An Introduction to Neural Networks for Regression and Classification
FYS-STK4155 - Project 2

Amund Solum Alsvik, Jørgen Vestly, & Kristoffer Stalker
University of Oslo
(Dated: November 10, 2025)

Deep learning has revolutionized virtually all industries during the last decade, and keeps showing novelties and remarkable results, yet the specific advantages of neural networks over traditional statistical models for fundamental tasks such as regression and classification remain an active topic of exploration. Understanding how network depth, non-linearity, and optimization influence performance is essential for applying neural networks effectively to real-world problems. We investigate the use of *feed forward neural networks* for regression and compare to conventional linear regression (see Alsvik & Vestly, [1]). Thereafter, we look at the use of classification with the same network, predicting digits between 0 and 9 based on the MNIST dataset for handwritten digits [2]. For regression, we explore the effect of non-linearity in the neural network by fitting to the 1D Runge's function. We find the neural network fits a better and smoother prediction compared to traditional interpolation, showing that the network learns structures that goes beyond linearity, and picks up complexity much better. Moreover, we considered overfitting as a function of the depth of the network for both regression and classification, and found that a more complex model doesn't always yield the best results. We also study the influence of activation functions and optimizers on model performance and computational efficiency. Our findings show that leaky ReLU gave the best accuracy for classification, and that the hyperbolic tangent was by far the most efficient, despite achieving slightly lower accuracy.

1. INTRODUCTION

Machine learning has become a cornerstone of modern science, and is the framework behind consumer based large language models, that have quickly entered everyday life for the population as a whole. Under the hood of machine learning there is a multitude of different methods and models, with perhaps the most prolific of these being neural networks. Neural networks are often considered *black boxes*, meaning that the user can see the inputs and outputs, but does not understand the actual mechanism that produces those outputs, as explained in Kosinski [3]. In this report, we aim to give a mathematical explanation of how neural networks are trained and produce predictions. Moreover, we also want to demonstrate the benefits and the wide use case neural networks, over simpler machine learning methods, such as linear regression. We demonstrate this through two different types of machine learning problems, regression and classification.

For the regression part, we are approximating *Runge's function*, which is given by

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

where we evaluate the function on the interval $[-1, 1]$. This is a function famous for being difficult to interpolate and approximate using linear regression, due to approximations often resulting in an oscillating behavior near the edges of the interval $[-1, 1]$, which is often called *Runge's phenomenon*. For a technical explanation of this, see Cosgun and Nies [4]. Moreover, for a demonstration of performance of various linear regression methods on Runge's function, see Alsvik and Vestly [1].

We also apply neural networks to classification problems, meaning problems where we wish to predict binary outcomes based on input data. This is a problem that linear regression models typically cannot solve. In this project, we use the MNIST dataset for handwritten digits [2], to demonstrate the flexibility of neural networks. Namely, the fact that neural networks can be applied to a vast class of problems, like regression and classification. We also explore how the performance of the classification models are affected by changes in hyperparameters.

In section 2 we introduce the theoretical framework for neural networks, and discuss their application to regression and classification. We consider advantages and shortcomings of these methods, and ways of coping with the latter. We also derive the expressions used for implementing the code. This part serves as an imperative foundation for conducting the research. In section 3 we show how we implemented the algorithms for the neural network, and discuss testing and verifications. We show pseudocode for our network implementation, and discuss the practical aspect of the research. In section 4, we present our results and discuss whether these were in accordance to our hypothesis, for regression and classification. Furthermore, in section 5, we summarize our main findings along with the discussion of how the results came to be. We also discuss the implications of our findings and perspectives for future work. Also, we consider whether our main results are in accordance to the theory.

2. THEORY AND METHODS

In this section we give an introduction to neural networks. We explain how they can be used for and adapted to problems in machine learning and how to evaluate their performance. Throughout this project, we use linear regression as a benchmark for testing our neural network models. We assume that the reader is familiar with linear regression with and without regularization. Moreover, we assume that the reader is familiar with various gradient descent methods with and without adaptive learning rate. For a thorough introduction to these topic see the Theory and Methods section in Alsvik and Vestly [1].

2.1. Introduction to Neural Networks

Neural networks were originally inspired by neuroscience, and were an attempt to mimic the way neurons in the brain transmits signals. One of the earliest mathematical models is the *perceptron*, which Rosenblatt proposed in his article [5] from 1958. In this model, a *neuron* receives several inputs, forms a weighted sum from these and then passes the result through an *activation function*, which is typically a nonlinear transformation. Modern neural networks uses the same fundamental idea, but uses a series of layers with multiple neurons. Given data points with p input features, the neural network has an *input layer* with p neurons, one for each feature. Next comes one or more *hidden layers*, that each contain a chosen amount of neurons. The final layer is called the *output layer*, and its number of neurons depends on what kind of problem the model is solving. For regression problems, the output layer usually consists of a single neuron. A graphical interpretation of the neural network architecture can be seen in figure 1.

We can consider a classic linear regression model as a neural network with no hidden layers and a linear activation function in the output layer. By having several layers with nonlinear activation functions, neural networks allows for more complex approximations of nonlinear properties of the data. We will get back to the specific reasons for this later. The choice of number of hidden layers and amount of neurons in them is often subject to tweaking and testing. For more challenging problems you may need a large number of hidden layers to capture complex structures in the data, but as explained in Isaac et al. [6], too many layers can also lead to overfitting. We will discuss this point in more detail in subsequent sections.

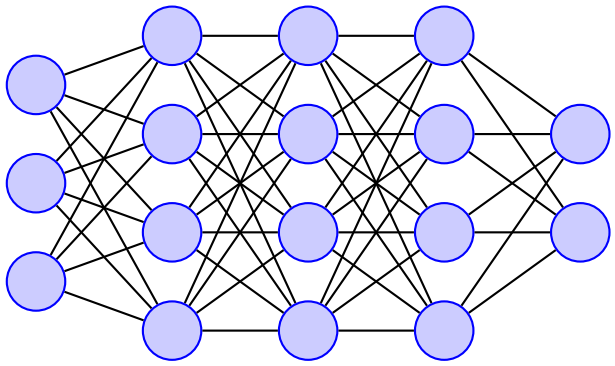

Figure 1. A neural network with three hidden layers, each with 4 neurons, and an output layer with 2 neurons. It takes in 3 input features. The diagram was generated based on TikZ-code available in the TikZ guide [7].

2.2. Mathematical Formulation and Feedforward Algorithm

A neural networks relies on two algorithms. One is for producing predictions, and the other is for training the network. In this section we give a detailed explanation of the first algorithm, which is called the *feedforward* algorithm. Each neuron has a set of weights corresponding to each input feature, together with a bias term. Thus, for the j-th layer of neurons, we can define a *weight matrix* W^{j-1}, where the rows consist of the weights for each neuron in the layer. Moreover, we can define a *bias vector* b^{j-1}, consisting of the bias terms for each neuron in the layer. Let a^{j-1} denote the inputs from the previous layer, and let the activation function for the neurons in layer j be given by σ^j. The feedforward process from layer $j-1$ to j is then given by

$$a^j = \sigma^j\left(W^{j-1}a^{j-1} + b^{j-1}\right),$$

where we denote

$$z^j = W^{j-1}a^{j-1} + b^{j-1},$$

so that $a^j = \sigma^j(z^j)$. Hence for a neural network with L layers, the prediction of the model is given by $f(x) = a^L$, where f denotes the function given by the neural network, and $x \in \mathbb{R}^n$ is the input data point. The computation from the input layer to the first hidden layer is illustrated in figure 2. Here a^0 denotes the input data point with n features.

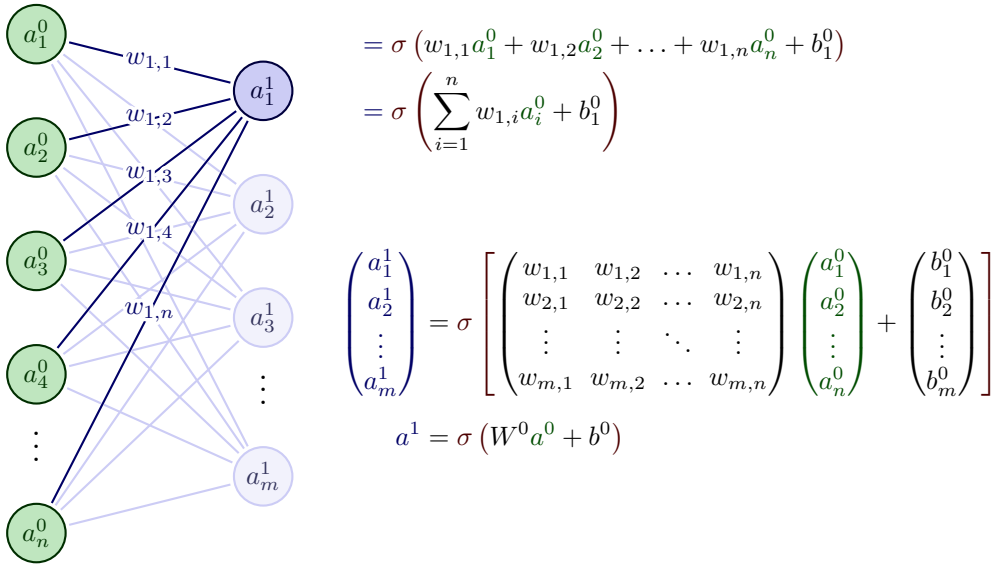

Figure 2. Feedforward from the input layer to the first hidden layer. The diagram is generated based on TikZ-code available in the TikZ guide [7].

2.3. Training Neural Networks with Backpropagation

So far we explained how a neural network produces predictions for a given set of weights and bias terms. However, we have not yet touched upon how these parameters are chosen. This is where our second algorithm comes in, namely the *backpropagation* algorithm. Assume that we have a neural network in the same setting as in the previous section. First, we must introduce the concept of a *cost function* in neural networks. Much like in linear regression, the cost function of a model determines what our model is trying to accomplish, and measures some kind of error between the models output a^L and the true data y. We will later discuss different cost functions and their purposes, but for now, we assume that our neural network has a given cost function $C(a^L, y)$, which depends implicitly on the sets of weights and bias terms. Backpropagation is an algorithm to find weights and bias terms that minimize the cost function. The algorithm is somewhat laborious, so we explain it stepwise.

Step 1. We initialize a set of weights and bias terms randomly. The weights are usually small values, normally distributed around zero, while the bias terms are usually set to either 0, or a small positive value, such as 0.01. It is also common to specify a stopping criterion, which terminates the algorithm when the cost is sufficiently small.

Step 2. Using the previously chosen weights, we perform a feedforward, which gives us an output. Using this output, we compute the cost.

Step 3. We compute the gradient of our cost function with respect to the weights and bias terms in each layer. The expressions for the derivatives is taken from Hjort-Jensen [8]. We start with the final layer. Using the chain rule from multi-variable calculus one finds that

$$\frac{\partial C}{\partial W^{L-1}} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial W^{L-1}}. \tag{1}$$

Moving on to the second-to-last layer, again by the chain rule, we have that

$$\frac{\partial C}{\partial W^{L-2}} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial W^{L-2}}.$$

The gradient for the final layer with respect to the bias terms, is given by

$$\frac{\partial C}{\partial b^{L-1}} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L},$$

with an analogous computation for the second-to-last layer. Notice that some terms in the gradient from the last layer, reappear in the gradient of the second-to-last layer. This is one of the strengths of backpropagation. By going backwards through the layers and reusing terms we have already computed, we can more efficiently compute the gradients.

Step 4. Now that we have computed all the gradients, we simultaneously perform one iteration of a chosen gradient descent method for each of the layers. Any gradient descent method can be used, but for simplicity, we use standard gradient descent notation here. Thus for each layer j, we get the updated weights and bias terms for a given learning rate η by

$$W^j \leftarrow W^j - \eta\frac{\partial C}{\partial W^j},$$

and

$$b^j \leftarrow b^j - \eta \frac{\partial C}{\partial b^j}.$$

Step 5. We now feedforward using our new weights, and compute the cost of our new output. If it satisfies the stopping criterion, our algorithm stops, meaning we have found a "good" set of weights and bias terms. If not, return to step 3, and repeat the following steps. Passing the whole dataset through this process once is referred to as an *epoch*.

As can be seen in the steps, using backpropagation involves multiplying the gradients as we move backwards through the network. The gradients may become near-zero, which is called *vanishing gradients*, or grow very large, which is called *exploding gradients*. These phenomenons can both lead to numerical problems. Vanishing gradients slows down training, due to the step size becoming very small for each iteration. Exploding gradients can cause numerical issues due to diverging weights, as explained in the lecture notes by Hjort-Jensen [8]. This can especially be a problem in neural networks with many layers, as it increases the amount of multiplications needed. The choice of activation function is an important tool to combat these problems. We will return to this topic in our section about the different activation functions.

When training neural networks, it is common to partition the dataset into several *mini-batches*, in an analogous way to stochastic gradient descent. This drastically lowers the computational expense per weight update, at the cost of requiring more iterations to complete one epoch.

2.4. The Universal Approximation Theorem

We have now explained how the neural network produces algorithms and how it is trained. However, it may not be immediately clear why one would use a neural network over simpler methods like linear regression. Neural networks are more computationally demanding, particularly if it contains many hidden layers. Moreover, there are many choices to make in regards to number of layers, the number of nodes and which activation functions to use. While these are valid concerns, neural networks have a big advantage over linear regression. This advantage is illustrated by the *universal approximation theorem*, which states the following.

Theorem (Adapted from Hjort-Jensen [9] and Cybenko [10])**.** *Let σ be any continuous sigmoidal function such that*

$$\sigma(z) \to \begin{cases} 1, & as\ z \to \infty, \\ 0, & as\ z \to -\infty. \end{cases}$$

Let $K \subset \mathbb{R}^d$ be compact, and let $F : K \to \mathbb{R}$ be a continuous and deterministic function. There is a one-hidden-layer neural network $f(x;\Theta)$, with $\Theta = (W, b)$, where W is a weight matrix and b is a bias vector, such that for all $\epsilon > 0$,

$$\sup_{x \in K} |F(x) - f(x;\Theta)| < \epsilon.$$

Although this theorem might seem mathematically esoteric, in practice it implies that any continuous function on a compact domain can be approximated arbitrarily well with a one-hidden-layer neural network with a suitable activation function. This theorem was originally proven for sigmoidal activation functions by Cybenko [10], but was later extended to a wider class of functions by Hornik [11]. We will revisit sigmoidal functions in subsequent sections, but for now, note that all activation functions we use in this project are sufficient for the theorem to apply. To summarize, the universal approximation theorem illustrates that the theoretical ceiling of the types of approximations that a neural network can perform is higher than for linear regression.

2.5. Neural Networks for Regression

Neural networks can be used for regression, where the goal is to approximate a function based on a set of inputs and the corresponding outputs. When constructing a neural network for regression, we usually have the number of variables of the function we are trying to approximate, as number of input features. Meaning for a one-dimensional function, we typically have one input feature. Moreover, while the hidden layers will have nonlinear activation functions, the output layer has the identity function. We will discuss choices of activation functions further in our dedicated section about activation functions.

Much like linear regression, the cost functions we use are variants of the *mean squared error* (MSE), given by

$$\text{MSE}(a^L, y) = \frac{1}{N} \sum_{i=1}^{N} (a_i^L - y_i)^2,$$

where $a^L = (a_1^L, \ldots, a_N^L)$ is the prediction of the network and y is the true data. As seen in equation 1, we need an expression for the gradients of the cost functions we use, with respect to the prediction a^L. The derivative of the L_1-norm used in this section was taken from Hjort-Jensen [12]. For the MSE, this gradient is given by

$$\frac{\partial \text{MSE}}{\partial a^L} = \frac{2}{N}(a^L - y).$$

Moreover, you can also include L_1 and L_2 regularization, by defining cost functions

$$C_1 = \text{MSE} + \lambda \|W\|_1,$$

and

$$C_2 = \text{MSE} + \lambda \|W\|_2^2.$$

For L_1 regularization, the gradient with respect to the weight matrix is given by

$$\frac{\partial C_1}{\partial W} = \frac{\partial \text{MSE}}{\partial W} + \frac{\partial \lambda \|W\|_1}{\partial W} = \frac{\partial \text{MSE}}{\partial W} + \lambda \text{sign}(W).$$

The computation is analogous for L_2 regularization, where the gradient is given by

$$\frac{\partial C_2}{\partial W} = \frac{\partial \text{MSE}}{\partial W} + 2\lambda W.$$

Thus we see that adding L_1 and L_2 regularization amounts to adding a penalizer to the MSE gradient. Which encourages smaller weights. As for the gradients with respect to the bias terms, the regularization terms does not depend on the bias terms, hence does not affect those gradients.

Similar to linear regression, adding a regularization term in the cost function is related to the bias variance tradeoff. An introduction to this concept can be found in Alsvik and Vestly [1]. To summarize, introducing some bias through the regularization terms can be useful for combating overfitting, which is particularly relevant for neural networks with many layers and neurons While L_2 regularization tends to shrink weights, L_1 will often favor setting weights to zero, creating a sparser network. This is due to the different cost landscapes of the L_1 and L_2-norms, as explained in chapter 11 of Géron [13]. We will show concrete examples of regularization mitigating overfitting in our Results and Discussion section.

2.6. Logistic regression and Classification Problems

One often wants a model to be able to make predictions on a certain outcome. For instance, suppose we want to predict whether a student passes or fails a test based on the number of hours studied. In this case, we want to interpret the output of our model as a probability. Linear regression is not suitable for this, as the predictions can be any real numbers, not just values between 0 and 1. This motives *logistic regression*. We begin by introducing what we call the *sigmoid function* $\sigma : \mathbb{R} \to (0, 1)$, which is given by

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

This is one of the activation functions used in this project, and it will, among other such functions, be revisited in the section dedicated to activation functions. Logistic regression uses this activation function to squeeze its predictions to probabilities. Assume that we have a dataset (y_i, x_i), where each y_i is a binary variable dependent on x_i. For a set of weights $w \in \mathbb{R}^p$ and a bias term b, we perform a computation analogous to the feedforward algorithm, namely we set $z = w^T x + b$. From here, we plug z into the sigmoid function, and we define $\hat{y}_i := \sigma(z)$ as the prediction of the model, which we can interpret as the probability $P(y = 1|x)$. For finding the set of weights and bias term, we minimize the *binary cross-entropy* cost function, which is given by

$$BCE(\theta) = -\sum_{i=1}^{N} \Big(y_i z_i - \log(1 + e^{z_i}) \Big),$$

although for interpretability, it can be convenient to rewrite this function as

$$BCE(\theta) = -\sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)].$$

This cost function penalizes large deviations between y_i and $\hat{y}_i$, as if $y_i = 1$, the function reduces to $-\sum_i^n log(\hat{y}_i)$, meaning that the cost becomes larger the further $\hat{y}_i$ is from 1. Similarly, when $y_i = 0$ the cross entropy increases the further $\hat{y}_i$ is from 0. Logistic regression is a tool for *classification problems*, where we want to find the probability that an input belongs to a certain class, which in this case is binary. Logistic regression can be viewed as the most basic example of a neural network for classification. It consists of a single neuron that takes in input features, and applies the sigmoid activation function to produce a probability.

We can generalize this concept to neural networks with many nodes and hidden layers, and also problems with multiple classes, where our true data is written as vectors $y_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$, such that the single nonzero entry denotes which class the data point belongs to. In classification problems, the output layer always has the same number of neurons as number of classes. For these types of problems, we use variants of the cross entropy as the cost function, as well as two different types of output functions. For problems with binary classes, we use the binary cross entropy cost function and the sigmoid function in our output layer. For problems with C, classes, we use the *categorical cross entropy* cost function given by

$$CCE = -\frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{C} y_{n,i} \log a_{n,i}^L,$$

which is a generalization of the binary cross entropy, where $P = 2$. Moreover, the output function we use is the *softmax function* $s : \mathbb{R}^C \to \mathbb{R}^C$, which for $x = (x_1, \ldots, x_C)$ is given by

$$s(x) = [s_i(x_i)]_{i=1}^C := \left(\frac{e^{x_i}}{\sum_{j=1}^C e^{x_j}} \right)_{i=1}^C.$$

Analogous to categorical cross entropy, the softmax function generalizes the sigmoid function to several dimensions. Note that while the neural networks for classification uses sigmoid or softmax as output functions, they can have other activation functions in the hidden layers. For the sake of backpropagation, we need the derivatives of the cross entropy cost functions, as well as the derivatives for the sigmoid and softmax function. These are taken from Mehta [14] and Kurbiel [15]. The derivative for the sigmoid and softmax function is respectively given by

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2},$$

and

$$\nabla\text{softmax} = \begin{pmatrix} s_1(1-s_1) & -s_1s_2 & \cdots & -s_1s_n \\ -s_2s_1 & s_2(1-s_2) & \cdots & -s_2s_n \\ \vdots & \vdots & \ddots & \vdots \\ -s_ns_1 & -s_ns_2 & \cdots & s_n(1-s_n) \end{pmatrix}.$$

In multiclass problems, the combination of the softmax function and the categorical cross entropy cost function leads to a simplification in the backpropagation step. The derivative of the cost with respect to the pre-activation in the final layer is given by

$$\frac{\partial CCE}{\partial z^L} = a^L - y.$$

Likewise for the binary cross entropy function, we have that

$$\frac{\partial BCE}{\partial z^L} = a^L - y.$$

In other words, for both binary and multiclass classification, the gradient of the cost with respect to the pre-activation output is simply the difference between the predicted probabilities and the true labels. We can add L_1 and L_2 regularization to the cross entropy cost function as well, and in that case, analogously to the mean squared error, the gradient becomes

$$\frac{\partial BCE}{\partial W} + \lambda\text{sign}(W)$$

and

$$\frac{\partial BCE}{\partial W} + 2\lambda W$$

for L_1 and L_2 respectively.

One of the more common use cases of neural networks for classification is image recognition. By representing an image as a matrix of pixel values and using that as training data, a neural network can reliably recognize and classify images. An implementation of this on handwritten digits can be seen in the Results and Discussion section.

2.7. ReLU and Leaky ReLU

We have repeatedly mentioned the importance of nonlinear activation function in neural networks. For classification we discussed the sigmoid and softmax function, as well as their derivatives. As mentioned earlier, a common problem with training neural networks is the problem of exploding and vanishing gradients, and researches found that this was particularly a problem for the sigmoid function. An activation function often used to combat this, is the *Rectified Linear Unit* function $\text{ReLU} : \mathbb{R} \to \mathbb{R}^+$, given by

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

The formal proof for why this function prevents exploding and vanishing gradients is beyond the scope of this report, but the fact that the sigmoid function saturates near zero and near one plays a part, as explained in chapter 11 of Géron [13]. The derivatives in this section was taken from Mehta [16]. The derivative of the ReLU function is given by

$$\text{ReLU}'(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x \geq 0. \end{cases}$$

While the ReLU function prevents vanishing and exploding gradients, it has its own problem, namely *dying neurons*, meaning they only output zero. A neuron usually dies when its weights are updated to a point where the weighted sum of the neurons output becomes negative for the whole training data. Since the derivative of the ReLU function is zero when inputs are negative, further iterations of gradient descent will not update the weights. To mitigate this, it is common to use a variant of the ReLU function, called the leaky ReLU function, which for a small $\alpha > 0$ is given by

$$\text{LeakyReLU}(x) = \max(\alpha x, x),$$

with derivative

$$\text{LeakyReLU}'(x) = \begin{cases} 1, & \text{if } x > 0, \\ \alpha, & \text{if } x < 0. \end{cases}$$

Thus instead of returning zero for negative inputs, it will instead produce a small negative value, allowing gradient descent to further update the weights. To read more about dead neurons, see chapter 11 of Géron [13]. For regression problems, using ReLU or leaky ReLU is common, and leaky ReLU is also often used for classification problems. In our results and discussion section, we compare the performance of the different activation functions regression and classification.

2.8. Performance metrics

There are several ways to evaluate the performance of a neural network, but in this project we mainly use two.

For evaluating performance in regression problems, we use the standard mean squared error. For classification problems, we measure accuracy by looking at the percentage of correct guesses by our model. In our results and discussion section we also implement a *confusion matrix*, which gives an intuitive visualization of the models accuracy in classification problems.

3. IMPLEMENTATION

Machine learning, as a practical discipline, is concerned with gaining insights and predictions on big datasets. Implementation of executable code and testing the code to obtain these insights is therefore imperative. In this section, we harness the theory from the previous section to create building blocks for code and testing. Another important aspect of any computer program is to consider the validity of the results. During this research project, we consistently test our code by comparing to inbuilt python libraries such as sci-kit learn and PyTorch. Later in this section, we'll look at specific examples of this usage.

Machine learning is also experimental in nature, and a fairly large block of time is devoted to finding the best *hyperparameters* of a given model to produce the best results, i.e. the intrinsic settings that can be tweaked as inputs to the model when testing. A full account of the general procedure for hyperparameters and the experimental protocol is given in Section 3.3 in Alsvik & Vestly (see [1]). Also, in the Jupyter testfiles on GitHub, denoted with the prefix ***test_***, there is a complete usage of this protocol with comments.

3.1. Implementing the neural network

Algorithm 1 TrainNetwork($X, Y, \eta,$ `epochs`)

Require: learning rate η, epochs, model with weights $\{W^{[l]}, b^{[l]}\}_{l=1}^{L}$
1: **for** $e = 1$ to `epochs` **do**
2: $(\nabla W, \nabla b) \leftarrow Backpropagation(X, Y)$
3: $UpdateWeights(\nabla W, \nabla b, \eta)$
4: **end for**

Algorithm 1 is called after creating the Neural Network object. This method does the training of the network by finding the gradients via *Backpropagation* and then update the weights based on these gradients using Gradient Descent. After calling this method, it updates the weights in the class to contain the ones minimizing a given cost function, C. The code is generic and shown for simplicity purposes. See the methods, *train_SGD* and *train_SGD_v2* for full implementation in our GitHub repository, Alsvik et. al, ([17]).

Algorithm 2 Backpropagation($\mathcal{B}$, W, b)

Require: batch $\mathcal{B} = \{(x_i, y_i)\}$, params $\{W_l, b_l\}_{l=1}^{L}$
Ensure: $\{\nabla_{W_l} C, \nabla_{b_l} C\}_{l=1}^{L}$
1: Feed forward:
2: **for** each $(x, y) \in \mathcal{B}$ **do**
3: $a_0 \leftarrow x$
4: **for** $l = 1 \dots L$ **do**
5: $z_l \leftarrow W_l a_{l-1} + b_l$; $a_l \leftarrow f_l(z_l)$
6: **end for**
7: $C \leftarrow \text{Cost}(a_L, y)$
8: $\delta_L \leftarrow \nabla_{a_L} C \; f'_L(z_L)$
9: Backprop:
10: **for** $l = L \dots 1$ **do**
11: $\nabla_{W_l} C \leftarrow a_{l-1}^{\top} \delta_l$
12: $\nabla_{b_l} C \leftarrow \sum_i \delta_l^{(i)}$
13: $\delta_{l-1} \leftarrow (\delta_l W_l^{\top}) \, f'_{l-1}(z_{l-1}) \;\; (l > 1)$
14: **end for**
15: **end for**
16: **return** $\{\nabla_{W_l} C, \nabla_{b_l} C\}_{l=1}^{L}$

Algorithm 2 finds all necessary gradients, by starting at the output layer and *propagating* backwards through the hidden layers. The method does a *feed forward pass* first to get the predictions. Then, it computes the cost function for the output predictions and the *delta term* (see 2.4 in section 2). It proceeds to do backpropagation, by iterating over each layer and finding the gradients for the weight-biases pairs, along with the delta terms. In the loop, we have provided the expressions without regularization. But check 2.5 in Section 2. Backpropagation returns the updated gradients for weights and biases for all layers.

Algorithm 3 UpdateWeights$\big(\{\nabla W^{[l]}\}, \{\nabla b^{[l]}\}, \eta\big)$

for $l = 1$ to L **do**
$W^{[l]} \leftarrow W^{[l]} - \eta \nabla W^{[l]}$
$b^{[l]} \leftarrow b^{[l]} - \eta \nabla b^{[l]}$
end for

The method *UpdateWeights* uses gradient optimization for the gradients found in backpropagation to find the weights and biases minimizing the given cost function. The Algorithm is describing an ordinary Gradient Descent, shown to illustrate the basic logic. In this project, we use the stochastic version of ordinary gradient, along with ADAM,- and RMSprop stochastic Gradient Descent. Check the codebase for full implementation of these algorithms along with the Neural Network class (see Alsvik et. al, [17]). The general ideas for implementing the pseudocode is taken from Goodfellow et. al. 6.5 in [18].

3.2. Practical considerations

Training deep neural networks can be very time consuming. Despite the great capacity of these networks to produce remarkable results, it might come at the expense

of other limitations. In this research, we've considered time complexity as a central part of testing neural networks. In the testfiles, there are consistent examples of time analysis for each neural network instance as function of model complexity. This is important in order to establish a tradeoff between performance and time consumption. Is a complex model always the best one?
Our neural network has at most a time complexity of $O(n^3)$, since the most computationally demanding method is the *train_SGD* with *functional*=True, containing a triple for-loop. See this method in the NeuralNetwork class in *classes.py* in our sourcecode, Alsvik et. al. ([17]).

3.3. Code verification and benchmark testing

An important part of software development is to verify that the workings of the algorithms match those of well-known Python machine learning libraries such as *sklearn* and *pytorch*. Furthermore, validating the results from the network is equally important to ensure that our network provides about the same outputs as the ones of a Python library. We plotted predictions for regression and classification against *Pytorch*, and *Sci-kit learn*. We found that there was a slight difference in the predictions, but at the same order of magnitude. Our conclusion is that our neural network performs well enough to provide good and consistent results. The file ***tests_c.ipynb*** in our GitHub repository, Alsvik et. al, [17] contains all benchmark tests with comments, with and without regularization.

3.4. Use of AI tools

We used ChatGPT as a sparring partner for discussion of the theory, as well as for testing our intuition. In the Theory and Methods section, it was also used to clean up spelling mistakes, clarify arguments and to make general improvements in the text. Other than this, ChatGPT have been used for help with the coding, particularly with finding errors in our code and debugging. It was also used for help with formatting the project in Overleaf. We have included an overview of the prompts and replies we got from ChatGPT in our Github repository (see the folder *LLM* in Alsvik et. al. ([17]).

4. RESULTS AND DISCUSSION

In this section, we present our results for the neural network on regression and classification. For regression, we consistently tested on a noise $N \sim (0, 0.1^2)$ and on 1000 datapoints, for the 1D Runge's function. We start by comparing linear regression from project 1, see Alsvik and Vestly ([1]), to our feed-forward neural network. Then we check for overfitting and the effect of regularization. For classification, we look at accuracy of our classifier for predicting handwritten digits between 0 and 9. We tested both with the full dataset of 74 000 images, but also with a smaller version of the dataset (1797 images) for speedup. Lastly, we consider the use of different activation functions for classification, and compare the results. In this report, we only show the most insightful results. On GitHub we have listed all our figures, see Alsvik et. al. ([17]).

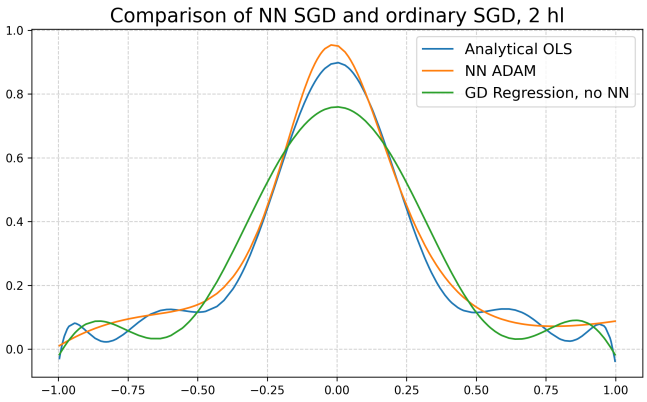

Figure 3. Comparison between NN and OLS methods

Figure 3 shows the comparison between linear regression with and without the use of neural networks, for ordinary least squares (OLS). The neural network has two hidden layers with 50 nodes. Polynomial degree 10 is used for linear regression without the use of neural networks.

4.1. Regression

Interestingly, as we can see in Figure 3 it becomes apparent that with the use of neural networks, the prediction for the Runge's function is smooth across the whole interval, i.e., without oscillations. Using two hidden layers, the neural network prediction is virtually identical to the one of the Runge's function, and hence plotting the Runge's function would overlap with this prediction. This figure is used to merely showcase the effect of using a neural network on linear regression. The use of non-linear activation functions, and excluding the use of a linear polynomial basis, provides flexibility for the network to learn more complex patterns, and therefore approximate the function better than traditional linear regression.

In figure 4, there is a clear tendency that the model overfits the training data by picking up too much noise and following the direction of the train data. By using fewer hidden layers and fewer nodes per hidden layer, we do not see any overfitting (see Figure 5). This is in accordance to our hypothesis, since the network has fewer datapoints to train on (less variance), and not enough complexity to learn the structure of the function deeply (generalization).

By using regularization for deep neural networks, we

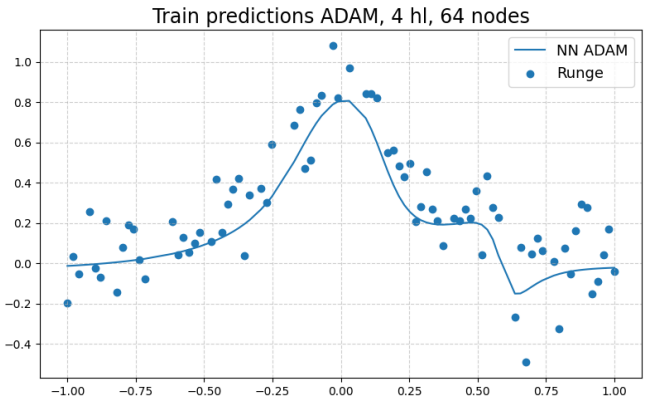

Figure 4. SGD ADAM optimizer with batchsize 32, 4 hidden layers with 64 nodes in each, and 100 datapoints. N $\sim (0, 0.15^2)$.

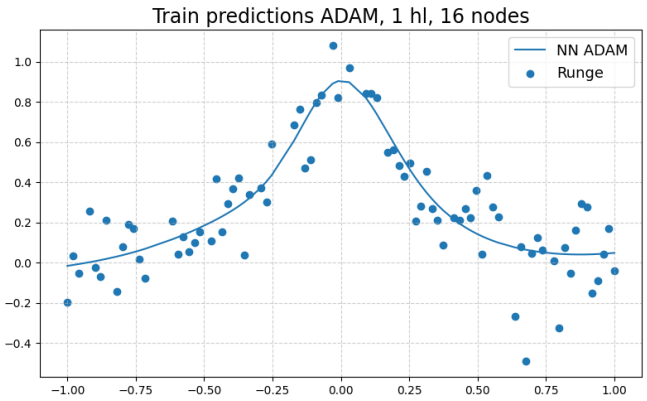

Figure 5. SGD ADAM optimizer with batchsize 32, 1 hidden layer with 16 nodes, and 100 datapoints. N $\sim (0, 0.15^2)$.

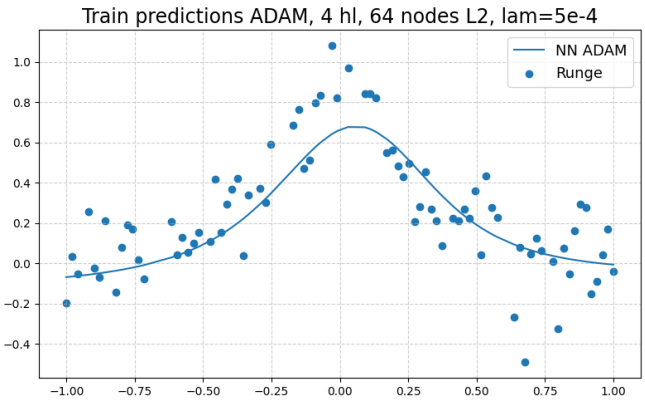

Figure 6. L2 regularization for a neural network with 4 layers, 64 nodes, lambda = 0.0005, and 100 datapoints. N $\sim (0, 0.15^2)$.

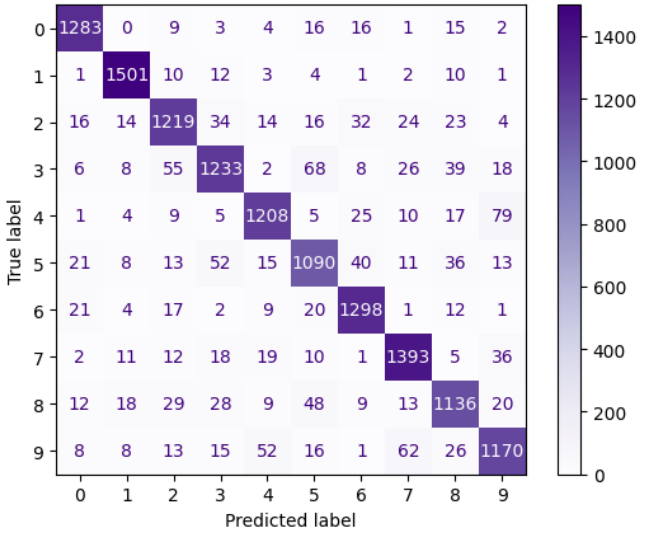

Figure 7. Confusion matrix for classifying digits between 0 and 9, with rows as true classes and columns as predicted classes. Using a neural network with one hidden layer with 50 nodes and Plain SGD with batchsize 100.

get a smoother prediction than the one without. In figure 6, we use L2 regularization to show this, but have also used L1 (see figure *l1_d_deep* in the Figures folder on GitHub by Alsvik et.al, [17]), but the plots look fairly similar, and is included merely to show the effects of regularization to cope with overfitting as the neural network gets deeper. For this particular case, using more hidden layers doesn't improve performance significantly, so having fewer hidden layers will lead to a lower computational cost and a more generalized model.

4.2. Classification

For the MNIST dataset we had 74 000 images of handwritten digits. The confusion matrix in Figure 7 shows how the model classified each individual image on the test data (n=14 800 images). The diagonal shows the amount of correctly classified images (TP, *true positives*). The sum of each column is how many images the classifier has predicted to belong to the class j. For class 0, we see that the classifier predicted 1283 correctly, but misclassified 67 images. For instance, the classifier classified the digit 0 to be 6, 16 times (FN, *false negative*). Equally, if we look at the rows, we can see that the classifier predicts images of 6's to be 0, 21 times (FP, *false positive*). However, this is just a raw count of predicted vs. true labels. In Figure 8, it's interesting to look at the diagonal elements, as they now represents the accuracy of the classifier for each class. The classifier predicts the digit 0 best by an accuracy of 95% and struggles the most with predicting the digit 5 (only 81% accuracy). The non-diagonal elements represents the fraction of which the model did wrong predictions. We see that these fractions are mostly small values across the matrix. However, the model predicted the number 3 to be 5, 68 times (8% error rate), being fairly high for a class containing about

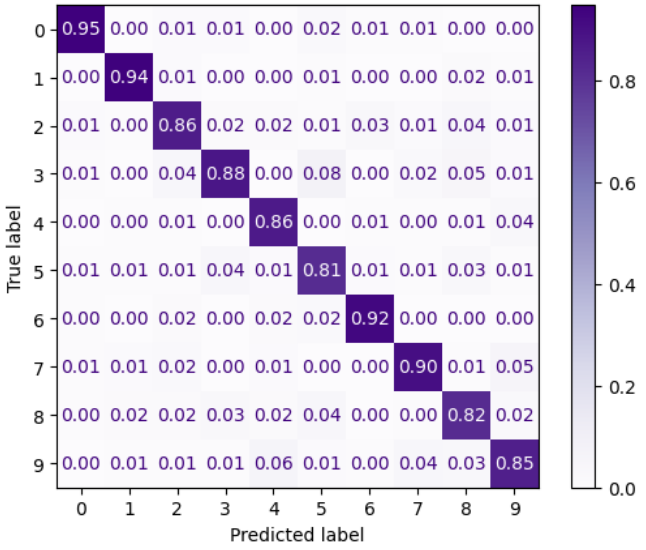

Figure 8. Normalized confusion matrix where each element is divided by the number of total images belonging to its class.

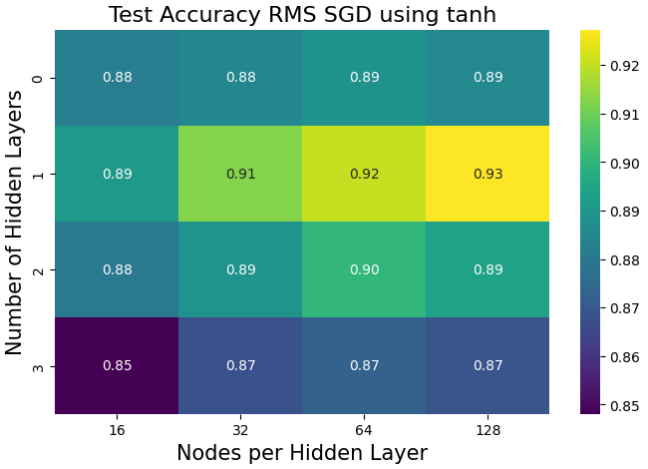

Figure 9. Accuracy for multiclass classification

9 shows accuracy plotted as a function of the depth of the network (no. of hidden layers), and number of nodes. In this particular case, by using RMS SGD optimizer with the hyperbolic tangent as the activation function in the hidden layers, and batchsize 4096.

1400 images.

Figure 9 serves as an illustrative example of (1) the *performance* of the network as a function of complexity, and (2) *overfitting*, using classifiers for multiclass logistic regression. Figure 11 shows the corresponding time consumption for each classifier. We can see that despite it taking longer to train the network as amount of hidden layers and nodes increases, the amount of time is still tolerable within our given use-case. Time is roughly linear in this example, O(n). However, for deeper neural networks, it will eventually be computationally infeasible to perform the training. In our case, we do not get

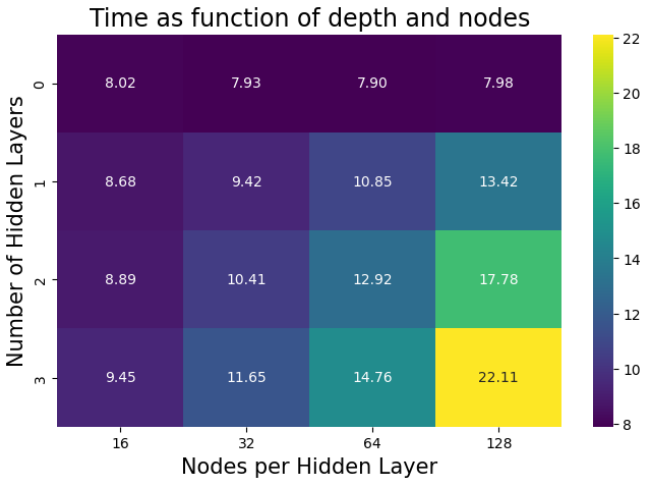

Figure 10. Time taken to create and train one neural network with a given amount of hidden layers and nodes in each hidden layer. Using the same optimizer and activation function as in 9

any better results from higher complexity. Conversely, we get overfitting when predicting on the test set. We note that using one layer is the best approach with regards to both performance and time complexity. Using a network of one layer and between 32 and 128 nodes is therefore, with this particular dataset, the sweet-spot for our analysis. These parameters were namely used when making Figure 7. Nevertheless, we see that regular mutliclass logistic regression (no hidden layers) is complex enough to decently learn from the data, and adding layers doesn't increase performance significantly, although using one layer increases the performance without any significant time increase.

4.3. Gradient optimizers and activation functions

In project 1, we looked at different Gradient Descent optimizers and compared them in terms of convergence rate, consistency, and different hyperparameter sets, see Alsvik & Vestly ([1]). In this research project, we'll consider the performance of different optimizers using different activation functions and batchsizes. Since we've already studied Gradient Descent on regression, we'll present the results for classification in this part.

Table I clearly shows the relationship between performance, time, and architectural complexity (memory) for different activation functions with their optimizer. In accordance to our theory, ReLU is faster than sigmoid, since its computation is cheaper. ReLU also *rectifies* vanishing gradients, and although one should expect a slightly better performance in general, sigmoid and ReLU obtains equal accuracy in this case. Additionally, the best accuracy for ReLU is achieved by ADAM as optimizer, which is usually faster than RMS. One also expects that leaky ReLU is slightly better than ReLU, ecpecially

Activation function	Optimizer	Accuracy	Time (s)	(Hidden, nodes)
Sigmoid	RMS	0.978	6.22s	(3, 64)
ReLU	ADAM	0.978	4.59s	(2, 128)
Leaky ReLU	ADAM	0.981	3.38s	(2, 64)
ELU	RMS	0.978	7.93s	(2, 128)
GELU	RMS	0.975	4.85s	(2, 128)
Tanh	ADAM	0.964	1.51s	(1, 64)

Table I. Best accuracy for each activation function, with corresponding oprimizer. Epochs ranging from 500 to 2000, with $\eta \in \{0.01, 0.1\}$

.

if ReLU suffers from dead nodes (see 2.7, 2). We observe that leaky ReLU performs better with a lower time-, and memory complexity (we use *memory* as a reference to network size, knowing that FFNN doesn't have any memory (feedback) from previous layers). Furthermore, we see that ELU and GELU performs about the same as ReLU, though with a higher computational cost. ELU takes almost 8s for the same memory as GELU and ELU. Interestingly, RMS is the optimizer yielding the best results for these, but it's ADAM for ReLU. This can be the reason for the time gap, in addition to ReLU likely being cheaper, since the gradient are either 0 or 1.
Lastly, the fastest and most memory efficient of them all is the hyperbolic tangent, tanh. Despite its performance being the worse, it is still very acceptable for a 10-class classifier. It needs no more than one layer and 1.5 s to achieve this accuracy.

5. CONCLUSION

We divided the result section into three parts.
We did regression, with and without the use of neural networks. We found that neural networks performance significantly better than linear regression. For the former, the MSE was about one to two orders of magnitude lower than for linear regression. Since the network doesn't follow a linear learning constraint, it picks up more complex, non-linear patterns in the data, and approximated the 1D Runge's function to an arbitrary accuracy. This results exhibits the *universal approximation theorem* for feedforward neural networks (see Section 2.4 in 2). We also considered overfitting as complexity grows, and mitigated this by using regularization.

In classification, we looked at accuracy, visually displaying it as a confusion matrix. Moreover, we considered both accuracy and time as function of network depth (number of hidden layers), and number of nodes for each hidden layer. We found that there is not necessarily a positive relationship between complexity and performance (accuracy), and that one hidden layer performed better than two and three, with the same amount of nodes. Also, time grows about linearly with complexity, and the optimal tradeoff between time and performance is found for few layers.

When studying different optimizers and activation functions. We found that leaky ReLU gave the best accuracy, although the difference between different activation functions was fairly negligible. ADAM with a complexity of 2 hidden layers and 64 nodes gave the best accuracy for leaky ReLU, with a time of 3.4s, and accuracy of 0.981. The fastest and most memory efficient was tanh with 1 hidden layer and 64 nodes, spending only 1.5s training, yielding 0.964 in accuracy with ADAM as optimizer.

Despite focusing on time-complexity and a comparative analysis of different activation functions, we did not find significant differences. Neither did we find obvious advantages and disadvantages for using different optimizers. We saw that ADAM was, as expected, slightly faster than RMS, but this does not provide any deeper insight than what we have found from using these methods on linear regression. It is possible that our datasets might not be complex enough to clearly illustrate the specific properties of the different optimizers. For future research, we would like to analyze more complex datasets with other neural network architecture, which could lead to a clearer distinction between the different activation functions and optimizers.

[1] A. S. Alsvik and J. Vestly, *Regularized Linear Regression Methods Applied to Runge's Function*, Project Report FYS-STK4155 Project 1 (University of Oslo, 2025).

[2] H. Khodabakhsh, Mnist dataset of handwritten digits, `https://www.kaggle.com/datasets/hojjatk/mnist-dataset`, accessed: 2025-31-10.

[3] M. Kosinski, "what is black box ai and how does it work?" (2024), accessed: 2025-24-10.

[4] A. Cosgun and J. Nies, Runge's phenomenon and its implications in interpolation, `https://math.uni.lu/eml/assets/reports/2022/runge.pdf` (2022), accessed: 2025-24-10.

[5] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review **65**, 386 (1958).

[6] C. Isaac, K. Zareinia, and M. Jain, Effect of excessive neural network layers on overfitting, World Journal of Advanced Research and Reviews **16** (2022).

[7] I. Neutelings, Neural network diagrams in tikz, `https://tikz.net/neural_networks/`, accessed: 2025-29-10.

[8] M. Hjorth-Jensen, Constructing a neural network code with examples, `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week42.html#the-derivatives` (2025), lecture notes for FYS-STK4155/3155: Applied Data Analysis and Machine Learning.

[9] M. Hjorth-Jensen, Neural networks and constructing a neural network, https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html (2025), lecture notes for FYS-STK4155/3155: Applied Data Analysis and Machine Learning.

[10] G. Cybenko, Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals, and Systems **2**, 303 (1989).

[11] K. Hornik, Approximation capabilities of multilayer feedforward networks, Neural Networks **4**, 251 (1991).

[12] M. Hjort-Hensen, Lecture notes: Chapter 2 – neural networks and deep learning, https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/chapter2.ipynb (2025), accessed: 2025-06-11.

[13] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. (O'Reilly Media, Sebastopol, CA, 2019).

[14] S. Mehta, Deriving categorical cross-entropy and softmax, https://shivammehta25.github.io/posts/deriving-categorical-cross-entropy-and-softmax/#softmax (2023), accessed: 2025-07-011.

[15] T. Kurbiel, Derivative of the softmax function and the categorical cross entropy loss, https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-cro (2021), accessed: 2025-07-011.

[16] S. Mehta, Universal approximatin theorem - the intuition, https://shivammehta25.github.io/posts/universal-approximation-theory-the-intuition/ (2023), accessed: 2025-09-11.

[17] A. Alsvik, J. Vestly, and Kristoffer DH Stalker, Project source code, https://github.com/JVestly/Project-2 (2025).

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

.

Appendix A: Selfie With a Legend

Figure 11. Selfie with Yoshua Bengio. He told us that he did not have a favorite activation function, but he seemed partial to ReLU. "It is all about the objective function"