USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Neural Networks - Foundations

Machine Learning

Michael Wand

TA: Vincent Herrmann

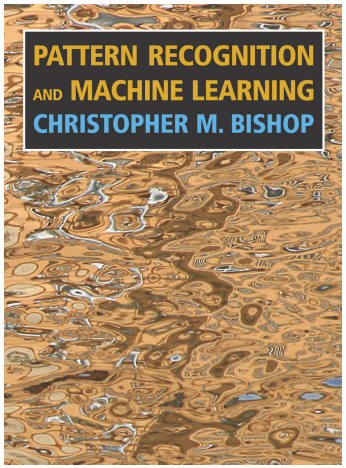
{michael.wand, vincent.herrmann}@idsia.ch

Dalle Molle Institute for Artificial Intelligence Studies (IDSIA) USI - SUPSI

Fall Semester 2024

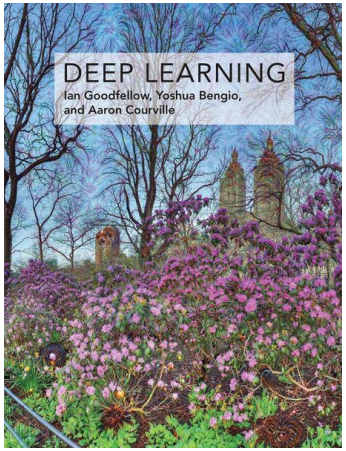
Contents

- The Perceptron
  - ➔ ...and training by *gradient descent*
- Multi-layer neural networks
- Training by *backpropagation*
- Design choices and tricks



Attribution: Some information about the perceptron taken from Bishop's Machine Learning textbook `https://www.microsoft.com/en-us/research/people/cmbishop/prml-book`. The description of classical neural network theory is excellent, the practical design and training considerations is somewhat outdated.

Further reading: Goodfellow/Bengio/Courville, *Deep Learning* `https://www.deeplearningbook.org`.

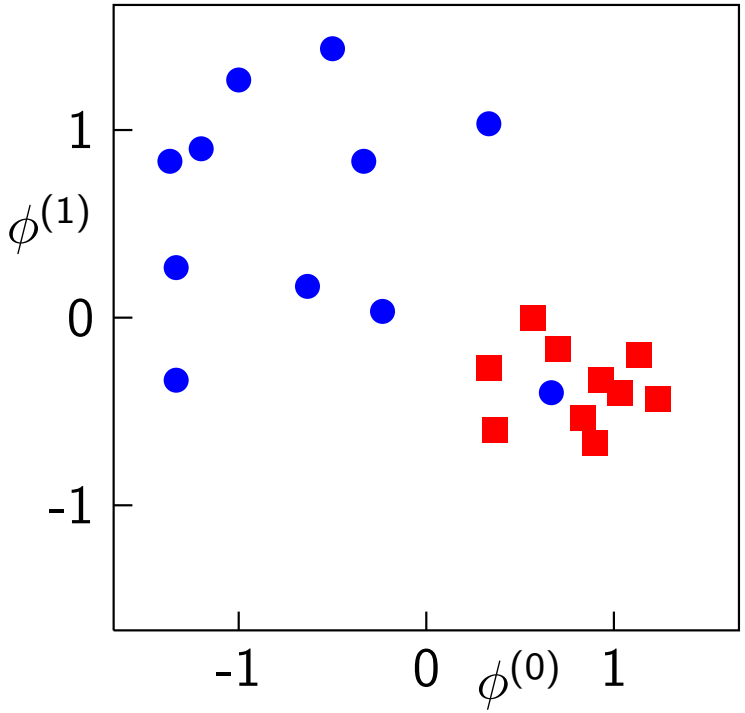


USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

The Perceptron

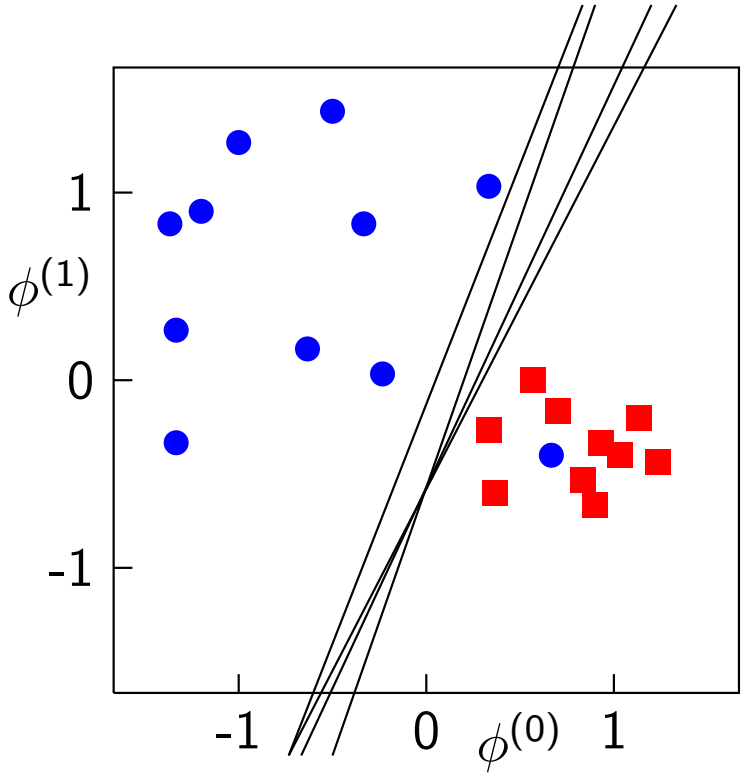
Introduction

- Again consider a two-class problem for classification.
- The classes may or may not be linearly separable.



Introduction

- Again consider a two-class problem for classification.
- The classes may or may not be linearly separable.
- We know the SVM and the maximum margin criterion, as well as Logistic Regression. Which other ways can we think of to
  1. parametrize a classifier,
  2. define a criterion for training it,
  3. actually compute the optimal solution?



The Perceptron Model

- We consider a linear model of the form

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$$

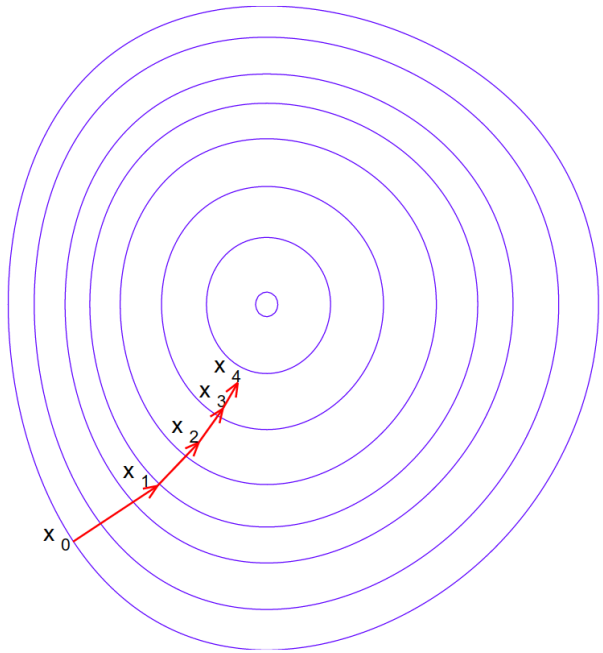
with the usual fixed feature transformation ϕ and an *activation function*

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

- For simplicity, the bias is included in the feature transformation as a fixed value $\phi_0(\mathbf{x}) = 1$.
- The class targets are encoded as $t = \pm 1$ to match the possible values of $y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$.

Gradient Descent

- We wish to use Gradient Descent to minimize the loss of a classifier.
- Idea:
  1. Start at any place $x = x_0$ in the "parameter space".
  2. Consider the *local* shape of the loss function by computing the gradient at the current position x. Note that the gradient points in the direction of steepest ascent.
  3. Take a "step" in the direction of the negative gradient to decrease the loss, arriving at a new position x.
  4. Repeat steps 2 and 3 until satisfied.



Img src: Wikipedia, *Gradient Descent*

Gradient Descent

- Advantages of gradient descent:
  - ➔ Conceptually simple and flexible
  - ➔ Works for any underlying function, only constraint: gradient must be defined and computable
  - ➔ Works in any dimensionality (even in infinite-dimensional spaces)
  - ➔ May offer a computationally tractable solution when other methods fail (e.g. for large amounts of training samples, high-dimensional spaces)
  - ➔ Iterative approach allows a lot of flexible engineering where necessary

Gradient Descent

- Disadvantages of gradient descent:
  - ➔ May get stuck in local minimum (or on a plateau)
  - ➔ Convergence may be slow
  - ➔ No (general) rule to determine step size
  - ➔ When the underlying function is not well known, no theoretical guarantees about quality of the solution, speed, etc.

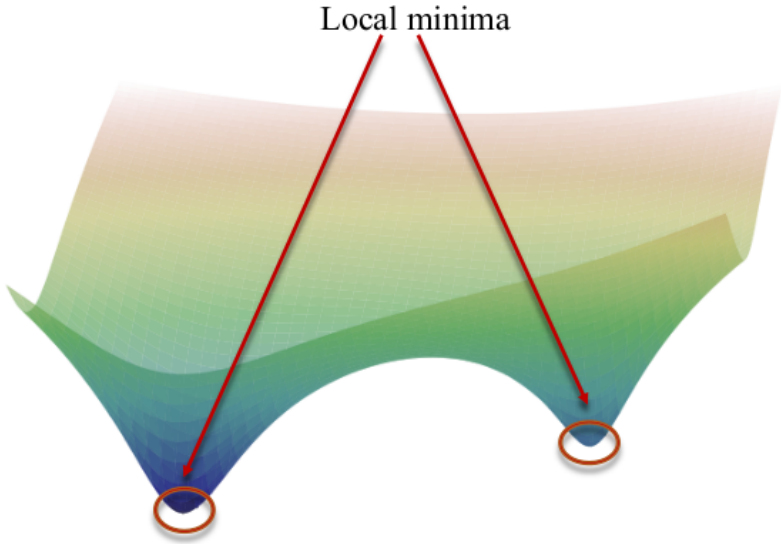


Image source: Belkin, *Fit without fear: remarkable mathematical phenomena of deep learning through the prism of interpolation*, arXiv:2105.14368

Gradient Descent

- In many cases, gradient descent requires some trial-and-error and some heuristics to work
- A lot of engineering has been done to fix fundamental issues of gradient descent (particularly for neural networks)
- But to the present day, it remains the method of choice for neural network training!
- There is some advanced, but really interesting research on why gradient descent works well in the specific case of neural networks

We start with applying gradient descent to the perceptron, which is a *linear* classifier.

The Perceptron Criterion

- Assume a linear classifier.
- Can we simply minimize the number of classification errors by gradient descent?

The Perceptron Criterion

- Assume a linear classifier.
- Can we simply minimize the number of classification errors by gradient descent?
  - ➔ No, because the number of classification errors is *not* differentiable (it only takes integers as values).

The Perceptron Criterion

- Assume a linear classifier.
- Can we simply minimize the number of classification errors by gradient descent?
  - ➔ No, because the number of classification errors is *not* differentiable (it only takes integers as values).
- We derive a differentiable criterion, the Perceptron Criterion (Rosenblatt, 1962).

The Perceptron Criterion

- A sample ϕ_n with target t_n is correctly classified if $\mathbf{w}^T \phi_n t_n > 0$ (in other words, $\mathbf{w}^T \phi_n$ and t_n must have matching signs).
- We assign
  - ➔ zero error to any correctly classified pattern
  - ➔ the error $e_n = -\mathbf{w}^T \phi_n t_n \geq 0$ to any wrongly classified pattern.
- The *Perceptron Criterion* is thus

$$E_P(\mathbf{w}) = \sum_{n \in \mathcal{M}} e_n = -\sum_{n \in \mathcal{M}} \mathbf{w}^T \phi_n t_n$$

  where $\mathcal{M}$ is the set of all misclassified patterns.
- Clearly, $\mathcal{M}$ can change in each gradient step.
- The error $E_P(\mathbf{w})$ is always nonnegative, we want to minimize it.

The Perceptron Criterion

- We apply Stochastic Gradient Descent to the error $E_P(\mathbf{w})$.
- This means that we evaluate the error gradient for a *single, randomly selected* (misclassified) sample $\mathbf{x}_n$:

$$\nabla_{\mathbf{w}} e_n = -\nabla_{\mathbf{w}} \mathbf{w}^T \phi_n t_n = -\phi_n t_n$$

- $\mathbf{w}$ is changed by taking a gradient step with *learning rate* η:

$$\mathbf{w}^{\text{new}} = \mathbf{w} - \eta \nabla_{\mathbf{w}} e_n = \mathbf{w} + \eta \phi_n t_n.$$

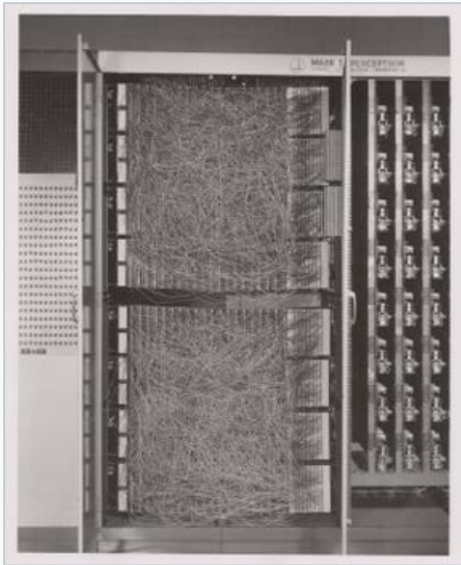
- It is easy to see that this reduces the error *for this particular sample* (but not necessarily the total error).
- Nonetheless, if the training set is linearly separable, the algorithm finishes in finitely many steps (there are no more misclassified samples). Yet even then, convergence can be very slow.

Perceptron Problems

The algorithm has a variety of shortcomings:

- If the data set is not linearly separable, the algorithm never converges (and may not find a good solution if stopped randomly).
- The method does not generalize to more than two classes.
- Convergence can be very slow.

Despite these limitations, the perceptron remains a major milestone in the theory and practice of neural networks (and of machine learning in general).



Mark I Perceptron machine, the first implementation of the perceptron algorithm. It was connected to a camera with 20×20 cadmium sulfide photocells to make a 400-pixel image. The main visible feature is a patch panel that set different combinations of input features. To the right, arrays of potentiometers that implemented the adaptive weights.[2]:213

Image source: Wikipedia, *Perceptron*, original image from Cornell University

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Multi-layer Neural Networks

About Nonlinearities

- We have seen that in many cases, classes (in classification) are not linearly separable, but may be better separable with a *nonlinear* function.
- Also for regression, we have seen that we may need a nonlinear function of the data.
- Conclusion: We need to perform some kind of nonlinear calculation.
- We have done this by using nonlinear basis functions to model the data, then applying a linear model in feature space.
- Are there better ways to parametrize a nonlinear model?

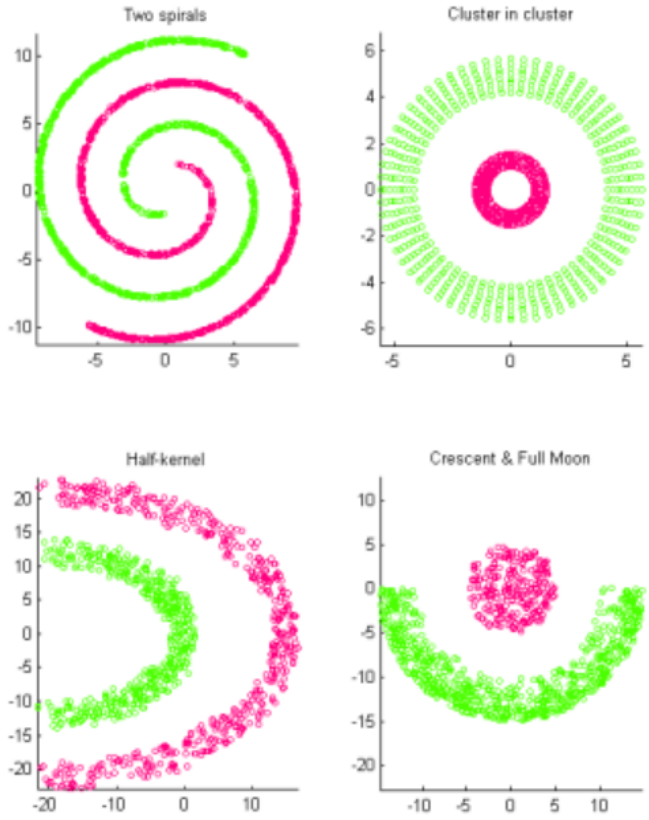


Image modified from Jeroen Kools (2020). 6 functions for generating artificial datasets (https://www.mathworks.com/matlabcentral/fileexchange/41459-6-functions-for-generating-artificial-datasets), MATLAB Central File Exchange.

About Iterated Computations

- Linear classification consists of exactly two computational steps (computation of features, computation of scalar product).
- (Formulation is a bit different for the SVM, but the situation is fundamentally the same.)
- Yet, the optimization was already quite complex.
- In order to be better, we want to allow the classifier to take even more complex functional forms—but how?

About Iterated Computations

- Linear classification consists of exactly two computational steps (computation of features, computation of scalar product).
- (Formulation is a bit different for the SVM, but the situation is fundamentally the same.)
- Yet, the optimization was already quite complex.
- In order to be better, we want to allow the classifier to take even more complex functional forms—but how?
- Idea: allow *multiple* computational steps
  - ➔ each one of which may be simple
- From mathematics (dynamical systems, complexity theory): Iterated application of simple rules can generate *very complex* behavior.

About Iterated Computations

- Linear classification consists of exactly two computational steps (computation of features, computation of scalar product).
- (Formulation is a bit different for the SVM, but the situation is fundamentally the same.)
- Yet, the optimization was already quite complex.
- In order to be better, we want to allow the classifier to take even more complex functional forms—but how?
- Idea: allow *multiple* computational steps
  - ➔ each one of which may be simple
- From mathematics (dynamical systems, complexity theory): Iterated application of simple rules can generate *very complex* behavior.
- Take inspiration from the human brain, a network of *neurons*: Each neuron has very simple behavior (and is somewhat understood), but the behavior of the whole brain, with billions of interconnected neurons, is *extremely* complex (and terribly hard to understand)!

Feedforward Fully-Connected Neural Networks

We define a *feedforward fully-connected neural network* as follows.

- Let $\mathbf{x} = x_1, \ldots, x_D$ be the D-dimensional input vector.
- $M^{(1)}$ neurons perform a perceptron-like computation

$$u_m^{(1)} = (\mathbf{w}_m^{(1)})^T \mathbf{x} + b_m^{(1)}, \qquad z_m^{(1)} = f(u_m^{(1)}), \qquad m = 1, \ldots, M^{(1)}$$

  with a differentiable *activation function* f (for gradient descent).
- This step is iterated multiple times, taking the outputs $\mathbf{z}^{(\ell-1)} = (z_m^{(\ell-1)})_{m=1,\ldots,M^{(\ell-1)}}$ of the previous step as input:

$$u_m^{(\ell)} = (\mathbf{w}_m^{(\ell)})^T \mathbf{z}^{(\ell-1)} + b_m^{(\ell)}, \qquad z_m^{(\ell)} = f(u_m^{(\ell)}),$$
$$m = 1, \ldots, M^{(\ell)} \qquad \text{and} \qquad \ell = 2, \ldots, L.$$

  (note that the weights are usually independent for each step).
- The output of the entire network is then $\mathbf{y} = \mathbf{z}^{(L)}$.

Feedforward Fully-Connected Neural Networks

- We additionally define $\mathbf{z}^{(0)}$ to be the input, i.e.

$$\mathbf{z}^{(0)} = \mathbf{x}.$$

- For each layer $\ell \in 1, \ldots, L$, the computation is

$$z_m^{(\ell)} = f\left((\mathbf{w}_m^{(\ell)})^T \mathbf{z}^{(\ell-1)} + b_m^{(\ell)}\right)$$

  which can be written as a matrix multiplication:

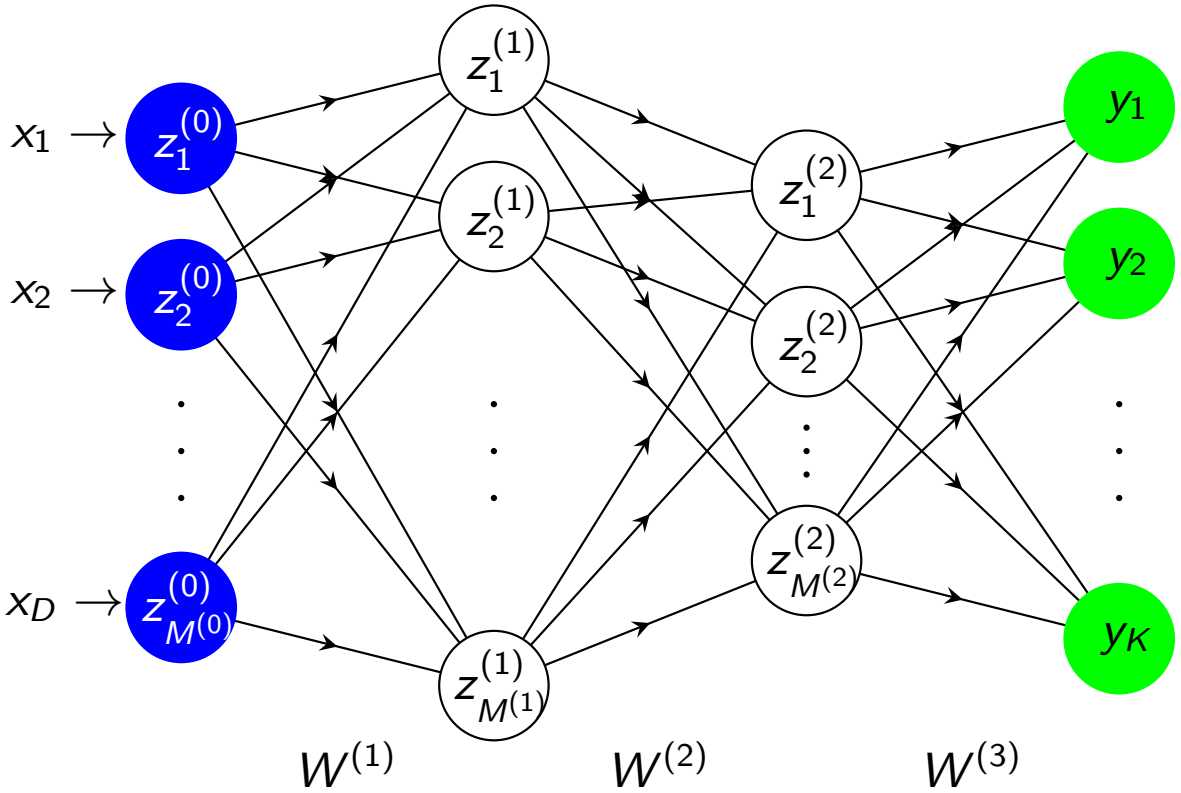
$$\mathbf{z}^{(\ell)} = f\left(\mathbf{W}^{(\ell)} \mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right).$$

- The activation function is usually applied component-wise, but can also be applied to the output vector as a whole.

Representation of a Neural Network

- The $z_m^{(\ell)}$ are neurons, each of which takes its input values and computes a single output value from them
- The inputs $x_1, \ldots, x_D$ are occasionally called input neurons (even though they do not compute anything)
- The neurons are organized in layers $1, \ldots, L$. (Some people consider the input the zeroth layer.)
- The weights $\mathbf{w}$ are directed connections between the neurons, e.g. the neurons of layer 2 are connected to the ones of layer 1 by the weights $w_{mn}^{(2)}$, $m = 1, \ldots, M^{(1)}$, $n = 1, \ldots, M^{(2)}$.

Representation of a Neural Network



The image graphically represents a neural network with three layers, or two *hidden* layers. Computation runs from the left to the right. Note that $M^{(0)} = D$ and $M^{(3)} = K$.

Feedforward Fully-Connected Neural Networks

- Each *neuron* computes the weighted sum of the connected inputs, followed by a differentiable *activation function*. The activation function should be nonlinear (why?); it can differ for each layer.
- The neurons are organized in *layers* to allow parallel computation, to avoid cyclic dependencies (we will discuss later how to implement cycles, or *recurrence*, in NNs), and to simplify reasoning about the system: a feedforward network.
- There is a full set of connections between successive layers: a fully connected network.
- The process of computing NN outputs from inputs is called forward propagation.
- This kind of network is also called a Multi-layer perceptron.

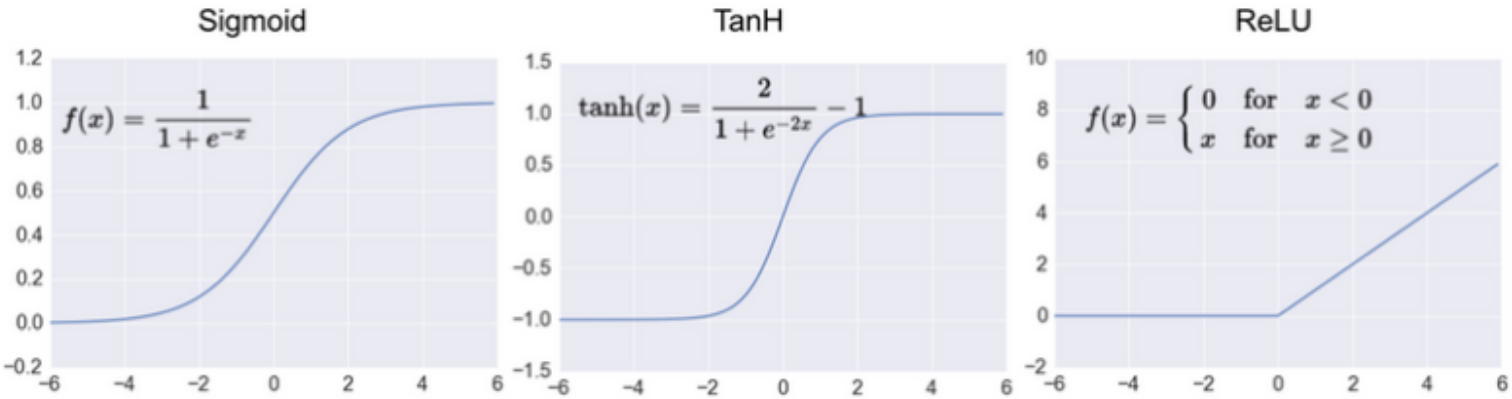
Feedforward Fully-Connected Neural Networks

- Activation functions need to be differentiable (because we wish to apply gradient descent training).
- For the hidden layers of the network, the activation function must be *nonlinear*, because multiple *linear* computations can be collapsed to a single one: In order to gain power from iterative computation, we thus need nonlinear steps.
- The activation function of the last layer usually depends on the task (e.g. classification or regression).
- Finally, in supervised training, we compare the output $\mathbf{y} = \mathbf{y}(\mathbf{x})$ with a *target* $\mathbf{t}$ and compute a scalar error $E = E(\mathbf{y}, \mathbf{t})$.
- The error allows us to measure the performance of the network, and to derive a criterion for training.

Feedforward Fully-Connected Neural Networks

- Many possible activation functions for the hidden layers of a neural network exist:
  - ➔ Sigmoid, Hyperbolic Tangent: Monotonic, squeeze output to a fixed range
  - ➔ ReLU: "Almost linear" (a clipped identity function), works very well. Encourages *sparsity* of representations. Currently state-of-the-art.

  A large number of variants (not covered here) has been proposed.



Feedforward Fully-Connected Neural Networks

- We see that the forward step comprises *as many computation steps as there are layers*.
- Thus we have achieved the goal of creating a "complex" calculation from multiple simple steps (matrix multiplication + nonlinearity).
- We now discuss how to set up the NN for a practical task.
- Then we will derive the standard training method for the neural network.

NN Setup for Regression

- Assume a *regression* task: compute a mapping $\mathbb{R}^D \to \mathbb{R}^K$.
- Since the output of the last layer can have arbitrary range, one usually chooses a *linear* activation function (for the last layer only!): $f(x) = x$.
- The hidden layers can have any nonlinear activation function.
- Use the well-known squared error: $E = \frac{1}{2}\sum_k (t_k - y_k)^2$, where the sum runs over the K components of the vectors[1].
- Note that the NN naturally handles multi-dimensional targets.

[1] this formula is for *one* sample only, for multiple samples take the mean

NN Setup for Classification

- For a *classification* task with K classes, we use a K-dimensional output layer.
- A sample $x \in \mathbb{R}^D$ is classified as belonging to class k if the output neuron y_k has the maximal value:

$$\hat{c} = \arg\max_k y_k .$$

- Problem: The arg max function has a degenerate gradient!

NN Setup for Classification

- For a *classification* task with K classes, we use a K-dimensional output layer.
- A sample $x \in \mathbb{R}^D$ is classified as belonging to class k if the output neuron y_k has the maximal value:

$$\hat{c} = \arg\max_k y_k .$$

- Problem: The arg max function has a degenerate gradient!
- This is solved by letting the neural network output a *probability distribution* over classes, i.e.

$$\mathbf{y} = (y_k)_{k=1,\ldots,K} \quad \text{with} \quad y_k \geq 0, \sum_k y_k = 1.$$

- Advantage: We can derive a (differentiable) measure of the quality of the output on *theoretical* grounds, using probability theory.

NN Setup for Classification

- In order to make the network output a probability distribution, we take exponentials and normalize. This is the *softmax* nonlinearity:

$$S(\mathbf{y}) = \left(\frac{e^{y_1}}{\sum_k e^{y_k}}, \ldots, \frac{e^{y_K}}{\sum_k e^{y_k}} \right).$$

  Note that in constrast to other activation functions, it is applied to the *full* last layer of the network, not to each independent component.
- The hidden layers can have any nonlinear activation function (just as for regression).

NN Setup for Classification

- Assume a neural network with softmax output.
- We compute the loss by measuring the *cross-entropy* between the output distribution and the target distribution.
  - ➔ We encode the targets in *one-hot* style, e.g. if a sample belongs to class k, the target is

$$\mathbf{t} = (0, \ldots, 0, \underset{\substack{\uparrow \\ k\text{-th element}}}{1}, 0, \ldots, 0)$$

  - ➔ Consider this a probability distribution: obviously, a perfect hypothesis $\mathbf{y}$ would exactly match this $\mathbf{t}$, assigning probability 1 to the correct class, and probability 0 otherwise.
  - ➔ The cross-entropy loss is defined as

$$E_{\text{Crossent}} = -\sum_k (t_k \log y_k).$$

Intuition: The cross-entropy corresponds to the number of additional bits needed to encode the correct output, given that we have access to the (possibly wrong) prediction of the network.

NN Setup for Classification

We note some properties of the cross-entropy loss:

- It is always nonnegative (do you see why?)?
- In the case of deterministic targets (exactly one $t_k = 1$, all others are zero), the formula simplifies to

$$E_{\text{Crossent}} = -\log y_{k_{\text{correct}}},$$

  and we see that the loss goes to zero if $y_{k_{\text{correct}}}$ approaches one (since the y_k must be a probability distribution, this implies that all other y_k must go to zero).
- However, the loss also works for probabilistic targets.
- The neural network gracefully handles probabilistic outputs and multi-class classification.
- Remark: For efficiency and numerical stability, one should merge softmax loss and cross-entropy criterion into one function.

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Training Neural Networks by Backpropagation

Gradient Descent by Backpropagation

- We will use *Gradient Descent* to train a neural network.
  - ➔ Remark 1: there are ways to perform gradient descent training even in unsupervised or semi-supervised scenarios (training targets unavailable or partially available).
  - ➔ Remark 2: it is also possible to optimize neural networks without gradient descent (e.g. by evolution `http://people.idsia.ch/~juergen/compressednetworksearch.html`).
- This requires to compute the gradient of the neural network error w.r.t. each weight.
- We will first derive the Backpropagation algorithm which allows performing this computation in an efficient way.

Backpropagation Training

- Assume that for a given sample $\mathbf{x}$, we have the error $E(\mathbf{y}) = E(\mathbf{z}^{(L)})$.
- We must compute the gradients of E w.r.t. the weights.
- We prepare ourselves by doing two simple computations: Since $z_n^{(\ell)} = f\left(u_n^{(\ell)}\right) = f\left(\sum_m w_{mn}^{(\ell)} z_m^{(\ell-1)} + b_n^{(\ell)}\right)$, we have (chain rule!)[2]

$$\frac{\partial z_n^{(\ell)}}{\partial w_{mn}^{(\ell)}} = f'\left(u_n^{(\ell)}\right) z_m^{(\ell-1)}$$

$$\frac{\partial z_n^{(\ell)}}{\partial b_n^{(\ell)}} = f'\left(u_n^{(\ell)}\right)$$

$$\frac{\partial z_n^{(\ell)}}{\partial z_m^{(\ell-1)}} = f'\left(u_n^{(\ell)}\right) w_{mn}^{(\ell)}$$

for any $\ell = 1, \ldots, L$.

[2] This assumes that the nonlinearity f is computed independently for each neuron, which in practice is true except for the softmax nonlinearity. We will remove this restriction later on.

Backpropagation Training

- For the *last* layer, we can now immediately compute the gradients:

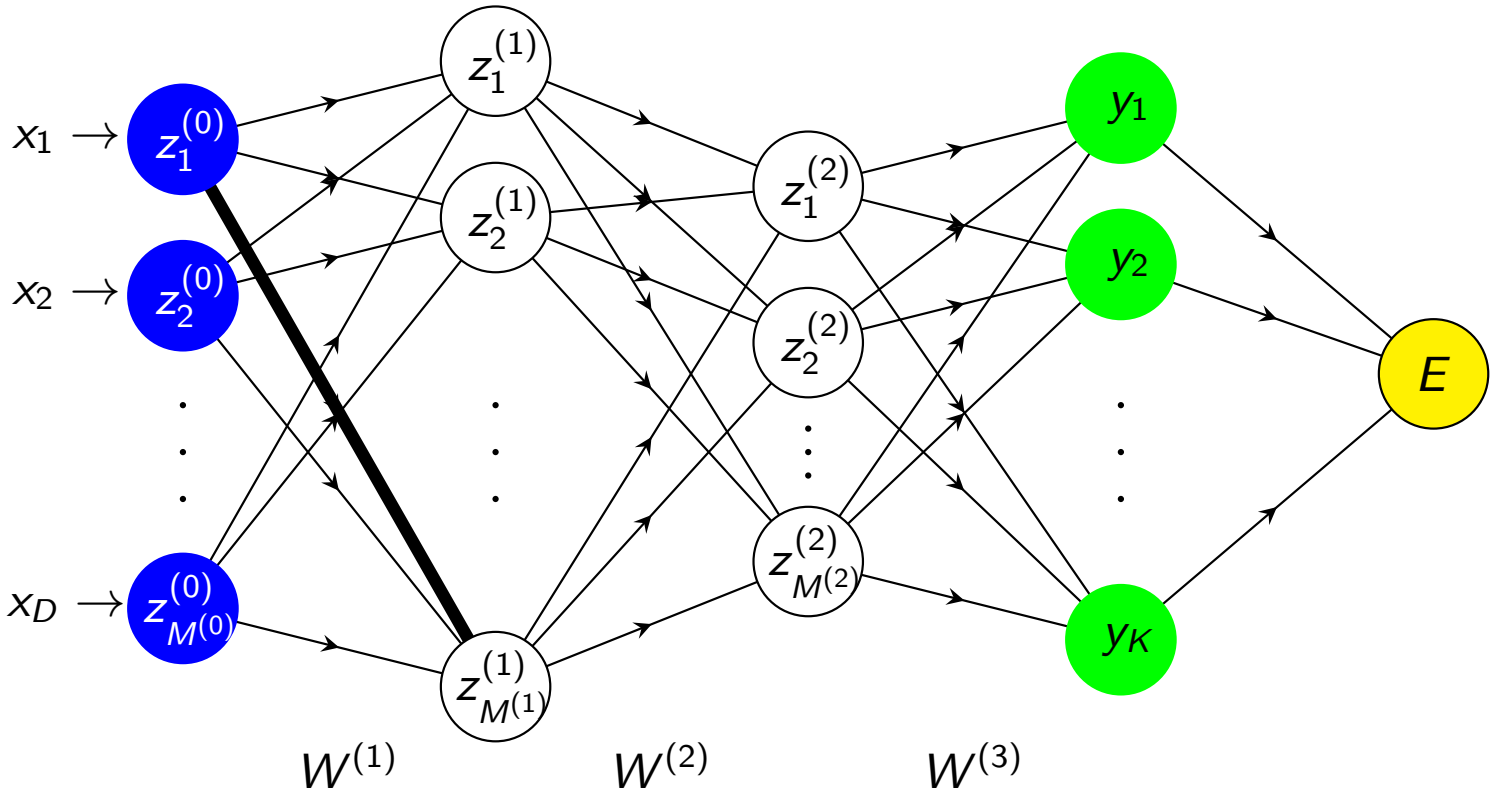
$$\frac{\partial E}{\partial w_{mn}^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} \frac{\partial z_n^{(L)}}{\partial w_{mn}^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} f'\left(u_n^{(L)}\right) z_m^{(L-1)}$$

$$\frac{\partial E}{\partial b_n^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} \frac{\partial z_n^{(L)}}{\partial b_n^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} f'\left(u_n^{(L)}\right).$$

- This computation is easiest for the last layer, since there is only one "path" in which the weight $w_{mn}^{(L)}$ influences the error[3].
- Let us now consider the general case.

[3] Again, this is not correct when the nonlinearity is computed on the entire layer.

Backpropagation Training



The situation is slightly more complicated for the lower layers, since we need to consider *all* paths which lead to a certain weight. In how many ways does the indicated weight influence the loss?

We write

$$\frac{\partial E}{\partial \mathbf{z}^{(\ell)}} = \left(\frac{\partial E}{\partial z_1^{(\ell)}}, \ldots, \frac{\partial E}{\partial z_{M^{(\ell)}}^{(\ell)}} \right) \in \mathbb{R}^{1 \times M^{(\ell)}};$$

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{z}^{(\ell-1)}} = \begin{pmatrix} \frac{\partial z_1^{(\ell)}}{\partial z_1^{(\ell-1)}} & \cdots & \frac{\partial z_1^{(\ell)}}{\partial z_{M^{(\ell-1)}}^{(\ell-1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_{M^{(\ell)}}^{(\ell)}}{\partial z_1^{(\ell-1)}} & \cdots & \frac{\partial z_{M^{(\ell)}}^{(\ell)}}{\partial z_{M^{(\ell-1)}}^{(\ell-1)}} \end{pmatrix} \in \mathbb{R}^{M^{(\ell)} \times M^{(\ell-1)}}; \quad \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \begin{pmatrix} \frac{\partial z_1^{(\ell)}}{\partial w_{ij}^{(\ell)}} \\ \vdots \\ \frac{\partial z_{M^{\ell}}^{(\ell)}}{\partial w_{ij}^{(\ell)}} \end{pmatrix} \in \mathbb{R}^{M^{(\ell)}} \times 1.$$

Remember the rules for derivatives of multivariate functions: Input variables go into columns, output components go into rows, i.e. for $\mathbf{f} : \mathbb{R}^D \to \mathbb{R}^K$,

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} | & | & & | \\ \frac{\partial \mathbf{f}}{\partial x_1} & \frac{\partial \mathbf{f}}{\partial x_2} & \cdots & \frac{\partial \mathbf{f}}{\partial x_D} \\ | & | & & | \end{pmatrix} = \begin{pmatrix} — & \frac{\partial f_1}{\partial \mathbf{x}} & — \\ — & \frac{\partial f_2}{\partial \mathbf{x}} & — \\ & \cdots & \\ — & \frac{\partial f_K}{\partial \mathbf{x}} & — \end{pmatrix} \in \mathbb{R}^{K \times D}$$

Backpropagation Training

- We furthermore decompose $\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}}$ into the gradient of the nonlinearity and the network part:

$$\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} = \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{u}^{(\ell+1)}} \frac{\partial \mathbf{u}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}}.$$

- We define $\mathbf{F}^{(\ell+1)} := \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{u}^{(\ell+1)}}$. In the case of a component-wise nonlinearity, $\mathbf{F}^{(\ell+1)}$ is a diagonal matrix.
- Also note that because of $\mathbf{u}^{(\ell+1)} = \mathbf{W}^{(\ell+1)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)}$, the second factor $\frac{\partial \mathbf{u}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}}$ is just the weight matrix $\mathbf{W}^{(\ell+1)}$!

Backpropagation Training

- Then, by the chain rule,

$$\frac{\partial E}{\partial w_{mn}^{(\ell)}} = \frac{\partial E}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdots \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{mn}^{(\ell)}}$$

  where the multiplications are matrix multiplications.
- We leave out the formulas for updating the bias since they are very similar.

Backpropagation Training

This gives us a straightforward way to compute the gradients for *all* weights in the network.

- Let $\delta^{(\ell)}$ be the gradient of the loss w.r.t. the activation of the ℓ-th layer:

$$\delta^{(\ell)} = \frac{\partial E}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdots \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{1 \times M^{(\ell)}}$$

  and note that it can be computed recursively:

$$\delta^{(\ell)} = \delta^{(\ell+1)} \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} = \delta^{(\ell+1)} \mathbf{F}^{(\ell+1)} \mathbf{W}^{(\ell+1)} \quad \text{and} \quad \delta^{(L)} = \frac{\partial E}{\partial \mathbf{z}^{(L)}}.$$

- Combining prior results, we also see

$$\frac{\partial E}{\partial w_{mn}^{(\ell)}} = \delta_\ell \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{mn}^{(\ell)}} = \delta_\ell \mathbf{F}^{(\ell)} z_m^{(\ell-1)}.$$

- . . . and that's all we need.

Gradient Descent by Backpropagation

Here is the complete algorithm to perform a gradient step in neural network training, using the backpropagation algorithm to compute the gradients:

- Perform the forward pass, save intermediate results
- for $\ell = L, \ldots, 1$,
  - ➔ compute $\delta^{(\ell)}$ from $\delta^{(\ell+1)}$ (except for $\ell = L$, the recursion start)
  - ➔ compute (and save) the weight gradients for layer ℓ
  - ➔ all required formulas are on the previous slide.
- Update all weights simultaneously:

$$\mathbf{w}^{\text{new}} = \mathbf{w} - \eta \nabla \mathbf{w}$$

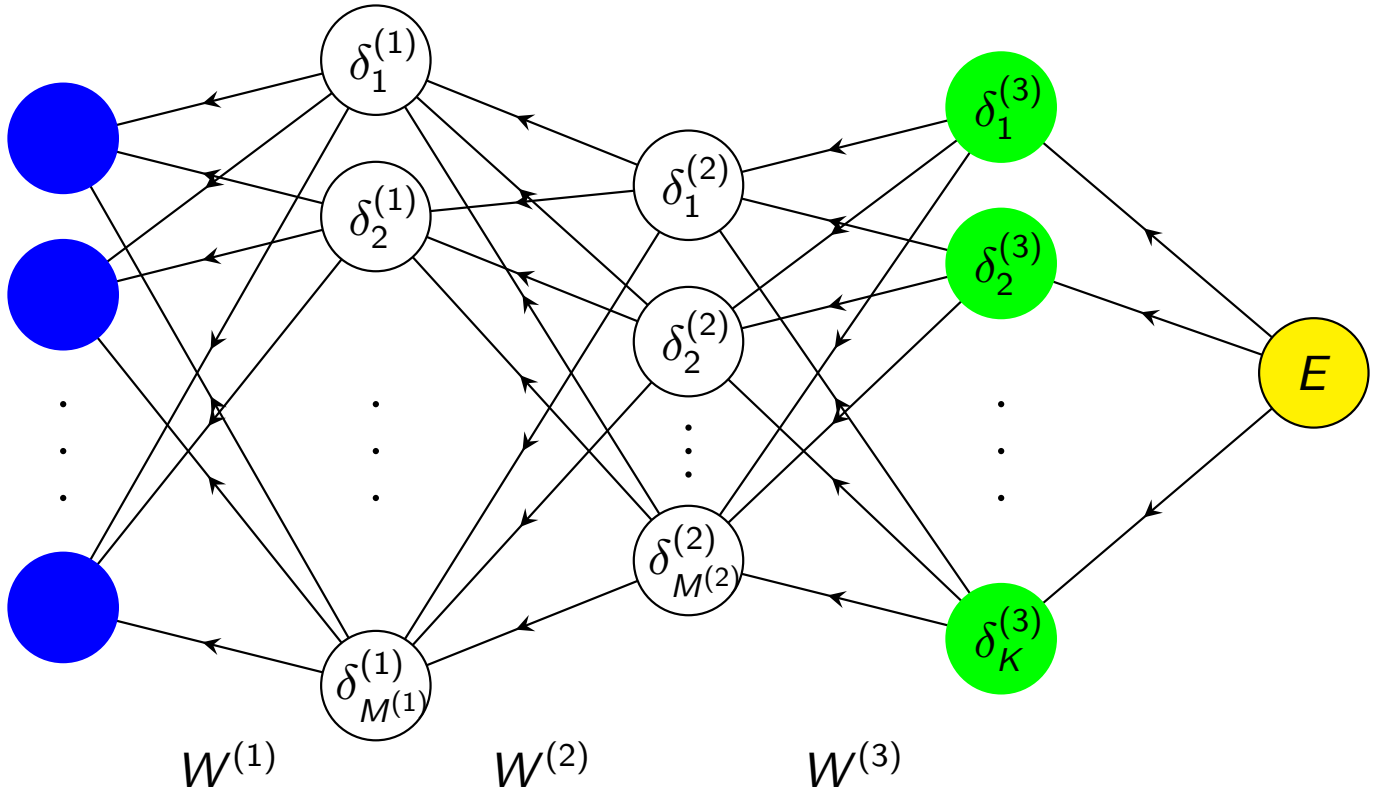
where η is the learning rate, and $\nabla \mathbf{w}$ collects the gradients.

Gradient Descent by Backpropagation

Here is the complete algorithm to perform a gradient step in neural network training, using the backpropagation algorithm to compute the gradients:

- The name *backpropagation* for this implementation of gradient descent stems from the way the error is *propagated* from the network output to its layers, in *backwards* order.
- Every partial derivative can be computed by a *local* computation (i.e. using the $\delta^{(\ell)}$ from the backward pass, and the $\mathbf{z}^{(\ell-1)}$ from the forward pass).
- The $\delta^{(\ell)}$ are also called errors, they assign *credit* or *blame* to each node in each layer. Thus the error of the entire network (which we want to minimize) is distributed over its components.
- Such credit assignment is a fundamental problem in machine learning.

Gradient Descent by Backpropagation



- The errors $\delta_i^{(\ell)}$ assign *credit* or *blame* to each node in each layer.
- Thus we have quantified the contribution of each node to the network loss.

Gradient Descent by Backpropagation

At this point, you should have learned about backpropagation:

- that it is very similar to reverse forward propagation!
- In the forward case, we compute neuron activations from layer 1 to layer L
- In the backward case, we compute errors from layer L to layer 1.
- (Clearly, this makes only sense *after* a forward pass.)

Note that we need to collect intermediate results in *both* passes in order to train the network.
Also note that backward and forward pass have the same complexity.
Finally, distinguish *(Stochastic) Gradient Descent* (an optimization method) and *backpropagation* (which is used to compute gradients which are required for gradient descent).

Backpropagation Training

In order to finish the picture, let us google the derivatives of the nonlinearities:

Function	formula	derivative
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$\tanh'(x) = 1 - \tanh^2(x)$
ReLU	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$
Linear	$f(x) = x$	$f'(x) = 1$
Softmax	$\mathrm{S}(\mathbf{x}) = (S_1, \ldots, S_K)$ with $S_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$	$\partial_i \mathrm{S}_j = \begin{cases} \mathrm{S}_i(1-\mathrm{S}_i) & i = j \\ -\mathrm{S}_i \mathrm{S}_j & i \neq j \end{cases}$

And here are the derivatives of the errors:

Function	formula	derivative
MSE	$E_{\text{MSE}} = \frac{1}{2}\sum_k (\mathbf{t}_k - \mathbf{y}_k)^2$	$\frac{\partial E_{\text{MSE}}}{\partial \mathbf{y}} = (y_k - t_k)_k$
Cross-Entropy	$E_{\text{Crossent}} = -\sum_k (t_k \log y_k)$	$\frac{\partial E_{\text{Crossent}}}{\mathbf{y}} = \left(-\frac{t_k}{y_k}\right)_k$
Cross-Entropy *and* Softmax	$E_{\text{CE + SM}} = -\sum_k (t_k \log S_k(\mathbf{y}))$	$\frac{\partial E_{\text{CE + SM}}}{\mathbf{y}} = \left(\sum_i t_i y_k - t_k\right)_k$

where in the latter case $\mathbf{y}$ is network output before the softmax nonlinearity.

Exercise: Which simple form does the combined softmax + cross-entropy error take if the target is deterministic (only one t_i is nonzero)?

Training Setup

- We now have defined (and proved) the complete algorithm for *backpropagation*.
- In practical setups, one usually accumulates gradient information from a *mini-batch* of several samples (say, 32 or 64)
  - ➔ makes the gradient steps more stable
  - ➔ parallelizes better.
- A full iteration over all training samples is called an *epoch*.
- The learning rate can be determined experimentally, but there are also algorithms which adapt it automatically.
- A simple stopping criterion could be derived by checking the error on the training dataset: When the change is small for a few steps, we have reached convergence and stop
  - ➔ but this is not how we usually do it.

In the next section, you will learn more on how network training is performed in practical situations.

Network Initialization

- At the beginning of the training, the NN parameters must be initialized with *random* values.
- In particular, if all the weights have identical initial values (e.g. zero), all neurons will learn the exact *same* input weights, causing the whole learning to fail.
- Several strategies have been proposed, for a simple network, initialization from a uniform distribution is usually OK.
- Usually, the mean over all weights should be zero. The standard deviation should not be too high (often depends on the layer size).
- Too high/low values could lead to exploding/vanishing gradients.

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Network Design Considerations

Advanced backpropagation

- The basic algorithm requires to fix a learning rate (and batch size)
- The optimal learning rate depends on the task, the data quality, the batch size, the error function, ...
- The optimal batch size depends on the task, the data quality, the learning rate, the error function, ...
- Trial and error: If you see very small error reduction, the learning rate might be too low, if the error fluctuates wildly (or even increases), the learning rate may be too high
- You could also use a learning rate *schedule* (e.g. higher learning rate in the beginning, smaller learning rate for final finetuning)
- If you observe high fluctuation, you may force smoother gradients by averaging the gradient over several batches (*momentum*)
- Several methods have been proposed to *adapt* the learning rate based on the observed convergence, a well-known one is the *Adam* optimizer (Kingma and Ba 2015).

Network Topologies

- The optimal network topology depends on the task, the data quality and the amount of data, ...
- No general rule, but note that if you have more than a few layers, training quality decreases (i.e. the trained network does not perform well).
  - ➔ This is due to the structure of backpropagation (ultimately due to the chain rule), where errors are computed by iterative multiplications: The error norm follows a power law, with gradients either *vanishing* or *exploding*.
  - ➔ This can be avoided by gating techniques, including *Highway Networks* (Srivastava et al. 2015) and *Residual Networks* (He et al. 2016, a special case of Highway networks).
  - ➔ The original gated neural network was the *LSTM* (Hochreiter & Schmidhuber 1997), which we will get to know in the context of *recurrent* neural networks.

Network Topologies

- If the network is too shallow and/or too small (i.e. the number of layers, or the number of neurons per layer is too small), the network tends to *underfit*.
- If the network is too large, it can *overfit* the training data, but in practice this is not such a great problem.
- You will usually make the network perform well on the training data, and then use *regularization* to improve generalization,

Early Stopping

- One of the simplest ways to prevent overfitting the network is to control the error on a separate validation set (we know that from the first lecture).
- When the *validation error* starts to rise, stop training (Early Stopping).
- Note that the error fluctuates a bit: Usually one defines a *patience* (say, 10 epochs) to wait if the validation error might fall again. If the validation error does not fall, select the best performing network so far.

In practice, if your task is small to medium-sized, train with a small number of hidden nodes, then keep doubling until no more significant improvement on the validation set.

Regularization

- The network can be *regularized* in various ways.
- For example, one can penalize the absolute value of the weights, or the sum of their squares (we know that from linear regression):

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \sum_{\lambda} |w_\lambda| \quad \text{or} \quad \tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \sum_{\lambda} |w_\lambda|^2$$

- Another large class of regularization ideas comes from augmenting data by adding noise:
  - ➔ *Input Noise* (e.g. white noise) can be added to the input data
  - ➔ Noise can also be injected into the network in the form of *Dropout*
  - ➔ If we have knowledge of the underlying data, we can use domain-specific noise (e.g. image transformation).

  In all cases, the idea is to artificially create more input samples (which should make sense, of course).

Special Layers

- There exist a variety of methods to help training the neural network. Often, they can be described as special layers (even though they are not really layers).
- As an example, *Batch Normalization* standardizes the input for each layer for *each* mini-batch.
  - ➔ can improve the quality of the solution
  - ➔ often speeds up the training process (less epochs needed)
- It also makes a lot of sense to standardize the input data.

Summary

In this lecture, you should have learned

- the intuition behind a neural network
- the practical implementation (as a series of matrix operations)
- training by *backpropagation* (it is easier than it looks)