CS 4/591: Neural Network
Assignment 3: Basic Deep Learning Algorithms

Jyrus Cadman Robert McCourt Bethany Peña Gabriel Urbaitis

15 November 2024

Contents

1 Introduction

Our previous work focused on implementing a foundational Feedforward Neural Network (FNN) in Python, where we developed a modular, object-oriented architecture capable of performing supervised learning tasks. This initial implementation featured forward and backward propagation for computing network outputs and calculating loss gradients, respectively, utilizing both standard and stochastic gradient descent for parameter optimization. Previously, the network's effectiveness was demonstrated across three distinct applications: simple regression, modeling the Van der Pol system's one-step reachability, and classifying handwritten digits from the MNIST dataset.

While our initial implementation provided valuable insights into multi-layer neural networks and fundamental learning algorithms, we observed limitations when scaling to deeper architectures with three or more layers. This challenge aligns with a well-documented phenomenon in neural network training, where traditional gradient descent methods become less effective as network depth increases. The current project builds upon our previous implementation to address these limitations by incorporating more sophisticated optimization algorithms and initialization techniques designed specifically for deep learning applications.

The primary objective of this updated implementation is to enhance our existing FNN implementation with four advanced techniques:

1. **Xavier Initialization**, which establishes optimal initial weight distributions based on layer dimensions to facilitate better gradient flow
2. **Nesterov Momentum-Based Learning**, which introduces momentum terms to accelerate convergence and avoid local minima
3. **Adam (Adaptive Moment Estimation)**, which combines the benefits of momentum with adaptive learning rates
4. **Newton's Method**, which utilizes second-order derivatives to improve optimization efficiency

Our implementation maintains the original modular architecture while introducing these new optimization strategies and initialization method, allowing for direct performance comparisons with our previous gradient descent approaches. We evaluate these methods using our established MNIST classification task and introduce the Iris Dataset from `scikit-learn` specifically chosen to challenge deeper network architectures. This comparative analysis focuses on key performance metrics including training time, convergence rate (measured in epochs), and prediction accuracy on holdout test sets.

Through this extension of our previous work, we aim to demonstrate the practical benefits and limitations of different optimization strategies and initialization techniques in deep learning, while developing a deeper understanding of the mathematical principles that underpin these advanced algorithms. This report details our implementation approach, provides comprehensive performance comparisons, and offers insights into the relative strengths of each method in different learning contexts.

2 Xavier Initialization

Xavier initialization is a weight initialization technique designed to keep the scale of gradients roughly the same in all layers of a neural network. This is particularly important in deep networks where improper initialization can lead to either vanishing or exploding gradients. The method sets initial weights by drawing

from a uniform distribution with limits calculated based on the number of input and output connections in the network layers.

In our FNN architecture, we implemented Xavier initialization by adding it as an initialization option in our Layer class. The initialization is performed through the `_xavier_init` helper method:

```
def _xavier_init(self, n_input, n_output):
    """
    Helper method for Xavier initialization with uniform distribution.
    """
    limit = np.sqrt(0.5 / (n_input + n_output))
    return np.random.uniform(-limit, limit, (n_input+1, n_output))
```

The implementation follows these key principles:

1. The limit for the uniform distribution is calculated as `sqrt(0.5 / (fan_in + fan_out))`, where:
   - `fan_in` is the number of input connections (`n_input`)
   - `fan_out` is the number of output connections (`n_output`)
2. The weights matrix includes an additional row for the bias term (`n_input+1`)
3. The initialization is selectable through the Layer constructor:

```
def __init__(self, n_input, n_output, init_type, activation):
    if init_type == 'xavier':
        self.weights = self._xavier_init(n_input, n_output)
    else:
        self.weights = np.random.uniform(-1, 1, (n_input+1, n_output))*.5
```

This implementation helps maintain stable gradients during training by ensuring that the variance of the weights is appropriate for the layer dimensions. The variance scaling helps prevent the aforementioned vanishing and exploding gradient problems, particularly in deeper networks where these issues are more pronounced.

3 Nesterov Momentum-Based Learning

Nesterov Momentum-Based Learning is an advanced optimization technique that improves upon traditional momentum by calculating gradients at a "look-ahead" position. This approach allows the algorithm to be more responsive to changes in the gradient, leading to better convergence rates than standard momentum methods. The key innovation is that it first applies the velocity to the parameters before computing the gradient, providing a more accurate update direction.

Implementation

Our implementation of Nesterov Momentum is integrated into the FNN class with several key components:

1. Initialization of momentum-related parameters:

```
def __init__(self, layers, momentum=0.9, use_nesterov=False):
    self.layers = layers
    self.momentum = momentum
    self.use_nesterov = use_nesterov
    # Initialize velocities for each layer
    for layer in self.layers:
        layer.velocity = np.zeros_like(layer.weights)
```

2. The core Nesterov update mechanism:

```
def nesterov_momentum_update(self, gradients_W, learning_rate):
    for layer, grad_W in zip(self.layers, gradients_W):
        # Update velocity
        layer.velocity = self.momentum * layer.velocity - learning_rate * grad_W
        # Update weights using Nesterov momentum
        layer.weights += layer.velocity
```

3. Integration with stochastic gradient descent:

```
def sgd(self, X, y, batch_size, learning_rate, use_adam, loss_func='mse'):
    # ...
    if self.use_nesterov:
        # Look ahead with current velocity
        for layer in self.layers:
            layer.weights += self.momentum * layer.velocity

        # Forward pass with look-ahead weights
        y_pred = self.forward(X_batch)

        # Revert weights
        for layer in self.layers:
            layer.weights -= self.momentum * layer.velocity

        # Backward pass
        gradients = self.backward(y_batch, y_pred, loss_func)

        # Update with Nesterov momentum
        self.nesterov_momentum_update(gradients, learning_rate)
```

The implementation follows these key steps:

1. **Look-Ahead Step:** Before computing gradients, we temporarily update the weights using the current velocity.
2. **Gradient Computation:** We compute the gradient at this look-ahead position.
3. **Weight Update:** The velocity is updated using the computed gradient, and then applied to the weights.

This approach differs from standard momentum in that it computes gradients at the predicted next position rather than the current position, allowing for more accurate corrections to the optimization trajectory. The momentum parameter (default 0.9) determines how much of the previous velocity is retained in each update, helping to smooth out oscillations while maintaining movement in promising directions.

Our implementation allows for easy switching between standard SGD and Nesterov momentum through the `use_nesterov` flag, making it simple to compare performance between the two approaches.

4 Adam Algorithm

Adam, which stands for Adaptive Moment Estimation, is well suited for training deep neural networks due to its ability to compute individual adaptive learning rates for different parameters. It customizes each parameter's learning rate based on its gradient history, which helps the neural network learn efficiently as a whole. Here are a few algorithms that attempt momentum-based learning and parameter-specific adaptations, which Adam builds upon and improves:

- **AdaGrad:** AdaGrad adjusts the learning rate based on the sum of the squared gradients for each parameter. This allows it to adapt to the geometry of the problem, making it useful for sparse data. However, its main limitation is that the learning rate monotonically decays, often leading to premature convergence. This makes it unsuitable for non-sparse, larger-scale problems.
- **RMSProp:** RMSProp modifies AdaGrad by using a moving average of squared gradients rather than the sum, which helps to address the issue of monotonically decaying learning rates. While RMSProp is more efficient for non-sparse problems and often leads to faster convergence, it still does not incorporate momentum, which can lead to oscillations in gradient directions.
- **Adam:** Adam combines the best aspects of both AdaGrad and RMSProp. It incorporates momentum through the use of moving averages of both the gradients (first moment) and the squared gradients (second moment). This allows Adam to adaptively adjust learning rates while also leveraging past gradient information for faster convergence. Furthermore, by bias-correcting the moment estimates, Adam prevents the early iterations from having a disproportionate effect on the parameter updates, which leads to more stable and effective optimization, especially for deep neural networks.

4.1 AdaGrad

The AdaGrad algorithm keeps track of the aggregated square magnitude of the partial derivative with respect to each parameter over the course of the algorithm

Let A_i be the aggregate value for the ith parameter, then in each iteration, the following update is performed:

$$A_i \leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2$$

Where the update for the ith parameter w_i is as follows:

$$w_i \leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right)$$

Scaling the derivative inversely with A_i is a kind of "signal-to-noise" normalization because A_i only measures the *historical magnitude* of the gradient rather than it *sign*. As a result, the progress of AdaGrad will eventually stop making progress. It also uses "stale" scaling factors, which can decrease inaccuracy due to their "ancient" history.

4.2 RMSProp

Instead of simply adding the squared gradients to estimate A_i, RMSProp uses *exponential averaging*, so the progress is not slowed prematurely from *aggregated* values like in AdaGrad. RMSProp introduces a decay factor, $\rho \in (0, 1)$, and weight the squared partial derivatives occurring t updates ago by ρ^t. So, if A_i is the exponentially averaged value of the ith parameter w_i, then we update A_i as follows:

$$A_i \leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2$$

We take the square root of this value for each parameter to normalize its gradient, then the following update is used for the global learning rate, α:

$$w_i \leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i} \right)$$

The drawback of RMSProp is that the running estimate, A_i, of the second-order moment is biased in early iterations, meaning early iterations will have a larger impact than later ones.

4.3 Adam

The Adam algorithm uses a similer "signal-to-noise" normalization as AdaGrad and RMSProp by incorporating momentum into the update. A_i, the exponentially averaged value of the ith parameter w_i is updated in the same way as RMSProp:

$$A_i \leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L}{\partial w_i} \right)^2$$

And is implemented in our `fnn.py` class as follows:

```
for i, gradient in enumerate(gradients_W):
    self.A[i] = rho*self.A[i] + (1 - rho) * (gradient ** 2)
```

Where $\rho = 0.999$. At the same time, an exponentially smoothed value of the gradient is maintained for which the ith component is denoted by F_i. This smoothing is performed with a different decay parameter, ρ_f:

$$F_i \leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial L}{\partial w_i} \right)$$

And is implemented as follows in our `fnn.py` class:

```
for i, gradient in enumerate(gradients_W):
    self.F[i] = rho_f * self.F[i] + (1-rho_f) * gradient
```

Where $\rho_f = 0.9$. Then, the following update is used at learning rate α_t at the tth iteration:

$$w_i \leftarrow -\frac{\alpha_t}{\sqrt{A_t}} F_i$$

And is implemented as follows in our `fnn.py` class, along with the bias adjustment calculation:

```
for i, layer in enumerate(self.layers):

    A_hat_i = self.A[i] / (1 - (rho ** self.t))
    F_hat_i = self.F[i] / (1 - (rho_f ** self.t))

    # Calculate alpha_t for the current time step
    alpha_t = learning_rate * ((np.sqrt(1 - (rho ** self.t))) / (1 - (rho_f ** self.t)))

    # Calculate the adaptive step
    adaptive_step = alpha_t * F_hat_i / (np.sqrt(A_hat_i) + epsilon)

    # Update weights
    layer.weights -= adaptive_step
```

Both F_i and A_i are initialized to zero, which causes bias in early iterations. Our implementation uses $\sqrt{A + \epsilon}$ for better conditioning, where $\epsilon = 1e - 8$. The Adam algorithm is extremely attractive in training deep-neural-networks because of its incorporation of adaptive learning rates for each parameter, as well as

its ability to adjust based on both first and second moments of the gradients, making it robust to sparse gradients and noisy data.

5 Newton's Method

Many of the most common optimization algorithms used in neural networks use first-order partial derivatives. Second order methods, including Newton's Method, allow us to include information about how fast the gradient changes by computing second-order partial derivatives. When gradient changes are large, large updates can result in poor optimization results. Newton's Method allows for more robust updates based on the curvature of the loss function, and so will not be as impacted by large instantaneous rates of change in the gradient.

Implementing Newtons' Method in code proved to be a challenging task, and as a result we explored many different approaches to calculating the second order partial derivatives and obtaining the Hessian matrix.

The Hessian matrix is defined by $H_{ij} = \frac{\partial^2 L(\mathbf{M})}{\partial w_i w_j}$. It contains the second derivative of the loss with respect to each pairwise combination of weights.

Newton's Method uses the Hessian to iteratively update the weights according to the following iterative equation:

$$W^{(k+1)} \leftarrow W^{(k)} - \alpha H^{-1} \nabla L(W^{(k)})$$

This section, we describe our initial attempts at obtaining the hessian matrix and applying newton's method in our code. We attempted both a full hessian method, and approximating the hessian by calculating only the diagonal elements during back propagation.

The modified portion of the assignment asks us to illustrate how the second-order derivatives in a Hessian is computed step by step for a single chain of neurons. We had two team members working in parallel on this portion, and due to time constraints were unable to coordinate a final, combined solution, so we present both results (section 5.3 and 5.4).

5.1 Full Hessian Method

The full hessian method was an attempt to calculate the hessian via a closed form solution.

5.2 Diagonal Approximation Method

Computing the full hessian matrix, particularly the cross-weight terms is a difficult and computationally intensive task. We attempted to approximate the hessian by only calculating the diagonal terms of the hessian in our backpropagation algorithm.

Note that this method was implemented before the second part of the assignment was completed, so our understanding of Newton's method and calculating the second order derivatives at this point was still developing.

5.2.1 Implementation Details

The implementation follows closely the methods in the `FNN` and `Layer` that implement the backpropagation algorithm to calculate the gradients.

The `FNN` class contains the method `hessian_diagonal_backward` which begins by calculating the derivative of the loss using the output nodes, and then loops backwards through the layers in the network, obtaining their diagonal second derivatives, and propagating them to previous layers using the `hessian_diagonal` method.

The `hessian_diagonal` method in the `Layer` class begins with the propagated partial derivatives, then calculates the first and second derivative of the activation function with respect to the forward values at this layer.

We attempt to calculate the second order partial derivatives that make up the diagonal elements of the hessian matrix by applying the chain rule and product rule to the gradient. We obtain the term $\frac{\partial^L}{w_i^2}$ from this part.

We then update the backpropagation term, which represents the gradient with respect to the loss for the current layer.

It represents a first attempt to use the chain rule and product rule to compute the second-order derivatives with backpropogation. However, after performing the simpler exercise that involved working through the backpropagation steps, we are no longer convinced of the validity of this implementation.

5.3 Backpropogation with the Second Order Derivative

The following algorithm is used to calculate the first order partial derivatives using backpropogation:

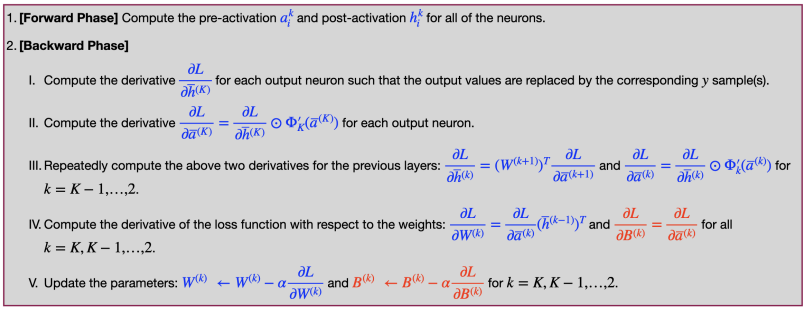

Figure 1: Algorithm for backpropogation provided in lecture.

We can extend this algorithm in the following way to obtain the second order partial derivatives. The forward phase remains the same, so we begin by modifying the backward phase.

We use the same notation as the algorithm where a_i^k is the pre-activation value of neuron i in layer k, and h_i^k is the post-activation value of neuron i in layer k.

Step 1:

Compute the second order derivatives $\frac{\partial^2 L}{\partial h_i^{2(k)}}$ and $\frac{\partial^2 L}{\partial h_i^{(k)} h_j^{(k)}}$ for each output neuron such that the output values are replaced by the corresponding y samples(s):

$$\frac{\partial^2 L}{\partial y_i^2} \text{ and } \frac{\partial^2 L}{\partial y_i y_j}$$

Step 2:

Compute the pre-activation second order derivative for each output neuron, $\frac{\partial^2 L}{\partial a_i^{2(k)}}$ and $\frac{\partial^2 L}{\partial a_i^{(k)} a_j^{(k)}}$.

If $\frac{\partial L}{\partial a_i^{(k)}} = \frac{\partial L}{\partial h_i^{(k)}} \odot \Phi_k'(a^{(k)})$, then:

$$\begin{aligned}\frac{\partial^2 L}{\partial a_i^{(k)^2}} &= \frac{\partial}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial h_i^{(k)}} \odot \Phi^{'}(a_i^{(k)})\right) \\ &= \frac{\partial L}{\partial h_i^{(k)}} \odot \Phi^{''}(a_i^{(k)}) + \frac{\partial^2 L}{\partial h_i^{(k)}} \odot \Phi^{'}(a_i^{(k)})\end{aligned}$$

We also need:

$$\begin{aligned}\frac{\partial^2 L}{\partial a_i^{(k)} a_j^{(k)}} &= \frac{\partial L}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial a_j^{(k)}}\right) \\ &= \frac{\partial L}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial h_j^{(k)}} \odot \Phi^{'}(a_j^{(k)})\right) \\ &= \frac{\partial^2 L}{\partial a_i^{(k)} h_j^{(k)}} \odot \Phi^{'}(a_j^{(k)}) + \frac{\partial L}{\partial h_j^{(k)}} \odot \Phi^{''}(a_j^{(k)}) \frac{\partial a_j^{(k)}}{\partial a_i^{(k)}}\end{aligned}$$

Step 3:

Repeatedly compute the second order derivatives for layers $k = K-1, \ldots, 2$:

We are given the following first order derivatives $\frac{\partial L}{\partial h_i^{(k)}} = w^{(k+1)} \frac{\partial L}{\partial a_i^{(k+1)}}$ and $\frac{\partial L}{\partial a_i^{(k+1)}} \frac{\partial L}{\partial h_i^{(k)}} \odot \Phi_i^{'}(a_i^{(k)})$.
We obtain the second order derivatives:

$$\begin{aligned}\frac{\partial^2 L}{\partial h_i^{2(k)}} &= \frac{\partial}{\partial h_i^{2(k)}}\left(w^{(k+1)} \frac{\partial L}{\partial a^{(k+1)}}\right) \\ &= w^{(k+1)} \frac{\partial^2 L}{\partial a_i^{2(k+1)}}\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 L}{\partial a_i^{2(k)}} &= \frac{\partial}{\partial h_i^{(k)}}\left(\frac{\partial L}{\partial h_i^{(k)}} \odot \Phi^{'}(a_i^{(k)})\right) \\ &= \frac{\partial L}{\partial h_i^{(k)}} \odot \Phi^{''}(a_i^{(k)}) + \frac{\partial^2 L}{\partial h_i} \odot \Phi^{'}(a_i^{(k)})\end{aligned}$$

So,

$$\frac{\partial^2 L}{\partial h_i^{2(k)}} = w^{(k+1)}\left(\frac{\partial L}{\partial h_i^k} \odot \Phi^{''}(a_i^{(k)}) + \frac{\partial^2 L}{\partial h_i^{(k)}} \odot \Phi^{'}(a_i^{(k)})\right)$$

Step 4:

Compute the second order derivative of the loss function with respect to the weights.
We are given the following first order derivative: $\frac{\partial L}{\partial W^{(k)}} = \frac{\partial L}{\partial a_i^{(k)}}(h_i^{(k-1)})$.

We obtain the second order derivatives:

$$\begin{aligned}\frac{\partial^2 L}{\partial^2 w_i^{(k)}} &= \frac{\partial}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial a_i^{(k)}} h_i^{(k-1)}\right) \\ &= \frac{\partial}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial a_i^{(k)}} \Phi(a_i^{(k-1))}\right) \\ &= \frac{\partial L}{\partial a_i^{(k)}} \Phi^{'}(a_i^{(k-1)}) + \frac{\partial^2 L}{\partial a_i^{2(k)}} \Phi(a_i^{(k-1)}) \frac{\partial a_i^{(k-1)}}{w_i^{(k)}}\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 L}{\partial w_i^{(k)} \partial w_j^{(k)}} &= \frac{\partial}{\partial a_i^{(k)}}\left(\frac{\partial L}{\partial a_j^{(k)}} \Phi(a_j^{(k-1)})\right) \frac{\partial a_j^{(k-1)}}{\partial w_j} \\ &= \left(\frac{\partial^2 L}{\partial a_i^{(k)} \partial a_j^{(k)}} \Phi(a_i^{(k-1)}) + \frac{\partial L}{\partial a_i^{(k)}} \Phi'(a_i^{(k-1)}) \frac{\partial a_i^{(k-1)}}{\partial w_j^{(k)}}\right) \frac{\partial a_j^{(k)}}{\partial w_j^{(k)}}\end{aligned}$$

5.3.1 Demonstration of Single Chain of Neurons (Case 1)

We demonstrate the backpropagation algorithm on a chain of five neurons as seen in figure 2.

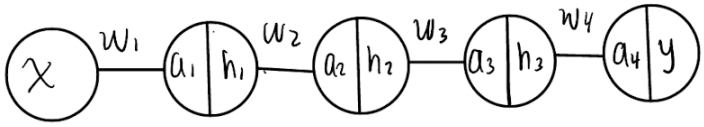

Figure 2: Algorithm for backpropogation provided in lecture.

We begin with a chain of neuron

Step 1

Compute $\frac{\partial^2 L}{\partial y^2}$

$$\frac{\partial^2 L}{\partial y_1^2} = 2$$

Step 2

Compute $\frac{\partial^2 L}{a_4^2}$, where a_1 is the pre-activation value of y.

$$\begin{aligned}\frac{\partial^2 L}{\partial a_4^2} &= \frac{\partial L}{\partial y} \odot \Phi^{''}(a_4) + \frac{\partial^2 L}{\partial y} \odot \Phi^{'}(a_4) \\ &= 2(y_{true} - y)\Phi(a_4)^{''} + 2\Phi(a_4)\end{aligned}$$

Step 3

For layers (4,3,2) we repeatedly compute the second order derivatives of the layer outputs by propagating backwards.

Hidden Layer 4

$$\begin{aligned}\frac{\partial^2 L}{\partial^2 h_3} &= w_4 \frac{\partial^2 L}{\partial a_1^2} \\ &= w_4 \left(\frac{\partial L}{y} \Phi^{''}(a_4) + \frac{\partial^2 L}{\partial y^2} \Phi^{'}(a_4) \right) \\ &= w_4 \left(2(y_{true} - y) \Phi^{''}(a_4) + 2\Phi^{'}(a_4) \right)\end{aligned}$$

$$\frac{\partial^2 L}{\partial^2 a_3} = \frac{\partial L}{\partial h_3} \Phi^{''}(a_3) + \frac{\partial^2 L}{\partial^2 h_3} \Phi^{'}(a_3)$$

Hidden Layer 3

$$\begin{aligned}\frac{\partial^2 L}{\partial^2 h_2} &= w_3 \frac{\partial^2 L}{\partial a_3^2} \\ &= w_3 \left[\frac{\partial L}{\partial h_3} \Phi^{''}(a_3) + \frac{\partial^2 L}{\partial^2 h_3} \Phi^{'}(a_3) \right]\end{aligned}$$

$$\frac{\partial^2 L}{\partial a_2^2} = \frac{\partial L}{\partial h_2} \Phi^{''}(a_2) + \frac{\partial^2 L}{\partial h_2^2} \Phi^{'}(a_2)$$

Hidden Layer 2

$$\begin{aligned}\frac{\partial^2 L}{\partial h_1} &= w_2 \frac{\partial^2 L}{\partial a_2^2} \\ &= w_2 \left[\frac{\partial L}{\partial^2 h_2} \Phi^{''}(a_2) + \frac{\partial^2 L}{\partial^2 h_2} \Phi^{'}(a_2) \right]\end{aligned}$$

$$\begin{aligned}&\frac{\partial^2 L}{\partial h_1} = w_2 \frac{\partial^2 L}{\partial a_2^2} \\ &w_2 \left[\frac{\partial L}{\partial h_1} \Phi^{''}(a_1) + \frac{\partial^2 L}{\partial h_1^2} \Phi^{'}(a_1) \right]\end{aligned}$$

Step 4

Now we can calculate the derivative of the losses with respect to the weights that we can use to populate the Hessian matrix. Since the matrix is symmetric, we compute the upper triangular part which we can use to fill the lower triangular part.

$$\frac{\partial^2 L}{\partial w_4^2} = \frac{\partial L}{\partial a_3}\Phi^{'}(a_3) + \frac{\partial^2 L}{\partial a_3^2}\Phi(a_3)$$
$$\frac{\partial^2 L}{\partial w_3^2} = \frac{\partial L}{\partial a_2}\Phi^{'}(a_2) + \frac{\partial^2 L}{\partial a_2^2}\Phi(a_2)$$
$$\frac{\partial^2 L}{\partial w_2^2} = \frac{\partial L}{\partial a_1}\Phi^{'}(a_1) + \frac{\partial^2 L}{\partial a_1^2}\Phi(a_1)$$
$$\frac{\partial^2 L}{\partial w_1^2} = \frac{\partial L}{\partial x}\Phi^{'}(x) + \frac{\partial^2 L}{\partial x^2}\Phi(x)$$
$$\frac{\partial^2 L}{\partial w_i^{(k)} \partial w_j^{(k)}} = \left(\frac{\partial^2 L}{\partial a_i^{(k)} \partial a_j^{(k)}} \Phi(a_i^{(k-1)}) + \frac{\partial L}{\partial a_i^{(k)}} \Phi'(a_i^{(k-1)}) \frac{\partial a_i^{(k-1)}}{\partial w_j^{(k)}} \right) \frac{\partial a_j^{(k)}}{\partial w_j^{(k)}}$$
$$\frac{\partial^2 L}{\partial w_4 \partial w_3} = \left(\frac{\partial^2 L}{\partial a_4 \partial a_3} \Phi(a_3) + \frac{\partial L}{\partial a_4} \Phi'(a_3) \frac{\partial a_3}{\partial w_3} \right) \frac{\partial a_4}{\partial w_4}$$
$$\frac{\partial^2 L}{\partial w_3 \partial w_2} = \left(\frac{\partial^2 L}{\partial a_3 \partial a_2} \Phi(a_2) + \frac{\partial L}{\partial a_3} \Phi'(a_2) \frac{\partial a_2}{\partial w_2} \right) \frac{\partial a_3}{\partial w_3}$$
$$\frac{\partial^2 L}{\partial w_2 \partial w_1} = \left(\frac{\partial^2 L}{\partial a_2 \partial a_1} \Phi(a_1) + \frac{\partial L}{\partial a_2} \Phi'(a_1) \frac{\partial a_1}{\partial w_1} \right) \frac{\partial a_2}{\partial w_2}$$

5.4 Case 1

Per the alternate assignment: "Illustrate how the entries (second-order derivatives) in a Hessian is computed step by step on the following example: Case 1: A neural network which has 1 input, 1 output and 3 hidden layers. Each layer has only one neuron. (It is like a chain of 5 neurons.)

Variable		1	2	3	4
x	2.000	**Note:** σ is tanh			
y	4.000				
w		0.460	0.500	-0.139	0.496
z		0.920	0.363	-0.048	-0.024
$\sigma(z)$		0.726	0.348	-0.048	-0.024
$\sigma'(z)$		0.473	0.879	0.998	0.999
$\sigma''(z)$		-0.687	-0.611	0.096	0.048

The loss function is the Mean Squared Error (MSE): $L = \frac{1}{2}(\hat{y} - y)^2$

Note: When running case1Newton, the factor of 2 was removed in layer's backward method to reflect this MSE function. self.z, self.a, self.activation_deriv(z) and self.activation_second_deriv were printed to calculate the forward pass. A random seed of 24 on random uniform initialization produced the weights. x and y were selected for simpler computation.

To calculate the error signal at the output layer: $\frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{2}(\hat{y} - y)^2 \right) = \hat{y} - y$

Next, $\hat{y} = \sigma_(z_4)$, so $\frac{\partial \hat{y}}{\partial z_4} = \sigma'(z_4)$
Using Chain rule: $\frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_4} = (\hat{y} - y) \cdot \sigma'(z_4)$
Since $z_4 = w_4 \cdot \sigma(z_3)$, we have: $\frac{\partial z_4}{\partial w_4} = \sigma(z_3) = a_3$
Getting our first gradient: $\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial w_4} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot a_3$

For: $\frac{\partial L}{\partial w_3}$, we first need $\frac{\partial L}{\partial a_3}$, which we propagate back using $\frac{\partial L}{\partial z_4}$.
$\frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3}$ and $\frac{\partial z_4}{\partial a_3} = w_4$, so $\frac{\partial L}{\partial a_3} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4$
Since $a_3 = \sigma(z_3)$ and $z_3 = w_3 \cdot a_2$, $\frac{\partial a_3}{\partial w_3} = \sigma'(z_3) \cdot a_2$ per chain rule.
Getting our second gradient: $\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial w_3} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot a_2$

For: $\frac{\partial L}{\partial w_2}$, we first need $\frac{\partial L}{\partial a_2}$, which we propagate back using $\frac{\partial L}{\partial z_3}$.
$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$ and $\frac{\partial a_3}{\partial a_2} = w_3 \cdot \sigma'(z_3)$, so $\frac{\partial L}{\partial a_2} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot w_3$
Since $a_2 = \sigma(z_2)$ and $z_2 = w_2 \cdot a_1$, $\frac{\partial a_2}{\partial w_2} = \sigma'(z_2) \cdot a_1$ per chain rule.
Getting our third gradient: $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_2} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot w_3 \cdot \sigma'(z_2) \cdot a_1$

For: $\frac{\partial L}{\partial w_1}$, we first need $\frac{\partial L}{\partial a_1}$, which we propagate back using $\frac{\partial L}{\partial z_2}$.
$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$ and $\frac{\partial a_2}{\partial a_1} = w_2 \cdot \sigma'(z_2)$, so $\frac{\partial L}{\partial a_1} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot w_3 \cdot \sigma'(z_2) \cdot w_2$
Since $a_1 = \sigma(z_1)$ and $z_1 = w_1 \cdot x$, $\frac{\partial a_1}{\partial w_1} = \sigma'(z_1) \cdot x$ per chain rule.
Getting our fourth gradient: $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} = (\hat{y} - y) \cdot \sigma'(z_4) \cdot w_4 \cdot \sigma'(z_3) \cdot w_3 \cdot \sigma'(z_2) \cdot w_2 \cdot \sigma'(z_1) \cdot x$

Weight	**Gradient** $\left(\frac{\partial L}{\partial w}\right)$
w_4	0.19415525
w_3	-0.69265235
w_2	0.17661406
w_1	0.11506182

Table 1: Gradients of the Loss with Respect to Each Weight

Gradients were calculated by printing grad_W in layer's backward method. They were confirmed by hand calculation.

5.4.1 Hessian Calculation

We will need the first order error propagations, they are collected here:
$\delta_4 = (\hat{y} - y) \cdot \sigma'(z_4) = -4.02165237$
$\delta_3 = \delta_4 \cdot w_4 \cdot \sigma'(z_3) = -1.9919189$
$\delta_2 = \delta_3 \cdot w_3 \cdot \sigma'(z_2) = 0.24329895$
$\delta_1 = \delta_2 \cdot w_2 \cdot \sigma'(z_1) = 0.05753091$
Error propagations were calculated by printing dL_dout after "dL_dout *= activation_deriv" in layer's backward method. They were confirmed by hand calculation.

5.4.2 Second Order Backward Pass 1

Let's begin by passing back $\frac{\partial L}{\partial w_1}$ as the "loss" in a second order backpropagation.

5.4.3 $\frac{\partial^2 L}{\partial w_1 \partial w_4}$ (Output Layer)

From: $\frac{\partial L}{\partial w_4} = \delta_4 \cdot a_3$, we can get: $\frac{\partial^2 L}{\partial w_1 \partial w_4} = \frac{\partial}{\partial w_1}\left(\frac{\partial L}{\partial w_4}\right) = \frac{\partial}{\partial w_1}(\delta_4 \cdot a_3)$
Using Product Rule: $\frac{\partial}{\partial w_1}(uv) = \frac{\partial u}{\partial w_1} \cdot v + u \cdot \frac{\partial v}{\partial w_1}$, we get: $\frac{\partial^2 L}{\partial w_1 \partial w_4} = \frac{\partial \delta_4}{\partial w_1} \cdot a_3 + \delta_4 \cdot \frac{\partial a_3}{\partial w_1}$
Plugging in from below, $\frac{\partial^2 L}{\partial w_1 \partial w_4} = (-0.023) \cdot (-0.048) + (-4.022) \cdot (-.058) = 0.001 + 0.233 = 0.234$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_1} = \left(\frac{\partial \hat{y}}{\partial w_1} \cdot \sigma'(z_4) + (\hat{y} - y) \cdot \sigma''(z_4) \cdot \frac{\partial z_4}{\partial w_1}\right)$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_1} = ((-.029) \cdot 0.999 + (-4.024) \cdot 0.048 \cdot (-.029)) = (-0.029) + 0.006 = -0.023$

To get $\frac{\partial \hat{y}}{\partial w_1}$:
$\hat{y} = \sigma(w_4 \cdot a_3)$, so $\frac{\partial \hat{y}}{\partial w_1} = \sigma'(z_4) \cdot w_4 \cdot \frac{\partial a_3}{\partial w_1} = 0.999 \cdot 0.496 \cdot (-0.058) = (-0.029)$
To get $\frac{\partial z_4}{\partial w_1}$:
$z_4 = w_4 \cdot a_3$, so $\frac{\partial z_4}{\partial w_1} = w_4 \cdot \frac{\partial a_3}{\partial w_1} = 0.496 \cdot (-0.058) = (-0.029)$
To get $\frac{\partial a_3}{\partial w_1}$:
$a_3 = \sigma(z_3)$, so $\frac{\partial a_3}{\partial w_1} = \sigma'(z_3) \cdot \frac{\partial z_3}{\partial w_1} = 0.998 \cdot (-0.058) = (-0.058)$
To get $\frac{\partial z_3}{\partial w_1}$:
$z_3 = w_3 \cdot a_2$, so $\frac{\partial z_3}{\partial w_1} = w_3 \cdot \frac{\partial a_2}{\partial w_1} = (-0.139) \cdot 0.416 = (-0.058)$
To get $\frac{\partial a_2}{\partial w_1}$:
$a_2 = \sigma(z_2)$, so $\frac{\partial a_2}{\partial w_1} = \sigma'(z_2) \cdot \frac{\partial z_2}{\partial w_1} = 0.879 \cdot 0.473 = 0.416$
To get $\frac{\partial z_2}{\partial w_1}$:
$z_2 = w_2 \cdot a_1$, so $\frac{\partial z_2}{\partial w_1} = w_2 \cdot \frac{\partial a_1}{\partial w_1} = 0.500 \cdot 0.946 = 0.473$
To get $\frac{\partial a_1}{\partial w_1}$:
$a_1 = \sigma(z_1)$, so $\frac{\partial a_1}{\partial w_1} = \sigma'(z_1) \cdot \frac{\partial z_1}{\partial w_1} = 0.473 \cdot 2.000 = 0.946$
To get $\frac{\partial z_1}{\partial w_1}$:
$z_1 = w_1 \cdot x$, so $\frac{\partial z_1}{\partial w_1} = x = 2.000$

5.4.4 $\frac{\partial^2 L}{\partial w_1 \partial w_3}$ (Layer 3)

From: $\frac{\partial L}{\partial w_3} = \delta_3 \cdot a_2$, we can get: $\frac{\partial^2 L}{\partial w_1 \partial w_3} = \delta_3^{(2)} \cdot a_2 + \delta_3 \cdot \frac{\partial a_2}{\partial w_1}$
$\frac{\partial^2 L}{\partial w_1 \partial w_3} = -.0003 \cdot 0.348 + (-1.992) \cdot 0.416 = -0.829$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_1} = \left(\frac{\partial \delta_4}{\partial w_1} \cdot w_4 \cdot \sigma'(z_3) + \delta_4 \cdot w_4 \cdot \left(\sigma''(z_3) \cdot \frac{\partial z_3}{\partial w_1}\right)\right)$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_1} = ((-0.023) \cdot 0.496 \cdot 0.998 + (-4.022) \cdot 0.496 \cdot (0.096 \cdot (-0.058))) = -.0003$

5.4.5 $\frac{\partial^2 L}{\partial w_1 \partial w_2}$ (Layer 2)

From: $\frac{\partial L}{\partial w_2} = \delta_2 \cdot a_1$, we can get: $\frac{\partial^2 L}{\partial w_1 \partial w_2} = \delta_2^{(2)} \cdot a_1 + \delta_2 \cdot \frac{\partial a_1}{\partial w_1}$
$\frac{\partial^2 L}{\partial w_1 \partial w_2} = -0.080 \cdot 0.726 + 0.243 \cdot 0.946 = 0.172$
$\delta_2^{(2)} = \frac{\partial \delta_2}{\partial w_1} = \left(\frac{\partial \delta_3}{\partial w_1} \cdot w_3 \cdot \sigma'(z_2) + \delta_3 \cdot w_3 \cdot \left(\sigma''(z_2) \cdot \frac{\partial z_2}{\partial w_1}\right)\right)$
$\delta_2^{(2)} = \frac{\partial \delta_2}{\partial w_1} = ((-0.0003) \cdot (-0.139) \cdot 0.879 + (-1.992) \cdot (-0.139) \cdot ((-0.611) \cdot 0.473)) = -0.080$

5.4.6 $\frac{\partial^2 L}{\partial w_1^2}$ (Layer 1)

From: $\frac{\partial L}{\partial w_1} = \delta_1 \cdot x$, we can get: $\frac{\partial^2 L}{\partial w_1^2} = \delta_1^{(2)} \cdot x + \delta_1 \cdot \frac{\partial x}{\partial w_1}$
x is the input and does not depend on w_1 , so $\frac{\partial x}{\partial w_1} = 0$. Therefore $\frac{\partial^2 L}{\partial w_1^2} = \delta_1^{(2)} \cdot x = (-0.186) \cdot 2.00 = -0.372$

$\delta_1^{(2)} = \frac{\partial \delta_1}{\partial w_1} = \left(\frac{\partial \delta_2}{\partial w_1} \cdot w_2 \cdot \sigma'(z_1) + \delta_2 \cdot w_2 \cdot \left(\sigma''(z_1) \cdot \frac{\partial z_1}{\partial w_1} \right) \right)$
$\delta_1^{(2)} = \frac{\partial \delta_1}{\partial w_1} = ((-0.080) \cdot 0.500 \cdot 0.473 + 0.243 \cdot 0.500 \cdot (-0.687 \cdot 2.000)) = -0.186$

5.4.7 Second Order Backward Pass 2

Let's continue by passing back $\frac{\partial L}{\partial w_2}$ as the "loss" in a second order backpropagation.

5.4.8 $\frac{\partial^2 L}{\partial w_2 \partial w_4}$ (Output Layer)

From: $\frac{\partial L}{\partial w_4} = \delta_4 \cdot a_3$, we can get: $\frac{\partial^2 L}{\partial w_2 \partial w_4} = \frac{\partial}{\partial w_2} \left(\frac{\partial L}{\partial w_4} \right) = \frac{\partial}{\partial w_2} (\delta_4 \cdot a_3)$
$\frac{\partial^2 L}{\partial w_2 \partial w_4} = \delta_4^{(2)} \cdot a_3 + \delta_4 \cdot \frac{\partial a_3}{\partial w_2} = (-0.242) \cdot (-0.048) + (-4.022) \cdot (-0.606) = 0.012 + 2.437 = 2.449$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_2} = \left(\frac{\partial \hat{y}}{\partial w_2} \cdot \sigma'(z_4) + (\hat{y} - y) \cdot \sigma''(z_4) \cdot \frac{\partial z_4}{\partial w_2} \right)$

$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_2} = ((-.300) \cdot 0.999 + (-4.024) \cdot 0.048 \cdot (-.301)) = -0.300 + 0.0581 = -0.242$

To get $\frac{\partial \hat{y}}{\partial w_2}$:
$\hat{y} = \sigma(w_4 \cdot a_3)$, so $\frac{\partial \hat{y}}{\partial w_2} = \sigma'(z_4) \cdot w_4 \cdot \frac{\partial a_3}{\partial w_2} = 0.999 \cdot 0.496 \cdot -0.606 = -0.300$
To get $\frac{\partial z_4}{\partial w_2}$:
$z_4 = w_4 \cdot a_3$, so $\frac{\partial z_4}{\partial w_2} = w_4 \cdot \frac{\partial a_3}{\partial w_2} = 0.496 \cdot -0.606 = -.301$
To get $\frac{\partial a_3}{\partial w_2}$:
$a_3 = \sigma(z_3)$, so $\frac{\partial a_3}{\partial w_2} = \sigma'(z_3) \cdot \frac{\partial z_3}{\partial w_2} = 0.998 \cdot -0.608 = -0.606$
To get $\frac{\partial z_3}{\partial w_2}$:
$z_3 = w_3 \cdot a_2$, so $\frac{\partial z_3}{\partial w_2} = w_3 \cdot \frac{\partial a_2}{\partial w_2} = -0.048 \cdot 0.638 = -0.608$
To get $\frac{\partial a_2}{\partial w_2}$:
$a_2 = \sigma(z_2)$, so $\frac{\partial a_2}{\partial w_2} = \sigma'(z_2) \cdot \frac{\partial z_2}{\partial w_2} = 0.879 \cdot 0.726 = 0.638$
To get $\frac{\partial z_2}{\partial w_2}$:
$z_2 = w_2 \cdot a_1$, so $\frac{\partial z_2}{\partial w_2} = a_1 = 0.726$

5.4.9 $\frac{\partial^2 L}{\partial w_2 \partial w_3}$ (Layer 3)

From: $\frac{\partial L}{\partial w_3} = \delta_3 \cdot a_2$, we can get: $\frac{\partial^2 L}{\partial w_2 \partial w_3} = \delta_3^{(2)} \cdot a_2 + \delta_3 \cdot \frac{\partial a_2}{\partial w_2}$
$\frac{\partial^2 L}{\partial w_2 \partial w_3} = (-0.004) \cdot 0.348 + (-1.992) \cdot 0.638 = -1.272$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_2} = \left(\frac{\partial \delta_4}{\partial w_2} \cdot w_4 \cdot \sigma'(z_3) + \delta_4 \cdot w_4 \cdot \left(\sigma''(z_3) \cdot \frac{\partial z_3}{\partial w_2} \right) \right)$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_2} = (-0.242 \cdot 0.496 \cdot 0.998 + (-4.022) \cdot 0.496 \cdot (0.096 \cdot (-0.606))) = (-0.120) + 0.116 = -0.004$

5.4.10 $\frac{\partial^2 L}{\partial w_2^2}$ (Layer 2)

From: $\frac{\partial L}{\partial w_2} = \delta_2 \cdot a_1$, we can get: $\frac{\partial^2 L}{\partial w_2^2} = \delta_2^{(2)} \cdot a_1 + \delta_2 \cdot \frac{\partial a_1}{\partial w_2}$
a_1 does not depend on w_2 , so $\frac{\partial a_1}{\partial w_2} = 0$ and $\frac{\partial^2 L}{\partial w_2^2} = \delta_2^{(2)} \cdot a_1 = -0.122 \cdot 0.726 = -0.089$
$\delta_2^{(2)} = \frac{\partial \delta_2}{\partial w_2} = \left(\frac{\partial \delta_3}{\partial w_2} \cdot w_3 \cdot \sigma'(z_2) + \delta_3 \cdot w_3 \cdot \left(\sigma''(z_2) \cdot \frac{\partial z_2}{\partial w_2} \right) \right)$
$\delta_2^{(2)} = \frac{\partial \delta_2}{\partial w_2} = (-0.004 \cdot -0.139 \cdot 0.879 + (-1.992) \cdot -0.139 \cdot ((-0.611) \cdot 0.726)) = -0.122$

5.4.11 $\frac{\partial^2 L}{\partial w_2 \partial w_1}$ (Layer 1)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_2 \partial w_1}$ is the same value as $\frac{\partial^2 L}{\partial w_1 \partial w_2}$.

5.4.12 Second Order Backward Pass 3

Let's continue by passing back $\frac{\partial L}{\partial w_3}$ as the "loss" in a second order backpropagation.

5.4.13 $\frac{\partial^2 L}{\partial w_3 \partial w_4}$ (Output Layer)

From: $\frac{\partial L}{\partial w_4} = \delta_4 \cdot a_3$, we can get: $\frac{\partial^2 L}{\partial w_3 \partial w_4} = \frac{\partial}{\partial w_3}\left(\frac{\partial L}{\partial w_4}\right) = \frac{\partial}{\partial w_3}(\delta_4 \cdot a_3)$
$\frac{\partial^2 L}{\partial w_3 \partial w_4} = \delta_4^{(2)} \cdot a_3 + \delta_4 \cdot \frac{\partial a_3}{\partial w_3}$

$\frac{\partial^2 L}{\partial w_3 \partial w_4} = 0.139 \cdot (-0.048) + (-4.022) \cdot 0.347 = (-0.007) + (-1.396) = -1.403$

$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_3} = \left(\frac{\partial \hat{y}}{\partial w_3} \cdot \sigma'(z_4) + (\hat{y} - y) \cdot \sigma''(z_4) \cdot \frac{\partial z_4}{\partial w_3}\right)$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_3} = (0.172 \cdot 0.999 + (-4.024) \cdot 0.048 \cdot 0.172) = 0.172 - 0.033 = 0.139$
To get $\frac{\partial \hat{y}}{\partial w_3}$:
$\hat{y} = \sigma(w_4 \cdot a_3)$, so $\frac{\partial \hat{y}}{\partial w_3} = \sigma'(z_4) \cdot w_4 \cdot \frac{\partial a_3}{\partial w_3} = 0.999 \cdot 0.172(frombelow) = 0.172$
To get $\frac{\partial z_4}{\partial w_3}$:
$z_4 = w_4 \cdot a_3$, so $\frac{\partial z_4}{\partial w_3} = w_4 \cdot \frac{\partial a_3}{\partial w_3} = 0.496 \cdot 0.347 = 0.172$
To get $\frac{\partial a_3}{\partial w_3}$:
$a_3 = \sigma(z_3)$, so $\frac{\partial a_3}{\partial w_3} = \sigma'(z_3) \cdot \frac{\partial z_3}{\partial w_3} = 0.998 \cdot 0.348 = 0.347$
To get $\frac{\partial z_3}{\partial w_3}$:
$z_3 = w_3 \cdot a_2$, so $\frac{\partial z_3}{\partial w_3} = a_2 = 0.348$

5.4.14 $\frac{\partial^2 L}{\partial w_3^2}$ (Layer 3)

From: $\frac{\partial L}{\partial w_3} = \delta_3 \cdot a_2$, we can get: $\frac{\partial^2 L}{\partial w_3^2} = \delta_3^{(2)} \cdot a_2 + \delta_3 \cdot \frac{\partial a_2}{\partial w_3}$
a_2 does not depend on w_3 , so $\frac{\partial a_2}{\partial w_3} = 0$ and $\frac{\partial^2 L}{\partial w_3^2} = \delta_3^{(2)} \cdot a_2$
$\frac{\partial^2 L}{\partial w_3^2} = 0.002 \cdot 0.348 = .0007$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_3} = \left(\frac{\partial \delta_4}{\partial w_3} \cdot w_4 \cdot \sigma'(z_3) + \delta_4 \cdot w_4 \cdot \left(\sigma''(z_3) \cdot \frac{\partial z_3}{\partial w_3}\right)\right)$
$\delta_3^{(2)} = \frac{\partial \delta_3}{\partial w_3} = (0.139 \cdot 0.496 \cdot 0.998 + (-4.022) \cdot 0.496 \cdot (0.096 \cdot 0.348)) = 0.069 - .067 = .002$

5.4.15 $\frac{\partial^2 L}{\partial w_3 \partial w_2}$ (Layer 2)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_3 \partial w_2}$ is the same value as $\frac{\partial^2 L}{\partial w_2 \partial w_3}$.

5.4.16 $\frac{\partial^2 L}{\partial w_3 \partial w_1}$ (Layer 1)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_3 \partial w_1}$ is the same value as $\frac{\partial^2 L}{\partial w_1 \partial w_3}$.

5.4.17 Second Order Backward Pass 4

Let's continue by passing back $\frac{\partial L}{\partial w_4}$ as the "loss" in a second order backpropagation.

5.4.18 $\frac{\partial^2 L}{\partial w_4^2}$ (Output Layer)

From: $\frac{\partial L}{\partial w_4} = \delta_4 \cdot a_3$, we can get: $\frac{\partial^2 L}{\partial w_4^2} = \frac{\partial}{\partial w_4}\left(\frac{\partial L}{\partial w_4}\right) = \frac{\partial}{\partial w_4}(\delta_4 \cdot a_3)$

$\frac{\partial^2 L}{\partial w_4^2} = \delta_4^{(2)} \cdot a_3 + \delta_4 \cdot \frac{\partial a_3}{\partial w_4}$ a_3 does not depend on w_4 , so $\frac{\partial a_3}{\partial w_4} = 0$ and $\frac{\partial^2 L}{\partial w_4^2} = \delta_4^{(2)} \cdot a_3$
$\frac{\partial^2 L}{\partial w_4^2} = 0.009 \cdot (-0.048) = 0.0004$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_4} = \left(\frac{\partial \hat{y}}{\partial w_4} \cdot \sigma'(z_4) + (\hat{y} - y) \cdot \sigma''(z_4) \cdot \frac{\partial z_4}{\partial w_4} \right)$

To get $\frac{\partial \hat{y}}{\partial w_4}$:
$\hat{y} = \sigma(w_4 \cdot a_3)$, so $\frac{\partial \hat{y}}{\partial w_4} = \sigma'(z_4) \cdot w_4 \cdot \frac{\partial a_3}{\partial w_4}$.
a_3 does not depend on w_4 , so $\frac{\partial a_3}{\partial w_4} = 0$ and so $\frac{\partial \hat{y}}{\partial w_4} = 0$,
thus $\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_4} = \left((\hat{y} - y) \cdot \sigma''(z_4) \cdot \frac{\partial z_4}{\partial w_4} \right)$
$\delta_4^{(2)} = \frac{\partial \delta_4}{\partial w_4} = ((-4.024) \cdot 0.048 \cdot (-0.048)) = 0.009$

To get $\frac{\partial z_4}{\partial w_4}$:
$z_4 = w_4 \cdot a_3$, so $\frac{\partial z_4}{\partial w_4} = a_3 = -0.048$

5.4.19 $\frac{\partial^2 L}{\partial w_4 \partial w_3}$ (Layer 3)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_4 \partial w_3}$ is the same value as $\frac{\partial^2 L}{\partial w_3 \partial w_4}$.

5.4.20 $\frac{\partial^2 L}{\partial w_4 \partial w_2}$ (Layer 2)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_4 \partial w_2}$ is the same value as $\frac{\partial^2 L}{\partial w_2 \partial w_4}$.

5.4.21 $\frac{\partial^2 L}{\partial w_4 \partial w_1}$ (Layer 1)

The Hessian is symmetrical so $\frac{\partial^2 L}{\partial w_4 \partial w_1}$ is the same value as $\frac{\partial^2 L}{\partial w_1 \partial w_4}$.

5.4.22 Full Hessian Matrix

The Hessian is a size n^2 matrix where n = the total number of weights in the network. Our 4x4 Hessian takes the following form:

$$H = \begin{bmatrix} \frac{\partial^2 L}{\partial w_1^2} & \frac{\partial^2 L}{\partial w_1 \partial w_2} & \frac{\partial^2 L}{\partial w_1 \partial w_3} & \frac{\partial^2 L}{\partial w_1 \partial w_4} \\ \frac{\partial^2 L}{\partial w_2 \partial w_1} & \frac{\partial^2 L}{\partial w_2^2} & \frac{\partial^2 L}{\partial w_2 \partial w_3} & \frac{\partial^2 L}{\partial w_2 \partial w_4} \\ \frac{\partial^2 L}{\partial w_3 \partial w_1} & \frac{\partial^2 L}{\partial w_3 \partial w_2} & \frac{\partial^2 L}{\partial w_3^2} & \frac{\partial^2 L}{\partial w_3 \partial w_4} \\ \frac{\partial^2 L}{\partial w_4 \partial w_1} & \frac{\partial^2 L}{\partial w_4 \partial w_2} & \frac{\partial^2 L}{\partial w_4 \partial w_3} & \frac{\partial^2 L}{\partial w_4^2} \end{bmatrix}$$

Filling it in with our calculations:

$$H = \begin{bmatrix} -0.372 & 0.172 & -0.829 & 0.234 \\ 0.172 & -0.089 & -1.272 & 2.449 \\ -0.829 & -1.272 & 0.0007 & -1.403 \\ 0.234 & 2.449 & -1.403 & 0.0004 \end{bmatrix}$$

6 Testing

6.1 MNIST Dataset

The MNIST dataset was used to test the performance of Xavier initialization, ADAM, Nesterov-Momentum, and Newton's Method on the same neural network architecture. The network was composed of four layers in total, with two hidden layers containing 64 neurons each. The hidden layers utilized the ReLU activation function, while the output layer employed the logsoftmax activation function. Below is the Python implementation that applies these techniques to the MNIST dataset.

6.2 Iris Dataset

The IRIS dataset was used to evaluate the performance of two optimization algorithms—Stochastic Gradient Descent (SGD) and Adam—on a neural network model. The network architecture consisted of four layers, including two hidden layers, each containing 10 neurons. ReLU activation functions were used in the hidden layers, with the output layer utilizing the logsoftmax activation function. The target labels were one-hot encoded, and the dataset was split into training and testing sets, followed by standardization of the features.

7 Results

7.1 MNIST Dataset

7.1.1 Comparison of ADAM Optimizer and Stochastic Gradient Descent on the MNIST Dataset

The performance of the ADAM optimizer and the Steepest (Stochastic) Gradient Descent (SGD) was evaluated on the MNIST dataset using a learning rate of 0.0001 over 100 epochs. The comparison, as summarized in Table 2, is based on two key metrics: computational time and classification accuracy.

In terms of computational efficiency, SGD demonstrated faster execution times compared to ADAM for both initialization schemes. Specifically, with Uniform initialization, SGD required 115.87 seconds, while ADAM took 180.92 seconds. Similarly, with Xavier initialization, SGD completed in 108.37 seconds, which is significantly quicker than ADAM's 180.92 seconds. This highlights the computational simplicity of SGD, which may be beneficial in time-critical applications.

However, when evaluating classification accuracy, ADAM outperformed SGD. As shown in Table 2, the ADAM optimizer achieved an accuracy of 96.85%, compared to 96.27% for SGD with Xavier initialization and 89.74% with Uniform initialization. This indicates that ADAM's adaptive learning rate mechanism contributes to better convergence and overall accuracy on this dataset.

These results suggest a trade-off between computational speed and accuracy. While SGD may be preferable for scenarios requiring faster training times, ADAM's superior accuracy makes it a better choice for applications prioritizing predictive performance.

7.1.2 Comparison of Nesterov vs Steepest Gradient Descent on the MNIST Dataset

Nesterov, which is a modification of traditional momentum-based gradient descent, outperforms standard SGD in this experiment. It achieves a higher accuracy of 96.65% compared to SGD's accuracy of 96.27%. Nesterov works by first making a "look-ahead" step, estimating the gradient based on the momentum, and

then applying the gradient descent step. This typically allows Nesterov to make more informed updates, resulting in faster convergence and sometimes better generalization.

On the other hand, SGD with a simple momentum term might be slower in terms of convergence, as seen in the results. Although SGD showed good performance with an accuracy of 96.27

7.1.3 Comparison of Newton vs Steepest Gradient Descent on the MNIST Dataset

We tested only the approximation version of Newton's method due to the run time of the alternate closed-form version. Over epochs, the test accuracy did not improve compared to the initial accuracy, indicating our model was not properly learning with this version of Newton's method. The variation of testing accuracy was within the range of variation from initial accuracy.

Method	Time (s)	Accuracy (%)	Initialization Type
Nesterov	150.63	96.65	Xavier
ADAM	180.92	96.85	Xavier
Newton's Method Approx 28x28	42.44	11.01	Uniform
Newton's Method Approx 14x14	44.17	11.52	Uniform
Newton's Method Approx 28x28	40.68	12.43	Xavier
Newton's Method Approx 14x14	44.64	9.43	Xavier
Stochastic Gradient Descent (1/2)	115.87	89.74	Uniform
Stochastic Gradient Descent (2/2)	108.37	96.27	Xavier

Table 2: Performance comparison of optimization methods with different initialization types on the MNIST dataset.

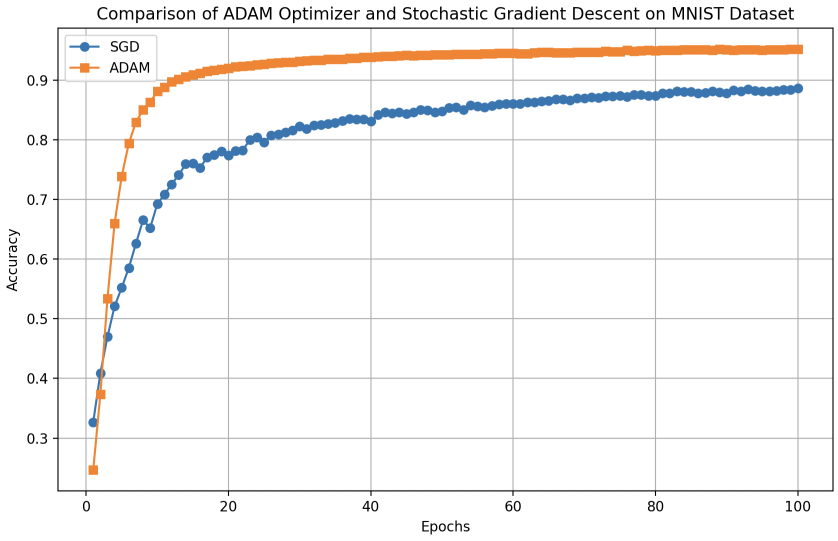

Figure 3: Comparison of accuracy of ADAM and Steepest Gradient Descent on the MNIST dataset

7.2 Iris Dataset

7.2.1 Comparison of ADAM and Steepest Gradient Descent on the IRIS Dataset

The performance of the ADAM optimizer and Steepest Gradient Descent (SGD) was evaluated on the IRIS dataset, focusing on computational time, classification accuracy, and model complexity. The comparison, as summarized in Table 3, highlights notable differences between the two methods under varying configurations.

In terms of accuracy, ADAM consistently achieved 100% classification accuracy, regardless of the model configuration. This demonstrates its robustness and ability to adapt effectively to the optimization landscape of the IRIS dataset. In contrast, the performance of SGD was highly sensitive to the initialization scheme and the number of neurons in the hidden layer. With Xavier initialization and 128 neurons, SGD reached an accuracy of 40%. However, when configured with Uniform initialization and 64 neurons, SGD achieved a notable accuracy of 93.33%, indicating that simpler models can perform well with SGD when paired with the appropriate initialization.

From a computational efficiency perspective, SGD outperformed ADAM. The fastest SGD configuration (128 neurons with Uniform initialization) completed in just 0.02 seconds, significantly faster than ADAM's runtime of 0.06 seconds. This highlights SGD's suitability for scenarios where rapid computation is a priority, particularly for simpler models.

Interestingly, SGD demonstrated improved performance with reduced model complexity (64 neurons compared to 128). This suggests that for simpler tasks, reducing the model size can enhance SGD's convergence and reduce overfitting. In contrast, ADAM's consistent accuracy across configurations reflects its strength in optimizing more complex models without a significant trade-off in computational time.

These results highlight the trade-offs between ADAM and SGD. While ADAM is ideal for achieving high accuracy with minimal tuning, SGD's efficiency and compatibility with simpler models make it a viable choice for lightweight or resource-constrained applications.

7.2.2 Comparison of Nesterov vs Steepest Gradient Descent on the IRIS Dataset

Nesterov, with its momentum-based updates and predictive nature, achieves a perfect accuracy of 100% with an exceptionally fast training time of just 0.16 seconds. This result highlights the effectiveness of Nesterov in optimizing the model for the IRIS dataset, where the momentum-based look-ahead step allows the method to converge quickly and reach an optimal solution.

7.2.3 Comparison of Newton vs Steepest Gradient Descent on the IRIS Dataset

We tested only the approximation version of Newton's method due to the run time of the alternate closed-form version. Although it appears based on the results reported in the table 3 that the method experienced some training success with Xavier, it appears the result is anomalous because on subsequent training runs, the accuracy is much worse and the training accuracy remains consistent throughout epochs, which indicates that the model is not learning with this optimization method.

Method	Time (s)	Accuracy (%)	Initialization Type	Neurons/Layer (Hidden)
Nesterov	0.16	100.00	Xavier	128
ADAM	0.06	100.00	Xavier	128
Steepest Gradient Descent (1/4)	0.03	20.00	Xavier	64
Steepest Gradient Descent (2/4)	0.06	40.00	Xavier	128
Steepest Gradient Descent (3/4)*	0.03	93.33	Uniform	64
Steepest Gradient Descent (4/4)	0.02	6.67	Uniform	128
Newton's Method Approx	0.06	60.00	Xavier	10
Newton's Method Approx	0.09	20.00	Uniform	10

Table 3: Performance comparison of optimization methods on the IRIS dataset. * indicates notable performance within the method's category.

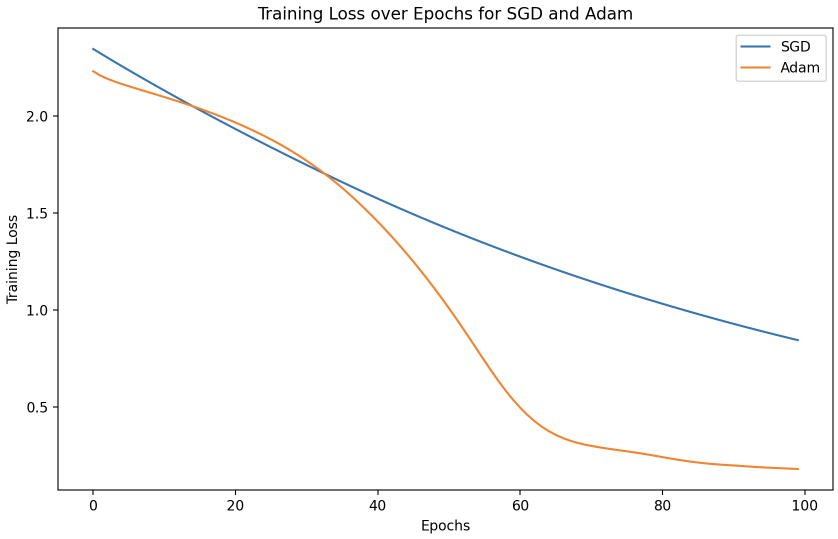

Figure 4: Comparison of training loss of ADAM and Steepest Gradient Descent on the IRIS dataset

8 Discussion

In this project, we were able to explore different optimization methods and compare them to gradient descent, which we have been using up until this point to train our models. Of the two datasets we tested, the MNIST dataset was the most complex. We observed that both Nesterov and ADAM achieved comparable results to stochastic gradient descent with Xavier initialization. These methods, however, achieved significantly better performance compared to gradient descent in the IRIS dataset. Although ADAM and Nesterov were computationally more expensive than gradient descent, the consistency of performance across datasets, may indicate they are more robust and reliable methods for optimization.

Our implementations of Newton's Method showed difficulty in training. This is likely due to errors in our implementation. Working through the exercise of demonstrating the Hessian calculation for a simple chain of neurons, improved our understanding of this method. It would be interesting to see if we could now obtain a more accurate implementation of Newton's method in our code after having worked through this exercise.

9 Conclusion

In this project, we explored and implemented three advanced optimization algorithms - Nesterov momentum, Adam, and Newton's method - on top of the steepest gradient descent algorithm from the midterm project. These deep learning algorithms were applied to neural networks with at least three layers.

We implemented each algorithm starting with Xavier initialization to ensure proper weight scaling, followed by Nesterov momentum to accelerate convergence, the Adam optimizer for adaptive learning rate, and Newton's method for second-order optimization. Through systematic testing, we compared these algorithms with shallow learning techniques using randomly generated datasets.

In conclusion, this assignment highlighted the importance of choosing the right optimization method for training deep neural networks. The results affirmed that advanced methods like Nesterov momentum, Adam, and Newton's method can provide significant improvements over shallow learning algorithms, particularly in terms of convergence speed and accuracy. However, they also come with trade-offs in terms of computational cost, especially for Newton's method. As deep learning continues to evolve, selecting the right algorithm for a given task will remain a crucial aspect of achieving optimal performance.

Contributions

Jyrus Cadman

Jyrus implemented two fundamental components that significantly impacted the network's performance across all optimization methods. He developed the Xavier initialization scheme in the `Layer` class, which provides more stable initial weights by accounting for layer dimensions, helping prevent vanishing and exploding gradients in deeper networks. This initialization method serves as the foundation for all optimization techniques implemented in this project. Jyrus also implemented the Nesterov Momentum-Based Learning algorithm in the `FNN` class, including the look-ahead gradient computation and velocity-based weight updates. His implementation allows for seamless switching between standard SGD and Nesterov momentum through a simple flag system. Additionally, Jyrus contributed to testing the network's performance across different optimization methods, helping evaluate the effectiveness of each approach under various conditions.

Robert McCourt

Robert implemented the ADAM optimizer in the FNN class to improve the efficiency and stability of the training process. ADAM, which stands for Adaptive Moment Estimation, is an advanced optimization algorithm that combines the advantages of two other popular optimizers: Momentum and RMSProp. It helps the model converge faster and more reliably by adapting the learning rate for each parameter based on the first and second moments of the gradients (i.e., the mean and variance of the gradients). The ADAM optimizer in this implementation is integrated into the FNN class, and can be activated by simply calling the train_adam or train_adam_batch functions with the necessary arguments. It allows for a more numerically stable and efficient training process compared to traditional gradient descent methods like stochastic gradient descent (SGD).

Bethany Peña

Bethany assisted Gabriel with the Newton's Method work. Bethany and Gabriel met to discuss and brainstorm the Newton's method work. Based on her initial understandings of backpropagation and the second order partial derivatives, she attempted to implement a simplified version by computing an approximation of the Hessian matrix, using only the diagonal terms. She tested this method on both the MNIST dataset and IRIS datasets, and found the method did not show signs of learning. Once the option of demonstrating a hessian calculation using the "case 1" chain of nodes, Bethany attempted to use the backpropagation algorithm presented in lecture to derive a backpropagation algorithm which calculated the second order partial derivatives needed to populate the hessian matrix (section 5.3). She attempted to demonstrate how this algorithm would calculate the derivatives needed to create a hessian matrix for the network described in case 1.

Gabriel Urbaitis

Gabriel performed all of the calculations (derivatives and numbers) for Case 1 of Newton's method (all of section 5.4). He attempted a closed form solution for the code including the hessian entry method in layer and hessian matrix and newton update methods in fnn. He attempted to run them on the iris dataset for case 4, and a lower resolution MNIST dataset in case 3 hessian, but found that the method showed no signs of learning. He attempted to debug with regularization of the hessian and gradient matrices, different logsoftmax changes, and other fixes but was unsuccessful so he moved on to the alternate option of calculating Case 1. Professor Chen asked us to note that we had a smaller team than average (3 undergrads and 1 grad student (myself). With his assistance and the adjustment down to the single case, we were luckily able to finish the project just in time.

10 Appendix

```python
import numpy as np

class FNN:
    """
    A Feed-Forward Neural Network.
    """

    # Initialize the network with a list of layers
    def __init__(self, layers, momentum=0.9, use_nesterov=False):
        self.layers = layers
        self.A = [np.zeros_like(layer.weights) for layer in self.layers]
        self.F = [np.zeros_like(layer.weights) for layer in self.layers]
        self.t=1
        # Add Nesterov-related parameters
        self.momentum = momentum
        self.use_nesterov = use_nesterov
        # Initialize velocities for each layer
        for layer in self.layers:
            layer.velocity = np.zeros_like(layer.weights)

    # Perform forward propagation through all layers
    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X

    """
    Calculate gradients for all layers.
    X: Input data
    y: True labels
    y_pred: Predicted output from the forward pass
    loss_func: Loss function ('mse' or 'nll')
    """
    def backward(self, y, y_pred, loss_func='mse'):
        if loss_func == 'mse':
            dL_dout = 2 * (y_pred - y) / y.shape[0]
        elif loss_func == 'nll':
            dL_dout = y_pred - y
        gradients_W = []
        # Proceeding backward through the layers, add each new calculation to the front
        # to create the gradients array
        #reg_lambda = 1e-2
        for layer in reversed(self.layers):
            grad_W, dL_dout = layer.backward(dL_dout)
            #grad_W += reg_lambda * layer.weights
            grad_W = np.clip(grad_W, -.5, .5)
            gradients_W.insert(0, grad_W)
        return gradients_W

    def hessian_diagonal_backward(self, y, y_pred, loss_func='mse'):
        # Initialize dL_dout based on the loss function
        if loss_func == 'mse':
            dL_dout = 2 * (y_pred - y) / y.shape[0]
        elif loss_func == 'nll':
            dL_dout = y_pred - y

        # Initialize dL_dout_prev for the first pass
        hessians_diag = []

        # Loop through layers in reverse order to accumulate second-order terms
        for layer in reversed(self.layers):
```

```
            # Each layer's hessian function should return both hessian_diag and an updated dL_dout for the next layer
            hessian_diag, dL_dout = layer.hessian_diagonal(dL_dout)
            hessians_diag.insert(0, hessian_diag)  # Insert at the beginning to accumulate in the correct order

        return hessians_diag

    def newton_update_diagonal_approx(self, X, y, learning_rate=0.01, loss_func='mse'):
        y_pred = self.forward(X)
        gradients = self.backward(y, y_pred, loss_func)
        diag_hessians = self.hessian_diagonal_backward(y, y_pred, loss_func)

        # Use Hessians (diagonal) and gradients to apply Newton's update rule
        for layer, grad_W, hessian_diag in zip(self.layers, gradients, diag_hessians):
            # Prevent division by zero by adding a small constant to hessian_diag
            # and invert only the diagonal
            hessian_diag_inv = 1.0 / (hessian_diag + 1e-8)
            update = learning_rate * (grad_W * hessian_diag_inv)
            # print(hessian_diag_inv)
            layer.weights -= update

    # Update weights and biases using gradient descent
    def gd(self, gradients_W, learning_rate):
        if self.use_nesterov:
            self.nesterov_momentum_update(gradients_W, learning_rate)
        else:
            for layer, grad_W in zip(self.layers, gradients_W):
                layer.weights -= learning_rate * grad_W

    def sgd(self, X, y, batch_size, learning_rate, use_adam, loss_func='mse'):
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)

        for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
            batch_indices = indices[start_idx:start_idx + batch_size]
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]

            if self.use_nesterov:
                # Look ahead with current velocity
                for layer in self.layers:
                    layer.weights += self.momentum * layer.velocity

                # Forward pass with look-ahead weights
                y_pred = self.forward(X_batch)

                # Revert weights
                for layer in self.layers:
                    layer.weights -= self.momentum * layer.velocity

                # Backward pass
                gradients = self.backward(y_batch, y_pred, loss_func)

                # Update with Nesterov momentum
                self.nesterov_momentum_update(gradients, learning_rate)

            # Check flag if we want to use adam optimizer
            elif use_adam:
                # Forward pass
                y_pred = self.forward(X_batch)
                # Backward pass
                gradients = self.backward(y_batch, y_pred, loss_func)

                self.update_A(gradients)
```

```
                self.update_F(gradients)

                # Perform adam update
                self.adam_update(learning_rate)

            else:
                # Forward pass
                y_pred = self.forward(X_batch)
                # Backward pass
                gradients = self.backward(y_batch, y_pred, loss_func)
                # Update weights using standard SGD
                for layer, gradient in zip(self.layers, gradients):
                    layer.weights -= learning_rate * gradient



    def update_A(self, gradients_W, rho=0.999):
        for i, gradient in enumerate(gradients_W):
            self.A[i] = rho*self.A[i] + (1 - rho) * (gradient ** 2)

    def update_F(self, gradients_W, rho_f=0.9):
        for i, gradient in enumerate(gradients_W):
            self.F[i] = rho_f * self.F[i] + (1-rho_f) * gradient

    def adam_update(self, learning_rate, rho=0.999, rho_f=0.9, epsilon=1e-8):
        for i, layer in enumerate(self.layers):

            A_hat_i = self.A[i] / (1 - (rho ** self.t))
            F_hat_i = self.F[i] / (1 - (rho_f ** self.t))

            # Calculate alpha_t for the current time step
            alpha_t = learning_rate * ((np.sqrt(1 - (rho ** self.t))) / (1 - (rho_f ** self.t)))

            # Calculate the adaptive step
            adaptive_step = alpha_t * F_hat_i / (np.sqrt(A_hat_i) + epsilon)

            # Update weights
            layer.weights -= adaptive_step


    # New method for Nesterov momentum update
    def nesterov_momentum_update(self, gradients_W, learning_rate):
        for layer, grad_W in zip(self.layers, gradients_W):
            # Update velocity
            layer.velocity = self.momentum * layer.velocity - learning_rate * grad_W
            # Update weights using Nesterov momentum
            layer.weights += layer.velocity

    # Train the network using forward and backward propagation
    def train(self, X, y, learning_rate, epochs, use_adam):
        print(f"Using adam optimizer ..." if use_adam else "")
        print(f"Using Nesterov momentum ..." if self.use_nesterov else "")

        for _ in range(epochs):
            y_pred = self.forward(X)
            gradients_W = self.backward(y,y_pred)


            self.gd(gradients_W, learning_rate)

    def train_adam(self, X, y, learning_rate, epochs):
        for _ in range(epochs):
            y_pred = self.forward(X)
            gradients_W = self.backward(y, y_pred)
            self.update_A(gradients_W)
```

```
            self.update_F(gradients_W)
            self.adam_update(learning_rate)
            self.t += 1

    def train_adam_batch(self, X, y, learning_rate, epochs, batch_size, loss_func):
        for epoch in range(epochs):
            indices = np.arange(X.shape[0])
            np.random.shuffle(indices)

            for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
                batch_indices = indices[start_idx:start_idx + batch_size]
                X_batch = X[batch_indices]
                y_batch = y[batch_indices]

                # Forward pass
                y_pred = self.forward(X_batch)

                # Backward pass
                gradients_W = self.backward(y_batch, y_pred, loss_func)

                # Update moments and weights using Adam optimizer
                self.update_A(gradients_W)
                self.update_F(gradients_W)
                self.adam_update(learning_rate)
                self.t += 1

            # Optionally, print loss and other metrics for monitoring
            y_pred_full = self.forward(X)
            #loss = self._calculate_loss(y, y_pred_full, loss_func)
            #print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss}")

    # Train the network using stochastic gradient descent
    def trainsgd(self, X, y, learning_rate, epochs, batch_size, use_adam, loss_func='mse'):
        print(f"Using adam optimizer for SGD ..." if use_adam else "")
        print(f"Using Nesterov momentum for SGD ..." if self.use_nesterov else "")

        for epoch in range(epochs):
            self.sgd(X, y, batch_size, learning_rate, loss_func)

            # Calculate and print loss for monitoring
            y_pred = self.forward(X)
            loss = self._calculate_loss(y, y_pred, loss_func)
            print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss}")

    def _calculate_loss(self, y, y_pred, loss_func):
        if loss_func == 'mse':
            return np.mean((y_pred - y) ** 2)
        elif loss_func == 'nll':
            # Clip predictions to prevent log(0)
            y_pred = np.clip(y_pred, 1e-10, 1.0)
            return -np.mean(np.sum(y * np.log(y_pred), axis=1))
        else:
            raise ValueError("Unsupported loss function")

    def hessian_matrix(self, X, y):
        y_pred = self.forward(X)

        # Flatten weights across all layers
        num_total_weights = sum(layer.weights.size for layer in self.layers)
        full_hessian = np.zeros((num_total_weights, num_total_weights))

        # Track the starting index for each layer
        start_idx = 0

        for layer_idx, layer in enumerate(self.layers):
```

```
            layer_size = layer.weights.size

            # calculate the Hessian entry for each pair of weights in the layer
            for i in range(layer_size):
                for j in range(layer_size):
                    hessian_entry = layer.hessian_entry(i, j, y_pred, y)
                    full_hessian[start_idx + i, start_idx + j] = hessian_entry

            start_idx += layer_size

        return full_hessian

    def newton_update(self, X, y, learning_rate=0.01, loss_func='mse', reg_lambda=1e-4):
        y_pred = self.forward(X)
        gradients = self.backward(y, y_pred, loss_func)
        gradient_vector = np.concatenate([grad.flatten() for grad in gradients])

        # calculate hessian matrix
        full_hessian = self.hessian_matrix(X, y)
        #Regularize
        full_hessian += reg_lambda * np.eye(full_hessian.shape[0])

        # pseudoinverse of the Hessian
        hessian_pinv = np.linalg.pinv(full_hessian)

        # Newton update
        update_step = -learning_rate * np.dot(hessian_pinv, gradient_vector)

        # update weights
        start_idx = 0
        for layer in self.layers:
            layer_size = layer.weights.size
            layer_update = update_step[start_idx:start_idx + layer_size].reshape(layer.weights.shape)
            layer.weights += layer_update
            start_idx += layer_size
```

Listing 1: `fnn.py`

```
import random

import numpy as np
from scipy.special import logsumexp


class Layer:
    """
    A layer in the Feedforward Neural Network (FNN).
    """

    # Randomly initialize weights and biases
    def __init__(self, n_input, n_output, init_type, activation):
        random.seed(2400)
        if init_type == 'xavier':
            self.weights = self._xavier_init(n_input, n_output)
        else:
            self.weights = np.random.uniform(-1, 1, (n_input+1, n_output))*.5
        self.activation_function = activation
        self.n_input = n_input
        #self.gradient_W = None

    def _xavier_init(self, n_input, n_output):
        """
        Helper method for Xavier initialization with uniform distribution.
        """
```

```
27          limit = np.sqrt(0.5 / (n_input + n_output))
28          return np.random.uniform(-limit, limit, (n_input+1, n_output))
29
30      def forward(self, X):
31          X = np.hstack([X, np.ones((X.shape[0], 1))])
32
33          self.z = np.dot(X, self.weights)
34          self.a = self.activate(self.z)
35          #print(
36          #    f"Layer forward pass - Activation {self.activation_function} mean: {np.mean(self.a)}, "
37          #    f"std: {np.std(self.a)}, min: {np.min(self.a)}, max: {np.max(self.a)}")
38          self.input_data = X
39
40          return self.a
41
42      def logsoftmax(self, z):
43          return z - np.log(np.sum(np.exp(z - np.max(z, axis=1, keepdims=True)), axis=1, keepdims=True)
                + 1e-8)
44
45      # Activation functions
46      def activate(self, z):
47          activations = {
48              'relu': lambda z: np.maximum(0, z),
49              'sigmoid': lambda z: 1 / (1 + np.exp(-np.clip(z, -100, 100))),
50              'id': lambda z: z,
51              'sign': lambda z: np.sign(z),
52              'tanh': lambda z: np.tanh(z),
53              'hard tanh': lambda z: np.clip(z, -1, 1),
54              'logsoftmax': lambda z: self.logsoftmax(z),
55              'leaky_tanh': lambda z: np.where(z > 0, np.tanh(z), 0.01 * z),
56              'softplus': lambda z: np.where(z > 20, z, np.log1p(np.exp(z)))
57          }
58
59          return activations[self.activation_function](z)
60
61      # Derivatives of activation functions
62      """
63      If an error arises using the 'sign' activation function, it is because the derivative is
            undefined at z = 0. (Will return NaN)
64      """
65      def activation_deriv(self, z):
66          derivs = {
67              'relu': lambda z: np.where(z > 0, 1, 0),
68              'sigmoid': lambda z: (sig := 1 / (1 + np.exp(-np.clip(z, -100, 100)))) * (1 - sig),
69              'id': lambda _: np.ones_like(z),
70              'sign': lambda z: np.zeros_like(z),  # Derivative undefined at z = 0
71              'tanh': lambda z: 1 - np.tanh(z) ** 2,
72              'hard tanh': lambda z: np.where(np.abs(z) <= 1, 1, 0),
73              'logsoftmax': lambda z: self._logsoftmax_derivative(z),
74              'leaky_tanh': lambda z: np.where(z > 0, 1 - np.tanh(z) ** 2, 0.01),
75              'softplus': lambda z: 1 / (1 + np.exp(-z))
76          }
77
78          return derivs[self.activation_function](z)
79
80      def _logsoftmax_derivative(self, z):
81          # Reshape z to ensure it is at least 2D for consistent axis handling
82          if z.ndim == 0:
83              z = z.reshape(1, 1)
84          elif z.ndim == 1:
85              z = z.reshape(1, -1)
86
87          # Compute log-softmax and its derivative
88          softmax = np.exp(z - np.max(z, axis=1, keepdims=True)) / np.sum(np.exp(z - np.max(z, axis=1,
                keepdims=True)),
```

```
                                                                    axis=1, keepdims=True)
        return softmax * (1 - softmax) + 1e-5

    def backward(self, dL_dout):
        dL_dout = np.nan_to_num(dL_dout)
        activation_deriv = self.activation_deriv(self.z)
        dL_dout *= activation_deriv
        #print(f"Backward dL_dout: mean {np.mean(dL_dout)}, min {np.min(dL_dout)}, max {np.max(
            dL_dout)}")
        # partial derivative of the loss w.r.t. the weights
        grad_W = np.dot(self.input_data.T, dL_dout)

        #print(
        #    f"Layer forward pass - Activation {self.activation_function} mean: {np.mean(self.a)}, "
        #    f"std: {np.std(self.a)}, min: {np.min(self.a)}, max: {np.max(self.a)}")
        # accumulation of partial derivative of the loss for each layer
        dL_din = np.dot(dL_dout, self.weights.T)

        # Remove the bias
        dL_din = dL_din[:, :-1]

        grad_W = np.clip(grad_W, -3, 3)
        #self.gradient_W = grad_W

        return grad_W, dL_din

    def activation_second_deriv(self, z):
        """
        Compute the second derivative of the activation function.
        """
        second_derivs = {
            'relu': lambda z: np.zeros_like(z),
            'sigmoid': lambda z: (sig := 1 / (1 + np.exp(-np.clip(z, -100, 100)))) * (1 - sig) * (1 -
                2 * sig),
            'id': lambda _: np.zeros_like(z),
            'tanh': lambda z: -2 * np.tanh(z) * (1 - np.tanh(z) ** 2),
            'leaky_tanh': lambda z: np.where(z > 0, -2 * np.tanh(z) * (1 - np.tanh(z) ** 2), 0),
            'softplus': lambda z: np.exp(-z) / ((1 + np.exp(-z)) ** 2)
        }
        return second_derivs.get(self.activation_function, lambda z: np.zeros_like(z))(z)

    def hessian_diagonal(self, dL_dout):
        dL_dout = np.nan_to_num(dL_dout)
        # Compute the first and second derivatives of the activation function
        activation_deriv = self.activation_deriv(self.z)
        second_activation_deriv = self.activation_second_deriv(self.z)

        # Calculate the diagonal of the Hessian with respect to weights in this layer
        # hessian_diag = np.sum((self.input_data ** 2) * (dL_dout * second_activation_deriv), axis=0)
        hessian_diag = np.sum((self.input_data ** 2).T @ (dL_dout * second_activation_deriv), axis=0)

        # Calculate dL_din for second-order backpropagation
        dL_din = np.dot(dL_dout * activation_deriv, self.weights.T)
        second_order_term = np.dot(dL_dout * second_activation_deriv, self.weights.T)
        dL_din += second_order_term

        dL_din = dL_din[:, :-1]

        return hessian_diag, dL_din


    def hessian_entry(self, i, j, y_pred, y):
        # inputs for weights w_i and w_j
        input_i = self.input_data.flatten()[i]
        input_j = self.input_data.flatten()[j]
```

```
152
153         # Calculate z
154         local_z = input_i * self.weights.flatten()[i] + input_j * self.weights.flatten()[j]
155
156         activation_deriv = self.activation_deriv(local_z)
157         activation_second_deriv = self.activation_second_deriv(local_z)
158
159         term1 = activation_second_deriv * input_i * input_j * (y_pred - y)
160         term2 = activation_deriv ** 2 * input_i * input_j
161         # Hessian for (i,j)
162         return np.sum(term1 + term2)
```

Listing 2: `layer.py`

```
1  import numpy as np
2  import torch
3  import random
4  from torch.utils.data import DataLoader, Subset
5  from torchvision import transforms
6  from torchvision.datasets import MNIST
7  import matplotlib.pyplot as plt
8  from layer import Layer
9  from fnn import FNN
10 from sklearn.model_selection import train_test_split
11 import time
12
13 # Set random seed for reproducibility
14 random_seed = 24
15 np.random.seed(random_seed)
16 torch.manual_seed(random_seed)
17 random.seed(random_seed)
18
19 def get_data_loader(is_train, subset_size=1000, downsample_size=14):
20     # Transform: Resize and convert to tensor
21     to_tensor = transforms.Compose([
22         transforms.Resize((downsample_size, downsample_size)),  # Resize to smaller image
23         transforms.ToTensor()
24     ])
25     data_set = MNIST("", is_train, transform=to_tensor, download=True)
26
27     # Reduce to a subset if specified
28     if subset_size and is_train:
29         indices = list(range(len(data_set)))
30         subset_indices, _ = train_test_split(indices, train_size=subset_size, stratify=[data_set[i
               ][1] for i in indices], random_state=random_seed)
31         data_set = Subset(data_set, subset_indices)
32
33     return DataLoader(data_set, batch_size=subset_size, shuffle=True)
34
35 def evaluate(test_data, net):
36     n_correct = 0
37     n_total = 0
38     for batch_X, batch_y in test_data:
39         batch_X = batch_X.view(batch_X.shape[0], -1).numpy()
40         batch_y = batch_y.numpy()
41
42         outputs = net.forward(batch_X)
43         predicted = np.argmax(outputs, axis=1)
44         n_correct += (predicted == batch_y).sum()
45         n_total += batch_y.shape[0]
46
47     return n_correct / n_total
48
49 # Create network with case4's Layer and FNN
```

```
input_size = 14 * 14  # Reduced dimensionality due to downsampling
hidden_size = 10
output_size = 10

# Define layers
layer1 = Layer(input_size, hidden_size, init_type= 'uniform', activation='relu')
layer2 = Layer(hidden_size, hidden_size, init_type= 'uniform', activation='relu')
layer3 = Layer(hidden_size, hidden_size, init_type= 'uniform', activation='relu')
layer4 = Layer(hidden_size, output_size, init_type= 'uniform', activation='logsoftmax')

# Initialize FNN with the modified architecture
net = FNN(layers=[layer1, layer2, layer3])

# Get data loaders
train_data = get_data_loader(is_train=True, subset_size=1000, downsample_size=14)  # Smaller input and subset
test_data = get_data_loader(is_train=False, downsample_size=14)

# Prepare training dataset as a single array
X_train = []
y_train = []
for batch_X, batch_y in train_data:
    X_train.append(batch_X.view(batch_X.shape[0], -1).numpy())
    y_train.append(batch_y.numpy())

X_train = np.concatenate(X_train)
y_train = np.concatenate(y_train)

# Convert labels to one-hot encoding
y_train_one_hot = np.zeros((y_train.size, output_size))
y_train_one_hot[np.arange(y_train.size), y_train] = 1

# Initial accuracy
initial_accuracy = evaluate(test_data, net)
print("Initial accuracy:", initial_accuracy)

# Training parameters
learning_rate = 0.001
epochs = 100
reg_lambda = 1e-2

# Start the timer
start_time = time.time()

# Train using Newton's update from case4
loss_func = 'nll'
use_adam = True
batch_size = 25
method = "diagonal_approx"
for epoch in range(epochs):

    if method == "diagonal_approx":
        net.newton_update_diagonal_approx(X_train,
                                          y_train_one_hot,
                                          learning_rate=learning_rate,
                                          loss_func=loss_func)
    elif method == "newton_update":
        net.newton_update(X_train,
                          y_train_one_hot,
                          learning_rate=learning_rate,
                          loss_func=loss_func,
                          reg_lambda=reg_lambda)

    # Evaluate accuracy after each epoch
    accuracy = evaluate(test_data, net)
```

```
    print(f"Epoch {epoch + 1}/{epochs}, Accuracy: {accuracy:.4f}")

# End the timer
end_time = time.time()
training_duration = end_time - start_time
print(f"Total training time: {training_duration:.2f} seconds")

# Visualize some predictions
for n, (x, _) in enumerate(test_data):
    if n > 5:
        break

    x_flat = x[0].view(-1).numpy()
    pred = np.argmax(net.forward(x_flat.reshape(1, -1)))

    plt.figure(n)
    plt.imshow(x[0].view(14, 14), cmap='gray')
    plt.title(f"Prediction: {pred}")

plt.show()
```

Listing 3: `case3hessian.py`

```
import numpy as np
import time
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from fnn import FNN
from layer import Layer

# Set random seed for reproducibility
random_seed = 24
np.random.seed(random_seed)

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# One-hot encode the target labels
num_classes = len(np.unique(y))
y_onehot = np.zeros((y.shape[0], num_classes))
for i, label in enumerate(y):
    y_onehot[i, label] = 1

# Verify the one-hot encoding
print("Original labels:", y[:5])
print("One-hot encoded labels:\n", y_onehot[:5])

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.1, random_state=
    random_seed)

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Network configuration
hidden_layer = 10
layers = [
```

```
    Layer(n_input=4, n_output=hidden_layer, init_type='xavier', activation='tanh'),
    Layer(n_input=hidden_layer, n_output=hidden_layer, init_type='xavier', activation='tanh'),
    Layer(n_input=hidden_layer, n_output=3, init_type='xavier', activation='logsoftmax')
]
nn = FNN(layers)

# Training parameters
epochs = 25
learning_rate = 0.001
reg_lambda = 1e-4
method = "diagonal_approx"

# Start the timer
start_time = time.time()

# Train using Newton's method
for epoch in range(epochs):
    print(f"\nEpoch {epoch + 1}/{epochs}")

    if method == "diagonal_approx":
        nn.newton_update_diagonal_approx(X_train,
                                         y_train,
                                         learning_rate=learning_rate,
                                         loss_func="nll")
    elif method == "newton_update":
        nn.newton_update(X_train,
                         y_train,
                         learning_rate=learning_rate,
                         loss_func="nll",
                         reg_lambda=reg_lambda)

    # Forward pass to get predictions on training set
    y_train_pred = nn.forward(X_train)
    train_loss = nn._calculate_loss(y_train, y_train_pred, loss_func='nll')
    train_accuracy = accuracy_score(y_train.argmax(axis=1), y_train_pred.argmax(axis=1))

    # Print training statistics
    print(f"Epoch {epoch + 1}/{epochs}")
    print(
        f"y_train_pred mean: {np.mean(y_train_pred)}, std: {np.std(y_train_pred)}, min: {np.min(
            y_train_pred)}, max: {np.max(y_train_pred)}")
    print(f"train_loss: {train_loss}, train_accuracy: {train_accuracy}")

    # Gradient inspection
    gradients_W = nn.backward(y_train, y_train_pred, loss_func='nll')
    for i, grad_W in enumerate(gradients_W):
        print(
            f"Layer {i + 1} Gradient Mean: {np.mean(grad_W)}, Std: {np.std(grad_W)}, Range: {np.min(
                grad_W)} to {np.max(grad_W)}")

# End the timer
end_time = time.time()
training_duration = end_time - start_time
print(f"Total training time: {training_duration:.2f} seconds")

# Test the network
y_test_pred = nn.forward(X_test)
test_loss = nn._calculate_loss(y_test, y_test_pred, loss_func='nll')
test_accuracy = accuracy_score(y_test.argmax(axis=1), y_test_pred.argmax(axis=1))

print(f"Test Loss: {test_loss}, Test Accuracy: {test_accuracy * 100:.2f}%")
```

Listing 4: `case4.py`