Neural Networks

Ben Schweizer

Fakultät für Mathematik, TU Dortmund

25. September 2024

Aims of this course

- **Learn basic neural network vocabulary**
- **Understand training of a network**
- **Loose your fear (if there was any)**

Disclaimer

I am not at all an expert in neural networks!

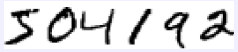
Literature: I am following the wonderful introduction of Michael Nielsen from 2019, see http://neuralnetworksanddeeplearning.com/

My main contribution is to adapt the exposition for mathematicians

Motivation

Aim

Let the computer recognize the pixel graphic



as the number 504192.

We provide:

- the 100 images on the right
- ... together with the information:
  First row is “0”, “4”, “1”, “9”, ...
  Second row is “5”, “3”, ...

Note: The given “5” is not *identical* to any “5” in the list

Idea of the analyst (not necessarily smart)

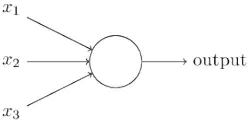
Introduce a measure of distance for pixel graphics

Perceptrons I

An idea of the 1950ies: The perceptron

$\longrightarrow$ A device to convert input into output

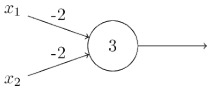
- Input $x = (x_1, x_2, x_3) \in \mathbb{R}^3$
- Decision parameters I: Weight vector $w = (w_1, w_2, w_3) \in \mathbb{R}^3$
- Decision parameters II: A threshold value $b \in \mathbb{R}$



$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Perceptrons II

An interesting example with only 2 inputs:
Choose $w_1 = w_2 = -2$ and $b = 3$



Result for $(x_1, x_2) = (1, 1)$:

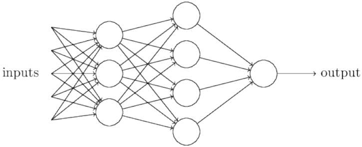
$w \cdot x + b = -4 + 3 = -1 \leq 0$, output: 0

Result in any other case:

$w \cdot x + b \geq 0$, output: 1

$\longrightarrow$ The above perceptron realizes a NAND

Imagine what you can do with this:



Sigmoid perceptron

Above: Sign function to define the output

$$\mathrm{sign}(z) := \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases}$$

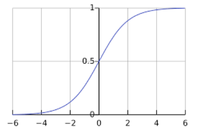
An input x gives the output

$$\mathrm{sign}(w \cdot x + b)$$

This makes the analyst happy: Let us define a smoothed version:

Sigmoid function to define the output

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

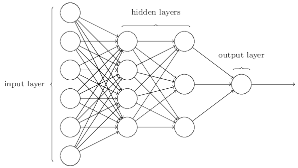


Input x gives output

$$\sigma(w \cdot x + b)$$

σ is a smooth function on $\mathbb{R}$, monotonically increasing from 0 to 1

Architecture: Names and formulas



Input: $x \in \mathbb{R}^6$

Values in first hidden layer: $y = y^{(1)} \in \mathbb{R}^4$

Four vectors w of the second column, each with 6 entries: matrix $A^{(1)} \in \mathbb{R}^{4\times 6}$

Four bias numbers "b" of the second column give a vector $b^{(1)} \in \mathbb{R}^4$

Simple math for a complicated network

y is calculated as

$$y = \sigma(A \cdot x + b)$$

Note: σ is applied to each entry of $A \cdot x + b$ separately

Entire network: input $= x =: y^{(0)}$, $\quad y^{(1)} := \sigma(A^{(1)} \cdot y^{(0)} + b^{(1)})$, $y^{(2)} := \sigma(A^{(2)} \cdot y^{(1)} + b^{(2)})$, output $:= \sigma(A^{(3)} \cdot y^{(2)} + b^{(3)})$

Idea of neural networks

What did we construct?

We constructed a map $f : \mathbb{R}^6 \to \mathbb{R}$ with values in $[0, 1]$.

The map depends on the entries of $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $b^{(1)}$, $b^{(2)}$, $b^{(3)}$

Idea of neural networks:

We seek a function $f : \mathbb{R}^N \to \mathbb{R}$ that realizes "this is a 4":

N: number of pixels for the graphic of one digit, e.g.: $N = 28 \times 28$

Input: 504192

f maps the *first* pixel square ("the 5") to something near 0

f maps the *second* pixel square ("the 0") to something near 0

f maps the *third* pixel square ("the 4") to something near 1, etc.

Task

Find parameters $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $b^{(1)}$, $b^{(2)}$, $b^{(3)}$
such that f as above realizes the function "this is a 4"

Cost function

For training, we have a finite set of inputs. For some $K \in \mathbb{N}$:

$$X_T = (x^1, ..., x^K)$$

We are given the values $R_T = (r^1, ..., r^K)$ of "correct" outputs

The perfect function would satisfy: $f(x^k) = r^k$ for all $k \leq K$

Cost function (= Error = Loss)

We use the squared ℓ^2-norm to measure the error,

$$C(A, b) := \frac{1}{2K} \sum_{k=1}^{K} |f_{A,b}(x^k) - r^k|^2$$

Left to do:

Use the steepest decent algorithm to find A and b such that C is minimal!

Gradients

Cost function

$$C(A,b) := \frac{1}{2K}\sum_{k\leq K} |f_{A,b}(x^k) - r^k|^2$$

Aim: Calculate the derivatives

$$\frac{\partial}{\partial a_{i,j}} C(A,b) \quad \text{and} \quad \frac{\partial}{\partial b_i} C(A,b)$$

We perform first the "natural" way to calculate all derivatives

Main difficulty: x is fixed and we differentiate with respect to parameters, e.g., $b_i^{(q)}$

Later, we learn *backpropagation* $\longrightarrow$ easier and faster to calculate (and harder to understand)

Derivatives for the first layer

The first layer of the network

$y^{(0)} := x$ (the input)

$z^{(1)} := A^{(1)} \cdot y^{(0)} + b^{(1)}, \qquad y^{(1)} := \sigma(z^{(1)})$

We calculate:

$$\frac{\partial y_\ell^{(1)}}{\partial a_{\ell,j}^{(1)}} = \sigma'(z_\ell^{(1)})\, \frac{\partial z_\ell^{(1)}}{\partial a_{\ell,j}^{(1)}} = \sigma'(z_\ell^{(1)})\, y_j^{(0)} \quad \text{and} \quad \frac{\partial y_\ell^{(1)}}{\partial b_\ell^{(1)}} = \sigma'(z_\ell^{(1)})$$

For $\ell \neq i$:

$$\frac{\partial y_\ell^{(1)}}{\partial a_{i,j}^{(1)}} = \sigma'(z_\ell^{(1)}) \frac{\partial z_\ell^{(1)}}{\partial a_{i,j}^{(1)}} = 0 \quad \text{and} \quad \frac{\partial y_\ell^{(1)}}{\partial b_i^{(1)}} = 0$$

For given x, all these real numbers can be evaluated!

Derivatives for the second layer

Note: Other derivatives vanish, e.g.: $\frac{\partial y_\ell^{(1)}}{\partial b_\ell^{(2)}} = 0$

$\longrightarrow$ the first layer does not know about the second layer

The second layer of the network

$y^{(0)} := x$ (the input)

$z^{(1)} := A^{(1)} \cdot y^{(0)} + b^{(1)}, \quad y^{(1)} := \sigma(z^{(1)})$

$z^{(2)} := A^{(2)} \cdot y^{(1)} + b^{(2)}, \quad \text{output} := y^{(2)} := \sigma(z^{(2)})$

Some derivatives are exactly as in the first layer, e.g.:

$$\frac{\partial y_\ell^{(2)}}{\partial a_{\ell,j}^{(2)}} = \sigma'(z_\ell^{(2)})\, y_j^{(1)}$$

As noted above, e.g.: $\frac{\partial y_\ell^{(2)}}{\partial b_\ell^{(3)}} = 0$

More Derivatives for the second layer

There are still interesting derivatives to calculate ...

The second layer of the network

$y^{(0)} := x$ (the input)

$z^{(1)} := A^{(1)} \cdot y^{(0)} + b^{(1)}, \quad y^{(1)} := \sigma(z^{(1)})$

$z^{(2)} := A^{(2)} \cdot y^{(1)} + b^{(2)}, \quad \text{output} := y^{(2)} := \sigma(z^{(2)})$

$$\frac{\partial y_\ell^{(2)}}{\partial a_{i,j}^{(1)}} = \sigma'(z_\ell^{(2)}) \, \frac{\partial z_\ell^{(2)}}{\partial a_{i,j}^{(1)}} = \sigma'(z_\ell^{(2)}) \, a_{\ell,i} \, \frac{\partial y_i^{(1)}}{\partial a_{i,j}^{(1)}}$$

Simplify by inserting ..... ?

$$\frac{\partial y_i^{(1)}}{\partial a_{i,j}^{(1)}} = \sigma'(z_i^{(1)}) \, y_j^{(0)}$$

Gradient of the cost function

In this way, simple evaluations yield all derivatives

$$\frac{\partial y_\ell^{(p)}}{\partial a_{i,j}^{(q)}} \quad \text{and} \quad \frac{\partial y_\ell^{(p)}}{\partial b_i^{(q)}} \quad \text{for every input } x = y^{(0)}$$

Cost function

$$C(A,b) := \frac{1}{2K} \sum_{k \leq K} |f_{A,b}(x^k) - r^k|^2$$

Output = value in last layer: $\quad f_{A,b}(x^k) = y_1^{(p)}$

(for p layers; 1 is the only index for the last layer)

Result

$$\frac{\partial}{\partial a_{i,j}^{(q)}} C(A,b) = \frac{1}{K} \sum_{k \leq K} \left(f_{A,b}(x^k) - r^k \right) \left. \frac{\partial y_1^{(p)}}{\partial a_{i,j}^{(q)}} \right|_{x = x^k}$$

Steepest decent algorithm

Choose a step size $\Delta t > 0$

Let a guess for the network be given: $A = A^{\text{old}}$ and $b = b^{\text{old}}$

For the training data $(x^k)_{k\leq K}$ and $(r^k)_{k\leq K}$ and in the point $(A,b) = (A^{\text{old}}, b^{\text{old}})$, calculate all derivatives

$$\frac{\partial}{\partial a_{i,j}^{(q)}} C(A,b) \quad \text{and} \quad \frac{\partial}{\partial b_i^{(q)}} C(A,b)$$

Update/improve coefficients by setting

$$a_{i,j}^{(q),\text{new}} := a_{i,j}^{(q),\text{old}} - \Delta t \, \frac{\partial}{\partial a_{i,j}^{(q)}} C(A,b)$$

$$b_i^{(q),\text{new}} := b_i^{(q),\text{old}} - \Delta t \, \frac{\partial}{\partial b_i^{(q)}} C(A,b)$$

Backpropagation

Here comes a really smart idea ...
Introduce the new variables

$$\delta_j^{(q)} := \frac{\partial C(A,b)}{\partial z_j^{(q)}}$$

This is confusing!

The input x is fixed
We can change the $a_{j,\ell}^{(q)}$ and the $b_j^{(q)}$, but not “directly” the $z_j^{(q)}$

More precisely:

$$\delta_j^{(q),k} := \frac{\partial}{\partial z_j^{(q)}} \frac{1}{2K} |y_1^{(p),k} - r^k|^2$$

This is well-defined

We consider layer q as input layer, keep all a and b fixed. We check how C changes when $z_j^{(q)}$ is modified with the data of input x^k

Backpropagation

The derivative with respect to the last layer (p) can be evaluated

$$\delta_1^{(p),k} = \frac{\partial C(A,b)}{\partial z_1^{(p)}} = \frac{1}{K}\left(f_{A,b}(x^k) - r^k\right)\sigma'(z_1^{(p)})$$

Next aim (suppressing k):

$$\delta_j^{(q)} := \frac{\partial C(A,b)}{\partial z_j^{(q)}}$$

Apply the chain-rule

$$\delta_\ell^{(q-1)} = \frac{\partial C(A,b)}{\partial z_\ell^{(q-1)}} = \sum_j \frac{\partial C(A,b)}{\partial z_j^{(q)}} \frac{\partial z_j^{(q)}}{\partial z_\ell^{(q-1)}}$$
$$= \sum_j \delta_j^{(q)}\, a_{j,\ell}^{(q)}\, \sigma'(z_\ell^{(q-1)})$$

This provides all the $\delta_\ell^{(q)}$ (calculating "backwards")!

Backpropagation

Assume: We have calculated all the

$$\delta_i^{(q),k} = \frac{\partial C(A,b)}{\partial z_i^{(q)}} = \frac{\partial}{\partial z_i^{(q)}} \frac{1}{2K} |y_1^{(p),k} - r^k|^2$$

Claim: This provides all the desired information!

We obtain all derivatives of C

$$\frac{\partial C(A,b)}{\partial a_{i,j}^{(q)}} = \frac{\partial}{\partial a_{i,j}^{(q)}} \frac{1}{2K} \sum_k |y_1^{(p),k} - r^k|^2 = \sum_k \delta_i^{(q),k} \, \frac{\partial z_i^{(q)}}{\partial a_{i,j}^{(q)}}$$
$$= \sum_k \delta_i^{(q),k} \, y_j^{(q-1)}$$

Similarly:

$$\frac{\partial C(A,b)}{\partial b_i^{(q)}} = \sum_k \delta_i^{(q),k}$$

Comparison of forward and backward differentiation

We had the following formulas for derivatives:

Forward

$$\frac{\partial y_\ell^{(q)}}{\partial a_{\ell,j}^{(q)}} = \sigma'(z_\ell^{(q)})\, y_j^{(q-1)}$$

$$\frac{\partial y_\ell^{(q)}}{\partial a_{i,j}^{(q-1)}} = \sigma'(z_\ell^{(q)})\, a_{\ell,i}\, \frac{\partial y_i^{(q-1)}}{\partial a_{i,j}^{(q-1)}}$$

Backward

$$\delta_\ell^{(q-1)} = \sum_j \delta_j^{(q)}\, a_{j,\ell}^{(q)}\, \sigma'(z_\ell^{(q-1)})$$

The number of unknowns is very different:

- Backward: Number of nodes of the network
- Forward: Number of nodes **times** number of edges

Some vocabulary

"weights" The values of the w's. For us: The entries $a_{i,j}^{(q)}$ of the matrices

"biases" The values of the b's, hence: the $b_i^{(q)}$

"activation-function" In our case: The sigmoid σ

"activations" The values $y_j^{(q)}$ (the $z_j^{(q)}$ are pre-activations)

"Forward-Pass" Go forward through the network, calculate all the $y_j^{(q)}$ and $z_j^{(q)}$

"Backward-Pass" Go backward through the network, calculate all derivatives, using the values of the Forward-Pass

"Output error " The $\delta_j^{(q)} = \frac{\partial C(A,b)}{\partial z_j^{(q)}}$

"learning rate" The Δt in the gradient descent scheme

```
def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x.  ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book.  Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on.  It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

Stochastic mini-batches

Let's recall what has to be done:

1. For every x^k: forward-pass to calculate activations
2. For every x^k: backward-pass to calculate derivatives

Taking an average $\frac{1}{K}\sum_{k=1}^{K}$ we find, for every i, j, and q:

$$\frac{\partial C(A,b)}{\partial a_{i,j}^{(q)}} \quad \text{and} \quad \frac{\partial C(A,b)}{\partial b_i^{(q)}}$$

Mini-batch stochastic gradient descent

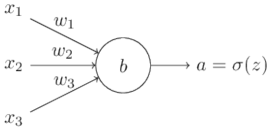
Take only $K_0 \leq K$ training inputs x^k, randomly chosen. Denote them as X_j (with desired outputs R_j) and use the modified cost functional

$$C_0(A,b) := \frac{1}{2K_0}\sum_j |f_{A,b}(X_j) - R_j|^2$$

Improve parameters with $\nabla_A C_0$ and $\nabla_b C_0$

Problem with small derivatives

Consider the last neuron with $z = \sum_j w_j x_j + b$



The output is $a = \sigma(z)$. The desired output is r.

Assume that the network is terribly wrong

Desired output is $r = 0$. **But:** $z = 100$ and $a \approx 1$

Derivatives of output a:

$$\frac{\partial a}{\partial w_j} = \sigma'(z)\,\frac{\partial z}{\partial w_j} = \sigma'(z)\,x_j$$

This expression contains $\sigma'(z)$, which is terribly small!

Cross-entropy cost function

Our cost function was $C_{\text{old}}(A,b) = \frac{1}{2K}\sum_k |f_{A,b}(x^k) - r^k|^2$

Then: All the x^k with terribly wrong results do not contribute to learning $\longrightarrow$ $\frac{\partial f_{A,b}(x^k)}{\partial a_{i,j}}$ is small

A smart idea:

The cross-entropy cost function

$$C = -\frac{1}{K}\sum_k \left[r \ln a + (1-r)\ln(1-a)\right]$$

a is the output for x^k and r is the desired output r^k

Is this a cost function?

1. For $r \in [0,1]$: C is always non-negative
2. For $r = 0$ and $r = 1$ holds: $C = 0$ for $a = r$

Derivatives of the cross-entropy cost function

The cross-entropy cost function

$$C = -\frac{1}{K}\sum_k \left[r \ln a + (1-r)\ln(1-a)\right]$$

With $a = \sigma(z)$ we calculate, suppressing k in $x = x^k$:

$$\begin{aligned}
\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial w_j} &= -\frac{1}{K}\sum_k \left(\frac{r}{a} - \frac{(1-r)}{1-a}\right)\frac{\partial a}{\partial w_j} \\
&= -\frac{1}{K}\sum_k \left(\frac{r}{a} - \frac{(1-r)}{1-a}\right)\sigma'(z)x_j \\
&= \frac{1}{K}\sum_k \frac{\sigma'(z)}{\sigma(z)(1-\sigma(z))}(\sigma(z) - r)\, x_j
\end{aligned}$$

Miracle:

$$\frac{\sigma'(z)}{\sigma(z)(1-\sigma(z))} = 1 \qquad \longrightarrow$$

small derivative is cancelled!

Conclusions

You have (hopefully) learned:

- Principles of a neural network: Inputs x^k and desired outputs r^k as learning data, weights A, biases b, activation function σ
- Cost functional C. Learning is steepest decent: Improve the A's and b's!
- How to calculate derivatives. How to use backpropagation.
- Mini-batches and cross-entropy cost function

Thank you for participating!