

Lecture Notes IV – Neural Networks, Part 1

Marina Meilă
mmp@uwaterloo.ca

With Thanks to Pascal Poupart & Gautam Kamath
Cheriton School of Computer Science
University of Waterloo

February 2, 2026

A little history

The single “neuron”

Two-layer Neural Networks

- Hidden layer options

- Output layer options

Multi-layer neural networks

Reading HTF Ch.: 11.3 Neural networks, Murphy Ch.: (16.5 neural nets), Bach Ch.: –, Deep Learning Book (Goodfellow, Bengio, Courville) 6.1-4, ResNet 7.6, ConvNet 9., Autoencoders 14.1, Dive Into Deep Learning 4.1-4.3.

A little history

First epoch – can computers mimic the brain?

- ▶ 40's-50's the first mathematical models of the neuron (McCullaugh & Pitts, Hebb)
- ▶ '58 the perceptron (Rosenblatt)
- ▶ First winter '69 "Perceptrons" (Minsky and Papert) – they can only classify linearly separable data (and von Neumann computers steal the show)

Revival in the 80's-'95 – let's use layers!

- ▶ '82 Hopfield associative net (contents adressable memory)
- ▶ '84 Boltzmann Machine (Ackley, Hinton, Sejnowski)
- ▶ autoencoders
- ▶ '86 backpropagation (Sejnowski, Rosenberg)
- ▶ Reinforcement learning (Shannon, Samuel, Barton, Sutton, Watkins, Tesauro)
- ▶ Second winter cca '95 Statistical learning takes the stage, especially SVM

Current epoch cca 2005 – more data, more layers

- ▶ Deep learning
- ▶ Generative models
- ▶ Attention
- ▶ LLMs

Brains vs. Computers

Computer (von Neumann)

- ▶ Electrical binary signals directed by gates
- ▶ Wiring is fixed
- ▶ Sequential and parallel computation
- ▶ Memory is retrieved by address
- ▶ **Fragile** (if a gate stops working, computer crashes). Also, “no friction” – a single instruction can stop the entire machine/program.

▶ Brain

- ▶ Electrical signals, units=neurons
- ▶ Wiring is dynamic, changes with brain development, experiences, learning
- ▶ Parallel (and some) sequential computation
- ▶ Memory is distributed
- ▶ **Robust** (when neuron/region dies, brain rewires itself to compensate). No On/Off master switch

Brains vs. Computers

Computer (von Neumann)

- ▶ Electrical binary signals directed by gates
- ▶ Wiring is fixed
- ▶ Sequential and parallel computation
- ▶ Memory is retrieved by address
- ▶ **Fragile** (if a gate stops working, computer crashes). Also, “no friction” – a single instruction can stop the entire machine/program.

▶ Brain

- ▶ Electrical signals, units=neurons
- ▶ Wiring is dynamic, changes with brain development, experiences, learning
- ▶ Parallel (and some) sequential computation
- ▶ Memory is distributed
- ▶ **Robust** (when neuron/region dies, brain rewires itself to compensate). **No On/Off master switch**

▶ Neural network

- ▶ Signals are numbers passed between units
- ▶ Network structure is fixed (and dense) but the learned weights allow “rewiring” during training
- ▶ Parallel (in layer) and sequential (feed forward/backward) computation
- ▶ Memory is distributed (in the weights)
- ▶ **Redundant/robust** (no single neuron can influence the output much)

(Artificial) Neural Network (nn) unit

- ▶ For each **unit** i

$$y_i \equiv f_i(x) = \phi\left(\sum_j w_{ij}x_j + w_{i0}\right) \quad (1)$$

- ▶ **Weight vector** w_i

- ▶ w_{ij} = strength of the link from unit i to input j
- ▶ $w_{ij} = 0$: no link
- ▶ w_{ij} can be positive or negative
- ▶ Sometimes we call the input vector $x = [x_{1:d}]$ **input units**

- ▶ **activation function** $\phi()$

- ▶ must be non-linear (otherwise the unit is a linear transformation)
- ▶ wanted: monotonically increasing, differentiable, gradient non-zero ¹

¹More technically: ϕ can be any continuous, bounded and strictly increasing function on \mathbb{R} .

(Artificial) Neural Network (nn) unit

- ▶ For each **unit** i

$$y_i \equiv f_i(x) = \phi\left(\sum_j w_{ij}x_j + w_{i0}\right) \quad (1)$$

- ▶ **Weight vector** w_i

- ▶ w_{ij} = strength of the link from unit i to input j
- ▶ $w_{ij} = 0$: no link
- ▶ w_{ij} can be positive or negative
- ▶ Sometimes we call the input vector $x = [x_{1:d}]$ **input units**

- ▶ **activation function** $\phi()$

- ▶ must be non-linear (otherwise the unit is a linear transformation)
- ▶ wanted: monotonically increasing, differentiable, gradient non-zero ¹

Notation ϕ is overloaded

- ▶ When we talk about nn in general: ϕ is any activation function
- ▶ When we do calculations with nn: ϕ is by default the **logistic** function (unless specified otherwise)

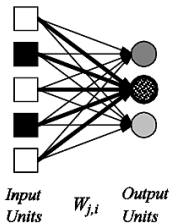
$$\text{logistic or sigmoid function} \quad \phi(u) = \frac{1}{1 + e^{-u}} \quad (2)$$

- ▶ When we do statistics or ML (but not nn): ϕ is the logistic function
- ▶ Exercise: compare $f_i(x)$ from (1) with $p(x) = \Pr[y = 1 | x]$ from logistic regression.

¹More technically: ϕ can be any continuous, bounded and strictly increasing function on \mathbb{R} .

Perceptron

- Single layer feed-forward network



Perceptron weights for OR, AND, XOR

Two-layer Neural Networks

- We build a **two-layer neural network** in the following way:

Inputs	x_j	$j = 1 : d$
Bottom layer ²	$u_i = \phi(w_i^T x)$	$i = 1 : m, w_i \in \mathbb{R}^d$
Top layer	$f = \beta^T u$	$\beta \in \mathbb{R}^m$
Output	f	$\in \mathbb{R}$

In other words, the neural network implements the function

$$f(x) = \sum_{i=1}^m \beta_i u_i = \sum_{i=1}^m \beta_i \phi\left(\sum_{j=1}^d w_{ij} x_j\right) \in (-\infty, \infty) \quad (3)$$

Note that this is just a linear combination of logistic functions.

- As we will see shortly, in general, $f(x)$ can also be non-linear

²In neural net terminology, each variable u_i is a **unit**, the bottom layer is **hidden**, while top one is **visible**, and the units in this layer are called hidden/visible units as well. Sometimes the inputs are called **input units**; imagine neurons or individual circuits in place of each x, u, y variable.

Activation functions for the hidden layer

For the hidden layer, we have to choose

- ▶ number of units m
- ▶ activation function

Common activation functions

- ▶ **Functions that approximate a step function**
 - ▶ threshold function (or **step function**) 1 for $u \geq 0$, and 0 otherwise (not used)
 - ▶ logistic ϕ
 - ▶ hyperbolic tangent \tanh , arctangent \tan^{-1}
- ▶ **Hinge** functions
 - ▶ $\text{RELU} = \max(u, 0)$
 - ▶ $\text{softplus} = \ln(1 + e^u)$
 - ▶ in practical implementations, these unbounded functions are bounded at a large value M
 - ▶ Why hinge functions? Gradient is 1 or 0 (approximately), **faster** computation!!, and no **saturation**

Output layer options

The function from u to y is dictated by the type of prediction problem.

Prediction problem	Output layer f		
regression	linear	$f = \beta^T u$	$\in \mathbb{R}$
binary classification	logistic	$f = \phi(\beta^T u)$	$\in [0, 1]$
multiway classification	softmax	$f_k = \phi_k(\beta_k^T u)$	$\in [0, 1]^r$

- ▶ Regression: **linear** layer as in (3) $f = \sum_i \beta_i u_i$
- ▶ Classification (binary): **logistic** layer $f(x) \in [0, 1]$ is interpreted as the probability of the $+$ class.

$$f(x) = \phi \left(\sum_{j=1}^m \beta_j u_j \right) = \phi \left(\sum_{i=1}^m \beta_j \phi \left(\sum_j w_{ij} x_j \right) \right) \quad (4)$$

- ▶ Multiway classification with r classes
 - ▶ Output is vector of r functions f_1, \dots, f_r
 - ▶ f_k is the probability of $y = k$
 - ▶ (sometimes f_k can be a “confidence”)
 - ▶ This is done with a **softmax** layer (next page)

The softmax function

- ▶ logistic $\phi(z) = \frac{e^z}{1+e^z}$
 - ▶ represents the probability that $y = 1$ in binary classification
 - ▶ in a nn, $f(x) = \phi(\beta^T u)$, with $u \in \mathbb{R}^m$ the activations of the hidden layer
- ▶ The **softmax** function generalizes the logistic to r classes
 $\phi(z) : \mathbb{R}^r \rightarrow (0, 1)^r$

$$\phi_k(z) = \frac{e^{z_k}}{\sum_{j=1}^m e^{z_j}}, \text{ for } k = 1 : r \quad (5)$$

$$\phi(z) = [\phi_1(z) \dots \phi_r(z)] \text{ (overload of } \phi \text{)} \quad (6)$$

- ▶ Properties of softmax
 - ▶ $\sum_{j=1}^m \phi_j(z) = 1$ for all z
 - ▶ for $z_k \gg z_j$, $\phi_k(z) \rightarrow 1$.
 - ▶ derivatives $\frac{\partial \phi_j}{\partial z_k} = \phi_k \delta_{jk} - \phi_j \phi_k$
- ▶ in a nn, with output activations $u \in \mathbb{R}^m$, we train r weight vectors $\beta_{1:r}$, and

$$f_k(x) = \phi_k(\beta_k^T u(x)), \quad \text{for } k = 1 : r \quad (7)$$

OPTIONAL - Generalized Linear Models (GLM)

A GLM is a regression where the “noise” distribution is in the exponential family.

- $y \in \mathbb{R}$, $y \sim P_\theta$ with

$$P_\theta(y) = e^{\theta y - \psi(\theta)} \quad (8)$$

- the parameter θ is a linear function of $x \in \mathbb{R}^d$

$$\theta = \beta^T x \quad (9)$$

- We denote $E_\theta[y] = \mu$. The function $g(\mu) = \theta$ that relates the mean parameter to the natural parameter is called the **link function**.

The log-likelihood (w.r.t. β) is

$$l(\beta) = \ln P_\theta(y|x) = \theta y - \psi(\theta) \quad \text{where } \theta = \beta^T x \quad (10)$$

and the gradient w.r.t. β is therefore

$$\nabla_\beta l = \nabla_\theta l \nabla_\beta (\beta^T x) = (y - \mu)x \quad (11)$$

This simple expression for the gradient is the generalization of the gradient expression you obtained for the two layer neural network in the homework. [Exercise: This means that the sigmoid function is the *inverse link function* defined above. Find what is the link function that corresponds to the neural network.]

Multi-layer/Deep neural networks

The construction can be generalized recursively to arbitrary numbers of layers. Each layer is a linear combination of the outputs from a previous layer (a multivariate operation), followed by a non-linear transformation via the logistic function ϕ . Let $x \equiv x^{(0)}, y \equiv x^{(L)}, m_0 = d, m_L = \dim y$ (typically 1) and define the recursion:

$$x_j^{(l)} = \phi \left(\underbrace{(w_j^{(l)})^T x^{(l-1)}}_{z^{(l)}} \right), \text{ for } j = 1 : m_l, l = 1 : L \quad (12)$$

The vector variable $x^{(l)} \in \mathbb{R}^{m_l}$ is the output of layer l of the network. As before, the sigmoid of the last layer may be omitted.

Are multiple layers necessary?

- ▶ 1990's: NO
- ▶ 2000's: YES
- ▶ 2020's: The more the better!
- ▶ A theoretical result

Theorem (Cybenko, \approx 1986)

Any continuous function from $[0, 1]^d$ to \mathbb{R} can be approximated arbitrarily closely by a linear output, two layer neural network defined in (3) with a sufficiently large number of hidden units m .

- ▶ A practical result



10 BREAKTHROUGH
TECHNOLOGIES 2013

Deep Learning

Deep learning = multi-layer neural net

- ▶ So, what is new?
 - ▶ small variations in the “units”, e.g. switch stochastically w.p. $\phi(\mathbf{w}^T \mathbf{x}^{in})$ (Restricted Boltzmann Machine), Rectified Linear units
 - ▶ training method stochastic gradient, auto-encoders vs. back-propagation (we will return to this when we talk about training predictors)
 - ▶ lots of data
 - ▶ double descent

T.B. EDITED

Example: Logistic regression

Training = Estimating the parameters by Max Likelihood

Problem setup

- ▶ Denote $y_* = (1 - y)/2 \in \{0, 1\}$
- ▶ The likelihood of a data point is $P_{Y|X}(y|x) = \frac{e^{y_* f(x)}}{1 + e^{f(x)}}$
- ▶ The log-likelihood is $l(\beta; x, y) = y_* f(x) - \ln(1 + e^{f(x)})$
- ▶ Log-likelihood of the data set \mathcal{D}

$$l(\beta; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^n l(\beta; (x^i, y^i)) \quad (13)$$

- ▶ Define the **loss function**

$$L_{logl}(\beta) = -l(\beta) \quad (14)$$

- ▶ and the optimization criterion

$$J(\beta) = \hat{L}_{logl} = \frac{1}{n} \sum_{i=1}^n -l(\beta; x^i, y^i) \quad (15)$$

Minimizing J is maximizing $l(\beta; \mathcal{D})$

Logistic regression

- ▶ loss function L_{logl}
- ▶ optimization criterion $J(\beta)$

Calculating the gradient

► $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$

This is a scalar, and $\text{sgn} \frac{\partial l}{\partial f} = y$

► We have also $\frac{\partial f(x)}{\partial \beta} = x$

► Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f} \frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1 + e^{-f(x)}} \right) x \quad (16)$$

Interpretation: The infinitesimal change of β to increase log-likelihood for a single data point is along the direction of x , with the sign of y Exercise Prove that (??) has a unique local optimum.

Calculating the gradient

► $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$

This is a scalar, and $\text{sgn} \frac{\partial l}{\partial f} = y$

► We have also $\frac{\partial f(x)}{\partial \beta} = x$

► Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f} \frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1 + e^{-f(x)}} \right) x \quad (16)$$

Interpretation: The infinitesimal change of β to increase log-likelihood for a single data point is along the direction of x , with the sign of y Exercise Prove that (??) has a unique local optimum.

Algorithm STEEPEST-DESCENT FOR LOGISTIC REGRESSION

Input $\beta^0 \in \mathbb{R}^d$ initial point

For $k = 0, 1, \dots$

1. calculate $d^k = \frac{1}{n} \sum_{i=1}^n \left(y_*^i - \frac{1}{1+e^{-f(x^i)}} \right) x^i$
2. find η^k by line minimization
3. $\beta^{k+1} \rightarrow \beta^k - \eta^k d^k$

until stopping condition satisfied

Output β^{k+1}

Example: Backpropagation

- ▶ The **Backpropagation** algorithm is steepest descent for neural networks
- ▶ Consider a two layer neural network

$$f(x) = \sum_{j=1}^m \beta_j z_j = \sum_{j=1}^m \beta_j \phi\left(\sum_{k=1}^n w_{kj} x_k\right) \quad (17)$$

The parameters are β and $W = [w_{kj}]_{j=1:m, k=1:n}$

- ▶ Let the loss be L_{LS} the **Least Squares** loss, $J(\beta, W) = \hat{L}_{LS}(\beta, W)$

Derivation of the gradient **Exercise** Derive this

$$\frac{\partial J}{\partial \beta_j} = \frac{1}{n} \sum_i \frac{\partial (y^i - f(x^i))^2}{\partial \beta_j} = \frac{1}{n} \sum_i 2(y^i - f(x^i)) z_j(x^i) \quad (18)$$

$$\begin{aligned} \frac{\partial J}{\partial w_{kj}} &= \frac{1}{n} \sum_i \frac{\partial L_{LS}(y^i, f(x^i))}{\partial z_j(x^i)} \frac{\partial z_j(x^i)}{\partial w_{kj}} = \frac{1}{n} \sum_i \left(2\beta_k (y^i - f(x^i)) \right) \underbrace{z_j(x^i)(1 - z_j(x^i))}_{\phi'} x_k^i \\ &= \frac{\beta_k}{n} \sum_i x_k^i (y^i - f(x^i)) \nabla(\text{logistic regressor}) \end{aligned} \quad (19)$$

In the above we used the identity $\phi' = \phi(1 - \phi)$ **Exercise** Prove it

Computational savings

- ▶ when $f(x^i)$ is computed, $z_j(x^i)$ are too; they should be “cached” and re-used
- ▶ the derivative of ϕ is easily obtained from the ϕ value
- ▶ **Exercise** The above gradient formulas can be easily written in matrix-vector form

Backpropagation extends recursively to multi-layer networks. **Exercise** Derive it. **Exercise** Calculate the gradient for the 2 layer neural network with logistic output

Practical properties of backpropagation

- ▶ Unlike in logistic regression, J has many local optima even for two layers and simple problems.
- ▶ Hence, initialization is important, and there are no general rules for a good initialization. Even if the neural network works well, we do not know if we are at the optimum.
- ▶ **Saturation** If $\tilde{z}_j = w_j^T x$ is large in magnitude, then $z_j = \phi(\tilde{z}_j)$ is near 0 or 1. In either case, $\phi'(\tilde{z}_j) = z_j(1 - z_j) \approx 0$. We say that that this sigmoid is **saturated**; z_j will be virtually insensitive to changes in w_j ³
To avoid saturation at the beginning of the training, one initializes W with “small” (w.r.t $\max ||x^i||$, random values. **Exercise** Why random and not exactly 0?
- ▶ To speed up training, it is useful to **standardize the input data**⁴ $x^{1:N}$ as a preprocessing step. **Exercise** Note that theoretically shifting and rescaling the data should NOT have any effect.
- ▶ J can have **plateaus**, i.e. regions where $\nabla J \approx 0$ but that do not contain a local minimum. **Exercise** What can cause plateaus? **Exercise** And what is bad about them?
- ▶ In conclusion, training neural networks by backpropagation is an art: requires experience with the algorithm, careful tuning, repeated restarts, and a long time.

³or to changes in previous layers, if this is a multilayer network.

⁴Standardization should NOT include the dummy coordinate $x^0 \equiv 1$.