ml_with_pytorch_book_part2

we start deep learning in this part

emergence of questions

- what does parameterization of nn means?
  - using only parameters to describe our function, our function here is our neural network.
- what does non-convex loss function means?
  - it simply means if we plot our loss against parameters then it does not have convex shape, it has lots of bumps and local minimas.
- what is automatic differentiation and how is it used in machine learning?
  - its computational way to find derivative i guess, its defining each step and then carrying out those steps, one at a time to find derivatives.
- what are jacobians?
  - probably these are matrixes of gradients i.e they store our partial derivatives

intuitions and random things

- we don't have weights for each example but we have weights for each feature of example so no. of weights == no. of features, given a input with multiple features we need the weights for each example so we can multiply each feature of that example with weight and then add it and then add bias in it and then find solution (yah i know we also need activation function)
- imp things to keep in mind: number of examples, number of features, number of weights, number of biases, number of neurons, number of layers.
- col one of weight matrix is like weights for 1st feature only and each row is for new activation, like weights for first feature only but for all the activations.
- if we put array inside array in numpy then it will return the values from first array that are on the indices where indices are the values from second array (inner array). given is the example of it:

```
arr = np.array([1,4,5,6])
nw = np.array([3,2,0])
```

```
nw_arr = arr[nw]
print(nw_arr)
```

- what we don't know yet (have to learn):
    - batch normalization
    - adaptive learning rates
    - sophisticated SGD based optimization algorithms
    - dropout
- visualization of weight matrices,

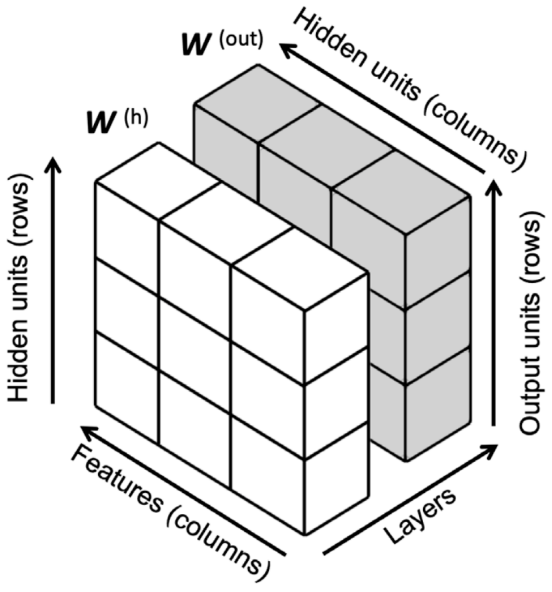


Figure 11.10: A visualization of a three-dimensional tensor

in given visualization it might seem that both W_h and W_{out} have same dimensions, but its not the usual case. both of these matrices differ in dimensions depending on hidden layer units and output units.

- lets take neural network with input layer, hidden layer and output layer, now when we multiply single input example with weight matrix and add bias the output that we will get from hidden layer will be one output for each activation right? i.e it will be like single vector of length which is equal to to number of activations in hidden layer, right?
    - Yes — exactly right. ✅

chapter 11 : Implementing a Multilayer Artificial Neural Network from Scratch

`page no. 335 to 368`

overview

- neural networks (NN)
  - single layer neural networks
  - multilayer neural networks/multilayer perceptron (mlp)/deep NN
  - forward propagation
- implementing mlp from scratch and training it on mnist dataset of handwritten digits
- coding a training loop to train network on mini-batches of the data via backpropagation
- evaluating a NN performance:
  - plotting loss curve, accuracy curve
  - plotting misclassified examples
- finetuning model
- deep dive into loss computation and backpropagation algorithm
- convergence

Single layer NN

- its the simplest neural network, which has only input layer and output layer connect by a single link, hence single layer NN.
- Adaline (ADAptive LInear NEuron) is the example of single layer NN.
- we use this adaline algo to do binary classification, given is the steps of how this algorithm works:

1. first we calculate **net input** z using given formula :

$$z = \sum_j w_j x_j + b = w^T x + b$$

in above formula w_j is the weight for jth example and x_j is the jth example and b is bias for that entire layer i.e for all the inputs, so simply saying we multiply all our inputs with weights and we have separate weight for each example, then we add bias in each example.

2. then we have **activation function**, for adaline tho our activation function is pretty simple it just forwards or net inputs as it is.
3. then we have **threshold function** to convert our net inputs into binary classes, for adaline we have given threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{otherwise.} \end{cases}$$

4. and this is the process to calculate the outputs. these binary classes are our outputs or prediction, and now we have to train our NN to make better predictions as these above predictions are nothing but random guesses. to do that we use **gradient descent** algorithm.

5. we use given formulae to update our weights and bias:

$$w := w + \Delta w, \quad b := b + \Delta b$$

here

$$\Delta w = -\eta \frac{\delta L}{\delta w_j}, \text{and } \Delta b - \eta \frac{\delta L}{\delta b}$$

η is the learning rate (its a hyperparameter we set) and $\frac{\delta L}{\delta w_j}$ and $\frac{\delta L}{\delta b}$ are weight gradient and bias gradient. as we know gradient represent the direction of steepest ascent, hence we take negative of gradient to minimize the loss.
for adaline we use mean of square error (MSE) as our **loss function** that we have to minimize for better predictions, given is the formula of how we calculate our loss gradient for MSE:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i \left(y^{(i)} - a^{(i)} \right)^2 = -\frac{2}{n} \sum_i \left(y^{(i)} - a^{(i)} \right) x_j^{(i)}$$

we run the above steps for multiple times each loop is called epoch and we run these steps for multiple epochs till we get satisfactory results. and that how single layer neural network works.

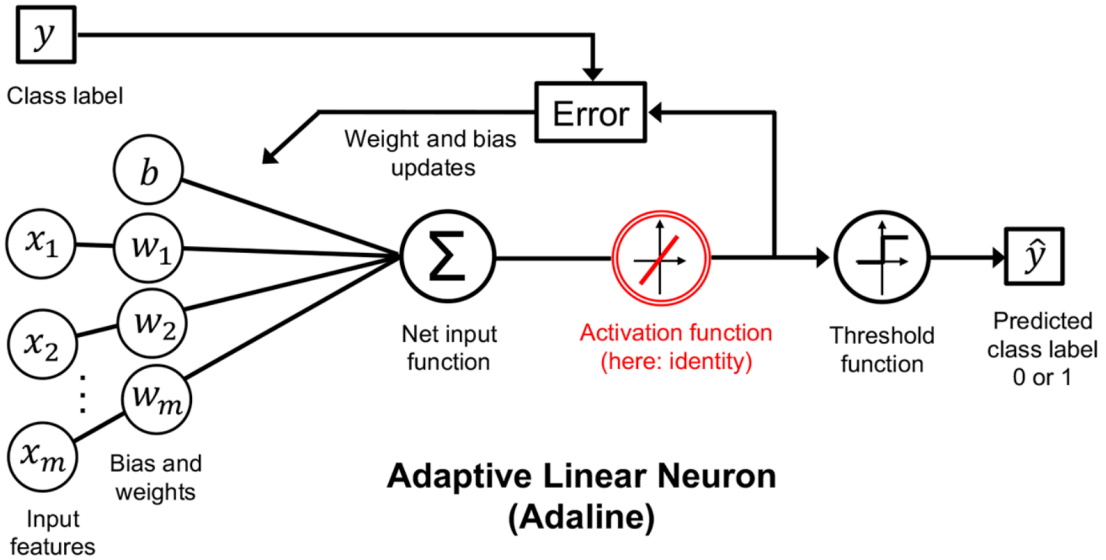


Figure 11.1: The Adaline algorithm

multilayer NN/MLP

- multilayered neural network is neural network that contains more than one layer between inputs and outputs.
- MLP means multilayered perceptron.
- in MLP each layers are fully connected to the next layer i.e each input feature (i.e dimension of each example or features) and activation is connected to all the activations in next layer.
- given is the diagram for two layered MLP:

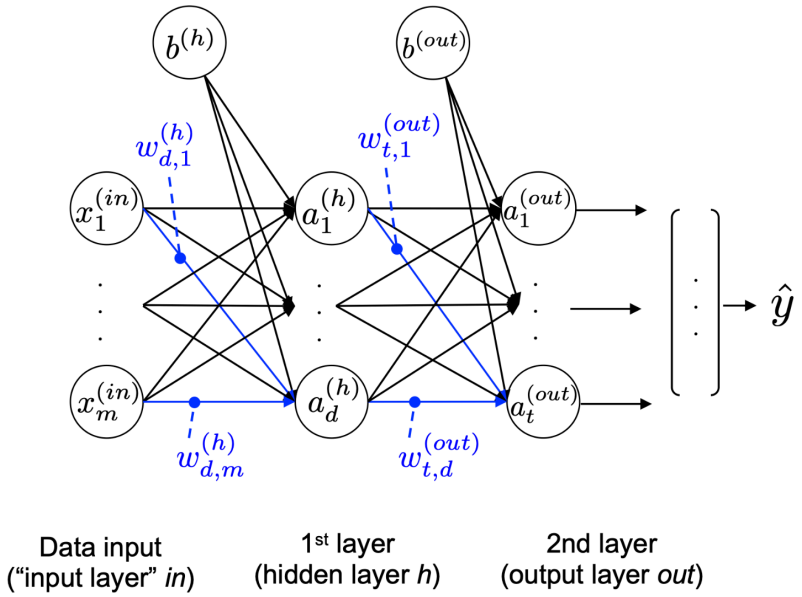


Figure 11.2: A two-layer MLP

- the first layers is input layer where give our inputs to a NN, second layer is hidden layer and last layer is output layer. input layer is not counted when we count the layers of our NN hence above NN is 2 layered. we can also understand this by seeing their are two connecting links between layers and hence it is two layered neural network.
- when we add more than one hidden layer in NN, it is called **deep neural network**.
- Each node in layer l is connected to all nodes in layer $l+1$ via a weight coefficient.

> ❛❛ the number of neurons / activations in the middle layer is an hyper parameter i.e we decide on how many neurons / activations to put in hidden layer. and this is how we get one of the dimension of our weight matrix.

in simple steps this is how MLP works:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the loss that we want to minimize using a loss function that we will describe later.

3. We backpropagate the loss, find its derivative with respect to each weight and bias unit in the network, and update the model.

forward propagation

- forward propagation is just a method to pass our data from NN in forward direction i.e passing our data through NN.

first we connect our inputs to activations in hidden layer using given formulae:

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma(z_1^{(h)})$$

here $z_1^{(h)}$ is the **net input** for first example and $\sigma(.)$ is the **activation function** for it. here $x_1, x_2 \ldots, x_m$ are the input features (each example in our training dataset have m features). we use activation function to bring non-linearity in our model, in our MLP we will use sigmoid function as activation function, which looks like this:

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

sigmoid function maps the net input z onto a logistic distribution in the range 0 to 1.

- sigmoid function intuition
- what sigmoid works when ur data is numerically very high - another reason to scale data

vectorized version of mlp formula:

$$z^{(h)} = x^{(in)} W^{(h)T} + b^{(h)}$$

$$a^{(h)} = \sigma(z^{(h)})$$

in above formulae, x is a single input example with dimensions $1 \times m$ i.e its $1 \times m$ vector. W is a weight matrix of $d \times m$ where d is the number of activations in hidden layer, in above formula we take transpose of weight matrix so that we can do matrix multiplication with input x.
and b is a bias vector, consisting d biases, one for each activation in hidden layer.

our z and a are both vectors with dimensions $1 \times d$

we can generalize above formula for all the training exampel, the only change will be our X matrix will become of size $n \times m$ where n is the number training example. and our output matrix will be of size $n \times d$.

and in similar we can apply the same function on hidden layer's output to find activations in output layer, formula is given:

$$Z^{(out)} = A^{(h)} W^{(out)T} + b^{(out)}$$

here our weight matrix is of size $t \times d$ where t is the number of output units, we use transpose of it for matrix multiplication, here A is the output of our previous layer and b is $n \times t$ dimentional matrix, here z will also become $n \times t$ dimension matrix.
understand: here t is the number of classes that we want to predict, commonly we keep it only 1, like we only predict probability for spam class and use logic that if probability of spam is low then its not spam.

- this is how forward propagation looks through each layer:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)^T} + \boldsymbol{b}^{(h)} \qquad \text{(net input of the hidden layer)}$$

$$\boldsymbol{A}^{(h)} = \sigma(\boldsymbol{Z}^{(h)}) \qquad \text{(activation of the hidden layer)}$$

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)^T} + \boldsymbol{b}^{(out)} \qquad \text{(net input of the output layer)}$$

$$\boldsymbol{A}^{(out)} = \sigma(\boldsymbol{Z}^{(out)}) \qquad \text{(activation of the output layer)}$$

computing loss function

as we know our output is $t \times 1$, where t is the number of classes our output can be.

our output after passing the image through all the layers will look like the given array, and y is our onehot encoded target variable, here 2nd class is the correct prediction.

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

when we calculate MSE loss, we minus a^{out} from y, and then take average of it and we repeat this for all the examples in our batch i.e n times, given is the mathematical formula for it,

$$L(W, b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{t}\sum_{j=1}^{t}\left(y_j^{[i]} - a_j^{(out)[i]}\right)^2$$

here i represents particular example from out dataset and in total we have n examples, and j is the output for particular class.

backpropagation

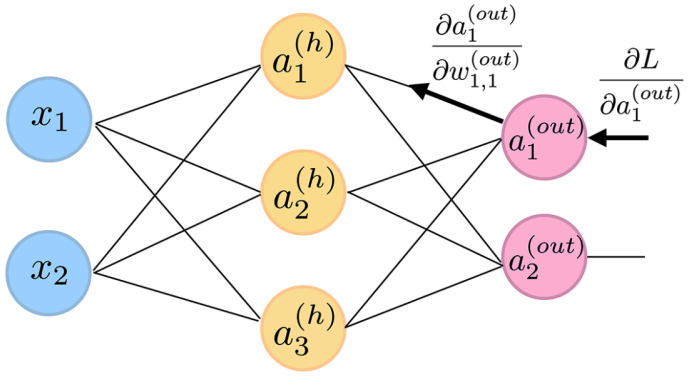
chain rule:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg}\cdot\frac{dg}{dh}\cdot\frac{dh}{du}\cdot\frac{du}{dv}\cdot\frac{dv}{dx}$$

> *back propagation is not just chain rule but applying chain rule effectively.*

here we use chain rule to calculate the gradients for weights and biases, gradients are derivative of loss with respect to weights and biases, they give us the steepest direction of the curve and also the magnitude which is rate at which the change is happening in the loss wrt parameters like weight or bias. these gradients are then used to update weights such that loss is minimized.

backpropagation is a chain rule applied in reverse direction to neural network, given images give good visualizations,



Gradient for output layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

Figure 11.12: Backpropagating the error of an NN

here's how chain rule would look like for calculating gradient for 1st weight in the weight matrix of output layer,

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

lets go in more details and calculate all of it mathematically,

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}}(y_1 - a_1^{(out)})^2 = 2(a_1^{(out)} - y)$$

next,

$$\frac{\partial a_1^{(\text{out})}}{\partial z_1^{(\text{out})}} = \frac{\partial}{\partial z_1^{(\text{out})}} \frac{1}{1+e^{-z_1^{(\text{out})}}} = \left(\frac{1}{1+e^{-z_1^{(\text{out})}}}\right)\left(1 - \frac{1}{1+e^{-z_1^{(\text{out})}}}\right) = a_1^{(\text{out})}\left(1 - a_1^{(\text{out})}\right)$$

so here's some more equations to make sense of what happened above,

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

$$\sigma(z)(1-\sigma(z)) = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \frac{e^{-z}}{(1+e^{-z})^2}$$

its simple derivative man, its not that hard, u did lot of them in ur 12th grade so jus chill out and use ur brain n u'll easily figure that shit out,

here's last derivative,

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}}\left(a_1^{(h)} w_{1,1}^{(out)} + b_1^{(out)}\right) = a_1^{(h)}$$

now we put all of them together,

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2(a_1^{(out)} - y) \cdot a_1^{(out)}(1 - a_1^{(out)}) \cdot a_1^{(h)}$$

and with that we get the gradient for $w_{1,1}$ i.e first weight in the weight matrix of output layer. we use this gradient to update the weights with given formula,

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

given is image that represent how to perform backpropagation for weights in hidden layer,

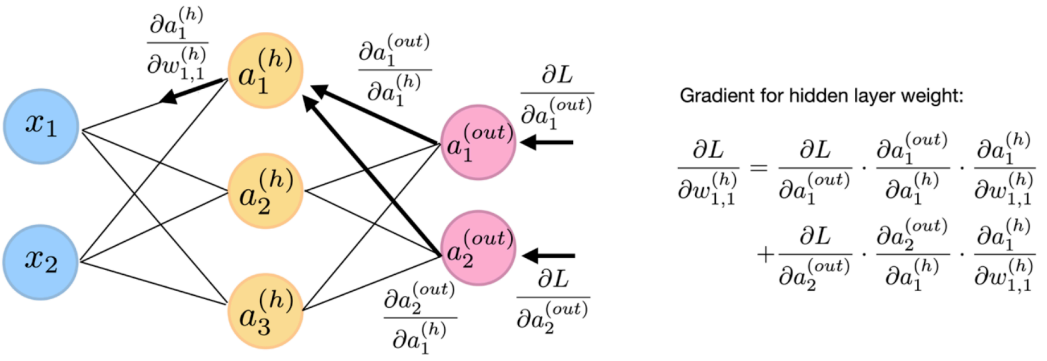


Figure 11.13: Computing the partial derivatives of the loss with respect to the first hidden layer weight

It is important to highlight that since the weight $w_{1,1}^{(h)}$ is connected to both output nodes, we have to use the multi-variable chain rule to sum the two paths highlighted with bold arrows. As before, we can expand it to include the net inputs z and then solve the individual terms:

$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(out)}} =& \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \\ &+ \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}\end{aligned}$$

convergence

Multilayer NNs are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines. In multilayer NNs, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface, and the optimization algorithm can easily become trapped in local minima, as shown in figure:

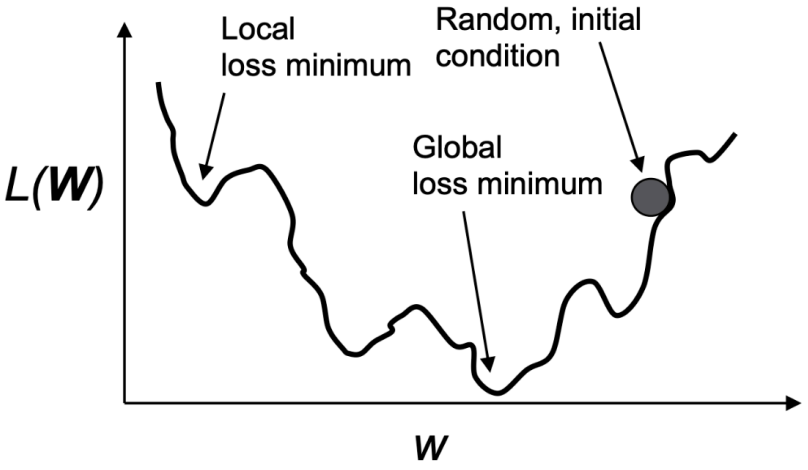


Figure 11.14: Optimization algorithms can become trapped in local minima

in multilayer nn's we use mini-batch gradient descent, which is like middle ground between stochastic gradient descent (aka online learning, uses only 1 example to update weights) and batch gradient descent (aka vanilla gradient descent, here we use entire dataset to update to calculate loss on and update weights), in mini-batch gradient descent we use batches of size like 32-512, and update weights on basis of loss calculated with these examples, its better because we can use vectorized form which increase computational efficiency over SGD and its better than GD because it can calculate weights faster than its, so its perfect middle ground that we use to converge our neural networks.