Implementing a Neural Network for 5-Bit XOR from scratch

Bahne J. Thiel-Peters

December 12, 2024

The *XOR Problem* with 5 inputs involves determining a binary output (0 or 1) based on the parity of 1*s* in the input. Specifically, the output is 1 if the number of 1*s* in the inputs is odd, and 0 if it is even. This problem serves as a benchmark for evaluating a neural network's ability to model non-linear relationships. In this report, a step-by-step implementation and evaluation of a *Multi-Layer Perceptron (MLP)* designed to solve this 5-input XOR problem is presented. The implementation of the proposed model is additionally available in a *Jupyter Notebook*, hosted *here*.

A Implementation

A.1 The Data

A.1.1 Creating the Dataset

The dataset for the XOR problem with 5 inputs consists of all possible binary permutations of length five. Each permutation represents a unique arrangement of *ones* and *zeros* in a specific order. To generate these permutations efficiently, we can use Python's `itertools.product` along with list comprehension. Given that the inputs are binary, combinatorics confirms that the size of our dataset will be $2^5 = 32$.

```
from itertools import product

def generate_binary_permutations(length=5):
    return [list(p) for p in product([0, 1], repeat=length)]

permutations = generate_binary_permutations()
assert len(permutations) == 2**5
```

A.1.2 Defining the Labels

The labels for the dataset are generated based on the mathematical definition of the XOR problem: if the sum of the binary inputs is odd, the output is 1; otherwise, it is 0. Below is the function used to compute the target value for a given input, followed by its application to the dataset.

```
def xor_target_value(binary_list):
    return 1 if sum(binary_list) % 2 != 0 else 0

targets = [[xor_target_value(x)] for x in permutations]
```

With this approach, we now have a complete dataset comprising all possible binary inputs and their corresponding XOR outputs.

A.1.3 Splitting the Data

To train and evaluate the model, the dataset needs to be split into training and testing sets. This can be achieved by creating a helper list of indices, shuffling them, and then using the shuffled indices to partition both the inputs and the labels consistently.

```
import numpy as np

X = np.array(permutations)
y = np.array(targets)

np.random.seed(27)
indices = np.arange(len(X))
np.random.shuffle(indices)

split_index = int(0.8 * len(X))
train_indices = indices[:split_index]
test_indices = indices[split_index:]

X_train, y_train = X[train_indices], y[train_indices]
X_test, y_test = X[test_indices], y[test_indices]
```

A.2 Multi-Layer Perceptron (MLP)

The whole logic of the *NN* is implemented in the *MLP* class. The following section presents the code in a bottom-up manner.

A.2.1 Constructor (`__init__`)

The constructor of the *MLP* class initializes variables essential for training and evaluation, such as the *learning rate*, *weights*, and metrics. During initialization, the weights are set randomly using *numpy.random*.

```
class MLP:
    def __init__(self, layer_sizes, learning_rate=0.01, loss_function="bce"):
        """
        Initialize a multi-layer perceptron with variable hidden layers.
        Args:
            layer_sizes: List of integers specifying the size of each layer.
            learning_rate: Learning rate for gradient descent.
            loss_function: The loss function to use ("bce" for Binary Cross-Entropy,
                ↪ "mse" for Mean Squared Error).
        """
        self.learning_rate = learning_rate
        self.layer_sizes = layer_sizes
        self.loss_function = loss_function.lower()  # Convert to lowercase for
            ↪ consistency

        # Initialize weights and biases
        self.weights = []
        self.biases = []

        for i in range(len(layer_sizes) - 1):
            self.weights.append(np.random.randn(layer_sizes[i], layer_sizes[i + 1]) *
                ↪  0.1)
            self.biases.append(np.zeros((1, layer_sizes[i + 1])))

        # Metrics storage
        self.metrics = {
            "train_loss": [],
            "test_loss": [],
            "train_accuracy": [],
            "test_accuracy": [],
        }
```

A.2.2 Activation- and Loss-Functions

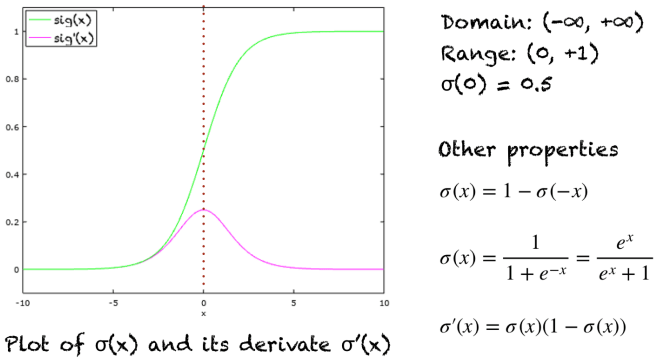

Figure 1: Sigmoid and it's derivative (Mehreen Saeed, 2021)

Sigmoid is the **Activation Function (AF)** within our *NN*. We use it to change (activate) the content of the *Neurons*. To update the *NN's* weights, we use the *Backpropagation Algorithm*, which is essentially just doing derivatives from the output to the input. *Sigmoid* and it's derivative are defined as in the literature (Mehreen Saeed, 2021). As shown in Figure 1, it forces the neurons to stay in the area between 0 and 1, to avoid the *Exploding Gradients Problem* (Katherine (Yi) Li, 2024).

```
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))
```

A.2.3 Loss Functions

The implementation of two distinct loss functions, *Binary Cross-Entropy (BCE)* and *Mean Squared Error (MSE)*, serves to evaluate the network's performance under different optimization perspectives. BCE is well-suited for binary classification tasks, directly aligning with the probabilistic outputs of the network in the XOR problem. It emphasizes the accuracy of predicting binary labels, ensuring robust convergence for classification problems. Conversely, MSE, while traditionally used for regression tasks, provides a smoother gradient landscape, potentially accelerating convergence and offering insights into the model's numerical stability. Using both loss functions enables a comprehensive assessment of the network's behavior, revealing the impact of the choice of loss on training dynamics and generalization.

- *Binary Crossentropy (BCE)* is suited for binary classification tasks. It measures the difference between the predicted probabilities and the actual binary labels, making it ideal for the XOR problem where the output is either 0 or 1 (Richmond Alake, 2023).
- *Mean Squared Error (MSE)*, on the other hand, calculates the average squared difference between predicted and actual values. While it can also be used for binary outputs, it is more commonly used for regression tasks, but still effective in situations like XOR where outputs are binary.

```
    def compute_loss(self, y_true, y_pred):
        if self.loss_function == "bce":
            # Binary Cross-Entropy Loss
            epsilon = 1e-8  # Prevent log(0)
            loss = -np.mean(y_true * np.log(y_pred + epsilon) + (1 - y_true) * np.log
                ↪ (1 - y_pred + epsilon))
        elif self.loss_function == "mse":
            # Mean Squared Error Loss
            loss = np.mean((y_pred - y_true) ** 2)
        else:
            raise ValueError(f"Unsupported loss function: {self.loss_function}")
        return loss
```

A.2.4 Forward-Pass

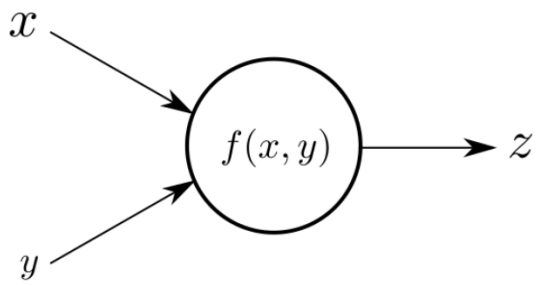

Figure 2: Forward-Pass (Frederik Kratzert, 2016)

Figure 2 above illustrates the essential concepts of the **Forward Pass** and **Backward Pass** in neural networks.
The **Forward Pass** refers to the process of calculating the activations of each layer, starting from the input layer up to the output layer. In this process:

- For each neuron in a layer, the weighted sum (z) of its inputs (x, y) is computed using the formula $z = f(x, y)$. This is represented in the figure as $f(x, y)$.
- The activation function (e.g., Sigmoid) is applied to z to produce the neuron's output (z in the figure).
- The outputs of one layer serve as the inputs to the next layer until the final prediction of the network is obtained.

The forward pass function processes the input data through the neural network to generate predictions. It starts by storing the initial input X in the list `activations`. Then, for each layer, the weighted sum z is calculated by multiplying the current layer's weights by the input from the previous layer and adding the bias. This sum is then passed through the Sigmoid activation function, producing the output for that layer. The process continues for each layer, with the outputs of one layer serving as the inputs to the next. Finally, the function returns the output of the last layer, which represents the network's prediction.

```
def forward(self, X):
    self.activations = [X]  # Store activations of each layer
    self.z_values = []      # Store weighted sums (z) for each layer

    for w, b in zip(self.weights, self.biases):
        z = np.dot(self.activations[-1], w) + b
        self.z_values.append(z)
        self.activations.append(self.sigmoid(z))

    return self.activations[-1]
```

A.2.5 Backward-Pass

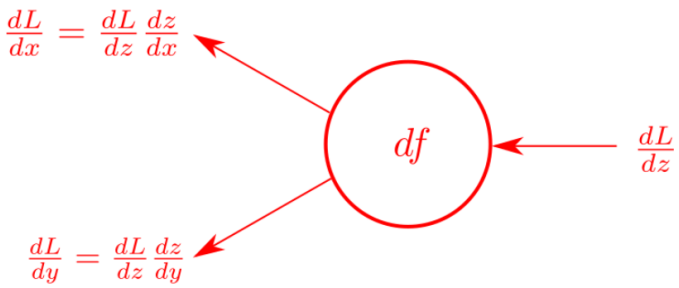

Figure 3: Backward-Pass (Frederik Kratzert, 2016)

The **Backward Pass** (as shown in Figure 3) is the process of computing gradients to adjust the weights and biases of the network. It follows these steps:

- The **output layer error** is calculated based on the difference between the predicted output and the true label ($\frac{dL}{dz}$ in the figure).
- Using the *chain rule*, the error is propagated backward through each layer. The gradients for weights ($\frac{dL}{dx}$, $\frac{dL}{dy}$) and biases are computed at each layer.
- The weights and biases are updated by subtracting the gradients multiplied by the learning rate, minimizing the loss function over time.

This process is crucial for training the network, as it ensures that the model improves its predictions iteratively.

```
def backward(self, y_true):
    gradients_w = [None] * len(self.weights)
    gradients_b = [None] * len(self.biases)

    if self.loss_function == "bce":
        # Error for BCE (sigmoid is derivative-friendly with BCE)
        delta = (self.activations[-1] - y_true) * self.sigmoid_derivative(self.
            ↪ z_values[-1])
    elif self.loss_function == "mse":
        # Error for MSE
        delta = 2 * (self.activations[-1] - y_true) * self.sigmoid_derivative(self.
            ↪ z_values[-1])

    # Gradients for the output layer
    gradients_w[-1] = np.dot(self.activations[-2].T, delta)
    gradients_b[-1] = np.sum(delta, axis=0, keepdims=True)

    # Backpropagate through hidden layers
    for l in range(len(self.weights) - 2, -1, -1):
        error = np.dot(delta, self.weights[l + 1].T)
        delta = error * self.sigmoid_derivative(self.z_values[l])

        gradients_w[l] = np.dot(self.activations[l].T, delta)
        gradients_b[l] = np.sum(delta, axis=0, keepdims=True)

    # Update weights and biases
    for i in range(len(self.weights)):
        self.weights[i] -= self.learning_rate * gradients_w[i]
        self.biases[i] -= self.learning_rate * gradients_b[i]
```

Using the concepts in the figure, we can understand that the forward pass computes z, while the backward pass calculates the gradients ($\frac{dL}{dz}$) and propagates them back to adjust parameters. This interplay of forward and backward passes drives the learning process in neural networks.

A.2.6 Accuracy

The `compute_accuracy` method calculates the accuracy of the model by comparing the predicted labels with the true labels. The predictions are first converted to binary values based on a threshold of 0.5, and the proportion of correct predictions is computed as the accuracy.

```
    def compute_accuracy(self, y_true, y_pred):
        y_pred_binary = (y_pred > 0.5).astype(int)
        accuracy = np.mean(y_pred_binary == y_true)
        return accuracy
```

A.2.7 Training

The training process for the *MLP* is implemented in the `train` method. The purpose of this method is to iteratively optimize the model's weights and biases to minimize the loss function and improve prediction accuracy. The method follows a structured approach:

1. **Forward Pass:** For each epoch, the data is passed through the network to compute the outputs of each layer, which are stored as activations.
2. **Loss Computation:** The difference between the predicted output and the true labels is calculated using the specified loss function, such as *MSE* or *BCE*.
3. **Backward Pass:** Using backpropagation, the gradients of the loss with respect to the weights and biases are computed. These gradients are then used to adjust the weights and biases to minimize the loss.
4. **Metric Computation:** For each epoch, the training loss and accuracy are computed and stored for performance monitoring.

The iterative training process ensures that the model learns to map inputs to their correct outputs through optimization of its internal parameters.

```
    def train(self, X_train, y_train, X_test, y_test, epochs=1000):
        """
        Train the MLP on the given training data.
        Args:
            X_train: Training input data.
            y_train: Training labels.
            epochs: Number of training epochs.
        """
        for epoch in range(epochs):
            # 1. Forward pass on training data.
            train_pred = self.forward(X_train)
            train_loss = self.compute_loss(y_train, train_pred)
            train_accuracy = self.compute_accuracy(y_train, train_pred)

            # 2. Backward pass for training data.
            self.backward(y_train)

            # 3. Forward pass on test data (Since we test our MLP with this step, we
                ↪ don't update our weights at this point.
            test_pred = self.forward(X_test)
            test_loss = self.compute_loss(y_test, test_pred)
            test_accuracy = self.compute_accuracy(y_test, test_pred)

            # 4. Store metrics.
            self.metrics["train_loss"].append(train_loss)
            self.metrics["test_loss"].append(test_loss)
            self.metrics["train_accuracy"].append(train_accuracy)
            self.metrics["test_accuracy"].append(test_accuracy)
```

A.3 Evaluation

`plot_combined_metrics` visualizes the training and testing metrics (Loss, Accuracy, and Bad Facts) for both the BCE and MSE models across epochs. The method uses the `matplotlib.pyplot` package to create subplots for each metric.
The `for` loop iterates over the `metrics_to_plot` dictionary, which maps metric names to their corresponding training and testing data keys for both models. For each metric, the training and testing data are plotted for both models using different colors: shades of blue for the BCE model and shades of red/orange for the MSE model. The loop also handles the labeling and legend to clearly differentiate between the different data sets. This allows for an easy comparison of model performance across epochs.

```
import matplotlib.pyplot as plt

def plot_combined_metrics(model_bce, model_mse):
    metrics_to_plot = {
        "Loss": ("train_loss", "test_loss"),
        "Accuracy": ("train_accuracy", "test_accuracy"),
        "Bad Facts": ("train_bad_facts", "test_bad_facts"),
    }

    plt.figure(figsize=(16, 4))
    for i, (title, (train_key, test_key)) in enumerate(metrics_to_plot.items(), 1):
        plt.subplot(1, len(metrics_to_plot), i)

        # Plot BCE Metrics
        plt.plot(model_bce.metrics[train_key], label=f"Train {title} (BCE)", color="
            ↪ darkblue")
        plt.plot(model_bce.metrics[test_key], label=f"Test {title} (BCE)", color="
            ↪ lightblue")

        # Plot MSE Metrics
        plt.plot(model_mse.metrics[train_key], label=f"Train {title} (MSE)", color="
            ↪ red")
        plt.plot(model_mse.metrics[test_key], label=f"Test {title} (MSE)", color="
            ↪ orange")

        plt.title(f"{title} vs. Epochs")
        plt.xlabel("Epochs")
        plt.ylabel(title)
        plt.legend()

    plt.tight_layout()
    plt.show()
```

A.4 Calling the *MLP*

The following example illustrates how to *initialize, train, and evaluate* the *MLP* with *BCE* as the loss function. The `train` method is employed to fit the model to the training data, while the `plot_metrics` method is used to visualize the performance metrics over time. Furthermore, the network architecture is flexible; by simply modifying the `layer_sizes` list, new layers can be added to the network. This allows for easy experimentation with different architectures. In the example, the model consists of two hidden layers, each with five neurons, and one output layer.

```
mlp_bce = MLP(layer_sizes=[5, 5, 1], learning_rate=0.1, loss_function="bce")
mlp_bce.train(X_train, y_train, X_test, y_test, epochs=5000)
mlp_bce.plot_metrics()
```

B Evaluation Results

In this evaluation, two models are compared: one trained with the **BCE** loss function and the other with the **MSE** loss function. Both models use a *fixed learning rate of 0.1* and are trained for *5000 epochs*. The evaluation explores the impact of model complexity—ranging from a simple architecture with one hidden layer of 5 neurons to deeper and wider configurations—and varying amounts of training data. This setup allows for analyzing how architectural choices, loss functions, and data availability affect the models' *performance, generalization, and convergence behavior*.

B.1 Evaluation of the models with 1 Hidden Layer and 5 Neurons

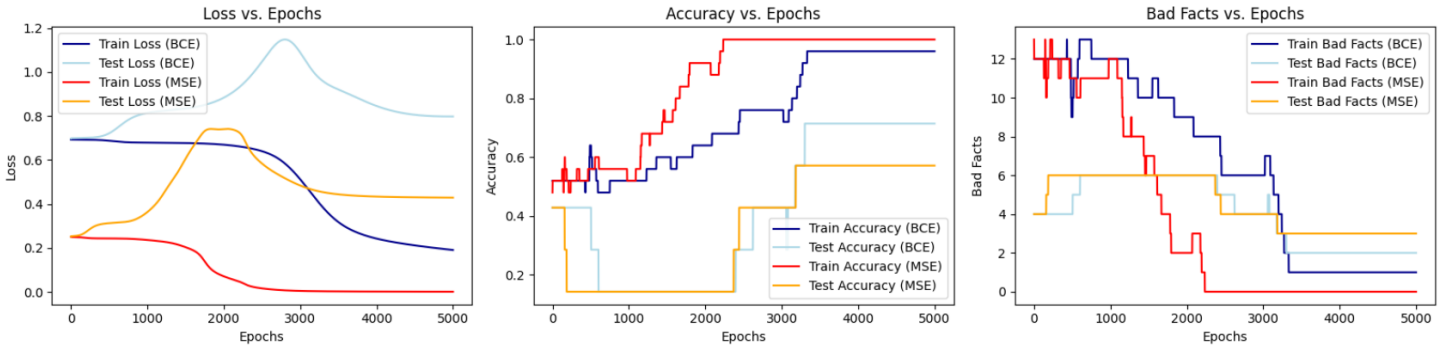

Figure 4: 5-5-1-NN, LR = 0.1, 5000 epochs

Figure 4 shows the performance of both models trained on a simple neural network with one hidden layer of 5 neurons. The *BCE* model demonstrates better **generalization**, with a stable reduction in loss and fewer signs of **overfitting**, while the *MSE* model shows a noticeable train-test loss gap during training. Both models converge after approximately *3000 epochs*, achieving similar *Accuracy* and *Bad Facts* levels.
Both models perform well considering their simplicity, but the *BCE* model demonstrates superior generalization, leading to better performance on test data.

B.2 Evaluation of the models with 1 Hidden Layer and 20 Neurons

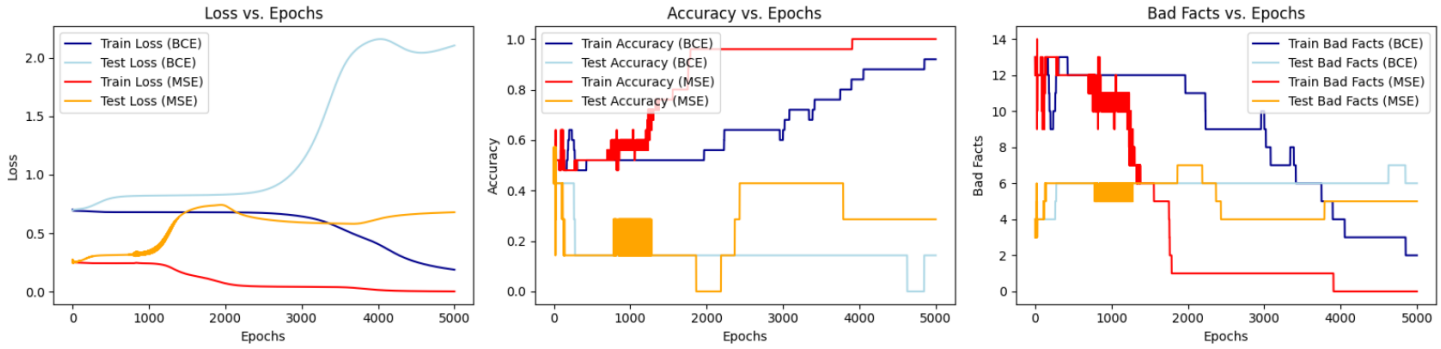

Figure 5: One hidden layer with 20 neurons

Figure 5 illustrates the performance of the models trained on a neural network with one hidden layer, consisting of 20 neurons. Both models exhibit stagnation during the first *3000 epochs*, likely due to the increased number of neurons (20) in the single hidden layer. While vanishing gradients are less likely in a single-layer network, the increased number of neurons may still make optimization more challenging, leading to slower progress during backpropagation. Interestingly, contrary to the results from the simpler network, the *MSE* model performs better than the *BCE* model on both training and test data. This can be observed in the lower loss values and the overall better **accuracy** achieved by the *MSE* model. However, both models show clear signs of overfitting, as seen in the relatively large gap between train and test performance, particularly in terms of loss and bad facts.

This overfitting may stem from the network's overly complex architecture relative to the simplicity of the task. In summary, while the *MSE* model demonstrates stronger performance in this case, the network's structure appears to hinder effective generalization for both models.

B.3 Evaluation of Model with 2 Hidden Layers and 5 Neurons each

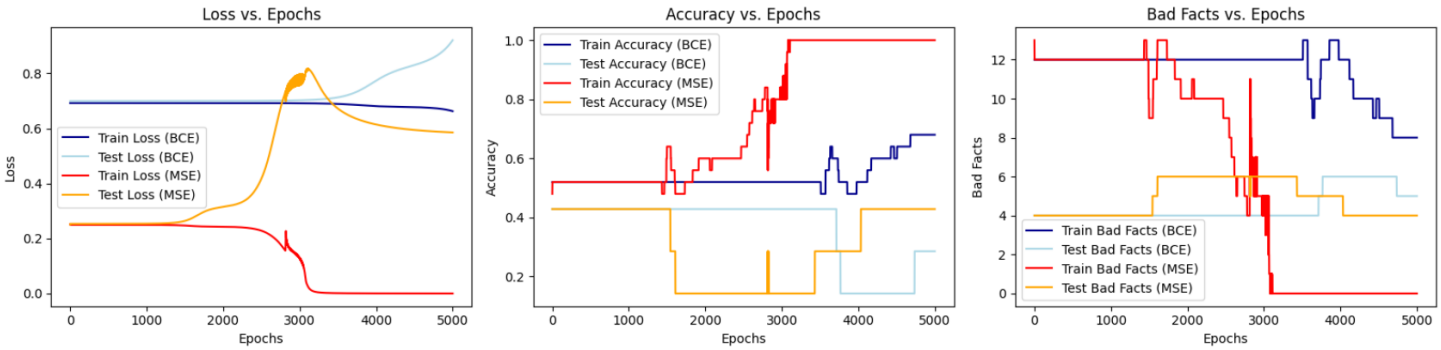

Figure 6: 2 hidden layers, 5 neurons each

Figure 6 illustrates the performance of both models trained on a neural network with two hidden layers, each containing 5 neurons. Both models exhibit prolonged stagnation during the initial *4000 epochs*, which is longer than in the previous models. This prolonged stagnation might be partially caused by a slight effect of vanishing gradients, as the increased complexity of having two hidden layers (compared to one previously) could make backpropagation more challenging. The issue is compounded by the fact that the *learning rate* remains at *0.1*, which, while relatively high, might not fully compensate for the added architectural depth. Despite these challenges, both models eventually converge, though their performance varies.

The *MSE* model shows superior performance compared to the *BCE* model on both training and test data. However, it suffers from significant overfitting, as evident from the train-test loss gap and its rapid convergence to near-zero loss on the training set. This overfitting is likely due to the model's excessive capacity relative to the simplicity of the problem. On the other hand, the *BCE* model performs worse than in previous configurations, struggling to achieve competitive accuracy and demonstrating less stability across metrics.

Overall, the results suggest that the increased model complexity leads to challenges in generalization, with the *MSE* model overfitting and the *BCE* model underperforming.

B.4 Evaluation of Model B.1 with Increased Data (3x Training Data)

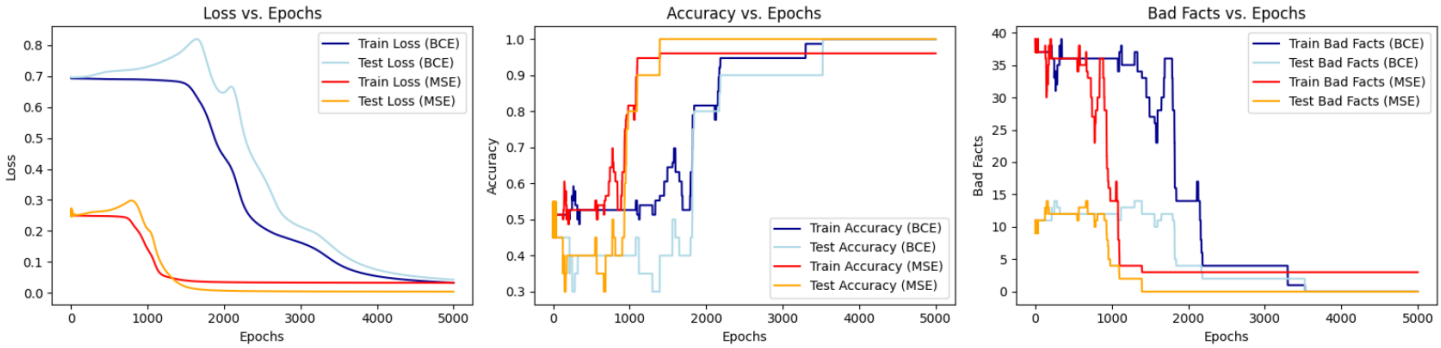

Figure 7: 2 hidden layers, 5 neurons each, 3x test-data

Figure 7 presents a plot of the metrics for the simplest model (Model B.1), but trained with three times the amount of data. Both models achieve near-perfect performance with the addition of more training data. The model trained with Binary Cross-Entropy (BCE) reaches perfect accuracy and zero errors after approximately 3500 epochs. In contrast, the Mean Squared Error (MSE) model converges much faster, achieving near-perfect results in roughly 1500 epochs. Interestingly, the MSE model

shows slightly better accuracy on the testing data, which might be attributed to the randomness in the dataset split. Overall, the added training data enables both models to excel, with differences in convergence speed and testing accuracy highlighting the trade-offs between BCE and MSE.

C Conclusion

The evaluation highlights the interplay between **model complexity**, **loss functions**, and **data availability** in determining the performance of neural networks. Even the simplest model architecture, with one hidden layer of 5 neurons (Model B.1), achieves good performance, demonstrating the suitability of neural networks for the given task. Increasing model complexity by adding more neurons or additional hidden layers yields mixed results. While these configurations occasionally outperform the simplest model in terms of training accuracy, they also introduce challenges such as slower convergence and increased susceptibility to overfitting. Incorporating dropout regularization could address the overfitting observed in more complex models, fostering better generalization without compromising performance (Srivastava et al., 2014).

Comparing the two **Loss Functions**, *Binary Cross-Entropy (BCE)* consistently demonstrates superior generalization, particularly in setups where overfitting is a concern. On the other hand, the *Mean Squared Error (MSE)* loss function often converges faster and sometimes achieves better test accuracy, though its rapid convergence can exacerbate overfitting in more complex models.

Notably, increasing the **dataset size** significantly improves performance across all models. With three times the amount of data, even the simplest model achieves near-perfect accuracy and zero errors, underscoring the importance of sufficient data for effective training. While BCE reaches perfect accuracy slightly later than MSE, its generalization remains robust, reaffirming its reliability for diverse training scenarios.

In summary, the results demonstrate that simple neural network architectures, paired with appropriate loss functions and sufficient training data, can achieve excellent performance. However, careful consideration of **model complexity**, **data size**, and **loss function** selection is essential to balance convergence speed, accuracy, and generalization.

References

Frederik Kratzert. (2016). *Understanding the backward pass through Batch Normalization Layer*. https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

Katherine (Yi) Li. (2024). *Vanishing and Exploding Gradients in Neural Network Models: Debugging, Monitoring, and Fixing*. https://neptune.ai/blog/vanishing-and-exploding-gradients-debugging-monitoring-fixing

Medium. (2023). *Deep Learning-XOR Implementation*. https://nowitsanurag.medium.com/deep-learning-xor-implementation-240d61e56fd4

Mehreen Saeed. (2021). *A Gentle Introduction To Sigmoid Function*. https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/

OpenAI. (2024). *ChatGPT 3.5* [This tool was used to formalise ideas in the form of bullet points.]. chatgpt.com

Richmond Alake. (2023). *Loss Functions in Machine Learning Explained*. https://www.datacamp.com/tutorial/loss-function-in-machine-learning

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*.

Yanling, Z., Bimin, D., & Zhanrong, W. (2002). Analysis and study of perceptron to solve xor problem. *The 2nd International Workshop on Autonomous Decentralized System*.