# Feed Forward and Backprop

Aziz Temirkhanov

LAMBDA lab
Faculty of Computer Science
Higher School of Economics

March 3, 2024

# Outline

# Introduction

# Empirical Risk Minimization

Usually, when solving ML problems, one is seeking for solution to ERM
**ERM:**

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}} = \mathcal{L}(\theta; x, y) \tag{1}$$

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(g_{\theta}(x_i), y_i) \tag{2}$$

# Networks

**Assume, we have:**

- $x \in \mathbb{R}^d$ - data features
- $y \in \mathbb{R}$ - target
- $\theta \in \Theta$ or $w \in W$ - network parameters
- $f_\theta(x) = x_1 \times \theta_1 + x_2 \times \theta_2 + ... + x_n \times \theta_n$ - a network parameterized by $\theta$
- $\{(x_i, y_i)\}_{i=1}^{\ell}$ - training set
- $\mathcal{L}(\hat{y}, y)$ - loss function

To minimize empirical risk one can use gradient descent (usually, its stochastic version). Thus, the loss function (and f itself, of course) should be differentiable

# Backpropagation

**Q: How to compute $\nabla(L)$?**

**A: Use chain rule!**

- $y = f(g(x)) \implies \frac{dy}{dx} = \frac{df}{dg} \times \frac{dg}{dx}$
- $y = f(g_1(x), g_2(x), ..., g_n(x)) \implies$

$$\frac{dy}{dx} = \sum_{i=1}^{n} \frac{df}{dg_i} \times \frac{dg_i}{dx}$$

# Fully-Connected Networks

# Fully-Connected Netwroks

**Now, let:**

$x \in \mathbb{R}^{d_0}, \qquad f_\theta(x) = \langle x, \theta \rangle = \theta^T \times x, \theta \in \mathbb{R}^{d_0}$

Where $x$ - is a data point, and $f_\theta(x)$ - a linear function, or, more general, a neural network.

**Then, define: $\{z_i\}_{i=1}^n$ - logit, or vector of outputs, where $n$ is a number of layers**

- $z_1 = \theta_1 x, \qquad \theta_1 \in \mathbb{R}^{d_0 \times d_1}$
- $z_2 = \theta_2 z_1, \qquad \theta_2 \in \mathbb{R}^{d_0 \times d_2}$
- $z_n = \theta_n z_{n-1}, \qquad \theta_n \in \mathbb{R}^{d_0 \times d_n}$

# Fully-Connected Netwroks

In other words, **logit** - is the intermediate output vector that every layer of the network yields. The last logit vector - $z_n$ - feeds to the last activation layer.

So, while being a linear combination of the layers, Fully-Connected Networks yield a factor of outputs - **logit** $z_n$:

$$z_n = \theta_n \theta_{n-1} ... \theta_2 \theta_1 x, \quad \theta \in \mathbb{R}^{d_{n0}},$$

– still a linear function. Thus, might not be able to fit some arbitrary complex (in $\mathbb{R}$) functions.

# Fully-Connected Networks

Let's add non-linearity!

**activation function**

$\sigma(\cdot) : \mathbb{R} \mapsto \mathbb{R}$

Now, one can re-define $z_i$ as:

$$z_i = \sigma(z_{i-1}\theta_i + b_i)$$

$b_i \in \mathbb{R}^{d^i}$ - a bias vector to turn linear transform to affine

# Fully-Connected Networks

**To sum up:**

### linear layer

$f(x; \theta, b) = \theta x + b$ or $f(x; W, b) = Wx + b$

### hidden(latent) representation or logit

$z_i = (z_i^1, ..., z_i^{d_i})$

### non-linearity

$\sigma(\cdot) = \sigma(z_i)$

# Activations

# Activation functions

## Sigmoid

$\sigma(x) = \frac{1}{1+\exp^{-x}}$ $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

## tanh

$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ $tanh'(x) = 1 - tanh^2(x)$

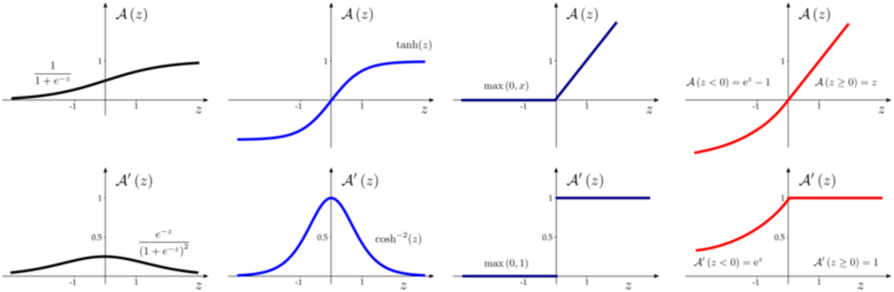
## ReLU

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

## Leaky ReLU

$LeakyReLU(x) = \max(\alpha x, x)$

# Activation functions



# Classification

# Classification

Consider a multi-class classification problem:

$$\{(x_i, y_i)\}_{i=1}^{\ell}, \quad y_i \in \{1, \dots, C\}, \text{where } C \text{ is the number of classes}$$

Then, ideally, we would like to solve this optimization problem:

$$\min_{\theta} \mathcal{L}(f(\theta; x_i), y_i) = \frac{1}{n} \sum_{i=1}^{\ell} [f(\theta; x_i) \neq y_i] \tag{3}$$

Here,

$$f(\theta; x_i) \neq y_i \tag{4}$$

– is the indicator function, that equals 1 if the condition holds, and 0 otherwise. One can already notice a problem with optimization tasks formulated this way.

# Softmax

**And that being: the derivative of indicator function either does not exist or equals to zero.**

So, instead of predicting the class label, a Neural Network can output a probability vector, such that:

$$\mathrm{p} = \begin{pmatrix} p(\hat{y}_i = 1) \\ p(\hat{y}_i = 2) \\ \vdots \\ p(\hat{y}_i = C) \end{pmatrix} \quad (5)$$

Here, each $i$-th element of the vector shows the probability that the input belongs to the $i$-th class. For example, if the $i$-th value of the vector $p$ is the largest one, then, w.p. $p_i$ input $x$ is derived from the class $i$. Recall, that the set of all classes is $\{1, ..., C\}$

# Classification

Now, recall what a **logit** $z_i$ is:

$$z_i = \sigma(\theta_i z_{i-1} + b_i) \tag{6}$$

And the network' output:

$$z_n = \theta_n \times z_{n-1} + b_n, \quad z_n \in \mathbb{R}^{d_n}, \quad d_n = C \tag{7}$$

We want to map this vector of some real-valued numbers to a probability vector, such that:

$$p = \{p_i\}_{i=1}^{C}, \quad p_i \geq 0, \quad \sum_{i=1}^{C} p_i = 1 \tag{8}$$

# Softmax

First of all, assume that the output vector $z_n$ has the number of elements as the number of classes:

$$z_n = \{z_1, z_2, ..., z_C\} \tag{9}$$

Then, define our mapping $\sigma(\cdot) : \mathbb{R}^C \mapsto (0, 1)^C$:

$$p_i = \frac{\exp z_i}{\sum_{j=1}^{C} \exp z_j} \tag{10}$$

In simple words, take each element $z_i$ of the vector $z$, divide it by the sum of every element in vector $z$, and assign this value to the $i$-th element of the vector $p$.

# Softmax

$y$ – target $\in \{1, ..., C\}$, or label of the **correct** class
Minimize:

$$-\sum_{k=1}^{C}[k = y] \log p_k \mapsto \min_{\theta} \tag{11}$$

Here, $p$ is our vector of probabilities, and $p_k$ is the $k$-th element of that vector, representing a probability of the object $x$ being derived from the class $k$. Notice, that indicator function $[k = y]$ is equal to 1 only once throughout this sum loop. Thus, we can simplify this equation to:

$$-\log p_k \mapsto \min_{\theta}, \quad \text{where } k = y \tag{12}$$

Which basically means the probability of the correct class. So, by minimizing this probability of the correct class with the minus sign, or in other words **negative log likelihood**, we maximize the likelihood itself.

# Softmax summary

Softmax is element-wise:

$$p(y = k) = p_k = softmax(z_k) = \frac{\exp z_k}{\sum_{j=1}^{C} \exp z_j} \tag{13}$$

Let us now use the **maximum likelihood estimation** to train the network:

$$-\sum_{k=1}^{C} [k = y] \log p_k \mapsto \min_{\theta} \tag{14}$$

Simplify and obtain negative log-likelihood to turn our MLE task to an optimization task:

$$-\log p_y \mapsto \min_{\theta} \quad \text{where } y \in \{1, \ldots, C\} \tag{15}$$

# Log Softmax

Let's now formulate the loss function for every observation in our sample:

$$\{x_i, y_i\}_i^{\ell}, \quad y \in \{1, ..., C\}$$

$$L = -\frac{1}{\ell}\sum_{i=1}^{\ell}\sum_{k=1}^{C}[y_i = k]\log p_k^{(i)} \mapsto \min_{\theta} \tag{16}$$

It is basically the same Negative Log-Likelihood loss function but now summarized over every observation in the dataset. By minimizing this loss function, we solve the Empirical Risk Minimization task and train our network!

# Logsoftmax

Our loss function

$$NLLLoss(softmax(z), y) \mapsto \min \tag{17}$$

is numerically unstable. Instead of calculating softmax, and then applying logarithm to is (inside the NLL), it is better to calculate log-softmax element-wise instead:

$$logsoftmax(z_k) = \log \frac{\exp z_k}{\sum_{j=1}^{C} \exp z_j} = \log \exp z_k - \log \sum_{j=1}^{C} \exp z_j \tag{18}$$

Remember, that we are dealing with **logit** $z$, implying the very last logit - $z_n$, from the last linear layer. Since we already know, that softmax and NLL are being applied to the last logit, we remove the subscription of it. Now, every subscription like $k, i, j$ representing the $k$-th, $i$-th, or $j$-th element of the vector $z$.

# Cross-Entropy

So, log softmax is now more numerically stable.

$$logsoftmax(z_k) = z_k - \log \sum_{j=1}^{C} \exp z_j \tag{19}$$

But to ensure that we need to get rid of really big numbers that can be derived from exponent. The solution is simple: just subtract the maximum value of our vector $z$ **element-wise**:

$$logsoftmax(z_k) = z_k - \max_i z_i - \log \sum_{j=1}^{C} \exp z_j - \max_i z_i \ , \tag{20}$$

where $i \in \{1, ..., \ell\}$, and $\ell$ is the number of observations (or the size of the dataset)

# Cross-Entropy

Now we've come close to the Cross-Entropy!
And while Negative Log-Likelihood is expecting a vector $p$ (in other words, $softmax(z)$) as an input, log softmax is doing all of it combined. And log softmax is a Cross-Entropy Loss:

$$CrossEntropyLoss(z, y) = -\sum_{i=1}^{C} q_i \log p_i, \tag{21}$$

where $q_i = [i = y]$
In conclusion, you need to feed the output $z$ of your network $f_\theta$ to the cross-entropy loss, along with the correct label $y$. Optimizing such loss will fit your network. In order to interpret the result after training, feed the network's output to the softmax, and then assign a text label to each element of vector $p$.


Thank you