Neural Network for recognizing handwritten numbers

Oleksii Symon

1 Activation function choice

To control outputs from our neurons and their impacts on the next layers of neurons we should have an activation function. A good activation function should give small outputs for input values those are less than some threshold and bigger outputs for others.

$$input\_data < threshold \rightarrow activation\_function(input\_data) = small\_output$$
$$input\_data >= threshold \rightarrow activation\_function(input\_data) = normal\_output$$

For providing such outputs we can use S-shaped functions. We will use **sigmoid function** $\frac{1}{1+e^{-x}}$.

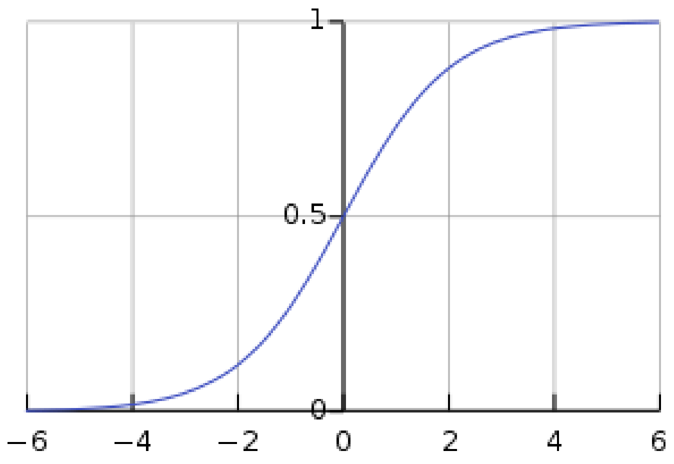

Figure 1: *sigmoid* function

2 Propagation input signal forward and getting an output

Our neural network will have 3 layers of nodes: input, hidden and output layers. Every node in input layer will be connected to every node in hidden layer $input \rightarrow hidden$, this is also true for hidden and output layers $hidden \rightarrow output$. We will present these connections as matrices.

$$W_{input,hidden} = \begin{pmatrix} w_{1,1} & w_{2,1} & w_{3,1} & \cdots & w_{n,1} \\ w_{1,2} & w_{2,2} & w_{3,2} & \cdots & w_{n,2} \\ \dots & \dots & \dots & \dots & \dots \\ w_{1,m} & w_{2,m} & w_{3,m} & \cdots & w_{n,m} \end{pmatrix} \tag{1}$$

The matrix 1 represents connections between input and hidden layers, where input layer has n and hidden has m nodes

$$W_{hidden,output} = \begin{pmatrix} w_{1,1} & w_{2,1} & w_{3,1} & \cdots & w_{m,1} \\ w_{1,2} & w_{2,2} & w_{3,2} & \cdots & w_{m,2} \\ \dots & \dots & \dots & \dots & \dots \\ w_{1,k} & w_{2,k} & w_{3,k} & \cdots & w_{m,k} \end{pmatrix} \tag{2}$$

The matrix 2 represents connections between hidden and output layers, where hidden layer has m and output has k nodes

Input (that is also an output) for the input layer is passed to the function as list.

$$input_{input} = \begin{pmatrix} i_1 & i_2 & i_3 & \cdots & i_n \end{pmatrix}$$

Input and output for the hidden layer are obtained:

$$input_{hidden} = W_{input,hidden} * (input_{input})^{\intercal}$$
$$output_{hidden} = sigmoid(input_{hidden})$$

Input and output for the output layer are obtained:

$$input_{output} = W_{hidden,output} * (output_{hidden})$$
$$final_{output} = sigmoid(input_{output})$$

3 Error backpropagating

To refine weights between different layers of nodes in our neural network we should know errors that layers produce. The output error is easy to find

$$error_{output} = training\_data - final_{output} = \begin{pmatrix} error_{output,1} \\ error_{output,2} \\ error_{output,3} \\ \dots \\ error_{output,k} \end{pmatrix}$$

To find a hidden error, for example, for the mth node, we look what contribution it made to errors in output layer (through link weights) and sum these contributions up.
Let $weights_{sum,k}$ be the sum of all weights those are connected to node k

$weights_{sum,k} = W_{hidden,output}(1,k) + W_{hidden,output}(2,k) + \cdots + W_{hidden,output}(m,k)$

$e_{hidden,m} = e_{output,1} * \frac{W_{hidden,output}(m,1)}{weights_{sum,output_1}} + e_{output,2} * \frac{W_{hidden,output}(m,2)}{weights_{sum,output_2}} + \cdots + e_{output,k} * \frac{W_{hidden,output}(m,k)}{weights_{sum,output_k}}$

We can ignore division as it is just a normalizing factor. After that we represent backpropagating in matrix form

$$error_{hidden} = W^{\intercal}_{hidden,output} * error_{output} =$$
$$\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,k} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,k} \\ \multicolumn{5}{c}{\dotfill} \\ w_{m,1} & w_{m,2} & w_{m,3} & \cdots & w_{m,k} \end{pmatrix} * \begin{pmatrix} error_{output,1} \\ error_{output,2} \\ error_{output,3} \\ \dots\dots\dots \\ error_{output,k} \end{pmatrix}$$

The same goes for input layer

$$error_{input} = W^{\intercal}_{input,hidden} * error_{hidden} =$$
$$\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,m} \\ \multicolumn{5}{c}{\dotfill} \\ w_{n,1} & w_{n,2} & w_{n,3} & \cdots & w_{n,m} \end{pmatrix} * \begin{pmatrix} error_{hidden,1} \\ error_{hidden,2} \\ error_{hidden,3} \\ \dots\dots\dots \\ error_{hidden,k} \end{pmatrix}$$

4 Weight updating

For updating weights we should minimize errors that our network produces. To achieve that we will use gradient descent method. It means that we will be minimizing an error step by step never reaching the minimum but having as good approximation as we want.

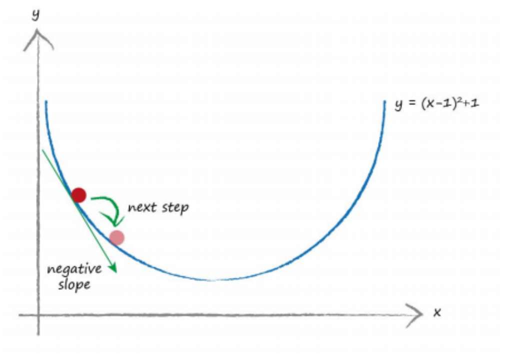

Figure 2: Negative gradient

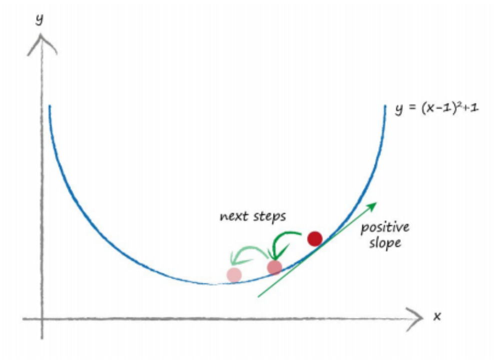

Figure 3: Positive gradient

If the gradient is negative we increase weight, if positive - decrease the weight

$$new\_w_{m,k} = old\_w_{m,k} - learning\_rate * \frac{\partial E}{w_{m,k}}$$

$\frac{\partial E}{w_{m,k}}$ - the slope of the error function that we want to descend towards the minimum.
$E = (target_{output} - final_{output})^2$- we square the error because it

- Simplifies the algebra needed to work out the slope for gradient descent
- Makes error function smooth and continuous - no gaps or abrupt jumps
- Makes the gradient get smaller near the minimum - minimizes chances of overshooting

$$\frac{\partial E}{\partial w_{m,k}} = \frac{\partial E}{\partial o_k} * \frac{\partial o_k}{\partial w_{m,k}} = \frac{\partial (t_k - o_k)^2}{\partial o_k} * \frac{\partial o_k}{\partial w_{m,k}} = -2(t_k - o_k) * \frac{\partial o_k}{\partial w_{m,k}}$$

As $o_k = sigmoid(input\_data)$ we should know the derivative of $sigmoid$ function

$$\frac{dsigmoid(x)}{dx} = \frac{1}{1+e^{-x}}\frac{d}{dx} = (1+e^{-x})^{-1}\frac{d}{dx} =$$
$$-\frac{\frac{de^{-x}}{dx}}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} =$$

Now we try to simplify the expression

$$\frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\frac{1}{1+e^{-x}}\right) = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right) =$$
$$sigmoid(x) * (1 - sigmoid(x))$$

As $input\_data = \sum_{i=1}^{m} w_{i,k} * output_{hidden,i}$ a complex function, we should also find its derivative

$$\frac{\partial \sum_{i=1}^{m} w_{i,k} * output_{hidden,i}}{w_{m,k}} = output_{hidden,m}$$

Summing everything up we get

$$\frac{\partial o_k}{\partial w_{m,k}} = \frac{\partial sigmoid(output_{input})}{\partial w_{m,k}} =$$
$$sigmoid(output_{input}) * (1 - sigmoid(output_{input})) * output_{hidden,m}$$

Now we can write down full gradient descent expression

$$\frac{\partial E}{\partial w_{m,k}} = -2(t_k - o_k) * sigmoid(output_{input}) * (1 - sigmoid(output_{input})) * output_{hidden,m}$$

We can also get rid of 2 at the front because we are only interested in the direction of the slope of the error function. We can do that as long as we are consistent with the constant 2

$$\frac{\partial E}{\partial w_{m,k}} = -(t_k - o_k) * sigmoid(output_{input}) * (1 - sigmoid(output_{input})) * output_{hidden,m}$$

So if we want to change the weights between a and b $a \to b$ we should

- Find errors in b - e_b
- Find *sigmoid* function value of b_{input} - $sigmoid(b_{input})$
- Know the output values of a - a_{output}

$$\frac{\partial E}{\partial w_{a,b}} = -e_b * S(b_{input}) * (1 - S(b_{input})) * a_{output}$$

We will now represent this equation in matrix form

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \cdots & \Delta w_{i,1} \\ \Delta w_{1,2} & \Delta w_{2,2} & \cdots & \Delta w_{i,2} \\ \cdots & \cdots & \cdots & \cdots \\ \Delta w_{1,j} & \Delta w_{2,j} & \cdots & \Delta w_{k,j} \end{pmatrix} = \begin{pmatrix} e_1 * s_1 * (1 - s_1) \\ e_2 * s_2 * (1 - s_2) \\ \cdots \\ e_j * s_j * (1 - s_j) \end{pmatrix} * \begin{pmatrix} o_1 & o_2 & \cdots & o_i \end{pmatrix}$$

So the final equation is

$$\frac{\partial E}{\partial w_{a,b}} = -e_b * S(b_{input}) * (1 - S(b_{input})) * a_{output}^{\intercal} \tag{3}$$

5 Preparing data

5.1 Inputs

Very big inputs or very small inputs will slow the learning (saturate a neural network) because of small gradient. We will calibrate values in range $[0.01, 1.0]$ for effective learning. Lower bound is not zero because this would kill the learning completely as multiply by a_{output} in equation 3. Another possible range is $[-1, +1]$

5.2 Target values

Sigmoid function can produce values in range $(0.0, 1.0)$ so there is no sense in setting target values higher or lower than those values. It is not good to set target values to 0 or 1 as function only asymptotically approaches those ones. We will calibrate target values in range $[0.01, 0.99]$

5.3 Random initial weights

Initial weights can't be constants or zeros: former would update all weights equally (kill network's ability to learn), latter would kill the input signal and weight update function wouldn't work properly.
Good choice for initial weights to be in range $\left[-\frac{1}{\sqrt{incoming\ links}}, +\frac{1}{\sqrt{incoming\ links}}\right]$ To generate numbers in this range we will exploit a normal distribution with mean *zero* and standard deviation of $\frac{1}{\sqrt{incoming\ links}}$

6 Improvements

6.1 Number of epochs

We can make our neural network train multiple times to help those weights to minimize their errors. It is bad to do too many training cycle because a network will get used to this particular data (this is called overfitting).
The best way to choose number of epochs is to experiment with different numbers (for example, in range $[1, 30]$) and watch the performance of the network.

6.2 Learning rate

- The best way to choose the learning rate is to experiment
- If number of epochs increases then it is better to lower the learning rate

6.3 Changing network shape

Our network has three layers: input, hidden and output. The size of input layer is fixed, because of images that we pass to our network have fixed size ($28x28$ pixels). The size of output layer is also fixed as there are 10 possible answers. What we can change is the number of nodes in hidden layer. If we set number of nodes in hidden layer too low our network wouldn't be able to recognize patterns in images as its **learning capacity** would be overwhelmed. If the number would be too big it would take too many epochs to train such network and thus too much time. The best way too choose number of nodes in hidden layer is to experiment with numbers and find the balance between performance and speed.

6.4 Extending existing training data

We can extend an existing training data by rotating images clockwise and counterclockwise. It is better not to rotate too much because some images may already be rotated (as different people write numbers differently) and so our network would train to recognize not what we would like it to recognize. A good choice is to rotate the images by $\pm 10°$

7 Backwards query

We can get inside the mind of our neural network to see how it sees the numbers. This can be achieved by backwards querying.
Suppose that layer a connected to next layer b - $a \to b$. Our task is to get ath input using bth output. *inverse_sigmoid* is called *logit*. We should follow these steps

1. $b_{in} = logit(b_{out}^{\intercal})$
2. $a_{out} = W_{a,b}^{\intercal} \cdot b_{in}$
3. Scale a_{out} to be in range of *logit* possible inputs, which is $(0, 1)$. We will scale a_{out} to be in range $(0.01, 0.99)$
4. $a_{in} = logit(a_{out})$

$logit(y)$ equals

$$
\begin{aligned}
&y = \frac{1}{1 + e^{-x}} \\
&1 + e^{-x} = \frac{1}{y} \\
&e^{-x} = \frac{1 - y}{y} \\
&-x = ln(\frac{1 - y}{y}) \\
&x = ln(\frac{y}{1 - y}) \\
&logit(y) = ln(\frac{y}{1 - y})
\end{aligned}
$$

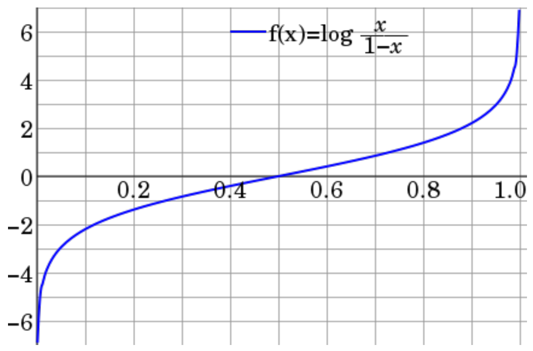

Figure 4: *logit* function

The first step is

$$b_{in} = \begin{pmatrix} logit(b_{out,1}) & logit(b_{out,2}) & \cdots & \end{pmatrix}^{\intercal} = \begin{pmatrix} logit(b_{out,1}) \\ logit(b_{out,2}) \\ \cdots \end{pmatrix}$$

The second step is

$$a_{out} = \begin{pmatrix} w_{a_1,b_1} & w_{a_2,b_1} & \cdots & \\ w_{a_1,b_2} & w_{a_2,b_2} & \cdots & \\ \multicolumn{4}{l}{\ldots\ldots\ldots\ldots\ldots\ldots} \end{pmatrix}^{\intercal} \cdot \begin{pmatrix} b_{in,1} \\ b_{in,2} \\ \ldots \end{pmatrix} = \begin{pmatrix} w_{a_1,b_1} & w_{a_1,b_2} & \cdots & \\ w_{a_2,b_1} & w_{a_2,b_2} & \cdots & \\ \multicolumn{4}{l}{\ldots\ldots\ldots\ldots\ldots\ldots} \end{pmatrix} \cdot \begin{pmatrix} b_{in,1} \\ b_{in,2} \\ \ldots \end{pmatrix}$$

The third step is

$$\begin{aligned} a_{out} &= a_{out} - min\_value(a_{out}) \\ a_{out} &= \frac{a_{out}}{max\_value(a_{out})} \\ a_{out} &= a_{out} * 0.98 \\ a_{out} &= a_{out} + 0.01 \end{aligned}$$

The fourth step is

$$a_{in} = \begin{pmatrix} logit(a_{out,1}) \\ logit(a_{out,2}) \\ \ldots\ldots\ldots \end{pmatrix}$$