Deep Learning and Neural Networks

Demetrio Labate

February 7, 2024

Part 1
Feedforward Neural Networks

Neural Networks

A **neural network** is an algorithm for processing an input $x \in \mathbb{R}^D$ and returning an output in $\mathbb{R}^k$.

Neural Networks

A **neural network** is an algorithm for processing an input $x \in \mathbb{R}^D$ and returning an output in $\mathbb{R}^k$.

The algorithm involves the repeated application of two simple operations:

1. an **affine linear transformation**, that is, a map of the form

$$x \mapsto W(x) = Ax + b$$

2. a **non-linear activation function** ρ applied coordinate-wise.

Neural Networks

A **neural network** is an algorithm for processing an input $x \in \mathbb{R}^D$ and returning an output in $\mathbb{R}^k$.

The algorithm involves the repeated application of two simple operations:

1. an **affine linear transformation**, that is, a map of the form

$$x \mapsto W(x) = Ax + b$$

2. a **non-linear activation function** ρ applied coordinate-wise.

In the simplest case, the two operations are repeated multiple times by composition.

Neural Networks

A **feedforward neural network** processes information by composition

Neural Networks

A **feedforward neural network** processes information by composition

1. Denote the **input** as $\hat{x}^0 = x$

Neural Networks

A **feedforward neural network** processes information by composition

1. Denote the **input** as $\hat{x}^0 = x$
2. For $1 \leq \ell \leq L$, set

$$x^\ell = W^\ell(x^{\ell-1}) = A^\ell \hat{x}^{\ell-1} + b^\ell, \qquad \hat{x}^\ell = \rho(x_\ell)$$

Neural Networks

A **feedforward neural network** processes information by composition

1. Denote the **input** as $\hat{x}^0 = x$
2. For $1 \leq \ell \leq L$, set

$$x^\ell = W^\ell(x^{\ell-1}) = A^\ell \hat{x}^{\ell-1} + b^\ell, \qquad \hat{x}^\ell = \rho(x_\ell)$$

3. The **output** is $\Phi(x) = x^L$

The values x^ℓ, $1 \leq \ell \leq L-1$, which are not seen by a user are the **hidden layers** or **latent variables** of the neural network.

Neural Networks

A **feedforward neural network** processes information by composition

1. Denote the **input** as $\hat{x}^0 = x$
2. For $1 \leq \ell \leq L$, set

$$x^\ell = W^\ell(x^{\ell-1}) = A^\ell \hat{x}^{\ell-1} + b^\ell, \qquad \hat{x}^\ell = \rho(x_\ell)$$

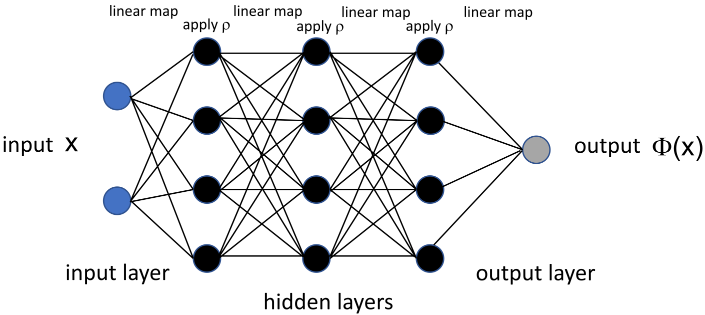
3. The **output** is $\Phi(x) = x^L$

The values x^ℓ, $1 \leq \ell \leq L-1$, which are not seen by a user are the **hidden layers** or **latent variables** of the neural network.

Remark: in practical implementations, the numerical values of the matrices A^ℓ and the biases b^ℓ are **learned** during an appropriate training process.

Neural Networks

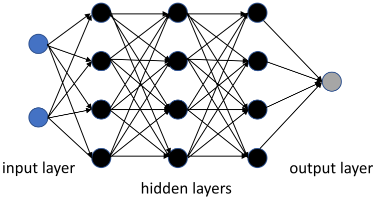
Graphical representation of a feedforward neural network with input $x \in \mathbb{R}^2$, output $\Phi(x) \in \mathbb{R}$ and 4 layers ($L = 4$).



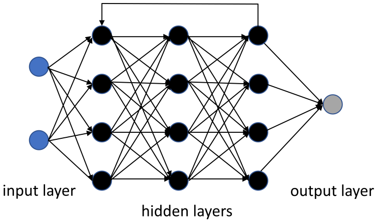
Information flows forward from input to output
⟶ **feedforward architecture**

Neural Networks

In a feedforward neural network, information flows forward from input to output.
When neural networks are extended to include **feedback connections**, they are called **recurrent neural networks**



Feedforward neural network



Recurrent neural network

Feedforward Neural Networks

Feedforward neural networks are also called **multilayer perceptrons** (MLPs) and colloquially referred to as the "vanilla" neural networks.

History note.

- In 1958, a layered network of perceptrons, consisting of an input layer, a hidden layer with randomized weights that did not learn, and an output layer with learning connections, was introduced by Frank Rosenblatt in his book *Perceptron*.
- Here a perceptron is a map of the form $x \mapsto \Phi(x) = \theta(w \cdot x + b)$ where w vector of real-valued weights and θ is the heaviside step-function.
- In 1967, S. Amari first trained a MLP by stochastic gradient descent to classify non-linearly separable pattern classes.

Feedforward Neural Networks

A feedforward neural network with only one hidden layer is a **shallow neural network.**

In this case, it takes a linear map of the input $x \in \mathbb{R}^D$, applies an activation function ρ and finally another linear map:

$$\Phi(x) = A^2(\rho(A^1 x + b^1)) + b^2$$

Feedforward Neural Networks

A feedforward neural network with only one hidden layer is a **shallow neural network.**

In this case, it takes a linear map of the input $x \in \mathbb{R}^D$, applies an activation function ρ and finally another linear map:

$$\Phi(x) = A^2(\rho(A^1 x + b^1)) + b^2$$

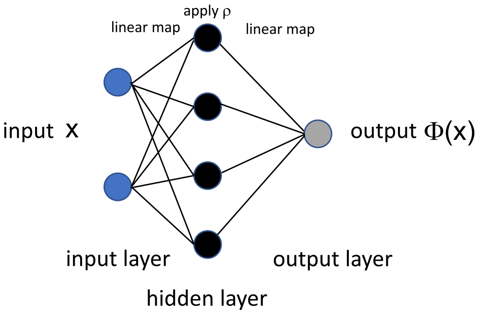
This can be written (for 1-dimensional output) as

$$\Phi(x) = \sum_{i=1}^{n} a_i \, \rho(w_i^t x + b_i)$$

where n is the number of hidden neurons.

Feedforward Neural Networks

Graphical representation of a shallow neural network with input $x \in \mathbb{R}^2$, output $\Phi(x) \in \mathbb{R}$. It only includes 1 hidden layer, so $L = 2$.



$$\Phi(x) = A^2(\rho(A^1 x + b^1)) + b^2$$

Feedforward Neural Networks

A **deep neural network** has several inner layers

$$\Phi(x) = A^4\rho(A^3\rho(A^2\rho(A^1x + b^1) + b^2) + b^3) + b^4$$

Here we have a feedforward neural network with 4 layers and 3 hidden layers.

Feedforward Neural Networks

A **deep neural network** has several inner layers

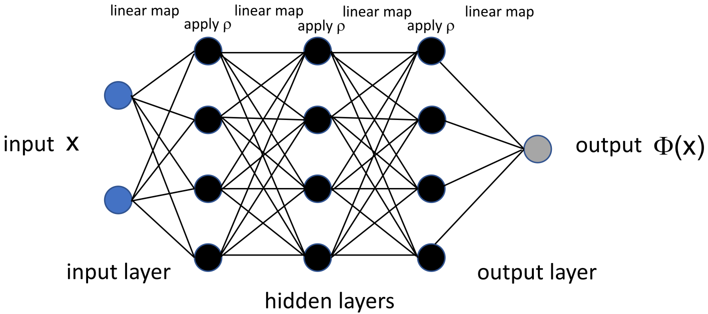
$$\Phi(x) = A^4\rho(A^3\rho(A^2\rho(A^1x + b^1) + b^2) + b^3) + b^4$$

Here we have a feedforward neural network with 4 layers and 3 hidden layers.

Note: for the function Φ to be well defined, the dimensions of $x \in \mathbb{R}^D$, the vectors b^i and the matrices A^i must be matched.

Feedforward Neural Networks

Graphical representation of a deep neural network with input $x \in \mathbb{R}^2$, output $\Phi(x) \in \mathbb{R}$ and 3 inner layers, that is, $L = 4$.



1.1 Approximation properties

Feedforward Neural Networks

A neural network produce a **structured parametric families of functions** of the form

$$\Phi(x) = W^{L} \circ \rho \circ W^{L-1} \circ \ldots \rho \circ W^{1}(x), \quad x \in \mathbb{R}^{D}$$

Feedforward Neural Networks

A neural network produce a **structured parametric families of functions** of the form

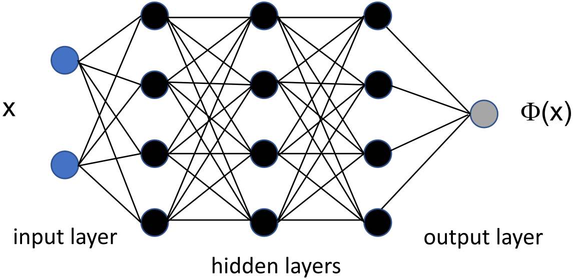
$$\Phi(x) = W^L \circ \rho \circ W^{L-1} \circ \dots \rho \circ W^1(x), \quad x \in \mathbb{R}^D$$

where

- $W^\ell(x) = A^\ell x + b^\ell, \quad \ell = 1, \dots, L$
- $A^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ are the **filters** and $b^\ell \in \mathbb{R}^{N_\ell}$ are the **biases**
- $\rho : \mathbb{R} \to \mathbb{R}$ is the **activation function**
- $L(\Phi)$ is the **number of layers** of Φ
- $N_\ell \in \mathbb{N}$, $i = \ell, \dots, L$ is the **width** of the ℓ-th layer, $N_0 = D$, and $N(\Phi) = \sum_{i=0}^{L} N_i$ is the **number of neurons** of Φ
- $M(\Phi) = \sum_{\ell=1}^{L} ||A^\ell||_0 + ||b^\ell||_0$ is the **number of weights** (or **parameters**) of Φ

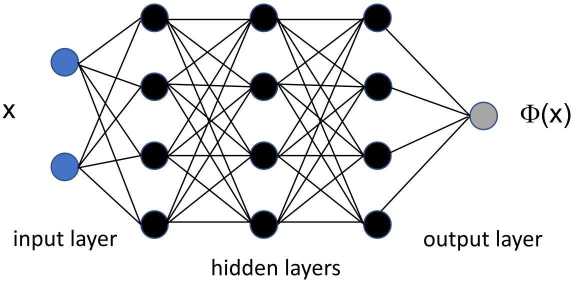
Feedforward Neural Networks

Graph representation



Feedforward Neural Networks

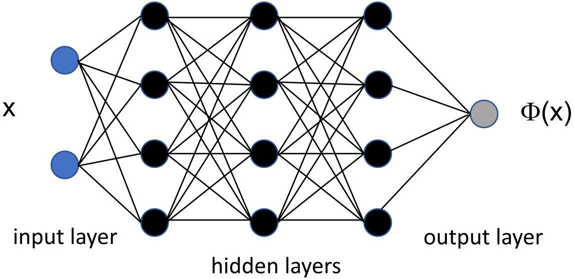
Graph representation



- Number of layers $L = 4$

Feedforward Neural Networks

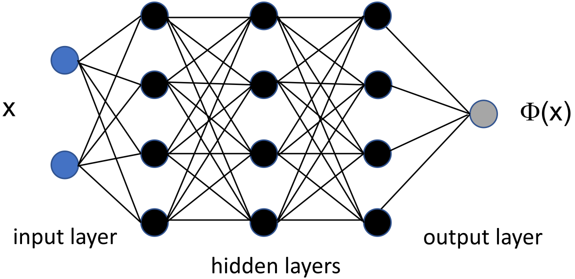
Graph representation



- Number of layers $\quad L = 4$
- Number of neurons $\quad N = 15$

Feedforward Neural Networks

Graph representation



- Number of layers $L = 4$
- Number of neurons $N = 15$
- Number of weights $M = \sum_{\ell=1}^{4} ||A^{\ell}||_0 + ||b^{\ell}||_0 = 44 + 13 = 57$

$$\Phi(x) = W^4(\rho W^3(\rho W^2(\rho W^1(x))))$$

Feedforward Neural Networks

Importance of network parameters.

The number of parameters (also called weights) of a neural network is very important.

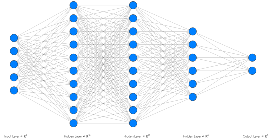
When designing a neural network supervised learning applications, the number of trainable parameters is a hyperparameter affecting the performance of the model.

Designing a deep network with a high number of parameters can be useful to learn more complex models for non-trivial tasks but it can also lead to severe overfitting if the training set is small or the task at hand is simple.

Overfitting happens when the learning model network memorizes the training set instead of learning the patterns of it. Therefore, monitoring the number of trainable parameters of a neural networks is important in neural network design for supervised learning applications.

Feedforward Neural Networks

One can derive a formula to **count the number of parameters** M of a feedforward neural network.


Input Layer ∈ $\mathbb{R}^5$
Hidden Layer ∈ $\mathbb{R}^{10}$
Hidden Layer ∈ $\mathbb{R}^{10}$
Hidden Layer ∈ $\mathbb{R}^6$
Output Layer ∈ $\mathbb{R}^2$

M depends on the number of layers and neurons at each layer.

Observation: For 2 fully connected layers with number of neurons N_i and N_{i+1}, the number of trainable parameters is

$$(N_i + 1) * N_{i+1}$$

where the $+1$ term in the equation takes into account the bias terms.

Note: Software to draw neural networks: `http://alexlenail.me/NN-SVG/`

Feedforward Neural Networks

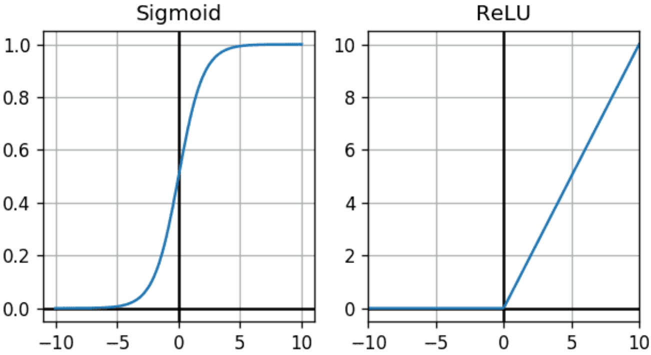
Examples of activation functions:

- **Sigmoid**: $\rho(x) = \frac{1}{1+e^{-x}}$
- **Rectified linear unit (ReLU)**: $\rho(x) = \max\{x, 0\}$

Feedforward Neural Networks

Examples of activation functions:

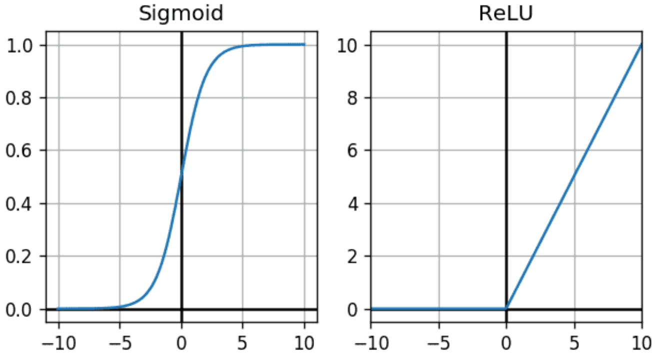
- **Sigmoid**: $\rho(x) = \frac{1}{1+e^{-x}}$
- **Rectified linear unit (ReLU)**: $\rho(x) = \max\{x, 0\}$



Feedforward Neural Networks

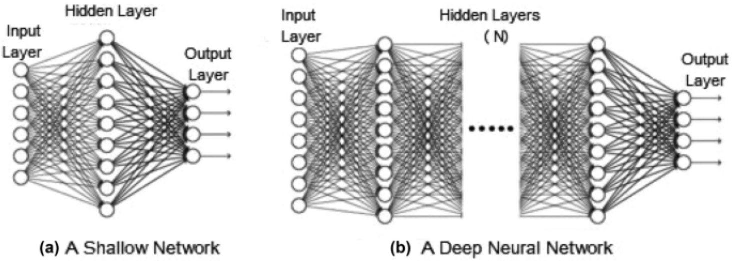
Examples of activation functions:

- **Sigmoid**: $\rho(x) = \frac{1}{1+e^{-x}}$
- **Rectified linear unit (ReLU)**: $\rho(x) = \max\{x, 0\}$



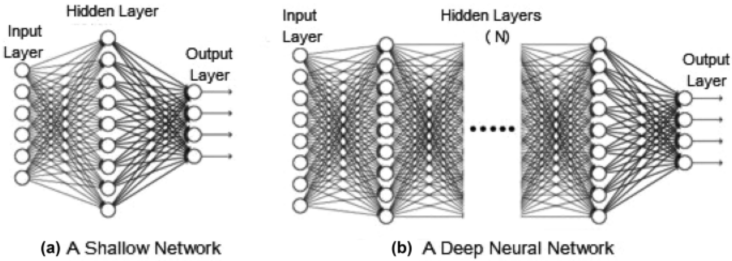
- ReLU is the most common activation function in applications
- The sigmoid is differentiable

Deep Neural Networks



- Modern network architectures are typically very deep

Deep Neural Networks

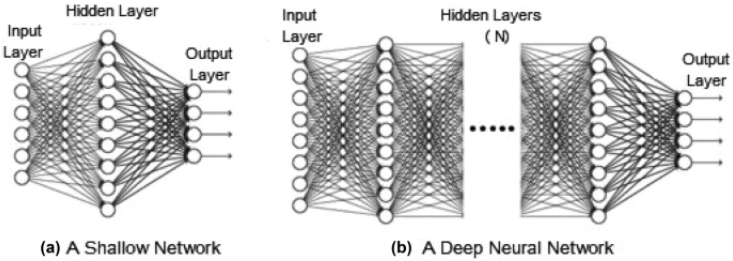


(a) A Shallow Network

(b) A Deep Neural Network

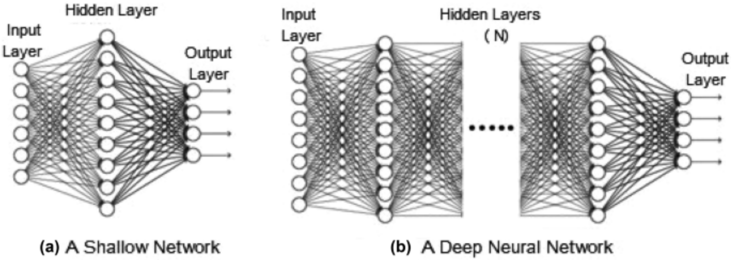
- Modern network architectures are typically very deep
- Depth improves expressive power

Deep Neural Networks



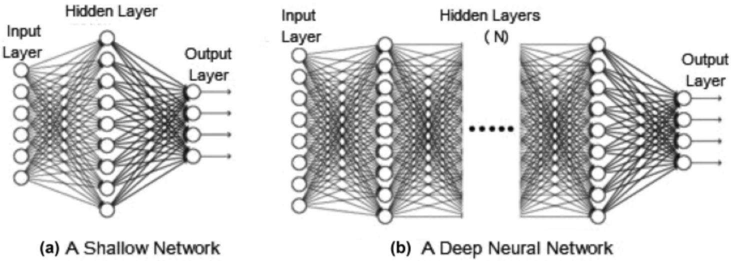
- Modern network architectures are typically very deep
- Depth improves expressive power
- With respect to shallow networks, deep neural networks can exploit **composition**

Deep Neural Networks



- Modern network architectures are typically very deep
- Depth improves expressive power
- With respect to shallow networks, deep neural networks can exploit **composition** → **Blessing of compositionality**

Deep Neural Networks



- Modern network architectures are typically very deep
- Depth improves expressive power
- With respect to shallow networks, deep neural networks can exploit **composition** → **Blessing of compositionality**
- Input and output can be multi-dimensional in general.

Note: Software to draw neural networks: `http://alexlenail.me/NN-SVG/`

Universal approximations

Theoretical results demonstrate the ability of feedforward neural networks to **implement large classes of multivariate functions**.

Universal approximation theorems have historically been used as a justification of the expressive power of neural networks.

Universal approximations

Theoretical results demonstrate the ability of feedforward neural networks to **implement large classes of multivariate functions**.

Universal approximation theorems have historically been used as a justification of the expressive power of neural networks.

Definition.

A class of functions $\mathcal{F}$ is called a **universal approximator** over a compact set S, e.g., $S = [0,1]^D$, if, for every continuous function g and target accuracy $\epsilon > 0$, there exists $f \in \mathcal{F}$ such that

$$\sup_{x \in S} |f(x) - g(x)| < \epsilon.$$

Universal approximations

Universal approximation theorems state roughly that: **regardless of what function we are trying to learn, a sufficiently large feedforward neural network will be able to represent this function**.

Theorem (Hornik, 1991)

Assume ρ is a C^{∞} non-polynomial function. Then the class of shallow neural networks is a universal approximator over $[0,1]^D$.

Note: the network may become arbitrarily wide.

Universal approximations

Universal approximation theorems state roughly that: **regardless of what function we are trying to learn, a sufficiently large feedforward neural network will be able to represent this function**.

Theorem (Hornik, 1991)

Assume ρ is a C^∞ non-polynomial function. Then the class of shallow neural networks is a universal approximator over $[0,1]^D$.

Note: the network may become arbitrarily wide.

Theorem [Kidger and Lyons, 2020]

Assume ρ is a nonaffine continuous function which is continuously differentiable at least one point, with nonzero derivative at that point. Then the class of deep neural networks where the number of neurons N_ℓ for each layer ℓ can be bounded by $D + d + 2$ where d is the output dimension is a universal approximator over $[0,1]^D$.

Note: the network may become arbitrarily deep.

Universal approximations

Remarks

- Universal approximation theorems indicate that neural networks have the ability to accurately approximate any continuous multivariate function.

Universal approximations

Remarks

- Universal approximation theorems indicate that neural networks have the ability to accurately approximate any continuous multivariate function.
- The representation power of a neural network increases with the number of neurons and layers.

Universal approximations

Remarks

- Universal approximation theorems indicate that neural networks have the ability to accurately approximate any continuous multivariate function.
- The representation power of a neural network increases with the number of neurons and layers.
- Warning: even if a neural network is able to represent a function, the training algorithm may fail to learn that specific function.

Universal approximations

Remarks

- Universal approximation theorems indicate that neural networks have the ability to accurately approximate any continuous multivariate function.
- The representation power of a neural network increases with the number of neurons and layers.
- Warning: even if a neural network is able to represent a function, the training algorithm may fail to learn that specific function.
- Learning can fail (i) because the optimization algorithm used for training may not be able to find the value of the parameters corresponding to the desired function or (ii) because the training algorithm might choose the wrong function as a result of overfitting.

Example: piecewise linear functions on $\mathbb{R}$

Triangle function:

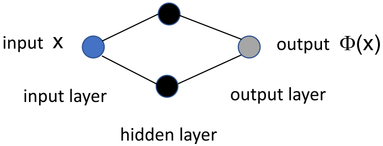
$$T(x) = \begin{cases} 2x & \text{if } 0 \le x < \frac{1}{2} \\ 2(1-x) & \text{if } \frac{1}{2} \le x \le 1 \end{cases} \qquad x \in \mathbb{R},$$

Example: piecewise linear functions on $\mathbb{R}$

Triangle function:

$$T(x) = \begin{cases} 2x & \text{if } 0 \leq x < \frac{1}{2} \\ 2(1-x) & \text{if } \frac{1}{2} \leq x \leq 1 \end{cases} \qquad x \in \mathbb{R},$$

I claim that T can be expressed using a shallow ReLU feedforward neural network with 1 hidden layer of width $N_1 = 2$

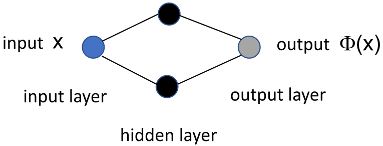


Example: piecewise linear functions on $\mathbb{R}$

Triangle function:

$$T(x) = \begin{cases} 2x & \text{if } 0 \le x < \frac{1}{2} \\ 2(1-x) & \text{if } \frac{1}{2} \le x \le 1 \end{cases} \qquad x \in \mathbb{R},$$

I claim that T can be expressed using a shallow ReLU feedforward neural network with 1 hidden layer of width $N_1 = 2$



We have a neural network with 7 parameters:

$$\Phi(x) = \begin{bmatrix} a_{21} & a_{22} \end{bmatrix} \rho\left(\begin{bmatrix} a_{11} \\ a_{12} \end{bmatrix} x + \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix}\right) + b_2,$$

where $\rho(\alpha x + \beta) = (\alpha x + \beta)_+$

Example: piecewise linear functions on $\mathbb{R}$

We can write explicitly

$$\Phi(x) = a_{21}(a_{11}x + b_{11})_+ + a_{22}(a_{12}x + b_{12})_+ + b_2.$$

Example: piecewise linear functions on $\mathbb{R}$

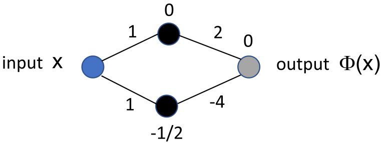
We can write explicitly

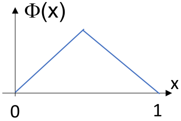
$$\Phi(x) = a_{21}(a_{11}x + b_{11})_+ + a_{22}(a_{12}x + b_{12})_+ + b_2.$$

We can solve it as

$$\Phi(x) = 2(x - 0)_+ - 4(x - \tfrac{1}{2})_+$$

where $(g(x))+$ denotes the non-negative part of $g(x)$



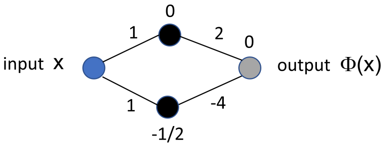


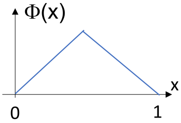
Note that each term in $\Phi(x)$ is associated with a linear section of the piece-wise linear function.

Example: piecewise linear functions on $\mathbb{R}$

The expression below is exactly matched to the graphical representation of the neural network with each summand identifying one branch

$$\Phi(x) = 2(x-0)_+ - 4(x-\tfrac{1}{2})_+$$

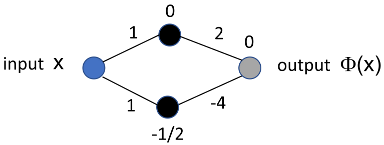


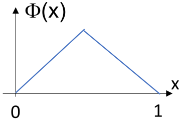


Example: piecewise linear functions on $\mathbb{R}$

The expression below is exactly matched to the graphical representation of the neural network with each summand identifying one branch

$$\Phi(x) = 2(x-0)_+ - 4(x - \tfrac{1}{2})_+$$



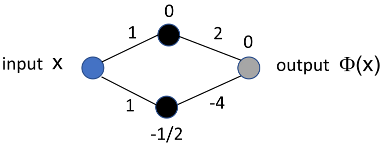


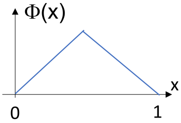
- One can write any triangle function on an interval using a similar network with $N_1 = 2$.

Example: piecewise linear functions on $\mathbb{R}$

The expression below is exactly matched to the graphical representation of the neural network with each summand identifying one branch

$$\Phi(x) = 2(x-0)_+ - 4(x-\tfrac{1}{2})_+$$



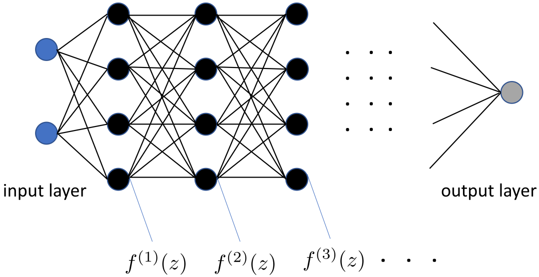


- One can write any triangle function on an interval using a similar network with $N_1 = 2$.
- One can write any piecewise linear function on a compact domain with N_1 linear sections using N_1 neurons.

More about it in the homework

Representation power of neural networks

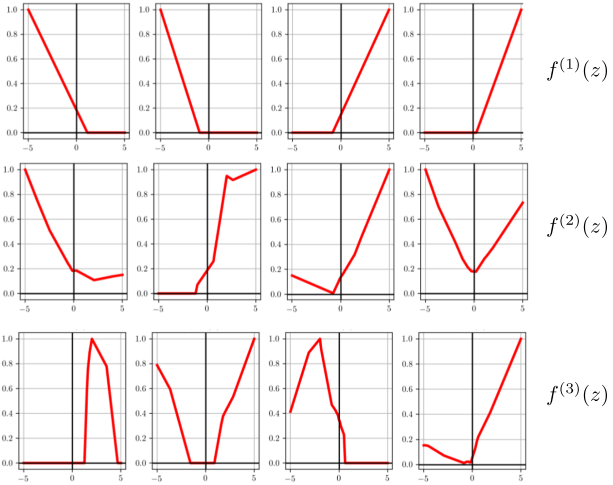
Using functional composition, feedforward neural networks can represent more complex information.



We are going to show the output function of neurons in different layers.

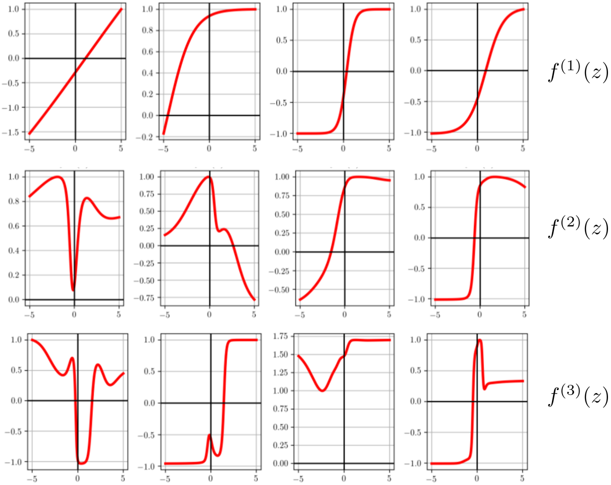
Representation power of neural networks

Here we show the output function of neurons in the first 3 layers, for random inputs, using the ReLU activation function



Representation power of neural networks

Here we show the output function of neurons in the first 3 layers, for random inputs, using the tanh activation function



1.2 Supervised Learning

Feedforward Neural Networks - Machine Learning

Feedforward Neural Networks can be adapted to various **supervised learning** setups including the following.

Feedforward Neural Networks - Machine Learning

Feedforward Neural Networks can be adapted to various **supervised learning** setups including the following.

- Univariate and multivariate **regression**. You want to predict the values of a function

  $$y = f(x), \quad x \in \mathbb{R}^d$$

  Example: predicting values of multivariate continuous function.

Feedforward Neural Networks - Machine Learning

Feedforward Neural Networks can be adapted to various **supervised learning** setups including the following.

- Univariate and multivariate **regression**. You want to predict the values of a function

  $$y = f(x), \quad x \in \mathbb{R}^d$$

  Example: predicting values of multivariate continuous function.

- Binary and multilabel **classification**. You want to predict whether an instance $x \in \mathbb{R}^d$ is associated to a label $y \in \{0, 1, \dots, m\}$.
  Example: predicting whether a patient has diabetic retinopathy or not, that is, $y \in \{0, 1\}$.
  Example: predicting the class of an handwritten digit in the label set $\{0, 1, 2, \dots, 9\}$.

Feedforward Neural Networks - Machine Learning

The output layer of a Feedforward Neural Network is constrained by the type of problem that you are modeling.

- **Regression problem.** Output is a single neuron and the neuron may have no activation function.
- **Binary classification problem.** Output is a single neuron and uses a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the class 1. This can be turned into a class assignment by using a threshold of 0.5 and assigning values less than the threshold to 0, otherwise to 1.
- **Multi-class classification problem.** Output layer may have multiple neurons, one for each class. In this case, a softmax activation function may be used to output a probability of the network predicting each of the class values.

Feedforward Neural Networks - Machine Learning

- **Binary classification.** The sigmoid (also called logistic function)

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

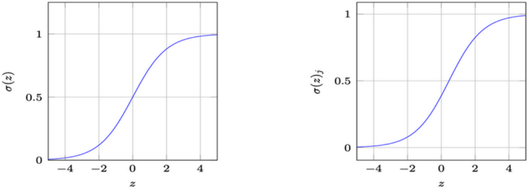
maps the output to values between zero and one. A threshold set to 0.5 would assign samples of outputs larger or equal 0.5 to the positive class, and the rest to the negative class.

- *K*-**class classification.** The softmax function is a vector valued function with components

$$s(x)_i = \frac{e^{-x_i}}{\sum_{k=1}^{K} e^{-x_k}}$$

where $x = (x_1, \ldots, x_K)$ The result is a vector containing the probabilities that a sample belongs to each class; the output is the class with the highest probability.

Feedforward Neural Networks - Machine Learning



(a) Sigmoid activation function.

(b) Softmax activation function.

- Given an input, the sigmoid maps the input in the interval $[0, 1]$. This property works well for binary classification.

- In the multi-class problem, we want the outputs for each of the class to be between 0 and 1 and the sum of these scores to be 1. A sigmoid activation function does not satisfy these properties.

- Softmax is an activation function applied on top of Logits (the outputs from the final layer) to get final scores (or probabilities) so that final scores are between 0 and 0 and their total sum is 1.

Feedforward Neural Networks - Machine Learning

To train a neural network, we need to define a **loss function** and find **parameters** that minimize the loss on the training data.

MLPs uses different loss functions depending on the problem type.

- For **regression**, MLPs typically use the **Mean Square Error** loss function.
- For **classification**, MLPs typically use the **Average Cross-Entropy** loss function.

In most implementations, the loss function also includes a **regularization** term (also known as **penalty**) that penalizes complex models.

Feedforward Neural Networks - Machine Learning

Let us consider the regression problem first.

Feedforward Neural Networks - Machine Learning

Let us consider the regression problem first.

• Let $\{(x_i, y_i) : i = 1, \ldots, m\}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$, be a training set.

Feedforward Neural Networks - Machine Learning

Let us consider the regression problem first.

• Let $\{(x_i, y_i) : i = 1, \ldots, m\}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$, be a training set.

• The prediction of the Neural Network on a single input is the result of the forward pass that we denote as

$$\mathcal{N}(x_i, \theta) = \hat{y}_i,$$

where θ is a parameter vector comprising all coefficients of the filters and the biases.

Feedforward Neural Networks - Machine Learning

To control the prediction error in the regression problem, a standard choice is to use the **square error**.

- For a single sample, we have

$$L(\hat{y}_i, y_i; \theta) = \frac{1}{2}(\mathcal{N}(x_i, \theta) - y_i)^2 = \frac{1}{2}(\hat{y}_i - y_i)^2.$$

- Averaging over the training set (x, y), we **mean square loss** is

$$L(\hat{y}, y; \theta) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2.$$

- To train the Neural Network for regression, we need to find the minimum of L as a function of the parameter vector θ.

This minimization problem has no analytic solution in general and need to be solved numerically.

Feedforward Neural Networks - Machine Learning

To improve the numerical convergence of the minimization problem, the loss function often includes a **regularization term**:

$$L(\hat{y}, y; \theta) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i(\theta) - y_i)^2 + \frac{\alpha}{2m}||\theta||_2^2,$$

that that penalizes more complex models.

- $||\theta||_2^2 = \sum_i \theta_i^2$ increases with the number and the size of the parameters of the MLP. Hence, having more non-zero parameters increases the loss.
- α is non-negative hyperparameter that controls the magnitude of the penalty.

Feedforward Neural Networks - Machine Learning

Let us now consider the classification problem. For simplicity we consider a binary classification problem.

Feedforward Neural Networks - Machine Learning

Let us now consider the classification problem. For simplicity we consider a binary classification problem.

• Let $\{(x_i, y_i) : i = 1, \ldots, m\}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{0, 1\}$, be a training set. Note that y_i can only take two possible values.

Feedforward Neural Networks - Machine Learning

Let us now consider the classification problem. For simplicity we consider a binary classification problem.

- Let $\{(x_i, y_i) : i = 1, \ldots, m\}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{0, 1\}$, be a training set. Note that y_i can only take two possible values.

- As above, the prediction of the Neural Network on a single input is the result of the forward pass that we denote as

$$\mathcal{N}(x_i, \theta) = \hat{y}_i,$$

where θ is a parameter vector comprising all coefficients of the filters and the biases.

Feedforward Neural Networks - Machine Learning

To control the prediction error in classification problem, a standard approach is to use the notion of **cross entropy**.

Given a true distribution t and a predicted distribution p, the **cross entropy** between them is given by the equation

$$H(t,p) = -\sum_{s \in S} t(s) \log p(s)$$

where S is the support of the probabilities.

This quantity can be used as a measure of error for categorical multi-class classification problems.

Feedforward Neural Networks - Machine Learning

For a binary classification problem, the **binary cross entropy** is

$$H(t,p) = -t \log p + (1-t)\log(1-p)$$

where p is the predicted probability and t is either $t = 0$ or $t = 1$.

Example:
Suppose the correct target is $t = 1$.
Then $H(t,p) = -\log p$.

The binary cross entropy will reward giving a correct prediction:

- p closer to 1 will give a lower loss ($H(t,p) = 0$ if $p = 1$);
- p closer to 0 will give a higher loss.

The behavior is the same when the correct target is $t = 0$.

Feedforward Neural Networks - Machine Learning

To control the prediction error in the binary classification problem, we use the **Average Cross-Entropy** loss function. For a sample set of size m, it is given by

$$L(\hat{y}, y; \theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right),$$

where y_i is the true label of sample i and $\hat{y}_i$ is the corresponding probability value (= the output of the sigmoid function).

By the observation above, this loss function is larger when there is miss-match between y_i and $\hat{y}_i$.

Feedforward Neural Networks - Machine Learning

For the K-class classification problem, for a sample set of size m, the **Average Cross-Entropy** loss function, is given by

$$L(\hat{y}, y; \theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{K} \left(y_i(c) \log \hat{y}_i(c)\right),$$

where y_i is the true label of sample i and $\hat{y}_i$ is the corresponding probability ($=$ the output of the softmax function).

The Average Cross-Entropy is larger when there is miss-match between y_i and $\hat{y}_i$.

Feedforward Neural Networks - Machine Learning

Example:
Suppose that, for a specific training instance, the true label is B, out of the possible labels A, B, and C.
Hence, for this instance, the correct distribution is $y = (0, 1, 0)$

Suppose the MLP predicts the probability distribution
$\hat{y} = (0.228, 0.619, 0.153)$

The Average Cross-Entropy is
$L = -(0.0 * \ln(0.228) + 1.0 * \ln(0.619) + 0.0 * \ln(0.153)) = 0.479$

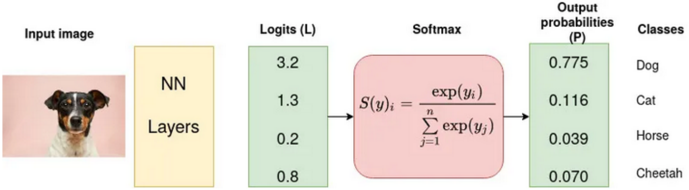
Next suppose the MLP predicts the new probability distribution
$\hat{y} = (0.001, 0.998, 0.001)$

The Average Cross-Entropy is now
$L = -(0.0 * \ln(0.001) + 1.0 * \ln(0.998) + 0.0 * \ln(0.001)) = 0.002$

The distance of the predicted probability from the true probability is very small.

Feedforward Neural Networks - Machine Learning

Example: 4-class classification task where an image is classified as either a dog, cat, horse or cheetah.



MLP outputs logits and the Softmax function converts logits into probabilities.

The cross-entropy
$H = -(1.0 * \ln(0.775) + 0 * \ln(0.116) + 0 * \ln(0.039) + 0 * \ln(0.070) = 0.255$
gives the distance from the true probability.
Note that $y = (1, 0, 0, 0)$ in the example.

Feedforward Neural Networks - Machine Learning

Regularization:

Also in this case, similar to the square loss for the regression problem, the loss function typically includes a **regularization term** to penalize more complex models:

$$L(\hat{y}, y; \theta) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)) + \frac{\alpha}{m} ||\theta||_2^2$$

$$L(\hat{y}, y; \theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{K} (y_i(c) \log \hat{y}_i(c)) + \frac{\alpha}{m} ||\theta||_2^2$$

Feedforward Neural Networks - Machine Learning

- To compute a classification or regression model using a MLP, given a collection of training samples, we need to to find the minimum of the loss function as a function of the weights and biases.

- MLP with hidden layers have a **non-convex loss function** where there exists more than one local minimum.
Therefore different random weight initializations can lead to different validation accuracies.

1.3 Gradient Descent and Backpropagation

Neural Networks - Gradient Descent

To mimimize the loss function of a MLP, the standard procedures are **Stochastic Gradient Descent, Adam, or L-BFGS**.

- **Stochastic Gradient Descent** is a variation of Gradient Descent that updates the model parameters based on the gradient of the cost function for a single training example, rather than the average gradient over the entire training set. It starts at a random point on the cost function surface and moves in the direction of the steepest descent, iteratively updating the parameters using samples of the training set.
- **Adam** is similar to Stochastic Gradient Descent in a sense that it is a stochastic optimizer, but it can automatically adjust the amount to update parameters based on adaptive estimates of lower-order moments.
- **L-BFGS** is a second-order optimization algorithm that approximates the Hessian matrix representing the second-order partial derivative of a function. It also approximates the inverse of the Hessian matrix to perform parameter updates.

Neural Networks - Gradient Descent

The minimum of the loss function of a MLP is not very easy to locate because it is not an easy function to differentiate.

In a MLP, the output is dependent on all the weights, so the error obtained at the last output node is also dependent on all the weights.

To compute the derivative of the loss function with respect to the weights, we need to backpropagate the error all the way to the input node from the output node.

Backpropagation is a method to estimate the gradient of a neural network model.

Backpropagation

Learning occurs in a feedforward neural network by changing the weights after each piece of data is processed, based on the amount of error in the output compared to the expected result.
This is an example of supervised learning, and is carried out through **backpropagation**.

We can represent the degree of error in an output node j in the n-nth data point (training sample) by

$$e_j(n) = d_j(n) - y_j(n)$$

where $d_j(n)$ is the desired target value for n-nth data point at node j and $y_j(n)$ is the value produced by the neural network at node j when the n-nth data point is given as an input.

Backpropagation

The node weights can then be adjusted based on corrections that minimize the **error function** $E(n)$ in the entire output for the n-nth data point, given by

$$E(n) = \frac{1}{2} \sum_{\text{output node } j} e_j^2(n)$$

Using the method of gradient descent, the change in each weight W_{ji} is

$$\Delta W_{ji}(n) = -\eta \frac{\partial E(n)}{\partial v_j} y_i(n)$$

where $y_i(n)$ is the output of the previous neuron i, η is the learning rate and $v_j(n)$ is the weighted sum of the input connections of neuron j.

The learning rate is a hyperparameter selected to ensure that the weights quickly converge to a response without oscillations.

Backpropagation

The calculation of the derivative $\frac{\partial E(n)}{\partial v_j}$ above depends on the induced local field v_j which itself varies.

For an **output node** the derivative can be simplified to

$$\frac{\partial E(n)}{\partial v_j} = -e_j(n)\rho^{'}(v_j(n))$$

where $\rho^{'}$ is the derivative of the activation function

For a **hidden node** the derivative can be simplified to

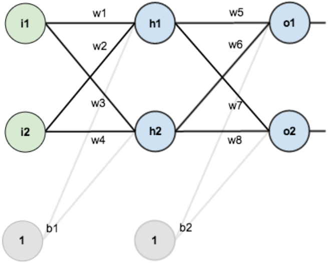
$$\frac{\partial E(n)}{\partial v_j} = -\rho^{'}(v_j(n))\sum_k \frac{\partial E(n)}{\partial v_k} W_{kj}(n)$$

where the k-th nodes represent the output layer.

In sum, to change the hidden layer weights, the output layer weights change according to the derivative of the activation function; hence this algorithm represents a backpropagation of the activation function

Backpropagation - Example

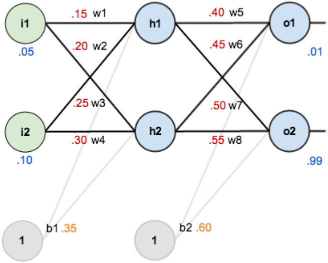
I will illustrate backpropagation on an simple neural network with a single hidden layer.



Input and output are 2-dimensional; additionally, there is a bias.

Backpropagation - Example

The network is initialized as below

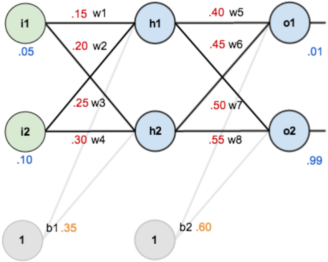


There is a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

We assume $\rho(x) = \frac{1}{1+e^{-x}}$. Note that $\rho'(x) = \rho(x)(1 - \rho(x))$.
We assume for the learning rate: $\eta = 0.5$.

Backpropagation - Example

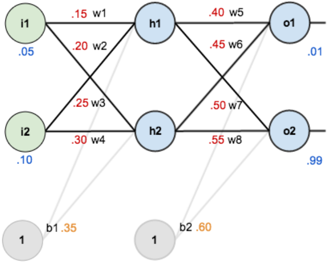
We start running a Forward Pass to compute what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10.



input to node h_1:
$v_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1 = 0.15*0.05+0.2*0.1+0.35*1 = 0.3775$
output from node h_1:
$y_{h1} = \rho(v_{h1}) = 0.593269992$

Backpropagation - Example

Similarly



input to node h_2:
$v_{h2} = w_3 * i_1 + w_4 * i_2 + b_1 * 1 = 0.25 * 0.05 + 0.3 * 0.1 + 0.35 * 1 = 0.3925$
output from node h_2:
$y_{h2} = \rho(v_{h2}) = 0.596884378$

Backpropagation - Example

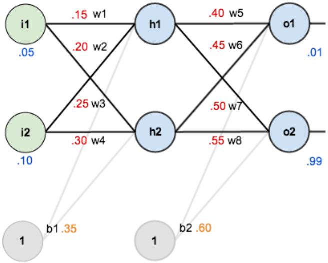
We repeat the same calculation for the output layer.

$v_{o1} = w_5 * o_1 + w_6 * 0_2 + b_2 * 1 =$
$0.4 * 0.593269992 + 0.45 * 0.5968843781 + 0.6 * 1 = 1.105905967$
$y_{o1} = \rho(v_{o1}) = 0.75136507$

$v_{o2} = w_7 * o_1 + w_8 * 0_2 + b_2 * 1 = (\ldots)$
$y_{o2} = \rho(v_{o2}) = 0.772928465$

Backpropagation - Example

We now calculate the error at the output.



$$e_{o1} = 0.01 - 0.75136507 = -0.74136507$$
$$e_{o2} = 0.99 - 0.772928465 = 0.217071535$$

Hence we have the total error

$$E = \frac{1}{2}(e_{o1}^2 + e_{o2}^2) = 0.298371109$$

Backpropagation - Example

The goal of backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output.

We apply the formulas stated above with learning rate $\eta = 0.5$. We first consider the weights of the output layer.

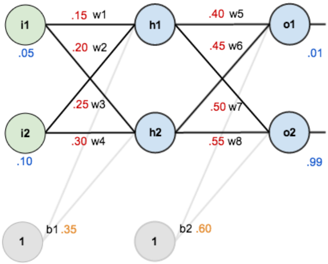
$$\frac{\partial E}{\partial v_{oi}} = -e_{oi}\,\rho^{'}(v_{oi})$$

Then

$$\frac{\partial E}{\partial v_{o1}} = -e_{o1}\,\rho^{'}(v_{o1}) = -e_{o1}\,\rho(v_{o1})(1-\rho(v_{o1})) = 0.1384985619$$
$$\frac{\partial E}{\partial v_{o2}} = -e_{o2}\,\rho^{'}(v_{o2}) = -e_{o2}\,\rho(v_{o2})(1-\rho(v_{o2})) = -0.03809823661$$

Backpropagation - Example

Observing that w_5 connects the nodes h_1 to o_1,



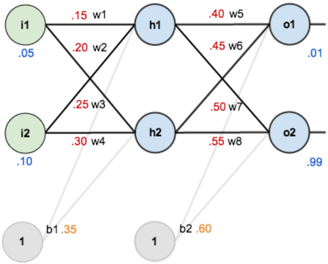
$$\Delta w_5 = -\eta \frac{\partial E}{\partial v_{o1}} y_{h1} = -0.04108352024$$

Hence the updated weight w_5 is
$w_5^+ = w_5 + \Delta w_5 = 0.3589164798$

Similarly we find
$w_6^+ = 0.408666186$, $w_7^+ = 0.511301270$, $w_8^+ = 0.561370121$.

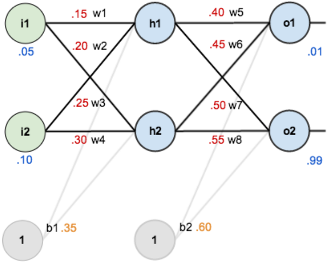
Backpropagation - Example

We next consider the hidden layer.



$$
\begin{aligned}
\frac{\partial E}{\partial v_{h1}} &= -\rho'(v_{h1}) \sum_k \frac{\partial E}{\partial v_k} W_{kj} \\
&= -\rho(v_{h1})(1-\rho(v_{h1}))\left(\frac{\partial E}{\partial v_{o1}} w5 + \frac{\partial E}{\partial v_{o2}} w6\right) \\
&= 0.2413007086(0.1384985619 * 0.4 - 0.03809823661 * 0.45) \\
&= 0.00923101128
\end{aligned}
$$

Backpropagation - Example



$$\Delta w_1 = -\eta \frac{\partial E}{\partial v_{h1}} y_{i1} = -0.5 * 0.00923101128 * 0.05 = -0.0002307753$$

$$w_1^+ = w1 + \Delta w_1 = 0.1497692247$$

Similarly we find

$w_2^+ = 0.19956143$, $w_3^+ = 0.24975114$, $w_4^+ = 0.29950229$

Backpropagation - Example

We have now updated all of the weights.

If we now run another Forward Pass with the updated weights, we find that the new total error is $E^{+} = 0.291027924$

As compared with the original error $E = 0.298371109$, we have reduced the error by 0.007343185.

It might not seem like much, but error is decreasing.

After repeating this process 10,000 times, the error plummets to 0.0000351085.

At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

See https://github.com/mattm/simple-neural-network

1.4 Implementation

MLP - Implementation from scratch

We introduce a notation that will be convenient for numerical implementation.

For a single perceptron with input $x \in \mathbb{R}^D$, we can write the one-dimensional output as

$$f(x) = \rho\left(w_0 + \sum_{i=1}^{D} w_i x_i\right) = \rho\left(w_0, \ldots, w_D\right) \cdot \tilde{x},$$

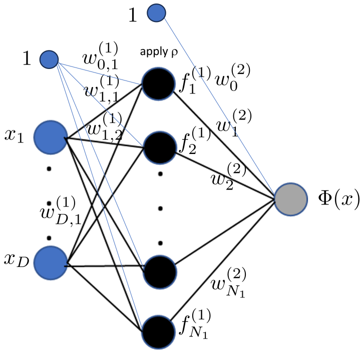
where $\tilde{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_D \end{pmatrix}$

Note that we have denoted the bias as w_0.
With this notation, we can write the **affine transformation** on x as a **linear transformation** on $\tilde{x}$.

MLP - Implementation from scratch

For an MLP with a single hidden layer and N_1 neurons, the output at each neuron j is

$$f_j^{(1)}(x) = \rho\left(w_{0,j}^{(1)} + \sum_{i=1}^{D} w_{i,j}^{(1)} x_i\right)$$

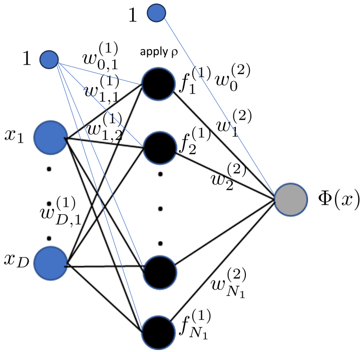


MLP - Implementation from scratch

The output of the hidden layer is the vector

$$f^{(1)}(x) = \rho\left(W^{(1)}\tilde{x}\right)$$

where $W^{(1)}$ is the transpose of the matrix with entries $(w_{i,j}^{(1)})$, for $i = 0, 1, \ldots D$, $j = 1, \ldots, N_1$.

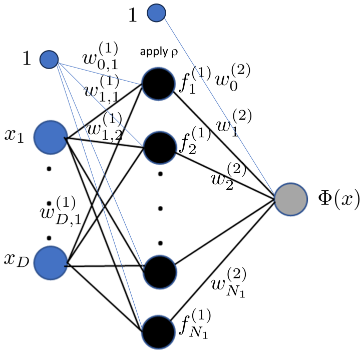


MLP - Implementation from scratch

The output is

$$\Phi(x) = W^{(2)}\rho\left(W^{(1)}\tilde{x}\right) + w_0^{(2)},$$

where $W^{(2)} = (w_1^{(2)}, \ldots, w_{N_1}^{(2)})$.

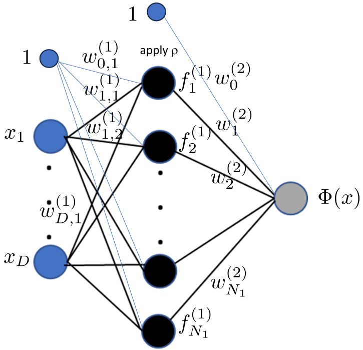


MLP - Implementation from scratch

The output can also be written as

$$\Phi(x) = W^{(2)}\tilde{\rho}\left(W^{(1)}\tilde{x}\right),$$

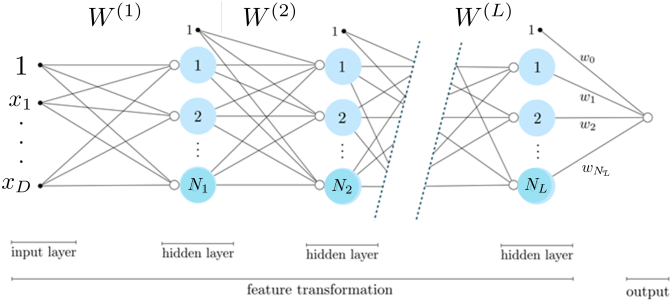
where $W^{(2)} = (w_0^{(2)}, w_1^{(2)}, \ldots, w_{N_1}^{(2)})$ and $\tilde{\rho}\left(W^{(1)}\tilde{x}\right) = \begin{pmatrix} 1 \\ W^{(1)}\tilde{x} \end{pmatrix}$



MLP - Implementation from scratch

This notation is very convenient.
For an MLP with L hidden layer, we can write the output as

$$\Phi(x) = W^{(L+1)}\tilde{\rho}\left(W^{(L)}\tilde{\rho}\left(W^{(L-1)}\tilde{\rho}\left(\dots W^{(1)}\tilde{x}\right)\dots\right)\right)$$



MLP - Implementation from scratch

We can implement in Python a module returning Φ as a function of the input data, and w, a tensor of weights $W^{(1)}$ through $W^{(L)}$

```
# choose a nonlinear activation function
def activation(t):
nonlinearity = np.tanh(t)
return nonlinearity

# fully evaluate our network features using the tensor of weights in w
def feature_transforms(a, w):
# loop through each layer matrix
for W in w:
# pad with ones (to take care of bias) for next layer computation
o = np.ones((1,np.shape(a)[1]))
a = np.vstack((o,a))

# compute inner product with current layer weights
a = np.dot(a.T, W).T


# output of layer activation

a = activation(a)

return a
```

MLP - Implementation from scratch

We create initial weights for our feedforward network and generate architectures with arbitrary numbers of layers

```
# create initial weights for arbitrary feedforward network
def initialize_network_weights(layer_sizes, scale):
# container for entire weight tensor
weights = []

# loop over desired layer sizes and create appropriately sized initial
# weight matrix for each layer
for k in range(len(layer_sizes)-1):
# get layer sizes for current weight matrix
N_k = layer_sizes[k]
N_k_plus_1 = layer_sizes[k+1]
# make weight matrix
weight = scale*np.random.randn(N_k+1,N_k_plus_1)
weights.append(weight)

# re-express weights so that w_init[0] = omega_inner contains all
# internal weight matrices, and w_init = w contains weights of
# final linear combination in predict function
w_init = [weights[:-1],weights[-1]]


return w_init
```

MLP - Implementation

A Python implementation of MLPs is available from `scikit-learn.org`: Multi-layer Perceptron

This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support.

MLP - Applications

MLPs are capable of approximating various **non-linear functions** so that they can perform non-linear regression and non-linear classification.

I will show some simple applications.

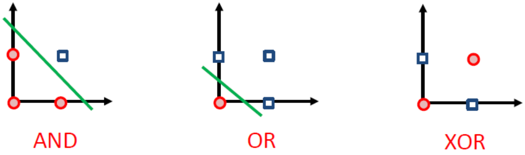
MLP - Applications

MLPs are capable of approximating various **non-linear functions** so that they can perform non-linear regression and non-linear classification.

I will show some simple applications.

Recall that the perceptron can only model linearly separable functions.

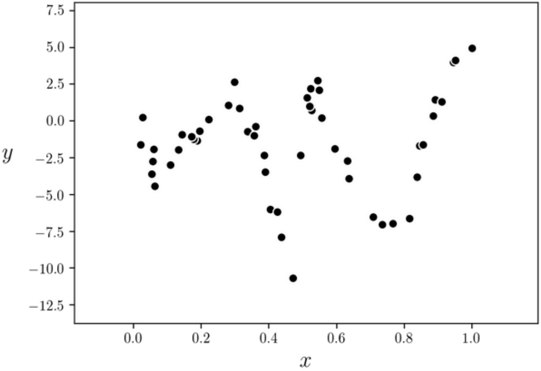
It can model the AND and OR logical functions but not the XOR function.



MLP - Applications

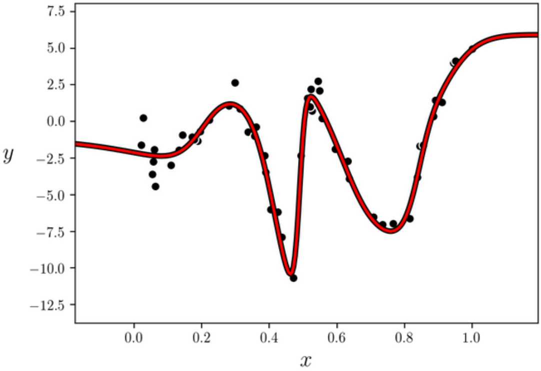
Example: Nonlinear regression using single hidden layer.

We want to approximate the noisy data below.



MLP - Applications

We apply an MLP with a single hidden layer with $N = 100$ neurons using the tanh activation. We run of 5,000 steps of gradient descent.

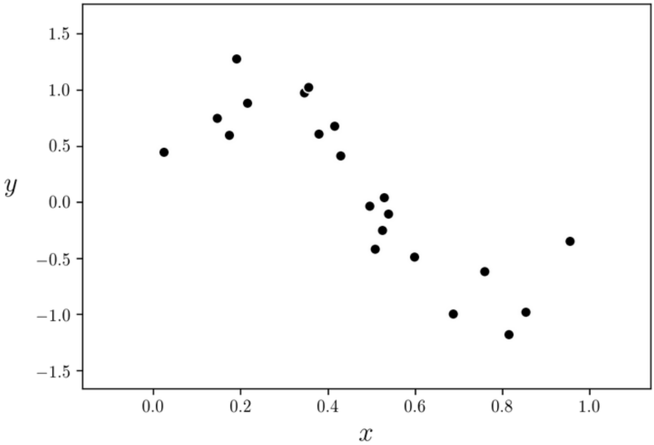


Questions: How is the approximation performance impacted by N? by the activation function?

MLP - Applications

Example: Nonlinear regression using MLP

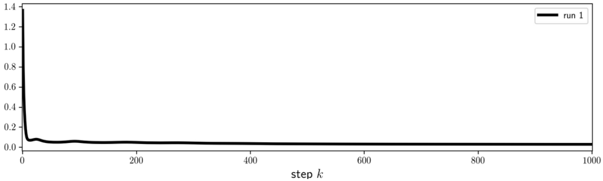
We want to approximate the noisy data below.



MLP - Applications

Here we use a 3 layer MLP with 10 units in each layer.
This model will overfit the dataset if we tune the parameters well.

We run the gradient descent for 1000 iterations with $\eta = 0.1$. We can then examine the cost function history to make sure gradient descent is converging properly.

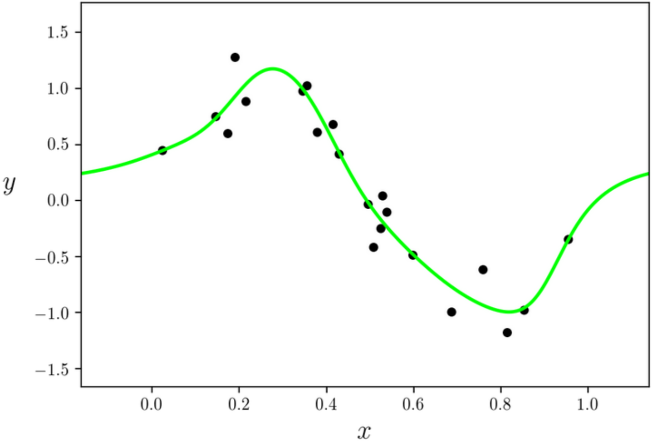


The loss (or cost) function plot decays rapidly.

MLP - Applications

Here is the nonlinear regression fit

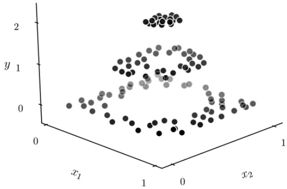
We overfit which is not a surprise since our network is very flexible.

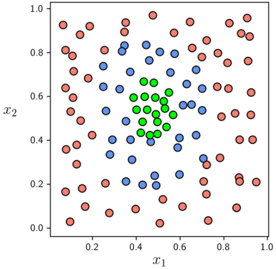


MLP - Applications

Example: Nonlinear classification using MLP

We want to classify the following 3-class data set





MLP - Applications

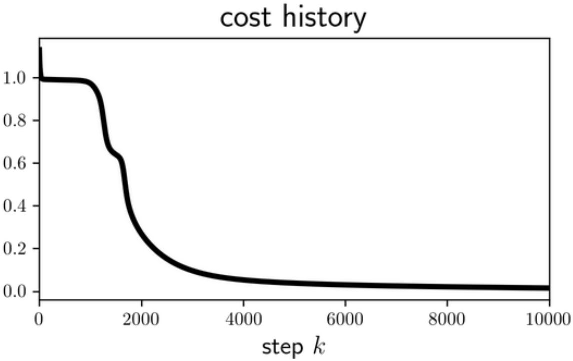
To solve this classification problem, we use a MLP with

- Number of hidden layers: 2
- Number of units per layer: $N_1 = 12$, $N_2 = 5$
- Activation: tanh
- Loss function: Multi-class Softmax
- Optimizer: gradient descent

MLP - Applications

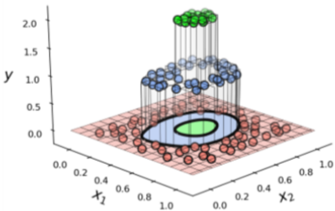
We train the network running gradient descent for 10,000 steps with learning rate η = 0.1

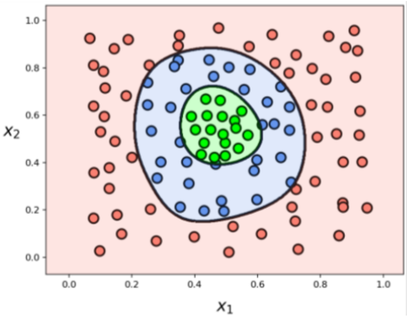
We see that the loss (or cost) function converges after about 5000 steps.



MLP - Applications

Below we plot the resulting fit, visualizing the surface fit and - simultaneously - the nonlinear boundary below.

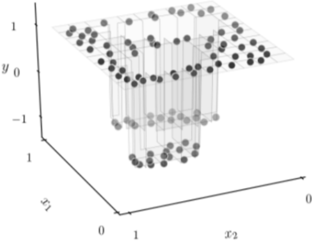


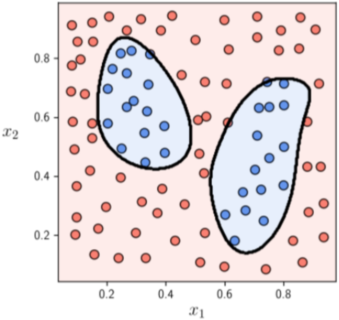


MLP - Applications

Example: Nonlinear two-class classification using MLP

We want to classify the following 2-class data set





MLP - Applications

Example: Nonlinear two-class classification using MLP

We use a MLP with

- Number of hidden layers: 4
- Number of units per layer: $N_i = 10, i = 1, \ldots, 4$
- Activation: tanh
- Loss function: Multi-class Softmax
- Optimizer: gradient descent