Implementation of a Feed Forward Neural Network, studied on Sparse Regression and MNIST Classification

Anton Nicolay Torgersen

University of Oslo

(Dated: December 11, 2025)

This study details the implementation of a flexible feed-forward neural network (FFNN) from scratch, featuring configurable layers, activation functions (Sigmoid, ReLU, Leaky ReLU), and optimization algorithms (RMSprop, ADAM). The FFNN is first benchmarked on a sparse regression task of Runge's function, where a tuned $L1$-regularized model achieved a final generalization MSE of -1.82, outperforming analytical OLS, Ridge, and LASSO models. The network is then adapted for classification, achieving 97.51% accuracy on the MNIST dataset. The analysis also highlights the methodological dangers of validation on sparse data and identifies key architectural failure modes, such as the misclassification of "5" as "3", demonstrating the FFNN's lack of spatial awareness.

I. INTRODUCTION

The challenge of function approximation lies at the heart of machine learning, where the goal is to construct models that can accurately capture underlying patterns in data. In a previous study, we explored polynomial regression techniques (OLS, Ridge, and LASSO) to fit Runge's function, $f(x) = 1/(1 + 25x^2)$. While foundational, this approach revealed two significant limitations. It required extensive manual feature engineering (the knowledge of selecting the right model and polynomial degree), and it was highly prone to overfitting, leading to poor generalization. This danger of overfitting is particularly severe when working with sparse datasets, a central theme of our previous work and a key challenge we revisit in this report.

Feed-Forward Neural Networks (FFNNs) offer a powerful solution to the first of these challenges. Their ability to automatically learn relevant features through a layered, non-linear architecture directly addresses the limitations of manual polynomial regression. However, FFNNs are not a simple fix; they introduce their own set of complex-in-practice challenges, particularly in hyperparameter tuning and regularization, to manage the same fundamental bias-variance tradeoff. Given that FFNNs are a dominant method in modern data analysis, a deep, practical understanding of their implementation, tuning, and failure modes is crucial.

This report documents the design and implementation of a flexible FFNN from scratch, built to explore these exact challenges. We validate this implementation on two distinct, complementary tasks:

First, we revisit the sparse regression problem of fitting Runge's function. This provides a direct benchmark against our previous OLS, Ridge, and LASSO models. More importantly, it serves as an extreme case to develop an intuition for the dangers of hyperparameter tuning and regularization when data is scarce, forcing a more extreme work area with analysis on how to achieve robust generalization.

Second, we adapt this validated framework to a well known data classification dataset, i.e. the MNIST hand-written digit dataset. By substituting the cost function (Cross-Entropy) and output layer (Softmax), we test the model's flexibility and analyze its performance in a completely different computational regime, comparing it against standard implementations.

The report is structured as follows, we first detail the methods and theory used to build the FFNN from the ground up. Where one can find the exact implementation in the project's GitHub repository [1]. We then present the results from both the sparse regression and classification tasks, followed by a summary of our findings and a discussion of potential future work.

II. METHODS

This section details the theory and methodology used to implement a flexible feed-forward neural network (FFNN) from scratch. We will first describe the data preparation process for the regression and classification tasks. Following this, we detail the fundamental building blocks of the FFNN, including its architecture, the activation functions (Sigmoid, ReLU, Leaky ReLU), the cost functions (Mean Squared Error and Cross-Entropy), and the gradient-based optimization algorithms (RMSprop, ADAM). Finally, we will discuss the implementation details and code structure.

A. Data Preparation

In this study we analyse two different datasets for two different tasks: regression and classification.

1. Regression: Runge's Function

The datasets used for the regression task are identical to those from our previous study [2]. The data is generated from Runge's function, $f(x) = 1/(1 + 25x^2)$, with x values uniformly distributed in the interval $[-1, 1]$.

As in the previous project, the primary analysis utilized a sparse dataset of $N = 18$ points to clearly demonstrate the challenges of overfitting. Two versions of this dataset were prepared: a primary noiseless set and a noisy set generated by adding Gaussian stochastic noise ($\epsilon \sim N(0, 0.1)$). Where we will be using the noisy dataset for the main results, as it better represents real-world scenarios. For all regression experiments, the datasets were consistently split into training and testing sets using an 80/20 ratio to evaluate generalization performance.

For the FFNN, the input feature for the regression task is the one-dimensional value x as compared to the linear regression case where we created multiple features from x (e.g., x^2, x^3). We then standardized the input feature to have zero mean and unit variance, which helps improve the convergence of the training process.

2. Classification: MNIST Dataset

For the classification task, we used the MNIST dataset of handwritten digits [3]. This dataset, containing 70,000 28x28 grayscale images, was loaded and preprocessed for the network. First, the pixel values of the features were normalized from their original [0, 255] range to [0, 1] to improve numerical stability and training performance. Second, the target labels were converted from single digits to a one-hot encoded vector format. This format is required for the categorical cross-entropy loss function, which compares the network's output probability distribution to the true label. Finally, the full, preprocessed dataset was split into training and testing sets using an 80/20 ratio to evaluate generalization. The complete implementation of this data preparation pipeline can be found in the project's GitHub repository [1].

B. FFNN Method

A feed-forward neural network (FFNN) is a model that approximates a function by composing multiple simpler functions. The network is organized into layers: an input layer, one or more hidden layers, and an output layer, as shown in Fig. 1.

1. Prediction

The process begins with the *feed-forward pass*. For a given input vector $\boldsymbol{x}$, the activation $\boldsymbol{a}^l$ of layer l is computed based on the activation of the previous layer $\boldsymbol{a}^{l-1}$ (where $\boldsymbol{a}^0 = \boldsymbol{x}$):

$$\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$$

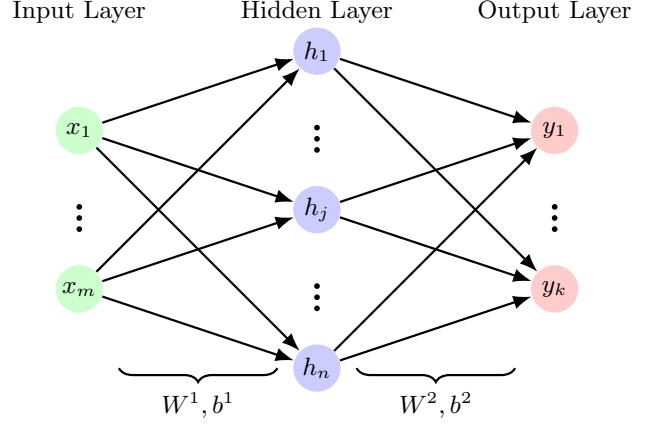$$\boldsymbol{a}^l = f_l(\boldsymbol{z}^l)$$

Figure 1. A simple Feed Forward Neural Network (FFNN) architecture with one input layer, one hidden layer, and one output layer. Each neuron in a layer is connected to every neuron in the subsequent layer through weights (W) and biases (b).

where $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$ are the weight matrix and bias vector for layer l, and f_l is the activation function. This process is repeated until the final output $\boldsymbol{a}^L = \hat{\boldsymbol{y}}$ is computed.

2. Learning

The network "learns" by minimizing a *cost function* $\mathcal{C}(\boldsymbol{W}, \boldsymbol{b})$, which measures the discrepancy between the predicted output $\hat{\boldsymbol{y}}$ and the true target $\boldsymbol{y}$. This minimization is achieved using gradient-based optimization.

The *backpropagation* algorithm is used to efficiently compute the gradient of the cost function with respect to every weight and bias in the network. It is an application of the chain rule, starting from the output layer and moving backward, here for a single training example:

1. Compute the error $\boldsymbol{\delta}^L$ at the output layer L:

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}^L} \mathcal{C} \odot f_L'(\boldsymbol{z}^L)$$

2. Propagate the error backward to compute the error $\boldsymbol{\delta}^l$ for layer l:

$$\boldsymbol{\delta}^l = ((\boldsymbol{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot f_l'(\boldsymbol{z}^l)$$

3. The gradient of the cost function with respect to the parameters of layer l is then:

$$\nabla_{\boldsymbol{W}^l} \mathcal{C} = \boldsymbol{\delta}^l (\boldsymbol{a}^{l-1})^T$$

$$\nabla_{\boldsymbol{b}^l} \mathcal{C} = \boldsymbol{\delta}^l$$

These gradients are then used by an optimization algorithm to update the weights and biases.

3. Hyperparameters

Just as with project 1, there are several hyperparameters to tune when setting up the FFNN, to get the best results.

- **Learning Rate (η):** Controls how much to change the model in response to the estimated error each time the model weights are updated.

- **Number of Epochs:** The number of times the learning algorithm will work through the entire training dataset.

- **Batch Size:** The number of training examples utilized in one learning iteration.

- **Number of Hidden Layers and Units:** The architecture of the network, including how many hidden layers to use and how many neurons in each layer.

- **Activation Functions:** The choice of activation functions for each layer (e.g., ReLU, sigmoid, tanh).

In the results we will go through them to find a good approximation of the optimal parameters. It is believed that deeper networks with more layers can capture more complex patterns, but they are also harder to train.

For more on FFNNs see chapter 1 and 2 in [4, Nielsen].

C. Different Building Blocks

1. Cost Functions

Mean Squared Error (MSE) For regression tasks, we use the Mean Squared Error (MSE) cost function, which is identical to the one used in [2]. For a set of n samples, it is:

$$\mathcal{C}_{MSE}(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Its derivative with respect to a single final activation $\hat{y}_i$ (which is a_i^L) is computed as:

$$\frac{\partial \mathcal{C}_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

where $\mathcal{C}_i = (y_i - \hat{y}_i)^2$ is the per-sample cost.

We extend this cost function to include L_1 and L_2 regularization, which helps prevent overfitting by penalizing large weights. The parameter λ controls the strength of the regularization, see 3.4 [5].

L1 Regularization (LASSO): The L_1 norm adds a penalty proportional to the absolute value of the weights, which encourages sparsity. This is the same technique used in LASSO regression in [2]. The cost function becomes:

$$\mathcal{C}_{L1} = \mathcal{C}_{MSE} + \frac{\lambda}{2n} \sum_{l=1}^{L} \sum_{ij} |W_{ij}^l|$$

The gradient of this term is:

$$\nabla_{\boldsymbol{W}^l} \mathcal{C}_{L1} = \nabla_{\boldsymbol{W}^l} \mathcal{C}_{MSE} + \frac{\lambda}{2n} \cdot \text{sign}(\boldsymbol{W}^l)$$

Where handling the non-differentiable L1 norm requires using a sub-gradient. For optimization, we use the sub-gradient of the penalty term, which is $\frac{\lambda}{2n} \cdot sign(W^l)$, see 3.4.4 [5].

L2 Regularization (Ridge): The L_2 norm adds a penalty proportional to the square of the weights. This is the same regularization technique used in Ridge regression in [2]. The cost function becomes:

$$\mathcal{C}_{L2} = \mathcal{C}_{MSE} + \frac{\lambda}{2n} \sum_{l=1}^{L} \sum_{ij} (W_{ij}^l)^2$$

The gradient of this term, added during backpropagation, is:

$$\nabla_{\boldsymbol{W}^l} \mathcal{C}_{L2} = \nabla_{\boldsymbol{W}^l} \mathcal{C}_{MSE} + \frac{\lambda}{n} \boldsymbol{W}^l$$

2. Activation Functions

Disregarding the linear activation function used in the output layer for regression, we have three commonly used activation functions for the hidden layers of the network. These are non-linear so that the FFNN model doesn't become the same as a network with zero hidden layers. i.e. matrices multiplied together is still a matrix.

Sigmoid An exponential function that has a range from 0 to 1, and a continues derivative. A common problem with this method is diminishing gradients where the network will struggle to learn for large gradients.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

ReLU The Rectified Linear Unit (ReLU) is a piecewise linear function that outputs the input directly if it is positive; otherwise, it will output zero. This function has become the default activation function for many types of neural networks because a model that uses it can often converge faster than those using sigmoid or tanh functions.

$$\text{ReLU}(z) = \max(0, z)$$

Leaky ReLU The Leaky ReLU is a variant of the ReLU that allows a small, non-zero gradient α when the input is negative. This helps to mitigate the dying ReLU problem, where neurons can become inactive and stop learning entirely.

$$\text{Leaky ReLU}(z) = \max(\alpha z, z)$$

For more on activation functions see 14.12 in the lecture notes [6].

3. Optimization Algorithms

In a previous paper [2], we implement three different gradient-based optimization algorithms to update the weights and biases of the gradient descent. Where the most important where Stochastic Gradient Descent (SGD), RMSprop and Adam. These methods are widely used in training neural networks due to their efficiency and effectiveness in handling large datasets (SGD) and adapting learning rates (RMSprop and Adam).

Stochastic Gradient Descent (SGD) is an optimization method that approximates the full batch gradient by utilizing a small, randomly sampled subset of the data, known as a mini-batch B_t of size m, at each iteration. This introduces stochastic noise into the optimization path. The update rule for SGD is the same as for standard gradient descent, but the gradient is computed over the mini-batch. See 8.3.1 in [7] for more details.

RMSprop is an optimization method designed to address the aggressively diminishing learning rates found in other adaptive methods like AdaGrad. It approximates a running average of the squared gradients by utilizing an exponentially decaying moving average, E_t, at each iteration. This helps prevent the learning rate from diminishing too quickly. The update rule for RMSprop scales the gradient by the inverse of this moving average:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E_t + \epsilon}} \odot \nabla_\theta J(\theta_t)$$

where E_t is the moving average of the squared gradients and ϵ is a small smoothing constant. See 8.5.2 in [7].

Adam (Adaptive Moment Estimation) is an optimization method that combines the momentum method with the adaptive learning rate scaling of RMSprop. It maintains two exponentially decaying moving averages: a moving average of the gradients (m_t, the first moment) and a moving average of the squared gradients (v_t, the second moment). Adam is often considered a highly efficient and stable general-purpose optimizer. The update rules are:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla_\theta J(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla_\theta J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_{t+1} + \epsilon}} \odot m_{t+1}$$

where β_1 and β_2 are decay rates for the moments and ϵ is a small smoothing constant. See 8.5.3 in [7].

The implementation of these methods for the FFNN followed a straightforward approach using the update rules or restrictions above. See the GitHub repository [1] for the exact implementation.

D. Image Classification

When applying our FFNN model to an image classification task, one only needs to make a few adjustments to the architecture and cost function to suit the nature of classification problems. Which we will then demonstrate on the MNIST dataset of handwritten digits. This alternations to the architecture follow the book shown in [4] chapter 3.

1. Output layer

For multiclass classification tasks, the output layer of the FFNN is modified to use the softmax activation function. The softmax function converts the raw output scores (logits) from the final layer into probabilities for each class. Given an input vector $\boldsymbol{z}$ from the final layer, the softmax function is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

for each class i. This ensures that the output probabilities sum to 1, making it suitable for multiclass classification tasks.

2. Cost Function

For classification tasks, we replace the mean-squared error cost function with the cross-entropy loss function. The cross-entropy loss measures the discrepancy between the predicted class probabilities and the true class labels. For a single training example with true label y (one-hot encoded) and predicted probabilities $\hat{\boldsymbol{y}}$, the cross-entropy loss is defined as:

$$\mathcal{C}_{CE} = -\sum_i y_i \log(\hat{y}_i)$$

where y_i is 1 if the true class is i and 0 otherwise. This loss function encourages the model to assign high probabilities to the correct classes while penalizing incorrect predictions.

3. Learning

These changes also affect the backpropagation process. Where we now calculate the gradient of the cross-entropy loss with respect to the softmax outputs. Lucky for us if we combine the softmax activation with the cross-entropy loss, the gradient simplifies to:

$$\frac{\partial \mathcal{C}_{CE}}{\partial z_i} = \hat{y}_i - y_i$$

This simplification makes the backpropagation process more efficient, as it avoids the need to compute the derivative of the softmax function separately. For the exact derivation of this see [8].

E. Implementation and Code

All code for this project was written in Python and is available in the GitHub repository [1]. Where we made use of the NumPy library for numerical computations [9], sklearn for machine learning utilities [10], and Matplotlib for plotting results [11]. The implementation is organized into two main parts:

- A Python file (Functions.py) containing the core NeuralNetwork class, all activation and cost functions, their derivatives, and optimization algorithms.

- Three Jupyter Notebooks used for running experiments, generating the results, and plotting the figures presented in this report.

1. FFNN Implementation

The FFNN was implemented from scratch using NumPy for all numerical computations. The core of the implementation is a NeuralNetwork class, which is designed to be modular and flexible. Upon initialization, it allows for easy adjustment of:

- The number of layers and nodes per layer.

- The activation function for each layer (Sigmoid, ReLU, Leaky ReLU).

- The cost function (Mean Squared Error for regression, Cross-Entropy for classification).

- The optimization algorithm (SGD, RMSprop, ADAM).

- Regularization type (L1 or L2) and strength.

This modularity allows the same core class to be used for both regression and classification tasks by simply changing the cost function and final activation function.

2. Testing and Comparison

Before analyzing the model's performance, the correctness of the implementation was verified. The most critical component, the manually implemented backpropagation algorithm (compute gradient), was tested against automatic differentiation. Using the autograd library, we computed the gradients numerically and compared them to our analytical gradients. As detailed in the Verification.ipynb notebook [1], the difference between the gradients was found to be on the order of 10^{-16} or lower, confirming that our backpropagation implementation is correct to a high degree of numerical precision.

Furthermore, a simple training run was conducted to verify that the cost function decreases over epochs and compared against the results from sklearn's MLPRegressor and our own implementation. This discrepancy is likely due to differences in default hyperparameters that are non-trivial to align, but the model was still comparable to the scikit-learn implementation in performance on a dataset with $N = 100$ samples. I.e. there is no large discrepancy in performance that would indicate a faulty implementation.

The final results for the classification task was also compared to the literature on the MNIST dataset, where our results are in line with what is expected for a simple FFNN implementation, implying a correct implementation.

F. Use of AI tools

While writing and correcting the code for this project github copilot was used to help with some of the boilerplate code and finding bugs. For writing the report Gemini 2.5 was used as an editor to help with structuring sentences and paragraphs. Though this editing was only done at the end of writing the complete structure of the report, to avoid any influence on the content itself.

III. RESULTS AND DISCUSSION

Before presenting the results, it is important to note the evaluation methodology used. For all regression experiments, we select the best model from all iterations based on its lowest achieved test MSE score during training. While this is a standard approach for model selection, we will later discuss the potential risks of this method when applied to a very sparse dataset.

A. Training the FFNN for Regression

To begin our exploration of the FFNN's capabilities in regression, we first established a performance baseline with a "naive" model. This configuration used simple gradient descent, a fixed learning rate of $\eta = 0.1$, and the Sigmoid activation function.
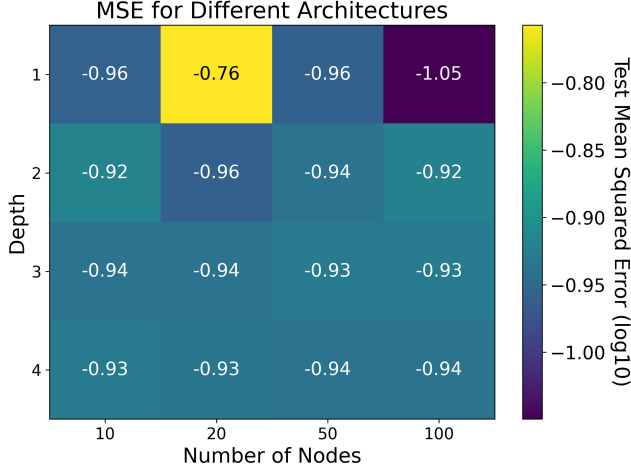
Figure 2. Heatmap showing that the FFNN without any optimizations is struggling to learn anything useful after 1000 epochs of training.

The results, shown in Figure 2, demonstrate that this simple configuration fails to learn the underlying Runge function. The test MSE remains high across all tested architectures, with the best-performing (and simplest) model only achieving a score that is not significantly better than the others.

This failure is characteristic of the well-known vanishing gradient problem. The Sigmoid activation function's saturated outputs stop the flow of gradients to earlier layers, effectively halting the learning process. This hypothesis was confirmed by a longer training run (Figure 3) on a more complex architecture, which revealed that the training score did not begin to decrease until approximately 10,000 epochs, demonstrating an extremely inefficient learning process.

The baseline failure highlights the necessity of modern components to overcome these training challenges. We first investigated the impact of replacing gradient descent with adaptive optimization algorithms: Stochastic Gradient Descent (SGD), Adam, and RMSprop.

As seen in Figure 4, this choice has a dramatic impact. Both RMSprop and Adam, which utilize adaptive learning rates, achieve significantly lower MSE scores than standard GD or mini-batch SGD. RMSprop, in particular, appears to be the most effective for this task.

To isolate the impact of the adaptive algorithm itself from the use of mini-batches, we can compare standard GD against mini-batch SGD. Both methods performed equally poorly, confirming that the superior performance
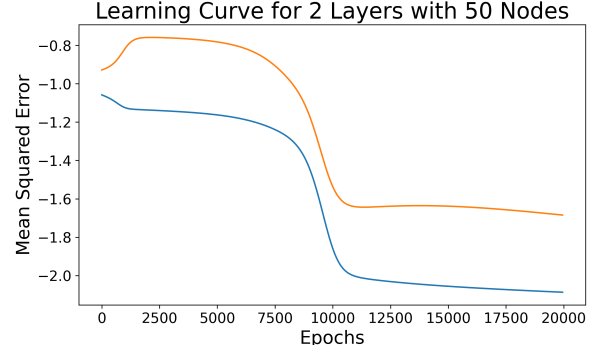
Figure 3. Training and test MSE over 20000 epochs for a FFNN with 2 hidden layers and 50 nodes per layer using plain gradient descent with a learning rate of 0.1 and sigmoid activation function.
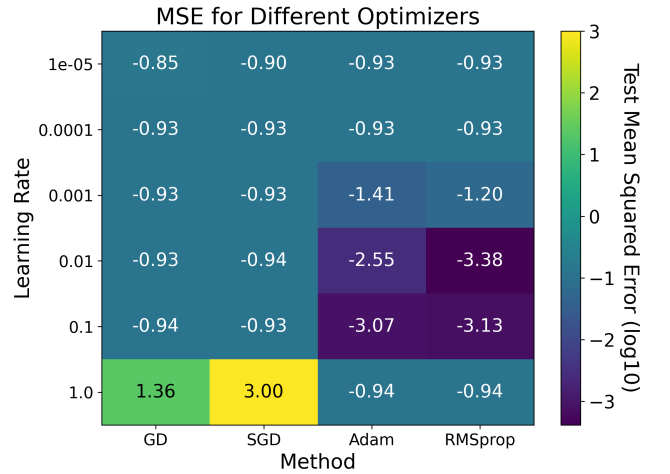
Figure 4. Heatmap showing that the choice of optimization algorithm has a significant impact on the FFNN's performance in learning, trained on 1000 epochs.

of RMSprop and Adam is not merely due to more frequent updates, but stems directly from their advanced adaptive learning rate mechanisms.

Having established the necessity of an adaptive optimizer, we then investigated the interaction between activation functions and learning rates.

Figure 5 reveals two key findings. First, the Sigmoid function consistently underperforms regardless of the learning rate, reinforcing its unsuitability. Second, it illustrates a strong interaction effect: the optimal learning rate is clearly dependent on the chosen activation function. Leaky ReLU, for instance, achieves its best performance at $\eta = 0.01$, while sigmoid performs best at $\eta = 0.1$.

This analysis also highlights the complex interplay between hyperparameters, motivating a more comprehensive grid search to identify optimal configurations.
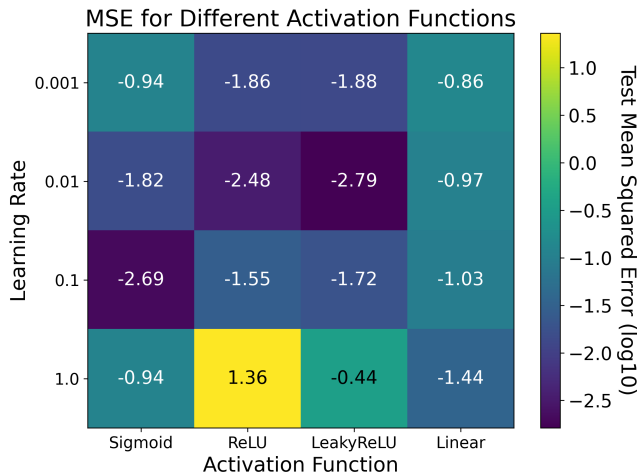
Figure 5. Heatmap showing that change in one hyperparameter can imply a change in the optimal value of another hyperparameter.

B. A deceptive search tuning and regularization on sparse data

The analysis in above established that a functional model requires an adaptive optimizer. The next logical step is to determine the optimal network architecture and, crucially, the correct regularization strategy to prevent overfitting on our sparse dataset.

To find these parameters, we performed a comprehensive, cross-validated grid search over the learning rate, number of layers, nodes per layer, and activation function. The methodology is detailed in the GridSearch.ipynb notebook.

However, it is critical to address the primary challenge of this regression task, the sparse dataset ($N = 18$). Our methodology of selecting the model with the lowest test MSE during training carries a significant risk of overfitting to the test set. The model may learn to fit the specific noise of the 4 test points, not the true underlying function.

Therefore, to obtain a true, unbiased measure of generalization, we adopted a two-stage evaluation:

1. First, use the cross-validated grid search on our "known" dataset to find our contender for the best hyperparameters for the unregularized (L0), L1-regularized, and L2-regularized models.

2. Second, compare the generalization performance of these three "best" models on a new, unseen dataset of 100 data points to see how well they generalize to the actual function.

1. No Regularization

The best unregularized model, table **??**, achieved a final test MSE of -2.834 on the original sparse test set. This test score is significantly higher than the best analytical OLS solution from our previous work (-1.997).

Table I. The distribution of classes in the Thrash dataset.

Class	Count	Percentage
Cardboard	461	
Food Organics	411	
Glass	420	
Metal	790	
Miscellaneous Trash	495	
Paper	500	
Plastic	921	
Textile Trash	318	
Vegetation	436	
Total	**4752**	**100%**

However, the fit in figure 6, shows that the chosen function has fitted to the test points rather than the necessarily the training points. Showing that this naive search for the optimal parameters has overfitted to the test set. We will later see that this overfitting does not translate to poorer generalization on new data, but it is a clear fault in our selection methodology on sparse data.
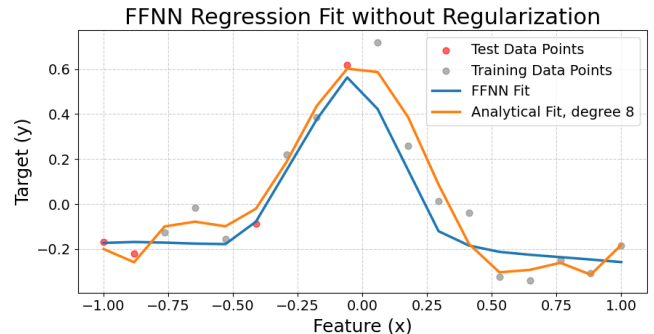

Figure 6. FFNN regression results without regularization on noisy data.

2. L1 Regularization

Echoing the same sentiment as earlier we did a grid search over several hyperparameters to find the best combination for the FFNN with $L1$ regularization. The best result for the regularization $L1$ with a noisy dataset was:

Which was found after 79 epochs of training. If we compare this to what we found in the previous project [2] of -2.10 for the lasso regression after training for 9998 epochs. We see that the FFNN is able to outperform the Lasso solution here as well.

Table II. Hyperparameters used for the neural network with $L1$ regularization

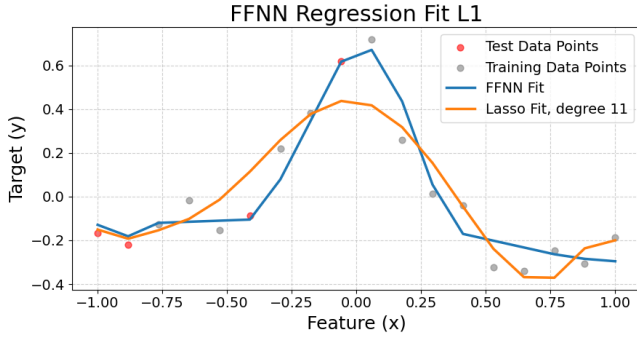| Parameter	Value
Hidden layers	2
Nodes per layer	20
Activation function	LeakyReLU
Learning rate	0.1
Batch size	10
Optimizer	RMSprop
Regularization(λ)	1e-05
Final Cost	**-2.934**

Figure 7. FFNN regression results with L1 regularization on noisy data.

Where we find the same trend as with the previous results [2], that since the data is so sparse the model doesn't benefit much from the regularization. Or at least not in a way that is visible from the test cost on new data.

3. L2 Regularization

Best result for the regularization $L2$ with a noisy dataset:

Table III. Hyperparameters used for the neural network with $L2$ regularization

Parameter	Value
Hidden layers	2
Nodes per layer	10
Activation function	LeakyReLU
Learning rate	0.1
Batch size	3
Optimizer	RMSprop
Regularization(λ)	2e-04
Final Cost	**-2.491**

Based on this search, one would conclude that the L1 model is the best. But given the clear overfitting in Figure 6, and the sparse test data, these scores are unreliable.
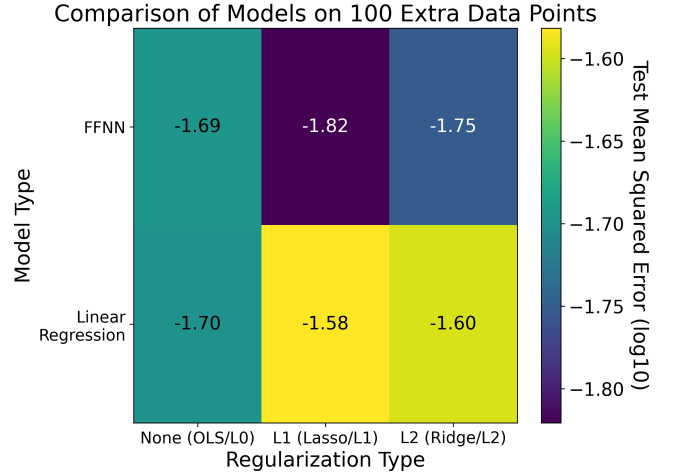

Figure 8. Final comparison of the three FFNN regression models on new data.

4. Final Comparison regression

The limitations of the sparse-set search are fully exposed when we evaluate these three best models on the new, 100-point dataset. This provides an unbiased measure of their true generalization.

The results of this benchmark figure 8 reveal the real story. Where The L1-regularized FFNN is the clear winner, achieving the best generalization score of all models -1.82. It outperforms not only the other FFNNs but also all analytical methods from our previous study.

The unregularized model, which scored an impressive -2.834 on the sparse set, generalizes poorly with a score of -1.715. This confirms that some regularization is indeed necessary to prevent overfitting, even if the sparse-set search failed to reveal this. One should use some amount of regularization to have a more generalizable model, which is especially pronounced problem in sparse datasets.

This benchmark provides a key insight into why the L1-regularized model won. The task involves an over-parameterized model (a 2-layer FFNN with hundreds of parameters) fitting a simple 1D function from extremely sparse data. L2 (Ridge) regularization shrinks all weights but keeps them non-zero, resulting in a complex, "dampened" function. L1 (Lasso), by contrast, induces sparsity by "pruning" weights, forcing many to zero. This performs a type of automatic model selection, simplifying the network.

For this specific problem, L1's aggressive simplification created a less complex model that was less prone to fitting noise and therefore generalized far better. The danger was not just in overfitting, but in trusting a validation metric that was, itself, overfitted.

C. Image Classification

Now we will use the methods we learned from the regression task to implement a FFNN for image classification on the MNIST dataset. Especially the knowledge gleamed around overfitting, the need for regularization and that slow and steady learning is better than fast and unstable learning.

To solve this, we will adapt the output layer to use the softmax activation function and the cross-entropy cost function. Where we in this case will have more data to train on and therefore escape the randomness contained in comparing sparse data sets.

As we experience before finetuning the hyper parameters one by one does not create a network of complex results where the relationship between the two show the full picture of how they interact with each other. I.e. we will try on this larger set to find a good baseline to search the hyperparameter space for the best hyper parameters for the MNIST dataset.

As the MNIST dataset is much larger than the sparse dataset, we needed to use a smaller grid search with fewer epochs. We are also not able to use the whole training dataset for each training run as it would be too computationally expensive. Therefore we used a subset of 1000 samples from the training set for each training run. After running this code for a while we found a suggestion for what the optimal parameters where when trained for 5 epochs on each combination.

Looking at the results that our naive grid search found and then continuing the training run. We saw that this give was a good starting point, but there was room for improvement as it had an accuracy of around 94% and we where able to improve it further by making some adjustments. We therefore continued to tune the parameters further, by alternating some of these parameters to see how these changes develop on a longer training run. One specific change we saw was beneficial was a slightly lower learning rate and a larger batch size to improve the stability in learning. After having done this final fine-tuning we ended up with the following hyperparameters in table IV.

Table IV. Hyperparameters used for the neural network training.

Parameter	Value
Learning rate	0.001
Batch size	64
Hidden layers	3
Nodes per layer	100
Activation function	sigmoid
Optimizer	RMSprop
Regularization Type	L2
Regularization(λ)	1e-03
Final Accuracy	**97.51%**

As the final results was a bit off from our new findings,

this implies the need for a better hyperparameter tuning method, one that perhaps tunes for training runs that has a consistent and steady improvement rather than finding the lowest cost after a few epochs, as running on the whole dataset for many epochs is computationally infeasible in practice.
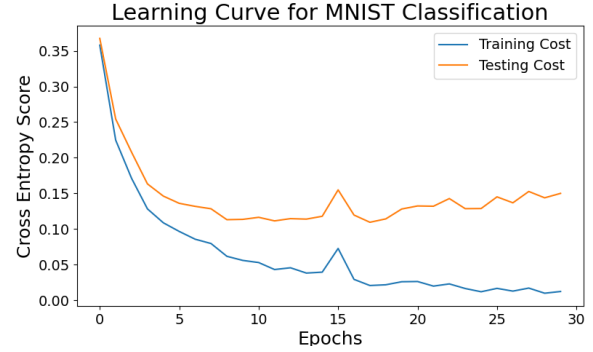

Figure 9. The training and test accuracy and loss for our MNIST classification model

Analyzing our training curve in fig. 9 we see that the model is learning quite well and is not beginning to overfit before after 17 epochs. Where we see that the training accuracy is still decreasing while the test accuracy begins to increase.

If we now want to look at how the model is actually performing on this test set we can look at the confusion matrix in fig. 10. Where we see that the model is performing quite well, with a high percentage being correct predictions, i.e. the diagonal.
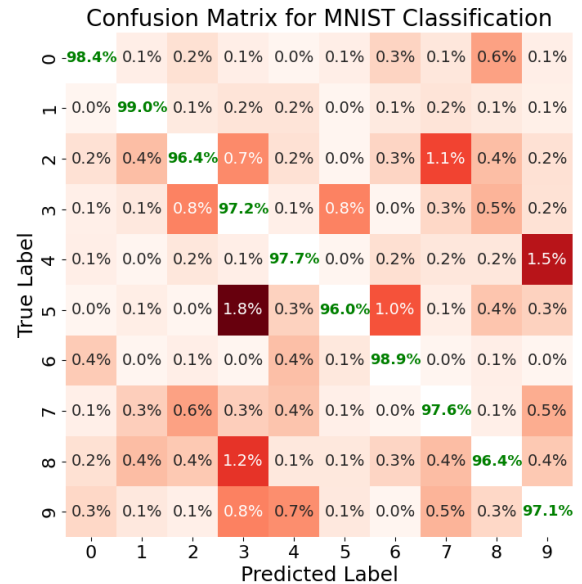

Figure 10. Heatmap of the confusion matrix for our MNIST classification model

Misclassified Images: Predicted 3, Actually 5

True: 5, Pred: 3   True: 5, Pred: 3   True: 5, Pred: 3
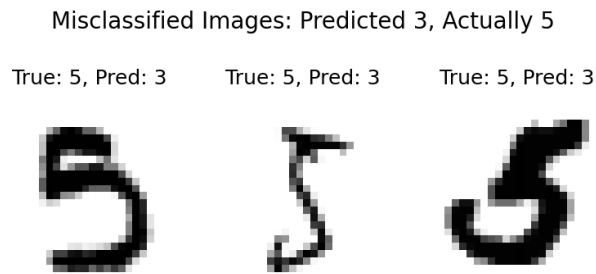
Figure 11. Misclassified 5 as 3 from the MNIST test set

One important observation from this matrix is the mistake of classifying 5 as a 3 is the most common mistake, which is interesting as they don't look that similar. We believe that this is in some part due to the way people write these digits and that the FFNN isn't necessarily trained to capture the same features as we look for. It might learn this through its neurons but it might also learn to classify it in other ways, e.g. the explainability of a prediction is what makes it hard to pinpoint why.

Looking closer at some of these mistakes in fig. 11 we see that there should be some room for improvement here. As all of these looks like 5s, perhaps the middle one can be seen as a 3, and a neural network that is better at capturing spatial hierarchies might help. In view of this there is reason to believe that there is more room for improvements to this classification. Perhaps by using a convolutional neural network (CNN) that is better suited for image data, as it can capture spatial hierarchies in images better than a FFNN.

As a concluding remark, we acknowledge that despite of these remarks the FFNN model is able to perform quite well out of the box on the MNIST dataset with an accuracy of 97.51%. Where it has shown clear capabilities and adaptability to learn complex functions from data.

IV. FURTHER WORK

Some problems that could use a closer study:

a) Finding more cost efficient ways to do hyper parameter tuning.As the naive approach we used is very computationally expensive.

b) How different architectures with regards to layers and nodes per layer affect the learning capabilities of the FFNN. Are triangles better than rectangles? Is it better to have one hidden activation function for all layers or to mix them?

c) How parallelization and GPU acceleration can speed up the training process for FFNNs implemented from scratch.

d) How Convolutional Neural Networks (CNNs) would perform in comparison to FFNNs on image classification tasks.

V. CONCLUSION

Just as with the analytical methods from [2] we found that working on a sparse dataset made it hard to see the benefits of regularization, even with the added noise. However, we still found that the FFNN was able to outperform the analytical methods in both the unregularized and regularized cases. Showing the power of neural networks in adapting to complex functions.

Though the real problem grasped from this project is the hyperparameter tuning. Where we found that the naive approach of freezing all but two hyperparameters and then plotting a heatmap to find the best combination was flawed. While doing an exhaustive grid search was computationally expensive and still didn't guarantee the best results. As the measurement was along a few epochs where the one that had the best run out of these few epochs was chosen. A better approach would perhaps be to see how steady the improvement was over time rather than just looking at the best result after a few epochs, perhaps by measuring slope and variance over time.

We also found that the choice of optimization algorithm had a significant impact on the learning capabilities of the FFNN. Where Adam and RMSprop was needed in order to get good results, as plain GD and SGD struggled to learn anything useful.

For specifically sparse datasets we found that when tuning the hyperparameters one will have a continuous struggle to find the best combination for the underlying dataset as each epoch will have a randomness where it might just inch closer to the test set and away from the training set.

When it comes to the image classification task we found that the FFNN was able to perform quite well on the MNIST dataset, achieving an accuracy of 97.51%. However, if we compare this to the performance of the best solutions to this problem we find that there is still much room for improvement as state-of-the-art models achieve accuracies above 99.7%. Which is even higher than what human performance on this task is believed to be within.

[1] A. N. Torgersen, "Project 2 - fys-stk4155," (2025).

[2] A. N. Torgersen, "A study of regularization and resam-

pling for high-degree polynomial regression on a sparse data set," (2025).

[3] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," (1998).

[4] M. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015).

[5] T. Hastie, R.Tibshirani, and J.Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer Series in Statistics* (Springer, New York, 2009) pp. 389–414.

[6] M. Hjorth-Jensen, *Computational Physics Lecture Notes 2015* (Department of Physics, University of Oslo, Norway, 2015) pp. 301–308.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[8] S. Mehta, "Deriving categorical cross-entropy and softmax," (2023).

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, Nature **585**, 357 (2020).

[10] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning* (2013) pp. 108–122.

[11] J. D. Hunter, Computing in Science & Engineering **9**, 90 (2007).