1 What is neural network?

Neural network is any Directed Acyclic Graph (DAG) in which every vertex i has the following attributes.

1. Set of previous vertices – $\mathscr{P}_i$.
2. Set of next vertices – $\mathscr{N}_i$.
3. Parametrized tensor function of the form

$$\boldsymbol{F}^{(i)}:\mathbb{R}^{\left(n_1^{(1)},\dots,n_{k_1}^{(1)}\right)}\times\dots\times\mathbb{R}^{\left(n_1^{(p)},\dots,n_{k_p}^{(p)}\right)}\times\Theta\mapsto\mathbb{R}^{(m_1,\dots,m_l)}$$

The function takes p tensor arguments of dimensions $k_1,\dots,k_p$ respectively (input tensor q of dimension k_q has $n_r^{(q)}$ elements along the r axis) and parameters $\boldsymbol{\theta}^{(i)}\in\Theta$ and returns a tensor of dimension l. Obviously it must satisfy $p=|\mathscr{P}_i|$ and the tensors returned by the parent nodes must have appropriate shapes.

4. The gradient functions of the function $\boldsymbol{F}^{(i)}$ w.r.t. to the parameters and w.r.t. all the inputs, that is for all $j\in\mathscr{P}_i$ we have a gradient functions

$$\frac{\partial F_\beta^{(i)}}{\partial\theta_\alpha^{(i)}}\,,\quad\frac{\partial F_\beta^{(i)}}{\partial F_\alpha^{(j)}}\,,$$

where α, β are the suitable multi-indices.

2 Loss functions

Training of a neural network consists of changing the parameters $\boldsymbol{\theta}^{(i)}$ of the nodes in such a way as to make the network perform the given task. The task is specified by a training set $\mathcal{X}$ which contains the "blueprint answers" of the network. To train the network we introduce the quantitative measure of networks performance on the dataset which implicitly (through the outputs of the network) depends on the parameters of the network (here collectively denoted by $\boldsymbol{\theta}$) $L(\mathcal{X},\boldsymbol{\theta})$. Training can be then phrased as an optimization problem of the form

$$\boldsymbol{\theta}^*=\arg\min_{\boldsymbol{\theta}}L(\mathcal{X},\boldsymbol{\theta})$$

for a fixed training set $\mathcal{X}$.

There is no single established way of constructing loss functions. One of the more motivated approaches is based on the maximum likelihood criterion. The idea is that we model our data using some parametrized statistical model and express the parameters of this model as an output of a neural network. The loss function is then taken to be the *negated log-likelihood function.* In this manner one can derive the most common loss functions.

2.1 Mean Squared Error

$$\boxed{L(\mathcal{X},\boldsymbol{\theta})=\frac{1}{n}\sum_\alpha\left[y_\alpha-\Phi_\alpha(\boldsymbol{X};\boldsymbol{\theta})\right]^2\,,}$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{y}$ is the corresponding tensor of n scalar continuous outputs for each feature-vector (so called target) and $\boldsymbol{\Phi}$ denotes the neural network.

2.2 (Binary) Cross Entropy

$$\boxed{L(\mathcal{X},\boldsymbol{\theta})=\frac{1}{n}\sum_\alpha\left[t_\alpha\log\pi_\alpha+(1-t_\alpha)\log(1-\pi_\alpha)\right]}$$

$$\boldsymbol{\pi}=\sigma\left(\boldsymbol{\Phi}(\boldsymbol{X};\boldsymbol{\theta})\right)\,,\quad\sigma(z)=\frac{1}{1+\mathrm{e}^{-z}}\,,$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{t}$ is the corresponding tensor of n binary (i.e. 0 or 1) values denoting the class for each feature-vector, σ is the logistic function, $\boldsymbol{\Phi}$ denotes the neural network and $\boldsymbol{\pi}$ is a tensor of the same shape as $\boldsymbol{t}$ which contains the probabilities of the positive class.

2.3 Cross Entropy

$$\boxed{L(\mathcal{X},\boldsymbol{\theta})=\frac{1}{n}\sum_\alpha\sum_\beta t_{\alpha\beta}\log\pi_{\alpha\beta}}$$

$$\boldsymbol{\pi}=\boldsymbol{\sigma}\left(\boldsymbol{\Phi}(\boldsymbol{X};\boldsymbol{\theta})\right)\,,\quad\sigma_{\alpha'\alpha}(\boldsymbol{z})=\frac{\exp z_{\alpha'\alpha}}{\sum_\beta\exp z_{\alpha'\beta}}\,,$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{t}$ is the corresponding thensor which can be interpreted as a stack of n 1-D one-hot-vectors residing in the last dimension of $\boldsymbol{t}$ encoding the correct class, $\boldsymbol{\sigma}$ is the soft-max function which given the stack of 1-D vectors independently normalizes each of them so that the entries are non-negative and sum to 1 and $\boldsymbol{\Phi}$ denotes the neural network.

3 Forward propagation

Let $\boldsymbol{v}^{(i)}$ be the (tensor) value of the function $\boldsymbol{F}^{(i)}$. To propagate the (tensor) inputs to the network and get the output we use the following recursive equation

$$\boxed{\boldsymbol{v}^{(i)}=\boldsymbol{F}^{(i)}\left[\left(\boldsymbol{v}^{(j)}\right)_{j\in\mathscr{P}_i};\boldsymbol{\theta}^{(i)}\right]}$$

and visit the nodes in the *topological order* as this guarantees that we visit every node exactly once. We assume here that nodes $\boldsymbol{v}^{(i)}$ such that $\mathscr{P}_i = \varnothing$ are the inputs to the network and nodes $\boldsymbol{v}^{(i)}$ such that $\mathscr{N}_i = \varnothing$ are the output of the network.

4 Backward propagation

Let L be the loss function. In order to compute the derivatives $\partial_{\boldsymbol{\theta}^{(i)}} L$ we use the following recursive equations

$$\boxed{\begin{aligned}\frac{\partial L}{\partial \theta_\alpha^{(i)}} &= \sum_\beta \frac{\partial L}{\partial F_\beta^{(i)}} \frac{\partial F_\beta^{(i)}}{\partial \theta_\alpha^{(i)}} \\ \frac{\partial L}{\partial F_\alpha^{(i)}} &= \sum_{j \in \mathscr{N}_i} \sum_\beta \frac{\partial L}{\partial F_\beta^{(j)}} \frac{\partial F_\beta^{(j)}}{\partial F_\alpha^{(i)}}\end{aligned}}$$

where α, β are the suitable multi-indices. We visit nodes in the *reversed topological order* and compute and store the values of loss function derivatives. All derivatives are computed for the current values of $\boldsymbol{v}^{(i)}$ and $\boldsymbol{\theta}^{(i)}$, therefore before backward propagation one must perform forward propagation to compute values $\boldsymbol{v}^{(i)}$.

5 Stochastic Gradient Descent

The standard optimization method used to train neural networks is the Stochastic Gradient Descent, which is an iterative, gradient-based algorithm in which in every step t we update the parameters $\boldsymbol{\theta}^{(i)}$ utilizing the gradient information. Let $\boldsymbol{\theta}^{(i,t)}$ denote the value of parameters $\boldsymbol{\theta}^{(i)}$ at step t and let $\boldsymbol{v}^{(i,t)}$ be the values of the function $\boldsymbol{F}^{(i)}$ at step t. In each step we take a batch $\mathcal{X}$ of training data, perform forward propagation to compute values $\boldsymbol{v}^{(i,t)}$ and the value of the loss function $L(\mathcal{X}, \boldsymbol{\theta}^{(t)})$, next perform backward propagation to compute the values of gradients $\boldsymbol{g}^{(i,t)} = \partial_{\boldsymbol{\theta}^{(i)}} L\left(\mathcal{X}, \boldsymbol{\theta}^{(t)}\right)$ and afterwards we update the parameters according to

$$\boxed{\theta_\alpha^{(i,t+1)} = \theta_\alpha^{(i,t)} - \eta g_\alpha^{(i,t)}}$$

where η is the learning rate.

5.1 Momentum

The problem with vanilla SGD is that it gets stuck in the local minima. To overcome this problem one can take inspiration from simple physics. We first introduce velocity tensor $\boldsymbol{V}$ with the following update rule

$$\boxed{V_\alpha^{(i,t+1)} = \gamma V_\alpha^{(i,t)} - \eta g_\alpha^{(i,t)}}$$

where $0 < \gamma < 1$ is the so called momentum term and update parameters using

$$\boxed{\theta_\alpha^{(i,t+1)} = \theta_\alpha^{(i,t)} + V_\alpha^{(i,t+1)}}$$

5.2 Adaptive Moment Estimation (Adam)

Another problem with vanilla SGD is that it uses the same learning rate for every scalar parameter. Adam algorithm solves this problem by introducing running averages with exponential forgetting of both the gradients and the second moments of the gradients.

$$\boxed{\begin{aligned}M_\alpha^{(i,t+1)} &= \beta_1 M_\alpha^{(i,t)} + (1-\beta_1) g_\alpha^{(i,t)} \\ V_\alpha^{(i,t+1)} &= \beta_2 V_\alpha^{(i,t)} + (1-\beta_2)\left(g_\alpha^{(i,t)}\right)^2 \\ \tilde{M}_\alpha^{(i,t+1)} &= \frac{M_\alpha^{(i,t+1)}}{1-\beta_1^t} \\ \tilde{V}_\alpha^{(i,t+1)} &= \frac{V_\alpha^{(i,t+1)}}{1-\beta_2^t} \\ \theta_\alpha^{(i,t+1)} &= \theta_\alpha^{(i,t)} - \eta \frac{\tilde{M}_\alpha^{(i,t+1)}}{\sqrt{\tilde{V}_\alpha^{(i,t+1)}} + \epsilon}\end{aligned}}$$

where $\epsilon \simeq 10^{-8}$ is used to prevent division by 0, η is the learning rate and β_1 and β_2 are the forgetting factors typically set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Additionally popular choice for η is $\eta = 3 \cdot 10^{-4}$.

6 Regularization and initialization

7 Architectures

7.1 MLP

7.2 CNN

7.3 RBM

7.4 Autoencoder

7.5 VAE

7.6 GAN

7.7 DDPM

7.8 Transformer

8 Interpretability