Evaluating Feedforward Neural Networks for Regression and Multiclass Classification Using Linear Regression as a Baseline for Regression

FYS-STK4155 - Project 2

https://github.com/Malenefk/MachineLearning_GroupBME/tree/main/Project2

Birk Tinius Skogsrud, Elias Micaelsen-Fuglesang and Malene Fjeldsrud Karlsen

University of Oslo

November 14, 2025

Abstract

Feedforward neural networks (FFNNs) are widely used for nonlinear regression and classification, yet their practical performance depends strongly on architectural and training choices. In this project we construct a flexible FFNN framework and evaluate it on a challenging one–dimensional regression task based on the noisy Runge function and on multiclass classification of the MNIST handwritten digit data, using linear regression models as baselines for the regression case. The central challenge is that the Runge function is known to be difficult for polynomial-based regression, and it is unclear a priori how activation functions, optimizers, and regularization should be chosen for stable and accurate learning in both tasks.

To address this, we implement a fully connected network that supports multiple hidden layers, Sigmoid, ReLU and Leaky ReLU activations, MSE and cross-entropy losses with optional L1/L2 regularization, and several gradient-based optimizers including SGD, Momentum, RMSprop, Adagrad and Adam, combined with mini-batching, gradient clipping and different scaling strategies. We systematically scan learning rates, numbers of epochs, batch sizes and regularization strengths, and compare our implementation to scikit-learn's linear models and neural network classes.

For the Runge regression problem we find that a low learning rate $\eta \approx 10^{-4}$, small batch size and the Adam optimizer with Sigmoid or Leaky ReLU activations yield the lowest and most stable train and test MSE, with no clear benefit from explicit L1/L2 regularization beyond a small constant term for numerical stability. For MNIST, Sigmoid hidden units combined with Softmax outputs and a learning rate around 0.0065 give the best trade-off between convergence speed and accuracy, outperforming ReLU-type activations under our initialization scheme and agreeing well with the scikit-learn MLP results. Overall, our results show that carefully tuned FFNNs can outperform linear baselines on both tasks and that optimizer and activation choices matter at least as much as explicit regularization for achieving good generalization.

1. Introduction

In this project, we construct a neural network which can be applied to both regression problems and classification problems. For the regression problem we will use Runge´s function, while for the classification problem we will use the mnist dataset.

In Project 1, it was observed that the Runge function:

$$f(x) = \frac{1}{1 + 25x^2}$$

was challenging to fit using various regression methods. The Runge function illustrates a well-known issue in high-order polynomial interpolation: the approximation tends to oscillate and diverge near the boundaries [6] [22]. This project investigates whether a neural network can outperform the regression methods applied in Project 1 for the Runge function.

The MNIST dataset is widely used for multiclass classification tasks. In this study, the dataset comprises a 70,000 × 784 matrix and a target array of length 70,000 containing integers from 0 to 9. Each of the 70,000 rows represents an image of a handwritten digit, and the 784 columns correspond to the pixel values (28 × 28) indicating pixel brightness. The target array specifies the correct digit label for each image.

The implemented neural network framework supports a range of cost functions, activation functions, and optimization algorithms. The cost functions include mean squared error and cross-entropy, both incorporating regularization terms. Activation functions such as Sigmoid, ReLU, Leaky ReLU, and Softmax are utilized. Sigmoid, ReLU, and Leaky ReLU are applied to hidden layers, while Softmax is used in the output layer for multiclass classification to compute class probabilities. The optimization algorithms employed are Adam, RMSprop, Adagrad, Momentum, and a constant learning rate scheduler.

The aim of this report is to analyze the performance of neural network models for both regression and classification

tasks, with particular attention to errors, accuracies, and output behavior, as well as to identify potential avenues for model improvement. Section 2 presents the theoretical background, including cost functions, activation functions, and the feedforward network with backpropagation. Section 3 details the implementation and experimental setup. Section 4 presents and critically discusses the numerical results and compares different modeling choices.

2. Theory

2.1. Loss Functions

The key feature of supervised learning is that the training data come with known target labels [7] and we seek to quantify how far model predictions deviate from their corresponding target values. A useful way to achieve this is through a loss function, which assigns a cost to each prediction and guides the training process to adjust the parameters $\boldsymbol{\beta}$ in order to minimize the overall risk. Mathematically, if we let $(X, Y) \sim \mathcal{D}$ denote the unknown data-generating distribution. The true risk is known to be the expected loss

$$\mathcal{R}(\boldsymbol{\beta}) = \mathbb{E}_{(X,Y)\sim\mathcal{D}}\big[\ell\big(f(X;\boldsymbol{\beta}),\, Y\big)\,\big]$$

[13, Ch.8] In practice, given i.i.d. samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ from $\mathcal{D}$, we minimize the empirical risk

$$\begin{aligned}\mathcal{J}(\boldsymbol{\beta}) &= \mathbb{E}_{(X,Y)\sim\hat{\mathcal{D}}}\big[\ell\left(f(X;\boldsymbol{\beta}), Y\right)\big]\,, \\ &= \frac{1}{N}\sum_{i=1}^{N} \ell\left(f(\mathbf{x}_i;\boldsymbol{\beta}),\, y_i\right), \\ &= \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{y}_i, y_i)\end{aligned}$$

[13, Ch.8] where $\mathbb{E}_{(X,Y)\sim\hat{\mathcal{D}}}$ denotes the empirical expectation under the empirical measure $\hat{\mathcal{D}}$.

2.1.1. Mean squared error

A particularly natural empirical risk for regression is the mean squared error (MSE), which averages the squared residuals. Squaring makes the loss non-negative and penalizes positive and negative deviations symmetrically, with larger errors receiving disproportionately higher penalties. The empirical MSE can be written as an expectation under the empirical distribution, yielding

$$\begin{aligned}\mathcal{J}_{\text{MSE}}(\boldsymbol{\beta}) &= \mathbb{E}_{(X,Y)\sim\hat{\mathcal{D}}}\Big[\big(Y - f(X;\boldsymbol{\beta})\big)^2\Big]\,, \\ &= \frac{1}{N}\sum_{i=1}^{N}\big(y_i - f(\mathbf{x}_i;\boldsymbol{\beta})\big)^2\,, \\ &= \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.\end{aligned}$$

By denoting that

$$\frac{\partial \mathcal{J}_{\text{MSE}}}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i),$$

we see that the penalty increases linearly with the prediction error.

2.1.2. Cross entropy

The empirical cross-entropy formalizes the idea that a probabilistic classifier should assign high probability to the observed labels while remaining calibrated about uncertainty. We write $\boldsymbol{\theta}$ for the collection of all trainable parameters of the model and for data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$,

$$\begin{aligned}\mathcal{J}_{\text{CE}}(\boldsymbol{\theta}) &= \mathbb{E}_{(X,Y)\sim\hat{\mathcal{D}}}\big[-\log p_{\boldsymbol{\theta}}(Y \mid X)\big] \\ &= -\frac{1}{N}\sum_{i=1}^{N}\log p_{\boldsymbol{\theta}}(y_i \mid \mathbf{x}_i)\,,\end{aligned}$$

is the empirical negative log-likelihood. Minimizing it encourages the model to assign high probability to the observed labels while penalizing overconfident mistakes. With one-hot targets $y_{i,k} = I\{y_i = k\}$ and $\pi_k(\mathbf{x};\boldsymbol{\theta}) := p_{\boldsymbol{\theta}}(Y{=}k \mid X{=}\mathbf{x})$, we get

$$\mathcal{J}_{\text{CE}}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\,\log \pi_k(\mathbf{x}_i;\boldsymbol{\theta}),$$

Because $y_{i,k}$ is one-hot, only the log-probability of the true class contributes for each i, such that all other class terms are zero.

2.1.3. Regularization

With finite and noisy data, highly flexible models can fit spurious patterns as easily as the signal. Regularization reduces overfitting by penalizing overly large parameters [13, Ch.7], which introduces a preference for simpler models. From an optimization viewpoint, the penalty reshapes the landscape. The ridge regulator creates circular level sets that shrink all directions smoothly and encourage small, but non-zero weights [24]. The lasso penalty forms diamond-shaped contours that pull solutions toward the coordinate axes, such that some coefficients can be exactly zero [24]. The circular and diamond-shaped level sets arise because ridge and lasso constrain the Euclidean and absolute norms of the parameter vector, respectively [15, Ch.3]. By altering the geometry of $\mathcal{J}$, regularization directs gradient steps away from high-variance solutions and toward stable ones, exchanging an increased bias for a reduction in variance [20]. We implement this by introducing a penalized empirical risk

$$\mathcal{J}_\lambda(\boldsymbol{\beta}) = \mathcal{J}(\boldsymbol{\beta}) + \lambda\,\Omega(\boldsymbol{\beta}),$$

where Ω measures model complexity and $\lambda \geq 0$ refers to the regularization parameter which controls the strength of the applied penalty. Mathematically, the ridge regularizer can be expressed as

$$\Omega_{\text{ridge}}(\boldsymbol{\beta}) = \|\boldsymbol{\beta}\|_2^2,$$

while the lasso regularizer is defined as

$$\Omega_{\text{lasso}}(\boldsymbol{\beta}) = \|\boldsymbol{\beta}\|_1 = \sum_j |\beta_j|.$$

These penalties yield particularly simple gradient contributions. For ridge regularization, differentiation gives

$$\frac{\partial}{\partial \boldsymbol{\beta}}\big[\lambda\,\Omega_{\text{ridge}}(\boldsymbol{\beta})\big] = 2\lambda\boldsymbol{\beta},$$

while for lasso, the derivative involves the subgradient

$$\frac{\partial}{\partial \beta_j}\big[\lambda\,\Omega_{\text{lasso}}(\boldsymbol{\beta})\big] = \lambda\,\text{sign}(\beta_j),$$

where

$$\operatorname{sign}(\beta_j) = \begin{cases} +1, & \beta_j > 0, \\ -1, & \beta_j < 0, \\ \in [-1,\,1], & \beta_j = 0, \end{cases}$$

reflecting that the absolute-value penalty is not differentiable at zero.

2.2. Linear regression

Linear regression is a simple setting where the empirical risk, measured by the mean-squared error, becomes a concrete model. Given a design matrix $\mathbf{X} \in \mathbb{R}^{N\times d}$, coefficients $\boldsymbol{\beta} \in \mathbb{R}^d$, predictions are $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$ and $\mathcal{J}_{\text{MSE}}$ introduced in 2.1.1 can be represented in matrix form as

$$\mathcal{J}_{\text{MSE}} = \frac{1}{N}\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2.$$

Minimizing this objective corresponds to the ordinary least squares (OLS) problem, which seeks the coefficient vector $\boldsymbol{\beta}$ that best fits the observed data and makes the residuals $\mathbf{y}-\hat{\mathbf{y}}$ small in the least squares sense, coinciding with maximum likelihood under Gaussian noise [18]. Geometrically, OLS projects $\mathbf{y}$ onto the column space of $\mathbf{X}$ [15, Ch.3], producing the unique solution with minimal squared error when the columns of $\mathbf{X}$ are well conditioned. When features are collinear or d is large relative to N, the variance of the OLS estimator can inflate, and the problem becomes numerically unstable [14]. Ridge and lasso regularization tame this by shrinking coefficients as described in 2.1.3. In practice, we either solve the normal equations in closed form or use gradient-based updates. Additionally, the intercept is usually unpenalized, and features are standardized so that coefficient magnitudes are comparable.

Because the least–squares objective is quadratic and convex, the minimizer admits a closed form characterized by the normal equations. Setting the gradient of $\mathcal{J}_{\text{MSE}}(\boldsymbol{\beta}) = \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2$ to zero gives

$$\nabla_{\boldsymbol{\beta}}\mathcal{J}_{\text{MSE}} = -\frac{2}{N}\mathbf{X}^\top(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) = \mathbf{0} \iff \mathbf{X}^\top\mathbf{X}\,\boldsymbol{\beta} = \mathbf{X}^\top\mathbf{y}.$$

If $\mathbf{X}^\top\mathbf{X}$ is invertible, the OLS estimator has the closed-form solution

$$\widehat{\boldsymbol{\beta}}_{\text{OLS}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y},$$

but if $\mathbf{X}^\top\mathbf{X}$ is singular or poorly conditioned, the OLS estimate can instead be obtained using the Moore–Penrose pseudoinverse, yielding

$$\widehat{\boldsymbol{\beta}}_{\text{pinv}} = \mathbf{X}^{+}\mathbf{y},$$

which can be computed, for instance, via singular value decomposition [1].

As previously discussed in 2.1.3, we can reduce variance and improve numerical conditioning by augmenting the least-squares objective with a penalty term on the coefficients. In that case we obtain $\widetilde{\mathcal{J}}_{\text{MSE}} = \mathcal{J}_{\text{MSE}}(\boldsymbol{\beta}) + \lambda\,\Omega(\boldsymbol{\beta})$.

For ridge, $\Omega(\boldsymbol{\beta}) = \|\boldsymbol{\beta}\|_2^2$, which modifies the normal equations to

$$(\mathbf{X}^\top\mathbf{X}+N\lambda\mathbf{I})\,\boldsymbol{\beta} = \mathbf{X}^\top\mathbf{y} \implies \widehat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X}+N\lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}.$$

As the ridge estimator remains strictly convex, it always admits a unique closed-form solution, provided that $(\mathbf{X}^\top\mathbf{X} + N\lambda\mathbf{I})$ is invertible.

For lasso, we instead use $\Omega(\boldsymbol{\beta}) = \|\boldsymbol{\beta}\|_1$, which is convex but non-differentiable at $\boldsymbol{\beta} = \mathbf{0}$. The corresponding optimization problem

$$\min_{\boldsymbol{\beta}} \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1$$

does not have a general closed-form solution and must be solved numerically.

2.3. Softmax regression

In a K-class setting with labels $Y \in 1,\dots,K$, the model assigns to each input $\mathbf{x}$ a vector of real-valued scores called logits $\mathbf{z}(\mathbf{x};\boldsymbol{\theta}) \in \mathbb{R}^K$ with components $z_k(\mathbf{x};\boldsymbol{\theta})$. The softmax transform turns these unconstrained scores into a valid probability vector $\boldsymbol{\pi}(\mathbf{x};\boldsymbol{\theta})$ whose entries are nonnegative and sum to one,

$$\pi_k(\mathbf{x};\boldsymbol{\theta}) = \frac{\exp\big(z_k(\mathbf{x};\boldsymbol{\theta})\big)}{\sum_{j=1}^K \exp\big(z_j(\mathbf{x};\boldsymbol{\theta})\big)}, \qquad k = 1,\dots,K.$$

Only relative differences between logits matter, since adding the same constant to all z_k leaves $\boldsymbol{\pi}$ unchanged because of $\exp(z_k + c) = \exp(z_k)\exp(c)$, and the common factor cancels in the softmax denominator.

When these probabilities are evaluated with the empirical cross-entropy from 2.1.2, the model is trained by maximum likelihood for a categorical distribution [5] parameterized by $\boldsymbol{\pi}(\mathbf{x};\boldsymbol{\theta})$. Substituting the softmax into the loss yields the log-partition structure,

$$\mathcal{J}_{\text{CE}}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^N\sum_{k=1}^K y_{i,k}\,\log\left[\frac{\exp(z_{i,k})}{\sum_{j=1}^K\exp(z_{i,j})}\right],$$

$$= \frac{1}{N}\sum_{i=1}^N\left[-z_{i,\,y_i} + \log\sum_{j=1}^K\exp(z_{i,j})\right],$$

where $z_{i,k} := z_k(\mathbf{x}_i;\boldsymbol{\theta})$, $y_{i,k} = I\{y_i = k\}$ and z_{i,y_i} is the logit corresponding to the true class of sample i. The term $-z_{i,y_i}$ encourages increasing this true-class logit, while the $\log\sum\exp$ normalizer balances all classes to prevent excessive scaling. Differentiating this objective with respect to each logit produces a simple signal,

$$\frac{\partial\mathcal{J}_{\text{CE}}}{\partial z_{i,k}} = \frac{1}{N}\big(\pi_k(\mathbf{x}_i;\boldsymbol{\theta}) - y_{i,k}\big).$$

namely, the difference between the model's predicted probability and the one-hot target.

To ground these ideas in a concrete parameterization, suppose the logits are linear in the input,

$$z_k(\mathbf{x}) \;=\; \mathbf{w}_k^\top\mathbf{x} + b_k \qquad \text{and} \qquad \mathbf{z}(\mathbf{x}) \;=\; \mathbf{W}^\top\mathbf{x} + \mathbf{b},$$

with class weight vectors $\mathbf{w}_k \in \mathbb{R}^d$, bias $\mathbf{b} \in \mathbb{R}^K$, and $\mathbf{W} = [\mathbf{w}_1,\dots,\mathbf{w}_K] \in \mathbb{R}^{d\times K}$. Chaining the logit gradient through this linear map gives

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial \mathbf{w}_k} = \frac{1}{N}\sum_{i=1}^{N}\big(\pi_k(\mathbf{x}_i) - y_{i,k}\big)\,\mathbf{x}_i$$

and

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial b_k} = \frac{1}{N}\sum_{i=1}^{N}\big(\pi_k(\mathbf{x}_i) - y_{i,k}\big).$$

This can equally be represented in matrix form with the design matrix $\mathbf{X} \in \mathbb{R}^{N\times d}$ with rows $\mathbf{x}_i^\top$, the target matrix $\mathbf{Y} \in \mathbb{R}^{N\times K}$, and the predicted probability matrix $\mathbf{\Pi} \in \mathbb{R}^{N\times K}$,

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial \mathbf{W}} = \frac{1}{N}\,\mathbf{X}^\top(\mathbf{\Pi} - \mathbf{Y}), \qquad \frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial \mathbf{b}} = \frac{1}{N}\,\mathbf{1}_N^\top(\mathbf{\Pi} - \mathbf{Y}).$$

Under this objective, learning amounts to reallocating probability mass. The gradient nudges weights in the direction that increases the true-class logit, while the log-partition term enforces global competition among classes. With linear logits the landscape is convex in $(\mathbf{W}, \mathbf{b})$ [23], and prediction reduces to choosing the largest score, $\arg\max_k z_k(\mathbf{x})$. For more details on the derivations of the gradients, see A.1.

2.4. Binomial logistic regression

When the problem reduces to only two possible outcomes, the softmax function of 2.3 simplifies notably. All its coupled exponentials and normalizations shrink to a single comparison. In this binary setting, we have just two logits, $z_{i,0}$ and $z_{i,1}$. Only the difference between the two logits matters, since the softmax is invariant to adding the same constant to both. Thus, the classifier depends solely on

$$s_i := z_{i,1} - z_{i,0},$$

which reveals the preferred class. The corresponding probability is

$$p_i = \sigma(s_i) = \frac{1}{1 + e^{-s_i}}.$$

with s_i equal to the log-odds, $s_i = \log\big[p_i/(1-p_i)\big]$. Large positive margins favor class 1, large negative ones favor class 0. Evaluated with the empirical cross-entropy introduced in 2.1.2, the objective reduces to the binary cross-entropy

$$\begin{aligned}\mathcal{J}_{\mathrm{BCE}}(\boldsymbol{\theta}) &= -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log p_i + (1-y_i)\log(1-p_i)\Big] \\ &= \frac{1}{N}\sum_{i=1}^{N}\Big(\log[1+\exp(s_i)] - y_i s_i\Big),\end{aligned}$$

which is nothing more than the $K{=}2$ specialization of the multiclass loss after collapsing to the single logit difference. The decomposition $\log[1+\exp(s_i)] - y_i s_i$ makes the trade-off explicit. Increasing the margin helps when $y_i{=}1$ and hurts when $y_i{=}0$, with the logarithmic term penalizing overconfident mistakes. Differentiation brings the same clean message as in the multiclass case, the gradient with respect to the logit is simply

$$\frac{\partial \mathcal{J}_{\mathrm{BCE}}}{\partial s_i} = \frac{1}{N}(p_i - y_i), \qquad p_i = \sigma(s_i).$$

where,

$$\frac{\partial \mathcal{J}_{\mathrm{BCE}}}{\partial z_{i,1}} = \frac{1}{N}(p_i - y_i), \qquad \frac{\partial \mathcal{J}_{\mathrm{BCE}}}{\partial z_{i,0}} = -\frac{1}{N}(p_i - y_i),$$

reminding us that only the difference $s_i = z_{i,1} - z_{i,0}$ matters.

To anchor this in a concrete parameterization, fix a reference $z_0 \equiv 0$ and let the margin be linear in the input,

$$s(\mathbf{x}) = \mathbf{w}^\top\mathbf{x} + b, \qquad p(\mathbf{x}) = \sigma(s(\mathbf{x}))$$

so the classifier is a Bernoulli model with a logit link, namely the standard binomial logistic regression. The binary cross-entropy written in logit space,

$$\mathcal{J}_{\mathrm{BCE}}(\mathbf{w}, b) = \frac{1}{N}\sum_{i=1}^{N}\Big(\log[1+\exp(s_i)] - y_i s_i\Big),$$

$$s_i = \mathbf{w}^\top\mathbf{x}_i + b,$$

differentiates cleanly through the linear mapping, producing the corresponding gradients

$$\frac{\partial \mathcal{J}_{\mathrm{BCE}}}{\partial \mathbf{w}} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)\mathbf{x}_i, \quad \frac{\partial \mathcal{J}_{\mathrm{BCE}}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i).$$

Learning, therefore, increases the margin on examples with $y_i{=}1$ and decreases it when $y_i{=}0$, with the sigmoid making parameter updates small when an example is already classified with high confidence. Because the margin is linear, the objective is convex [21] in $(\mathbf{w}, b)$, and the optimization landscape contains a single global minimum.

Regularization introduces a prior preference into the model without breaking its logic. Augment the objective with a penalty on the weights,

$$\widetilde{\mathcal{J}}_{\mathrm{BCE}} = \mathcal{J}_{\mathrm{BCE}}(\mathbf{w}, b) + \lambda\Omega(\mathbf{w})$$

With ridge, $\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2$, the gradients read

$$\begin{aligned}\frac{\partial \widetilde{\mathcal{J}}^{\mathrm{ridge}}_{\mathrm{BCE}}}{\partial \mathbf{w}} &= \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)\mathbf{x}_i + 2\lambda\mathbf{w}, \\ \frac{\partial \widetilde{\mathcal{J}}^{\mathrm{ridge}}_{\mathrm{BCE}}}{\partial b} &= \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i),\end{aligned}$$

and the landscape remains smooth and convex [23]. Switch to lasso, $\Omega(\mathbf{w}) = \|\mathbf{w}\|_1$, and the surface remains convex but becomes sharply edged. Sparsity arises naturally through its subgradient

$$\frac{\partial \widetilde{\mathcal{J}}^{\mathrm{lasso}}_{\mathrm{BCE}}}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)x_{i,j} + \lambda\ \mathrm{sign}(w_j),$$

with $\mathrm{sign}(0) \in [-1, 1]$.

In both cases, the penalty does not break the probabilistic nature of logistic regression. The predicted probabilities still originate from the sigmoid applied to the margin, the gradients still depend on the simple residual term $p_i - y_i$, and learning still works by shifting the linear decision boundary. The only difference is that regularization guides this adjustment by shrinking the weights as discussed in 2.1.3.

2.5. Feed-forward neural networks

Neural networks define a family of parametric functions $f(\mathbf{x};\boldsymbol{\theta})$ that seek to approximate an unknown target mapping f^* from data [13, Ch.6]. Conceptually, they are computational architectures inspired by biological neural systems [19], where interconnected units transmit signals that activate others once a threshold is reached. In the artificial counterpart, this mechanism is modeled by layers of simple processing units called neurons, which transform their inputs through weighted linear combinations followed by nonlinear activation functions.

A standard and historically foundational design is the feed-forward neural network (FFNN). It consists of an ordered sequence of layers through which information flows strictly forward, from input to output, without feedback connections [8]. The first layer receives the input features, one or more intermediate hidden layers perform nonlinear transformations, and the final layer produces the network's prediction. Each neuron in layer l connects to every neuron in the next layer, forming what is known as a fully connected or dense architecture. Each connection carries a trainable weight, representing the strength of influence between units, and each neuron possesses a bias term that shifts its activation threshold.

Let n_l denote the number of neurons in layer l, where layer 0 corresponds to the input and layer and L to the output. The computation proceeds iteratively. For the first hidden layer, the pre-activation values are

$$u_i^{(1)} = \sum_{j=1}^{n_0} w_{ij}^{(1)} x_j + b_i^{(1)}, \qquad i = 1, \dots, n_1,$$

where $w_{ij}^{(1)}$ is the weight from input component x_j to hidden unit i and $b_i^{(1)}$ is its associated bias. The nonlinear activation function of layer 1, denoted by $g^{(1)} : \mathbb{R} \to \mathbb{R}$, then produces the activations

$$a_i^{(1)} = g^{(1)}\big(u_i^{(1)}\big).$$

For deeper layers, each neuron in layer l computes

$$u_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}, \qquad g^{(l)}\big(u_i^{(l)}\big),$$

for $i = 1, \dots, n_l$. The full network is obtained by composing these transformations successively until the output layer is reached.

To express this compactly, we can collect the weights and biases of layer l into a matrix and a vector,

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1\,n_l}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{n_{l-1}\,1}^{(l)} & \cdots & w_{n_{l-1}\,n_l}^{(l)} \end{bmatrix} \in \mathbb{R}^{n_{l-1} \times n_l},$$

$$\mathbf{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix}.$$

With this notation, the pre-activation vector at layer l is

$$\mathbf{u}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)},$$

and the activation is obtained elementwise as $\mathbf{a}^{(l)} = g^{(l)}\big(\mathbf{u}^{(l)}\big)$.

For convenience, we can extend $g^{(l)} : \mathbb{R} \to \mathbb{R}$ to act on vectors by applying it to each component independently.

Defining the affine map of layer l as

$$\boldsymbol{\phi}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)},$$

we can represent the overall network as a composition of affine transformations and nonlinearities,

$$\mathbf{F}(\mathbf{x}) = g^{(L)}(\boldsymbol{\phi}^{(L)}(\cdots g^{(1)}(\boldsymbol{\phi}^{(1)}(\mathbf{x}))\cdots))$$

where $\boldsymbol{\theta} = \{\boldsymbol{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ collects all trainable parameters. This architecture defines a differentiable map $\boldsymbol{F} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ that can be optimized to minimize a loss function.

A central theoretical result underpinning the expressive power of such networks is the Universal Approximation Theorem. It asserts that a feed-forward network with at least one hidden layer containing finitely many neurons can approximate any continuous function on a compact subset of $\mathbb{R}^d$ to arbitrary accuracy, provided the activation function is chosen appropriately [13, Ch.6]. The theorem highlights that the depth and nonlinearity of neural networks grant them a powerful ability to capture and represent complex relationships between inputs and outputs. Moreover, as we will discuss later, the specific choice of activation function has a critical influence on both the representational capacity and the trainability of the network.

2.5.1. Activation functions

Activation functions should be simple, applied elementwise, almost everywhere differentiable, and avoid saturation across most of their domain to maintain stable gradient flow.

Sigmoid.

The logistic sigmoid is defined by

$$\sigma(x) = \frac{1}{1+e^{-x}}, \qquad \sigma'(x) = \sigma(x)\big(1-\sigma(x)\big).$$

For all $x \in \mathbb{R}$ it holds that $0 < \sigma(x) < 1$, and since $\sigma'(x) > 0$ it is strictly increasing everywhere. The sigmoid is also characterized by

$$\lim_{x\to-\infty} \sigma(x) = 0, \qquad \lim_{x\to+\infty} \sigma(x) = 1,$$

meaning the map saturates near the extreme values. This contributes to vanishing gradients when $|x|$ is large and $\sigma'(x) \approx 0$, making parameter updates small [13, Ch.6] through the chain rule. This can lead to Sigmoid being less effective in deep layers, even though it remains useful for bounded outputs such as Bernoulli probabilities.

ReLU.

The rectified linear unit is defined by

$$\mathrm{ReLU}(x) = \max\{0, x\}, \qquad \mathrm{ReLU}'(x) = \begin{cases} 0 & \text{for } x < 0, \\ 1 & \text{for } x > 0, \end{cases}$$

and is not differentiable at the origin. Its output range is $[0, \infty)$ because $\mathrm{ReLU}(x) = 0$ for $x \le 0$ and it grows without bound as $x \to \infty$. The function is monotonic,

meaning it never decreases, and scales linearly for nonnegative constants, so $\mathrm{ReLU}(\alpha x) = \alpha\mathrm{ReLU}(x)$ when $\alpha \geq 0$. These properties create simple, piecewise linear activations that let gradients pass freely for positive inputs, where $\mathrm{ReLU}'(x) = 1$. This makes training faster compared to activations that saturate.

However, because $\mathrm{ReLU}(x) = 0$ for negative x, some neurons may stop updating, which are known as dead neurons [13, Ch.6]. Despite this, ReLU remains widely used because it is simple, avoids saturation, and helps maintain strong gradient flow through deep networks.

Leaky ReLU.

As a modification to ReLU, the leaky rectified linear unit introduces a small negative slope on the inactive side,

$$\mathrm{LReLU}_\alpha(x) = \begin{cases} \alpha\, x & \text{for } x < 0, \\ x & \text{for } x \geq 0, \end{cases}$$

$$\mathrm{LReLU}'_\alpha(x) = \begin{cases} \alpha & \text{for } x < 0, \\ 1 & \text{for } x > 0, \end{cases}$$

with a fixed coefficient $\alpha \in (0, 1)$. The function is not differentiable at the origin. Its image equals $\mathbb{R}$ because negative inputs map to negative outputs through the linear branch αx and positive inputs pass unchanged. The map is monotone nondecreasing and piecewise linear, and it's positively homogeneous for nonnegative scalars because $\mathrm{LReLU}_\alpha(\gamma x) = \gamma\,\mathrm{LReLU}_\alpha(x)$ for $\gamma \geq 0$.

Leaky ReLU introduces a small negative slope α on the inactive branch, preserving gradient flow for negative preactivations and reducing the risk of dead units [13, Ch.6]. It retains much of ReLU's sparsity and simplicity while mitigating vanishing gradients on the negative side. The choice of α balances sparsity against stability. Smaller values behave more like ReLU, whereas larger ones improve gradient flow. The Leaky ReLU thus serves as a practical alternative when the ReLU suffers from inactive units or unstable early optimization.

2.6. Learning via gradients

Training a neural network involves minimizing an objective function that depends on all tunable parameters. The same principle extends to simpler models such as ordinary least squares, ridge, and lasso regression.

In our notation, the empirical risk is a map $\mathcal{J}(\boldsymbol{\theta}) : \mathbb{R}^p \to \mathbb{R}$, where $\boldsymbol{\theta}$ stacks every weight and bias. Starting from an initialization $\boldsymbol{\theta}_0$, gradient descent updates parameters by stepping along the negative gradient,

$$\boldsymbol{\theta}_{t+1} \;=\; \boldsymbol{\theta}_t \;-\; \eta\, \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t),$$

with learning rate $\eta > 0$ controlling the step size. Iterations continue until a stopping rule is met. This could be defined by requiring a small gradient norm or by imposing a limit on the number of iterations, among other criteria. As the optimization landscape is typically nonconvex, the algorithm typically converges only to a local minimum. Therefore, using randomized initializations, tuning, or employing an adaptive learning rate, and applying SGD can help the optimizer explore a broader region of the parameter space, thereby avoiding being trapped in poor local minima [2].

Stochastic gradient descent

Computing the gradient on the entire dataset can be computationally expensive. A practical alternative is to estimate it from a mini-batch $B \subset \{1, \ldots, N\}$. Define the mini-batch empirical risk

$$\mathcal{J}_B(\boldsymbol{\theta}) \;:=\; \frac{1}{|B|} \sum_{i \in B} \ell\big(f(\mathbf{x}_i; \boldsymbol{\theta}),\, y_i\big),$$

and perform the update

$$\boldsymbol{\theta}_{t+1} \;=\; \boldsymbol{\theta}_t \;-\; \eta\, \nabla_{\boldsymbol{\theta}} \mathcal{J}_B(\boldsymbol{\theta}_t).$$

After shuffling the dataset, one epoch processes all disjoint batches in sequence. Strictly speaking, SGD refers to the case $|B| = 1$, but in modern usage, the term is often applied to any small-batch variant. The stochasticity introduces gradient noise, which reduces the computation per step and, as mentioned, can help traverse shallow local minima. Increasing $|B|$ lowers the variance of the gradient estimate by averaging over more data, but increases the computational cost per update.

A common refinement adds momentum, which smooths the trajectory by accumulating a velocity as running averages of past gradients [13, Ch.8]. With a velocity vector $\mathbf{v}_t$ and coefficient $\gamma \in [0, 1]$,

$$\mathbf{v}_t \;=\; \gamma\, \mathbf{v}_{t-1} \;+\; \eta\, \nabla_{\boldsymbol{\theta}} \mathcal{J}_B(\boldsymbol{\theta}_t), \qquad \boldsymbol{\theta}_{t+1} \;=\; \boldsymbol{\theta}_t \;-\; \mathbf{v}_t.$$

Setting $\gamma = 0$ recovers plain mini-batch gradient descent, while choosing $\gamma > 0$ helps maintain progress through low-curvature regions and tends to damp oscillations [13, Ch.8].

Adaptive learning-rate methods

Rather than use a single global step size, adaptive optimizers rescale updates per parameter using running statistics of past gradients. RMSProp refines AdaGrad to perform better in non-convex settings by replacing the cumulative gradient sum with an exponentially weighted moving average [13, Ch.8], introducing forgetfulness that keeps the algorithm responsive. Adam merges the benefits of RMSProp and momentum by keeping track of exponentially decaying averages of both the gradients (first moments) and their squares (second moments), together with bias corrections [13, Ch.8]. This enables more stable and adaptive updates to both the direction and scale of the parameters.

2.6.1. Backpropagation

Backpropagation computes the gradient of the empirical risk with respect to all network parameters by applying the chain rule through the layered composition $f(\mathbf{x}; \boldsymbol{\theta})$ [13, Ch.6]. Each layer forms a pre-activation by an affine map and then applies its nonlinearity. One can propagate an error signal backward, starting with the output layer first, then successively through the hidden layers, with each layer reusing the signal from the one above. This reverse sweep yields the gradients for every weight matrix and bias vector needed by a gradient-based optimizer.

For a network with pre-activations $\mathbf{u}^{(l)}$, activations $\mathbf{a}^{(l)}$, weights $\mathbf{W}^{(l)}$, biases $\mathbf{b}^{(l)}$, and layer index $l = 1, \ldots, L$ with $\mathbf{a}^{(0)} = \mathbf{x}$ and $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$, the core backprop relations are

$$\boldsymbol{\delta}^{(L)} \;=\; \nabla_{\mathbf{a}^{(L)}} \ell\big(\hat{\mathbf{y}}, \mathbf{y}\big) \;\odot\; g^{(L)\prime}\big(\mathbf{u}^{(L)}\big),$$

$$\boldsymbol{\delta}^{(l)} \;=\; \big(\mathbf{W}^{(l+1)}\big)^{\top}\boldsymbol{\delta}^{(l+1)} \;\odot\; g^{(l)\prime}\big(\mathbf{u}^{(l)}\big),$$

[13, Ch.6] and the corresponding empirical-risk gradients are

$$\nabla_{\mathbf{W}^{(l)}}\mathcal{J}(\boldsymbol{\theta}) \;=\; \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{\delta}_i^{(l)}\,\big(\mathbf{a}_i^{(l-1)}\big)^{\top},$$

$$\nabla_{\mathbf{b}^{(l)}}\mathcal{J}(\boldsymbol{\theta}) \;=\; \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{\delta}_i^{(l)}.$$

Any gradient-based method then updates the parameters. This reverse-mode computation is what enables large neural networks to be trainable in practice.

2.6.2. Performance Metrics for Classification Models

Accuracy score

Accuracy is a commonly used metric for evaluating the performance of classification models. It represents the proportion of correct predictions out of all predictions made [17], providing a straightforward measure of overall model performance.

Given true class labels $y_i \in \{1,\dots,K\}$ and predicted class labels $\hat{y}_i = \arg\max_k \hat{p}_{ik}$ obtained from the predicted class probabilities $\hat{p}_{ik}$, the accuracy score is formally defined as

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N} I\big(\hat{y}_i = y_i\big)$$

[17], where N denotes the total number of samples, and the indicator function $I(\hat{y}_i = y_i)$ evaluates to 1 if the prediction matches the true label and 0 otherwise.

While accuracy offers a simple and effective way to summarize model performance, it can sometimes be misleading in the presence of class imbalance when some classes occur much more frequently than others [3]. In such cases, a high accuracy could indicate the model's bias toward the dominant class rather than true predictive capability.

Confusion Matrix

Another popular tool for evaluating classification models is the confusion matrix. Unlike accuracy, it offers insight into the specific types of errors a model makes, showing not only the number of correct predictions but also the distribution of misclassifications across different classes. For K classes and N evaluated samples, the confusion matrix $\mathbf{M} \in \mathbb{N}^{K\times K}$ has entries

$$\mathbf{M}_{jk} = \sum_{i=1}^{N} I\big(y_i = j,\ \hat{y}_i = k\big),$$

where $I(\cdot)$ denotes the indicator function. Thus, $\mathbf{M}_{jk}$ counts the number of samples whose true class is j and whose predicted class is k. Rows of the matrix correspond to true classes, while columns correspond to predicted classes. By summing each row of the matrix,

$$n_j = \sum_{k=1}^{K}\mathbf{M}_{jk},$$

we obtain the total number of instances associated with class j. The diagonal elements $\mathbf{M}_{jj}$ represent correctly classified samples for each class, while the off-diagonal entries indicate misclassifications.

2.6.3. Bias-Variance Tradeoff

The bias-variance tradeoff is a fundamental concept governing generalization in feedforward neural networks. It describes how model complexity influences two sources of prediction error bias and variance [4]. Bias is an error resulting from overly simplistic model assumptions, known as underfitting, whereas variance is an error caused by excessive sensitivity to small fluctuations in the training data, referred to as overfitting. As a network's capacity increases, for example, by adding more layers or neurons, it can capture more complex patterns, which reduces the bias [15, Ch.11]. However, this often comes at the cost of increased variance. The model may fit spurious noise or oddities of the training set, degrading performance on unseen data [4]. In general, making a model more flexible to reduce bias tends to increase its variance, and vice versa. Therefore, a balance must be struck between the two.

Underfitting refers to a regime characterized by high bias. The network is too limited to learn the underlying structure of the data, leading to poor performance even on the training set. Overfitting corresponds to high variance, meaning the network fits the training data extremely well but fails to generalize to new data, resulting in a significant gap between training and test performance [4]. In practice, there is often an optimal model capacity that achieves the best generalization. If we plot the error on a validation set against model complexity, we typically observe a U-shaped curve [4]. Initially, the error decreases as the model becomes more flexible and bias is reduced, but then increases again once the model becomes too complex and variance begins to dominate [4]. Thus, one aims to choose a network at the optimal point of this curve, balancing underfitting and overfitting. In other words, finding the right tradeoff between bias and variance is crucial for building effective neural networks that perform well on unseen data.

3. Methods

3.1. FFNN Design and Learning procedure

For a single example, the pre-activation and activation at layer l are $\mathbf{u}^{(l)} \in \mathbb{R}^{n_l}$ and $\mathbf{a}^{(l)} \in \mathbb{R}^{n_l}$. In code we process a mini-batch of size B by stacking B single-example vectors as rows to form matrices $\mathbf{X}_B \in \mathbb{R}^{B\times n_0}$, $\mathbf{U}^{(l)} \in \mathbb{R}^{B\times n_l}$, $\mathbf{A}^{(l)} \in \mathbb{R}^{B\times n_l}$. Each row of $\mathbf{U}^{(l)}$ equals the per-sample $\mathbf{u}^{(l)}$ from 2.5.

With $\mathbf{a}^{(0)} = \mathbf{X}_B$, weights $\mathbf{W}^{(l)} \in \mathbb{R}^{n_{l-1}\times n_l}$, biases $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$, and $\mathbf{1}_B$ the B-vector of ones,

$$\mathbf{U}^{(l)} = \mathbf{A}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{1}_B\mathbf{b}^{(l)\top}$$

$$\mathbf{A}^{(l)} = \begin{cases} g^{(l)}(\mathbf{U}^{(l)}), & l < L, \\ \mathbf{U}^{(l)}, & \text{regression (linear output)}, \\ \boldsymbol{\Pi} = \text{softmax}(\mathbf{U}^{(l)}), & \text{classification}, \end{cases}$$

and the mini-batch objective

$$\mathcal{J}_B(\boldsymbol{\theta}) = \frac{1}{|B|}\sum_{i=1}^{B}\ell(\hat{\mathbf{y}}_i, \mathbf{y}_i) + \lambda\,\Omega(\boldsymbol{\theta}),$$

$$\mathcal{J} \in \{\mathcal{J}_{\text{MSE}}, \mathcal{J}_{\text{CE}}\},\ \Omega \in \{\|\cdot\|_2^2, \|\cdot\|_1\}.$$

Backprop in matrix form initializes at the output layer and recurses through hidden layers

$$\boldsymbol{\Delta}^{(L)} = \begin{cases} \frac{1}{B}\,(\boldsymbol{\Pi} - \mathbf{Y}_B), & \text{Softmax-CE,} \\ \frac{2}{B}\,(\mathbf{A}^{(L)} - \mathbf{Y}_B), & \text{MSE.} \end{cases}$$

$$\boldsymbol{\Delta}^{(l)} = \big(\boldsymbol{\Delta}^{(l+1)}(\mathbf{W}^{(l+1)})^\top\big) \odot g^{(l)\prime}(\mathbf{U}^{(l)}),$$

$$\nabla_{\mathbf{W}^{(l)}} \mathcal{J}_B = (\mathbf{A}^{(l-1)})^\top \boldsymbol{\Delta}^{(l)} + \lambda\, \nabla\Omega(\mathbf{W}^{(l)}),$$

$$\nabla_{\mathbf{b}^{(l)}} \mathcal{J}_B = \mathbf{1}_B^\top \boldsymbol{\Delta}^{(l)}.$$

See Algorithm 1 in the appendix for details.

3.1.1. Gradient-Based Optimization in FFNN Training

With the matrix-shaped gradients $\nabla_{\mathbf{W}^l}$ and $\nabla_{\mathbf{b}^l}$ we update the parameters layer-wise using plain SGD, SGD with RMSprop, and SGD with Adam, implemented via the two routines Algorithm 2 and Algorithm 3 shown in the appendix.

In RMSprop we keep, for each layer l, per-parameter exponential moving averages of squared gradients, $\mathbf{S}_W^{(l)}$ and $\mathbf{S}_b^{(l)}$. These tensors have the same shapes as $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ and are initialized to $\mathbf{0}$. At every mini-batch they are updated with decay $\rho \in (0,1)$, and the step is scaled element-wise by $(\sqrt{\mathbf{S}^{(l)}} + \epsilon)^{-1}$ using a small numerical stabilizer $\epsilon > 0$.

In Adam, we instead track, for each parameter, both a first-moment EMA of the gradients, $\mathbf{M}_W^{(l)}, \mathbf{M}_b^{(l)}$, and a second-moment EMA of the squared gradients, $\mathbf{V}_W^{(l)}, \mathbf{V}_b^{(l)}$. All are initialized to $\mathbf{0}$. With decay rates $\mathcal{B}_1$ and $\mathcal{B}_2$ and global step t, we form bias-corrected moments $\widehat{\mathbf{M}} = \mathbf{M}/(1-\mathcal{B}_1^t)$ and $\widehat{\mathbf{V}} = \mathbf{V}/(1-\mathcal{B}_2^t)$, and update parameters using the element-wise ratio $\widehat{\mathbf{M}} \oslash (\sqrt{\widehat{\mathbf{V}}} + \epsilon)$ with $\epsilon > 0$.

3.2. Training of linear baselines

For OLS, ridge, and lasso, we train with their usual objectives

$$\min_{\boldsymbol{\beta}} \ \frac{1}{N_{\text{train}}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 \quad \text{(OLS)},$$

$$\min_{\boldsymbol{\beta}} \ \frac{1}{N_{\text{train}}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_2^2 \quad \text{(ridge)},$$

and

$$\min_{\boldsymbol{\beta}} \ \frac{1}{N_{\text{train}}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1 \quad \text{(lasso)}.$$

We report linear model results via the closed-form estimators for OLS and ridge as described in 2.2, and via the same gradient-based training loop used for the FFNN discussed in 2.6. In the gradient setting we fit coefficients $\boldsymbol{\beta} \in \mathbb{R}^p$ in the model $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$ using mini-batches $(\mathbf{X}_B, \mathbf{y}_B)$. Writing the batch residual as

$$\mathbf{r}_B = \mathbf{X}_B\boldsymbol{\beta} - \mathbf{y}_B,$$

the OLS mini-batch objective

$$\mathcal{J}_B^{\text{OLS}}(\boldsymbol{\beta}) = \frac{1}{B}\,\|\mathbf{r}_B\|_2^2$$

has gradient

$$\nabla_{\boldsymbol{\beta}} \mathcal{J}_B^{\text{OLS}} = \frac{2}{B}\,\mathbf{X}_B^\top \mathbf{r}_B.$$

To obtain ridge we augment the loss with a $\|\cdot\|_2^2$ penalty on the coefficients. The gradient becomes

$$\nabla_{\boldsymbol{\beta}} \mathcal{J}_B^{\text{ridge}} = \frac{2}{B}\,\mathbf{X}_B^\top \mathbf{r}_B + 2\lambda\,\mathbf{I}\boldsymbol{\beta},$$

but in practice, we don't penalize the intercept.

For lasso we instead add a $\|\cdot\|_1$ penalty, using the subgradient $\mathbf{s} = \text{sign}(\boldsymbol{\beta})$ resulting in the gradient

$$\nabla_{\boldsymbol{\beta}} \mathcal{J}_B^{\text{lasso}} = \frac{2}{B}\,\mathbf{X}_B^\top \mathbf{r}_B + \lambda\,\mathbf{s},$$

where

$$s_j = \text{sign}(\beta_j) = \begin{cases} +1, & \beta_j > 0, \\ -1, & \beta_j < 0, \\ 0, & \beta_j = 0. \end{cases}$$

again without penalizing the intercept. Parameter updates are then applied with the same optimizers as in the FFNN experiments with plain SGD, RMSprop, or Adam acting element-wise on $\boldsymbol{\beta}$ and the corresponding batch gradient. We only swap the gradient formula, while mini-batching, shuffling, and the optimizer state remain unchanged. This procedure is described in more detail under A.2.2.

3.3. Performance metrics

3.3.1. Regression

For regression, we use the mean squared error (MSE) as both the training loss for linear outputs and the evaluation metric.

Given targets y_i and predictions $\hat{y}_i = f(\mathbf{x}_i; \boldsymbol{\theta})$, the test MSE on a held-out set $\hat{\mathcal{D}}_{\text{test}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_{\text{test}}}$ is

$$\text{MSE}_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \big(y_i - \hat{y}_i\big)^2 = \frac{1}{N_{\text{test}}} \|\mathbf{y}_{\text{test}} - \hat{\mathbf{y}}_{\text{test}}\|_2^2.$$

3.4. Classification

For multiclass classification with labels $y_i \in \{1, \ldots, K\}$ and FFNN outputs $\text{softmax}\big(\mathbf{u}^{(L)}(\mathbf{x}_i)\big)$, we predict

$$\hat{y}_i \ = \ \arg\max_{k \in \{1,\ldots,K\}} \pi_{i,k}.$$

On a held-out test set $\hat{\mathcal{D}}_{\text{test}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_{\text{test}}}$, the accuracy is

$$\text{Accuracy}_{\text{test}} \ = \ \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} I\,\{\,\hat{y}_i = y_i\,\}.$$

The unnormalized confusion matrix $\mathbf{M} \in \mathbb{N}^{K \times K}$ summarizes class-wise errors

$$M_{j,k} \ = \ \sum_{i=1}^{N_{\text{test}}} I\,\{\,y_i = j,\ \hat{y}_i = k\,\}, \qquad j,k \in \{1,\ldots,K\}.$$

Row-normalized entries reveal the recall for each class

$$\widehat{M}_{j,k} \ = \ \frac{M_{j,k}}{\sum_{k'=1}^{K} M_{j,k'}}, \quad \text{so that} \quad \sum_{k=1}^{K} \widehat{M}_{j,k} = 1.$$

In our implementation, we compute $\hat{y}_i$ by arg max over the softmax probabilities and then form $\mathbf{M}$ by counting $(y_i, \hat{y}_i)$ pairs on the test set.

3.5. Experimental setup

Split, reproducibility, and shuffling.

We partition each dataset into disjoint training and test sets using an 80-20 split via `sklearn`'s `train_test_split`,

fixing the random seed to ensure deterministic comparisons. We also fix the NumPy RNG seed so that initialization and data order are reproducible. During optimization, each epoch reshuffles the training rows using `sklearn.utils.shuffle`, applying the same permutation to inputs and targets. This yields reproducible mini-batches while still randomizing their composition across epochs.

Preprocessing and feature scaling.

Preprocessing is matched to the domain. For MNIST, raw pixel intensities in $[0, 255]$ are mapped to $[0, 1]$ via

$$\widetilde{\mathbf{X}} \;=\; \frac{\mathbf{X}}{255},$$

applied identically to train and test. For tabular regression, we compute feature-wise moments on the training data,

$$\hat{\mu}_j \;=\; \frac{1}{N}\sum_{i=1}^{N} x_{ij}, \qquad \hat{\sigma}_j \;=\; \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_{ij}-\hat{\mu}_j)^2},$$

and standardize both splits using these training moments:

$$\widetilde{x}_{ij}^{\text{train}} \;=\; \frac{x_{ij}^{\text{train}} - \hat{\mu}_j}{\hat{\sigma}_j},$$

$$\widetilde{x}_{ij}^{\text{test}} \;=\; \frac{x_{ij}^{\text{test}} - \hat{\mu}_j}{\hat{\sigma}_j}.$$

This normalization is used for the ridge and lasso to place predictors on a comparable scale. In ordinary least squares, we mean-center the data but do not rescale the features.

$$\widetilde{x}_{ij}^{\text{train}} \;=\; x_{ij}^{\text{train}} - \hat{\mu}_j,$$

$$\widetilde{x}_{ij}^{\text{test}} \;=\; x_{ij}^{\text{test}} - \hat{\mu}_j,$$

This improves numerical conditioning while preserving feature units.

The Runge function.

For a controlled 1D regression benchmark, we use the Runge function

$$f(x) \;=\; \frac{1}{1+25x^2},$$

sampled on a uniform grid $x_i \in [-1, 1]$ generated by `np.linspace`. Observations are corrupted with i.i.d. Gaussian noise $\varepsilon_i \sim \mathcal{N}(0, 1)$:

$$y_i \;=\; f(x_i) + \varepsilon_i$$

and the resulting dataset is then split and centered/standardized following the protocol described above.

Model targets and numerical stability.

For regression, the network uses a linear output and raw targets $\mathbf{y}_B \in \mathbb{R}^{B\times 1}$ and for multiclass classification it uses one-hot targets $\mathbf{Y}_B \in \{0,1\}^{B\times K}$ with a Softmax output. The implementation uses the standard Softmax–cross-entropy shortcut, giving output sensitivity $A^{(L)} - \mathbf{Y}_B$.

Weights are initialized layer-wise from a zero-mean Gaussian, where the first row encodes the bias. The Softmax is computed with a row-wise max shift for numerical stability, meaning for logits $\mathbf{Z} \in \mathbb{R}^{B\times K}$,

$$\mathbf{\Pi}_{i,k} = \frac{\exp\big(\mathbf{Z}_{i,k} - m_i\big)}{\sum_{j=1}^{K}\exp\big(\mathbf{Z}_{i,j} - m_i\big)}, \quad i = 1,\dots,B,\; k = 1,\dots,K,$$

where

$$m_i = \max_{1\le j\le K} \mathbf{Z}_{i,j}.$$

4. Results and discussion

4.1. Regression

4.1.1. Sigmoid Activation

For the regression tasks, we used the Runge function with scaled training and test data to produce results with n = 500 data points for each training set. We have used Sigmoid as an activation for the hidden layers and a standard regression as the output activation in this subsection. We also use Stochastic Gradient Descent (SGD) througout the project, as the training errors were considerably lower using this approach.

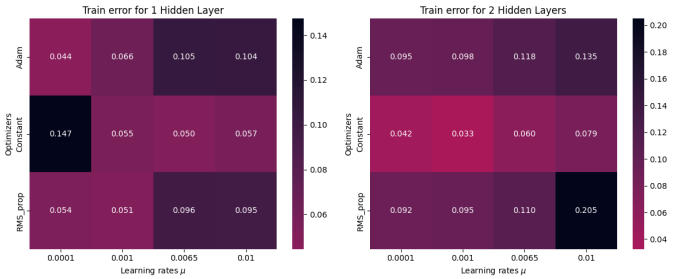

Figure 1: Heatmap showing the training errors with gradient optimizers on the y-axis and varying learning rates μ on the x-axis.

When beginning to train our network, we performed several tests to determine which gradient optimizer performed best over 2000 epochs, while also updating the learning rate μ. The results in Figure 1 1 show low training errors, indicating that it performs well for several values of μ. In particular, the Adam and RMSprop optimizers perform the best with lower learning rates for both one and two hidden layers, while the constant optimizer produces more unstable results. When comparing these results to the test errors calculated from the mean squared error, we find that the training errors are slightly larger than the test errors. This could be due to the natural variability between the test and training datasets, suggesting that the neural network adapts well to new unseen data. It can also indicate that the test data might be of a simpler form than the training data. A majority of the test errors are below 10^{-3}, indicating good performance on unseen data and no clear signs of overfitting.

Using two hidden layers produces a slightly higher training error and test error for all learning rates when tested with 2000 epochs. It is of interest to research how this changes with an increasing number of iterations to check whether the increased complexity of a two hidden layer network needs longer training. We've achieved this by maintaining a steady learning rate of $\mu = 0.0001$, which was identified as the optimal learning rate in our previous results. The results presented in Figure 2 show the training and test errors for the network with two hidden layers.

Figure 2 shows that the network with two hidden layers achieves better results as the number of epochs increases. Both the training errors (left) and the test errors (right) decrease significantly with more epochs, indicating that a two-layer model benefits from a longer training period. For a smaller number of epochs, RMSprop and Adam produce the lowest errors, which continue to decrease as the number of epochs increases. Although the constant optimizer does not perform as well for a lower number of epochs, it

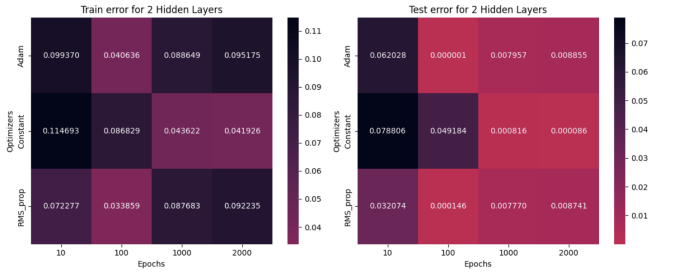

Figure 2: Heatmap showing the training errors (left) and test errors (right) with the gradient optimizers on the y-axis and varying number of epochs on the x-axis.

improves noticeably when the number of epochs exceeds 1000 and yields slightly better results than other optimizers. Running the same experiment for the one hidden layer shows that the training and test errors also improve with more epochs, although the improvement is not as significant as for the two hidden layers.

To further test the network's performance, we experimented with changing the batch size and the regularization parameter λ. Increasing the batch size did not result in any significant changes in the training error for the one-hidden-layer model, but it slightly increased the error for the two-hidden-layer model. Regarding the regularization term, changing it from its default value of 0.01 did not improve the results. By including the regularization term throughout the training, we obtained numerical stability, but it did not significantly improve or worsen the results. The regularization parameter was therefore kept at its default value throughout the experiment, except for when we explicitly test the L1 and L2 regularization, as discussed in relation to Figure 6 and 7. The batch size is also maintained at 4 for the remainder of the project.

Additionally, when testing the neural network, we encountered several cases of overflow errors when using the Sigmoid function as the activation function for the hidden layers. Therefore, we introduced gradient clipping to the neural network, where the gradients are rescaled if their value is higher than a given tolerance [16] [11]. Implementing gradient clipping helped lower the training error and eliminated all cases of overflow error.

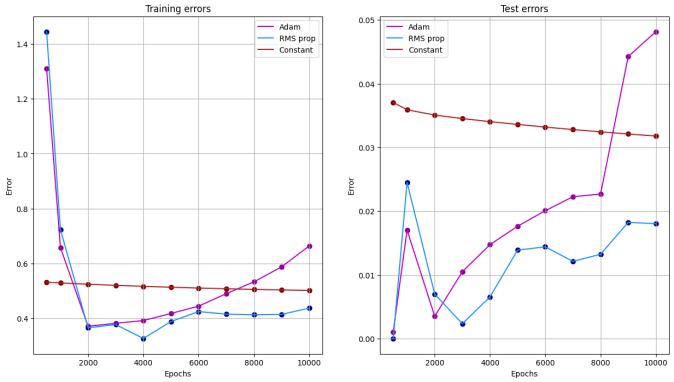

Figure 3: Line-plot showing the test and train errors with number of epochs on the x-axis and the error on the y-axis. The figure includes the following gradient optimizers: Adam (Pink), RMSprop (Blue), and Constant (Maroon).

Figure 3 shows signs of overfitting for varying epochs, as the number of epochs increases for the different gradient optimizers. The figure shows a gradual increase in the test errors for Adam and RMSprop, while the training error remains low. This can be seen as an early sign of overfitting [9] [12].

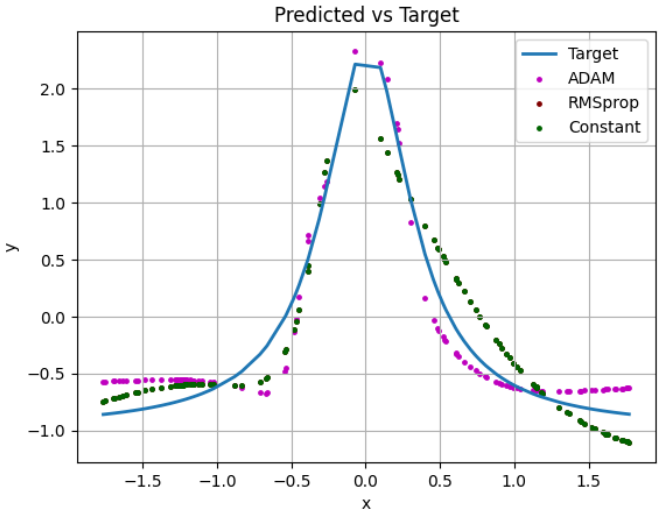

Figure 4: Runge function plotted against scatter plots of the gradient optimizers: Adam (Pink), RMSprop (Blue), and Constant (Maroon). The maroon color for RMSprop is partly hidden underneath the Constant scatter points.

Predictions from the Adam, RMSprop, and Constant optimizers were plotted against the Runge function for 2000 epochs and a learning rate of 0.0001. Figure 3 demonstrates that Adam (pink) most closely approximates the Runge function, although the Constant (green) and RMSprop (maroon) optimizers also perform well. Overall, all three optimizers yield reliable results.

4.1.2. RELU and the Leaky RELU

Moving on, we conducted additional experiments using the RELU and the Leaky RELU (LRELU) as activation functions instead of the Sigmoid function. Since the tolerance value chosen to distinguish between RELU and LRELU is set to a small value, we only include the results for the LRELU, as it performs slightly better than the RELU. We once again tested how the gradient optimizers perform with varying values of μ as shown in Figure 5.

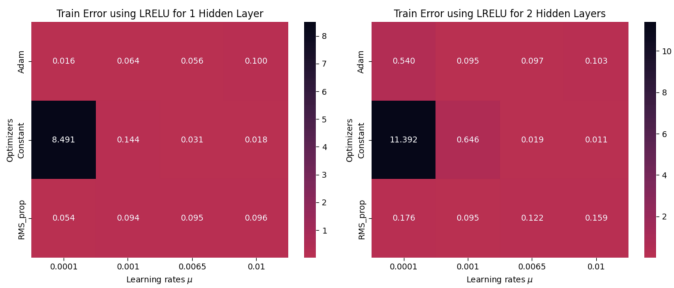

Figure 5: Heatmap showing the training errors with gradient optimizers on the y-axis and varying learning rates μ on the x-axis using the Leaky RELU as an activation function for the hidden layers

We can make the same conclusions as in Figure 1. Adam and RMSprop perform overall best with smaller test and

training errors, while the Constant optimizer remains unstable, but performs reasonably well for higher learning rates. The optimal learning rate for both Adam and RMSprop is $\mu = 0.0001$.

During the experiments, we initially employed a standard scaling method, which involved calculating the standard deviation and mean, resulting in significantly higher values for both the training and test errors. We decided to switch to a new scaling method, using the minimum and maximum values of x and y, which produced significantly better results, as presented in Figure 5. It is, however, interesting to note that using a standard scaling in the first experiment effectively illustrated the optimizer's behavior, even though the errors differ from those of the new method. This highlights a sensitivity to the scaling method, and the rest of the training is based on the min/max scaling.

4.1.3. L1 and L2 regularization

Figure 6: Heatmap showing the training errors (left) and test errors (right) for the gradient optimizer Adam with L1 regularization for varying learning rates μ on the x-axis and varying λ on the y-axis.

Furthermore, we have applied the L1 and L2 regularizations to our neural network, and we see a clear difference between the test and the training errors for these two cases, where the test errors are much smaller than the training errors. Focusing on the Adam optimization, we have tested how these errors evolve for different values of lambda and learning rates. The training performs well for lower learning rates and lower values for the regularization parameter λ. However, it explodes as λ increases even slightly, indicating that the gradients might be penalized to high. L1 and L2 seem to work better with a lower penalty, though it does not measure up to the results found without the L1 and L2 regularizations.

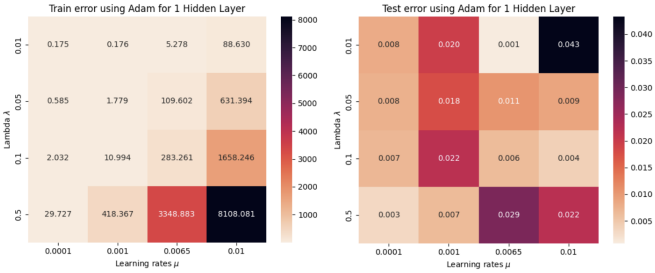

Figure 7: Heatmap showing the training errors (left) and test errors (right) for the gradient optimizer Adam with L2 regularization for varying learning rates μ on the x-axis and varying λ on the y-axis.

The L1 and L2 regularizations are also implemented for two hidden layers, but as the network with one layer showed more stable and lower errors, we have decided to focus mainly on these results to avoid including unnecessary model complexity if it's not needed.

4.1.4. Comparison with Project 1

From our results, we find that not using any L1 or L2 regularization yields the best performance for the Runge function in this neural network. The errors from subsection 1 are numerically stable, whereas applying either L1 or L2 regularization leads to considerably higher errors. While the results improve slightly when both the regularization parameter and the learning rate have a low value, the model without these regularizations gives the best results overall. These findings are interesting as they contrast with the results from project 1, where we concluded that neither the Ordinary Least Squares (OLS) nor Ridge Regression performed particularly well, but that Ridge was still the better option with slightly better results. When using our neural network, however, we find that the network performs the best without added regularizations. These outcomes are closer to OLS than Ridge, indicating that the results from the previous project were likely highly influenced by the added noise in the dataset.

4.1.5. Validation

To validate the neural network, we have also tested the training against Scikit-learn's built-in neural network, using Adam as an optimizer and the Sigmoid activation for the hidden layers. Though our network performs well with small errors, particularly when using Adam with a small learning rate, the Scikit-learn library produces slightly better results overall. Additionally, we have verified all activation and cost function derivatives using Autograd, confirming that all the computations are correctly differentiated and implemented in our neural network.

4.2. Classification

As mentioned earlier, the data we´re using for the classification problem, is the mnist dataset which is a multiclass problem. Using our neural network on this dataset, we have produced figures which show how changing learning rates, activation functions, etc., changes the results. In figure 8, one can see that the accuracy increases with the amount of epochs - This is expected as the learning rate is quite low (0.0001), so there is no sign of overfitting here. In figure 9, we see the training error for different epochs and activation functions. We see that the training error when using sigmoid as the activation function is much lower than when using ReLU or LReLU, especially for runs with smaller epochs. This is a contradiction to what we expected, as we mentioned in the theory section that we expect training to be faster when using ReLU compared to other activation functions which saturates. An explanation for this could be that ReLU and LReLU might receive negative values. As the weights are initialized with mean=0 and std=1, the intermediary might contain many negative values. This causes the the gradients of ReLU and LReLU to be zero (or close to 0 for LReLU), which prevents learning. In contrast to this, Sigmoid will compute non-zero gradients even if the intermediary is negative. This could be the cause of the difference in performance between Sigmoid and ReLU/LReLU.

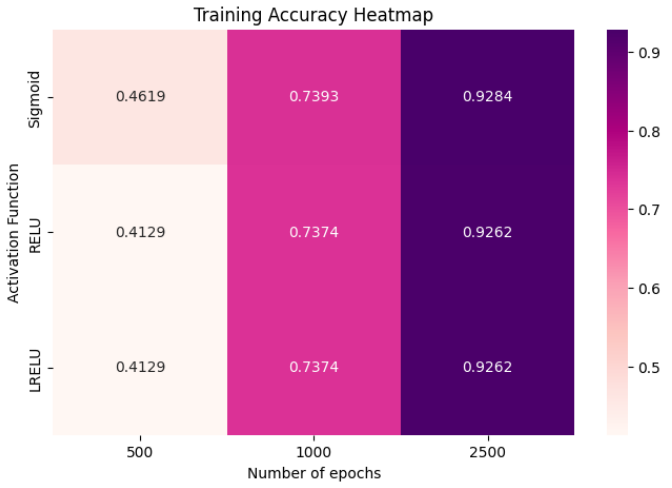

Figure 8: This figure shows the accuracy using different number of epochs (x-axis) and activation functions (y-axis). This run is with learning rate: eta = 0.0001, without batches, regularization parameteter: lam = 0.001, and with 2 hidden layers with 100 nodes.

Figure 9: This figure shows the error using different number of epochs (x-axis) and activation functions (y-axis). Same run as figure 8.

Furthermore, we have tested different learning rates and regularization parameters for the classification problem, as can be seen in figure 10. Since Sigmoid was the activation function for the hidden layers which worked the best in the previous run, we used Sigmoid for testing different regularization parameters and learning rates as well. We can see in figure 10 that the learning rate which performs best when using 2000 epochs, is 0.0065. We see that the error is significantly lower than when the learning rate is 0.1 and 0.0001. We also see that the accuracy is higher. We would expect a lower learning rate to perform better for this number of epochs. A small learning rate learns slower but is more stable and a larger learning rate might overshoot the optimal solution [10]. Therefore the smallest learning rate might cause learning to be too slow. The different values for the regularization parameter do not impact the smaller learning rates, but the results for the largest learning rate (0.1) vary quite a lot. This indicates that this learning rate makes the model less stable.

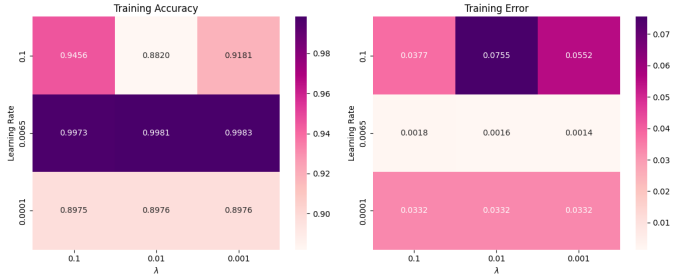

Figure 10: This figure shows the accuracy and training error for different regularization parameters (x-axis) and learning rates (y-axis), with the number of epochs being 2000 and Sigmoid as the hidden activation function.

The code using MLPClassifier supports the results from figure 10. In figure 11 we see the results using MLPClassifier. We can see that the accuracy is significantly larger for when the learning rate is 0.0065 than for the others. The error is also much lower.

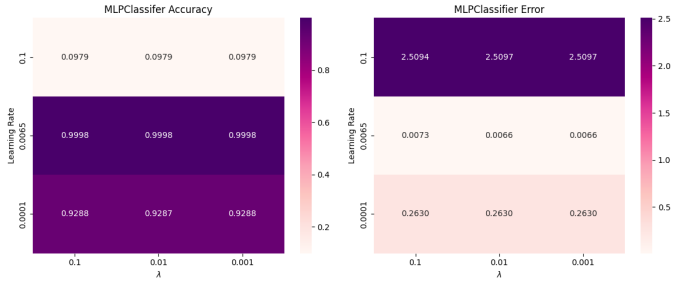

Figure 11: This figure shows the accuracy and training error for different regularization parameters (x-axis) and learning rates (y-axis), when using MLPClassifier from sklearn.

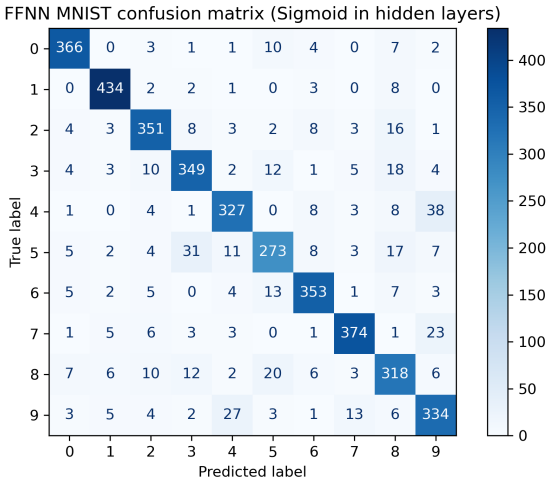

Figure 12: Confusion matrix showing the classification results when using Sigmoid as the activation function for hidden layers.

5. Conclusion

To conclude, for the regression part of the project, we find that a low learning rate, ideally $\mu = 0.0001$ produces the best results for both the Sigmoid activation and the Leaky RELU activation. We also find that Adam is the gradient optimizer that performs the overall best throughout

the report. Although RMSprop achieves nearly as good results, Adam remains numerically stable across the different experiments. The Constant optimizer performs well in certain cases, but often shows instability, making it less reliable for the regression tasks.

While Project 1 concluded the Ridge Regression provided the best fit to the Runge function, the results from the neural network in Project 2 contradict this finding. Here, the model performs best without explicit L1 and L2 regularization, except from a constant $\lambda = 0.01$ added for numerical stability.

Throughout all the experiments, we find that using 2000 epochs works well for both the one- and two-hidden-layer models, while maintaining a batch size of 4. The neural network performs well throughout the experiments with low training and test errors. Although there are a few signs of overfitting for the Sigmoid activation layer for a higher number of epochs, overall, the results indicate that the network generalizes well to new data, while maintaining low training errors.

The results from the classification problem indicated that Sigmoid was the most effective activation function for the hidden layers. and that the optimal learning rate was 0.0065 when using 2000 epochs. A possible reason why the Sigmoid function was the most effective activation function could be the way we have initialized the weights. For future work, we could initialize the weights differently to make ReLU and LReLU more effective. When using a larger learning rate, we found that the model might overshoot the optimal solution, meanwhile a smaller learning rate caused learning to be too slow when using 2000 epochs.

References

[1] Alex Adams. Comparing the moore–penrose pseudoinverse and gradient descent for solving linear regression problems: A performance analysis, 2025. Accessed: 2025-11-13 at https://arxiv.org/html/2505.23552v1.

[2] Shazan Ansar. "escaping the trap of local minima: Why optimization algorithms get stuck and how to break free". *Medium*, 2018. Accessed: 2025-11-13.

[3] Aditi Babu. Why accuracy isn't enough: A guide to ai/ml model evaluation, March 2025. Accessed: 2025-11-13.

[4] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences of the United States of America*, 116(32):15849–15854, 2019.

[5] Fernando Berzal. Dl101 neural network outputs and loss functions. arXiv, 2025. Available at `https://arxiv.org/abs/2511.05131`.

[6] P. Ong J. R. Boyd, J. Exponentially-convergent strategies for defeating the runge phenomenon for the approximation of non-periodic functions, part i: Single-interval schemes. *University of Michigan*, 2007.

[7] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*, pages 21–49. Springer, 2008.

[8] Akash Das, Piyush Kumar Kumawat, and Nitin Dutt Chaturvedi. A study to target energy consumption in wastewater treatment plant using machine learning algorithms. In Metin Türkay and Rafiqul Gani, editors, *31st European Symposium on Computer Aided Process Engineering (ESCAPE-31)*, volume 50 of *Computer Aided Chemical Engineering*, pages 1511–1516. Elsevier, Amsterdam, 2021.

[9] GeeksforGeeks. Overfitting and regularization in ml, 2025.

[10] GeeksforGeeks. Stochastic gradient descent classifier, 2025.

[11] GeeksforGeeks. Understanding gradient clipping, 2025.

[12] GeeksforGeeks. Train error vs test error, Unknown.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[14] Román Salmerón Gómez, Catalina García García, and Ainara Rodríguez Sánchez. Enlarging of the sample to address multicollinearity. *arXiv*, 2024. Accessed: 2025-11-13.

[15] 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.

[16] Morten Hjorth-Jensen. Constructing a neural network code with examples. `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week42.ipynb`, 2025. Applied Data Analysis and Machine Learning Lecture Notes, University of Oslo. (Accessed 2025-11-13.

[17] Morten Hjorth-Jensen. Project 2 on machine learning, deadline november 10 (midnight). Lecture notes for *Applied Data Analysis and Machine Learning* (FYS-STK3155/FYS4155), *University of Oslo, Norway*, October 2025. Available online: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/project2.html`.

[18] Morten Hjorth-Jensen. Ridge and lasso regression. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter2.html`, 2025. Applied Data Analysis and Machine Learning Lecture Notes, University of Oslo. (Accessed 2025-11-13.

[19] Grace Huckins. Neural networks help us understand how the brain recognizes numbers, July 2023. Accessed: 2025-11-14.

[20] Jacob Murel and Eda Kavlakoglu. What is regularization?, November 2025. `https://www.ibm.com/think/topics/regularization` (accessed 2025-11-13).

[21] Christopher Musco. Cs-gy 6923: Lecture 5 — linear classification, logistic regression, gradient descent. https://www.chrismusco.com/machinelearning2023_grad/lectures/lec5.pdf, 2023. Convexity of logistic loss discussed on slides 25–26, 65.

[22] Anthony Pierce. *Lecture 3: The Runge Phenomenon and Piecewise Polynomial Interpolation*. UBC mathematics department, 2017.

[23] M. Rennie, Jason D.˙ Regularized logistic regression is strictly convex. Technical report, Massachusetts Institute of Technology, January 2005. Available at https://people.csail.mit.edu/jrennie/writing/convexLR.pdf.

[24] Max Wienandts. Regularization in linear regression: A deep dive into ridge and lasso. https://medium.com/@maxwienandts/regularization-in-linear-regression-a-deep-dive-into-ridge-and-lasso-3d2853e5e2b0, May 4 2025. Accessed: 2025-11-13.

A. Appendix

A.1. Derivatives of the Softmax–Cross-Entropy Model

We begin with the per-sample cross-entropy loss

$$\ell = -\sum_{m=1}^{K} y_{i,m} \log \pi_m(\mathbf{x}_i), \qquad \pi_m(\mathbf{x}_i) = \frac{e^{z_{i,m}}}{\sum_{j=1}^{K} e^{z_{i,j}}}.$$

The full objective is

$$\mathcal{J}_{\mathrm{CE}} = \frac{1}{N} \sum_{i=1}^{N} \ell.$$

A.1.1. Gradient with respect to logits

To make the derivation explicit, recall that the per-sample loss is

$$\ell \;=\; -\sum_{m=1}^{K} y_{i,m} \log \pi_{i,m}, \qquad \pi_{i,m} = \frac{e^{z_{i,m}}}{\sum_{j=1}^{K} e^{z_{i,j}}}.$$

We wish to compute the derivative of ℓ with respect to a single logit $z_{i,k}$. Applying the chain rule gives

$$\frac{\partial \ell}{\partial z_{i,k}} \;=\; -\sum_{m=1}^{K} y_{i,m} \frac{\partial}{\partial z_{i,k}} \log \pi_{i,m}.$$

Next, differentiate the logarithm:

$$\frac{\partial}{\partial z_{i,k}} \log \pi_{i,m} \;=\; \frac{1}{\pi_{i,m}} \frac{\partial \pi_{i,m}}{\partial z_{i,k}}.$$

Thus,

$$\frac{\partial \ell}{\partial z_{i,k}} \;=\; -\sum_{m=1}^{K} y_{i,m} \frac{1}{\pi_{i,m}} \frac{\partial \pi_{i,m}}{\partial z_{i,k}}.$$

To evaluate $\partial \pi_{i,m} / \partial z_{i,k}$, we use the well–known Jacobian of the softmax function. Since

$$\pi_{i,m} = \frac{e^{z_{i,m}}}{\sum_{j=1}^{K} e^{z_{i,j}}},$$

its derivative with respect to $z_{i,k}$ is

$$\frac{\partial \pi_{i,m}}{\partial z_{i,k}} \;=\; \pi_{i,m} \left(\delta_{m,k} - \pi_{i,k}\right),$$

where $\delta_{m,k}$ is the Kronecker delta. Substituting this into the previous expression gives

$$\frac{\partial \ell}{\partial z_{i,k}} \;=\; -\sum_{m=1}^{K} y_{i,m} \frac{1}{\pi_{i,m}} \Big[\pi_{i,m} \left(\delta_{m,k} - \pi_{i,k}\right)\Big].$$

The $\pi_{i,m}$ terms cancel:

$$\frac{\partial \ell}{\partial z_{i,k}} \;=\; -\sum_{m=1}^{K} y_{i,m} \left(\delta_{m,k} - \pi_{i,k}\right).$$

We now distribute the summation:

$$\frac{\partial \ell}{\partial z_{i,k}} = -\sum_{m=1}^{K} y_{i,m} \delta_{m,k} \;+\; \pi_{i,k} \sum_{m=1}^{K} y_{i,m}.$$

Because the label vector is one–hot,

$$\sum_{m=1}^{K} y_{i,m} \delta_{m,k} = y_{i,k}, \qquad \sum_{m=1}^{K} y_{i,m} = 1.$$

Therefore,

$$\frac{\partial \ell}{\partial z_{i,k}} = -y_{i,k} \;+\; \pi_{i,k} = \pi_{i,k} - y_{i,k}.$$

Finally, averaging over the dataset yields the logit gradient

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial z_{i,k}} = \frac{1}{N} \big(\pi_k(\mathbf{x}_i) - y_{i,k}\big).$$

A.1.2. Chain rule to obtain parameter gradients

Using the chain rule,

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial \mathbf{w}_k} = \sum_{i=1}^{N} \frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial z_{i,k}} \frac{\partial z_{i,k}}{\partial \mathbf{w}_k}, \qquad \frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial b_k} = \sum_{i=1}^{N} \frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial z_{i,k}} \frac{\partial z_{i,k}}{\partial b_k}.$$

Substituting the expressions above yields

$$\frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial \mathbf{w}_k} = \frac{1}{N} \sum_{i=1}^{N} \big(\pi_k(\mathbf{x}_i) - y_{i,k}\big)\, \mathbf{x}_i, \qquad \frac{\partial \mathcal{J}_{\mathrm{CE}}}{\partial b_k} = \frac{1}{N} \sum_{i=1}^{N} \big(\pi_k(\mathbf{x}_i) - y_{i,k}\big).$$

A.2. Training algorithm

A.2.1. Optimization algorithms for the FFNN

Algorithm 1 Train-FFNN: row-major batches, L layers

Require: Data $(\mathbf{X}, \mathbf{Y})$, architecture $(n_0, \ldots, n_L)$, activations $g^{(1:L-1)}$, output type (regression or classification), per sample loss ℓ, regularizer Ω, $\lambda \geq 0$, step size η, batch size B, epochs T.

Initialize $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$, optimizer states and set RNG seed.
for $t = 1$ **to** T **do** ▷ epoch loop
  Shuffle rows of $(\mathbf{X}, \mathbf{Y})$ identically.
  for each mini-batch $(\mathbf{X}_B, \mathbf{Y}_B)$ of size B **do**
    Forward pass
    $\mathbf{A}^{(0)} \leftarrow \mathbf{X}_B$
    for $l = 1$ **to** $L - 1$ **do**
      $\mathbf{U}^{(l)} \leftarrow \mathbf{A}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{1}_B \mathbf{b}^{(l)\top}$
      $\mathbf{A}^{(l)} \leftarrow g^{(l)}(\mathbf{U}^{(l)})$
    end for
    $\mathbf{U}^{(L)} \leftarrow \mathbf{A}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{1}_B \mathbf{b}^{(L)\top}$
    $\mathbf{A}^{(L)} \leftarrow \begin{cases} \mathbf{U}^{(L)}, & \text{regression} \\ \mathrm{softmax}(\mathbf{U}^{(L)}), & \text{classification} \end{cases}$ ▷ row-wise max-shift for stability
    Compute batch loss $\mathcal{J}_B = \frac{1}{B} \sum \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i) + \lambda\, \Omega(\mathbf{W}^{(l)})$
    Backward pass
    $\mathbf{\Delta}^{(L)} \leftarrow \begin{cases} \frac{1}{B}\big(\mathbf{A}^{(L)} - \mathbf{Y}_B\big), & \text{Softmax-CE} \\ \frac{2}{B}\big(\mathbf{A}^{(L)} - \mathbf{Y}_B\big), & \text{MSE} \end{cases}$
    for $l = L - 1$ **down to** 1 **do**
      $\mathbf{\Delta}^{(l)} \leftarrow \big(\mathbf{\Delta}^{(l+1)}(\mathbf{W}^{(l+1)})^\top\big) \odot g^{(l)\prime}(\mathbf{U}^{(l)})$
    end for
    Gradients & regularization (biases unpenalized)
    for $l = 1$ **to** L **do**
      $\nabla \mathbf{W}^{(l)} \leftarrow (\mathbf{A}^{(l-1)})^\top \mathbf{\Delta}^{(l)} + \lambda\, \nabla \Omega(\mathbf{W}^{(l)})$
      $\nabla \mathbf{b}^{(l)} \leftarrow \mathbf{1}_B^\top \mathbf{\Delta}^{(l)}$
    end for
    Stability: weight-gradient
    for $l = 1$ **to** L **do**
      $n \leftarrow \|\nabla \mathbf{W}^{(l)}\|_F$
      if $n > 1.0$ **then**
        $\nabla \mathbf{W}^{(l)} \leftarrow \nabla \mathbf{W}^{(l)} / n$ ▷ clip each layer's weight-gradient to unit Frobenius norm
      end if ▷ $\nabla \mathbf{b}^{(l)}$ is not clipped
    end for
    Optimizer update (SGD/RMSprop/Adam)
    for $l = 1$ **to** L **do**
      $(\mathbf{W}^{(l)}, \mathbf{b}^{(l)}) \leftarrow \mathrm{OptimizerStep}\big((\mathbf{W}^{(l)}, \mathbf{b}^{(l)}), (\nabla \mathbf{W}^{(l)}, \nabla \mathbf{b}^{(l)}), \eta\big)$
    end for
  end for
end for
return trained $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$

Algorithm 2 FFNN Optimizer-Init

Require: Choice $\in \{\text{SGD}, \text{RMSprop}(\rho, \epsilon), \text{Adam}(\mathcal{B}_1, \mathcal{B}_2, \epsilon)\}$; layers $l = 1{:}L$

Ensure: For RMSprop: $\{\mathbf{S}_W^{(l)}, \mathbf{S}_b^{(l)}\}_{l=1}^{L}$ initialized to $\mathbf{0}$.

Ensure: For Adam: $\{\mathbf{M}_W^{(l)}, \mathbf{M}_b^{(l)}, \mathbf{V}_W^{(l)}, \mathbf{V}_b^{(l)}\}_{l=1}^{L}$ to $\mathbf{0}$, and $t \leftarrow 0$.

if SGD **then**
    No optimizer state.
else if RMSprop **then**
    for $l = 1$ **to** L **do**
        $\mathbf{S}_W^{(l)} \leftarrow \mathbf{0}; \quad \mathbf{S}_b^{(l)} \leftarrow \mathbf{0}$
    end for
else if Adam **then**
    for $l = 1$ **to** L **do**
        $\mathbf{M}_W^{(l)} \leftarrow \mathbf{0}; \quad \mathbf{V}_W^{(l)} \leftarrow \mathbf{0}$
        $\mathbf{M}_b^{(l)} \leftarrow \mathbf{0}; \quad \mathbf{V}_b^{(l)} \leftarrow \mathbf{0}$
    end for
    $t \leftarrow 0$
end if

Algorithm 3 FFNN Optimizer-step per mini-batch

Require: Gradients $\{\nabla\mathbf{W}^{(l)}, \nabla\mathbf{b}^{(l)}\}$; params $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}$; step size η

Require: Choice $\in \{\text{SGD}, \text{RMSprop}(\rho, \epsilon), \text{Adam}(\mathcal{B}_1, \mathcal{B}_2, \epsilon)\}$; state from FFNN Optimizer-Init 2

if SGD **then**
    for $l = 1$ **to** L **do**
        $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\, \nabla\mathbf{W}^{(l)}$
        $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta\, \nabla\mathbf{b}^{(l)}$
    end for
else if RMSprop **then**
    for $l = 1$ **to** L **do**
        $\mathbf{S}_W^{(l)} \leftarrow \rho\, \mathbf{S}_W^{(l)} + (1-\rho)\left(\nabla\mathbf{W}^{(l)} \odot \nabla\mathbf{W}^{(l)}\right)$
        $\mathbf{S}_b^{(l)} \leftarrow \rho\, \mathbf{S}_b^{(l)} + (1-\rho)\left(\nabla\mathbf{b}^{(l)} \odot \nabla\mathbf{b}^{(l)}\right)$
        $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\, \nabla\mathbf{W}^{(l)} \oslash \left(\sqrt{\mathbf{S}_W^{(l)}} + \epsilon\right)$
        $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta\, \nabla\mathbf{b}^{(l)} \oslash \left(\sqrt{\mathbf{S}_b^{(l)}} + \epsilon\right)$
    end for
else if Adam **then**
    $t \leftarrow t + 1$
    for $l = 1$ **to** L **do**
        $\mathbf{M}_W^{(l)} \leftarrow \mathcal{B}_1\, \mathbf{M}_W^{(l)} + (1-\mathcal{B}_1)\, \nabla\mathbf{W}^{(l)}$
        $\mathbf{V}_W^{(l)} \leftarrow \mathcal{B}_2\, \mathbf{V}_W^{(l)} + (1-\mathcal{B}_2)\left(\nabla\mathbf{W}^{(l)} \odot \nabla\mathbf{W}^{(l)}\right)$
        $\widehat{\mathbf{M}}_W^{(l)} \leftarrow \mathbf{M}_W^{(l)} / \left(1 - \mathcal{B}_1^t\right)$
        $\widehat{\mathbf{V}}_W^{(l)} \leftarrow \mathbf{V}_W^{(l)} / \left(1 - \mathcal{B}_2^t\right)$
        $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\, \widehat{\mathbf{M}}_W^{(l)} \oslash \left(\sqrt{\widehat{\mathbf{V}}_W^{(l)}} + \epsilon\right)$
        $\mathbf{M}_b^{(l)} \leftarrow \mathcal{B}_1\, \mathbf{M}_b^{(l)} + (1-\mathcal{B}_1)\, \nabla\mathbf{b}^{(l)}$
        $\mathbf{V}_b^{(l)} \leftarrow \mathcal{B}_2\, \mathbf{V}_b^{(l)} + (1-\mathcal{B}_2)\left(\nabla\mathbf{b}^{(l)} \odot \nabla\mathbf{b}^{(l)}\right)$
        $\widehat{\mathbf{M}}_b^{(l)} \leftarrow \mathbf{M}_b^{(l)} / \left(1 - \mathcal{B}_1^t\right)$
        $\widehat{\mathbf{V}}_b^{(l)} \leftarrow \mathbf{V}_b^{(l)} / \left(1 - \mathcal{B}_2^t\right)$
        $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta\, \widehat{\mathbf{M}}_b^{(l)} \oslash \left(\sqrt{\widehat{\mathbf{V}}_b^{(l)}} + \epsilon\right)$
    end for
end if

A.2.2. Optimization algorithms for linear baselines

Algorithm 4 Linear-Optimizer-Init

Require: Choice $\in \{\text{SGD}, \text{RMSprop}(\rho, \epsilon), \text{Adam}(\mathcal{B}_1, \mathcal{B}_2, \epsilon)\}$
Ensure: For RMSprop: $\mathbf{S}_\beta \in \mathbb{R}^p$ initialized to $\mathbf{0}$
Ensure: For Adam: $\mathbf{M}_\beta, \mathbf{V}_\beta \in \mathbb{R}^p$ initialized to $\mathbf{0}$, and $t \leftarrow 0$
if SGD **then**
    No optimizer state
else if RMSprop **then**
    $\mathbf{S}_\beta \leftarrow \mathbf{0}$
else if Adam **then**
    $\mathbf{M}_\beta \leftarrow \mathbf{0}; \quad \mathbf{V}_\beta \leftarrow \mathbf{0}; \quad t \leftarrow 0$
end if

Algorithm 5 Linear-Optimizer-step (per mini-batch)

Require: Gradient $\mathbf{g} = \nabla_{\boldsymbol{\beta}} \mathcal{J}_B$; current $\boldsymbol{\beta}$; step size η
Require: Choice $\in \{\text{SGD}, \text{RMSprop}(\rho, \epsilon), \text{Adam}(\mathcal{B}_1, \mathcal{B}_2, \epsilon)\}$ and state from Alg. 4
if SGD **then**
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta\, \mathbf{g}$
else if RMSprop **then**
    $\mathbf{S}_\beta \leftarrow \rho\, \mathbf{S}_\beta + (1-\rho)\, (\mathbf{g} \odot \mathbf{g})$
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta\, \mathbf{g} \oslash \left(\sqrt{\mathbf{S}_\beta} + \epsilon\right)$
else if Adam **then**
    $t \leftarrow t + 1$
    $\mathbf{M}_\beta \leftarrow \mathcal{B}_1\, \mathbf{M}_\beta + (1-\mathcal{B}_1)\, \mathbf{g}$
    $\mathbf{V}_\beta \leftarrow \mathcal{B}_2\, \mathbf{V}_\beta + (1-\mathcal{B}_2)\, (\mathbf{g} \odot \mathbf{g})$
    $\widehat{\mathbf{M}}_\beta \leftarrow \mathbf{M}_\beta / (1-\mathcal{B}_1^t); \quad \widehat{\mathbf{V}}_\beta \leftarrow \mathbf{V}_\beta / (1-\mathcal{B}_2^t)$
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \eta\, \widehat{\mathbf{M}}_\beta \oslash \left(\sqrt{\widehat{\mathbf{V}}_\beta} + \epsilon\right)$
end if
return $\boldsymbol{\beta}$

Algorithm 6 Train Linear Model: OLS and Ridge (mini-batch, SGD/RMSprop/Adam)

Require: Data $(\mathbf{X}, \mathbf{y})$; batch size B; epochs T; step size η; choice $\in \{\text{OLS}, \text{Ridge}\}$; $\lambda \geq 0$
Initialize $\boldsymbol{\beta} \in \mathbb{R}^p$
Linear-Optimizer-Init (Alg. 4)
for $t = 1$ **to** T **do** ▷ epochs
    Shuffle $(\mathbf{X}, \mathbf{y})$ identically; form mini-batches $(\mathbf{X}_B, \mathbf{y}_B)$
    for each $(\mathbf{X}_B, \mathbf{y}_B)$ **do**
        $\mathbf{r}_B \leftarrow \mathbf{X}_B \boldsymbol{\beta} - \mathbf{y}_B$
        $\mathbf{g} \leftarrow \frac{2}{B}\, \mathbf{X}_B^\top \mathbf{r}_B$ ▷ OLS gradient
        if Ridge **then**
            $\mathbf{g} \leftarrow \mathbf{g} + 2\lambda\, \mathbf{D}\, \boldsymbol{\beta}$ ▷ $\mathbf{D} = \text{diag}(0, 1, \ldots, 1)$, intercept is left unpenalized
        end if
        $\boldsymbol{\beta} \leftarrow$ Linear-Optimizer-step (Alg. 5)
    end for
end for
return $\boldsymbol{\beta}$

Algorithm 7 Train Linear Model: Lasso (mini-batch SGD/RMSprop/Adam)

Require: Data $(\mathbf{X}, \mathbf{y})$; batch size B; epochs T; step size η; $\lambda > 0$.

Initialize $\boldsymbol{\beta} \in \mathbb{R}^p$; LINEAR-OPTIMIZER-INIT (Alg. 4)
for $t = 1$ **to** T **do**
  Shuffle $(\mathbf{X}, \mathbf{y})$; form mini-batches $(\mathbf{X}_B, \mathbf{y}_B)$
  for each $(\mathbf{X}_B, \mathbf{y}_B)$ **do**
    $\mathbf{r}_B \leftarrow \mathbf{X}_B\boldsymbol{\beta} - \mathbf{y}_B$
    $\mathbf{g} \leftarrow \frac{2}{B}\,\mathbf{X}_B^\top \mathbf{r}_B$ ▷ OLS part
    $\mathbf{s} \leftarrow \operatorname{sign}(\boldsymbol{\beta})$ (element-wise)
    $\mathbf{s} \leftarrow \operatorname{sign}(\boldsymbol{\beta})$ ▷ element-wise
    $\mathbf{g} \leftarrow \mathbf{g} + \lambda\,\mathbf{D}\,\mathbf{s}$ ▷ L1 subgradient, $\mathbf{D} = \operatorname{diag}(0, 1, \ldots, 1)$ leaves intercept unpenalized
    $\boldsymbol{\beta} \leftarrow$ LINEAR-OPTIMIZER-STEP (Alg. 5)
  end for
end for
return $\boldsymbol{\beta}$