Lecture Notes

The world of the neuron: from the perceptron to deep architectures

Financial Machine Learning
April 10, 2023

Oualid Missaoui
AI4Quants Land

Agenda

1. **Introduction**
2. **Feed-forward Neural Network**
3. **Learning neural networks: loss function**
   - Regression
   - Classification
   - Backpropagation and dynamic programming
4. **Issues of the back-propagation**
5. **References**

Perceptron

- Starting point: linearly separable two-class (ω_1, ω_2) classification task
- $(\boldsymbol{x}_i, y_i), i = 1, \cdots, N$, with $y_i \in \{-1, +1\}$, $\boldsymbol{x}_i \in \mathbb{R}^D$
- We assume there is a hyperplane

$$\boldsymbol{\theta}_*^T \boldsymbol{x} = 0$$

such that

$$\begin{aligned} \boldsymbol{\theta}_*^T \boldsymbol{x} &> 0, \quad \text{if} \quad \boldsymbol{x} \in \omega_1, \\ \boldsymbol{\theta}_*^T \boldsymbol{x} &< 0, \quad \text{if} \quad \boldsymbol{x} \in \omega_2, \end{aligned}$$

The bias term of the hyperplane has been absorbed in $\boldsymbol{\theta}_*$.

- Goal: developing an algorithm that iteratively computes a hyperplane that classifies correctly all the patterns from both classes.

Perceptron cost function

- Let the available estimate at the current iteration step of the unknown parameters be $\boldsymbol{\theta}$. Then there are two possibilities:
  - all points are classified correctly and a solution has been obtained, or
  - $\boldsymbol{\theta}$ classifies correctly some of the points and the rest are misclassified
- Let $\mathcal{Y}$ be the set of all misclassified samples
- The perceptron cost is defined as

$$\mathcal{J}(\boldsymbol{\theta}) = -\sum_{i:\boldsymbol{x}_i \in \mathcal{Y}} y_i \boldsymbol{\theta}^T \boldsymbol{x}_i \quad (1)$$

$$= \Big(-\sum_{i:\boldsymbol{x}_i \in \mathcal{Y}} y_i \boldsymbol{x}_i^T \Big) \boldsymbol{\theta} \quad (2)$$

Perceptron Algorithm

- Starting from an arbitrary point $\boldsymbol{\theta}^{(0)}$, the following iterative (batch) update

$$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} + \mu_t \sum_{i:\boldsymbol{x}_i \in \mathcal{Y}} y_i \boldsymbol{x}_i \tag{3}$$

converges after a finite number of steps, where μ_t is the user-defined step size.

- Stochastic version: considering one sample per iteration in a cyclic fashion

$$\boldsymbol{\theta}^{(t)} = \begin{cases} \boldsymbol{\theta}^{(t-1)} + \mu_t y_i \boldsymbol{x}_i, & \text{if } \boldsymbol{x}_i \text{ is misclassified by } \boldsymbol{\theta}^{(t-1)}; \\ \boldsymbol{\theta}^{(t-1)}, & \text{otherwise.} \end{cases}$$

Once all samples have been considered, we say that one epoch has been completed. If no convergence has been attained, all samples are reconsidered in a second epoch and so on.

McCulloch-Pitts Neuron

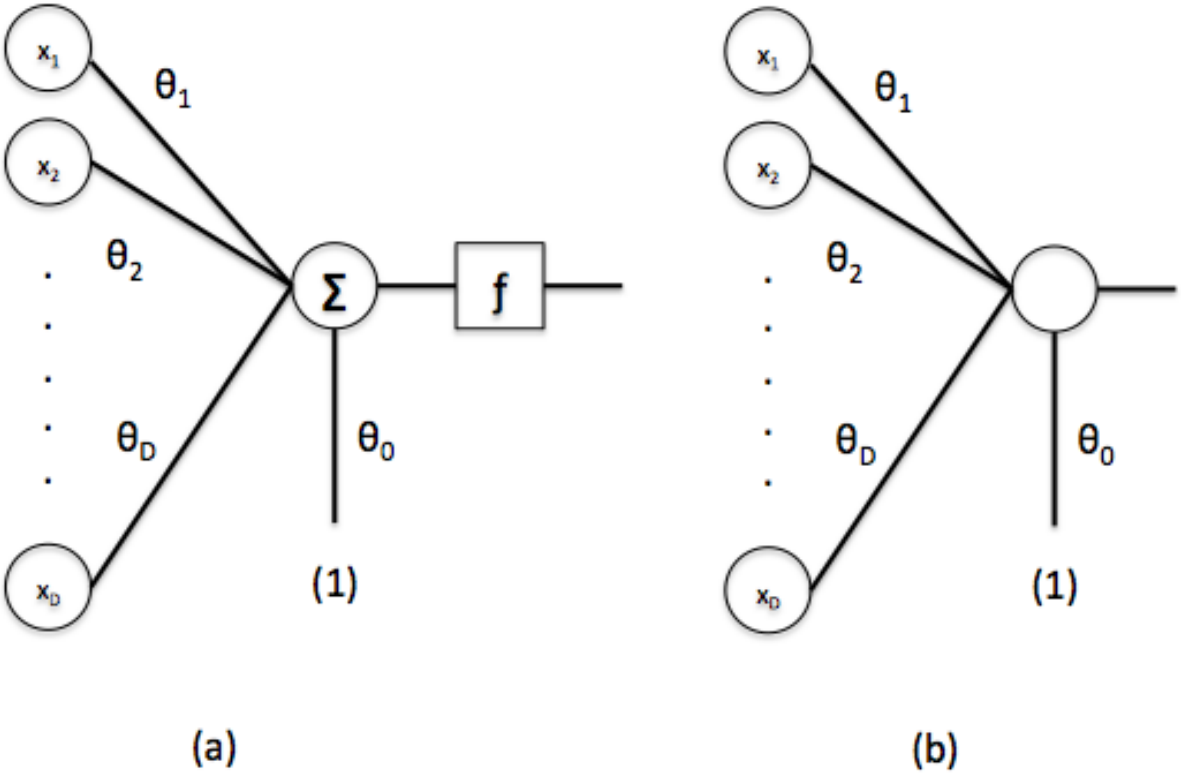


Figure: Schematic of a single hidden layer, feed-forward neural network (also called two layer feed-forward neural network).

The activation function is the Heaviside one,

$$f(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases}$$

Feed-forward Neural Network

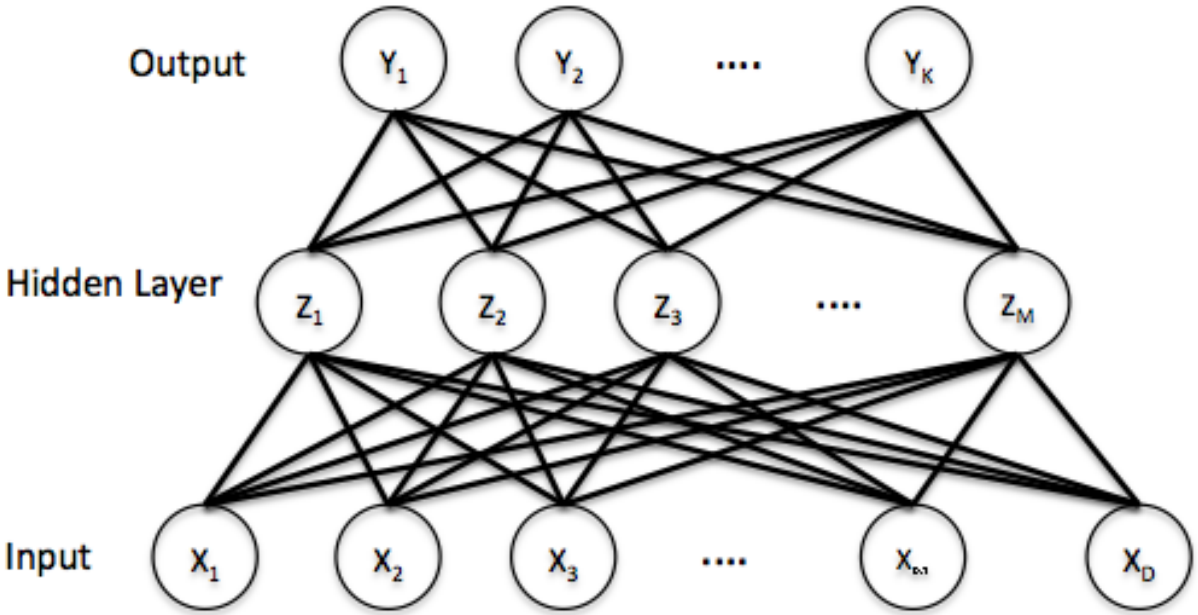


Figure: Schematic of a single hidden layer, feed-forward neural network (also called two layer feed-forward neural network).

Formal setup

- Let $(\boldsymbol{x}_i, y_i), i = 1, \cdots, N$, be the set of training samples.
- We assume that the network comprises of L layers; $L-1$ hidden and one output layers.
- Each layer consists of k_r , $r = 1, \cdots, L$ neurons, thus

$$\boldsymbol{y}_i = [y_{i1}, \cdots, y_{ik_L}]^T \in \mathbb{R}^{k_L}$$

- Let $\boldsymbol{\theta}_j^r$ denote the synaptic weights associated with the jth neuron and the rth layer, with $j = 1, \cdots, k_r$ and $r = 1, 2, \cdots, L$, where the bias term is included in $\boldsymbol{\theta}_j^r$, that is,

$$\boldsymbol{\theta}_j^r = [\theta_{j0}^r, \theta_{j1}^r, \cdots, \theta_{jk_{r-1}}^r]^T$$

- the synaptic weights link the respective neuron to all neurons in layer k_{r-1}

Feed-forward Neural Network

A standard network would be defined as:

$$\begin{aligned} z_{ij}^r &= \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_i^{r-1} \quad \text{Pre-activation} \\ \boldsymbol{y}_i^r &= \sigma(\boldsymbol{z}_i^r) \quad \text{Post-activation} \end{aligned}$$

Unrolling the function composition

$$\hat{f}(x) = \sigma(\theta\sigma(\theta \cdots \sigma(\theta x) \cdots))$$

σ is the activation function

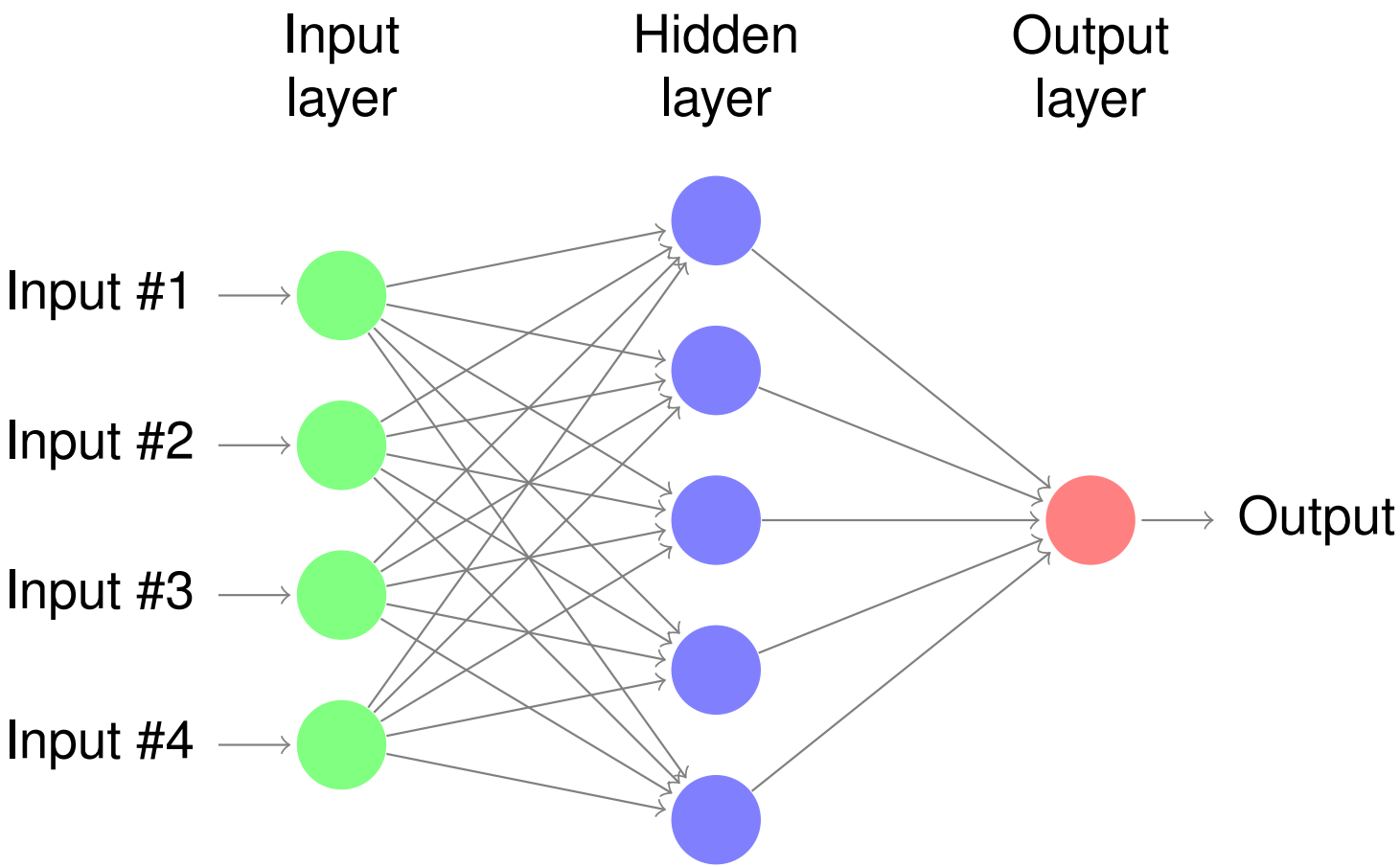
- for regression: the identity function
- for classification: the softmax function

$$\frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

Deep feed-forward neural networks

- We will refer to the number of layers in a network as the *depth* of the network
- Networks with up to three (two hidden) layers, are known as *shallow*
- Those with more than three are called *deep* networks
- Network with more than 10 hidden layers are considered very deep neural networks.

Feed-forward Neural Network



Activation functions

- step-function

$$\sigma(x) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases}$$

- Sigmoid (Fermi function)

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

- Hyperbolic tangent

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

- Rectified Linear unit (ReLu)

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

- Leaky rectified linear unit (Leaky ReLu)

$$\sigma(x) = \begin{cases} ax & \text{if } x \leq 0, \\ x & \text{otherwise.} \end{cases}$$

- Exponential linear units

$$\sigma(x) = \begin{cases} a \cdot (e^{x} - 1) & \text{if } x \leq 0, \\ x & \text{otherwise.} \end{cases}$$

Illustration of Activation functions

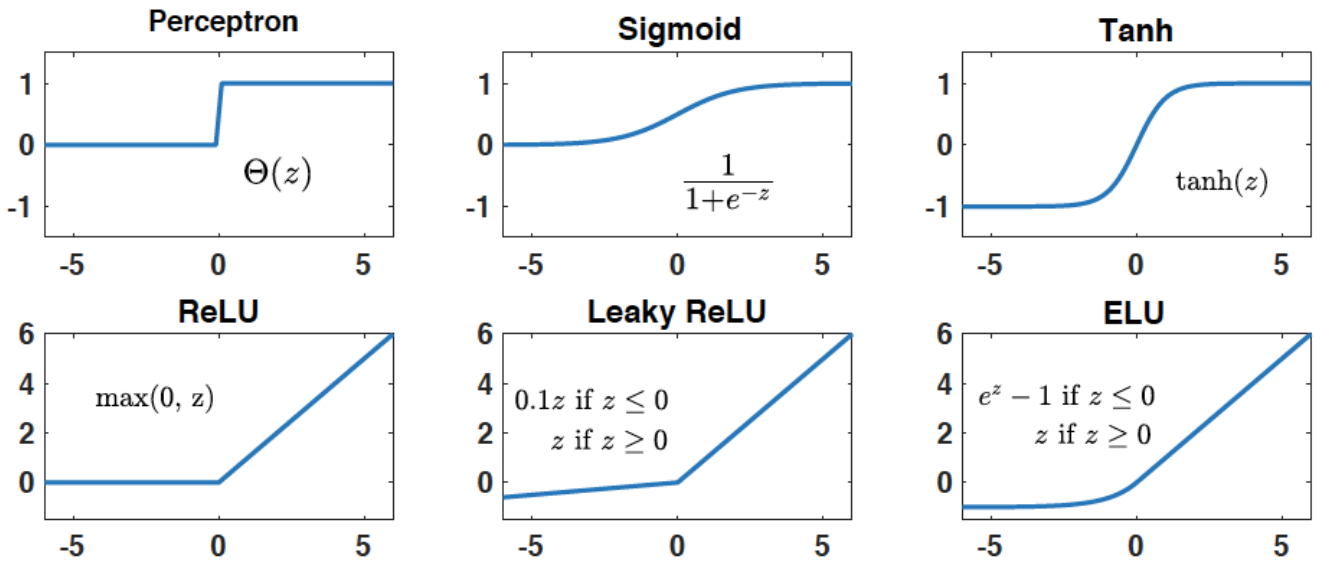


Figure: Possible non-linear activation functions for neurons.

Source: Pankaj Mehta, Ching-Hao Wang, Alexandre G. R. Day, and Clint Richardson *A high-bias, low-variance introduction to Machine Learning for physicists*

Neural Networks for regression: loss function

If the square error loss is adopted, we have

$$\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \mathcal{J}_i(\theta)$$

, where

$$\mathcal{J}_i(\theta) = \frac{1}{2} \sum_{k=1}^{k_L} (\hat{y}_{ik} - y_{ik})^2$$

Neural Networks for classification: loss function

For classification, as is the case for logistic regression, the cost/loss function can be defined as the log-likelihood function

$$\begin{aligned}
\mathcal{J}(\boldsymbol{\theta}) &= \log P(\mathcal{D}|\boldsymbol{\theta}) \\
&= \log \prod_{i=1}^{N} P(\boldsymbol{x}_i|\boldsymbol{\theta}) \\
&= \log \prod_{i=1}^{N} \sigma((\boldsymbol{\theta}^L)^T \boldsymbol{y}_i^{L-1})^{y_i} (1 - \sigma((\boldsymbol{\theta}^L)^T \boldsymbol{y}_i^{L-1}))^{1-y_i} \\
&= \sum_{i=1}^{N} y_i \log\left(\sigma((\boldsymbol{\theta}^L)^T \boldsymbol{y}_i^{L-1})\right) + (1 - y_i) \log\left(1 - \sigma((\boldsymbol{\theta}^L)^T \boldsymbol{y}_i^{L-1})\right)
\end{aligned}$$

The right hand side of the previous equation is known in statistics as the cross-entropy.

Formal setup

- Gradient descent scheme:

$$\boldsymbol{\theta}_j^r(t+1) = \boldsymbol{\theta}_j^r(t) + \Delta\boldsymbol{\theta}_j^r$$

where

$$\Delta\boldsymbol{\theta}_j^r = -\mu \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}_j^r}\bigg|_{\boldsymbol{\theta}_j^r(t)}$$

- If the square error loss is adopted, we have

$$\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \mathcal{J}_i(\theta)$$

, where

$$\mathcal{J}_i(\theta) = \frac{1}{2}\sum_{k=1}^{k_L}(\hat{y}_{ik} - y_{ik})^2$$

- For the neurons at the layer r, we denote its output by $\boldsymbol{y}_i^r = [1, y_{i1}^r, \cdots, y_{ik_r}^r]^T$
  - at the output layer, $r = L$, $y_{im}^L = \hat{y}_{im}$, $m = 1, 2, \cdots, k_L$
  - for $r = 1$, we have $y_{im}^0 = x_{im}$, $m = 1, 2, \cdots, k_0$; that is, y_{im}^0 are set equal to the input feature values.

Gradient of loss function

- Let z_{ij}^r denote the output of the linear combiner of the jth neuron in the rth layer

$$z_{ij}^r = \sum_{m=0}^{k_{r-1}} \theta_{jm}^r y_{im}^{r-1} = \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_i^{r-1}$$

and

$$\boldsymbol{z}_i^r = [z_{i1}^r, z_{i2}^r, \cdots, z_{ik_r}^r]^T$$

- It follows that

$$\boldsymbol{y}_i^r = \sigma(\boldsymbol{z}_i^r)$$

- We can now write that:

$$\frac{\partial \mathcal{J}_i}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial \mathcal{J}_i}{\partial z_{ij}^r} \frac{\partial z_{ij}^r}{\boldsymbol{\theta}_j^r} = \frac{\partial \mathcal{J}_i}{\partial z_{ij}^r} \boldsymbol{y}_i^{r-1}$$

- Let us now define

$$\delta_{ij}^r = \frac{\partial \mathcal{J}_i}{\partial z_{ij}^r}$$

- Then we have

$$\Delta \boldsymbol{\theta}_j^r = -\mu \sum_i^N \delta_{ij}^r \boldsymbol{y}_i^{r-1}, r = 1, 2, \cdots, L$$

Computation of δ_{ij}^r

Here is where the heart of the back-propagation algorithm beats.
For the computation of the gradients, δ_{ij}^r , one starts at the last layer $r = L$, and proceeds backwards toward $r = 1$ which justifies the name given to the algorithm.

- $r = L$, we have that $\delta_{ij}^L = \frac{\partial \mathcal{J}_i}{\partial z_{ij}^L}$.
  - For the squared error loss function,
    $J_n = \frac{1}{2} \sum_{k=1}^{k_L} \left(f(z_{ik}^L) - y_{ik} \right)^2$
  - Hence, $\delta_{ij}^L = (\hat{y}_{ij} - y_{ij}) f'(z_{ij}^L)$
- $r < L$: due to the successive dependence between the layers, the value of z_{ij}^{r-1} influences all the values $z_{ik}^r, k = 1, 2, \cdots, k_r$ of the next layer. Employing the chain rule for differentiation, we get
  - $\delta_{ij}^{r-1} = \frac{\partial \mathcal{J}_i}{\partial z_{ij}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{J}_i}{\partial z_{ik}^r} \frac{\partial z_{ik}^r}{\partial z_{ik}^{r-1}} = \sum_{k=1}^{k_r} \delta_{ik}^r \frac{\partial z_{ik}^r}{\partial z_{ik}^{r-1}}$
  - $\frac{\partial z_{ik}^r}{\partial z_{ik}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^r y_{im}^{r-1} \right)}{\partial z_{ik}^{r-1}} = \theta_{kj}^r f'(z_{ij}^{r-1})$ since $y_{im}^{r-1} = f(z_{ij}^{r-1})$
  - Thus

$$\delta_{ij}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{ik}^r \theta_{kj}^r \right) f'(z_{ij}^{r-1})$$

Alternative representation of δ_i^r

We can clearly see that δ_{ij}^{r-1} can be rewritten in a matrix format:

$$\delta_{ij}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{ik}^r \theta_{kj}^r \right) f'(z_{ij}^{r-1})$$

where

$$\boldsymbol{\delta}_i^{r-1} = \boldsymbol{\Sigma}'(\boldsymbol{z}_i^{r-1})(\boldsymbol{\theta}^r)^T \boldsymbol{\delta}_i^r$$

where $\boldsymbol{\Sigma}'(\boldsymbol{z}_i^{r-1})$ is a square matrix whose diagonal entries are the values $\sigma'(z_{ij}^{r-1})$, and whose off-diagnoal entires are zero.

Similarly, since $\delta_{ij}^L = (\hat{y}_{ij} - y_{ij}) f'(z_{ij}^L)$, we can rewrite the error at the final layer as:

$$\boldsymbol{\delta}_i^L = \boldsymbol{\Sigma}'(\boldsymbol{z}_i^L) \nabla_y \mathcal{J}$$

Combining the above, we get

$$\boldsymbol{\delta}_i^r = \boldsymbol{\Sigma}'(\boldsymbol{z}_i^r)(\boldsymbol{\theta}^{r+1})^T \cdots \boldsymbol{\Sigma}'(\boldsymbol{z}_i^{L-1})(\boldsymbol{\theta}^L)^T \boldsymbol{\Sigma}'(\boldsymbol{z}_i^L) \nabla_y \mathcal{J}$$

Naive/Brute force computation of the gradient

$$\delta_i^0 = \Sigma'(z_i^0)(\theta^1)^T \boxed{\Sigma'(z_i^1)(\theta^2)^T \cdots \Sigma'(z_i^{L-1})(\theta^L)^T \Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

$$\delta_i^1 = \boxed{\Sigma'(z_i^1)(\theta^2)^T \Sigma'(z_i^2)(\theta^3)^T \cdots \Sigma'(z_i^{L-1})(\theta^L)^T \Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

$$\vdots$$

$$\delta_i^r = \Sigma'(z_i^r)(\theta^{r+1})^T \boxed{\Sigma'(z_i^{r+1})(\theta^{r+2})^T \cdots \Sigma'(z_i^{L-1})(\theta^L)^T \Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

$$\delta_i^{r+1} = \boxed{\Sigma'(z_i^{r+1})(\theta^{r+2})^T \Sigma'(z_i^{r+2})(\theta^{r+3})^T \cdots \Sigma'(z_i^{L-1})(\theta^L)^T \Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

$$\vdots$$

$$\delta_i^{L-1} = \Sigma'(z_i^{L-1})(\theta^L)^T \boxed{\Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

$$\delta_i^L = \boxed{\Sigma'(z_i^L)\nabla_y \mathcal{J}}$$

Clearly, computing the gradients independently would have been very costly and inefficient due to the redundant operations.

Back-propagation algorithm

The following back-propagation algorithm computes the gradient of the loss function with respect of all the network parameters effeciently

Input : Set the corresponding activation for the input layer

Feedforward : For each hidden layer $2 \leq r \leq L$, compute $\boldsymbol{z}_i^r$

Output error : Compute the vector $\boldsymbol{\delta}_i^L = \boldsymbol{\Sigma}'(\boldsymbol{z}_i^L)\nabla_y \mathcal{J}$

Backpropagate the error : For each $r = L-1, L-2, \cdots, 2$, compute $\boldsymbol{\delta}_i^{r-1} = \boldsymbol{\Sigma}'(\boldsymbol{z}_i^{r-1})(\boldsymbol{\theta}^r)^T \boldsymbol{\delta}_i^r$ (hence the name bac-propagation)

Output : The gradient of the cost function is given by $\delta_{ij}^r \boldsymbol{y}_i^{r-1}$

Gradient descent and its generalization

In the simplest gradient descent (GD) algorithm, we update the parameters as follows. Initialize the parameters to some value $\boldsymbol{\theta}_0$ and iteratively update the parameters to the equation

$$\begin{aligned} \boldsymbol{v}_t &= \eta_t \nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t \end{aligned}$$

where $\nabla_\theta \mathcal{J}(\boldsymbol{\theta})$ is the gradient of $\mathcal{J}(\boldsymbol{\theta})$ w.r.t $\boldsymbol{\theta}$ and η_t is the learning rate that controls the magnitude of the step we would take in the direction of the gradient at time step t.

- For η_t sufficinetly small, the method will converge to a local minimum
- The smaller the η_t, the more steps we have to take to reach the local minimum
- If η_t is too large, we can overshoot the minimum and the algorithm becomes unstabe (it either oscillates or even move away from the minimum)

Understanding Gradient descent

In Newton's method, we choose the step $\boldsymbol{v}_t$ for the parameters in such a way as to minimize a second-order Taylor expansion to the energy function

$$\mathcal{J}(\boldsymbol{\theta}+\boldsymbol{v}) \approx \mathcal{J}(\boldsymbol{\theta}) + \nabla_\theta \mathcal{J}(\boldsymbol{\theta})\boldsymbol{v} + \frac{1}{2}\boldsymbol{v}^T \boldsymbol{H}(\theta)\boldsymbol{v}$$

where $\boldsymbol{H}(\theta)$ is the Hessian matrix of the second derivatives. Ideally, using the second order approximation, we can find $\boldsymbol{v}$ that makes $\mathcal{J}(\boldsymbol{\theta}+\boldsymbol{v})$ minimal, which translates to minimizing

$$\mathcal{J}(\boldsymbol{\theta}) + \nabla_\theta \mathcal{J}(\boldsymbol{\theta})\boldsymbol{v} + \frac{1}{2}\boldsymbol{v}^T \boldsymbol{H}(\theta)\boldsymbol{v}$$

with respect to $\boldsymbol{v}$, which leads to

$$\boldsymbol{v}_{\text{opt}} = \boldsymbol{H}(\theta)^{-1}\nabla_\theta \mathcal{J}(\boldsymbol{\theta})$$

and hence Newton's method

$$\begin{aligned} \boldsymbol{v}_t &= \boldsymbol{H}(\theta)^{-1}\nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t \end{aligned}$$

Understanding Gradient descent

Considering the one dimensional case and a quadratic loss function, the second order expansion becomes

$$\mathcal{J}(\theta + v) \approx \mathcal{J}(\theta) + \partial_\theta \mathcal{J}(\theta) v + \frac{1}{2} \partial_\theta^2 \mathcal{J}(\theta) v^2$$

From a gradient descent perspective, the optimal learning rate η_{opt} can be reliably derived from the Newton's method updating step

$$v = \eta_t \partial_\theta \mathcal{J}(\theta) = [\partial_\theta^2 \mathcal{J}(\theta)]^{-1} \partial_\theta \mathcal{J}(\theta)$$

which gives

$$\eta_{\text{opt}} = [\partial_\theta^2 \mathcal{J}(\theta)]^{-1}$$

Understanding Gradient descent

It can be shown that there are four qualitatively different regimes possible

- $\eta < \eta_{\text{opt}}$: multiple small steps to reach the solution
- $\eta = \eta_{\text{opt}}$: one single step is required
- $\eta_{\text{opt}} < \eta < 2\eta_{\text{opt}}$: ascillation across both sides of the potential before eventually converging to the minimum
- $\eta > 2\eta_{\text{opt}}$: the gradient descent diverges

source:
LeCun Yann, Leon Bottou, Genevieve B Orr, and Klaus Robert Muller (1998b), *Efficient backprop,* in Neural networks: Tricks of the trade (Springer) pp. 9-50.

Understanding Gradient descent

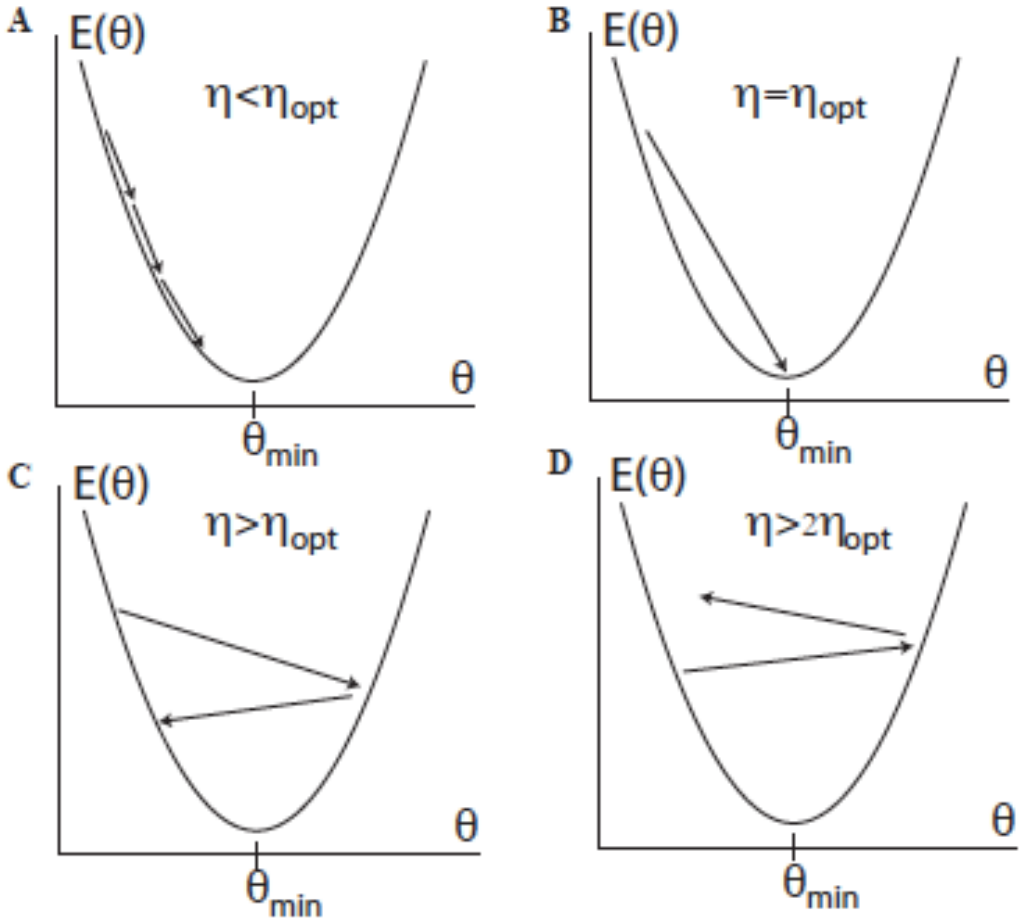


Figure: Effect of learning rate on convergence

Source: Pankaj Mehta, Ching-Hao Wang, Alexandre G. R. Day, and Clint Richardson *A high-bias, low-variance introduction to Machine Learning for physicists*

Limitations of the simplest gradient descent algorithm

- The gradient descent finds local minima of the cost function
- Gradients are computationally expensive to calculate for large datasets
- The gradient descent is very sensitive to choices of the learning rates
- The gradient descent treats all directions in parameter space uniformly
- The gradient descent is sensitive to initial conditions
- The gradient descent can take exponential time to escape saddle points, even with random initialization

Stochastic Gradient Descebt (SGD)

- Traditionally, SGD is a variation of the GD where the gradient is approximated on one data point at a time. The stochasticity is incorporated through the choice of the order of the data points presented to the gradient descent.
- This can be generalized to a subset of the data instead of one data point at each iteration: we call this subset *minibatch*
- The size of the minibatches is almost alwyas much smaaller than the total number of data points
- At each gradient descent step we approximate the gradient using a single minibatch, and update the parameters θ
- if N is the total number of the data points, and M is the size of the minibatch, then the number of SGD steps/iterations is $\frac{N}{M}$
- A full iteration over all N data pointsnis called an *epoch*

Gradient Descent with momentum (GDM)

$$\begin{aligned} \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t \end{aligned}$$

where γ is the momentum parameter with $0 \geq \gamma \geq 1$.

- momentum helps the gradient descent algorithm gain speed in directions with persistent but small gradients even in the presence of stochasticity,
- while suppressing oscillations in highcurvature directions.
- This becomes especially important in situations where the landscape is shallow and flat in some directions and narrow and steep in others

Nesterov Accelerated Gradient (NAG)

$$\begin{aligned} \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t + \gamma \boldsymbol{v}_{t-1}) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t \end{aligned}$$

where γ is the momentum parameter with $0 \geq \gamma \geq 1$.
One of the major advantages of NAG is that it allows for the use of a larger learning rate than GDM for the same choice of γ.

Second order: RMSprop

$$
\begin{aligned}
\boldsymbol{g}_t &= \nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t) \\
\boldsymbol{s}_t &= \beta \boldsymbol{s}_{t-1} + (1-\beta)\boldsymbol{g}_t^2 \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\boldsymbol{g}_t}{\boldsymbol{s}_t + \epsilon}
\end{aligned}
$$

where β controls the averaging time of the second moment and is typically taken to be about $\beta = 0.9$, η_t is a learning rate typically chosen to be 10^{-3}, and $\epsilon \approx 10^{-8}$ is a small regularization constant to prevent divergences.

- learning rate is reduced in directions where the gradient is consistently large.
- This greatly speeds up the convergence by allowing us to use a larger learning rate for flat directions.

Second order: ADAM (adaptive moment estimation)

$$
\begin{aligned}
\boldsymbol{g}_t &= \nabla_\theta \mathcal{J}(\boldsymbol{\theta}_t) \\
\boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g} \\
\boldsymbol{s}_t &= \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \\
\hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\
\hat{\boldsymbol{s}}_t &= \frac{\boldsymbol{s}_t}{1-\beta_2^t} \\
\sigma_t^2 = \hat{\boldsymbol{s}}_t - (\hat{\boldsymbol{m}}_t)^2 \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\sigma_t^2 + \hat{\boldsymbol{m}}_t^2 + \epsilon}}
\end{aligned}
$$

where β_1 and β_2 set the memory lifetime of the first and second moment and are typically taken to be 0.9 and 0.99

- adapting our step size so that we cut off large gradient directions (and hence prevent oscillations and divergences)
- measuring gradients in terms of a natural length scale, the standard deviation

Remarks on the back-propagation

- Stopping criterion:
  - value of the cost function gets smaller than a preselected threshold
  - the gradient values become small, i.e. the values of the weights do not change much from iteration to iteration
- the choice of the step-size (or the learning rate) is very critical
  - it has to be small to guarantee convergence
  - but not too small; otherwise convergence speed slows down
- multiple local minima where the algorithm can be trapped due to the high non-linearity of the neural network problem: reinitialization
- Stochastic versus batch learning scheme

Universal approximation property of feed-forward neural networks

Theorem

Let $g(x)$ be a continuous function defined in a compact subset $S \subset \mathbb{R}^I$ and any $\epsilon > 0$. Then there is a two layer neural network $\hat{g}(x)$ with $K(\epsilon)$ hidden nodes so that

$$|g(x) - \hat{g}(x)| < \epsilon \quad \forall x \in S$$

Proof.

The proof can be found in:
George Cybenko. *Approximation by superpositions of a sigmoidal function*. Mathematics of control, signals and systems, 2(4):303-314,1989. □

theoretical justification of deep architectures

- A deep architecture can represent certain functions (exponentially) more compactly
- Example: Boolean functions
  - a Boolean circuit is a sort of feed-forward network where hidden units are logic gates (i.e. AND, OR or NOT functions of their arguments)
  - any Boolean function can be represented by a *single hidden layer* Boolean circuit
    - however, it might require an exponential number of hidden units
  - it can be shown that there are Boolean functions which
    - require an exponential number of hidden units in the single layer case
    - require a polynomial number of hidden units if we can adapt the number of layers

Issues of the back-propagation

- Initialization
- Overfitting
- Scaling of the inputs
- Number of hidden units and layers
- Multiple Minima

Initialization

- Usually, starting values for weights are chosen to be random values near zero.
- Hence the model starts out nearly linear, and becomes nonlinear as the weights increase.
- Use of exact zero weights leads to zero derivatives and the algorithm never moves.
- Starting instead with large weights often leads to poor solutions

Overfitting

- Often neural networks have too many weights and will overfit the data.
- An early stopping rule can be used to avoid overfitting
  - A validation dataset is useful for determining when to stop, since we expect the validation error to start increasing.
  - Since the weights start at a highly regularized (linear) solution, this has the effect of shrinking the final model toward a linear model.
- a more explicit method for regularization is weight decay, which consists of adding a penalty to the loss function

$$\mathcal{J}(\theta) + \lambda\Omega(\theta)$$

where

$$\Omega(\theta) = \sum_{rjk} (\theta_{jk}^{r})^2$$

- This is analogous to ridge regression

Overfitting

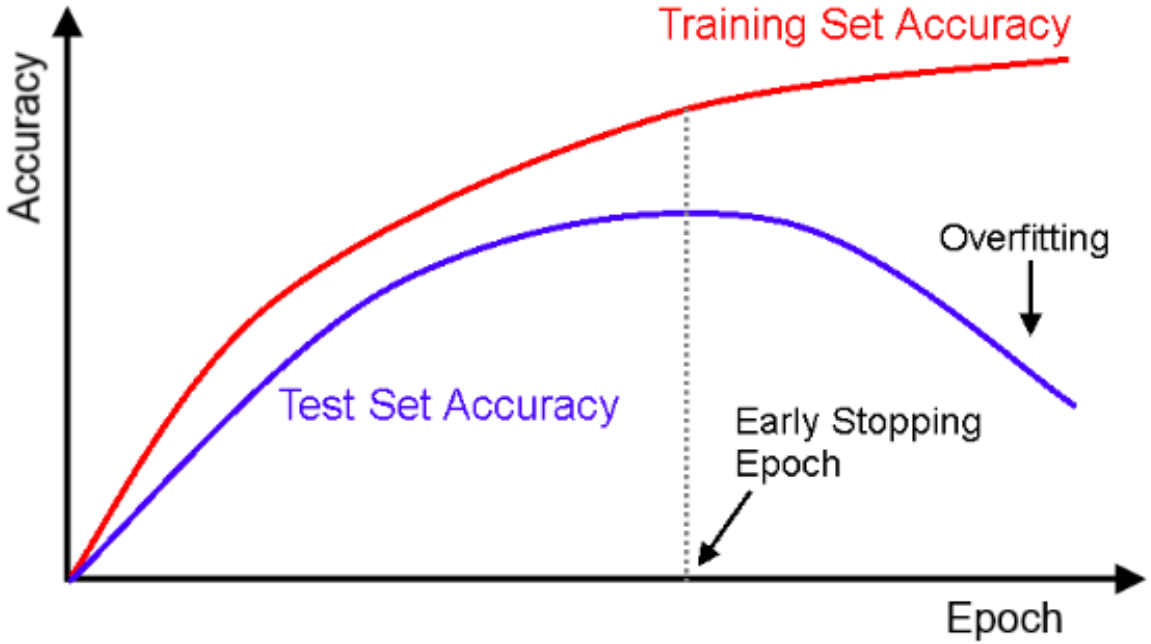


Figure: Early stopping

Multiple minima

- The loss function is nonconvex, possessing many local minima.
- As a result, the final solution obtained is quite dependent on the choice of starting weights.
- One must at least try a number of random starting configurations and choose the solution giving lowest (penalized) error.
- Probably a better approach is to use the average predictions over the collection of networks as the final prediction.
- This is preferable to averaging the weights, since the nonlinearity of the model implies that this averaged solution could be quite poor.
- Another approach is via bagging, which averages the predictions of networks training from bootstraps of the training data.

Number of hidden units and layers

- Generally speaking it is better to have too many hidden units than too few.
- With too few hidden units, the model might not have enough flexibility to capture the non linearities in the data
- with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used
- Typically the number of hidden units is somewhere in the range of 5 to 100, with the number increasing with the number of inputs and number of training cases.
- It is most common to put down a reasonably large number of units and train them with regularization.
- Some researchers use cross-validation to estimate the optimal number
- Choice of the number of hidden layers is guided by background knowledge and experimentation
- Each layer extracts features of the input
- Use of multiple hidden layers allows construction of hierarchical features at different level of resolution.

Training deep networks

Why training is hard:

- First hypothesis: optimization is harder (underfitting)
  - vanishing gradient problem
- Second hypothesis: overfitting
  - we are exploring a space of complex functions
  - deep nets usually have lots of parameters
  - Might be in a high variance/low bias situation

Training deep networks

- Depending on the problem, one or the other situation will tend to dominate
- If first hypothesis (underfitting): better optimize
  - use better optimization methods
  - use non saturating activation functions
- If second hypothesis (overfitting): use better regularization
  - Implicit regularization using SGD
  - Dropout
  - Batch Normalization

The problem of vanishing or exploding gradients

- This problem is especially pronounced in neural networks that try to capture long-range dependencies, such as Recurrent Neural Networks for sequential data.
- We can illustrate this problem by considering a simple network with one neuron in each layer
- We further assume that all weights are equal, and denote them by θ

$$\delta_j^1 = \delta_j^L \prod_{j=0}^{L-1} \theta \sigma'(z_j) = \delta_j^L (\theta)^L \prod_{j=0}^{L-1} \sigma'(z_j)$$

where δ_j^L is the error in the L-th topmost layer, and $(\theta)^L$ is the weight to the power L.

- Let us now also assume that the magnitude $\sigma'(z_j)$ is fairly constant and we can approximate $\sigma'(z_j) \approx \sigma_0'$

The problem of vanishing or exploding gradients (cont.)

Under these simplifying assumptions:

$$\delta_j^1 = \delta_j^L(\theta\sigma_0')^L$$

for large L, the error δ_j^1 has very different behavior depending on the value of $\theta\sigma_0'$

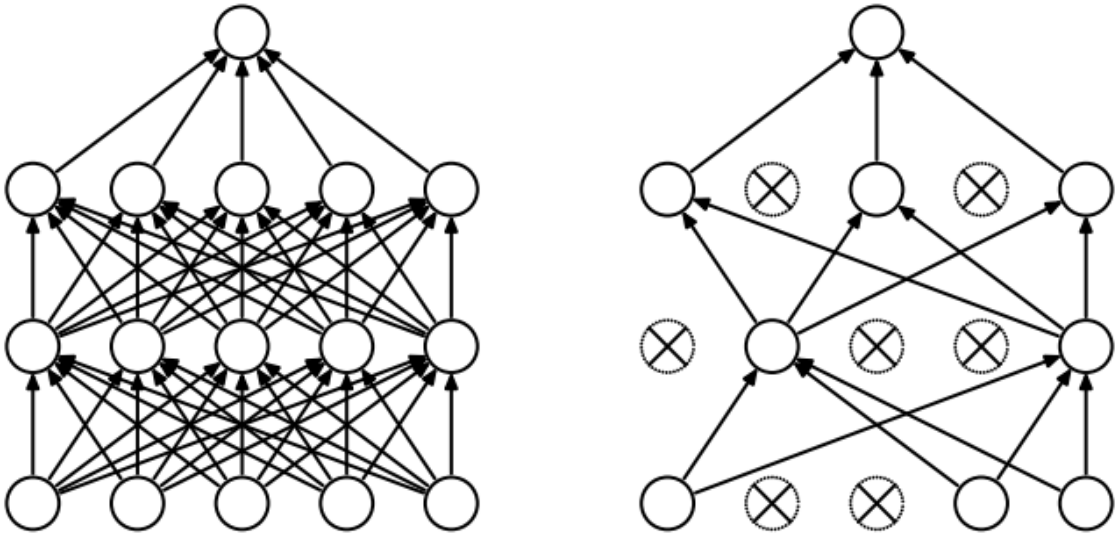
- if $\theta\sigma_0' > 1$, the errors and the gradient blow up
- if $\theta\sigma_0' < 1$, the errors and the gradient vanish
- only when the weights satisfy $\theta\sigma_0' \approx 1$ and the neurons are not saturated will the gradient stay well behaved for deep networks
- if any activation unit saturates in either direction, the errors and gradients vanish

Implicit regularization using SGD

- SGD acts as an implicit regularizer by introducing stochasticity (from the use of mini-batches) that prevents overfitting.
- it is important that the weight initialization is chosen randomly,
- another important thing is to appropriately choose the learning rate or step-size by searching over five logarithmic grid points. If the best performance occurs at the edge of the grid, repeat this procedure until the optimal learning rate is in the middle of the grid parameters.
- it is common to center or whiten the input data
- Another important form of regularization that is often employed in practice is Early Stopping.

Drop-out

- "Cripple" neural network by removing hidden units stochastically
  - each hidden unit is set to 0 with probability p (e.g. 0.5)



(a) Standard Neural Net (b) After applying dropout.

Figure: Drop out illustration

Source: NNitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. 2014. *Dropout: A simple way to prevent neural networks from overfitting*

Drop-out

- "Cripple" neural network by removing hidden units stochastically
- This averaging of weights is similar in spirit to the Bagging procedure

Drop-out during testing

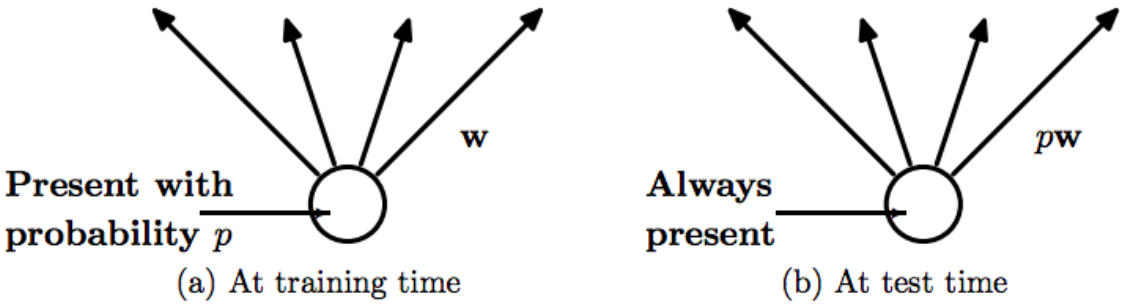


Figure: Drop out during testing

Drop-out formal setup

This section describes the dropout neural network model.
Consider a neural network with L hidden layers. Let $r \in \{1, \cdots, L\}$ index the hidden layers of the network.
A standard network would be defined as:

$$\begin{aligned} z_{ij}^r &= \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_i^{r-1} \\ \boldsymbol{y}_i^r &= \sigma(\boldsymbol{z}_i^r) \end{aligned}$$

With drop-out, the feed-forward operation becomes

$$\begin{aligned} q_j^r &\sim \text{Bernoulli}(p) \\ \tilde{\boldsymbol{y}}_i^{r-1} &= \boldsymbol{q}^r * \boldsymbol{y}_i^{r-1} \\ z_{ij}^r &= \boldsymbol{\theta}_j^{rT} \tilde{\boldsymbol{y}}_i^{r-1} \\ \boldsymbol{y}_i^r &= \sigma(\boldsymbol{z}_i^r) \end{aligned}$$

where $*$ denotes an element-wise product.

Some theoretical justifications

Dropout Decreases the Reliance of Neural Networks on Spurious Interaction Effects.

Source: B. Lengerich, E. P. Xing, R. Caruana, *On Dropout, Overfitting, and Interaction Effects in Deep Neural Networks*, arXiv 2007.00823, July 3, 2020

Batch Normalization

- The basic inspiration behind Batch Normalization is the long-known observation that training in neural networks works best when the inputs are centered around zero with respect to the bias.
- The reason for this is that it prevents neurons from saturating and gradients from vanishing in deep nets.
- In the absence of such centering, changes in parameters in lower layers can give rise to saturation effects in higher layers, and vanishing gradients.

Batch Normalization

The idea of Batch Normalization is to introduce additional new *BatchNorm* layers that standardize the inputs by the mean and variance of the mini-batch. Consider a lyaer r with K_r neurons whose inputs are $(\boldsymbol{z}_1^r, \cdots, \boldsymbol{z}_{K_r}^r)$. We standardize each dimension so tht

$$\boldsymbol{z}_k^r \longrightarrow \hat{\boldsymbol{z}}_k^r = \frac{\boldsymbol{z}_k^r - \mathbb{E}[\boldsymbol{z}_k^r]}{\sqrt{\mathbb{V}\mathrm{ar}[\boldsymbol{z}_k^r]}}$$

where the mean and variance are taken over all samples in the mini-batch. To avoid unwanted changes in the representational power of the neural network, we introdue new parameters that can additionally shift and scale the normalized input

$$\hat{\boldsymbol{z}}_k^r \longrightarrow \hat{\boldsymbol{z}}_k^r = \gamma_k^r \hat{\boldsymbol{z}}_k^r + \beta_k^r$$

References

1. Michael Nielsen, *Neural Networks and Deep Learning*, Online, 2019
2. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, The MIT Press, 2016
3. Charu C. Aggarwal *Neural Networks and Deep Learning*, Springer, 2018
4. Martin Anthony and Peter L. Bartlett *Neural Network Learning: Theoretical Foundations*, Cambridge University Press, 2009
5. Sergios Theodoridis, *Machine Learning: A Bayesian and optimization perspective*, Elsevier, 2015.
6. Christopher M. Bishop, *Pattern Recognition and Machine Learning*, 4Myeloma Press, 2010.