Deep Network Developments

Lecture #5

Viktor Varga
Department of Artificial Intelligence, ELTE IK

Software for neural networks

NumPy

CPU only

PyTorch

CPU & GPU
computational graphs
automatic differentiation

Software for neural networks

Both supports array programming!

NumPy

PyTorch

CPU only

CPU & GPU
computational graphs
automatic differentiation

Logistic regression - NumPy vs. PyTorch

```
def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z) + self.eps)

h = self.__sigmoid(np.dot(X, self.theta))
```

label prediction in NumPy

```
def __sigmoid(self, z):
        return 1 / (1 + torch.exp(-z) + self.eps)

h = self.__sigmoid(torch.mm(X, self.theta[:, None]))
```

label prediction in PyTorch

Logistic regression - NumPy vs. PyTorch

```
def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z) + self.eps)

h = self.__sigmoid(np.dot(X, self.theta))
```

label prediction in NumPy

Much simpler with torch.nn:

```
z = torch.nn.Linear(X.shape[1], 1)(X)
h = torch.nn.functional.sigmoid(z)
```

label prediction in PyTorch

Logistic regression - NumPy vs. PyTorch

```
loss = np.mean(-y * np.log(h + self.eps) - \
            (1 - y) * np.log(1 - h + self.eps))
```

loss value in NumPy

```
loss = torch.mean(-y * torch.log(h + self.eps) - \
            (1 - y) * torch.log(1 - h + self.eps))
```

loss value in PyTorch

Logistic regression - NumPy vs. PyTorch

```
loss = np.mean(-y * np.log(h + self.eps) - \
               (1 - y) * np.log(1 - h + self.eps))
```

loss value in NumPy

Much simpler with torch.nn:

```
loss = torch.nn.BCELoss()(h, y)
```

loss value in PyTorch

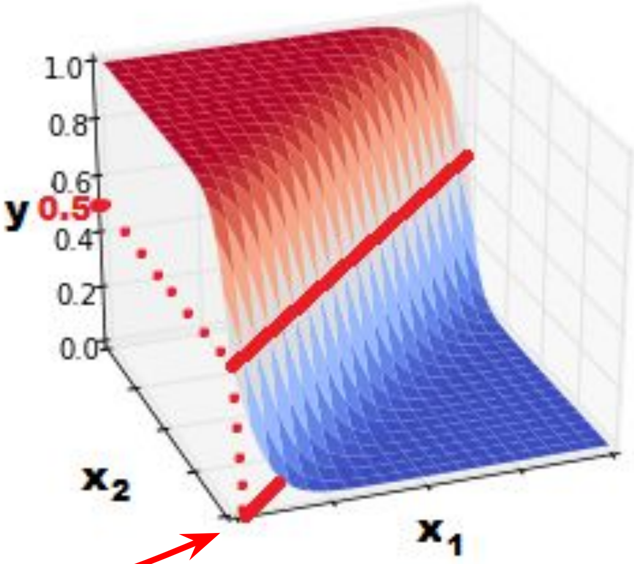
The artificial neuron model

What can a single neuron (with sigmoid) **represent?**

The artificial neuron model

What can a single neuron
(with sigmoid) **represent?**

A single linear decision surface
(since we are talking about
logistic regression).



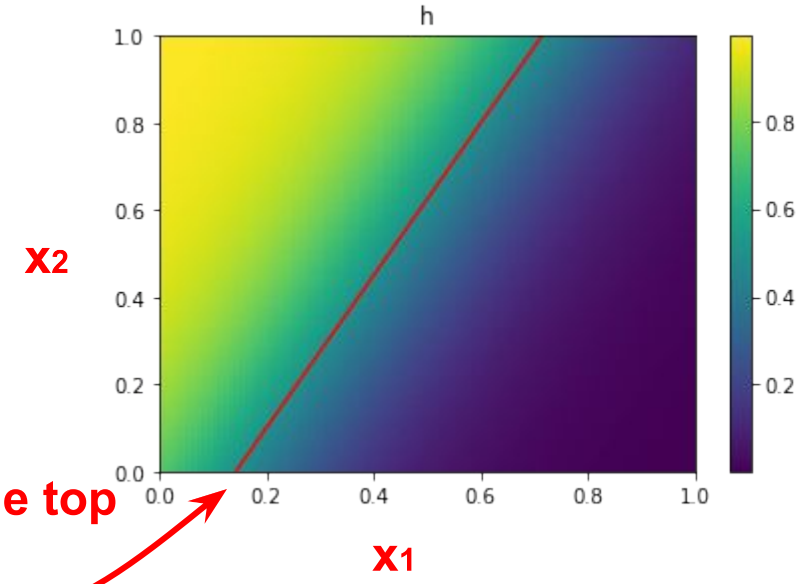
Decision boundary

In case of **two input variables** (x_1, x_2)
this is a line in the **x_1, x_2** plane**.**

The artificial neuron model

What can a single neuron (with sigmoid) **represent?**

A single linear decision surface (since we are talking about logistic regression).

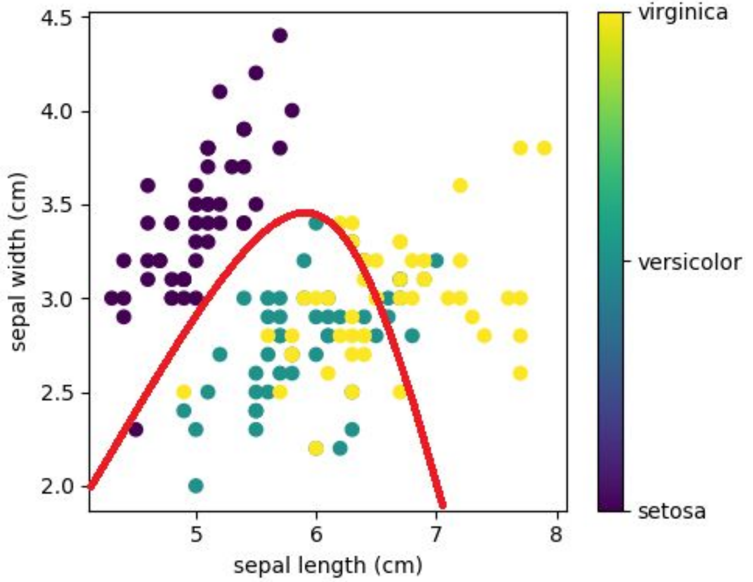


The same graph viewed from the top

Decision boundary

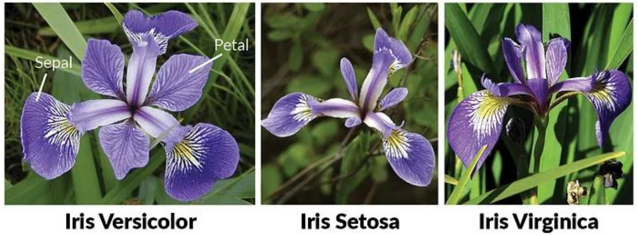
The artificial neuron model

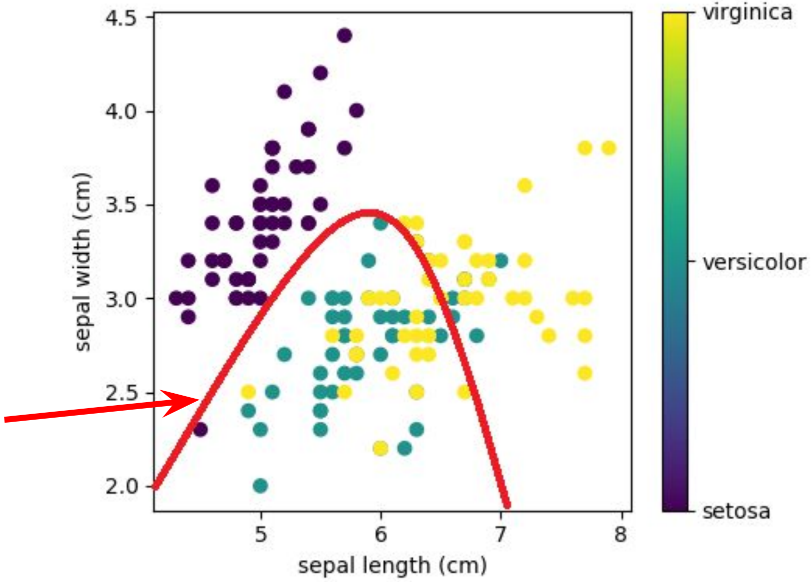
IRIS dataset: Let’s try to separate the data points in the *“versicolor”* category!



The artificial neuron model

IRIS dataset: Let's try to separate the data points in the *"versicolor"* category!





IRIS dataset: Classification of three varieties of Iris flowers based on petal (or sepal) length and width. (This is not an image dataset.)

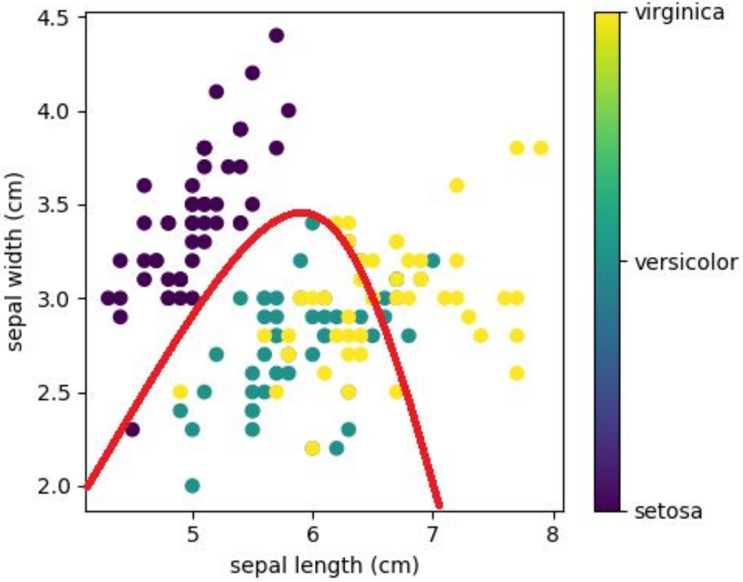
A straight line is not sufficient to distinguish the "versicolor" variety...

The artificial neuron model

IRIS dataset: Let's try to separate the data points in the *"versicolor"* category!

An artificial neuron is not enough!

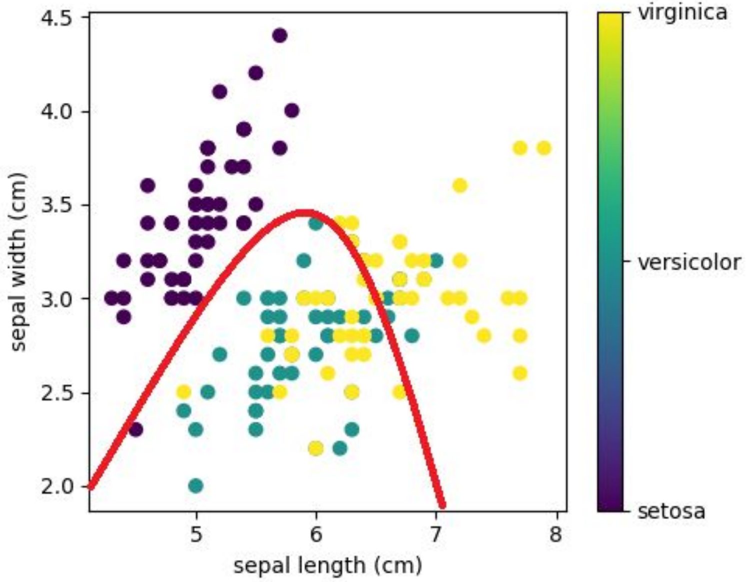
The decision boundary
(a straight line, generally a hyperplane) represented by **a neuron is not able to separate** the points belonging to individual categories!



The artificial neuron model

IRIS dataset: Let's try to separate the data points in the *"versicolor"* category!

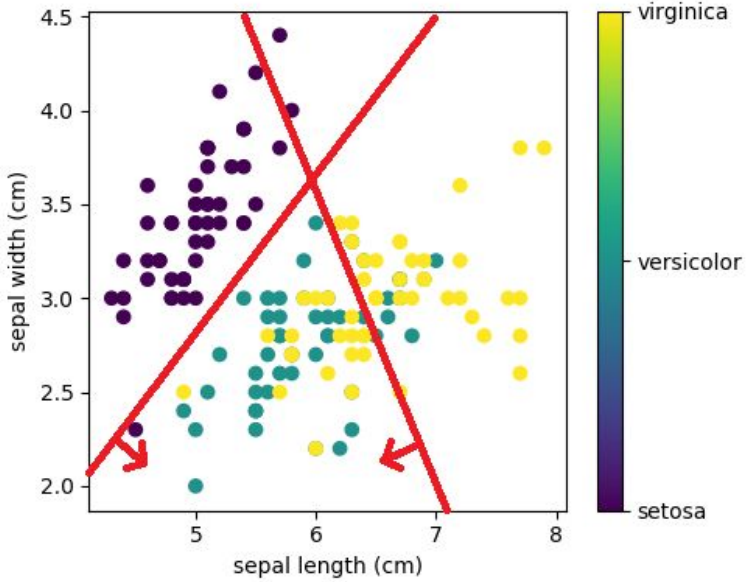
How to solve this task?



The artificial neuron model

IRIS dataset: Let’s try to separate the data points in the *“versicolor”* category!

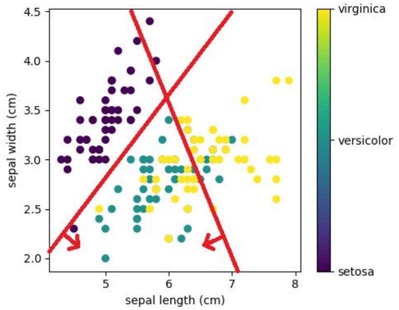
How to solve this task?



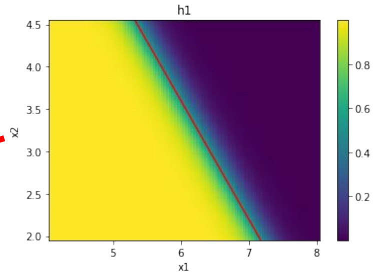
Perhaps, with **two linear decision surfaces and their "combination"** (AND)

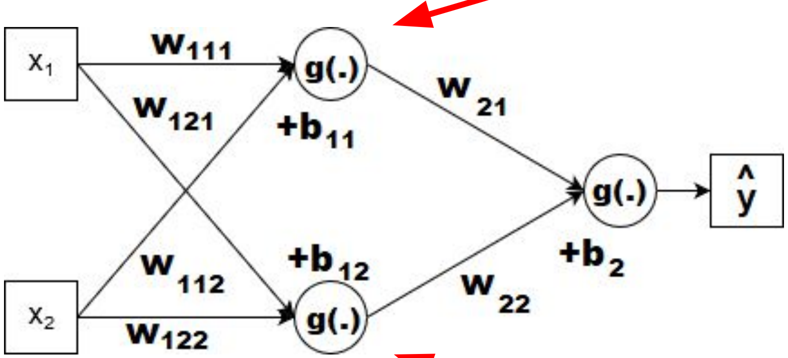
→ we could **connect neurons sequentially**… (with neural networks!)

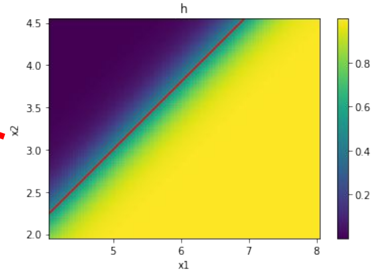
The expressive power of neural networks - An example



$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$

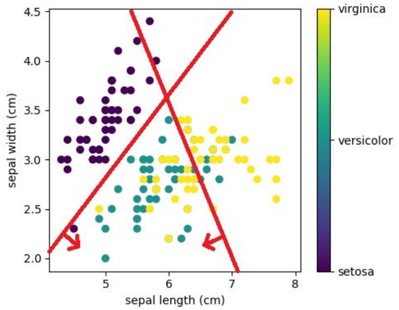




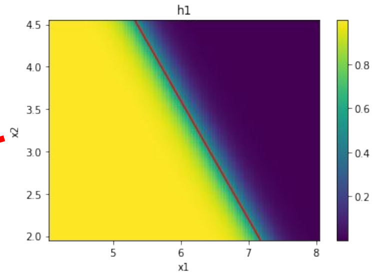


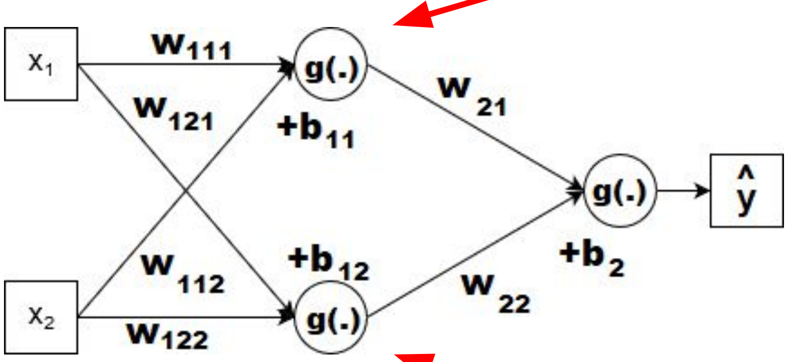
$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

The expressive power of neural networks - An example



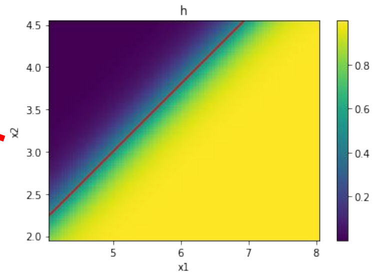
$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$





We want our single neuron in the second layer to output 1 where both neurons in the first layer output 1.

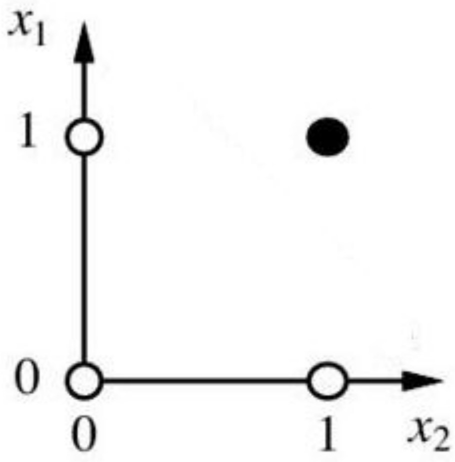
In other words, we need something like an AND operation...



$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

The expressive power of neural networks - An example

Approximation of binary logical functions: x_1 AND x_2

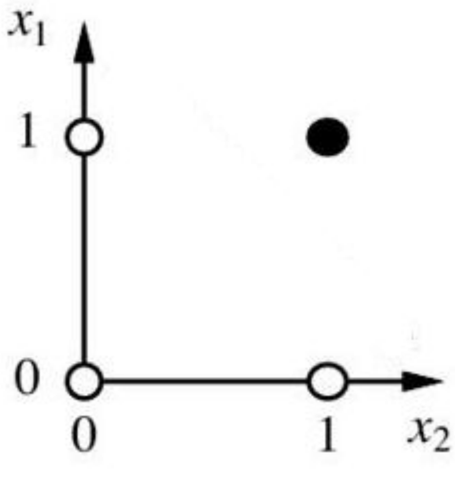


x_1 **and** x_2

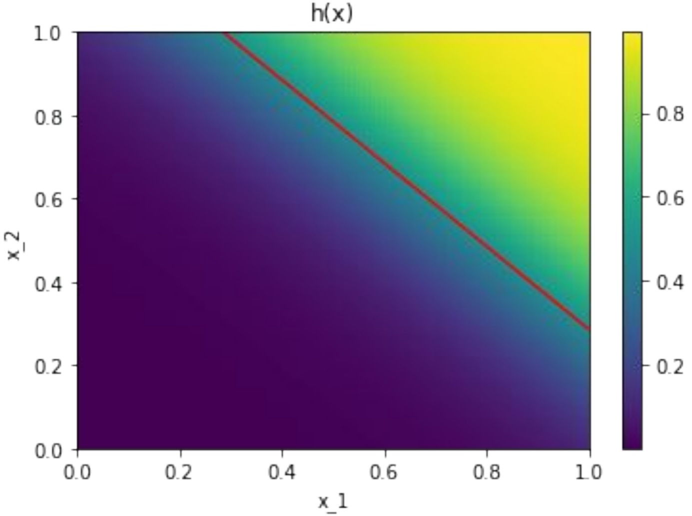
Where should the decision boundary be?

The expressive power of neural networks - An example

Approximation of binary logical functions: x_1 AND x_2



x_1 **and** x_2

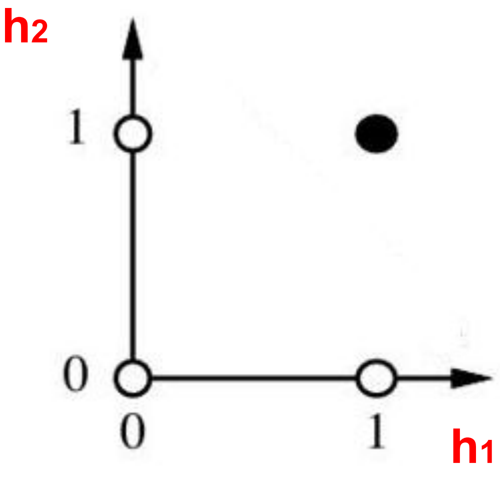


$$w_1 = 7,\ w_2 = 7,\ b = -9$$

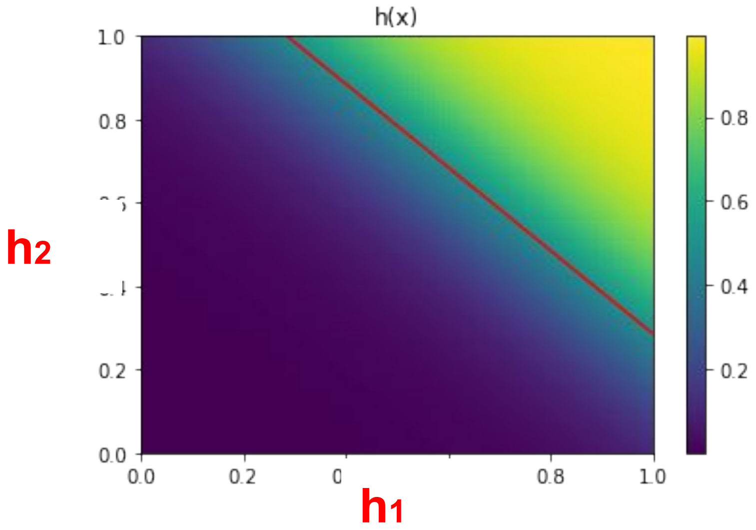
Instead of x_1 and x_2, the output of the two neurons in the first layer will be the input (e.g., h_1, h_2).

The expressive power of neural networks - An example

Approximation of binary logical functions: x_1 AND x_2



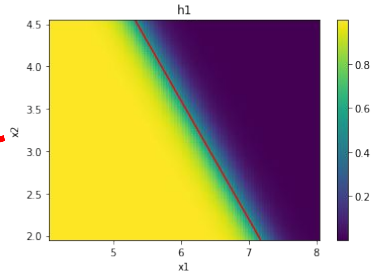
h_1 and h_2

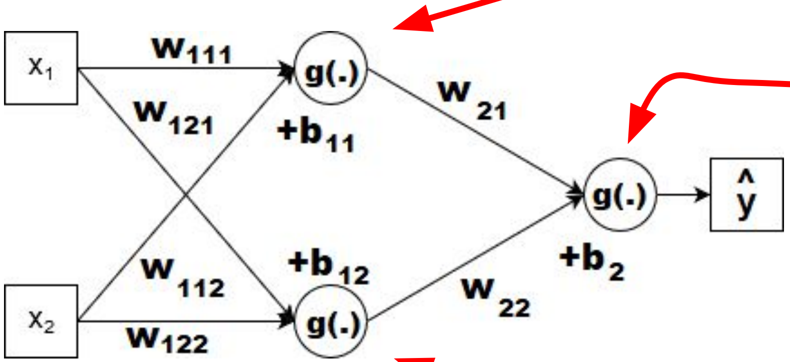


$$w_1 = 7,\ w_2 = 7,\ b = -9$$

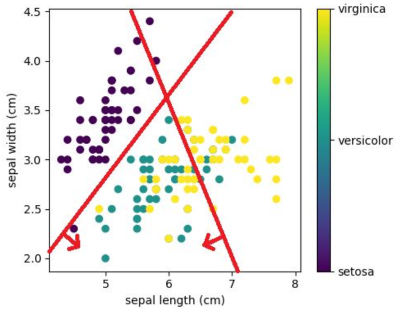
The expressive power of neural networks - An example

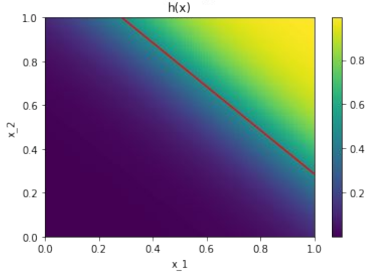
$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$

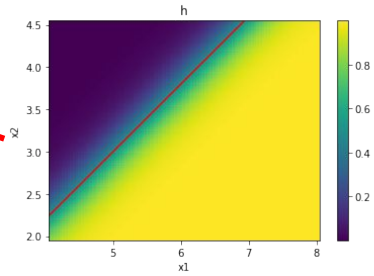




$$w_{21} = 7,\ w_{22} = 7,\ b_2 = -9$$



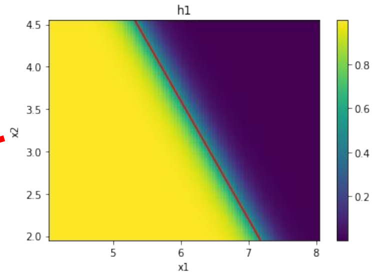


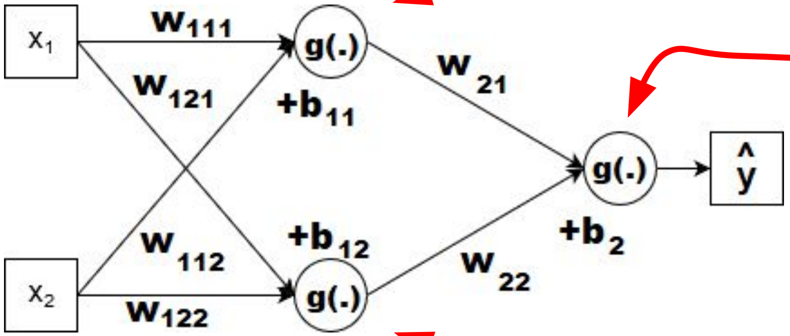


$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

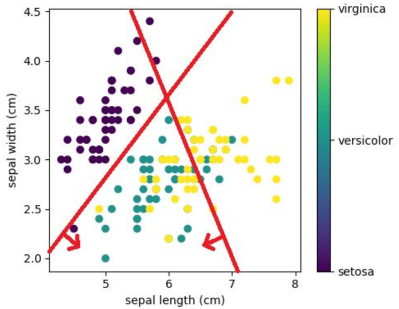
The expressive power of neural networks - An example

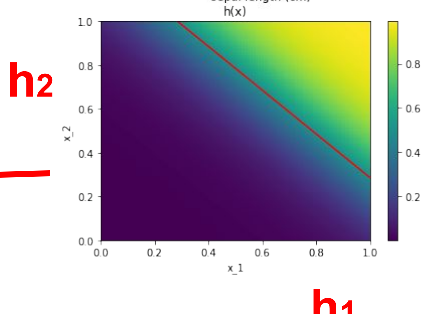
$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$

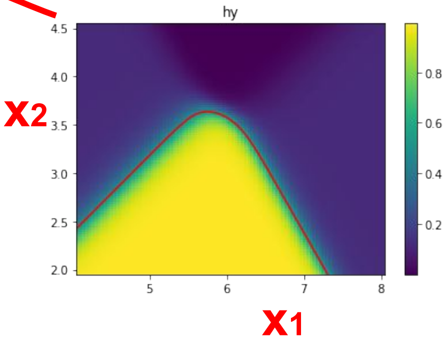


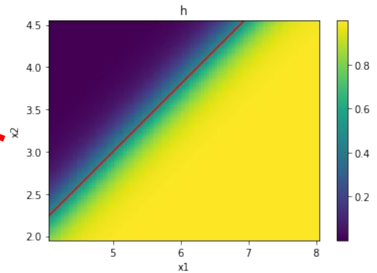


$$w_{21} = 7,\ w_{22} = 7,\ b_2 = -9$$





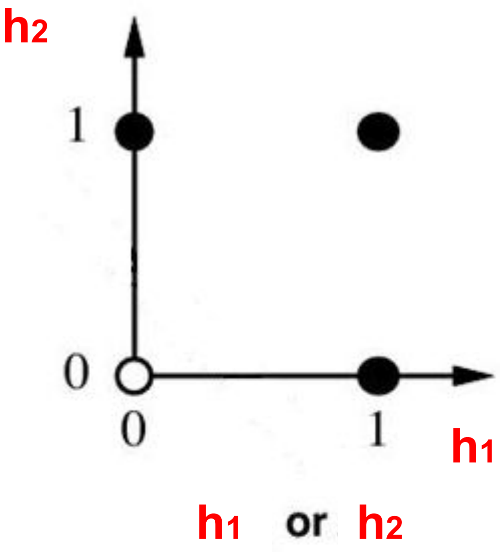


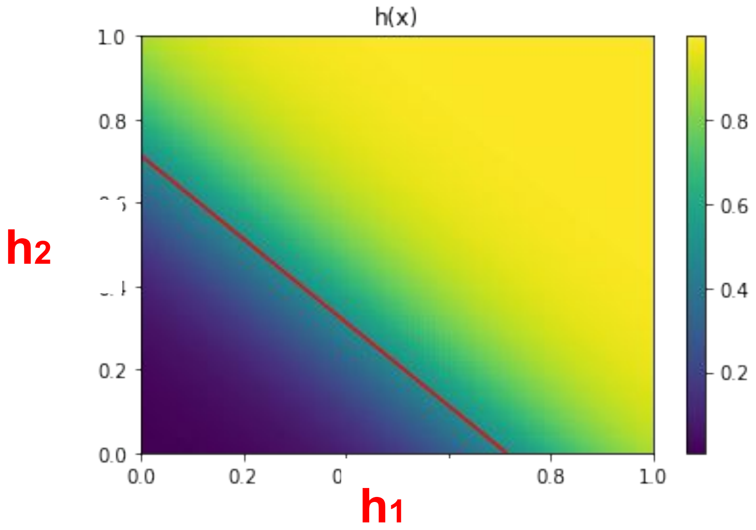


$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

The expressive power of neural networks - An example

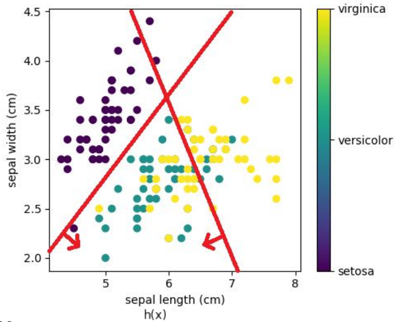
Approximation of binary logical functions: x_1 OR x_2



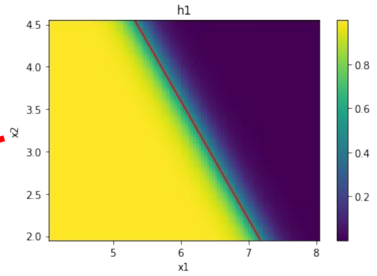


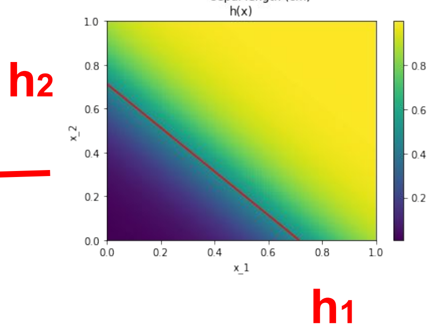
$$w_1 = 7,\ w_2 = 7,\ b = -5$$

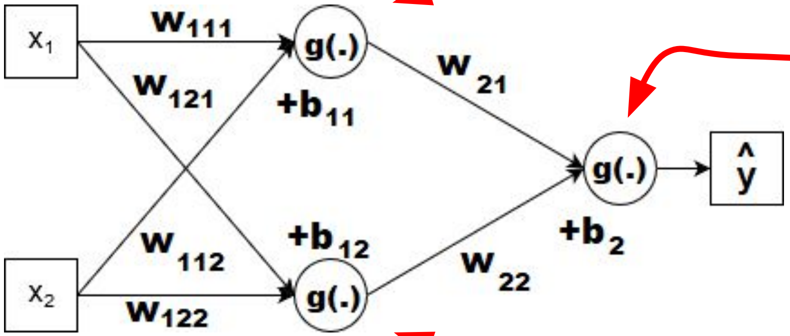
The expressive power of neural networks - An example



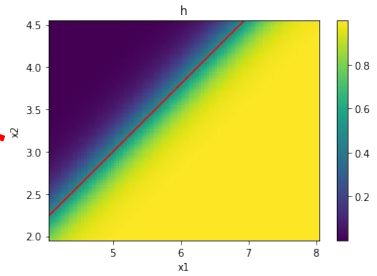
$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$

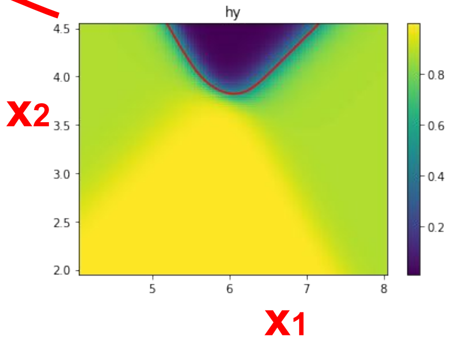






$$w_{21} = 7,\ w_{22} = 7,\ b_2 = -5$$





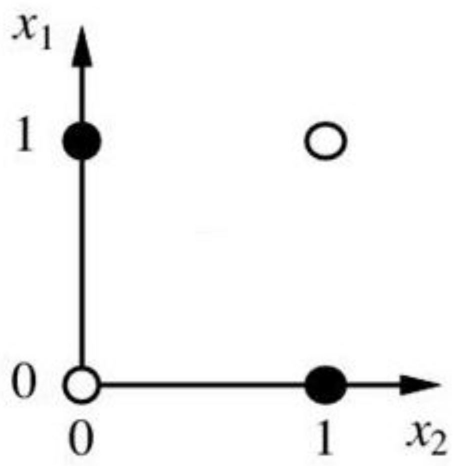
$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

Approximation of binary logical functions

Is a single neuron capable of producing arbitrary logical functions?

The “XOR” problem

The exclusive-OR (XOR) function

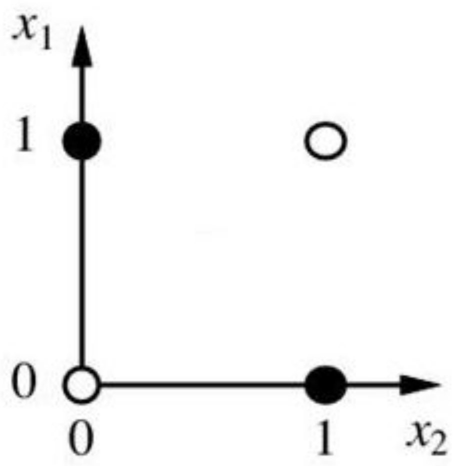


x_1 **xor** x_2

???

The “XOR” problem

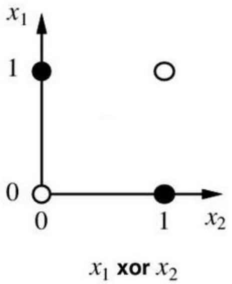
The exclusive-OR (XOR) function



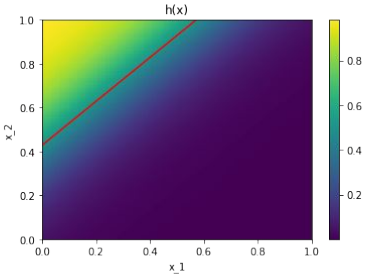
x_1 **xor** x_2

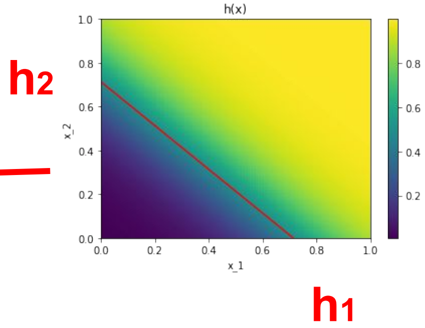
The logical function "exclusive OR"
is not linearly separable,
so it cannot be approximated well
by a single neuron.

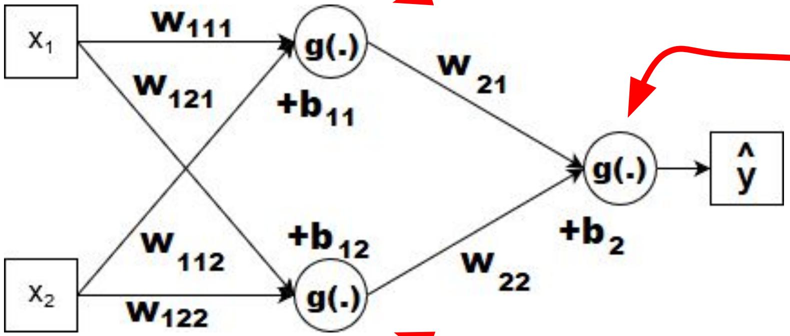
The “XOR” problem



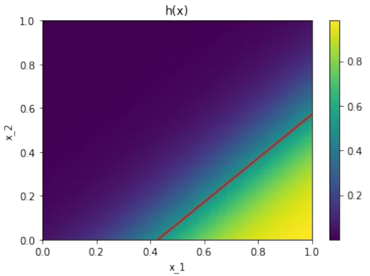
$$w_{111} = -7,\ w_{121} = 7,\ b_{11} = -3$$

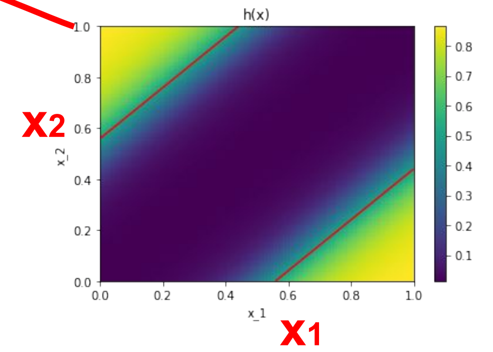






$$w_{21} = 7,\ w_{22} = 7,\ b_2 = -5$$





$$w_{112} = 7,\ w_{122} = -7,\ b_{12} = -3$$

Approximation of binary logical functions

Is a single neuron capable of producing arbitrary logical functions?

No!

A single neuron can only solve linearly separable problems.

Approximation of binary logical functions

Is a single neuron capable of producing arbitrary logical functions?

No!

A single neuron can only solve linearly separable problems.

The "XOR" problem: The expressive power of a single artificial neuron is severely limited, which justifies the use of multilayer neural networks...

Can be proven: Even a two-layer neural network (with sigmoids) can approximate any function to any degree with the appropriate weights, if we have enough neurons available.

Approximation of binary logical functions

Is a single neuron capable of producing arbitrary logical functions?

No!

A single neuron can only solve linearly separable problems.

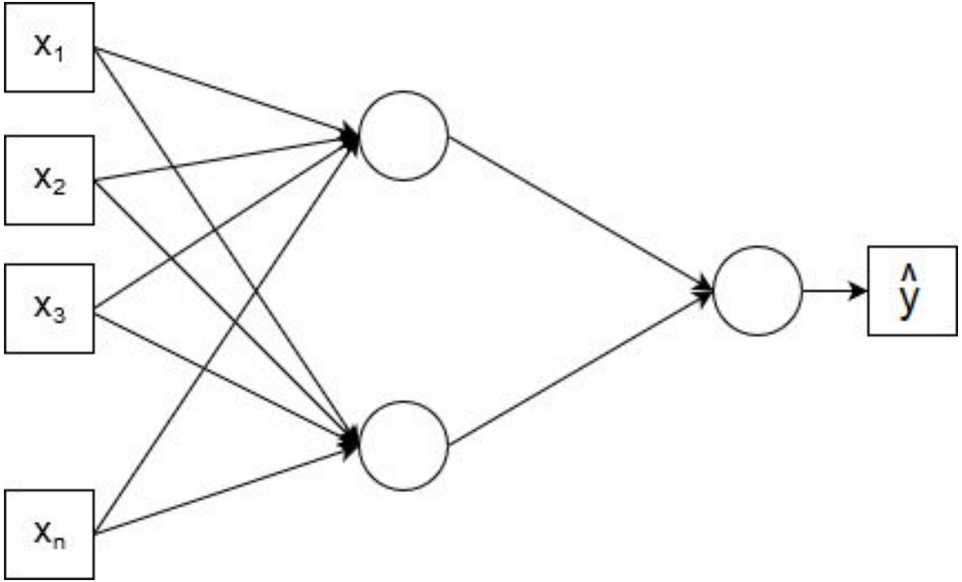
Not a very useful result:
Finding these weights faster than exponential time
(brute force) is not guaranteed!

The "XOR" problem: The expressive power of a single artificial neuron is severely limited, which justifies the use of multilayer neural networks...

Can be proven: Even a two-layer neural network (with sigmoids)
can approximate any function to any degree with the appropriate weights,
if we have enough neurons available.

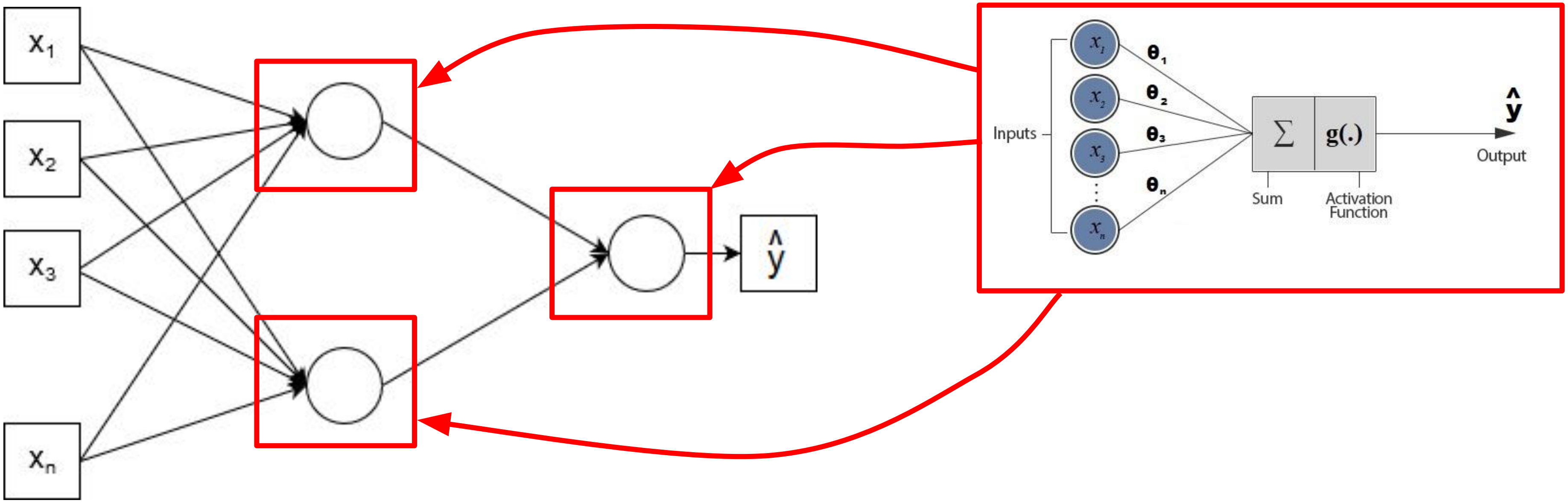
Multilayer Perceptron (MLP)

Artificial neurons are the building blocks of one of the basic types of artificial neural networks (the Multilayer Perceptron, MLP).

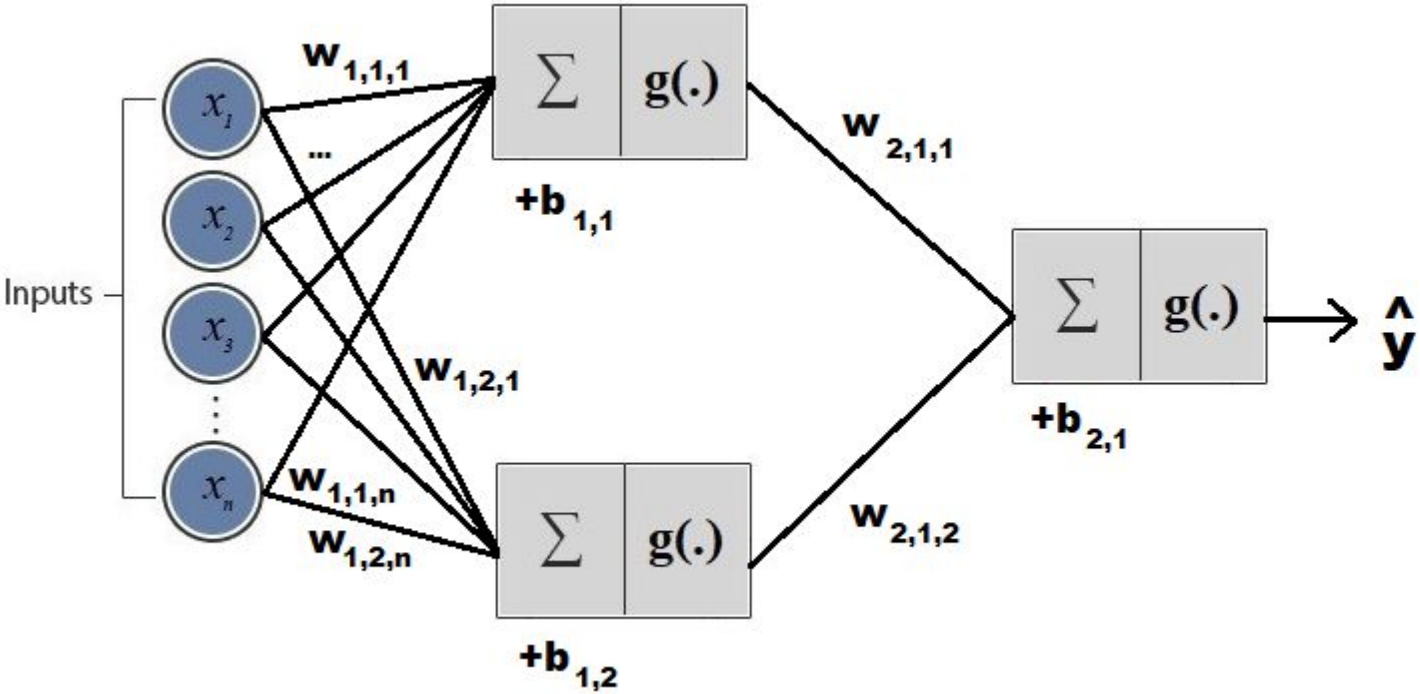


Multilayer Perceptron (MLP)

Artificial neurons are the building blocks of one of the basic types of artificial neural networks (the Multilayer Perceptron, MLP).



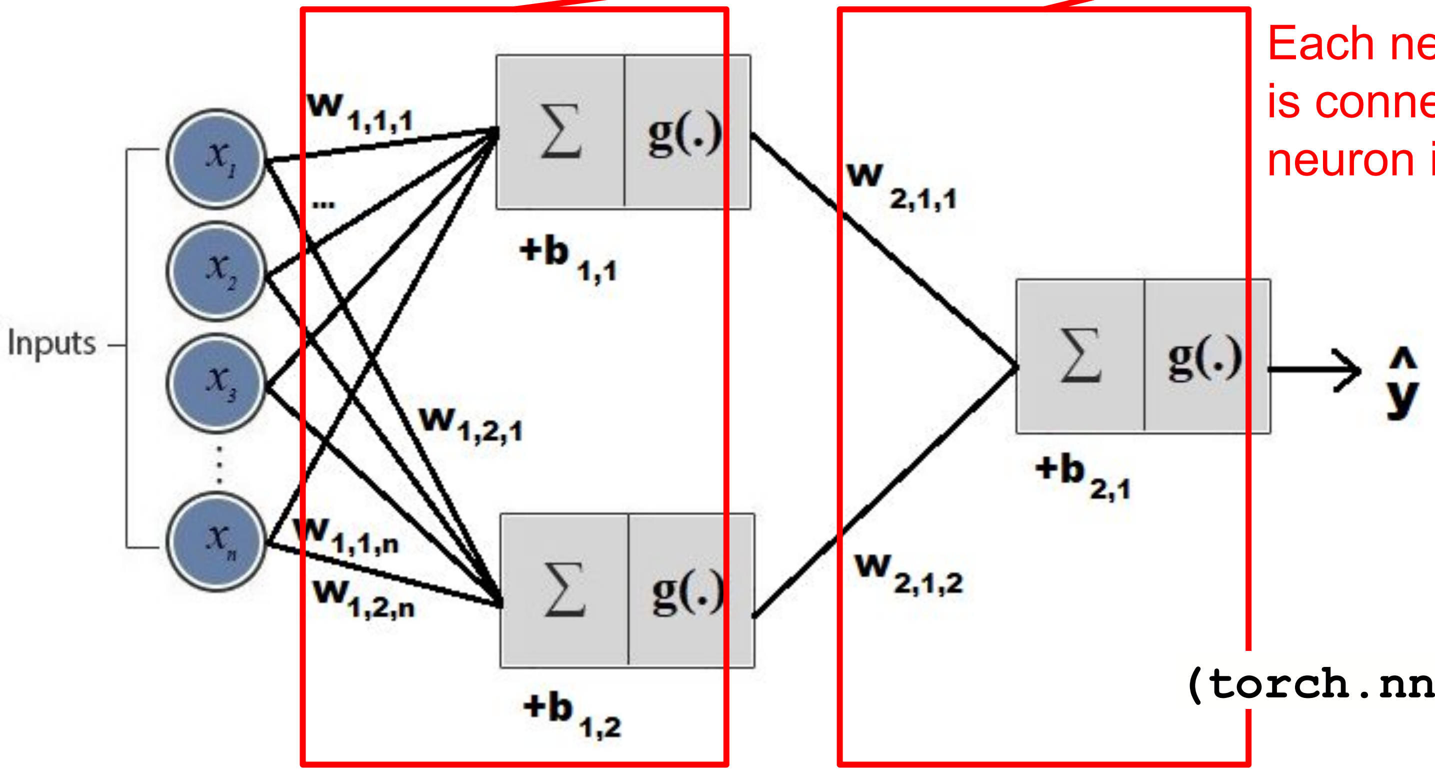
Multilayer Perceptron (MLP)



Multilayer Perceptron (MLP)

Two **fully connected** (or dense) **layers:**

Each neuron in the layer is connected to every neuron in the next layer.

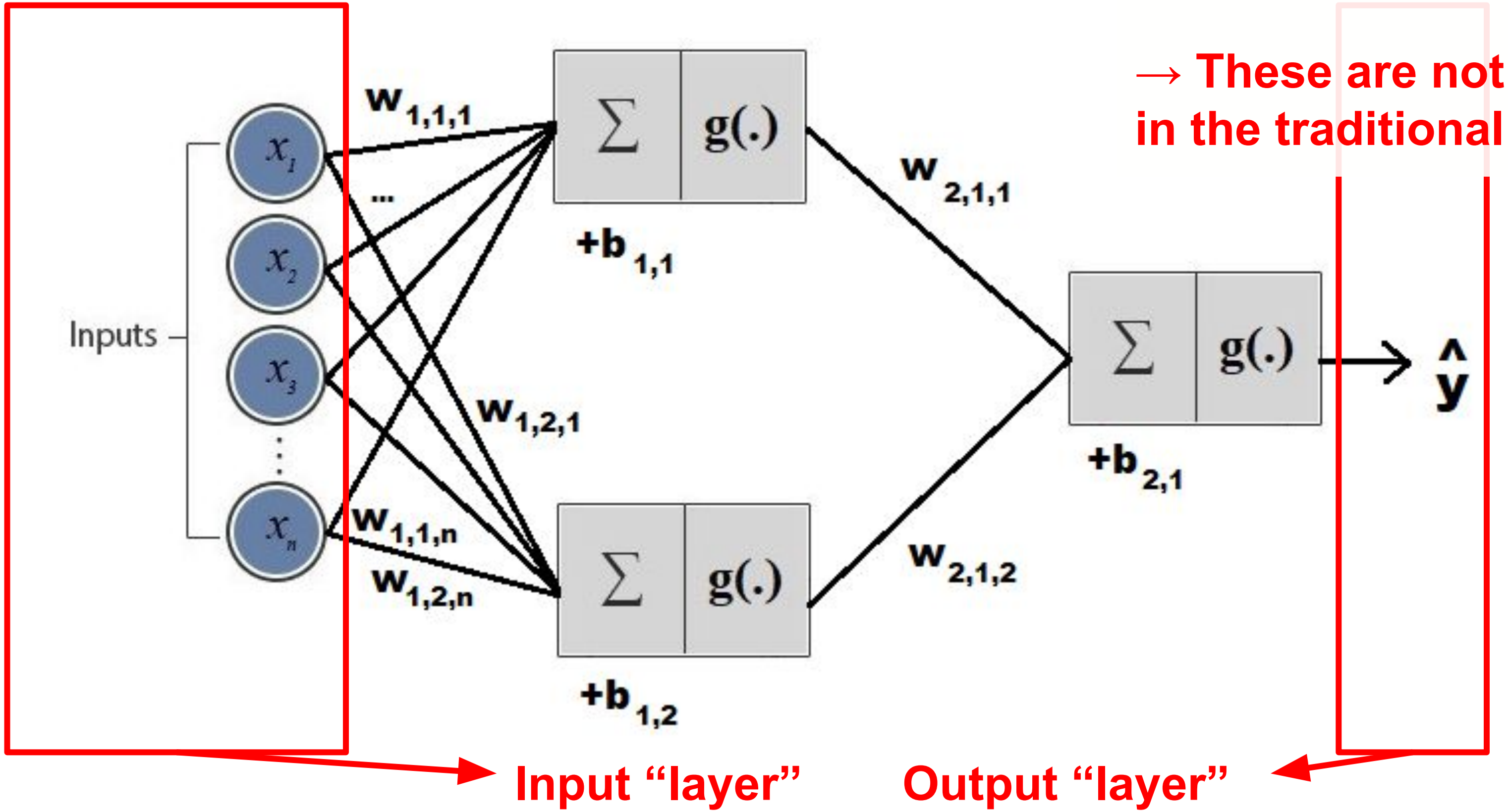


`(torch.nn.Linear)`

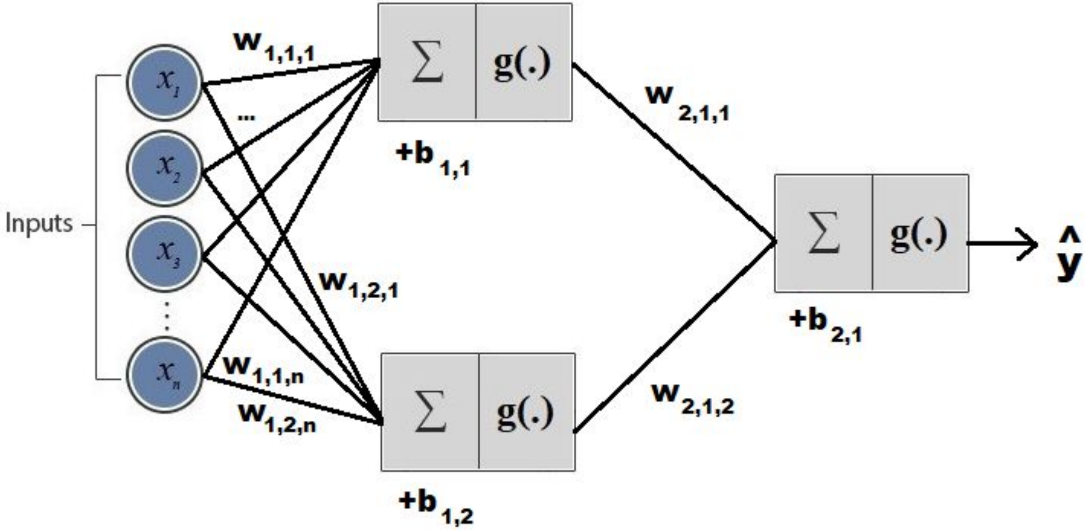
Multilayer Perceptron (MLP)

There are no neurons or weights (parameters) **in the input and output "layers."**

→ These are not neuron layers in the traditional sense.



Multilayer Perceptron (MLP)



$$W_1 \in \mathbb{R}^{2 \times n}$$
$$b_1 \in \mathbb{R}^2$$

$$W_2 \in \mathbb{R}^{1 \times 2}$$
$$b_2 \in \mathbb{R}^1$$

New notation: Theta is the set of all weight matrices and bias vectors.
(i.e., the parameters)

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

Multilayer Perceptron (MLP)

The size of **weight matrices** and **bias vectors**, generally:

$$\boldsymbol{W_k} \in \mathbb{R}^{S_k \times S_{k-1}}$$

$$\boldsymbol{b_k} \in \mathbb{R}^{S_k}$$

where $\boldsymbol{S_k}$ is the number of neurons in layer #k.

$$S_0 := n$$

$$x \in \mathbb{R}^n$$

Multilayer Perceptron (MLP)

The size of **weight matrices** and **bias vectors**, generally:

$$W_k \in \mathbb{R}^{S_k \times S_{k-1}}$$

$$b_k \in \mathbb{R}^{S_k}$$

where S_k is the number of neurons in layer #k.

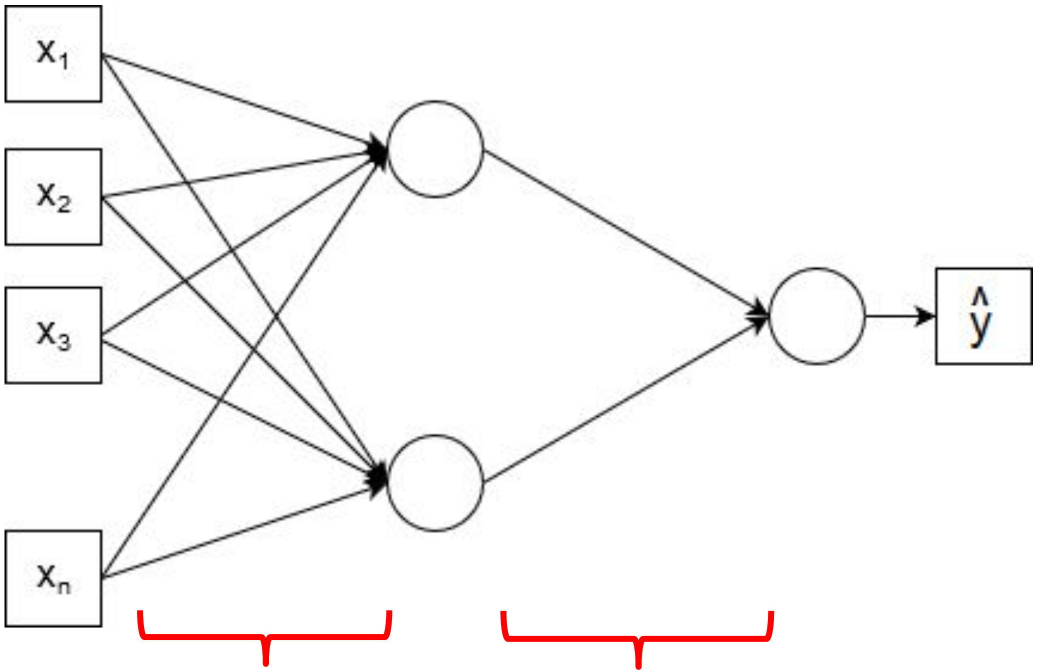
$$S_0 := n$$

$$x \in \mathbb{R}^n$$

Notation: In matrix form, the parameters are typically denoted by **W** (weight matrix) and **b** (bias vector). These correspond to the **θ** parameters used in linear and logistic regression (**b** replaces the constant term, $\boldsymbol{\theta_0}$ parameter).

$\mathbf{S_0}$ is the size of the input "layer", i.e the number of input variables.

Multilayer Perceptron (MLP)



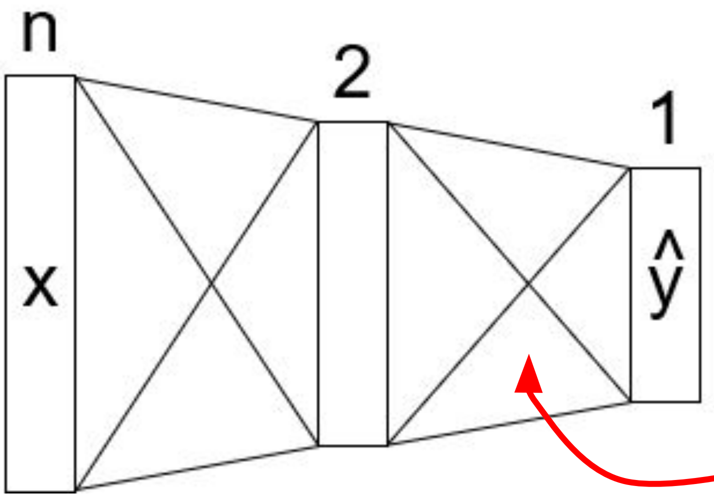
A simplified visual representation of an MLP…

$$W_1 \in \mathbb{R}^{2\times n}$$
$$b_1 \in \mathbb{R}^{2}$$

$$W_2 \in \mathbb{R}^{1\times 2}$$
$$b_2 \in \mathbb{R}^{1}$$

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

Multilayer Perceptron (MLP)



An even more simplified visual representation of an MLP…

A typical way to represent a fully connected layer in a neural network architecture diagram.

$W_1 \in \mathbb{R}^{2\times n}$

$b_1 \in \mathbb{R}^2$

$W_2 \in \mathbb{R}^{1\times 2}$

$b_2 \in \mathbb{R}^1$

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2 \underbrace{g_1(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2) = \hat{y} \approx y$$

Loss functions remain the same, for now:

- **Classification:** Logistic loss (BCE)
- **Regression:** MSE

Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2 \underbrace{g_1(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2) = \hat{y} \approx y$$

Activation functions:

- **Classification:** Sigmoid (same as in case of logistic regression)
- **Regression:** ???

Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2 \underbrace{g_1(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2) = \hat{y} \approx y$$

Activation functions:

- **Classification:** Sigmoid (same as in case of logistic regression)
- **Regression:** ??? **We did not use an activation function** (nonlinearity) **in linear regression...** Following this, in the case of regression, we should not put an activation function in our neurons...

Multilayer Perceptron (MLP)

Is the following hypothesis function suitable for regression?

$$h(x) = \underbrace{W_2 \left(W_1 x + b_1\right)}_{} + b_2 = \hat{y} \approx y$$

The output of the first layer

Multilayer Perceptron (MLP)

Is the following hypothesis function suitable for regression?

$$h(x) = W_2\,\underbrace{(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2 = \hat{y} \approx y$$

It doesn't make much sense, as **its expressive power corresponds to a single linear layer:**

$$W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (W_2 b_1 + b_2)$$

Multilayer Perceptron (MLP)

Is the following hypothesis function suitable for regression?

$$h(x) = \underbrace{W_2\,(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2 = \hat{y} \approx y$$

The output of the first layer

It doesn't make much sense, as **its expressive power corresponds to a single linear layer:**

$$W_2(W_1 x + b_1) + b_2 = \overbrace{(W_2 W_1)}^{\in \mathbb{R}^{S_2 \times S_0}} x + (W_2 b_1 + b_2)$$

Composition of multiple linear functions is still linear → without nonlinearity, the expressive power of the neural network is identical to that of linear regression...

Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2\, \underbrace{g_1(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2) = \hat{y} \approx y$$

The output of the first layer

In the case of regression, we will also use this hypothesis function.

However, it is worth omitting g_2 (the last activation function)**.**

Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2 \underbrace{g_1(W_1 x + b_1)}_{\text{The output of the first layer}} + b_2) = \hat{y} \approx y$$

In the case of regression, we will also use this hypothesis function.

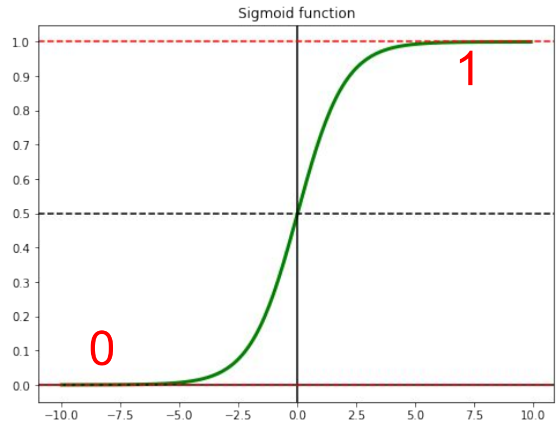
However, it is worth omitting g_2 (the last activation function).

After all, if **g_2** is sigmoid, the output of the network will be between 0 and 1, which is unsuitable for estimating age, for example. **g_2** is therefore typically an identity function in the case of regression.

Activation functions

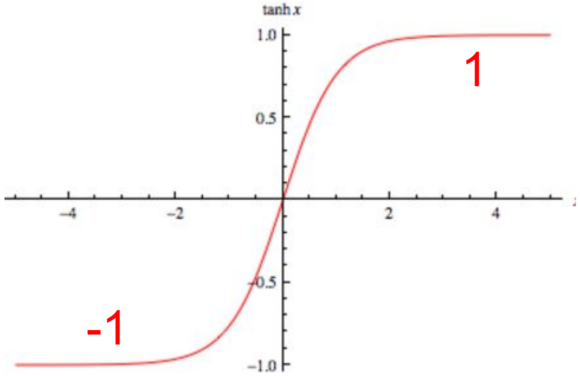
Popular activation functions

sigmoid



$$g(z) = \frac{1}{1+e^{-z}}$$

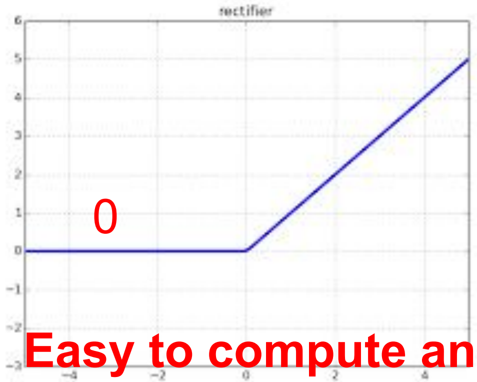
tanh



$$g(z) = \tanh(z) = \frac{e^{2x}-1}{e^{2x}+1}$$

ReLU

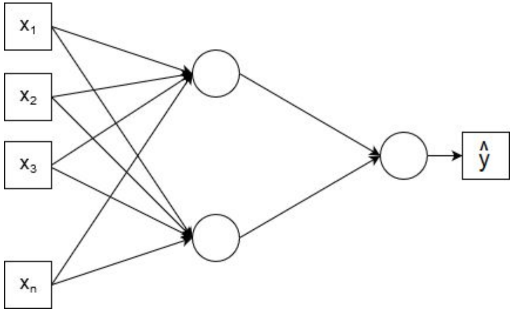
(Rectified Linear Unit)



Easy to compute and almost always works well

$$g(z) = \mathrm{ReLU}(z) = \max(0, z)$$

Multilayer Perceptron (MLP)



The hypothesis function of a two-layer MLP:

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, 1)
        )
  def forward(self, x):
      return self.layers(x)
```

PyTorch code when
g_1 = ReLU
g_2 = identity

Training an MLP

We will use gradient descent…

$$
\begin{aligned}
&\text{repeat until convergence} \quad \{ \\
&\quad \text{for } \forall\, \theta \in \boldsymbol{\Theta} \quad \{ \\
&\qquad grad_\theta = \frac{\partial}{\partial \theta} J(\boldsymbol{\Theta}) \\
&\quad \} \\
&\quad \text{for } \forall\, \theta \in \boldsymbol{\Theta} \quad \{ \\
&\qquad \theta = \theta - \alpha\, grad_\theta \\
&\quad \} \\
&\}
\end{aligned}
$$

Training an MLP

We will use gradient descent…

Θ: the set of all parameters
(contains the elements of weight matrices, and bias vectors)

$$
\begin{aligned}
&\text{repeat until convergence} \quad \{ \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\qquad grad_\theta = \frac{\partial}{\partial \theta} J(\Theta) \\
&\quad \} \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\qquad \theta = \theta - \alpha\, grad_\theta \\
&\quad \} \\
&\}
\end{aligned}
$$

We compute the gradient of the loss function with respect to each parameter.

Fortunately, we don't have to calculate the gradients by hand.
PyTorch's **automatic derivation** algorithm does this for us...

→ **Next lecture**

Training an MLP

“Batch” gradient descent

We average the loss over data points in the entire training set.

$$
\begin{aligned}
&\text{repeat until convergence} \quad \{ \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\quad\quad grad_\theta = \frac{\partial}{\partial \theta} J(\Theta) \\
&\quad \} \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\quad\quad \theta = \theta - \alpha\; grad_\theta \\
&\quad \} \\
&\}
\end{aligned}
$$

Training an MLP

"Batch" gradient descent

We average the loss over data points in the entire training set.

For example:

$$J(\theta) = \frac{1}{2m}\sum_{j=1}^{m}(h_\theta(x^{(j)}) - y^{(j)})^2$$

$$
\begin{aligned}
&\text{repeat until convergence} \quad \{ \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\qquad grad_\theta = \frac{\partial}{\partial \theta} J(\Theta) \\
&\quad \} \\
&\quad \text{for } \forall\, \theta \in \Theta \quad \{ \\
&\qquad \theta = \theta - \alpha\, grad_\theta \\
&\quad \} \\
&\}
\end{aligned}
$$

Training an MLP

“Batch” gradient descent

We average the loss over data points in the entire training set.

Problem: To compute a single step, we calculate the gradient over the entire dataset.

→ **Enormous computational cost**

$$
\begin{aligned}
&\text{repeat until convergence} \quad \{ \\
&\quad \text{for} \; \forall\, \theta \in \Theta \quad \{ \\
&\qquad grad_\theta = \frac{\partial}{\partial \theta} J(\Theta) \\
&\quad \} \\
&\quad \text{for} \; \forall\, \theta \in \Theta \quad \{ \\
&\qquad \theta = \theta - \alpha \, grad_\theta \\
&\quad \} \\
&\}
\end{aligned}
$$

Training an MLP - The SGD algorithm

The **Stochastic Gradient Descent** (SGD) algorithm

Let's randomly select a few data points from the training set and use only those to compute the next step!

Training an MLP - The SGD algorithm

The **Stochastic Gradient Descent** (SGD) algorithm

- **The smaller the size** of the selected mini-batch, the more likely it is that we will move in the wrong direction with the parameter update (variance increases).
- **The larger the size** of the selected mini-batch, the greater the computing and memory requirements.

`(torch.optim.SGD)`

Training an MLP - The SGD algorithm

The **Stochastic Gradient Descent** (SGD) algorithm

- **The smaller the size** of the selected mini-batch, the more likely it is that we will move in the wrong direction with the parameter update (variance increases).
- **The larger the size** of the selected mini-batch, the greater the computing and memory requirements.

For larger neural networks, efficiency considerations often determine the mini-batch size: choose as many examples at a time as will fit in the GPU memory!

If we can only use a small mini-batch size, we need to reduce the learning rate to achieve convergence.

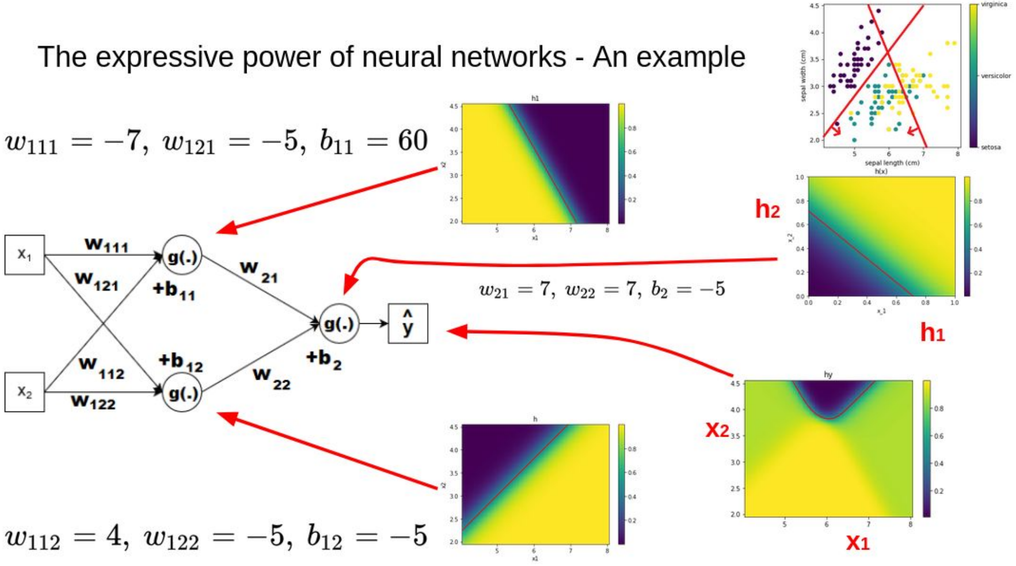
Training an MLP - The SGD algorithm

Variants of the **Stochastic Gradient Descent** (SGD) algorithm:

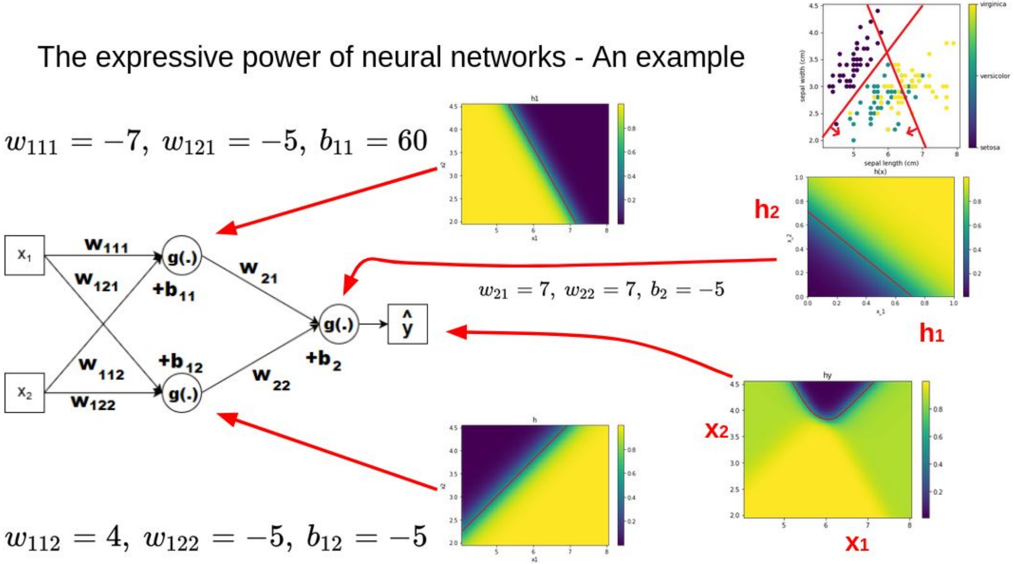
- AdaGrad
- Adam
- Adamax
- RMSprop
- …

`(torch.optim.*)`

The expressive power of neural networks - An example



The expressive power of neural networks - An example



We can see that the neural network is capable of representing the above decision boundaries, but **we set the weights** required for this **manually**!

Can we find (learn) **a good solution using gradient descent?**

The expressive power of neural networks - An example

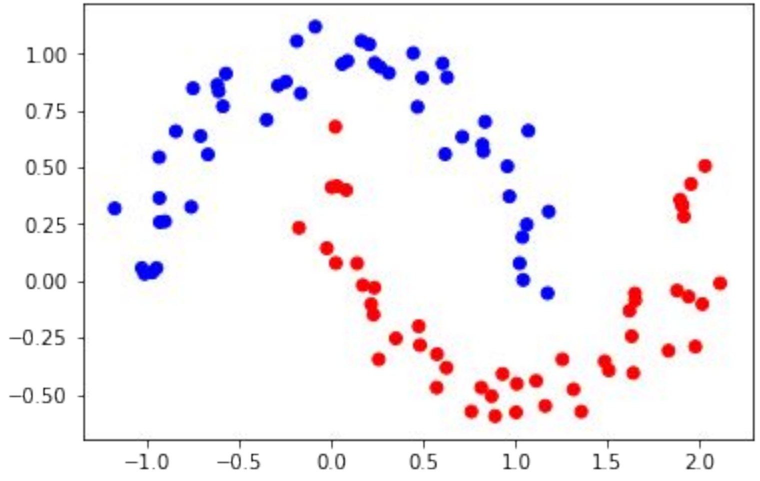
In the previous example we set the weights manually.

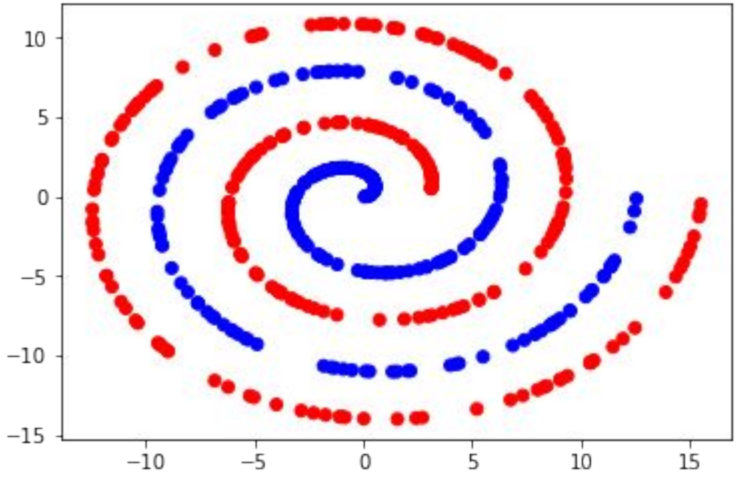
Now let's see what can a neural network **learn** by itself!

Notebook on Google Colab:
https://colab.research.google.com/drive/1jrGJFbZY5TFk-NN2epcAPvBoxyKyWI1H

The expressive power of neural networks - An example

Two more complex classification tasks

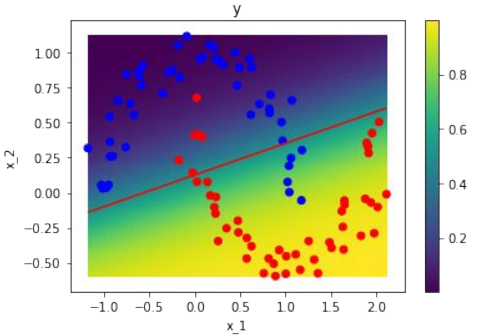


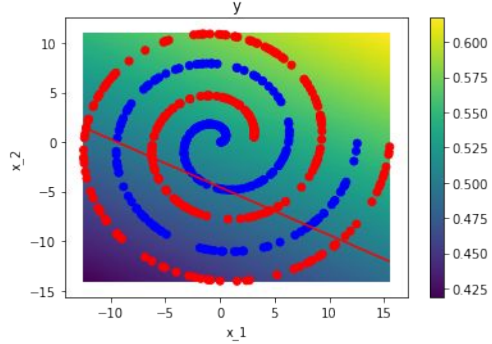


Binary classification: Let's learn a decision boundary that separates data points from the two categories!

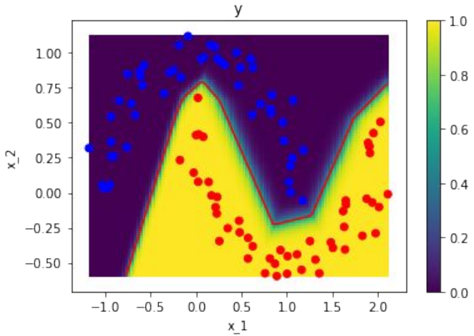
The expressive power of neural networks - An example

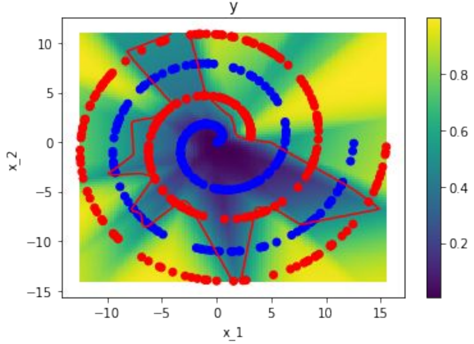
Logistic regression



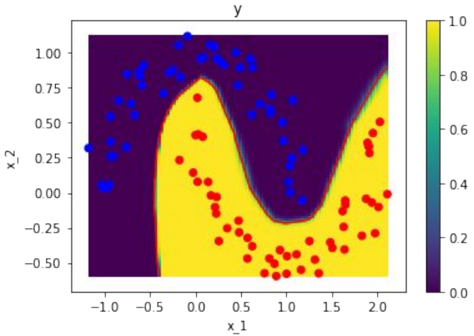


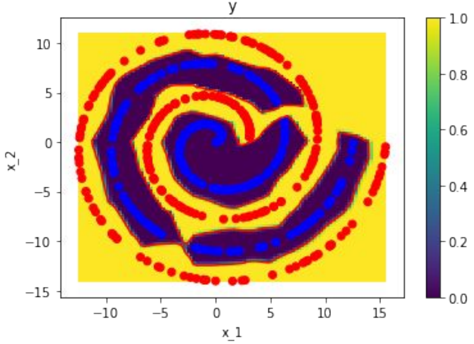
MLP, 2 layers, 20+1 neurons





MLP, 4 layers, 20+20+20+1 neurons

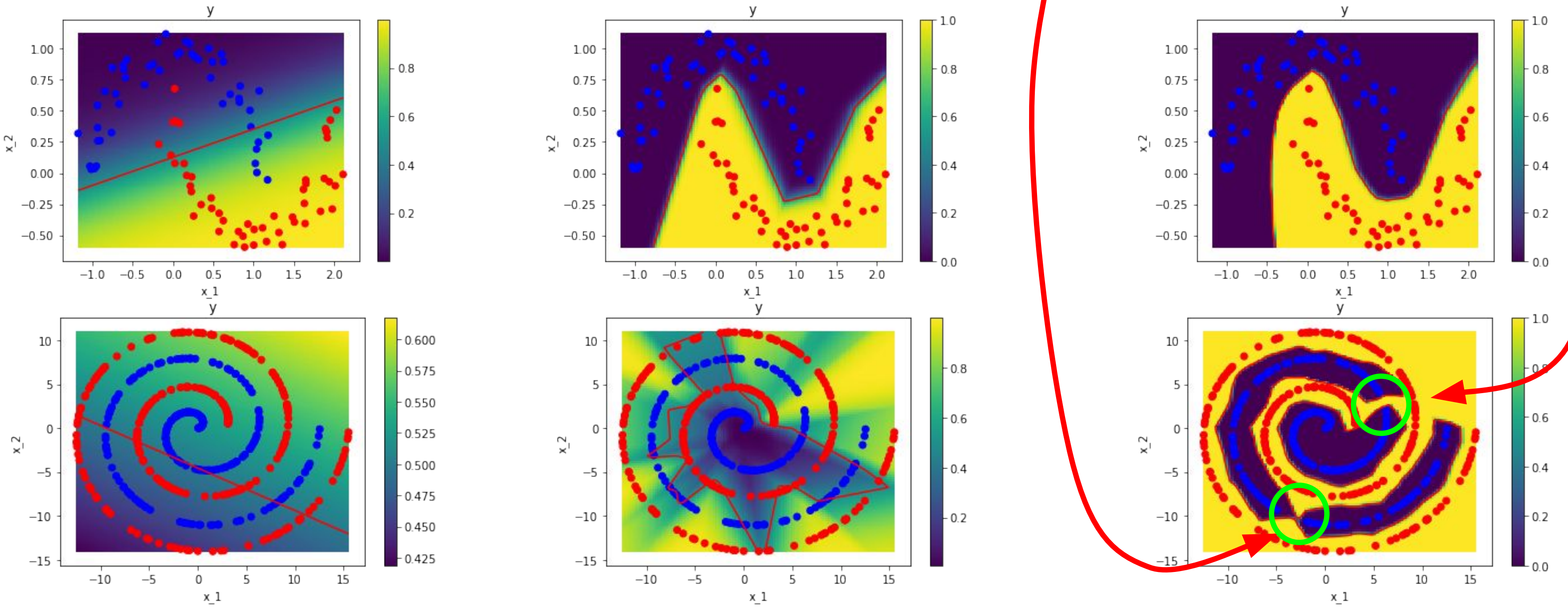




The expressive power of neural networks – An example

Túltanulás (overfitting)

Logistic regression	**MLP**, 2 layers, 20+1 neurons	**MLP**, 4 layers, 20+20+20+1 neurons



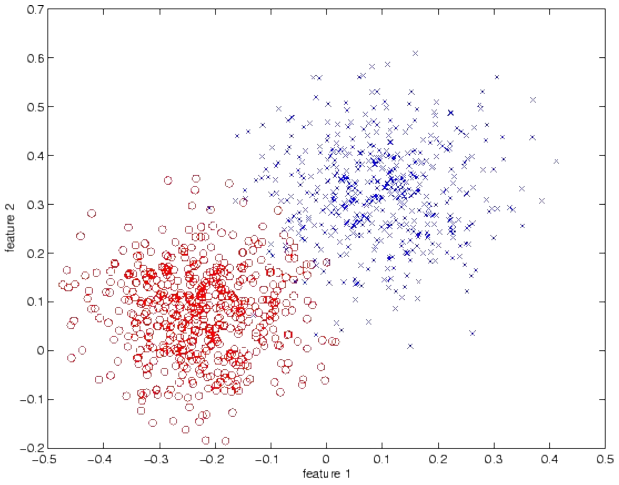
The expressive power of neural networks - An example

Interactive neural network simulator: https://playground.tensorflow.org/

The simulator can be used to examine the effect of overfitting in neural networks with different architectures.

Suggested settings:
Data: Gaussian, (optional: L2 reg. with > 0 rate)
Noise: >25

Classification: We learn a decision boundary that separates data points from two categories.



Regression - Examples

Example task until now - a single label variable:

x_1: Weight of a patient
x_2: Age of a patient
x_3: Sex of a patient

→ **y:** Cholesterol levels of the patient

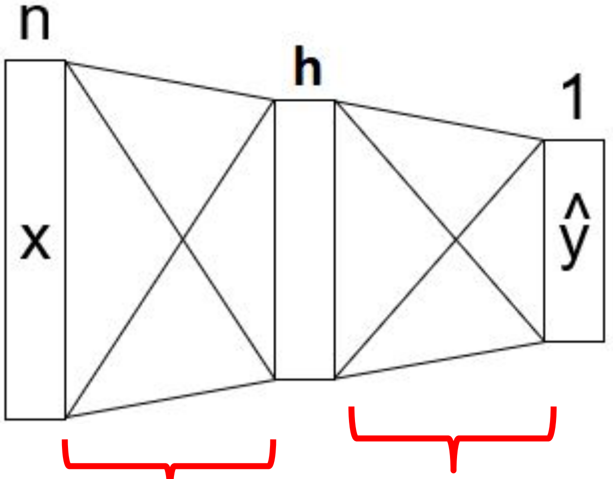
Example task from now - possibly multiple label variables:

x_1: Weight of a patient
x_2: Age of a patient
x_3: Sex of a patient

→ **y_1:** Cholesterol levels of the patient
y_2: Blood sugar levels of the patient

MLP - Multiple label variables

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$



$W_1 \in \mathbb{R}^{h \times n}$

$b_1 \in \mathbb{R}^h$

$W_2 \in \mathbb{R}^{1 \times h}$

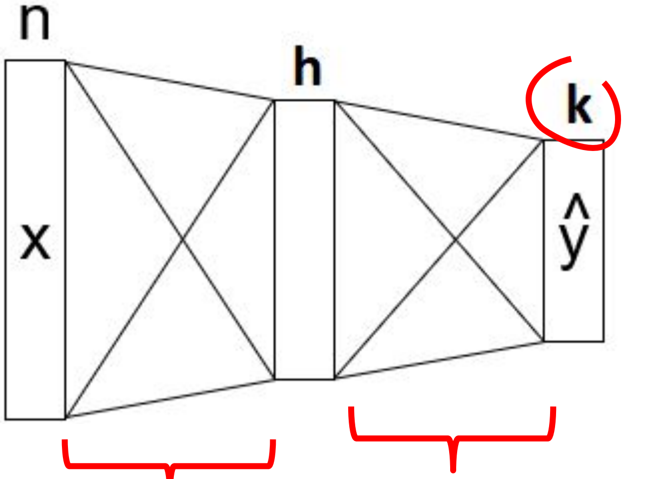
$b_2 \in \mathbb{R}$

Until now, y was always a scalar.
In regression, this limits us to estimating a value, and in classification, it limits us to estimating a single probability (2 categories)...

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

MLP - Multiple label variables

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$



$W_1 \in \mathbb{R}^{h \times n}$ $\quad$ $W_2 \in \mathbb{R}^{k \times h}$

$b_1 \in \mathbb{R}^{h}$ $\quad$ $b_2 \in \mathbb{R}^{k}$

Let y be a vector, similarly to x!

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

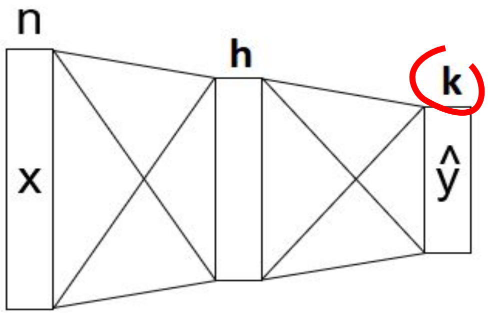
MLP - Multiple label variables

Regression

y hat and **y** are vectors

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Since in regression our labels can contain arbitrary numbers, g_2 is typically an identity function (i.e., it can be omitted).



MLP - Multiple label variables

Regression

y hat and **y** are vectors

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Loss: We average the squared loss over the elements of the label vector.

Until now (y was a scalar): $J(\Theta) = \frac{1}{2m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)})^2$

y is a vector: $J(\Theta) = \frac{1}{2mk}\sum_{j=1}^{m}||\hat{y}^{(j)} - y^{(j)}||_2^2 = \frac{1}{2mk}\sum_{j=1}^{m}\sum_{i=1}^{k}(\hat{y}_i^{(j)} - y_i^{(j)})^2$

MLP - Multiple label variables

Regression

y hat and **y** are vectors

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Loss: We average the squared loss over the elements of the label vector.

Until now (y was a scalar): $$J(\Theta) = \frac{1}{2m}\sum_{j=1}^{m}(\hat{y}^{(j)} - y^{(j)})^2$$

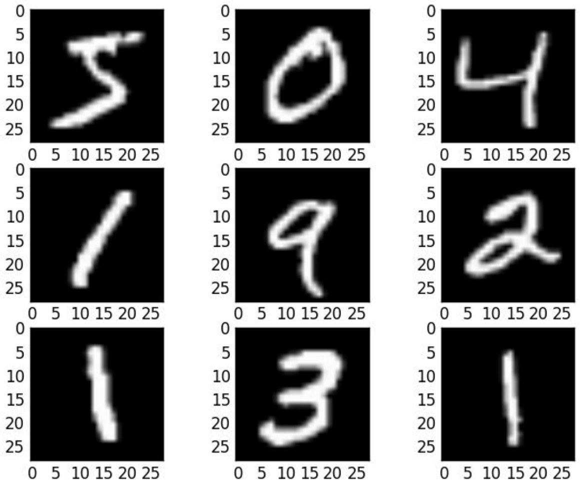
y is a vector: $$J(\Theta) = \frac{1}{2mk}\sum_{j=1}^{m}||\hat{y}^{(j)} - y^{(j)}||_2^2 = \frac{1}{2mk}\sum_{j=1}^{m}\sum_{i=1}^{k}(\hat{y}_i^{(j)} - y_i^{(j)})^2$$

MSE as before**, but now we also average over the elements of the label vector.**

Application of an MLP to classify handwritten digits

MNIST dataset

- Handwritten digits
- 28 × 28 pixel image size
- 10 categories (digits: 0 .. 9)
- 60,000 training examples
- 10,000 test examples

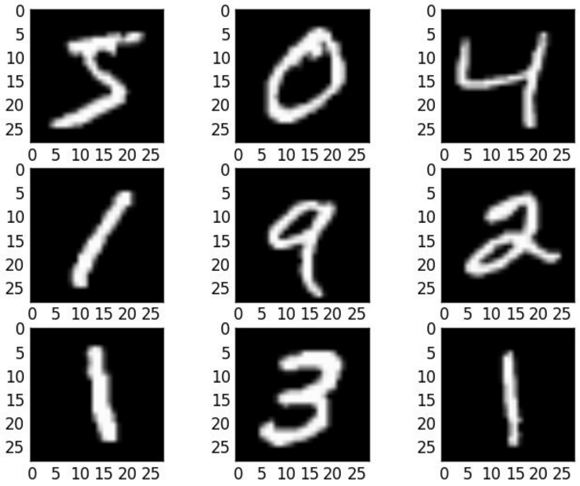


Application of an MLP to classify handwritten digits

MNIST dataset

- Handwritten digits
- 28 × 28 pixel image size
- 10 categories (digits: 0 .. 9)
- 60,000 training examples
- 10,000 test examples

784 input variables: The brightness of each pixel is an input variable.

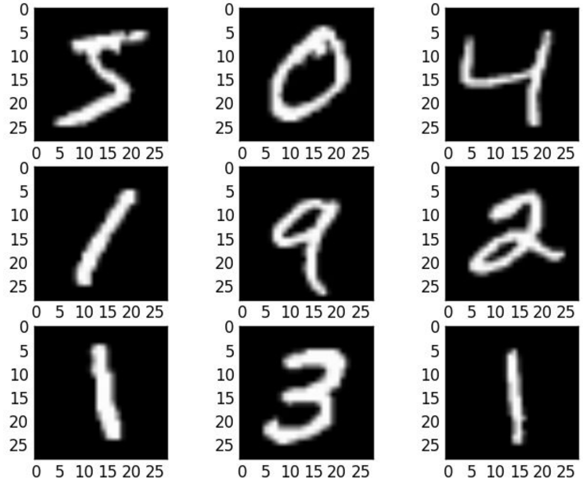


Application of an MLP to classify handwritten digits

MNIST dataset

- Handwritten digits
- 28 × 28 pixel image size
- 10 categories (digits: 0 .. 9)
- 60,000 training examples
- 10,000 test examples

784 input variables: The brightness of each pixel is an input variable.



How do we classify them into 10 categories?

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Until now (y was a scalar): Sigmoid was estimating a value between 0 and 1, which we interpreted as a probability → **Suitable for 2 categories**

How to classify into more than 2 categories?

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Until now (y was a scalar): Sigmoid was estimating a value between 0 and 1, which we interpreted as a probability → **Suitable for 2 categories**

How to classify into more than 2 categories?

Let’s estimate a probability for each category!

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Until now (y was a scalar): Sigmoid was estimating a value between 0 and 1, which we interpreted as a probability → **Suitable for 2 categories**

How to classify into more than 2 categories?

Let's estimate a probability for each category!

The sum of the estimated probabilities should be 1,
since each example belongs to exactly one category.

MLP - Multiple label variables

The **Softmax** activation function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

$$\boldsymbol{\sigma} : \mathbb{R}^{\boldsymbol{k}} \to \mathbb{R}^{\boldsymbol{k}}$$

$$\sigma\left(\begin{bmatrix} 2.6 \\ 1.5 \\ 0.2 \\ 0.6 \end{bmatrix}\right) = \begin{bmatrix} 0.64 \\ 0.21 \\ 0.06 \\ 0.09 \end{bmatrix}$$

MLP - Multiple label variables

The **Softmax** activation function

$$e^{z_i}$$

The i-th element of the input vector is raised to the exponential.

$$\sum_{j=1}^{k} e^{z_j}$$

The sum of vector elements raised to the exponential.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

$$\sigma : \mathbb{R}^k \rightarrow \mathbb{R}^k$$

$$\sigma\left(\begin{bmatrix} 2.6 \\ 1.5 \\ 0.2 \\ 0.6 \end{bmatrix}\right) = \begin{bmatrix} 0.64 \\ 0.21 \\ 0.06 \\ 0.09 \end{bmatrix}$$

The sum of the elements of the result vector is 1,
so it can be interpreted as the mass function of a probability distribution.

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Until now (y was a scalar): Sigmoid was estimating a value between 0 and 1, which we interpreted as a probability → **Suitable for 2 categories**

From now y can be a vector: In this case, g_2 will be the softmax function.

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Loss: Cross-entropy (CE), the generalization of Binary CE / Log. loss

Until now (y was a scalar): $J(\theta) = \frac{1}{m}\sum_{j=1}^{m}[-y^{(j)}\log(\hat{y}^{(j)}) - (1 - y^{(j)})\, log(1 - \hat{y}^{(j)})]$

From now y can be a vector: $J(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_i log(\hat{y}_i)$

MLP - Multiple label variables

Classification

y hat and **y** are vectors

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Loss: Cross-entropy (CE), the generalization of Binary CE / Log. loss

Until now (y was a scalar): $$J(\theta) = \frac{1}{m}\sum_{j=1}^{m}[-y^{(j)}\log(\hat{y}^{(j)}) - (1 - y^{(j)})\,log(1 - \hat{y}^{(j)})]$$

Equals to the formula above in the binary case (k = 2)

From now y can be a vector: $$J(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_i log(\hat{y}_i)$$

How does the true label (y) **look like?**

MLP - Multiple label variables

Cross-entropy (CE) **loss:**

$$J(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_i log(\hat{y}_i)$$

True (target/ground truth) **label, one-hot encoded:**
The vector element representing the true category of the example is 1, the others are 0.

0
1
0
0

0.64
0.21
0.06
0.09

Estimated label:
Estimated probabilities of belonging to each category

MLP neural network model for regression - PyTorch

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, 1)
        )
  def forward(self, x):
      return self.layers(x)


loss_fn = nn.MSELoss()
```

y is a scalar
(regression, 1 label variable)

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, k)
        )
  def forward(self, x):
      return self.layers(x)


loss_fn = nn.MSELoss()
```

y is a vector
(regression, **k** label variables)

MLP neural network model for classification - PyTorch

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, 1),
            nn.Sigmoid()
        )
  def forward(self, x):
      return self.layers(x)

loss_fn = nn.BCELoss()
```

y is a scalar

(binary classification)

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, k)
        )
  def forward(self, x):
      return self.layers(x)

loss_fn = nn.CrossEntropyLoss()
```

y is a vector

(multi-class classification)

MLP neural network model for classification - PyTorch

CE loss in PyTorch includes the softmax activation function, so we don't need to include it here.
Because of this, **we need to pay attention when making predictions:**
if we need probabilities, we need to add a torch.nn.Softmax() to the network's estimation...

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, h_dim),
            nn.ReLU(),
            nn.Linear(h_dim, k)
        )
  def forward(self, x):
      return self.layers(x)


loss_fn = nn.CrossEntropyLoss()
```

y is a vector
(multi-class classification)

MLP neural network model for classification - PyTorch

CE loss in PyTorch automatically generates one-hot encoding, expecting the true (**y**) labels to be given in categorical form...

```
class MyTwoLayerMLP(nn.Module):
  def __init__(self, input_dim, h_dim):
        super().__init__()
        self.layers = nn.Sequential(
              nn.Linear(input_dim, h_dim),
              nn.ReLU(),
              nn.Linear(h_dim, k)
        )
  def forward(self, x):
        return self.layers(x)


loss_fn = nn.CrossEntropyLoss()
```

y is a vector
(multi-class classification)