Matthew Morris
Professor Kontothanassis
DS210
25 April 2025

DS210 HW9 Journal

```
use csv::ReaderBuilder;
use ndarray::{Array, Array2};
use ndarray_rand::rand_distr::Uniform;
use ndarray_rand::RandomExt;
use serde::Deserialize;
use std::{error::Error, fs::File};
```

Importing csv readerbuilder from csv crate in order to create a csv parser for reading the MNIST data files. Also imported array (general n-dimensional array) and array 2 (2d array) from ndarray. These are used for storing network weights, inputs, and performing linear algebra operations. Imports the Uniform distribution type from the ndarray_rand crate (which bridges ndarray and the rand crate). This specific distribution is used to generate random numbers uniformly within a given range, needed for initializing the network's weights randomly but within controlled bounds. Imports the RandomExt trait from ndarray_rand. This trait extends ndarray's array types with methods like .random(), providing a convenient way to create arrays filled with random values drawn from a specified distribution (like the Uniform distribution imported above).Imports the Deserialize trait from the serde crate (Serde stands for SERialization/DEserialization). This trait allows Rust data structures (like the MnistData struct defined later) to be automatically created by parsing data from formats like CSV. The csv crate leverages serde for this purpose.error::Error: Imports the standard Error trait. This is a common way to represent any kind of error in Rust. Box<dyn Error> is often used as a general-purpose error type that can hold any type implementing the Error trait. fs::File: Imports the File type from the standard library's file system (fs) module. This type is used to represent and interact with files on the operating system, specifically for opening the MNIST CSV files.

```
struct NeuralNetwork {
   learning_rate: f32,
   weights_input_to_layer1: Array2<f32>,
   weights_layer1_to_layer2: Array2<f32>,
   weights_layer2_to_output: Array2<f32>,
}
```

Created a struct called neuralnetwork which is the main data structure representing the neural network itself. It contains 3 arrays and an f32 type. They are for each of the transformation matrices as well as the learning rate of the neural network, which I adjust to tune the model.

```
#[derive(Debug, Deserialize)]
struct MnistData {
```

```
    label: u8,
    pixels: Vec<u8>,
}
```

Struct defines the format for representing a single row read from the MNIST CSV field. Additionally enables the csv crate's reader (serde) to automatically parse a csv row and populate the field. Additionally creates a struct with label and pixels. The pixel will store intensity values (0-255 rgb), even though it should only be 0 or 1 for black or white. I decided to do so just in case there were other cases within the training set that were outliers, just to be safe.

```
fn load_mnist_data(file_path: &str) -> Result<(Array2<f32>, Array2<f32>), Box<dyn
Error>> {
    let file = File::open(file_path)?;
    let mut rdr = ReaderBuilder::new().has_headers(false).from_reader(file);

    let mut features = Vec::new();
    let mut labels = Vec::new();

    for result in rdr.deserialize() {
        let record: MnistData = result?;

        let mut one_hot = vec![0.0; 10];
        one_hot[record.label as usize] = 1.0;
        labels.push(one_hot);

        let normalized_pixels: Vec<f32> = record.pixels.iter().map(|&x| x as f32 /
255.0).collect();
        features.push(normalized_pixels);
    }

    let features_array = Array2::from_shape_vec((features.len(), 784),
features.concat())?;
    let labels_array = Array2::from_shape_vec((labels.len(), 10), labels.concat())?;

    Ok((features_array, labels_array))
}
```

This function handles the reading of a specified file, transforming the contents into ndarray matrices for the neural network. First opens the file, then initializes features and labels as new vectors and iterates through rdr. Used deserialize() iterator. This is a result<t> where T is the type being deserialized into.

I create a one-hot vector which is essentially a normalized vector. I created it with 10 elements each initialized with the value 0. The vector will represent the label in one-hot encoding scheme.

```
    one_hot[record.label as usize] = 1.0;
       labels.push(one_hot);
```

Appends the newly created one-hot vector to the end of the labels vector.

```
let normalized_pixels: Vec<f32> = record.pixels.iter().map(|&x| x as f32 /
255.0).collect();
       features.push(normalized_pixels);
   }
```

Access pixel vectors, creating an interator over the elements in the pixels vector. Then applies a transformation to each element yielded by the vector.

```
 let features_array = Array2::from_shape_vec((features.len(), 784),
features.concat())?;
   let labels_array = Array2::from_shape_vec((labels.len(), 10), labels.concat())?;

   Ok((features_array, labels_array))
```

Converts the collected feature data a vec<vec<f32>> into a single 2d ndarray matrix. Converts the collected one-hot encoded label data into a single 2d nd array matrix. If the previous steps were successful, the line wraps the created features arrays and labels array matrices into a tuple and then wraps the tuple in the ok variant of the result enum.

```
fn sigmoid(x: f32) -> f32 {
   1.0 / (1.0 + (-x).exp())
}
```

Implements sigmoid component

```
fn sigmoid_derivative(x: f32) -> f32 {
   x * (1.0 - x)
}
```

Manually took derivative of sigmoid.

```
impl NeuralNetwork {
   fn new(
       input_size: usize,
       layer1_size: usize,
       layer2_size: usize,
       output_size: usize,
       learning_rate: f32,
   ) -> Self {
       let scale1 = (1.0 / (input_size + layer1_size) as f32).sqrt();
       let scale2 = (1.0 / (layer1_size + layer2_size) as f32).sqrt();
       let scale3 = (1.0 / (layer2_size + output_size) as f32).sqrt();
```

```
        let weights_input_to_layer1 =
            Array::random((input_size, layer1_size), Uniform::new(-scale1, scale1));
        let weights_layer1_to_layer2 =
            Array::random((layer1_size, layer2_size), Uniform::new(-scale2, scale2));
        let weights_layer2_to_output =
            Array::random((layer2_size, output_size), Uniform::new(-scale3, scale3));

        NeuralNetwork {
            learning_rate,
            weights_input_to_layer1,
            weights_layer1_to_layer2,
            weights_layer2_to_output,
        }
    }
```

Block defines methods for the neural network struct. First defines new function defining input size, layer1 size, layer2 size, output size and learning rate, which are all changeable for tuning the model. Then self caclulates a scaling factor for intializing the weights between the input and the first hidden layer. This then calculates the xavier factor for the weights. I do this for each individual layer.

Neural network {} creates and returns a new instance of the neural network struct. It uses literal syntax.

```
fn forward(&self, input: &Array2<f32>) -> (Array2<f32>, Array2<f32>, Array2<f32>) {
        let z1 = input.dot(&self.weights_input_to_layer1);
        let a1 = z1.mapv(sigmoid);

        let z2 = a1.dot(&self.weights_layer1_to_layer2);
        let a2 = z2.mapv(sigmoid);

        let z3 = a2.dot(&self.weights_layer2_to_output);
        let a3 = z3.mapv(sigmoid);

        (a1, a2, a3)
    }
```

This is the code for forward propagation. First it takes the dot product between the input matrix and the weight matrix connecting the input to layer 1. It then uses mapv, mapping and applying the sigmoid function to every value in the z1 matrix.
Then the sigmoid outputs are mapped for a specific digit (0-9).

```
fn backward(
```

```
        &mut self,
        input: &Array2<f32>,
        layer1_output: &Array2<f32>,
        layer2_output: &Array2<f32>,
        final_output: &Array2<f32>,
        target: &Array2<f32>,
    ) {
        let error = target - final_output;

        let delta3 = error * final_output.mapv(sigmoid_derivative);

        let delta2 =
            delta3.dot(&self.weights_layer2_to_output.t()) *
layer2_output.mapv(sigmoid_derivative);

        let delta1 =
            delta2.dot(&self.weights_layer1_to_layer2.t()) *
layer1_output.mapv(sigmoid_derivative);

        self.weights_layer2_to_output += &(self.learning_rate *
layer2_output.t().dot(&delta3));
        self.weights_layer1_to_layer2 += &(self.learning_rate *
layer1_output.t().dot(&delta2));
        self.weights_input_to_layer1 += &(self.learning_rate * input.t().dot(&delta1));
    }
}
```

This is the backwards propagation method. First, it takes a mutable reference to the neural network instance, then it takes an input of two array2's, which allows for the two hidden layers. Then, it has a final_output 2d array, resulting in an activation of the output layer (a3 from the forward pass, the network's prediction). Then it takes target, which is a reference to the target outpt matrix. This method doesn't return a proper value, rather it is designed to adjust the weights in place.

Then I calculate the error of the output layer which is an element iwse difference between the desired output and the networks actual output. Then for delta3, I calculate the error signal for the output layer.

I then calculate the delta2, which is the dot product of delta3, and weights_layer_2_to output * the sigmooid derivative's value map of layer 2 output.

From there, I adjust the weights for layer 2 and 1 and delta 1, by calculating their respective dot products as well. This part is relatively self explanatory, but was given as a part of the structured code.

```
fn main() -> Result<(), Box<dyn Error>> {
   let (train_features, train_labels) =
load_mnist_data("../MNIST_CSV/mnist_train.csv")?;
   let (test_features, test_labels) = load_mnist_data("../MNIST_CSV/mnist_test.csv")?;

   let mut network = NeuralNetwork::new(
       784, //input size
       512, //hiddenlayer1 size
       256, //hidden layer2 size
       10,  //output size
       0.2, //learning rate
   );
```

This part of the main function loads the mnist data and then I pull the neuralnetwork array, and I assign values to the vector values we created earlier.

```
for epoch in 0..3 {
       let mut correct = 0;
       let total = train_features.nrows();

       for i in 0..total {
           let input = train_features.row(i).into_owned().into_shape((1, 784))?;
           let target = train_labels.row(i).into_owned().into_shape((1, 10))?;

           let (layer1_output, layer2_output, final_output) = network.forward(&input);

           network.backward(
               &input,
               &layer1_output,
               &layer2_output,
               &final_output,
               &target,
           );
```

I create 3 epochs, which essentially iterate a fixed number of times over the entire training dataset. I then get the total number of rows in the training feature matrix train_features using the .nrows() method. I store it in the total variable.
I then start the inner loop, iterating through each image individually.

Then, I prepare the input data for the current training example. Select the i-th row form the train_features matrix, then I convert the view into owned array1. Then I attempt to reshape the 1d array into a 2d array. The ? acts as a try function.

I then perform the forward pass through the network using the current input. Network.forward(&input) calls the forward method o nthe network instance, passing a reference to the input matrix. Then , I destructre the tuple returned by forward into three variables holding the activatinos of layer 1, layer 2, and the output layer, respectively.

I then perform the backward pass to update the network weights.

```
    let predicted = final_output
                .row(0)
                .iter()
                .enumerate()
                .max_by(|(_, a), (_, b)| a.partial_cmp(b).unwrap())
                .map(|(i, _)| i)
                .unwrap();
            let actual = target
                .row(0)
                .iter()
                .enumerate()
                .max_by(|(_, a), (_, b)| a.partial_cmp(b).unwrap())
                .map(|(i, _)| i)
                .unwrap();

            if predicted == actual {
                correct += 1;
            }
        }
```

I then determine the network’s predicted digit for the current sample. I then determine the actual correct digit from the one-hot encoded target vector using the exact same logic as finding the predicted digit. The max value since it is one-hot encoded will be 1, and the index should correspond to the correct digit.

Then I compare the networks prediction with the actual correct label, then if it is correct it will increment the counter for the epoch.

```
let epoch_accuracy = correct as f32 / total as f32 * 100.0;
      println!(
          "Epoch {} training accuracy: {:.2}%",
          epoch + 1,
```

```
        epoch_accuracy
    );
}
```

Prints the training accuracy for the current epoch.

```
let mut correct = 0;
for i in 0..test_features.nrows() {
    let input = test_features.row(i).into_owned().into_shape((1, 784))?;
    let target = test_labels.row(i).into_owned().into_shape((1, 10))?;

    let (_, _, final_output) = network.forward(&input);

    let predicted = final_output
        .row(0)
        .iter()
        .enumerate()
        .max_by(|(_, a), (_, b)| a.partial_cmp(b).unwrap())
        .map(|(i, _)| i)
        .unwrap();
    let actual = target
        .row(0)
        .iter()
        .enumerate()
        .max_by(|(_, a), (_, b)| a.partial_cmp(b).unwrap())
        .map(|(i, _)| i)
        .unwrap();

    if predicted == actual {
        correct += 1;
    }
}
```

This code performs a forward pass usign the trained network on the current test input. Only the final_output is needed for evaluation so the intermediate layer outputs are ignored using _. Network.backward is not used here because we are only evaluating the network’s performance.

```
let accuracy = (correct as f32) / (test_features.nrows() as f32) * 100.0;
  println!("Final test accuracy: {:.2}%", accuracy);

  Ok(())
}
```

Calculates the accuracy.

Output:

Epoch 1 training accuracy: 79.18%
Epoch 2 training accuracy: 94.35%
Epoch 3 training accuracy: 96.06%
Final test accuracy: 96.26%

LLM Queries

Neural Network Initialization

explain he intialization

Sure — in the context of **neural networks**, "**initialization**" usually refers to **how the model's weights are set before training begins**.

At a high level:

- Neural networks learn by adjusting **weights** through training (via backpropagation and optimization like SGD).
- If you start all the weights **as zero** (or the same value), every neuron will learn the same thing — the model won't actually learn useful patterns (**symmetry problem**).
- If you start the weights **too large**, activations can explode.
- If you start them **too small**, gradients can vanish.

Initialization is about **setting the starting weights carefully** to avoid these problems and help the model learn faster and better.

Common Initialization Strategies:

Strategy	Idea	Use Case
Random Uniform	Pick weights randomly from a uniform distribution (e.g., $[-a, a]$)	Simple networks or early prototypes
Random Normal	Pick weights randomly from a normal (Gaussian) distribution	Also simple setups, sometimes for deeper models
Xavier (Glorot) Initialization	Scale weights based on number of input/output nodes: keeps variance of activations steady across layers	Best for activations like **tanh**
He Initialization	Like Xavier but slightly larger weights, tuned for **ReLU** activations	Best for **ReLU** or similar non-linearities

---

In Your Code Example:

You probably saw something like:

```rust
use ndarray_rand::RandomExt;
use ndarray_rand::rand_distr::Uniform;

self.weights_input_to_layer1 = Array::random((input_size, layer1_size),
Uniform::new(-0.5, 0.5));
```

This means:

- Weights are **randomly initialized** between **-0.5 and 0.5**.
- Each connection between input → layer1 gets a different starting weight.

This randomization breaks symmetry and gives the network **a unique starting point** to learn patterns from data.

Summary:

> Initialization gives the network "good starting points" for learning — not too uniform, not too wild — enabling faster and more stable convergence.

Would you like me to show you a visual of why bad initialization can totally mess up training? 🎯

how does xavier initialization work? how can i implement in rust?

Good — you're asking the right question if you want your neural network to **train well**.

Let's break it down **fast and clearly**:

What is Xavier Initialization?

(Xavier = Glorot, same thing)

- It **sets the initial weights** based on the **number of inputs** and **number of outputs** of a layer.
- Goal: **Keep the variance of activations and gradients the same across every layer** (no exploding or vanishing).

Formula for Xavier (for uniform distribution):

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Where:

- n_{in} = number of input neurons
- n_{out} = number of output neurons
- $\mathcal{U}$ = Uniform distribution

How to implement Xavier Initialization in Rust?

You can use `ndarray` and `ndarray-rand` like this:

```rust
use ndarray::{Array2};
use ndarray_rand::RandomExt;
use ndarray_rand::rand_distr::Uniform;

/// Xavier Initialization for a layer
fn xavier_init(input_size: usize, output_size: usize) -> Array2<f32> {
    let limit = (6.0 / (input_size as f32 + output_size as f32)).sqrt();
    Array2::random((input_size, output_size), Uniform::new(-limit, limit))
}
```

✅ This correctly:

- Calculates the right range `[-limit, limit]`
- Randomly initializes the matrix

Where to use it in your network?

In your `NeuralNetwork::new` (or wherever you create weights), instead of:

```rust
self.weights_input_to_layer1 = Array::random((input_size, layer1_size),
Uniform::new(-0.5, 0.5));
```

You would do:

```rust
self.weights_input_to_layer1 = xavier_init(input_size, layer1_size);
self.weights_layer1_to_layer2 = xavier_init(layer1_size, layer2_size);
self.weights_layer2_to_output = xavier_init(layer2_size, output_size);
```

Why it matters:

Without Xavier, if your inputs are big or small, the signals can explode or die out. With Xavier, signals stay "stable" across layers — much better learning, especially in networks that have a few layers.

Summary in one line:

> Xavier spreads initial weights just right — not too big, not too small — based on the shape of the layer.

Would you like me to also show how to do **He initialization** (which is better if you use **ReLU** activation)? 🚀
(It's very similar but slightly different formula.)

explain how to implement backwards and forwards propogation

Perfect — you're asking exactly what you need to **build a working neural network**.

Let's break it down clearly and fast:

---

1. Forward Propagation

Forward pass = move input → through layers → produce output

At each layer:

- **Linear combination:** $z = W \cdot x + b$
- **Activation function:** $a = \sigma(z)$ (e.g., sigmoid, ReLU)

Example (3-layer network):

```css
Input (x)
  ↓
Layer 1:
  z1 = W1 * x + b1
  a1 = activation(z1)
  ↓
Layer 2:
  z2 = W2 * a1 + b2
  a2 = activation(z2)
  ↓
Layer 3 (Output):
  z3 = W3 * a2 + b3
  y_pred = activation(z3)
```

✅ At the end, you have **y_pred** (the model's output).

---

2. Backward Propagation

Backward pass = calculate errors → update weights to reduce loss

At each layer:

- **Loss derivative**: How wrong was the output?
- **Chain rule**: How much did each weight contribute to the error?
- **Gradient**: Calculate derivatives wrt each weight
- **Update rule**: Move weights to minimize error (e.g., gradient descent)

Key equations:

- Error at output:

$$\delta^{(output)} = (y_{\text{pred}} - y_{\text{true}}) \times \sigma'(z^{(output)})$$

- Error at hidden layers:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \times \sigma'(z^{(l)})$$

- Gradients:

$$\frac{\partial \text{Loss}}{\partial W} = a^{(l-1)} \times (\delta^{(l)})^T$$

✅ Then you **subtract gradients from the weights** (with learning rate).

---

3. How to implement in Rust

✅ **Forward propagation function:**

```rust
fn forward(&self, input: &Array2<f32>) -> (Array2<f32>, Array2<f32>, Array2<f32>) {
    let z1 = input.dot(&self.weights_input_to_layer1); // input -> layer1
    let a1 = z1.mapv(sigmoid);

    let z2 = a1.dot(&self.weights_layer1_to_layer2); // layer1 -> layer2
    let a2 = z2.mapv(sigmoid);

    let z3 = a2.dot(&self.weights_layer2_to_output); // layer2 -> output
    let y_pred = z3.mapv(sigmoid);
```

```
    (a1, a2, y_pred)
}
```

Here, `sigmoid` is your activation function.

✅ Backward propagation function:

```rust
fn backward(
    &mut self,
    input: &Array2<f32>,
    a1: &Array2<f32>,
    a2: &Array2<f32>,
    y_pred: &Array2<f32>,
    y_true: &Array2<f32>,
) {
    // Compute output error
    let delta_output = (y_pred - y_true) * &y_pred.mapv(sigmoid_derivative);

    // Compute error for layer 2
    let delta_layer2 = delta_output.dot(&self.weights_layer2_to_output.t()) *
&a2.mapv(sigmoid_derivative);

    // Compute error for layer 1
    let delta_layer1 = delta_layer2.dot(&self.weights_layer1_to_layer2.t()) *
&a1.mapv(sigmoid_derivative);

    // Gradients for weights
    let grad_w3 = a2.t().dot(&delta_output);
    let grad_w2 = a1.t().dot(&delta_layer2);
    let grad_w1 = input.t().dot(&delta_layer1);

    // Update weights
    self.weights_layer2_to_output -= &(self.learning_rate * grad_w3);
    self.weights_layer1_to_layer2 -= &(self.learning_rate * grad_w2);
    self.weights_input_to_layer1 -= &(self.learning_rate * grad_w1);
}
```

✅ **Training loop**:

```rust
for epoch in 0..num_epochs {
    let (a1, a2, y_pred) = network.forward(&inputs);
    network.backward(&inputs, &a1, &a2, &y_pred, &labels);
}
```

4. Quick Notes

Topic	Tip
Activation Function	Use `sigmoid` or `ReLU`. But sigmoid is easier to start.
Loss	Use Mean Squared Error (MSE) or Cross Entropy.
Learning Rate	Start with 0.01 or 0.1, adjust if it doesn't converge.
Weight Initialization	Use Xavier (like we discussed earlier).
Momentum (Optional)	Helps with faster convergence if added.

Summary in One Line:

> Forward pass makes predictions; backward pass calculates errors and updates weights to minimize the loss.

Would you like me to show you a full **Rust skeleton** that ties this all together — forward, backward, train function — in one piece? 🚀 It'll be like a mini working neural net!

(5-minute version, very clean.)