

# ECE 6545 - Deep Learning for Image Analysis - Spring 2025

## Assignment 1 Report

Ananya Ananth  
u1520797

February 22, 2025

### 1 Exercise 1.4 (C) - Fitting a Third-Degree Polynomial

Here we fit a third-degree polynomial to the provided training data  $(x_{ex1\_train}, y_{ex1\_train})$  using least squares regression. The goal was to model the underlying function that generated the data while accounting for noise. We constructed a polynomial equation of the form  $y = a_3x^3 + a_2x^2 + a_1x + a_0$  and estimated the coefficients using a closed-form solution. The results are visualized in Figure 1, where blue points represent the training data, and the red curve represents the fitted polynomial. The polynomial captures the general trend in the data, smoothing out fluctuations while avoiding excessive complexity. The model does a good job of fitting the data without overfitting, striking a balance between bias and variance. This polynomial serves as a baseline for further exploration of how different polynomial degrees impact validation performance.

### 2 Exercise 1.6 (T) - Maximum Likelihood and MSE Equivalence

Maximum Likelihood Estimation (MLE) is a statistical method for estimating the parameters of a model by maximizing the likelihood function. In the context of polynomial fitting, we want to derive the same estimated coefficients as obtained using the Mean Squared Error (MSE) criterion. To achieve this, we make the following assumptions:

- The errors (or noise) in the data follow an independent and identically distributed (i.i.d.) Gaussian distribution with zero mean and constant variance, i.e.,

$$\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{1}$$

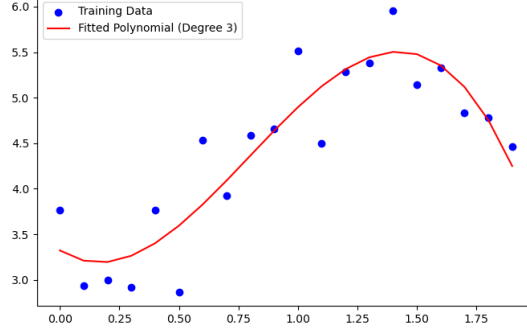


Figure 1: Fitted third-degree polynomial compared to training data

This assumption ensures that deviations from the true polynomial function are normally distributed, making MLE equivalent to minimizing the sum of squared errors.

- The observations  $y$  are conditionally independent given  $x$ . This means that the likelihood function can be expressed as:

$$L(\theta) = \prod_{i=1}^N p(y_i | x_i, \theta) \quad (2)$$

- Taking the logarithm of the likelihood function results in the log-likelihood:

$$\log L(\theta) = -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f(x_i))^2 \quad (3)$$

- Maximizing this function is equivalent to minimizing the sum of squared errors, which is exactly the MSE loss function used in polynomial regression.

Thus, under the assumption of Gaussian noise with constant variance and independent observations, the Maximum Likelihood Estimation of polynomial coefficients is equivalent to minimizing the Mean Squared Error.

### 3 Exercise 1.7 (A) - Model Capacity Analysis

Here we treated the polynomial degree as a hyperparameter and explored its effect on model performance. We trained polynomials of degrees ranging from 1 to 10 and computed the Mean Squared Error (MSE) on both the training and

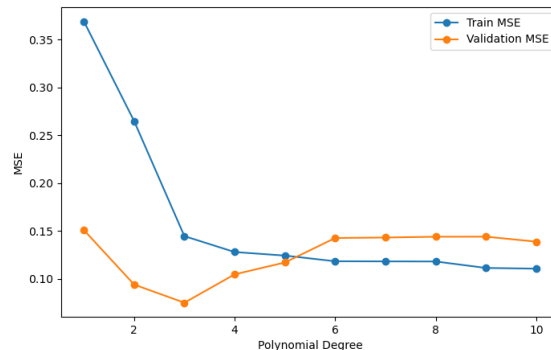


Figure 2: Mean Squared Error vs. Polynomial Degree

validation datasets. The results are visualized in Figure 2, which shows how the training and validation errors evolve as the polynomial degree increases.

From the plot, we observe three distinct patterns:

For **low-degree polynomials** (e.g., degrees 1-2), both training and validation errors remain high. This suggests that the model is too simple to capture the underlying trend in the data, leading to **underfitting**. Such models have high bias and fail to represent the complexity of the dataset.

For **moderate-degree polynomials** (e.g., degrees 3-5), we see that training and validation errors decrease and remain relatively close. This indicates a good balance between bias and variance, as the model is complex enough to capture patterns in the data while still generalizing well to unseen validation data.

For **high-degree polynomials** (e.g., degrees 7-10), the training error continues to drop, but the validation error begins to rise. This is a classic sign of **overfitting**, where the model memorizes the training data instead of learning general patterns. These models exhibit high variance and do not perform well on new data.

In summary, the best-performing polynomial degrees are around 3 to 5, where the training and validation errors remain balanced. Degrees below 3 result in underfitting, while degrees above 6 lead to overfitting. This experiment highlights the importance of selecting an appropriate model complexity to achieve optimal generalization.

## 4 Exercise 2.1 (T) - Number of Parameters in the Network

In this exercise, we calculate the total number of parameters in a fully connected neural network with one hidden layer. The network consists of three layers: an input layer, a hidden layer with activation, and an output layer. The number

of neurons in these layers is determined by three variables:  $n_{inputs}$  (number of input features),  $n_{hidden}$  (number of hidden layer neurons), and  $n_{outputs}$  (number of output neurons).

To compute the total number of trainable parameters, we consider both the weights and biases in each layer:

- The first layer (input to hidden) has a weight matrix of size  $n_{inputs} \times n_{hidden}$ , resulting in  $n_{inputs} \times n_{hidden}$  parameters. Additionally, each hidden neuron has an associated bias, adding  $n_{hidden}$  more parameters.
- The second layer (hidden to output) has a weight matrix of size  $n_{hidden} \times n_{outputs}$ , leading to  $n_{hidden} \times n_{outputs}$  parameters. Similarly, each output neuron has a bias term, contributing  $n_{outputs}$  additional parameters.

Summing up these contributions, the total number of parameters in the network is given by:

$$(n_{inputs} \times n_{hidden}) + (n_{hidden} \times n_{outputs}) + n_{hidden} + n_{outputs} \quad (4)$$

This formula helps us determine the size of the parameter space, which is crucial for understanding the model's complexity and computational requirements. The larger the number of parameters, the more capacity the model has, but also the higher the risk of overfitting if not properly regularized.

## 5 Exercise 2.4 (C) - Stochastic Gradient Descent and Training Process

In this exercise, we implemented the `run_batch_sgd` function, which performs parameter updates using Stochastic Gradient Descent (SGD). The function calculates the gradients of the network parameters using backpropagation and then updates them according to the SGD update rule. The update rule follows the equation:

$$\theta = \theta - \alpha \nabla L(\theta) \quad (5)$$

where  $\theta$  represents the model parameters,  $\alpha$  is the learning rate, and  $\nabla L(\theta)$  is the computed gradient of the loss function with respect to the parameters. The gradients were computed using the chain rule of differentiation during backpropagation, ensuring that each parameter receives an appropriate update based on its contribution to the loss.

After implementing the SGD update function, we used it to train the neural network over multiple epochs. The training process involved iterating through mini-batches of the training data, computing the loss, calculating gradients, and updating the parameters accordingly. This iterative process allows the network to learn by minimizing the error between predicted and actual values.

To visualize the training results, we plotted the neural network's output alongside the training data, as shown in Figure 3. The blue dots represent the

actual training data, while the red line represents the output of the trained neural network. From the plot, we can see that the neural network successfully approximates the underlying pattern in the data, though some deviations are observed due to noise and model constraints.

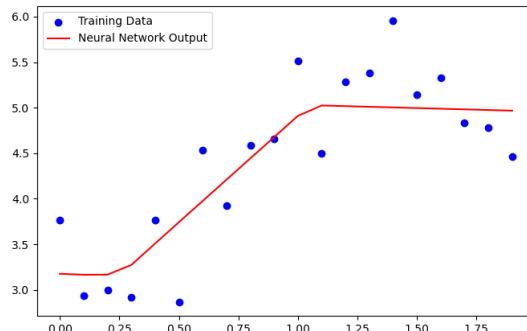


Figure 3: Neural Network Output vs. Training Data

Overall, this experiment demonstrated how a simple two-layer neural network can learn from data using SGD. The effectiveness of the training depends on factors like the learning rate, batch size, and number of epochs. If the learning rate is too high, the model may fail to converge, whereas a very small learning rate might result in slow learning. Tuning these hyperparameters is crucial for achieving optimal performance.

## 6 Exercise 3.2 (A) - Effect of Weight Decay (L2 Regularization) on MNIST Training

In this exercise, we trained a two-layer neural network on a reduced MNIST dataset while applying L2 regularization (weight decay). The goal was to understand how different values of the regularization parameter  $\lambda$  impact the final accuracy of the model. L2 regularization helps prevent overfitting by penalizing large weight values, encouraging the model to learn simpler patterns that generalize better to unseen data.

During training, we experimented with different hyperparameters such as learning rate, batch size, and the number of epochs to optimize performance. The best results were achieved with  $\lambda = 0.001$ , where the model reached a validation accuracy of **91.28%**. This indicates that an appropriate amount of weight decay improved generalization without significantly reducing the model's capacity to learn useful features. When  $\lambda$  was set too high, the model underfit the data, failing to capture complex patterns. Conversely, with little to no regularization, the model overfit the training data, leading to lower validation

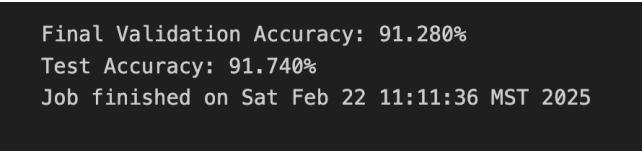
accuracy.

This experiment highlights the importance of properly tuning regularization strength. A well-balanced  $\lambda$  value prevents excessive weight growth while still allowing the model to learn effectively. The results confirm that L2 regularization plays a crucial role in improving the robustness of neural networks for image classification tasks like MNIST.

## 7 Exercise 3.3 (C) - Testing the Best Model on the Test Set

After determining the optimal training configuration, we evaluated the final model on the test set. The goal was to confirm whether the trained network generalized well to completely unseen data. The test accuracy achieved was **91.74%**, which is consistent with the validation accuracy, indicating that the model successfully learned patterns that extend beyond the training data.

Figure 4 shows the recorded accuracy results from the training process, displaying both the final validation and test accuracy values. These results validate that our model is well-trained and benefits from the regularization applied during training.



```
Final Validation Accuracy: 91.280%  
Test Accuracy: 91.740%  
Job finished on Sat Feb 22 11:11:36 MST 2025
```

Figure 4: Final validation and test accuracy of the trained model.

Overall, this experiment demonstrated that with proper tuning of  $\lambda$ , learning rate, and training epochs, a simple two-layer neural network can achieve high accuracy on MNIST. The results also emphasize the importance of using L2 regularization to improve generalization and reduce overfitting.