Deep learning: Neural Network from scratch
creating a neural network library to solve the mnist dataset

Adrien PELFRESNE - Alexis VAPAILLE

May 30, 2024

Contents

1 Abstract

We have created a deep neural network "library" from scratch, in rust, without the use of any machine learning or deep learning library.
The goal of this project was to solve the mnist dataset of handwritten digits, and to provide performances comparaison between two multi-layer-perceptron, with and without convolution.
Our library let you create your own sequential neural network, with a simple and declarative api and is widely open to extension. This project also include a drawing mode, when you can draw your own digit and see what the model guess.

2 Motivation

Our motivation with this *from scratch* approach was to deepen our understanding of neural networks. Because high level library like keras and the incredible level of abstraction that come with it, let us sometimes forgot the inner machinery. Nevertheless, this underlying work that those library are doing is passionating, and we through it was a shame to be a simple user of those interface.
The only (methematical) helper library that we have used is ndarray which provides an n-dimensional container for general elements and for numerics, as well of providing efficients matrix operations on the gpu.
You can find our project on github, we widely encourage reader to check the source code, as in this report, we won't go into code specific details and focus more on the abstract explanation.
The workspace is split into multiple crates

1. The neural network library, under the crate *nn_lib*

2. The mnist exemple, which is using the *nn_lib* to define the networks and benchmark the dataset, Under the crate mnist

3. the gui code to run a interactive drawing mode is included in the workpsace in the app file.

3 Sequential neural network

The multi-layer perceptron is known as a *Sequential* neural network, meaning that the **output** Y_N of the layer L_N is the **input** X_{N+1} of the next layer L_{N+1}.

$$Y_N = X_{N+1}$$

and conversely

$$X_N = Y_{N-1}$$

with subscript N indentifying layers in sequential order. from the input layer, when we pass our data, through the output layer, data flows in the network, passing through different kind of layers.

3.1 General layer interface

We have defined a general *Layer* interface, to be implemented by all the differents layer we gonna use.
This interface has two main method : **feed_forward** and **propagate_backward**, The *feed forward* method take the input vector X as parameter and return the output vector Y of the layer. and the *propagate backward* method take as parameters the layer output gradient $\frac{\partial C}{\partial Y}$ and return the layer input gradient $\frac{\partial C}{\partial X}$. We will cover the use of these function later, alongside the theorical explanation of these concepts.

3.2 Dense layer

A dense layer, also knowed as *Linear layer*, is a layer where each input neuron is connected to every output neurone, each connexion is called a *weight*, w_{ij} is the weight connecting neurone x_i (input) to neurone y_j (output).

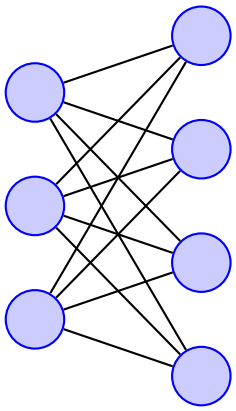

Figure 1: Dense Layer with 3 input nodes and 4 output nodes

We calculate a dense layer neuron output value y_j as

$$y_j = \sum_{i=1}^{n} x_i w_{i,j} + b_j$$

which is the bias corrected weighted sum of all the connected input nodes. b_j is the bias term for the output neuron j. We can rewrite this operation to use matrix operation on whole vectors.

$$Y = XW + B$$

with $Y \in \mathbb{R}^j$, $X \in \mathbb{R}^i$, $W \in M_{i*j}(\mathbb{R})$ and $B \in \mathbb{R}^j$
In this report, we will annotate vector like vector, but note that to be able to do the matmul in practice, a vector $X \in \mathbb{R}^i$ is actually broadcasted to a matrice $M_{1*i}(\mathbb{R})$

3.3 Activation layer

There are two school of thought for defining activations.

1. activations are used to produce the output value **inside** a dense layer, in this case, the weighted sum of the dense layer is called z, and y is calculated by applying an activation σ to z : $y : \sigma(z)$.
2. activations are used **outside** dense layer in this case, the output of a dense layer y is equal to the weighted sum z. An activation Layer is used as a seperate layer, that applie the activation function to each neuron.

We implemented our sequential neural network with the second paradigm, because it is actually easier in code to manage a dense and a activation layer separatly, mainly for backpropagation calculation ease.

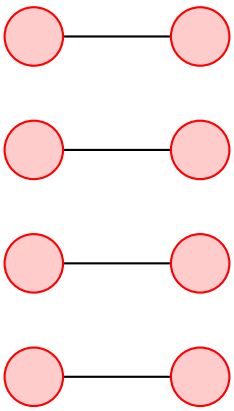

Figure 2: Activation layer with 4 input neuron

The Activation layer take a vector $X \in \mathbb{R}^n$ as input and produce a vector $Y \in \mathbb{R}^n$ as output, with the activation function applied to each neuron.
We must note that some activations function transform vector by mapping the vector individual components x through the function, like ReLU

$$ReLU(x) = max(0, x)$$

While other activation functions, like *softmax* work on whole vectors. Softmax take a vector $z \in \mathbb{R}^K$, convert it into a probability distribution of K possible outcomes.

$$\sigma : \mathbb{R}^K \to (0,1)^K, K \geq 1$$

is computed by taking a vector $z = (z_1, ..., z_k) \in \mathbb{R}^K$ and compute each component of a vector $\sigma(\mathbf{z}) \in (0,1)^K$

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Currents activations function implemented in our library are :

- Sigmoid
- ReLU
- Tanh
- Softmax

4 Cost function

To mesure how well a neural network perform, we use a cost function, which output a real number by comparing the neural network *output*, and the *observed* value.
A simple cost function is the meaned squared error :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The MSE function is great for regression neural network when we want to predict a continuous value from a input. To do classification, like in the mnist data of handwritten digit, we use another cost function, the cross entropy cost function :

$$\text{Cross Entropy} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

with C the number of class (eg: 10 for the mnist dataset), y_c the observed probability, and $\hat{y}_c$ the neural network output.
This expression simpifie as

$$\text{Cross Entropy} = -\log(\hat{y}_c)$$

if y_c is one hot encoded (ie: 1 for the observed class and 0 for others).

5 The learning process

5.1 Gradient

The gradient of a function ∇f is the vector field that represents the direction and rate of the quickest increase of the function from a point p.
We can visualize the gradient and the curve of a function $f(x, y) = -(\cos^2 x + \cos^2 y)^2$

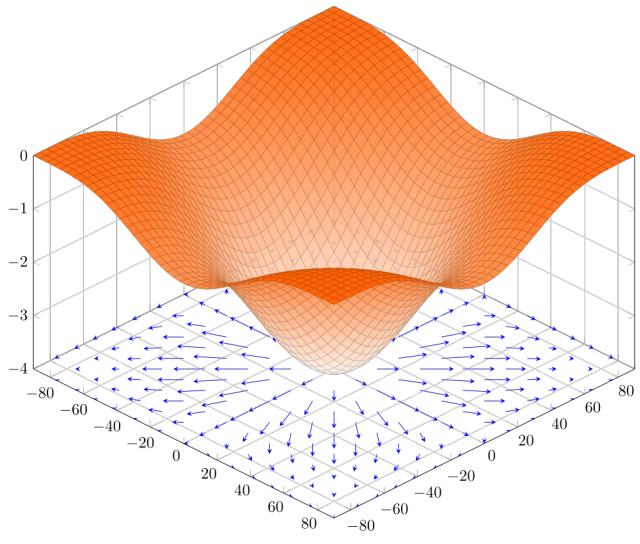

Figure 3: Function representation and gradient field of the function $-(\cos^2 x + \cos^2 y)^2$

The field vector below the function curve represent the gradient, for each point $p = (x, y)$ the arrow at p point toward the direction with the fastest increase of f.

5.2 Minimization

To make our neural network lean, we adjust his parameters (ie: weights and biases for a dense layer) to **minimize the cost function C of the network**. We can use the cost function gradient, which tell us the direction and magnitude of the fastest **increase** of cost, and proceed to go in the opposite direction, which will make us go toward the fastest **decrease** of the cost function.
The algorithm used to minimize the cost function is the *gradient descent*, which will repeatedly descent the curve by going toward the opposite of the gradient of the cost function.

Here is the process for a signle gradient descent step :

1. feed all input data points into the network, calculating the cost for every output.
2. calculate the averaged cost.
3. compute the gradient of the averaged cost function **with respect to the network parameters**.
4. update the parameters in the opposite direction of the gradient

5.3 backpropagation

To do a gradient descent step, we need the gradient of the cost function with respect to the neural network parameters, the algorithm used to calculate this gradient is called **backpropagation**.
Backpropagation precisely calculate (and not only estimate like the finite difference method) the gradient of the cost function C with respect to the neural network parameters, that we will call parameters gradient $\frac{\partial C}{\partial P}$.
For each layer L, we calculate the gradient with respect to the parameters of L, starting from the last layer L_n and going all the way back to the first layer L_0.
The key methematical tool used in backpropagation is the chain rule, which allow the gradient of the cost function to be split into gradient of simpler functions at each neuron within the network.
For each layer L_n, to calculate the parameters gradient $\frac{\partial C}{\partial P_n}$, we need to know the output gradient $\frac{\partial C}{\partial Y_n}$, this gradient is given to us by the next layer in the sequential order, because in the sequential architecture $\frac{\partial C}{\partial X_{n+1}} = \frac{\partial C}{\partial Y_n}$.
We want to also calculate $\frac{\partial C}{\partial X_i}$, the input X gradient, to transmit it to the previous layer, which need it to calculate her own parameters gradient $\frac{\partial C}{\partial W_{i-1}}$.
You can probably feel why this process is called backpropagation, we are sharing gradient, from layer to layer, starting at the last layer and going back up to the start of the net. Each layer implementation will have a specific implementation of the **propagate backward method**, in the next sections, we will cover the calculation for the layer we used in our library.

5.4 Dense layer backpropagation

The dense layer have two trainable parameters, the weigths, W and the biases B. Starting with the gradient of the cost function with respect to the weights, we are given the output gradient :

$$\frac{\partial C}{\partial Y} = \begin{bmatrix} \frac{\partial C}{\partial y_1} \\ \frac{\partial C}{\partial y_2} \\ \vdots \\ \frac{\partial C}{\partial y_j} \end{bmatrix} \tag{1}$$

The cost function C depend on Y which depend on W thus C depend on W (chain rule)

$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial Y}\frac{\partial Y}{\partial W}$$

We calculate each weight gradient component $\frac{\partial C}{\partial W_{ij}}$

$$\frac{\partial C}{\partial W_{ij}} = \sum_{l=1}^{N} \frac{\partial L}{\partial Y_{lj}}\frac{\partial Y_{lj}}{\partial W_{ij}}$$

because W_{ij} is used in every example for calculating the j^{th} column of the output matrix. Let's look at the formula for Y_{lj} to get the derivative

$$Y_{lj} = X_{l1}W_{1j} + \cdots + X_{li}W_{ij} + \cdots + X_{lp}W_{pj} + b_j$$

$$\frac{\partial Y_{lj}}{\partial W_{ij}} = X_{li}$$

$$\implies \frac{\partial C}{\partial W_{ij}} = \sum_{l=1}^{N} \frac{\partial C}{\partial Y_{lj}} X_{li} = \sum_{l=1}^{N} X_{il}^T \frac{\partial C}{\partial Y_{lj}}$$

So finally

$$\frac{\partial C}{\partial W} = X^T \frac{\partial C}{dY}$$

for the biases B.

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial Y}\frac{\partial Y}{\partial b} \tag{2}$$

$$\frac{\partial C}{\partial b_i} = \sum_{l=1}^{N} \frac{\partial C}{\partial Y_{li}}\frac{\partial Y_{li}}{db} \tag{3}$$

since the bias term b_i is used in the evaluation of the entire column of Y.

$$\frac{\partial Y_{li}}{\partial b_i} = 1$$

$$\frac{\partial C}{db_i} = \sum_{l=1}^{N} \frac{\partial C}{\partial Y_{li}}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial Y}$$

and lastly, we calculate the gradient with respect to X

$$\frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y}\frac{\partial Y}{\partial X} \tag{4}$$

$$\frac{\partial C}{\partial X_{ij}} = \sum_{l=1}^{k} \frac{\partial C}{\partial Y_{il}}\frac{\partial Y_{il}}{\partial X_{ij}} \tag{5}$$

since the data point i will only influence the data point i in Y. Other data points will not be affected. Further, X_{ij} is used in calculation of every dimension of $Y_{i,:}$. To calculate the gradient,

$$Y_{il} = \sum_{t=1}^{p} X_{it}W_{tl}$$

$$\frac{\partial Y_{il}}{\partial X_{ij}} = W_{jl}$$

$$\frac{\partial C}{\partial X_{ij}} = \sum_{l=1}^{k} \frac{\partial C}{\partial Y_{il}}W_{jl} = \sum_{l=1}^{k} \frac{\partial C}{\partial Y_{il}}W_{lj}^{T}$$

$$\implies \frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y}W^{T}$$

5.5 Activation layer backpropagation

An activation layer doesn't have any trainable parameters, so we just gonna focus on the input gradient calculation.

we are given the output gradient $\frac{\partial C}{\partial Y}$: we know that

$$Y = \begin{bmatrix} f(X_1) \\ f(X_2) \\ \vdots \\ f(X_i) \end{bmatrix}$$

$$\frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y}\frac{\partial Y}{\partial X}$$

$$\frac{\partial C}{\partial X_{ij}} = \sum_{l=1}^{N} \frac{\partial C}{\partial Y_{lj}}\frac{\partial Y_{lj}}{\partial X_{ij}}$$

using the same procedure as for the previous calculations

$$\implies \frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y} \odot Y\prime$$

with

$$Y\prime = \begin{bmatrix} f\prime(X_1) \\ f\prime(X_2) \\ \vdots \\ f\prime(X_i) \end{bmatrix}$$

6 Optimizers

At this point, to updates a neural network parameter θ, we the gradient descent formula, with a fix learning rate η

$$\theta := \theta - \eta\frac{\partial C}{\partial \theta}$$

This is know, in the keras api for exemple as the *Stochastic gradient descent optimizer*, but it exist other optimizers such as *Adam*.

6.1 Adam

Adam accelerate the convergence towards the optimal set of weights by taking into account passed gradient, and adam will adapt the learning rate for each individual weight.

ADAM initializes two vectors, m and v, which store the exponential moving averages of past gradients and past squared gradients, respectively. These vectors are used to scale the gradient updates adaptatively.

1. **Initialization:** Initialize two vectors, m and v, to store the moving averages of the gradients and their squares, respectively.

2. **Computing Moving Averages of the Gradients:**

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t,$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2,$$

   where g_t is the gradient at time step t, and β_1 and β_2 are factors that control the decay rates.

3. **Bias Correction:** Correct initial bias in m and v:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t},$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

4. **Update Weights:** Adjust weights based on the corrected moments:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t,$$

   where θ are the parameters, η is the learning rate, and ϵ is a small constant for numerical stability.

7 Batching

We went through several feed forward implementations. The first one was to pass data point one by one in the network $x \in \mathbb{R}^i \dots$. This was very slow, for exemple, for the mnist dataset, which has 60k training exemples, we where going through the whole neural network : feed forward, cost evaluation, back propagation, 60k times. But we then leraned that we can instead pass *batch* of data in the neural network, in one time, and this batch can be of any length. the input then becomes $x \in M_{nxi}(\mathbb{R})$ with

each line of the input matrice representing a single data point.

The calculation became much faster, because we can now feed n element at a time in the neural network.

8 Initializer

Initializers determine the starting weights of a neural network, it is important to choose the right initializer, depending on our netwrok architecture, to avoid problem like vanishing and exploding gradient, and to also optimizer convergence.

At the moment we are writing this report, we have imoplemented two initializer inside our library,

1. he The He initializer, designed for ReLu activation functions, initializes weights from a normal distribution centered at zero with a standard deviation of $\sqrt{\frac{2}{n}}$ (where n is the number of input nodes). This setup helps prevent the vanishing gradient problem in deep networks with ReLu activations, ensuring that gradients remain large enough to sustain effective learning through many layers.

2. Glorot uniform Conversely, the GlorotUniform initializer, also known as Xavier Uniform, is ideal for networks using sigmoid or tanh activations. It selects weights from a uniform distribution within $[-c, c]$, where $c = \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}$. This approach maintains a balance in the variance of neurons' outputs across the network, aiding in stable gradient flow in deep networks with saturating activations.

9 Convolutional neural network

ALEXIS a toi de travailler

10 Metrics

With the mnist datset, which is a classification problem, we can implements multiple metrics to evaluate the network performances, we will explain the one we implemented in our library.

1. Accuracy measures the proportion of correct prediction to the total observations in the dataset. for exemple if we have correctly classified 950 image out of 1000, the accuracy will be 95percent.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

   Precision and Recall are two metrics acting on a specified class. **Retrieved elements** are all the network prediction for a specific class. **Relevant elements** are all the correct instances of the class [2].

2. Precision answer the question : *How many retrieved items are relevant*

$$\text{Precision} = \frac{\text{Relevant retreived instance}}{\text{All retrieved instance}}$$

3. Recall answers the question: *How many relevant item are retrieved*

$$\text{Recall} = \frac{\text{Relevant instances}}{\text{All relevants instance}}$$

11 Training, Validation, and Test Sets

To monitor the performances of a neural network effectively, it is important to break down the dataset into three distinct subsets: training, validation, and test sets.

- **Training set** : The training set is used during the leraning phase, with training data directly influencing the parameters updates

- **Validation set** : The validation set is used to provide an unbiased evaluation of a model fit on the training dataset while tuning the model's hyperparameters. This set is used during training, to prevent overfitting when the model performs well on the training data but poorly on unseen data.

- **Test set** : The test set is used only after the model's training and validation phases are complete. It provides the final, unbiased performance metric of the neural network.

12 Results

We include a small code snippet, showing our custom library api to declare a sequential neural network.

```
let net = NeuralNetworkBuilder::new()
  .push(DenseLayer::new(
    28 * 28,
    32,
    InitializerType::He,
  ))
  .push(
    ActivationLayer::from(Activation::ReLU)
  )
  .push(
    DenseLayer::new(32, 10,
    ↪  InitializerType::He)
  )
  .push(
    ActivationLayer::from(
      Activation::Softmax
    )
  );
  .watch(MetricsType::Accuracy)
  .watch(MetricsType::Precision)
  .watch(MetricsType::Recall)
Ok(net.compile(GradientDescent::new(0.01),
↪  CostFunction::CrossEntropy)?)
```

the layer are added in sequential order with the push method, we watched Accuracy, Precision and Recall. the *watch* method will add a *MetricsType* to the list of metrics that the program need to calculate. those metrics can be given over epochs for the training and validation dataset. and for the test dataset.

To create a great network, we will try to reach the *VC trade-off principle* [6] [8], this principle state that:

1. A more complex model might fit the training data very well (low training error) but risks overfitting, meaning it could perform poorly on new data because it is too tailored to the training set (high variance).
2. A less complex model might not fit the training data as closely (higher training error) but can generalize better on new data because it does not capture the noise and specific details of the training set (low variance).

Our challenge will thus be, to find the right level of complexity that minimizes both the error on the training data.

12.1 Multi-layer perceptron

before trying to get a great fitting model, we will anlyse the result of an overfitting model, to see how this behavior translate into metrics. To make a model overfit, we have multiple options

- increase the model complexity (number of hidden layers, number of neuron per layer)
- reduce the batch size to make the learning process very granular and increase the chance of the network leraning to much specific patterns

In our case we have choosen to lower the batch size, here is our network hyperparameters

hyperparameters	epochs	batch size
Value	30	5

Table 1: hyperparameters overfitting model

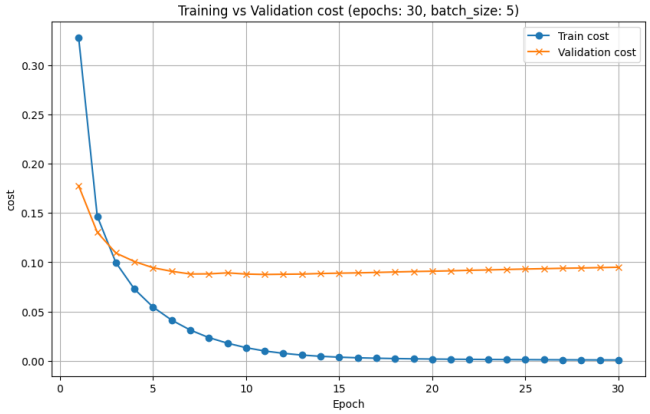

Figure 4: Training vs Validation cost overfitting model

In the cost comparaison between our training data and our validation data, the overfitting clearly stand out, our train cost (or loss) get very low, while the validation cost remain stable this mean that our neural network has learn to much specific pattern in the training data, making the error going very low on this set, but on the validation set, the network doesn't achieve a great generalization.

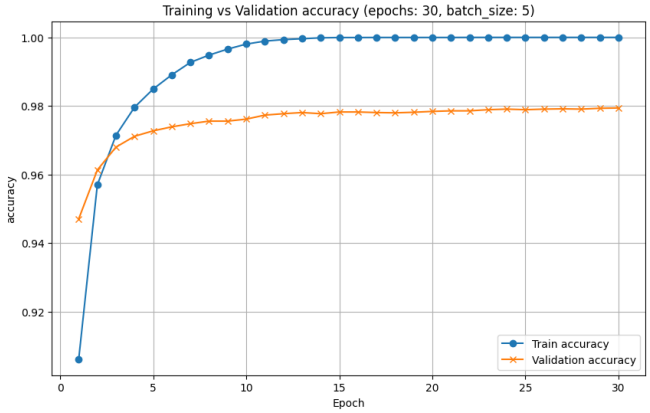

Figure 5: Training vs Validation cost overfitting model

The same observation applies to the evolution of accuracy over epochs. the accuracy reach a stunning 100percent on our training data, meaning that with the point used to train the network, it dont make any mistakes, but the validation accuracy struggle to get up.

We have achieved our best accuracy with the following hyperparameters and network definition.

hyperparameters	epochs	batch size
Value	15	100

Table 2: hyperparameters greatfitting model

```
let net = SequentialBuilder::new()
  .push(
    DenseLayer::new(784, 256,
    InitializerType::He))
  .push(
    ActivationLayer::from(
      Activation::ReLU)
    )
  .push(
    DenseLayer::new(256, 256,
    InitializerType::He))
  .push(
    ActivationLayer::from(
      Activation::ReLU)
    )
  .push(
    DenseLayer::new(256, 10,
    InitializerType::GlorotUniform))
  .push(
    ActivationLayer::from
    (
      Activation::Softmax)
    )
  .watch(MetricsType::Accuracy);
Ok(net.compile(GradientDescent::new(0.04),
↪  CostFunction::CrossEntropy)?)
```

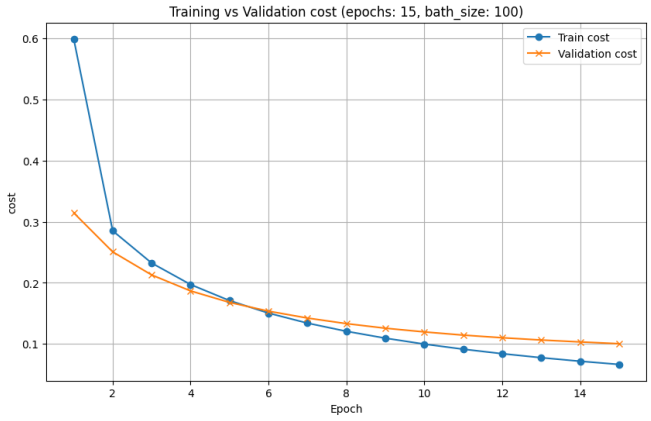

Figure 6: Training vs Validation cost greatfitting model

Here, our model seems to have generalized patterns, because the training cost and the validation cost follow the same path, slowly increasing, with the gap between curves never blowing off the roof.

we reach a validation cost of 0.097, the cost for the testing data set is 0.091.

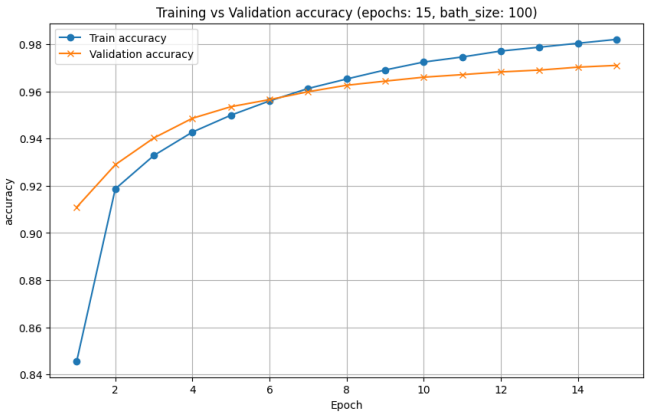

Figure 7: Training vs Validation cost greatfitting model

we reach a validation accuracy of 97.10percent for the final eppochs, the accuracy for the testing data set is 97.31percent. which is not a *state of the art* accuracy but is a good result.

12.2 Convolutional multi-layer perceptron

13 Conclusion

This project of creating our own *from scratch* library has brought us a lot of knowledge, It as made us greatly understand the whole neural network process. We have planned to add more functionality to our library and to optimize the code base even after the end of this project. we want to add more optimizers, the dropout layer, and a learning rate scheduler. and over the longer term, we'd like to add other neural network paradigm, such as Recurrent neural network.

References

[1] Papers with Code. *He Initialization*. n.d. URL: https://paperswithcode.com/method/he-initialization.

[2] Wikipedia contributors. *Precision and recall — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-May-2024]. 2024. URL: https://en.wikipedia.org/wiki/Precision_and_recall.

[3] Stack Exchange. *Backpropagation with Softmax Cross Entropy*. 2016. URL: https://stats.stackexchange.com/questions/235528/backpropagation-with-softmax-cross-entropy.

[4] Google. *Cross Entropy*. 2024. URL: https://www.google.com/search?q=cross+entropy&sourceid=chrome&ie=UTF-8.

[5] V7 Labs. *A Guide to Cross-Entropy Loss*. n.d. URL: https://www.v7labs.com/blog/cross-entropy-loss-guide.

[6] Yann LeCun. *Quand la machine apprend: La révolution des neurones artificiels et de l'apprentissage profond*. Odile Jacob, 2019.

[7] Towards Data Science. *Derivative of the Softmax Function and the Categorical Cross Entropy Loss*. 2019. URL: https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1.

[8] Vladimir Vapnik and Alexey Chervonenkis. "Theory of pattern recognition". In: (1974).

[9] Wikipédia. *Matrice Jacobienne*. 2024. URL: https://fr.wikipedia.org/wiki/Matrice_jacobienne.

[10] Wikipedia. *Hadamard product (matrices)*. 2024. URL: https://en.wikipedia.org/wiki/Hadamard_product_(matrices).

[11] Wikipedia. *One-hot*. 2024. URL: https://en.wikipedia.org/wiki/One-hot.

[12] Wikipedia. *Precision and recall*. 2024. URL: https://en.wikipedia.org/wiki/Precision_and_recall.

[13] Wikipedia. *Softmax function*. 2024. URL: https://en.wikipedia.org/wiki/Softmax_function.