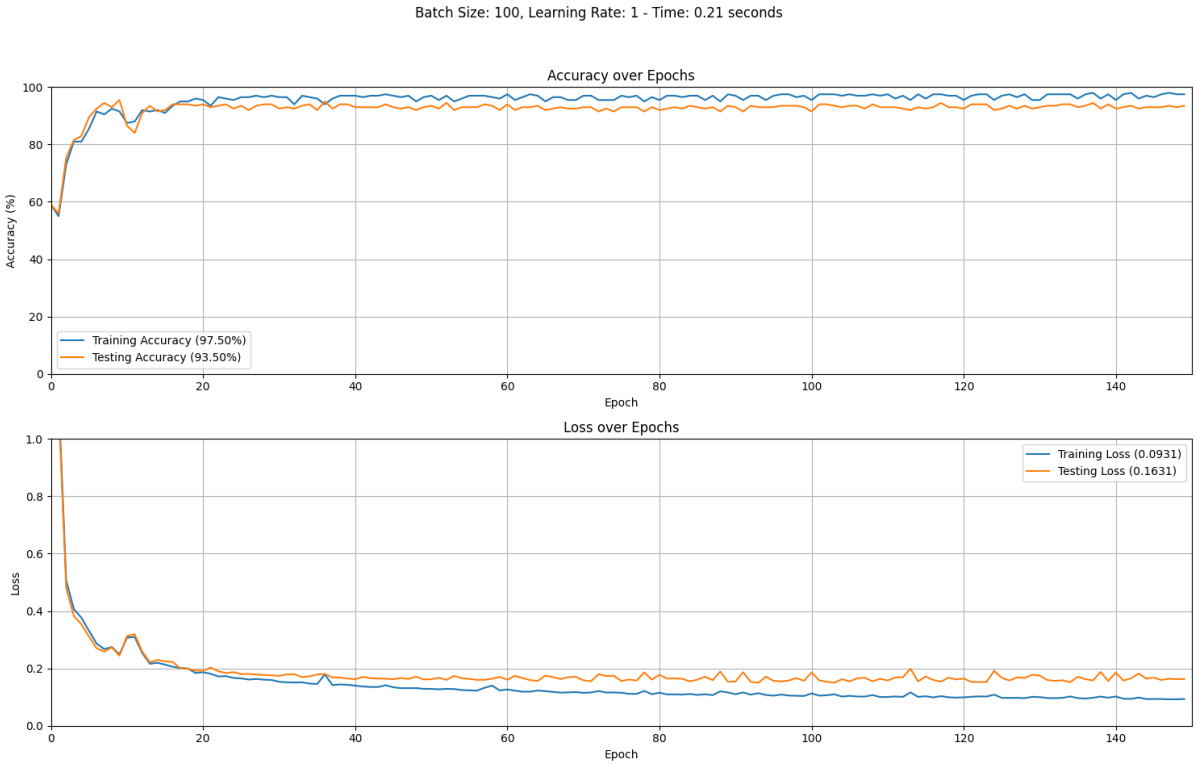

Figure 24: Batch Size: 100, Learning Rate: 1

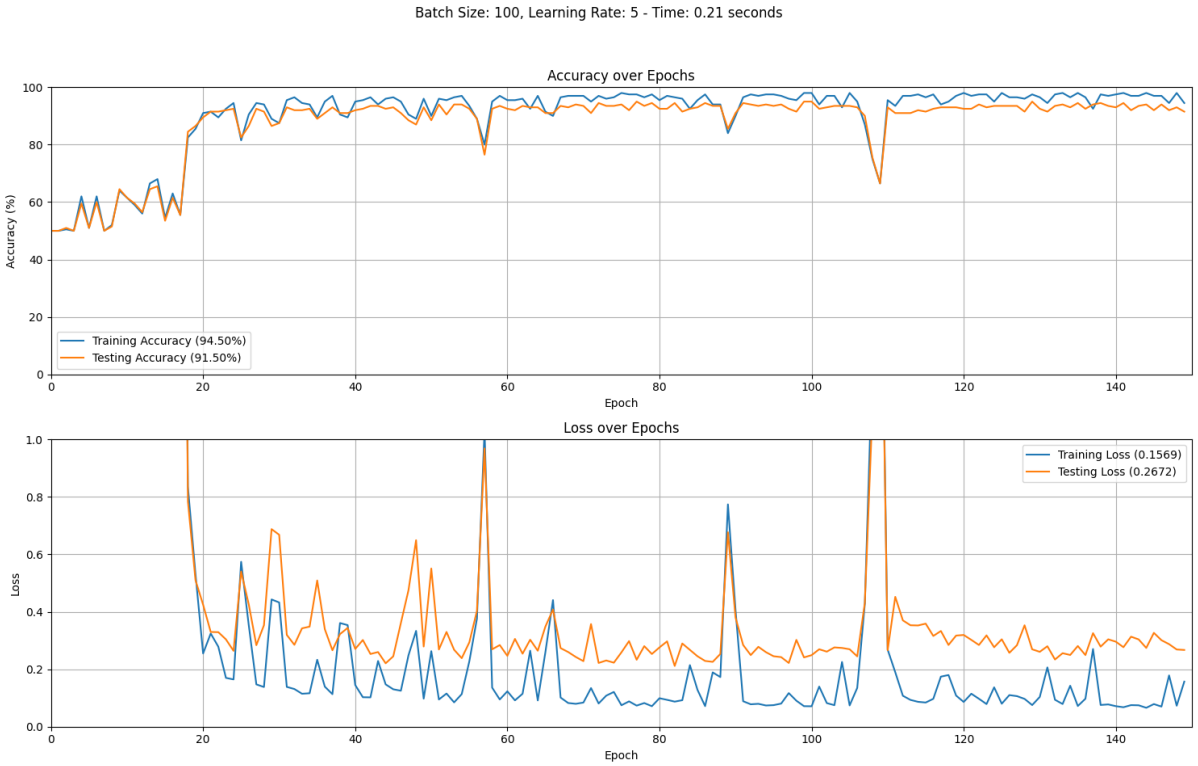

Figure 25: Batch Size: 100, Learning Rate: 5

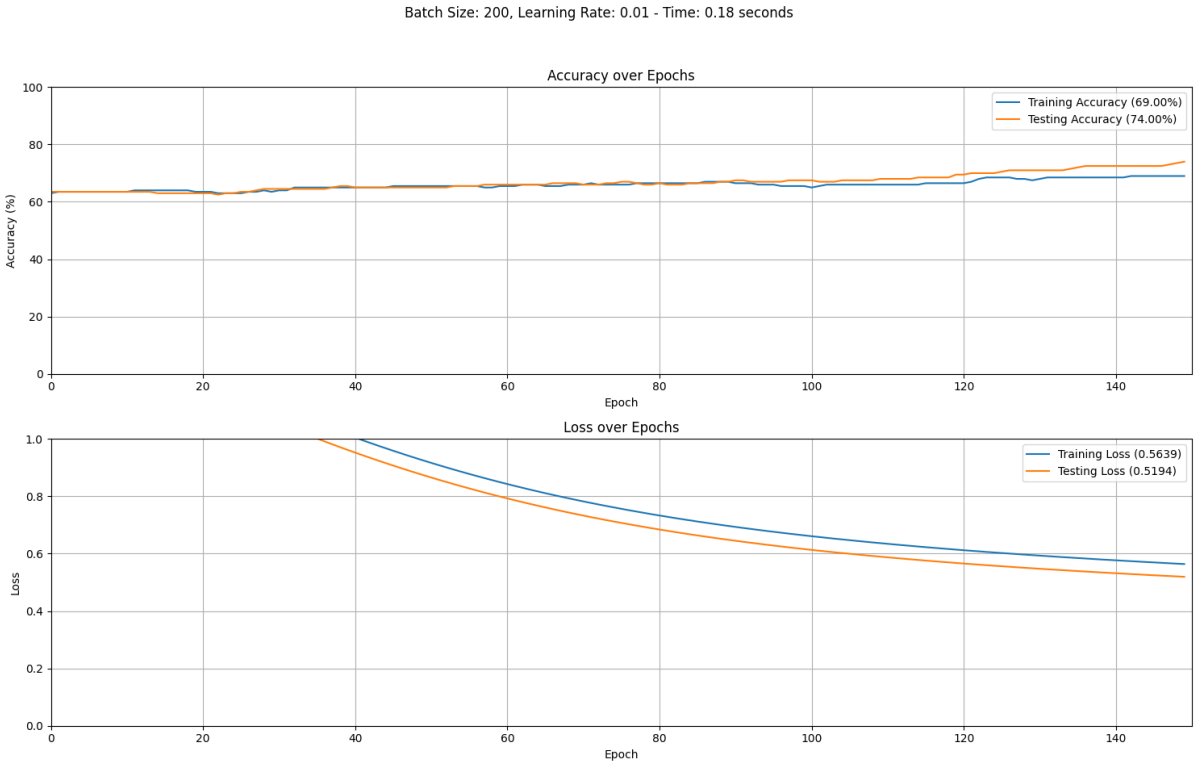

Figure 26: Batch Size: 200, Learning Rate: 0.01

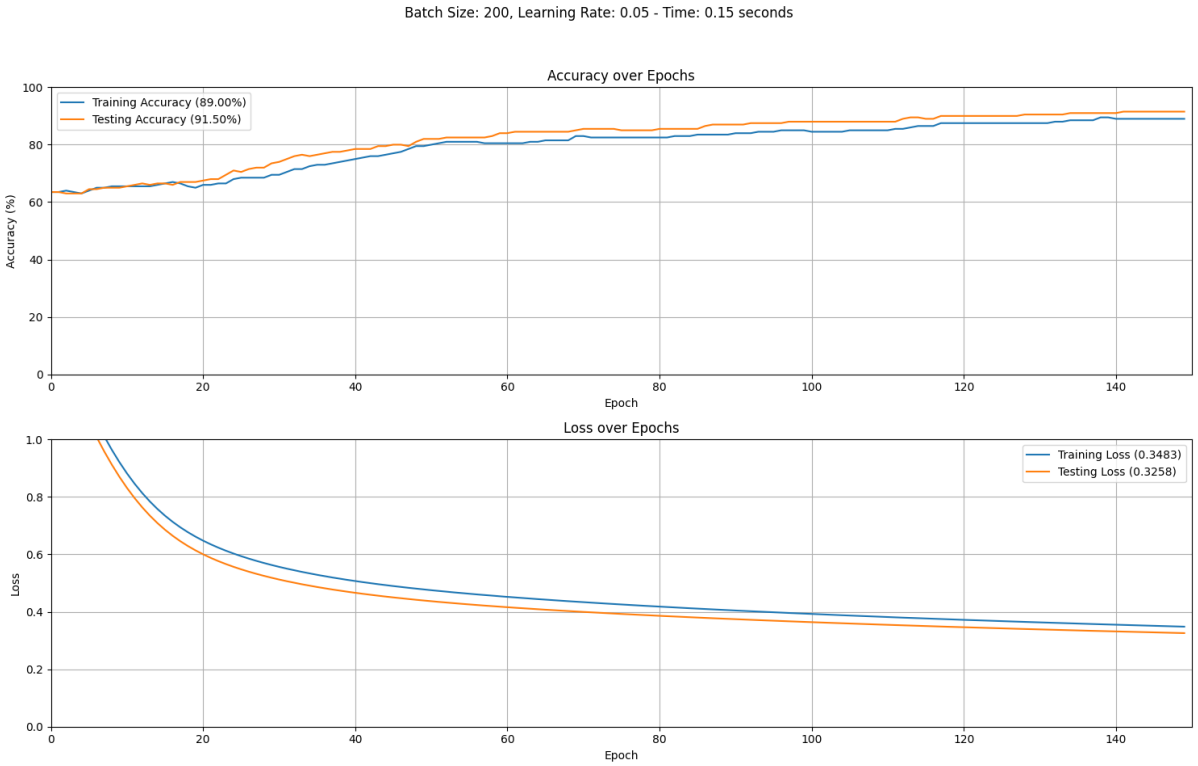

Figure 27: Batch Size: 200, Learning Rate: 0.05

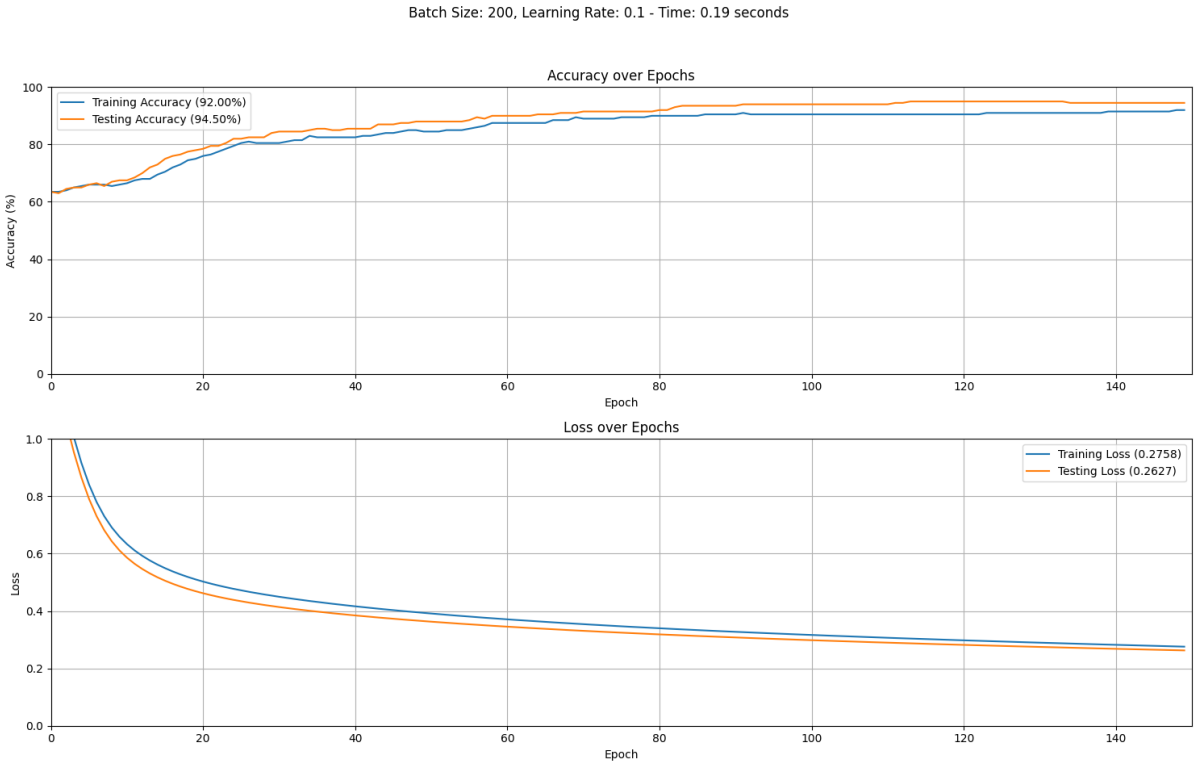

Figure 28: Batch Size: 200, Learning Rate: 0.1

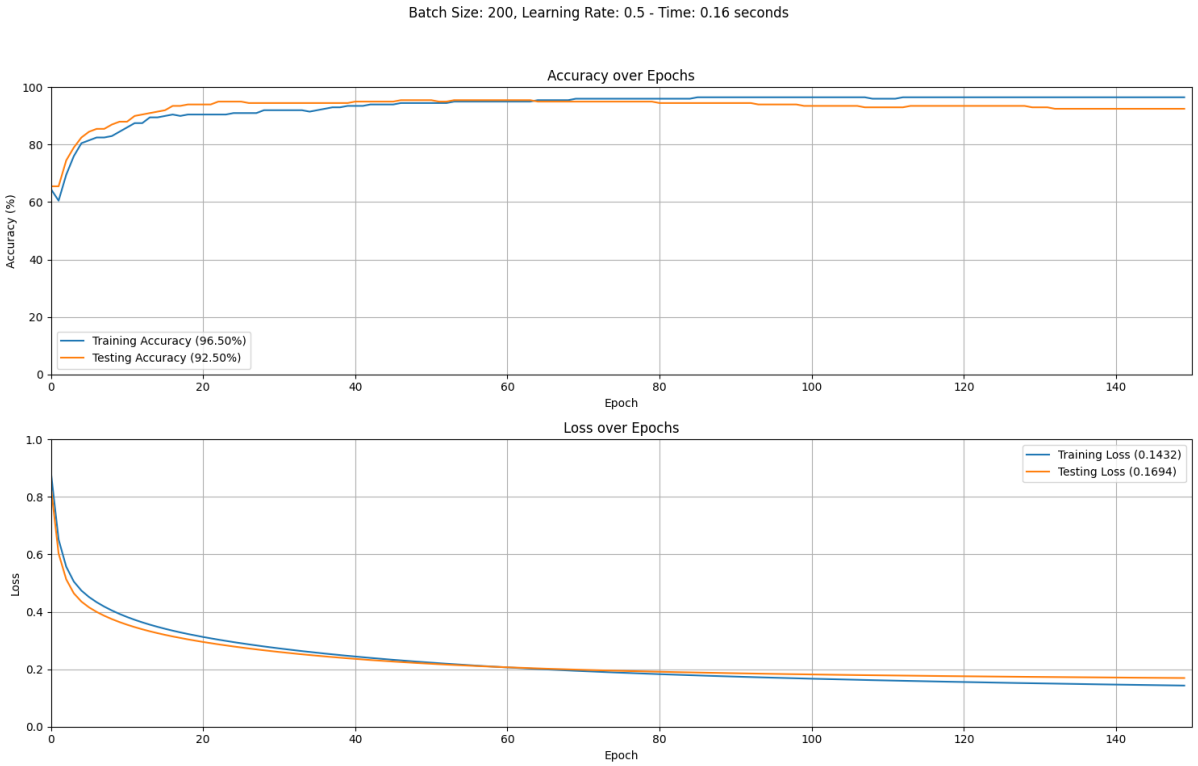

Figure 29: Batch Size: 200, Learning Rate: 0.5

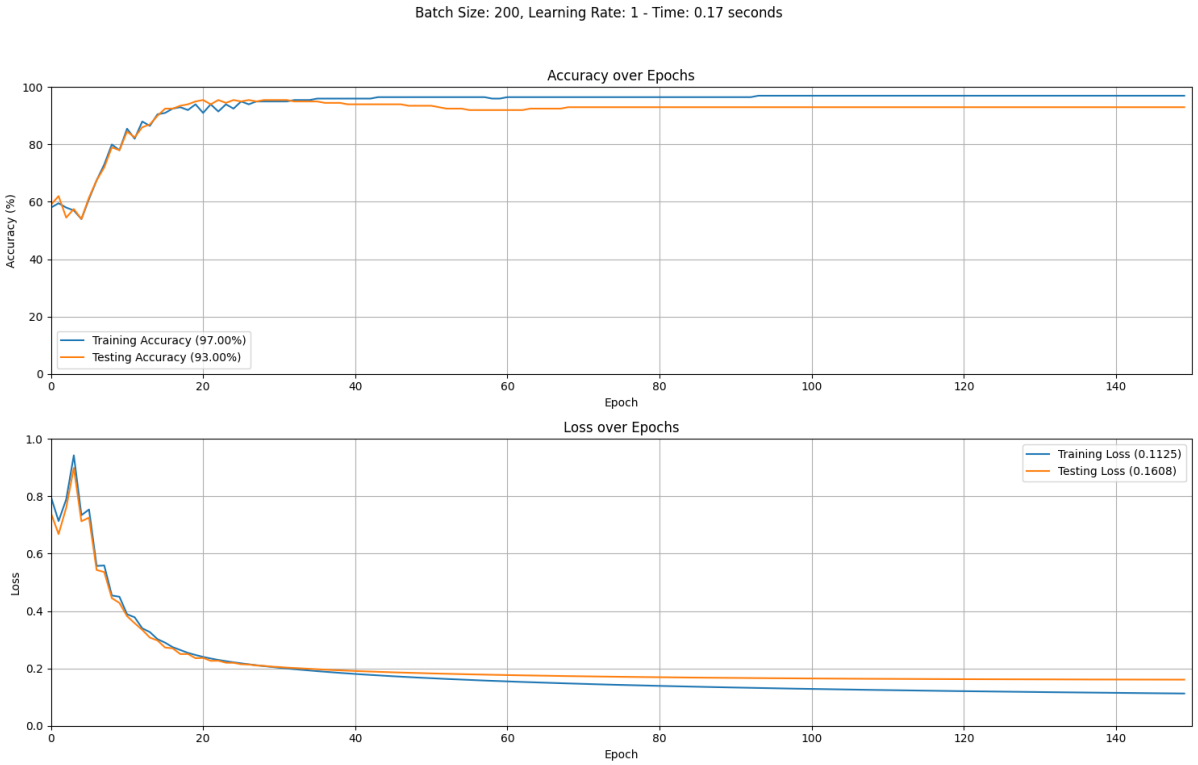

Figure 30: Batch Size: 200, Learning Rate: 1

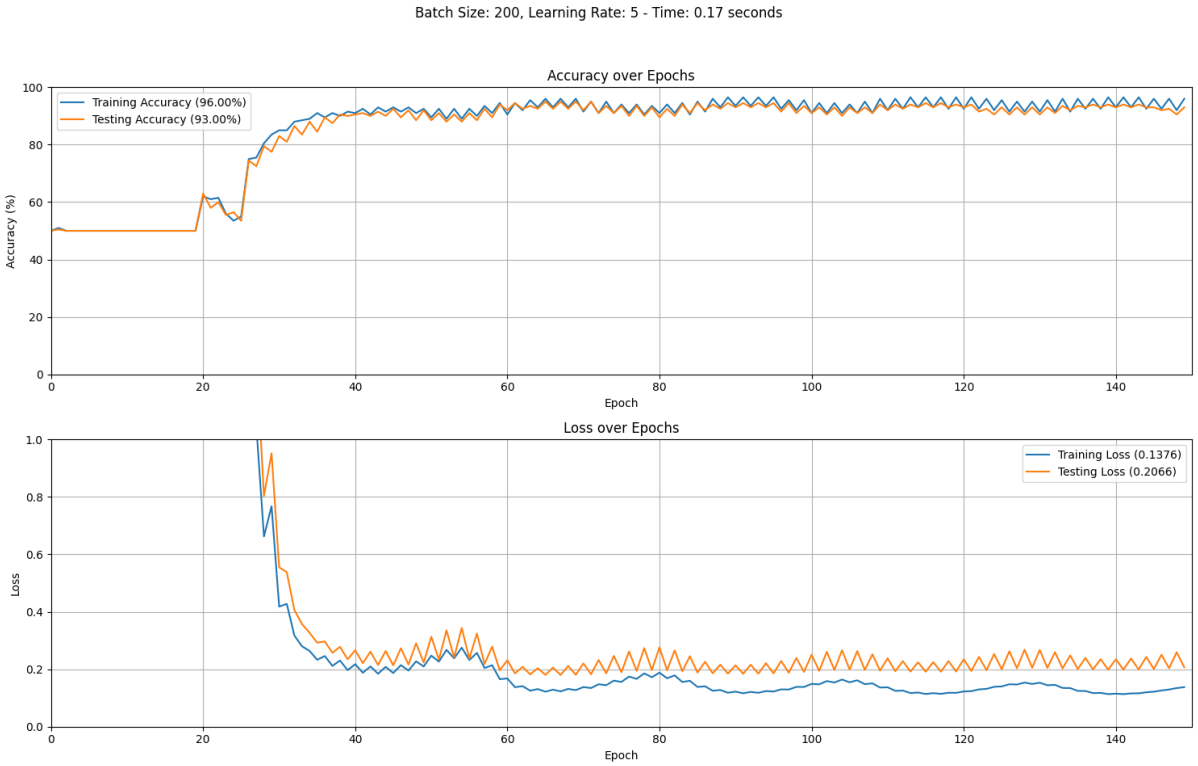

Figure 31: Batch Size: 200, Learning Rate: 5