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

Adrien PELFRESNE - Alexis VAPAILLE

May 27, 2024

Contents

1 Abstract **1**

2 Motivation **3**

3 Feed forward neural network **3**
3.1 General layer interface . . . 3
3.2 Dense layer . . . 3
3.3 Activation layer . . . 3

4 Cost function **4**

5 The learning process **4**
5.1 backpropagation . . . 4
5.2 Dense layer backpropagation . . . 4
5.3 Activation layer backpropagation . . . 5

6 Optimizers **5**
6.1 Adam . . . 5

7 All about batch **6**

8 Initializer **6**

9 Convolutional neural network **6**

10 Metrics **6**

11 Training, Validation and Test **6**

12 Results **6**
12.1 Multi-layer perceptron . . . 6
12.2 Convolutional multi-layer perceptron . . . 7

13 Optimisations **7**

14 Discussion **7**

15 Conclusion **7**

1 Abstract

In this report, we are going to review, and present our work of creating 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. Our implementation currently suport

- Dense Layer
- Activation Layer with various activation functions
- Convolutional Layer
- Rehshape Layer
- Optimizer (Adam and Stochastic gradient descent)
- Various parameters initialization methods

2 Motivation

The objective with this *from scratch* approach was to learn in details how neural network is really working. because high level library like keras in python and the incredible level of abstraction that come with it, doesn't teach us anything on the underlying process of learning. nevertheless, this underlying work that those library are doing is passionating, we thought it was a shame not to fully understand the whole deep learning process. The only helper library that we have used is ndarray which is similar to the *ndarray* container from the numpy python library. This *crate* (the name of a rust library) allowed us to do eifficents matrix multiplication and element wise operation efficently. You can find our rust project on github.

3 Feed forward neural network

The multi-layer perceptron that we have built is known as a *Feed forward neural network*, (or *Sequential*) in the keras api). 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 through the neural network and through different kind of layer.

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 as parameter the input vector X 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 $\frac{\partial C}{\partial X}$

3.2 Dense layer

The dense layer, also knowed as *Fully connected layer* or *linear layer* is the single most important layer in a neural network architecture. In a dense layer, each input neuron is connected to every output neurone, each connexion is called a *weight*, w_{ij} is the weight connecting neurone i to neurone j.

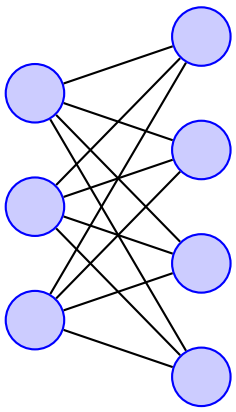

Figure 1: a Dense Layer

The output value y_j of a dense layer is given by

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

with b_j the bias term for the output neuron j this equation can be written using matrices

$$Y = XW + B$$

3.3 Activation layer

In a lot of deep neural network content, the activation function is applied inside the dense layer to produce the y output value from the weighted sum z. but in our implementation, to enhance the seperation of concerns, we opted for a decoupled version.

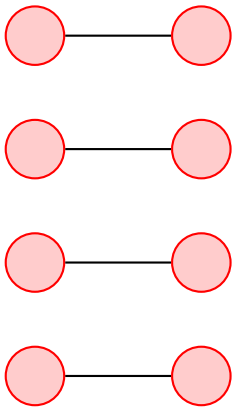

Figure 2: Element-wise Connections Between Two Layers

The Activation layer take a vector of size n as input and produce a same length vector as a output, with the activation function applied to each neuron. we must note that some activations function transform a vector by mapping individual component through the function, like the ReLU

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

while other activation functions such as *softmax* as input a vector z of K real number, and is used to convert a vector of K real number 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 $\sigma(\mathbf{z}) \in (0,1)^K$

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

we have implemented a variety of activations function, to try out different option and see what works the best with the mnist dataset, more on that in the results section.

4 Cost function

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

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

this cost function is great for regression neural network, but for a classification task, like the mnist data set (when we need to classifie each image as a digit) we can use a cost function like the multi-class cross entropy, which is defined as :

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

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.
and even

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

as for a one hot encoded observed values vector, only the observed class will be one and every other ocmponent will be 0.

5 The learning process

To make our neural network lean, we want to adjust his parameters (ie: weights and biases) to minimize a cost function C. The algorithm we used to minimize the cost function is known as *Gradient descent*. In a classical gradient descent step, you :

1. feed **all** your input data points into the network, calculating the cost function for every output.
2. calculate the overall cost function by averaging each data point cost.
3. compute the gradient of the overall cost function **with respect to the network parameters**.
4. update the parameters

5.1 backpropagation

The algorithm used to calculate the gradient of the cost function with respect to the neural network parameters is called backpropagation. It is a way to precisely calculate (and not only estimate like the finite difference method) this gradient. This gradient will be calculated for each layer, starting from the last layer and going all the way back to the first layer. 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, we want to calculate $\frac{\partial C}{\partial P}$, with P a trainable parameters of the layer, we also want to calculate $\frac{\partial C}{\partial X}$ the gradient with respect to input X, to be passed to the previous layer.

5.2 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

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

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,

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

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

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

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

5.3 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 pass 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 All about batch

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

9 Convolutional neural network

ALEXIS a toi de travailler

10 Metrics

11 Training, Validation and Test

To monitor the performances of our neural network, it is important to break down our

To measure how well the neural network has performed over our dataset, we can calculate multiple metrics, the first one, that we already have in hand is the cost.

12 Results

With our library, we can declarativaly create neural network, we are gonna list once the code to declare a net, and describe it.

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

here we have defined a neural network with only one hidden layer, containing 32 neuron. we have initialized our weights with the Glorot uniform method, and used the ReLU activation function after the hidden layer after the output layer, we have normalized the logits with the softmax function, and used cross entropy cost function.

in our library, you can use the *watch* method to watch a *MetricsType*, here we watched the accuracy of the network this will give us the network accuracy, with the training, validation and test set.

throughout the analysis of the results, to reach for the best neural network, we will try to reach the *VC trade-off principle* [1] [2], this principle explain 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 and the error on unseen data. We will often start with a low complexity model, and try to lift the complexity up unitl we met overfitting.

12.1 Multi-layer perceptron

for a simple sequential neural network, wiith one hidden layer of 32 neuron.

Parameters	epochs	batch size
Value	15	128

Table 1: Table 1 of the **original article** comparing MSE values for various denoising algorithms

12.2 Convolutional multi-layer perceptron

13 Optimisations

vérifier que l'interpretation du flamegrpah est correct et que je dis pas de la merde.

In additions to batching, to further optimize our rust code, we can look at the flamegraph of our functions call,

14 Discussion

15 Conclusion

References

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

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