Neural Networks

Based on slides by Gilles Louppe, Daniel Lowd

Threshold Logic Unit

The Threshold Logic Unit (McCulloch and Pitts, 1943) was the first mathematical model for a neuron. Assuming Boolean inputs and outputs, it is defined as:

$$f(\mathbf{x}) = 1_{\{\sum_i w_i x_i + b \geq 0\}}$$

This unit can implement:

- $\mathrm{or}(a, b) = 1_{\{a+b-0.5 \geq 0\}}$
- $\mathrm{and}(a, b) = 1_{\{a+b-1.5 \geq 0\}}$
- $\mathrm{not}(a) = 1_{\{-a+0.5 \geq 0\}}$

Perceptron

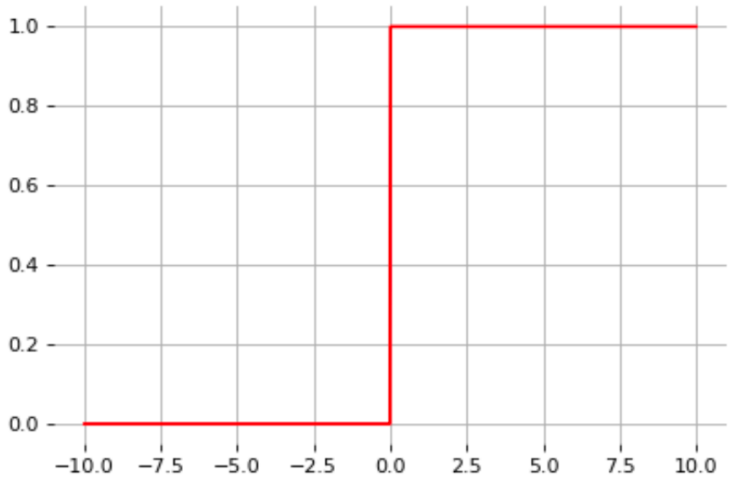
The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This model was originally motivated by biology, with w_i being synaptic weights and x_i and f firing rates.

Perceptron

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

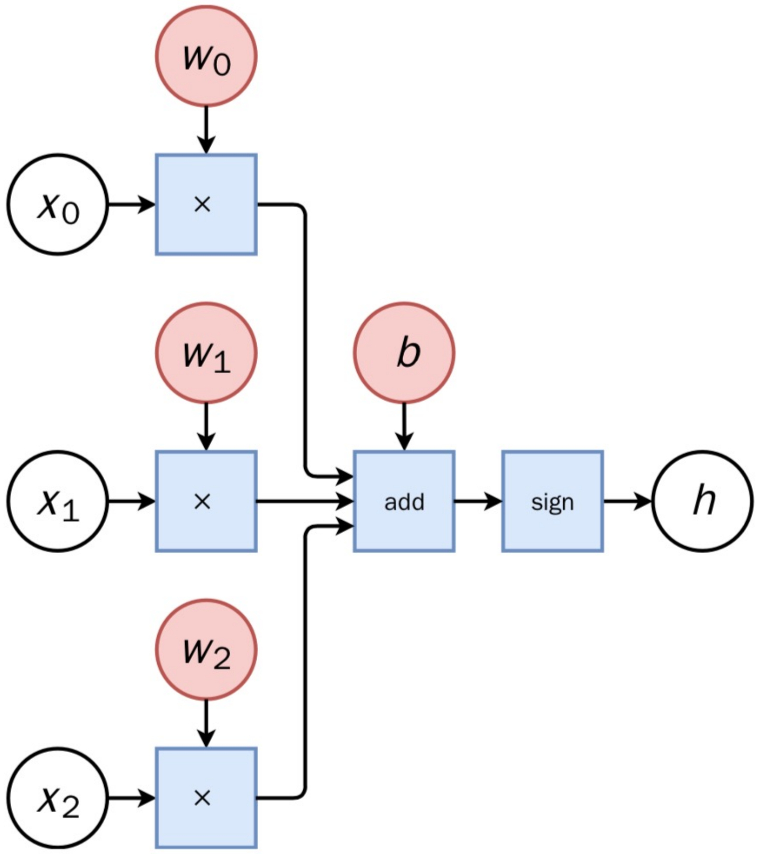


The perceptron classification rule can be rewritten as

$$f(\mathbf{x}) = \text{sign}(\sum_i w_i x_i + b).$$

Computational Graphs

Computational graphs



The computation of

$$f(\mathbf{x}) = \operatorname{sign}(\sum_i w_i x_i + b)$$

can be represented as a computational graph where

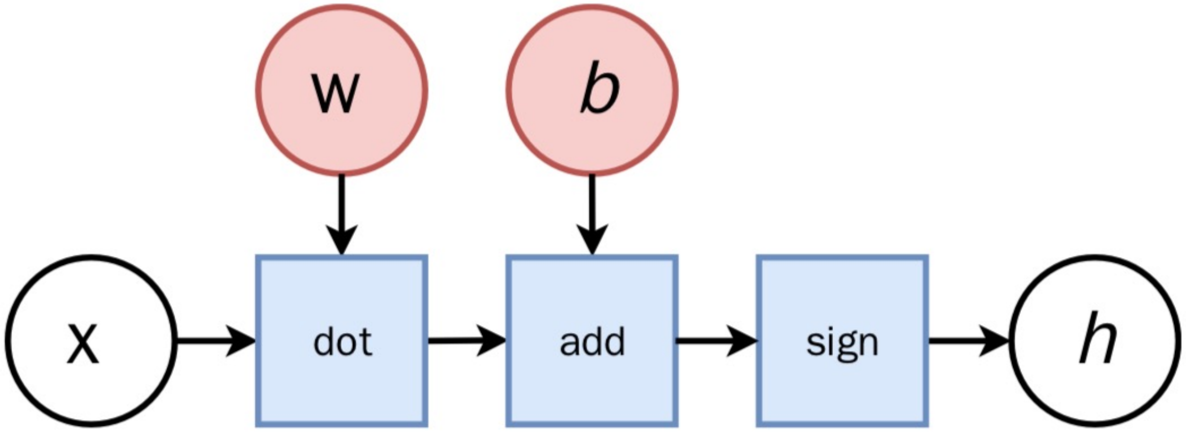
- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations.

Computational Graphs

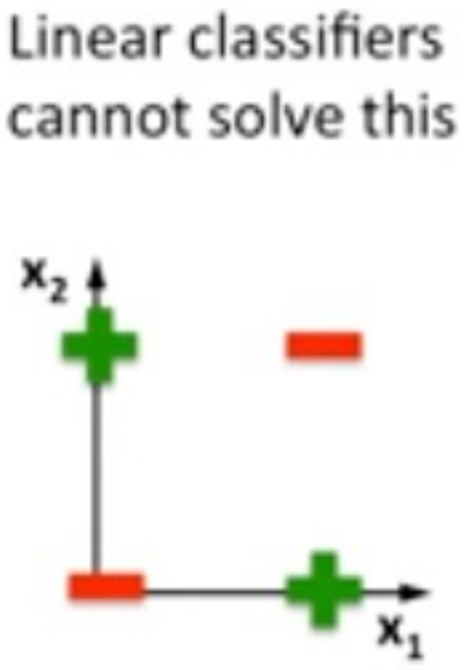
In terms of tensor operations, f can be rewritten as

$$f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^T\mathbf{x} + b),$$

for which the corresponding computational graph of f is:



How can we deal with non-linearly separable data?



Remember Logistic Regression?

Same model

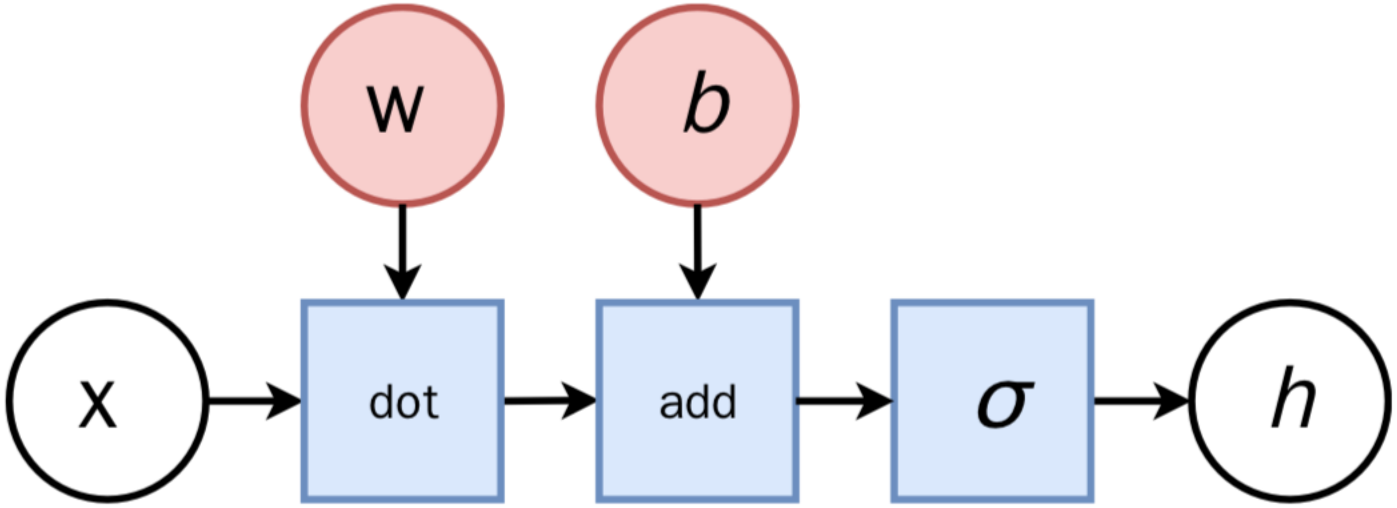
$$P(Y = 1|\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)$$

as for linear discriminant analysis.

But,

- ignore model assumptions (Gaussian class populations, homoscedasticity);
- instead, find $\mathbf{w}$, b that maximizes the likelihood of the data.

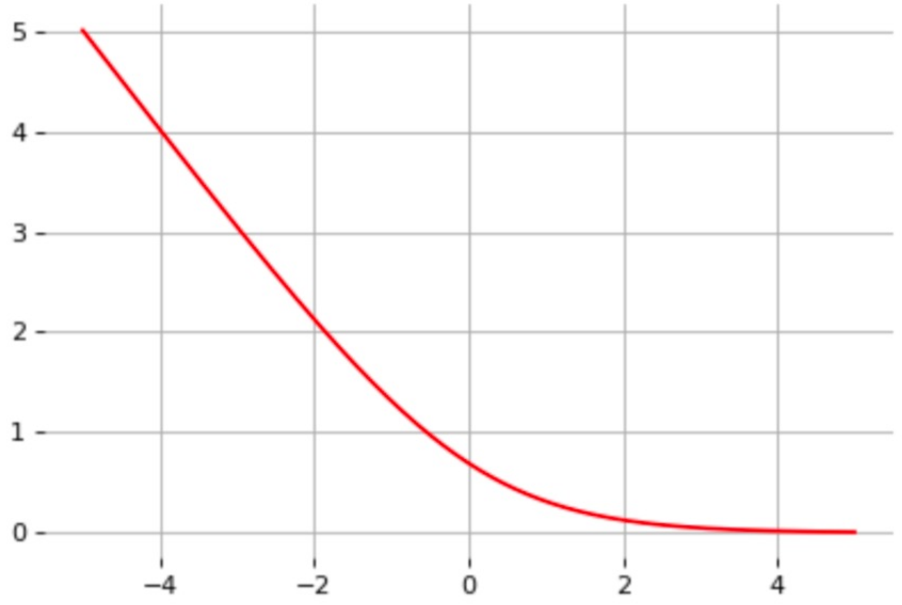
Computational Graphs



This unit is the lego brick of all neural networks!

The logit loss

$$\mathcal{L}(\mathbf{w}, b) = -\sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \log \sigma\left(y_i(\mathbf{w}^T\mathbf{x}_i + b)\right).$$



Cross Entropy

We have,

$$
\begin{aligned}
&\arg\max_{\mathbf{w},b} P(\mathbf{d}|\mathbf{w},b)\\
&= \arg\max_{\mathbf{w},b} \prod_{\mathbf{x}_i,y_i\in\mathbf{d}} P(Y=y_i|\mathbf{x}_i,\mathbf{w},b)\\
&= \arg\max_{\mathbf{w},b} \prod_{\mathbf{x}_i,y_i\in\mathbf{d}} \sigma(\mathbf{w}^T\mathbf{x}_i+b)^{y_i}(1-\sigma(\mathbf{w}^T\mathbf{x}_i+b))^{1-y_i}\\
&= \arg\min_{\mathbf{w},b} \underbrace{\sum_{\mathbf{x}_i,y_i\in\mathbf{d}} -y_i\log\sigma(\mathbf{w}^T\mathbf{x}_i+b)-(1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i+b))}_{\mathcal{L}(\mathbf{w},b)=\sum_i \ell(y_i,\hat{y}(\mathbf{x}_i;\mathbf{w},b))}
\end{aligned}
$$

This loss is an instance of the cross-entropy

$$H(p,q)=\mathbb{E}_p[-\log q]$$

for $p=Y|\mathbf{x}_i$ and $q=\hat{Y}|\mathbf{x}_i$.

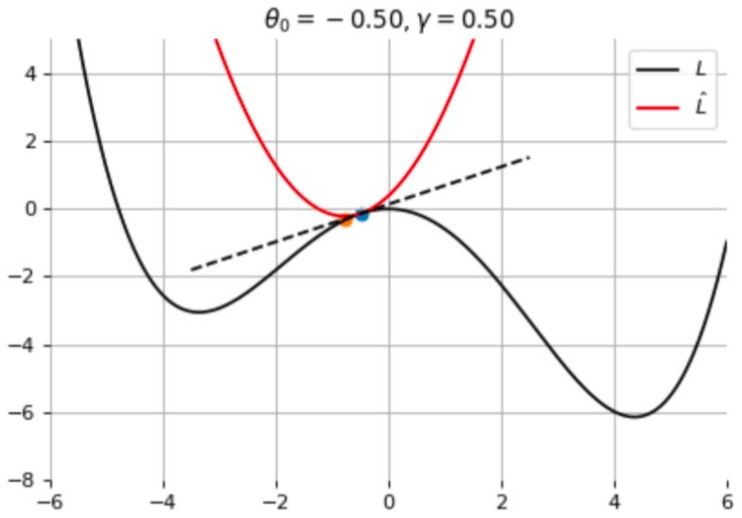
Gradient Descent

Let $\mathcal{L}(\theta)$ denote a loss function defined over model parameters θ (e.g., $\mathbf{w}$ and b).

To minimize $\mathcal{L}(\theta)$, gradient descent uses local linear information to iteratively move towards a (local) minimum.

For $\theta_0 \in \mathbb{R}^d$, a first-order approximation around θ_0 can be defined as

$$\hat{\mathcal{L}}(\theta_0 + \epsilon) = \mathcal{L}(\theta_0) + \epsilon^T \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{2\gamma}||\epsilon||^2.$$



Gradient Descent

A minimizer of the approximation $\hat{\mathcal{L}}(\theta_0 + \epsilon)$ is given for

$$\begin{aligned} \nabla_\epsilon \hat{\mathcal{L}}(\theta_0 + \epsilon) &= 0 \\ &= \nabla_\theta \mathcal{L}(\theta_0) + \frac{1}{\gamma}\epsilon, \end{aligned}$$

which results in the best improvement for the step $\epsilon = -\gamma \nabla_\theta \mathcal{L}(\theta_0)$.

Therefore, model parameters can be updated iteratively using the update rule

$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(\theta_t),$$

where

- θ_0 are the initial parameters of the model;
- γ is the learning rate;
- both are critical for the convergence of the update rule.

Stochastic Gradient Descent

In the empirical risk minimization setup, $\mathcal{L}(\theta)$ and its gradient decompose as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i; \theta))$$

$$\nabla \mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Therefore, in batch gradient descent the complexity of an update grows linearly with the size N of the dataset.

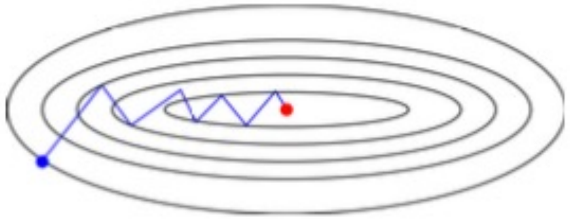
More importantly, since the empirical risk is already an approximation of the expected risk, it should not be necessary to carry out the minimization with great accuracy.

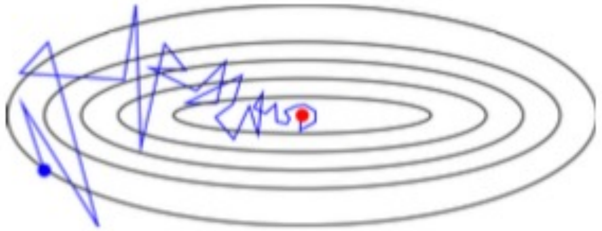
Stochastic Gradient Descent

Instead, stochastic gradient descent uses as update rule:

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of N.
- The stochastic process $\{\theta_t | t = 1, \dots\}$ depends on the examples $i(t)$ picked randomly at each iteration.


Batch gradient descent


Stochastic gradient descent

Stochastic Gradient Descent

Why is stochastic gradient descent still a good idea?

- Informally, averaging the update

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

  over all choices $i(t+1)$ restores batch gradient descent.

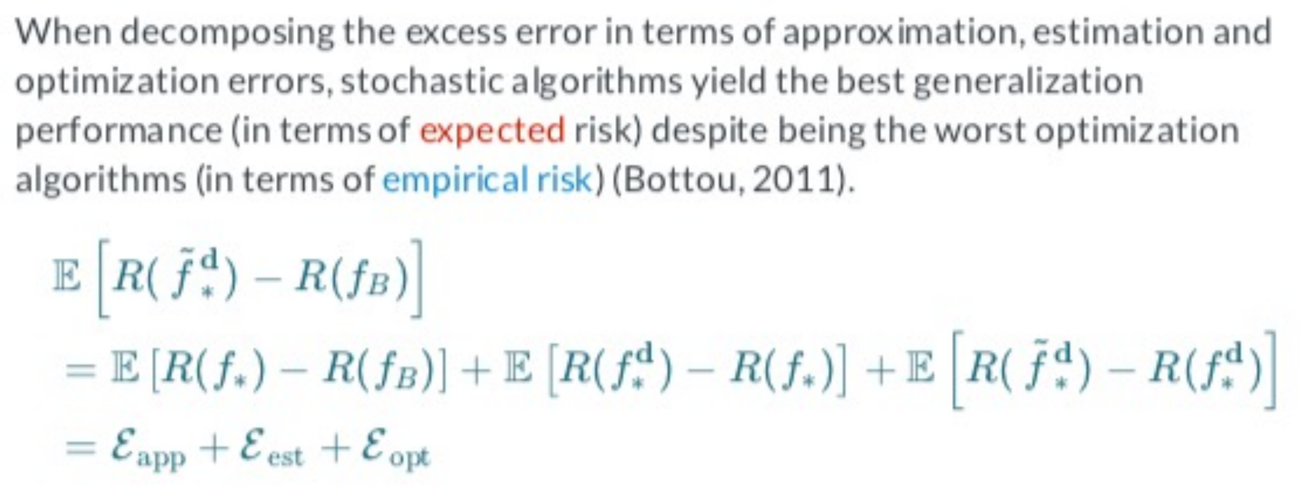
- Formally, if the gradient estimate is unbiased, e.g., if

$$\begin{aligned}\mathbb{E}_{i(t+1)}[\nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))] &= \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta_t)) \\ &= \nabla \mathcal{L}(\theta_t)\end{aligned}$$

  then the formal convergence of SGD can be proved, under appropriate assumptions

- Interestingly, if training examples $\mathbf{x}_i, y_i \sim P_{X,Y}$ are received and used in an online fashion, then SGD directly minimizes the expected risk.

Stochastic Gradient Descent



When decomposing the excess error in terms of approximation, estimation and optimization errors, stochastic algorithms yield the best generalization performance (in terms of expected risk) despite being the worst optimization algorithms (in terms of empirical risk) (Bottou, 2011).

$$\mathbb{E}\left[R(\tilde{f}^{\mathbf{d}}_{*}) - R(f_B)\right]$$
$$= \mathbb{E}\left[R(f_*) - R(f_B)\right] + \mathbb{E}\left[R(f^{\mathbf{d}}_{*}) - R(f_*)\right] + \mathbb{E}\left[R(\tilde{f}^{\mathbf{d}}_{*}) - R(f^{\mathbf{d}}_{*})\right]$$
$$= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}$$

Divergence: Leave-one-out cross-validation

- Consider a training dataset with m examples. We need to choose the best value for a hyper-parameter d:

1. For each d:
   (a) Repeat the following procedure m times:
      i. Leave out *ith instance* from the training set, to estimate the true prediction error; we will put it in a *validation set*
      ii. Use all the other instances to find best parameter vector, $\mathbf{w}_{d,i}$
      iii. Measure the error in predicting the label on the instance left out, for the $\mathbf{w}_{d,i}$ parameter vector; call this $J_{d,i}$
      iv. This is a *(mostly) unbiased estimate of the true prediction error*
   (b) Compute the average of the estimated errors: $J_d = \frac{1}{m}\sum_{i=1}^{m} J_{d,i}$
2. Choose the d with lowest average estimated error: $d^* = \arg\min_d J(d)$

Can also generalize to k-fold cross-validation: divide the training data into k even portions, and use each portion as the validation data (the others are training data) in turn

Divergence: Leave-one-out cross-validation

d	Error$_{\text{train}}$	Error$_{\text{valid}}$ (J_d)
1	0.2188	0.3558
2	0.1504	0.3095
3	0.1384	0.4764
4	0.1259	1.1770
5	0.0742	1.2828
6	0.0598	1.3896
7	0.0458	38.819
8	0.0000	6097.5
9	0.0000	6097.5

- Typical overfitting behavior: as d increases, the training error decreases, but the validation error decreases, then starts increasing again
- Optimal choice: $d = 2$. Overfitting for $d > 2$

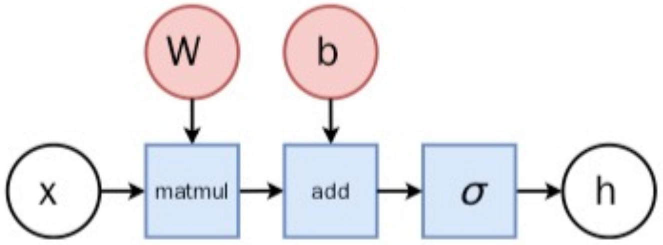
Layers

So far we considered the logistic unit $h = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)$, where $h \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^p$, $\mathbf{w} \in \mathbb{R}^p$ and $b \in \mathbb{R}$.

These units can be composed in parallel to form a layer with q outputs:

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

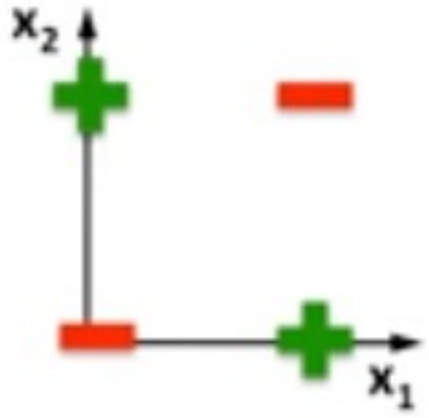
where $\mathbf{h} \in \mathbb{R}^q, \mathbf{x} \in \mathbb{R}^p, \mathbf{W} \in \mathbb{R}^{p\times q}, b \in \mathbb{R}^q$ and where $\sigma(\cdot)$ is upgraded to the element-wise sigmoid function.



Can we solve the non-linearly separate data now?

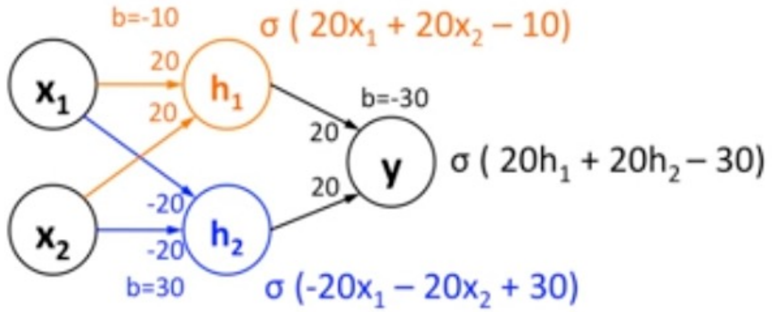
Can logistic regression or LDA solve this XOR problem now (i.e., get zero error on the training data)?

Linear classifiers cannot solve this



Can we solve the non-linearly separate data now?

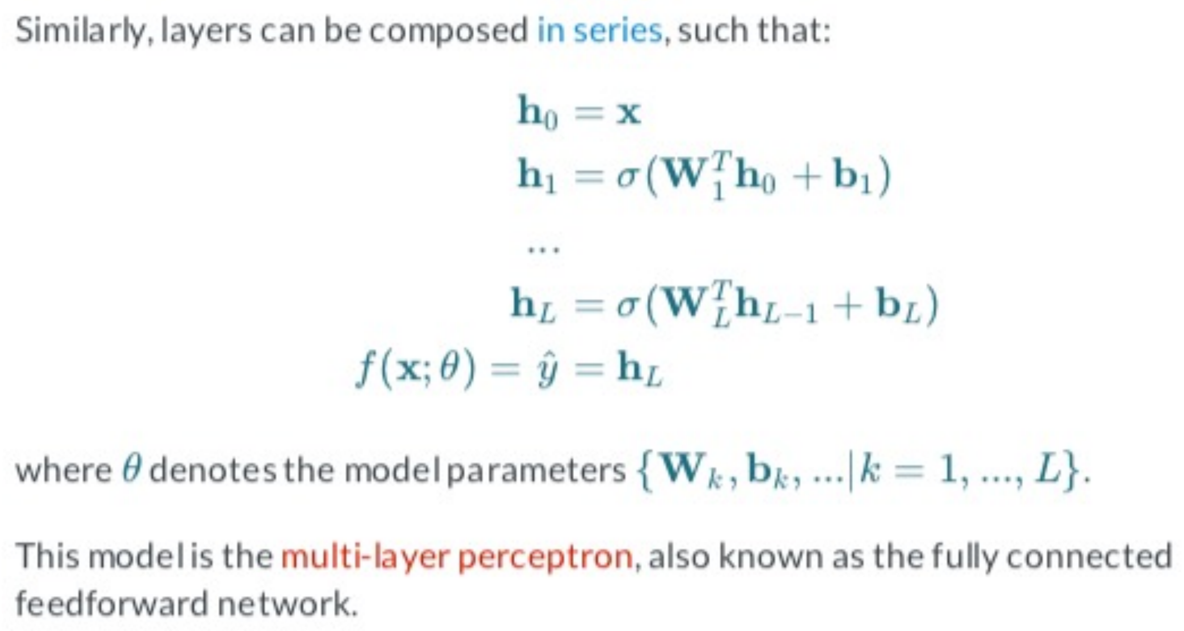
- With a single neuron, we cannot do it! No way to draw a hyperplane to separate the data. This is why neural nets die for the first time.
- But with two neurons, we can!



b=-10
$\sigma(20x_1 + 20x_2 - 10)$
20
20
x_1
h_1
b=-30
20
20
y
$\sigma(20h_1 + 20h_2 - 30)$
-20
-20
x_2
h_2
b=30
$\sigma(-20x_1 - 20x_2 + 30)$

$\sigma(20*0 + 20*0 - 10) \approx 0$	$\sigma(-20*0 - 20*0 + 30) \approx 1$	$\sigma(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$	$\sigma(-20*1 - 20*1 + 30) \approx 0$	$\sigma(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*0 + 20*1 - 10) \approx 1$	$\sigma(-20*0 - 20*1 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$	$\sigma(-20*1 - 20*0 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$

Multi-layer Perceptron/Neural Nets (MLPs)



Similarly, layers can be composed in series, such that:

$$\begin{aligned} \mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\ &\dots \\ \mathbf{h}_L &= \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L) \\ f(\mathbf{x};\theta) = \hat{y} &= \mathbf{h}_L \end{aligned}$$

where θ denotes the model parameters $\{\mathbf{W}_k, \mathbf{b}_k, ... | k = 1, ..., L\}$.

This model is the multi-layer perceptron, also known as the fully connected feedforward network.

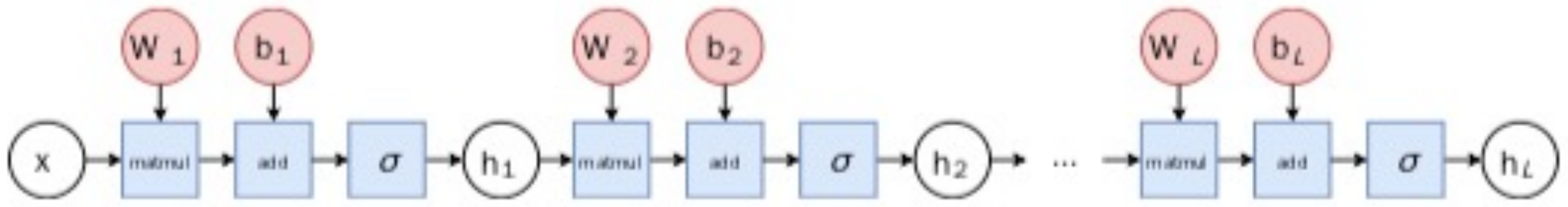
What if we don't have the non-linear functions?

Activation Functions

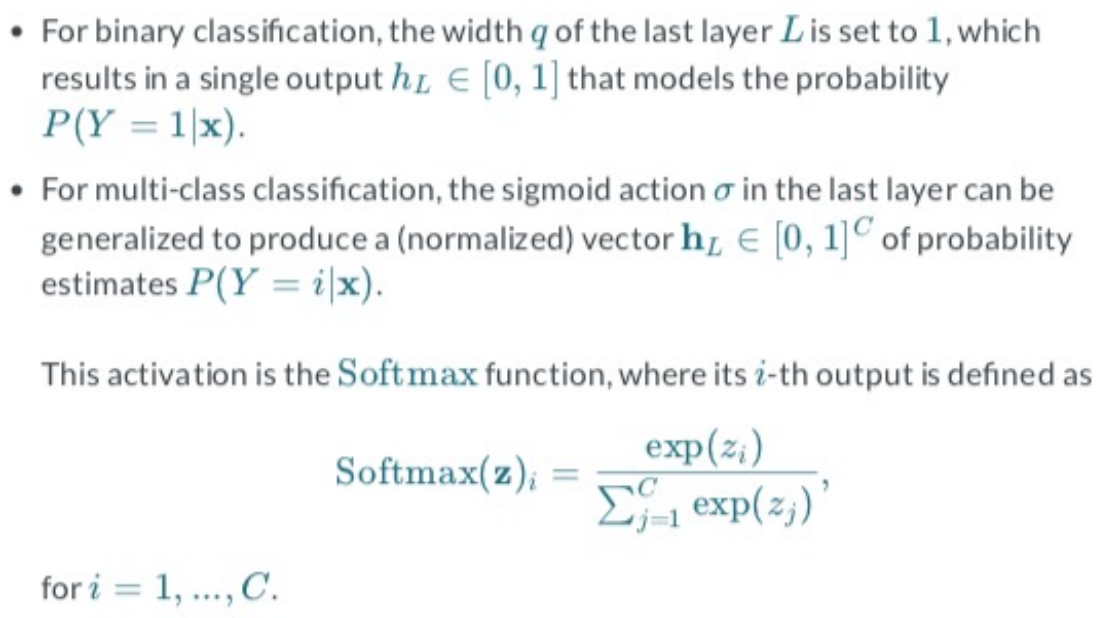
Also called “link functions”

$$\mathrm{sign}(a)$$
$$\sigma(a) = \frac{1}{1+e^{-a}}$$
$$\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$$
$$\mathrm{ReLU}(a) = \max(a,0)$$
$$\mathrm{SoftPlus}(a) = \log(1+e^{a})$$
$$\mathrm{ELU}(a) = \left\{ \begin{array}{ll} a, & \text{for } a \geq 0 \\ \alpha(e^{a}-1), & \text{for } a < 0 \end{array} \right\}$$

Computational Graph



Classification



- For binary classification, the width q of the last layer L is set to 1, which results in a single output $h_L \in [0,1]$ that models the probability $P(Y=1|\mathbf{x})$.
- For multi-class classification, the sigmoid action σ in the last layer can be generalized to produce a (normalized) vector $\mathbf{h}_L \in [0,1]^C$ of probability estimates $P(Y=i|\mathbf{x})$.

  This activation is the $\mathrm{Softmax}$ function, where its i-th output is defined as

$$\mathrm{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)},$$

  for $i = 1, \ldots, C$.

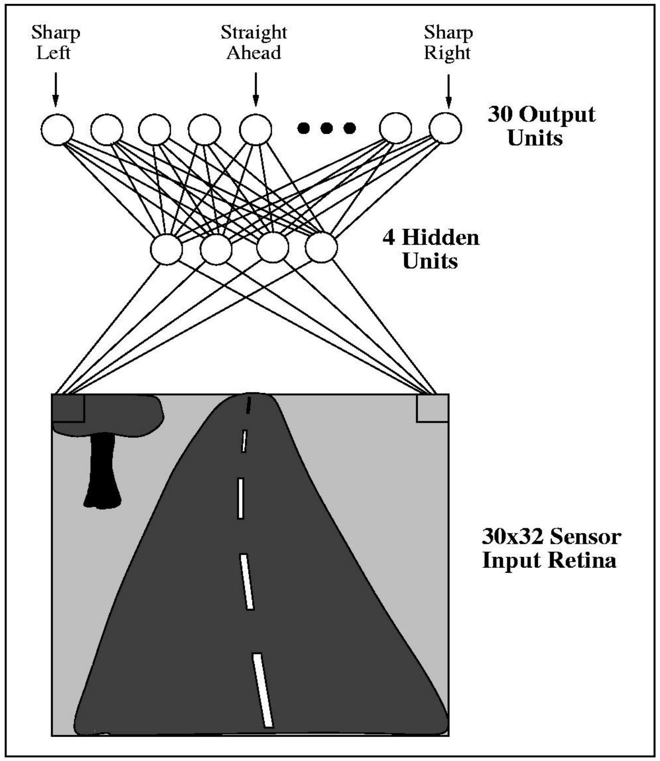
What is the loss function in this multi-class setting?

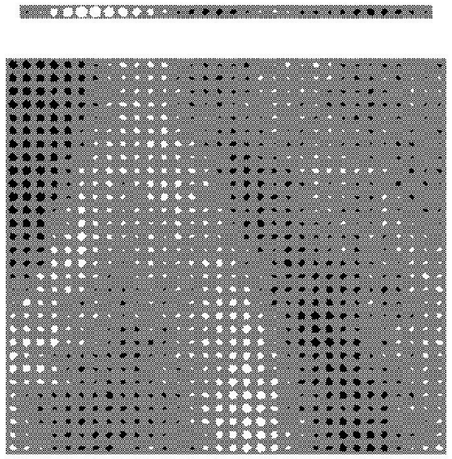
Regression

The last activation σ can be skipped to produce unbounded output values $h_L \in \mathbb{R}$.

Self-driving cars

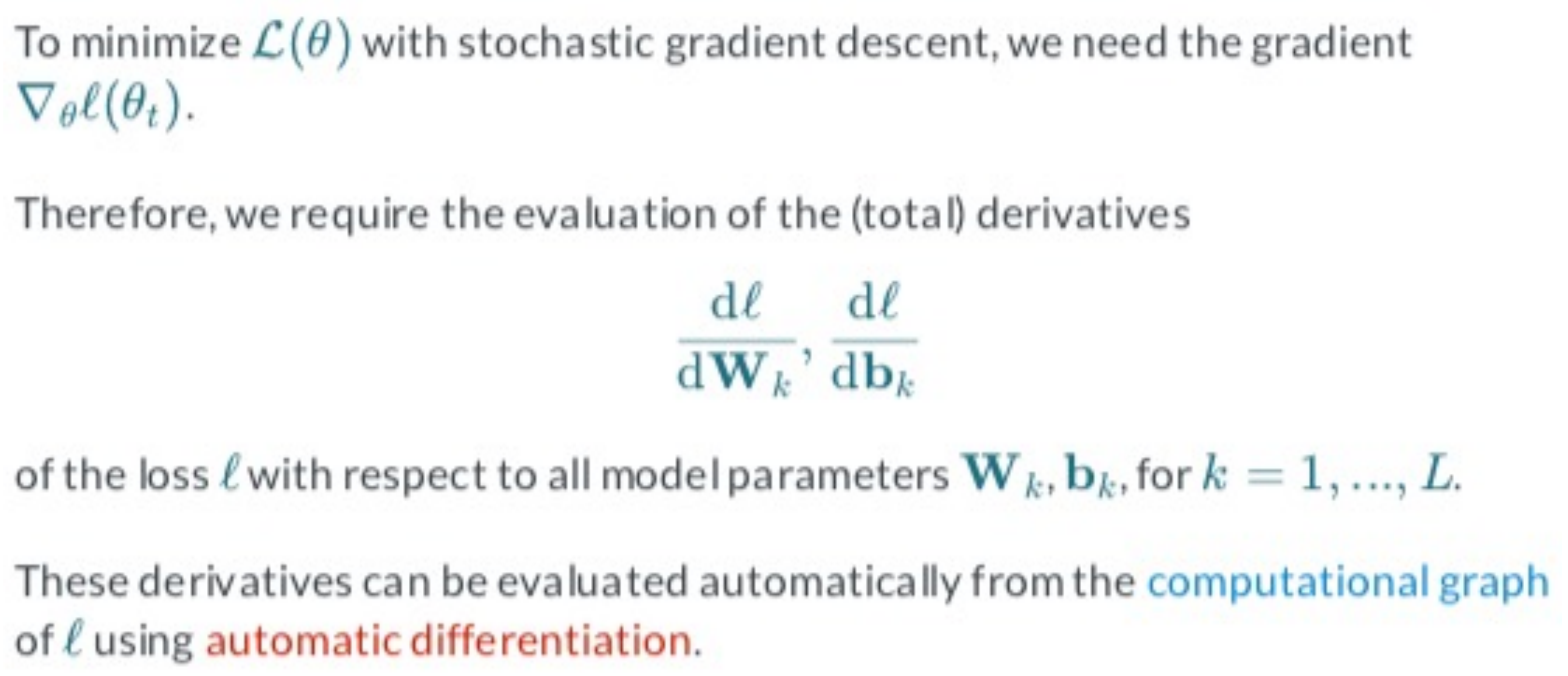
Self-driving cars





ALVINN: Autonomous Land Vehicle In a Neural Network (1989)

Automatic Differentiation



To minimize $\mathcal{L}(\theta)$ with stochastic gradient descent, we need the gradient $\nabla_\theta \ell(\theta_t)$.

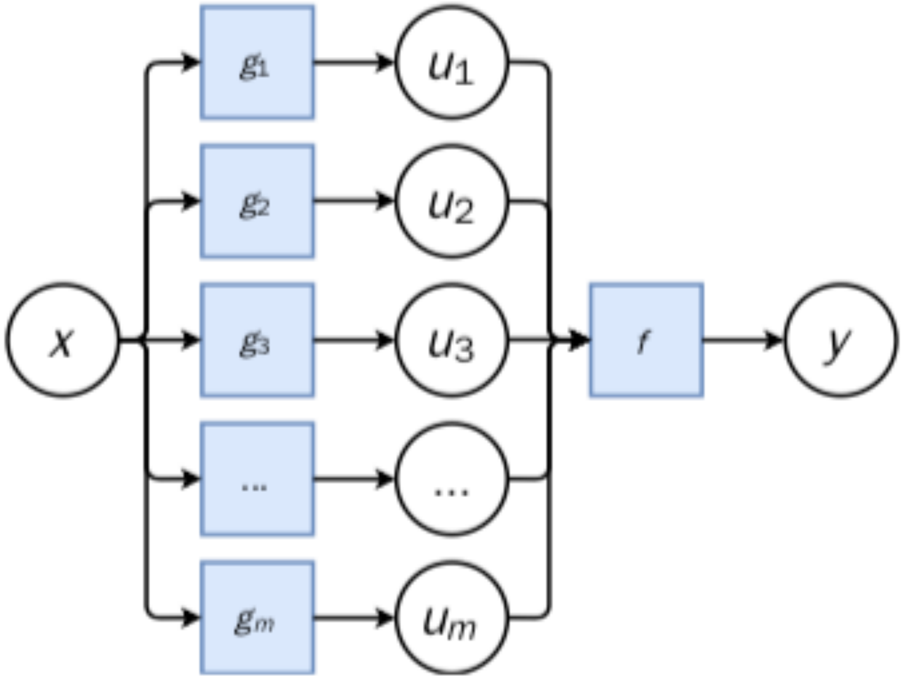
Therefore, we require the evaluation of the (total) derivatives

$$\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_k}, \frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{b}_k}$$

of the loss ℓ with respect to all model parameters $\mathbf{W}_k$, $\mathbf{b}_k$, for $k = 1, ..., L$.

These derivatives can be evaluated automatically from the computational graph of ℓ using automatic differentiation.

Chain Rule



Let us consider a 1-dimensional output composition $f \circ g$, such that

$$y = f(\mathbf{u})$$
$$\mathbf{u} = g(x) = (g_1(x), \ldots, g_m(x)).$$

Chain Rule

The chain rule states that $(f \circ g)' = (f' \circ g)g'$.

For the total derivative, the chain rule generalizes to

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \underbrace{\frac{\mathrm{d}u_k}{\mathrm{d}x}}_{\text{recursive case}}$$

Reserve Automatic Differentiation

- Since a neural network is a composition of differential functions, the total derivatives of the loss can be evaluated backward, by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called reserve automatic differentiation.

Example

Let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma\left(\mathbf{W}_2^T \sigma\left(\mathbf{W}_1^T \mathbf{x}\right)\right)$$
$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \mathrm{cross\_ent}(y, \hat{y}) + \lambda\left(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2\right)$$

for $\mathbf{x} \in \mathbb{R}^p$, $y \in \mathbb{R}$, $\mathbf{W}_1 \in \mathbb{R}^{p \times q}$ and $\mathbf{W}_2 \in \mathbb{R}^q$.

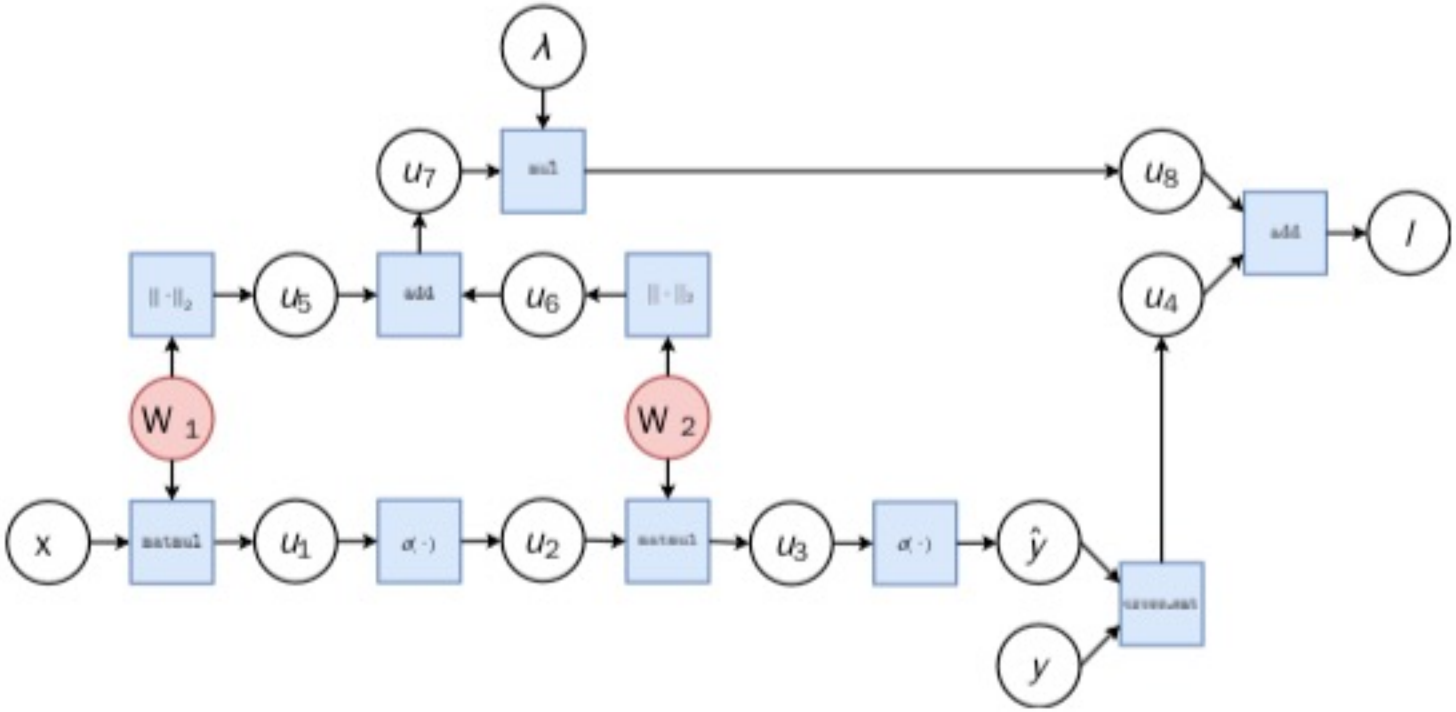
Example

Let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma\left(\mathbf{W}_2^T \sigma\left(\mathbf{W}_1^T \mathbf{x}\right)\right)$$
$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \text{cross\_ent}(y, \hat{y}) + \lambda\left(||\mathbf{W}_1||_2 + ||\mathbf{W}_2||_2\right)$$

for $\mathbf{x} \in \mathbb{R}^p, y \in \mathbb{R}, \mathbf{W}_1 \in \mathbb{R}^{p \times q}$ and $\mathbf{W}_2 \in \mathbb{R}^q$.

In the forward pass, intermediate values are all computed from inputs to outputs, which results in the annotated computational graph below:

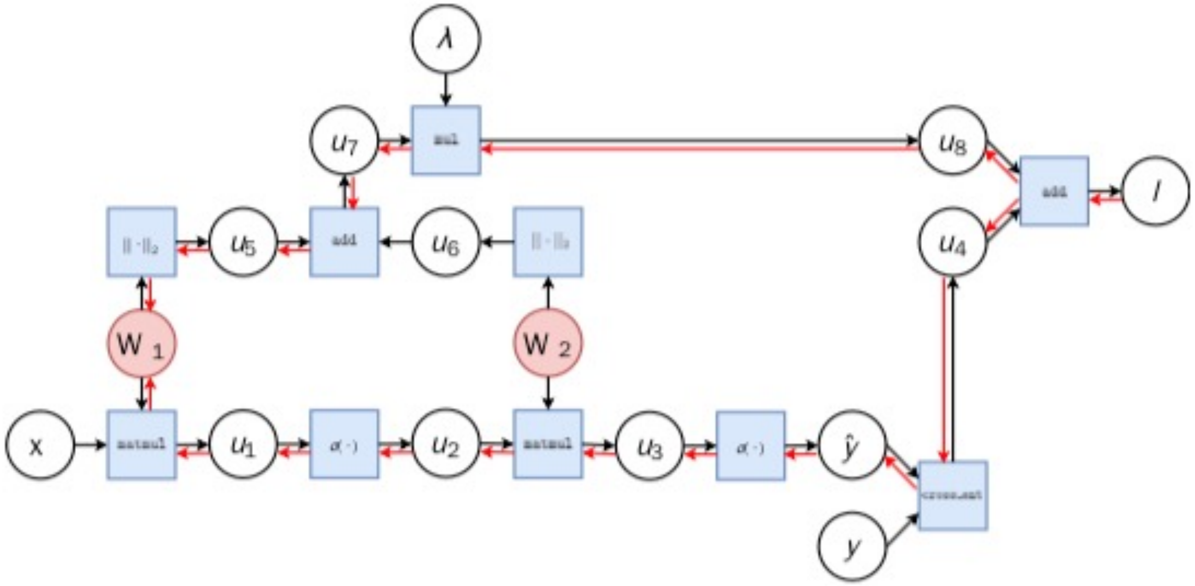


Example

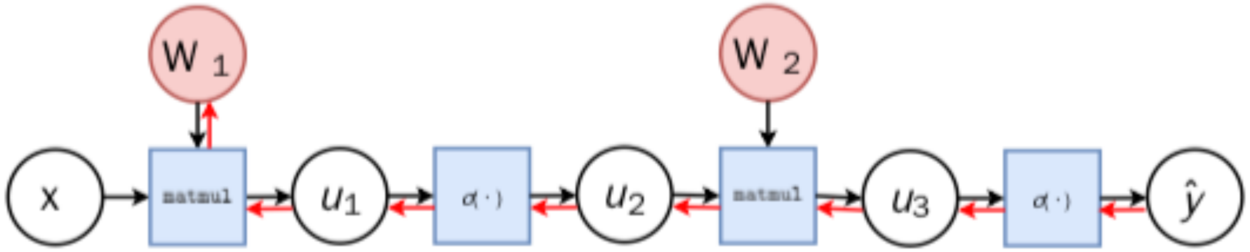
The total derivative can be computed through a backward pass, by walking through all paths from outputs to parameters in the computational graph and accumulating the terms. For example, for $\frac{d\ell}{d\mathbf{W}_1}$ we have:

$$\frac{d\ell}{d\mathbf{W}_1} = \frac{\partial \ell}{\partial u_8}\frac{du_8}{d\mathbf{W}_1} + \frac{\partial \ell}{\partial u_4}\frac{du_4}{d\mathbf{W}_1}$$

$$\frac{du_8}{d\mathbf{W}_1} = \ldots$$



Example



Let us zoom in on the computation of the network output $\hat{y}$ and of its derivative with respect to $\mathbf{W}_1$.

- Forward pass: values u_1, u_2, u_3 and $\hat{y}$ are computed by traversing the graph from inputs to outputs given $\mathbf{x}$, $\mathbf{W}_1$ and $\mathbf{W}_2$.
- Backward pass: by the chain rule we have

$$\begin{aligned}\frac{\mathrm{d}\hat{y}}{\mathrm{d}\mathbf{W}_1} &= \frac{\partial \hat{y}}{\partial u_3}\frac{\partial u_3}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial \mathbf{W}_1} \\ &= \frac{\partial \sigma(u_3)}{\partial u_3}\frac{\partial \mathbf{W}_2^T u_2}{\partial u_2}\frac{\partial \sigma(u_1)}{\partial u_1}\frac{\partial \mathbf{W}_1^T u_1}{\partial \mathbf{W}_1}\end{aligned}$$

Note how evaluating the partial derivatives requires the intermediate values computed forward.

Back-propagation

- This algorithm is also known as **back-propagation**
- An equivalent procedure can be defined to evaluate the derivatives in forward mode, from inputs to outputs.
- Since differentiation is a linear operator, automatic differentiation can be implemented efficiently in terms of tensor operations.

Back-propagation

- Gradient descent + chain rule
- Want to minimize overall loss (e.g., squared loss):

$$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\Big(\underbrace{y_n - \sum_i v_i \overbrace{f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)}^{h_{n,i}}}_{e_n}\Big)^2$$

- Gradient for outer weights v, where h_n is hidden units:

$$\nabla_v = -\sum_n e_n \boldsymbol{h}_n$$

Back-propagation, continued:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right)^2$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial \boldsymbol{w}_i}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right) v_i = -e v_i$$

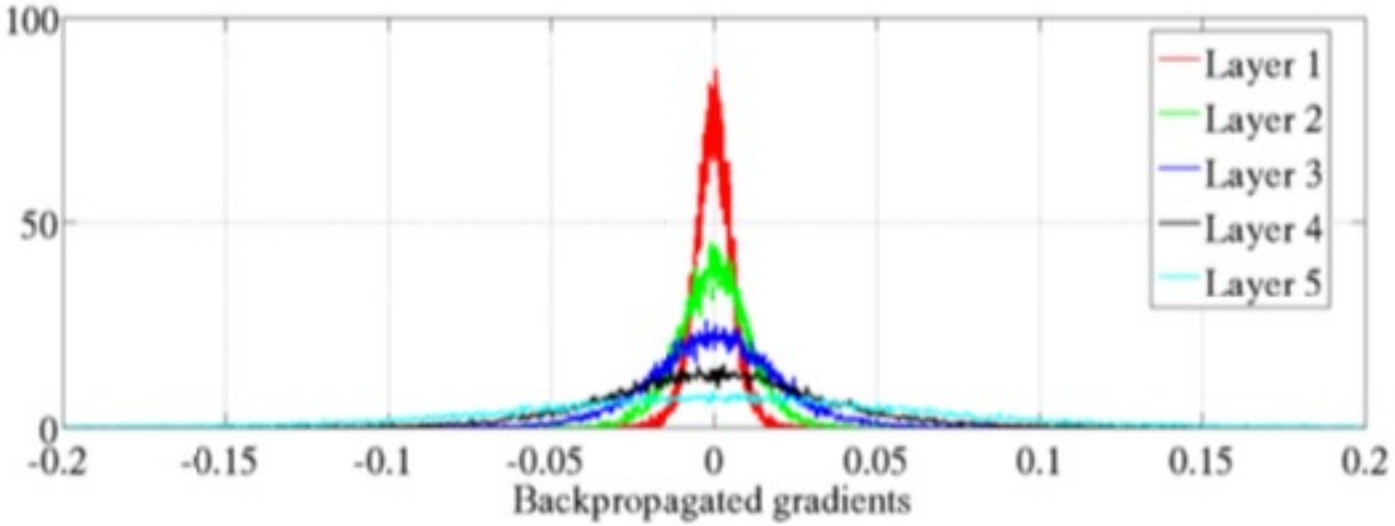
$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

$$\nabla_{\boldsymbol{w}_i} = -e v_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

Vanishing Gradients

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the vanishing gradient problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010). Gradients for layers far from the output vanish to zero.

Vanishing Gradients

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Under the hood, this would be evaluated as

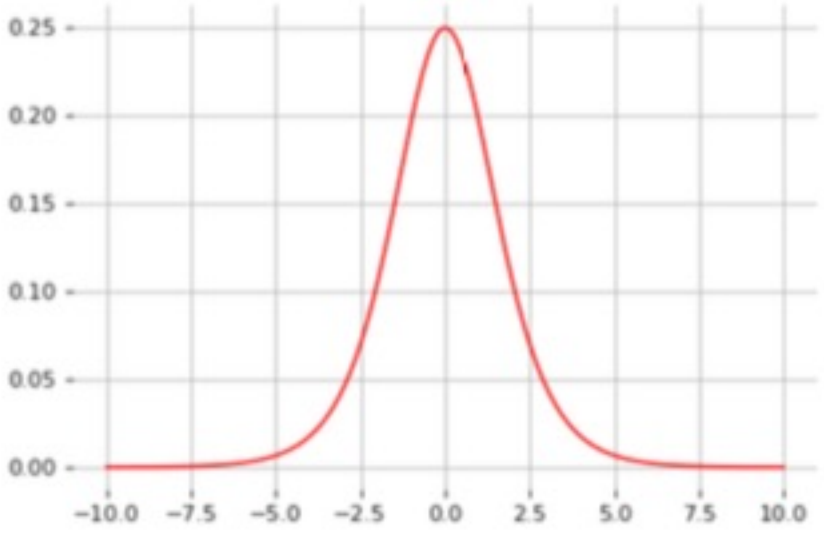
$$\begin{aligned} u_1 &= w_1 x \\ u_2 &= \sigma(u_1) \\ u_3 &= w_2 u_2 \\ u_4 &= \sigma(u_3) \\ u_5 &= w_3 u_4 \\ \hat{y} &= \sigma(u_5) \end{aligned}$$

and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

$$\begin{aligned} \frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x \end{aligned}$$

Vanishing Gradients

The derivative of the sigmoid activation function σ is:



$$\frac{d\sigma}{dx}(x) = \sigma(x)(1 - \sigma(x))$$

Notice that $0 \leq \frac{d\sigma}{dx}(x) \leq \frac{1}{4}$ for all x.

Vanishing Gradients

Assume that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.

Then,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{\leq\frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{\leq\frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq\frac{1}{4}} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ exponentially shrinks to zero as the number of layers in the network increases.

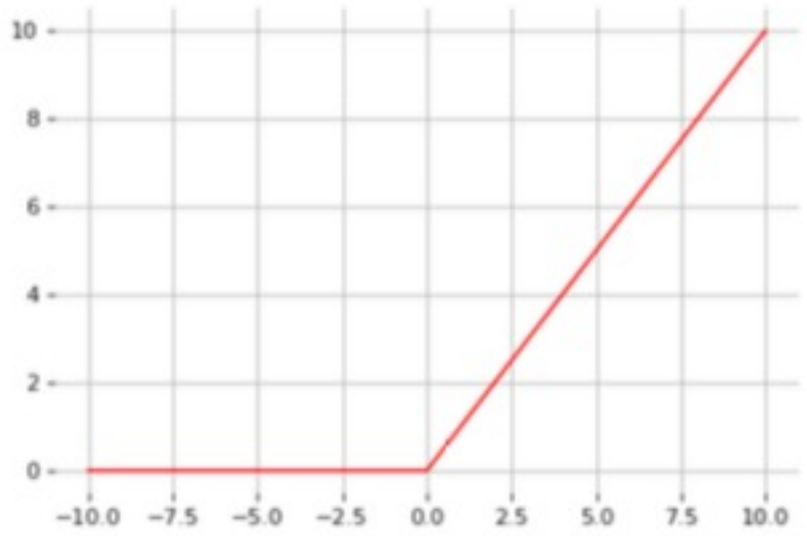
Hence the vanishing gradient problem.

- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

Rectified Linear Units (ReLU)

Instead of the sigmoid activation function, modern neural networks are for most based on rectified linear units (ReLU) (Glorot et al, 2011):

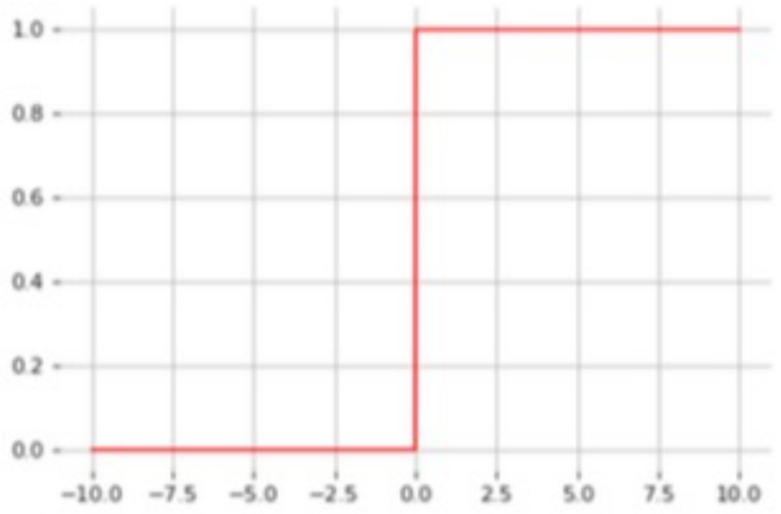
$$\mathrm{ReLU}(x) = \max(0, x)$$



Rectified Linear Units (ReLU)

Note that the derivative of the ReLU function is

$$\frac{d}{dx}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



For $x = 0$, the derivative is undefined. In practice, it is set to zero.

Rectified Linear Units (ReLU)

Therefore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial\sigma(u_1)}{\partial u_1}}_{=1} x$$

This solves the vanishing gradient problem, even for deep networks! (provided proper initialization)

Note that:

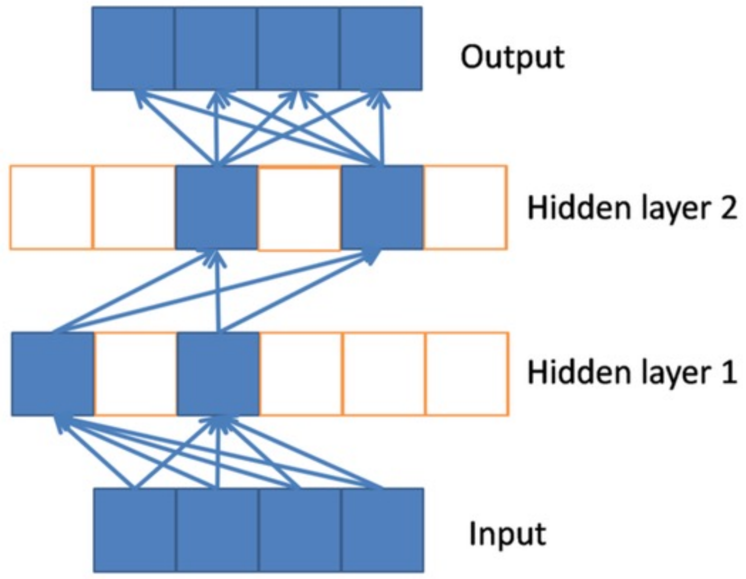
- The ReLU unit dies when its input is negative, which might block gradient descent.
- This is actually a useful property to induce sparsity.
- This issue can also be solved using leaky ReLUs, defined as

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x)$$

for a small $\alpha \in \mathbb{R}^+$ (e.g., $\alpha = 0.1$).

Sparsity in ReLU

- From biology: if the inputs sum to less than zero, don't let the signal pass, but if it sums to greater than zero, let the signal pass (hyperbolic tangent or sigmoid are approximators, but cannot achieve true zero activation)
- Biological neurons encode information in a "sparse and distributed way". This means that the percentage of neurons that are active *at the same time* are very low (1–4%).



Universal Approximation

Theorem. (Cybenko 1989; Hornik et al, 1991) Let $\sigma(\cdot)$ be a bounded, non-constant continuous function. Let I_p denote the p-dimensional hypercube, and $C(I_p)$ denote the space of continuous functions on I_p. Given any $f \in C(I_p)$ and $\epsilon > 0$, there exists $q > 0$ and $v_i, w_i, b_i, i = 1, ..., q$ such that

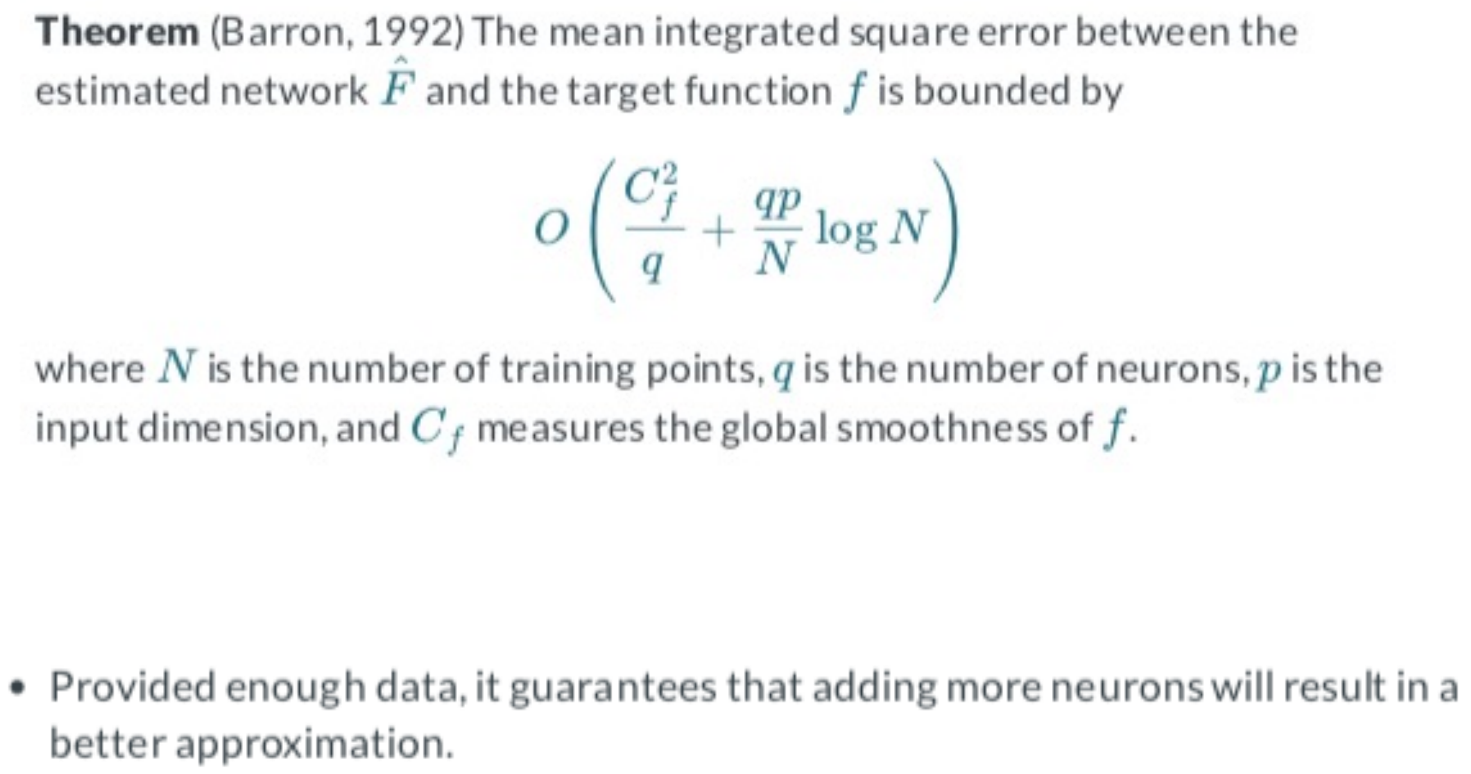
$$F(x) = \sum_{i \leq q} v_i \sigma(w_i^T x + b_i)$$

satisfies

$$\sup_{x \in I_p} |f(x) - F(x)| < \epsilon.$$

- It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth);
- It does not inform about good/bad architectures, nor how they relate to the optimization procedure.
- The universal approximation theorem generalizes to any non-polynomial (possibly unbounded) activation function, including the ReLU (Leshno, 1993).

Universal Approximation



Theorem (Barron, 1992) The mean integrated square error between the estimated network $\hat{F}$ and the target function f is bounded by

$$O\left(\frac{C_f^2}{q} + \frac{qp}{N}\log N\right)$$

where N is the number of training points, q is the number of neurons, p is the input dimension, and C_f measures the global smoothness of f.

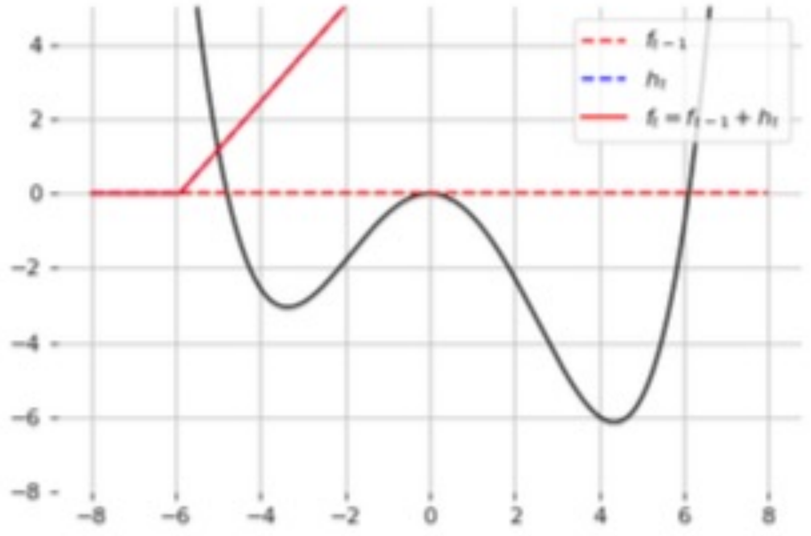
- Provided enough data, it guarantees that adding more neurons will result in a better approximation.

Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

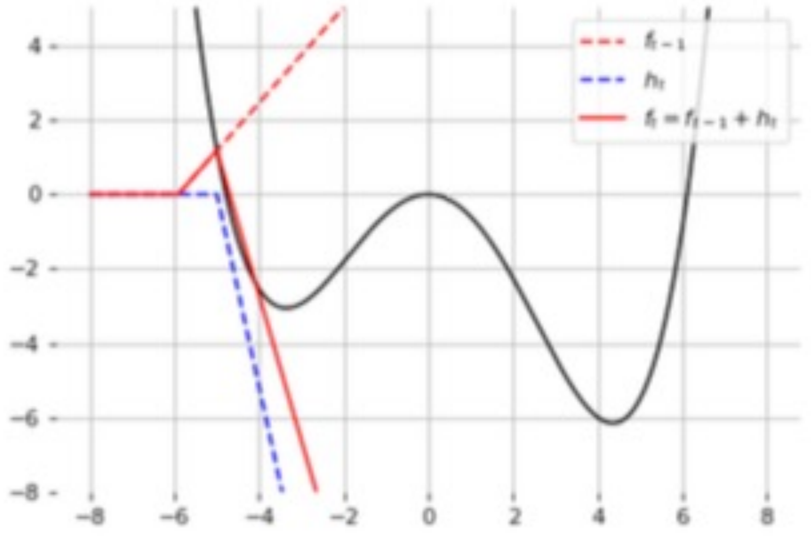


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

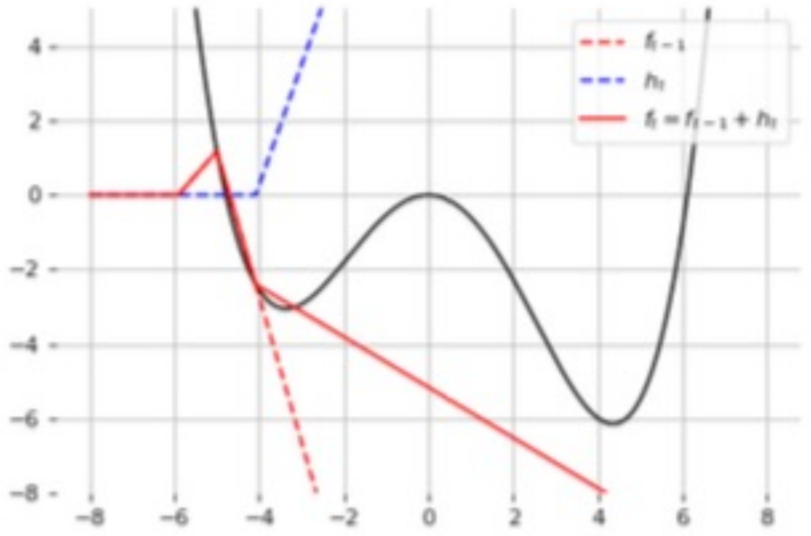


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

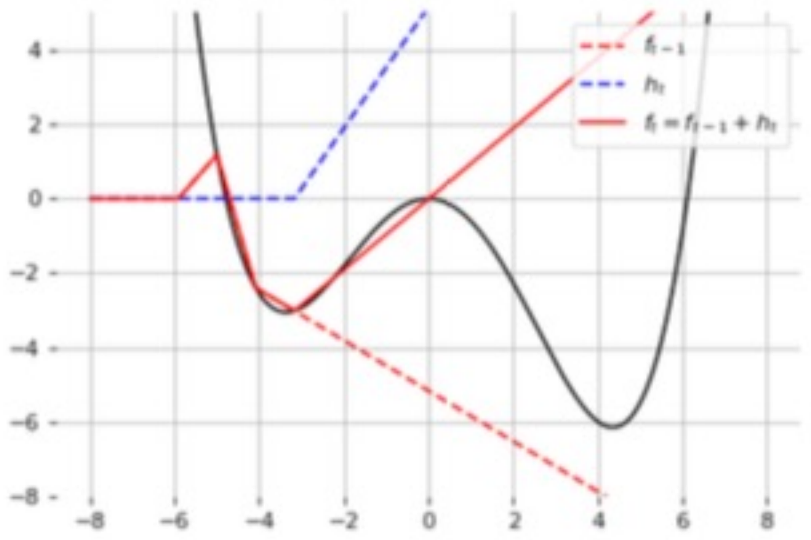


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

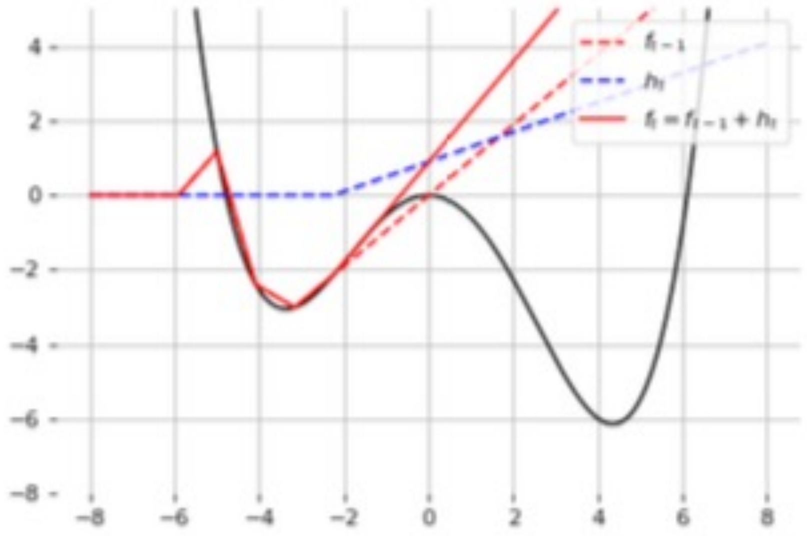


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

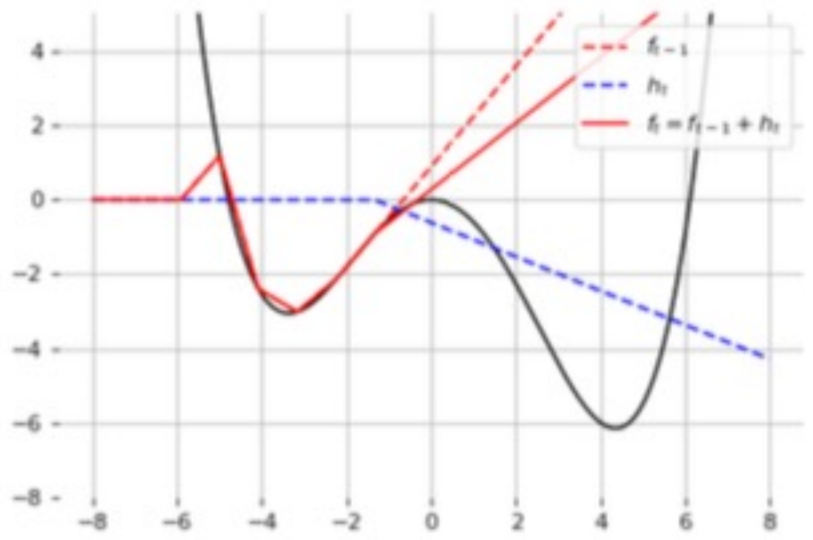


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

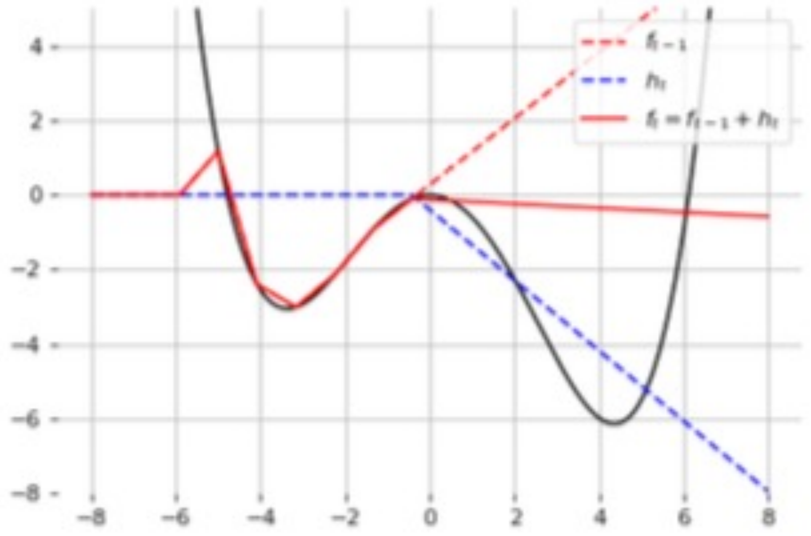


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

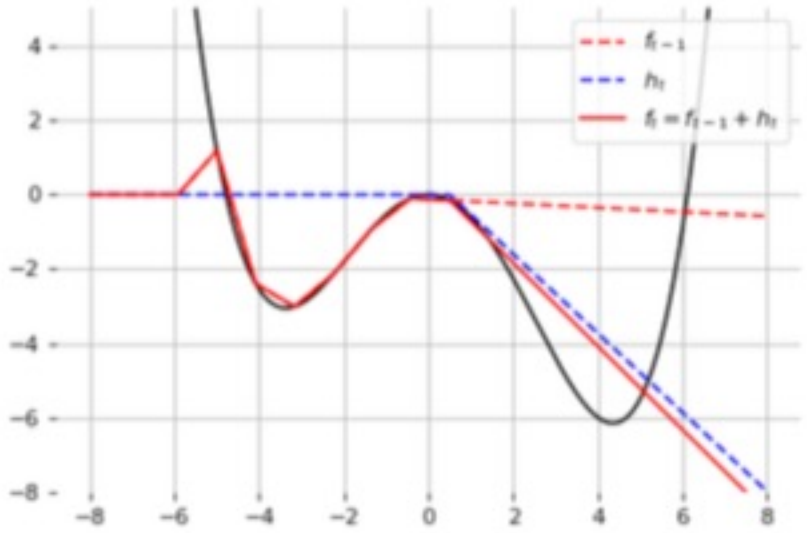


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

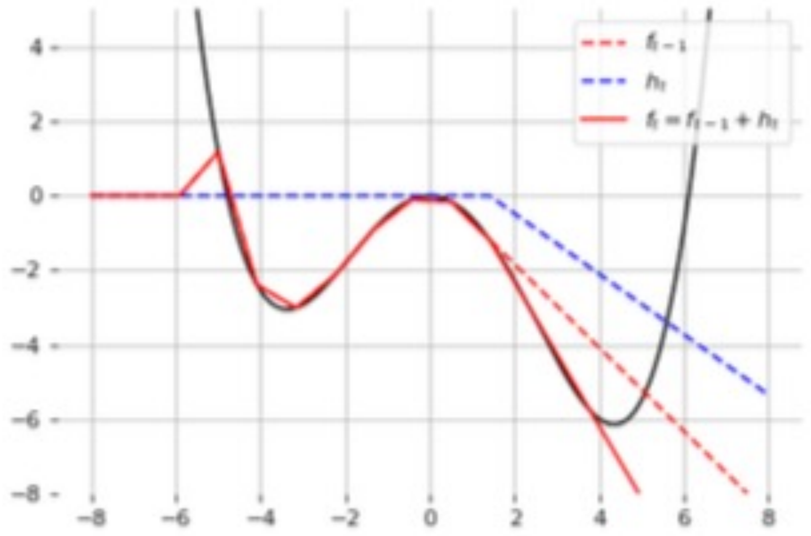


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

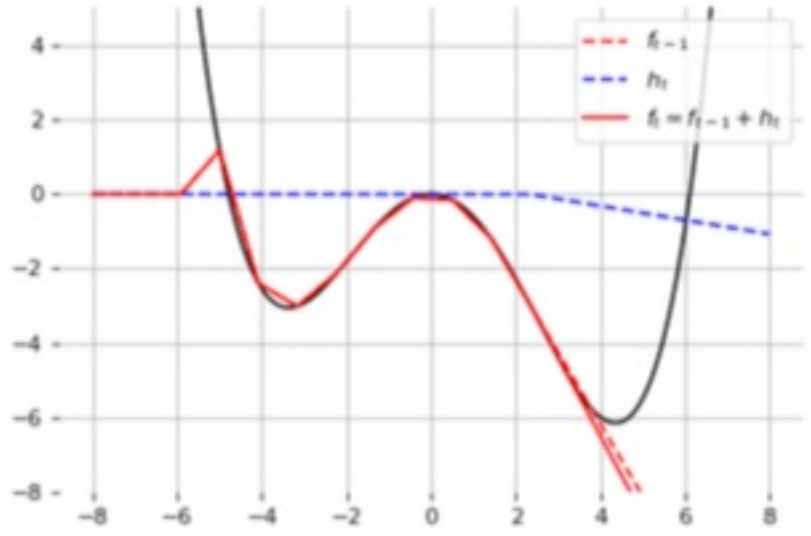


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

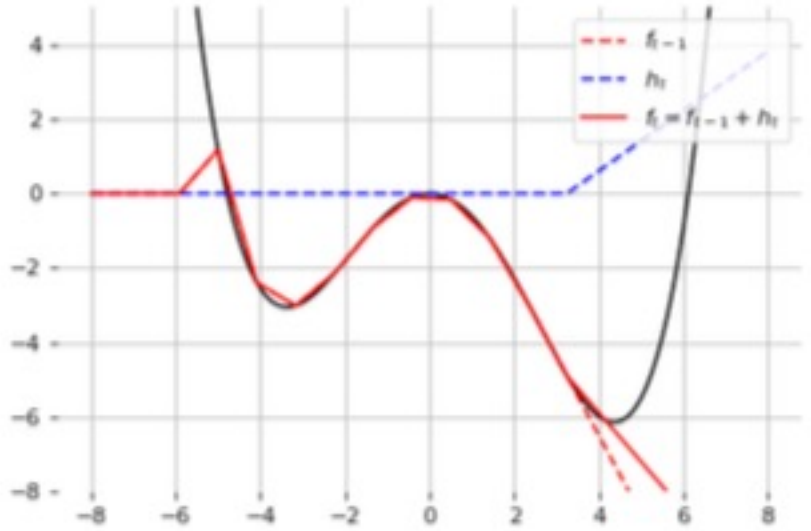


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

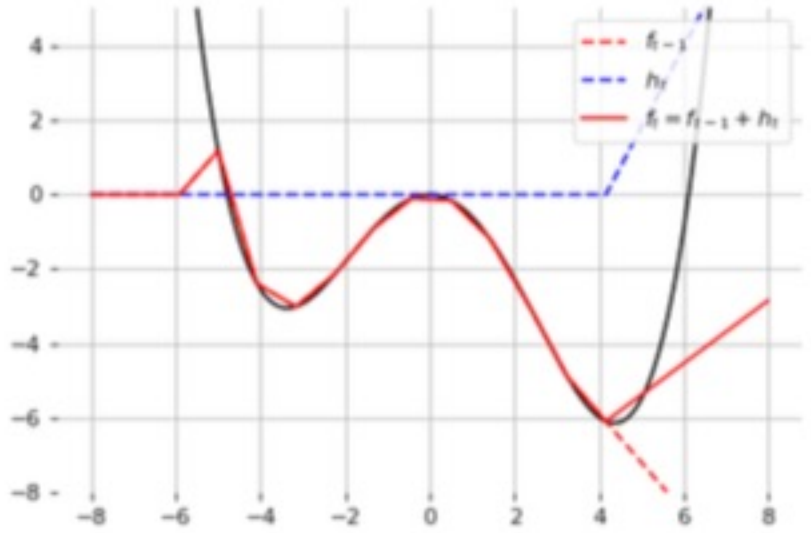


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \mathrm{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.

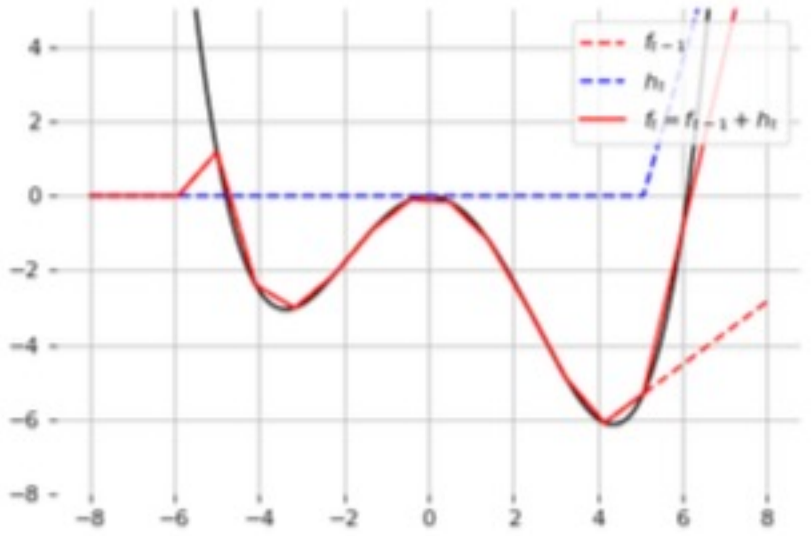


Example

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.



Deep Learning

Recent advances and model architectures in deep learning are built on a natural generalization of a neural network: a graph of tensor operators, taking advantage of

- the chain rule
- stochastic gradient descent
- convolutions
- parallel operations on GPUs.

This generalization allows to compose and design complex networks of operators, possibly dynamically, dealing with images, sound, text, sequences, etc. and to train them end-to-end.