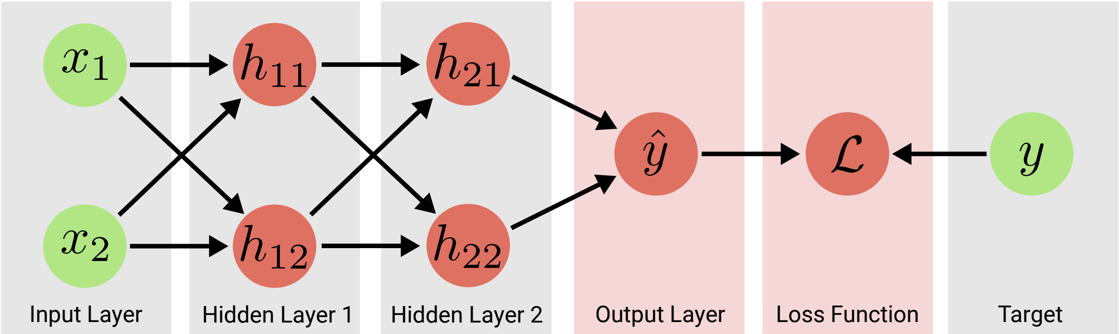
4.1
Output and Loss Functions

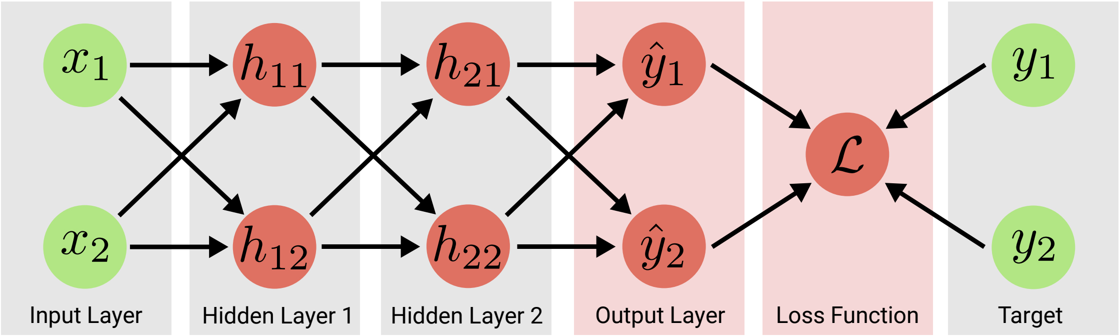
By: Andreas Ginger

Output and Loss Functions



- The **output layer** is the last layer in a neural network which computes the output
- The **loss function** compares the result of the output layer to the target value(s)
- Choice of output layer and loss function depends on task (discrete, continuous, ..)

Output and Loss Functions

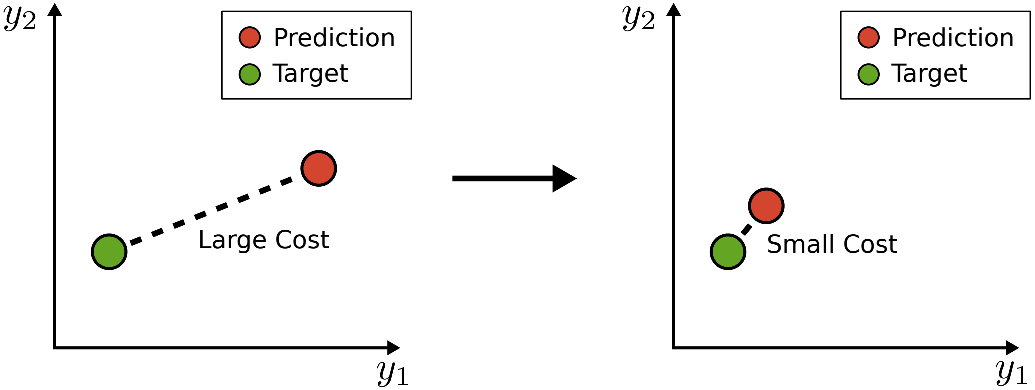


- The **output layer** is the last layer in a neural network which computes the output
- The **loss function** compares the result of the output layer to the target value(s)
- Choice of output layer and loss function depends on task (discrete, continuous, ..)

Loss Function

What is the goal of optimizing the loss function?

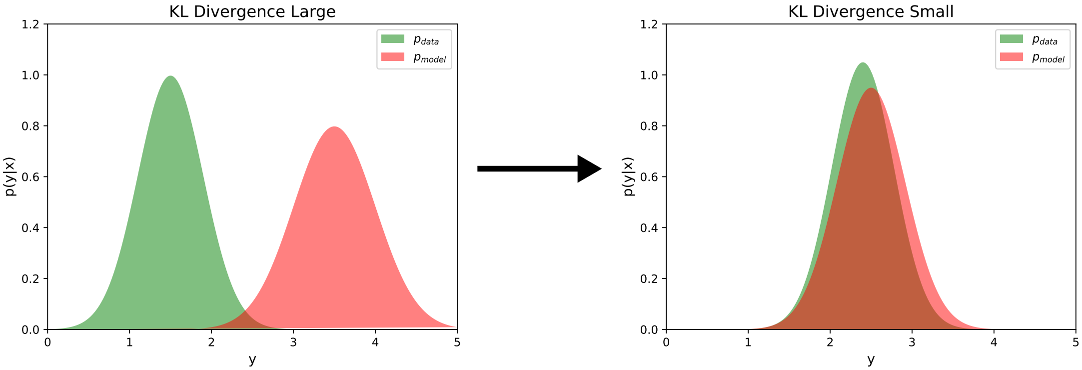
- Tries to make the **model output** (=prediction) similar to the **target** (=data)
- Think of the loss function as a **measure of cost** being paid for a prediction



Loss Function

What is the goal of optimizing the loss function?

- Tries to make the **model output** (=prediction) similar to the **target** (=data)
- Think of the loss function as a **measure of cost** being paid for a prediction



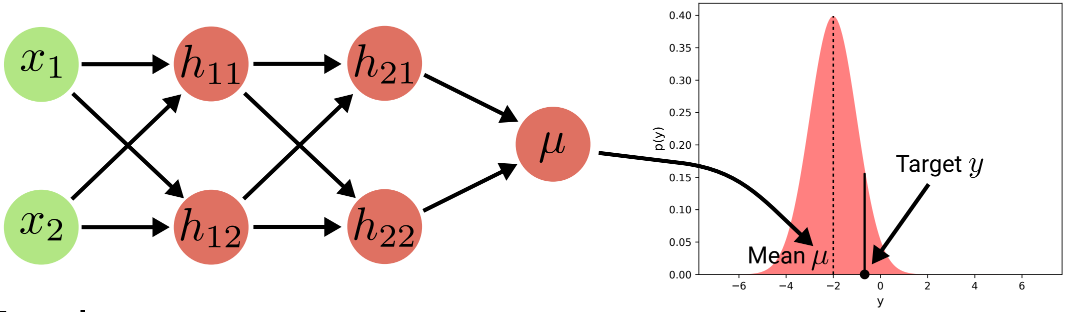
Loss Function

How to design a good loss function?

- A loss function can be any differentiable function that we wish to optimize
- Deriving the cost function from the **maximum likelihood principle** removes the burden of manually designing the cost function for each model
- Consider the output of the neural network as **parameters of a distribution** over y_i

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}}\ p_{model}(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\
&\overset{\text{iid}}{=} \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{i=1}^{N} p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Log-Likelihood}}
\end{aligned}
$$

Loss Function



Example:

- Neural network $f_{\mathbf{w}}(\mathbf{x})$ predicts mean μ of Gaussian distribution over y:

$$p(y|\mathbf{x},\mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f_{\mathbf{w}}(\mathbf{x}))^2}{2\sigma^2}\right)$$

- We want to maximize the probability of the target y under this distribution

Recap: Regression

Input | Model | Output

$\longrightarrow$ $\boxed{f_{\mathbf{w}}}$ $\longrightarrow$ 143,52 €

- **Mapping:** $f_{\mathbf{w}} : \mathbb{R}^N \rightarrow \mathbb{R}$

Recap: Binary Classification

Input | Model | Output

$\rightarrow$ $\boxed{f_{\mathbf{w}}}$ $\rightarrow$ "Beach"

- **Mapping:** $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow \{\text{"Beach"}, \text{"No Beach"}\}$

Recap: Multi-Class Classification

Input

Model

Output

$\rightarrow$ $f_{\mathbf{w}}$ $\rightarrow$ "Beach"

- **Mapping:** $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \rightarrow$ {"Beach", "Mountain", "City", "Forest"}

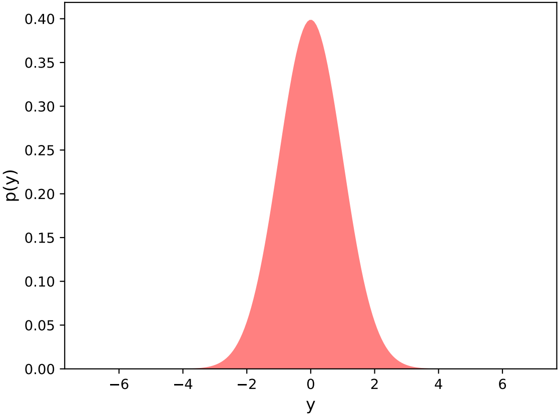
Regression Problems

Gaussian Distribution

Gaussian distribution:

$$p(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$$

- μ : mean
- σ : standard deviation
- The distribution has thin "tails": $p(y) \to 0$ quickly as $y \to \infty$
- It thus penalizes outliers strongly



Gaussian Distribution / L_2 Loss

Let $p_{model}(y|\mathbf{x},\mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-f_{\mathbf{w}}(\mathbf{x}))^2}{2\sigma^2}\right)$ be a **Gaussian distribution**. We obtain:

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i,\mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} \frac{1}{2}\log(2\pi\sigma^2) - \sum_{i=1}^{N} \frac{1}{2\sigma^2}\left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} \left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{\left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2}_{L_2 \text{ Loss}}
\end{aligned}
$$

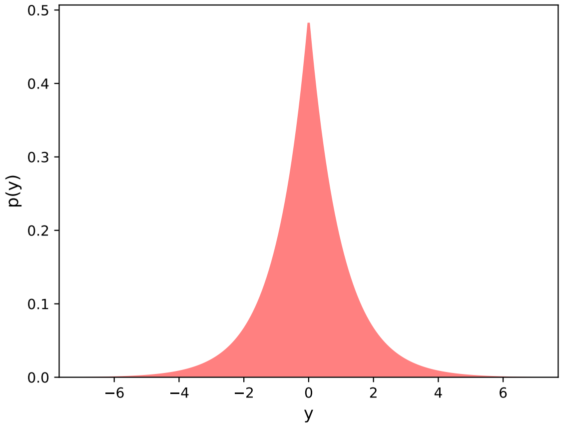
In other words, we minimize the **squared loss** (=L_2 loss), affected strongly by outliers.

Laplace Distribution

Laplace distribution:

$$p(y) = \frac{1}{2b} \exp\left(-\frac{|y-\mu|}{b}\right)$$

- μ : location
- b : scale
- The distribution has heavy "tails": $p(y) \to 0$ more slowly as $y \to \infty$
- Penalizes outliers less strongly
- Thus often preferred in practice



Laplace Distribution / L_1 Loss

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = \frac{1}{2b} \exp\left(-\frac{|y - f_{\mathbf{w}}(\mathbf{x})|}{b}\right)$ be a **Laplace distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} - \sum_{i=1}^{N} \log(2b) - \sum_{i=1}^{N} \frac{1}{b} |f_{\mathbf{w}}(\mathbf{x}_i) - y_i| \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} - \sum_{i=1}^{N} |f_{\mathbf{w}}(\mathbf{x}_i) - y_i| \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{|f_{\mathbf{w}}(\mathbf{x}_i) - y_i|}_{L_1 \text{ Loss}}
\end{aligned}$$

We minimize the **absolute loss** (=L_1 loss) which is more robust than L_2.

Predicting all Parameters

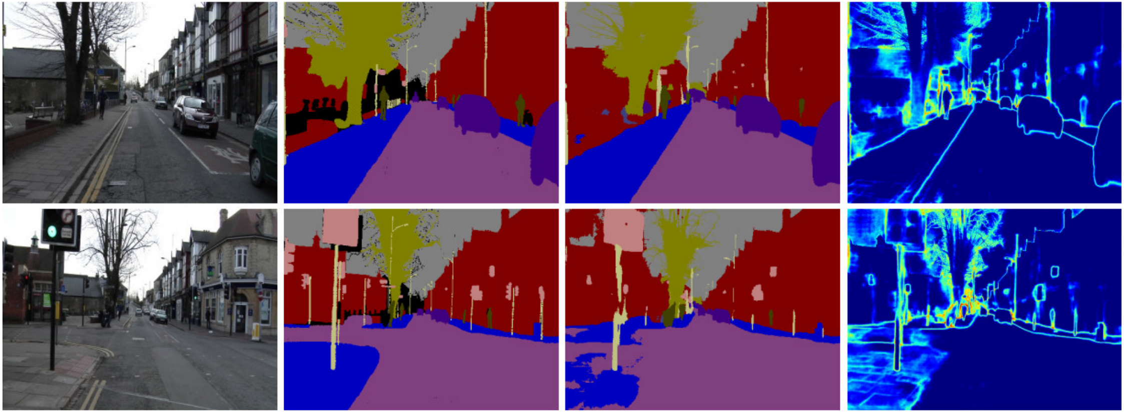
Let $p_{model}(y|\mathbf{x},\mathbf{w}) = \frac{1}{2\,g_{\mathbf{w}}(\mathbf{x})}\exp\left(-\frac{|y-f_{\mathbf{w}}(\mathbf{x})|}{g_{\mathbf{w}}(\mathbf{x})}\right)$ be a **Laplace distribution**. We obtain:

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i,\mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} \log(2\,g_{\mathbf{w}}(\mathbf{x})) - \sum_{i=1}^{N} \frac{1}{g_{\mathbf{w}}(\mathbf{x})}|f_{\mathbf{w}}(\mathbf{x}_i) - y_i|
\end{aligned}
$$

In this case, we predict both the **location** μ and the **scale** b with a neural network. $f_{\mathbf{w}}(\mathbf{x}_i)$ and $g_{\mathbf{w}}(\mathbf{x})$ are typically the same except for the last layer. Allows for estimating the **aleatoric uncertainty** (observation noise) with the neural network itself.

Kendall, Gal: What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? NIPS, 2017.

Predicting all Parameters



Input Image Ground Truth Segmentation Aleatoric Uncertainty

Kendall, Gal: What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? NIPS, 2017.

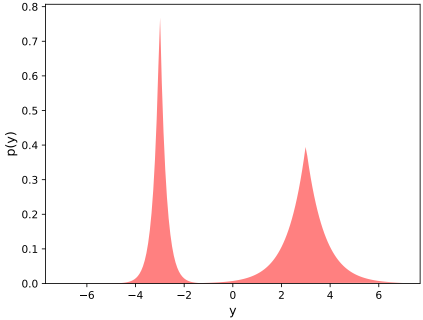
Mixture Density Networks

To represent multi-modal distributions, we can also model **mixture densities:**

$$p_{model}(y|\mathbf{x},\mathbf{w}) = \sum_{m=1}^{M} \pi_m \frac{1}{2\, g_{\mathbf{w}}^{(m)}(\mathbf{x})} \exp\left(- \frac{|y - f_{\mathbf{w}}^{(m)}(\mathbf{x})|}{g_{\mathbf{w}}^{(m)}(\mathbf{x})} \right)$$

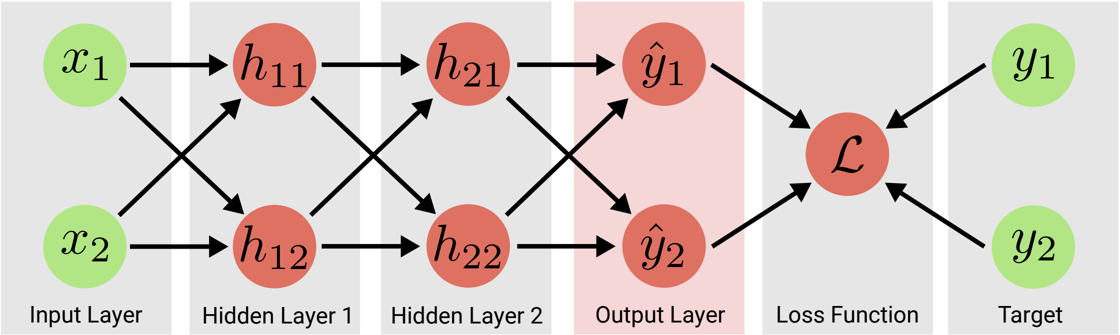
Example:

- Mixture of Laplace distribution
- $\pi_m \in [0,1]$: weight of mode m
- Constraint $\sum_m \pi_m = 1$
- Location μ_m and scale b_m of each mode m modeled by neural network



Bishop. Mixture Density Networks. 1994.

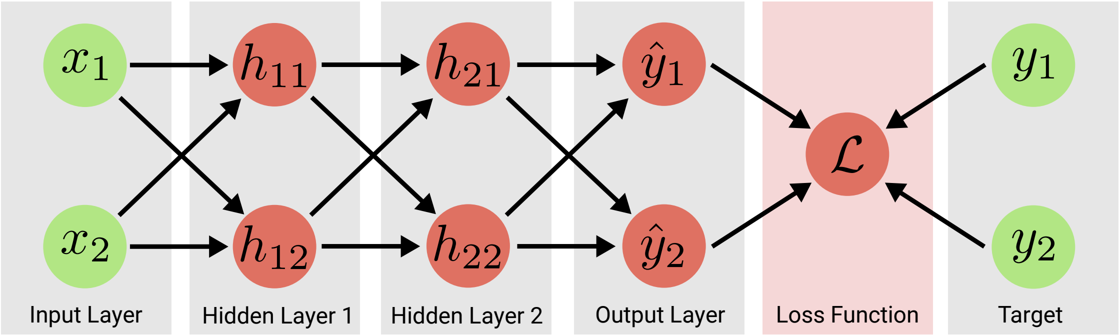
Output Layer for Regression Problems



- For most outputs (e.g., $\mu \in \mathbb{R}$), the output layer is linear (no non-linearity)
- For some outputs (e.g., $b \in \mathbb{R}^+$), we need a squashing function (ReLU, softplus)

Bishop. Mixture Density Networks. 1994.

Loss Function for Regression Problems

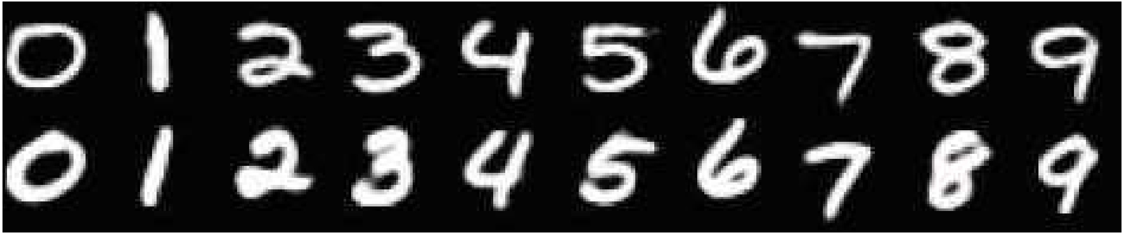


- Gaussian/Laplacian model distribution correspond to L_2 and L_1 loss functions
- It is also possible to predict uncertainty (variance/scale) or multiple modes (MDN)

Bishop. Mixture Density Networks. 1994.

Classification Problems

Image Classification

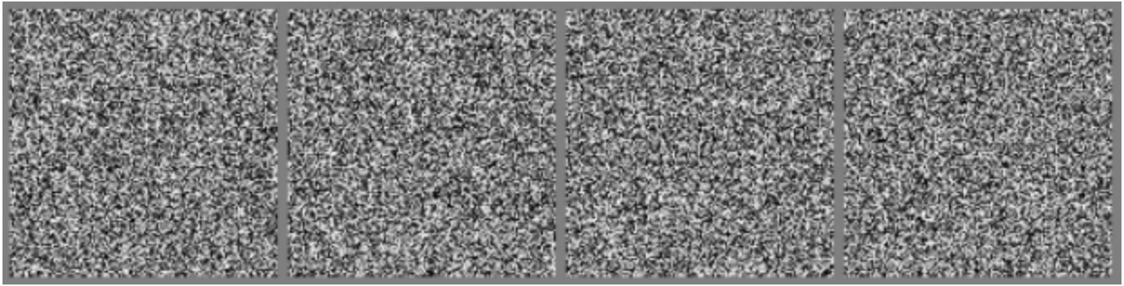


MNIST Handwritten Digits:

- One of the most popular datasets in ML (many variants, still in use today)
- Based on a data from the National Institute of Standards and Technology
- Hand written by Census Bureau employees and high-school children
- Resolution: 28 x 28 pixels, 60k training samples with labels, 10k test samples

LeCun, Bottou, Bengio and Haffner. Gradient-based learning applied to document recognition. IEEE, 1998.

Image Classification



Curse of Dimensionality:

- There exist $2^{784} = 10^{236}$ possible binary images of resolution 28 x 28 pixels
- MNIST is gray-scale, thus 256^{784} combinations $\Rightarrow$ impossible to enumerate
- Why is image classification with just 60k labeled training images even possible?
- Answer: Images concentrated on low-dimensional manifold in $\{1, \ldots, 256\}^{784}$

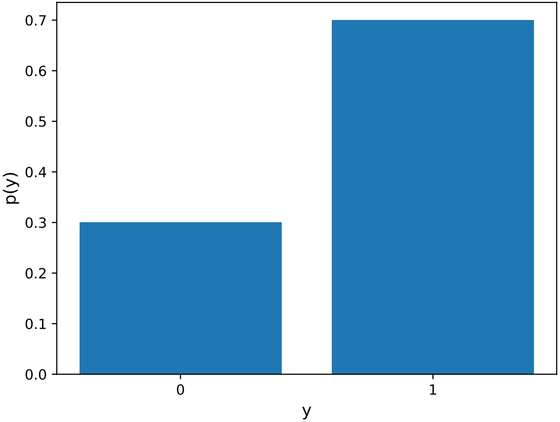
LeCun, Bottou, Bengio and Haffner. Gradient-based learning applied to document recognition. IEEE, 1998.

Bernoulli Distribution

Bernoulli distribution:

$$p(y) = \mu^y \, (1 - \mu)^{(1-y)}$$

- μ: probability for $y = 1$
- Handles only two classes
  e.g. ("cats" vs. "dogs")



Bernoulli Distribution / BCE Loss

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = f_{\mathbf{w}}(\mathbf{x})^y \, (1 - f_{\mathbf{w}}(\mathbf{x}))^{(1-y)}$ be a **Bernoulli distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \left[f_{\mathbf{w}}(\mathbf{x}_i)^{y_i} \, (1 - f_{\mathbf{w}}(\mathbf{x}_i))^{(1-y_i)} \right] \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{-y_i \log f_{\mathbf{w}}(\mathbf{x}_i) - (1 - y_i) \log(1 - f_{\mathbf{w}}(\mathbf{x}_i))}_{\text{BCE Loss}}
\end{aligned}$$

In other words, we minimize the **binary cross-entropy (BCE)** loss.

Remark: Last layer of $f_{\mathbf{w}}(\mathbf{x})$ can be a sigmoid function such that $f_{\mathbf{w}}(\mathbf{x})^y \in [0, 1]$.

How can we scale this to multiple classes?

Categorical Distribution

Categorical distribution:

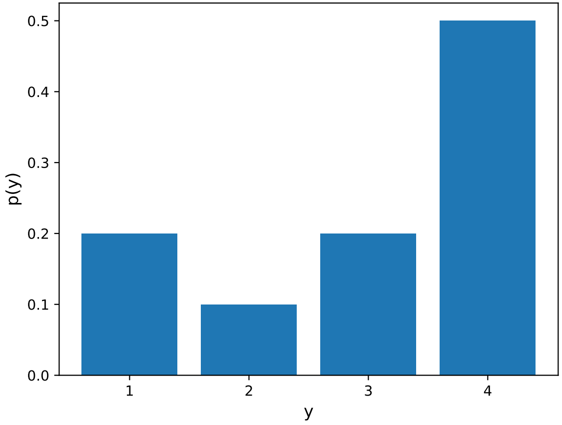
$$p(y = c) = \mu_c$$

- μ_c: probability for class c
- Multiple classes, multiple modes

Alternative notation:

$$p(\mathbf{y}) = \prod_{c=1}^{C} \mu_c^{y_c}$$

- $\mathbf{y}$: "one-hot" vector with $y_c \in \{0, 1\}$
- $\mathbf{y} = (0, \dots, 0, 1, 0, \dots, 0)^\top$ with all zeros except for one (the true class)



One-Hot Vector Representation

class	y	$\mathbf{y}$
	1	$(1,0,0,0)^\top$
	2	$(0,1,0,0)^\top$
	3	$(0,0,1,0)^\top$
	4	$(0,0,0,1)^\top$

- One-hot vector $\mathbf{y}$ with binary elements $y_c \in \{0,1\}$
- Index c with $y_c = 1$ determines the correct class, and $y_k = 0$ for $k \neq c$
- Interpretation as discrete distribution with all probability mass at the true class
- Often used in ML as it can make formalism more convenient

Categorical Distribution / CE Loss

Let $p_{model}(\mathbf{y}|\mathbf{x},\mathbf{w}) = \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x})^{y_c}$ be a **Categorical distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i,\mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)^{y_{i,c}} \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{\sum_{c=1}^{C} -y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)}_{\text{CE Loss}}
\end{aligned}$$

In other words, we minimize the **cross-entropy (CE)** loss.

The target $\mathbf{y} = (0,\ldots,0,1,0,\ldots,0)^\top$ is a “one-hot” vector with y_c its c’th element.

Softmax

How can we ensure that $f_{\mathbf{w}}^{(c)}(\mathbf{x})$ predicts a **valid Categorical (discrete) distribution?**

- We must guarantee (1) $f_{\mathbf{w}}^{(c)}(\mathbf{x}) \in [0,1]$ and (2) $\sum_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}) = 1$
- An element-wise sigmoid as output function would ensure (1) but not (2)
- Solution: The **softmax function** guarantees both (1) and (2):

$$\text{softmax}(\mathbf{x}) = \left(\frac{\exp(x_1)}{\sum_{k=1}^{C} \exp(x_k)}, \dots, \frac{\exp(x_C)}{\sum_{k=1}^{C} \exp(x_k)} \right)$$

- Let $\mathbf{s}$ denote the network output after the last affine layer (=scores). Then:

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C} \exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- Remark: s_c is a direct contribution to the loss function, i.e., it does not saturate

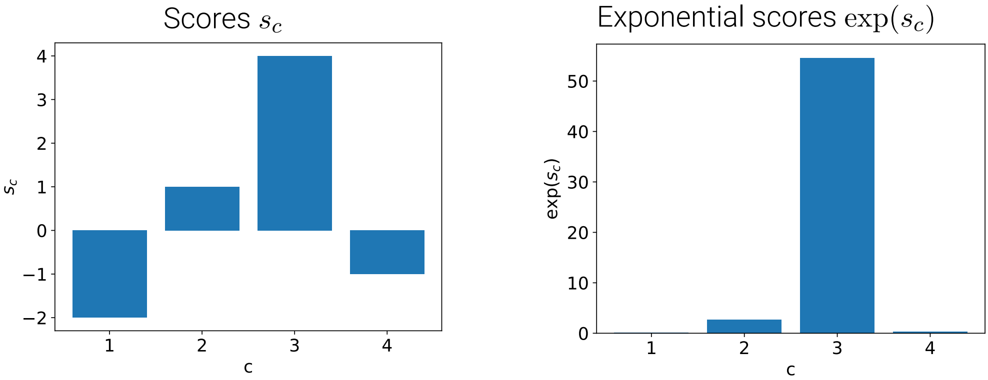
Log Softmax

Intuition: Assume c is the correct class. Our goal is to maximize the log softmax:

$$\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

- The first term encourages the score s_c for the correct class c to increase
- The second term encourages all scores in $\mathbf{s}$ to decrease
- The second term can be approximated by: $\log \sum_{k=1}^{C} \exp(s_k) \approx \max_k s_k$ as $\exp(s_k)$ is insignificant for all $s_k < \max_k s_k$
- Thus, the loss always strongly penalizes the most active incorrect prediction
- If the correct class already has the largest score (i.e., $s_c = \max_k s_k$), both terms roughly cancel and the example will contribute little to the overall training cost

Log Softmax Example



- The second term becomes: $\log \sum_{k=1}^{C} \exp(s_k) = 4.06 \approx s_3 = \max_k s_k$
- For $c = 2$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 1 - 4.06 \approx -3$
- For $c = 3$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 4 - 4.06 \approx 0$

Softmax

- Predicting C values/scores overparameterizes the Categorical distribution
- As the distribution sums to 1 only $C1$ parameters are necessary
- Example: Consider $C = 2$ and fix one degree of freedom ($x_2 = 0$):

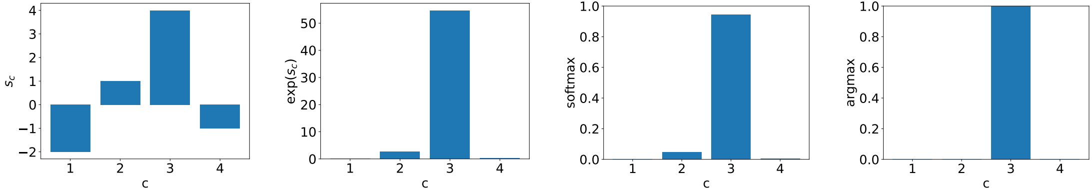
$$
\begin{aligned}
\text{softmax}(\mathbf{x}) &= \left(\frac{\exp(x_1)}{\exp(x_1) + \exp(x_2)}, \frac{\exp(x_2)}{\exp(x_1) + \exp(x_2)} \right) \\
&= \left(\frac{\exp(x_1)}{\exp(x_1) + 1}, \frac{1}{\exp(x_1) + 1} \right) \\
&= \left(\frac{1}{1 + \exp(-x_1)}, 1 - \frac{1}{1 + \exp(-x_1)} \right) \\
&= (\sigma(x_1), 1 - \sigma(x_1))
\end{aligned}
$$

- The softmax is a **multi-class generalization** of the sigmoid function
- In practice, the overparameterized version is often used (simpler to implement)

Softmax

$$\text{softmax}(\mathbf{s}) = \left(\frac{\exp(s_1)}{\sum_{k=1}^{C} \exp(s_k)}, \ldots, \frac{\exp(s_C)}{\sum_{k=1}^{C} \exp(s_k)} \right)$$

- The name “softmax” is confusing, “soft argmax” would be more precise as it is a continuous and differentiable version of argmax (in one-hot representation)
- Example with four classes:



Softmax

- Softmax responds to differences between inputs
- It is invariant to adding the same scalar to all its inputs:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} + c)$$

- We can therefore derive a numerically more stable variant:

$$\mathrm{softmax}(\mathbf{x}) = \mathrm{softmax}(\mathbf{x} - \max_{k=1..L} x_k)$$

- Allows accurate computation even when $\mathbf{x}$ is large
- Illustrates again that softmax depends on differences between scores

Cross Entropy Loss with Softmax Example

Putting it together: Cross Entropy Loss for a single training sample $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}$:

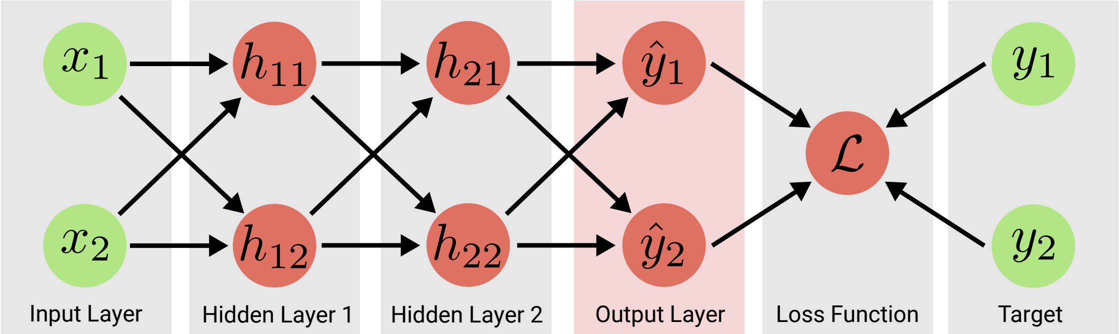
$$\text{CE Loss:} \quad \sum_{c=1}^{C} -y_c \log f_{\mathbf{w}}^{(c)}(\mathbf{x})$$

Example: Suppose $C = 4$ and 4 training samples $\mathbf{x}$ with labels $\mathbf{y}$

Input $\mathbf{x}$	Label $\mathbf{y}$	Predicted scores $\mathbf{s}$	softmax($\mathbf{s}$)	CE Loss
	$(1, 0, 0, 0)^\top$	$(+3, +1, -1, -1)^\top$	$(0.85, 0.12, 0.02, 0.02)^\top$	0.16
	$(0, 1, 0, 0)^\top$	$(+3, +3, +1, +0)^\top$	$(0.46, 0.46, 0.06, 0.02)^\top$	0.78
	$(0, 0, 1, 0)^\top$	$(+1, +1, +1, +1)^\top$	$(0.25, 0.25, 0.25, 0.25)^\top$	1.38
	$(0, 0, 0, 1)^\top$	$(+3, +2, +3, -1)^\top$	$(0.42, 0.16, 0.42, 0.01)^\top$	4.87

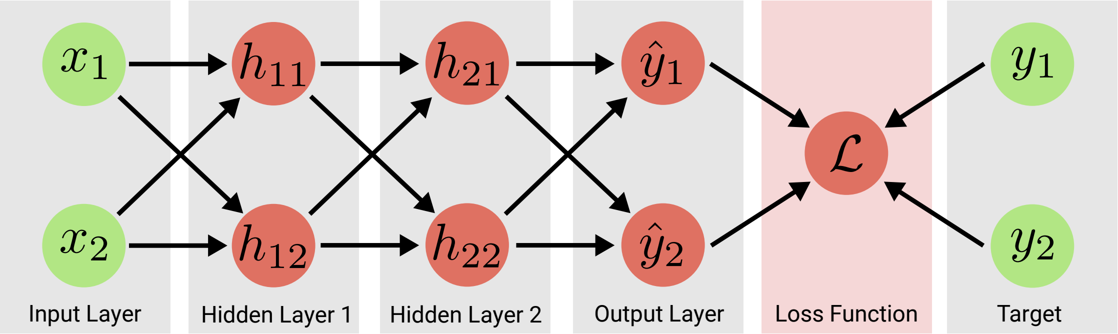
- Sample 4 contributes most strongly to the loss function! (elephant in the room)

Output Layer for Classification Problems



- For 2 classes, we can predict 1 value and use a sigmoid, or 2 values with softmax
- For $C > 2$ classes we typically predict C scores and use a softmax non-linearity

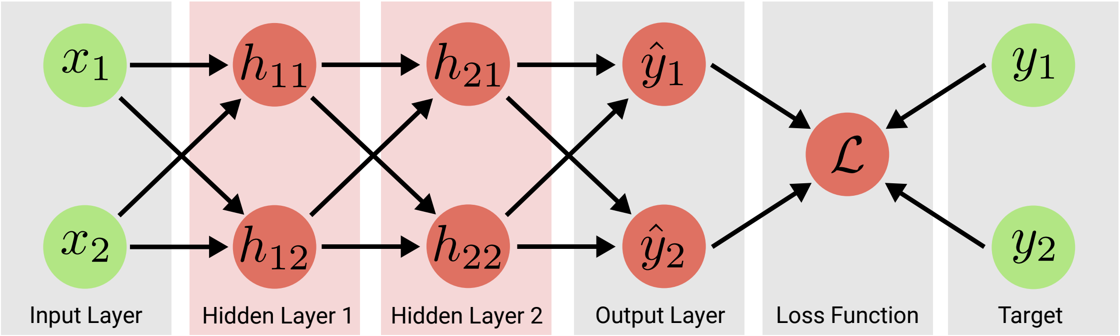
Loss Function for Classification Problems



- For 2 classes, we use the binary cross-entropy loss (BCE)
- For $C > 2$ classes, we use the cross-entropy loss (CE)

4.2
Activation Functions

Activation Functions



- Hidden layer $\mathbf{h}_i = g(\mathbf{A}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$
- The activation function is frequently applied **element-wise** to its input
- Activation functions must be **non-linear** to learn non-linear mappings
- Some of them are not differentiable everywhere (but still ok for training)

Activation Functions

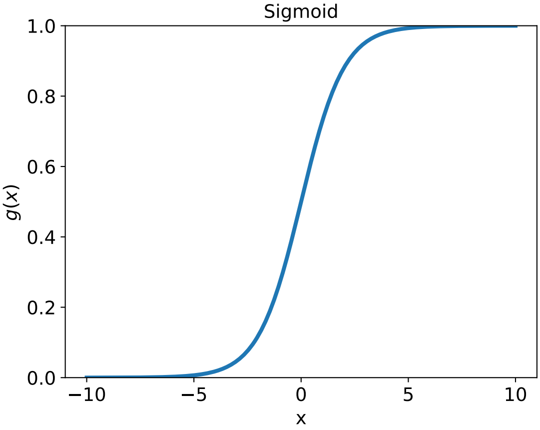
Sigmoid:

$$g(x) = \frac{1}{1 + \exp(-x)}$$

- Maps input to range $[0, 1]$
- Neuroscience interpretation as saturating "firing rate" of neurons

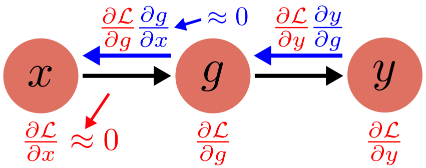
Problems:

- Saturation "kills" gradients
- Outputs are not zero-centered
- Introduces bias after first layer

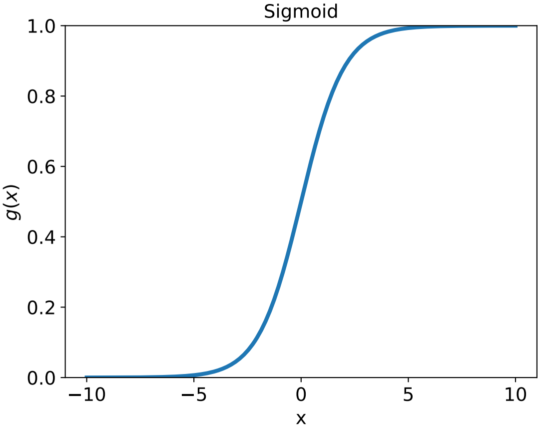


Activation Functions

Sigmoid Problem #1:



- Downstream gradient becomes zero when input x is saturated: $g'(x) \approx 0$
- No learning if x is very small (< -10)
- No learning if x is very large (> 10)



Sigmoid Problem #2:

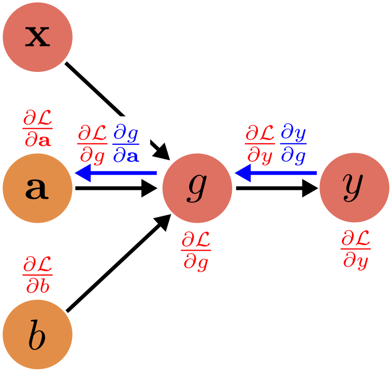
$$g(x) = \frac{1}{1+\exp(-x)} \qquad x = \sum_i a_i x_i + b$$

- Sigmoid is always positive $\Rightarrow$ x_i also
- Gradient of sigmoid is always positive

The gradient wrt. parameter a_i is given by:

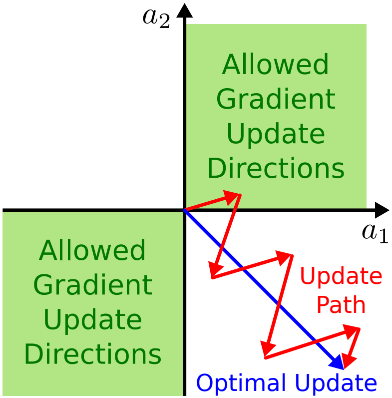
$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}\frac{\partial x}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i$$

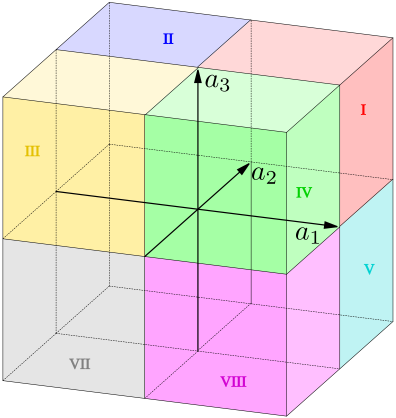
- Therefore: $\text{sgn}(\partial \mathcal{L}/\partial a_i) = \text{sgn}(\partial \mathcal{L}/\partial g)$
- All gradients have the same sign (+ or -)



Activation Functions

Sigmoid Problem #2:





- Restricts gradient updates and leads to inefficient optimization (minibatches help)

Activation Functions

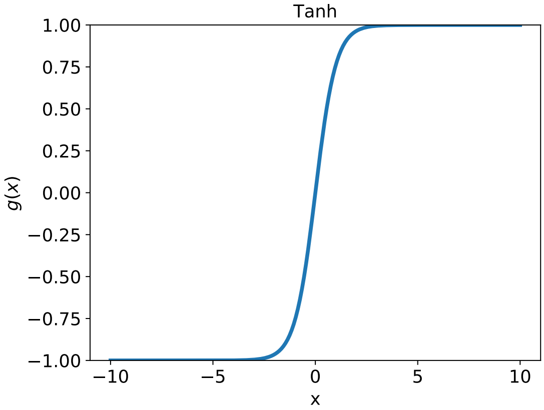
Tanh:

$$g(x) = \frac{2}{1+\exp(-2x)} - 1$$

- Maps input to range $[-1, 1]$
- Anti-symmetric
- Zero-centered

Problems:

- Again, saturation "kills" gradients



LeCun, Kanter and Solla: Second-order properties of error surfaces: learning time and generalization. NIPS, 1991.

Activation Functions

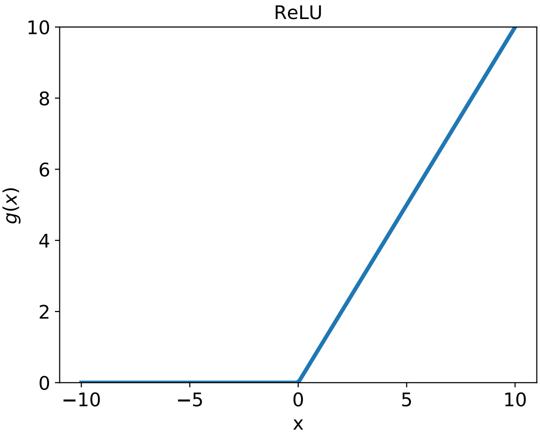
Rectified Linear Unit (ReLU):

$$g(x) = \max(0, x)$$

- Does not saturate (for $x > 0$)
- Leads to fast convergence
- Computationally efficient

Problems:

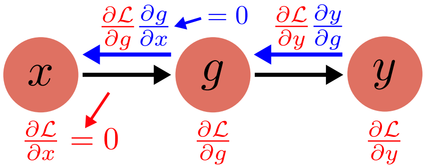
- Not zero-centered
- No learning for $x < 0 \Rightarrow$ dead ReLUs



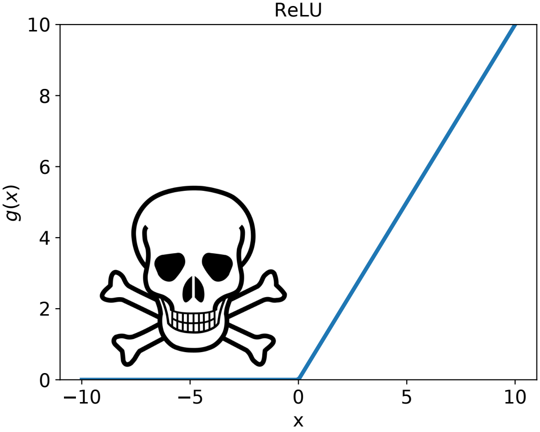
Nair and Hinton: Rectified linear units improve restricted boltzmann machines. ICML, 2010.

Activation Functions

ReLU Problem:



- Downstream gradient becomes zero when input $x < 0$
- Results in so-called "dead ReLUs" that never participate in learning
- Often initialize with pos. bias $(b > 0)$



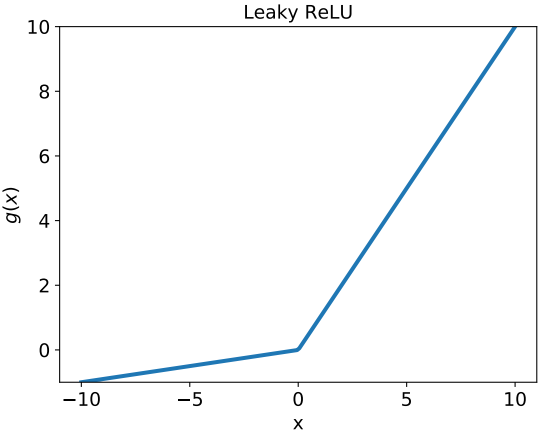
Nair and Hinton: Rectified linear units improve restricted boltzmann machines. ICML, 2010.

Activation Functions

Leaky ReLU:

$$g(x) = \max(0.01x, x)$$

- Does not saturate (i.e., will not die)
- Closer to zero-centered outputs
- Leads to fast convergence
- Computationally efficient



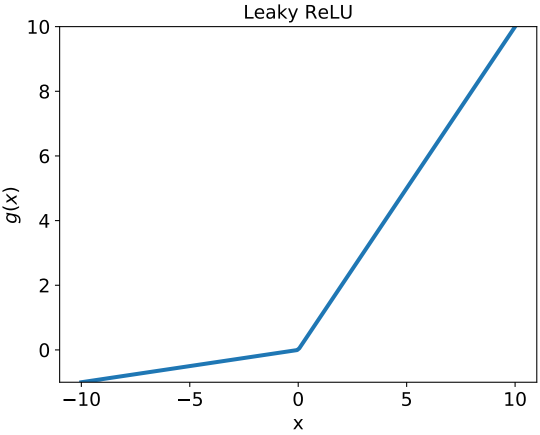
Maas: Rectifier nonlinearities improve neural network acoustic models. ICML, 2013.

Activation Functions

Parametric ReLU:

$$g(x) = \max(\alpha x, x)$$

- Does not saturate (i.e., will not die)
- Leads to fast convergence
- Computationally efficient
- Parameter α learned from data



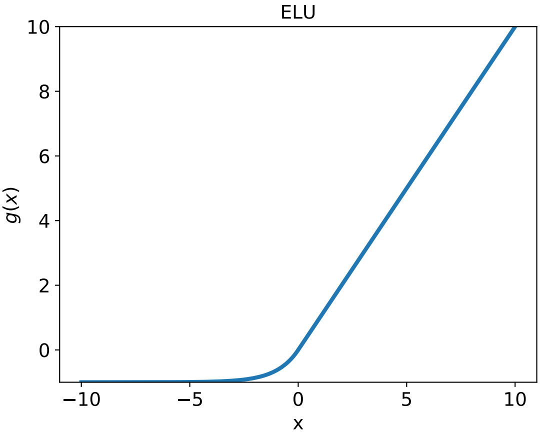
He, Zhang, Ren and Sun: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. ICCV, 2015.

Activation Functions

Exponential Linear Units (ELU):

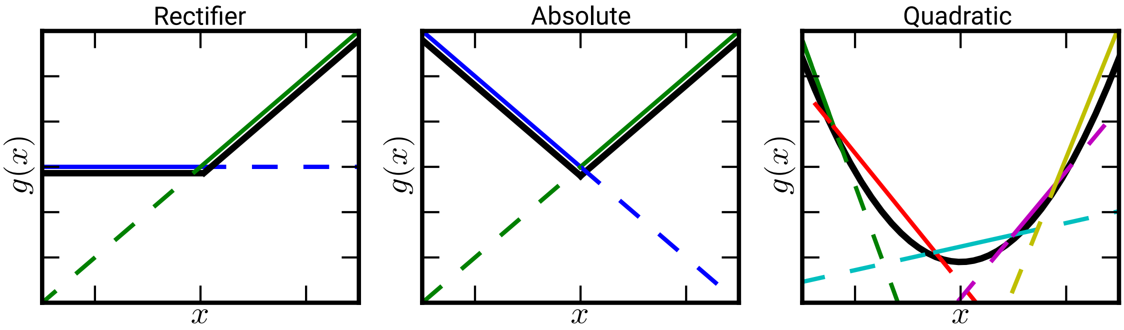
$$g(x) = \begin{cases} x & \text{if } \; x > 0 \\ \alpha(\exp(x) - 1) & \text{if } \; x \leq 0 \end{cases}$$

- All benefits of Leaky ReLU
- Adds some robustness to noise
- Default $\alpha = 1$



Clevert, Unterthiner and Hochreiter: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). ICLR, 2016.

Activation Functions



Maxout: $g(x) = \max(\mathbf{a}_1^\top \mathbf{x} + b_1, \mathbf{a}_2^\top \mathbf{x} + b_2)$

- Generalizes ReLU and Leaky ReLU
- Increases the number of parameters per neuron

Goodfellow, Warde-Farley, Mirza, Courville and Yoshua Bengio: Maxout Networks. ICML, 2013.

Activation Functions

Summary:

- No one-size-fits-all: Choice of activation function depends on problem
- We only showed the most common ones, there exist many more
- Best activation function/model is often found using trial-and-error in practice
- It is important to ensure a good "gradient flow" during optimization

Rule of Thumb:

- Use ReLU by default (with small enough learning rate)
- Try Leaky ReLU, Maxout, ELU for some small additional gain
- Prefer Tanh over Sigmoid (Tanh often used in recurrent models)

Implementation

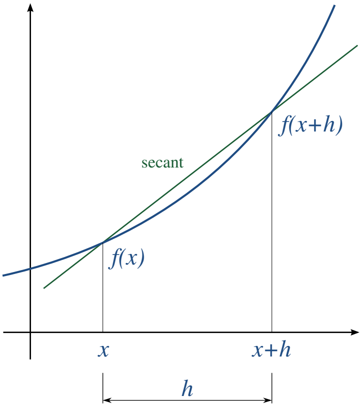
Numerical Differentiation

- Murphy: "Anything that can go wrong will."
- When implementing the backward pass of activation, output or loss functions it is important to ensure that all gradients are correct!
- Verify via Newton's difference quotient:

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Even better: Symmetric difference quotient

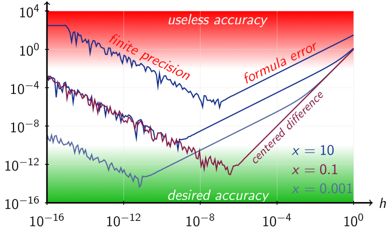
$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$



Numerical Differentiation

How to choose h?

- For $h = 0$ the expression is undefined
- Choose h to trade-off:
  - Rounding error (finite precision)
  - Approximation error (wrong)
- Good choice: $\sqrt[3]{\epsilon}$ with ϵ the machine precision
- Examples:
  - $\epsilon = 6 \times 10^{-8}$ for single precision (32 bit)
  - $\epsilon = 1 \times 10^{-16}$ for double precision (64 bit)

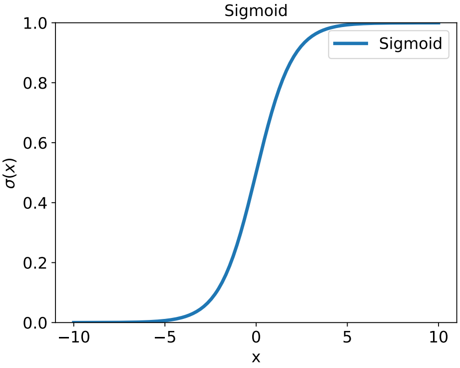


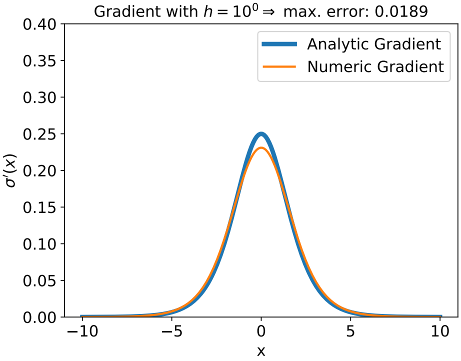
en.wikipedia.org/wiki/Numerical_differentiation

Numerical Differentiation

Example: Sigmoid derivative using symmetric differences with single precision:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

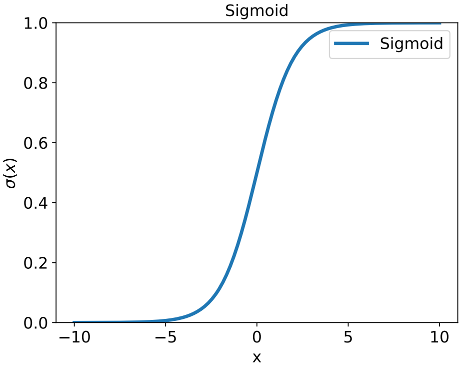


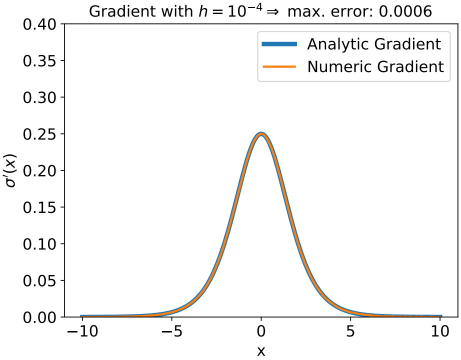


Numerical Differentiation

Example: Sigmoid derivative using symmetric differences with single precision:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

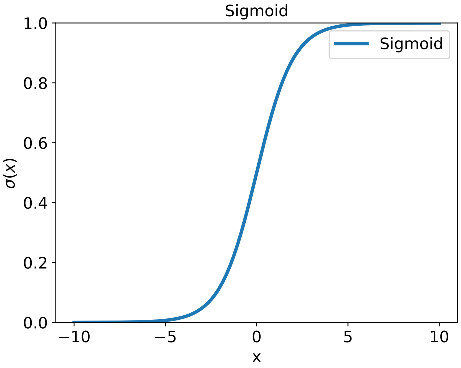


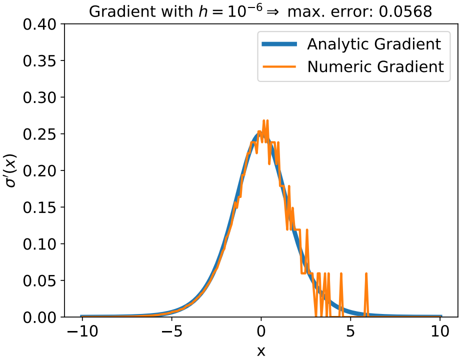


Numerical Differentiation

Example: Sigmoid derivative using symmetric differences with single precision:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$





4.3
Preprocessing and Initialization

Data Preprocessing

Data Preprocessing

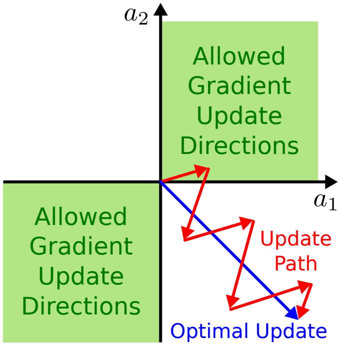
Remember what happens for positive inputs:

$$g(x) = g\left(\sum_i a_i x_i + b\right)$$

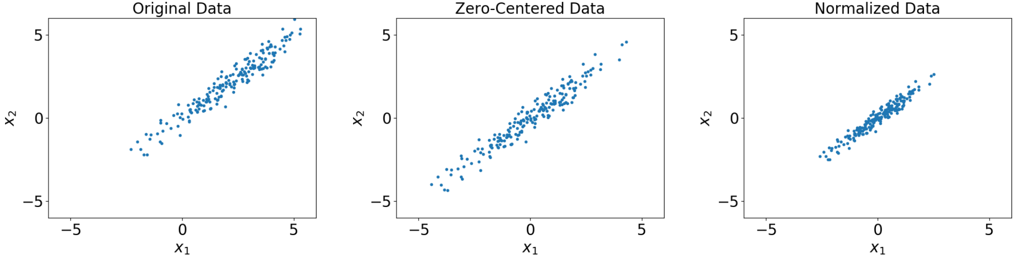
The gradient wrt. parameter a_i is given by:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial x} x_i$$

- Both terms in blue are positive
- All gradients have the same sign (+ or -)
- We should pre-process the input data such that it is "well distributed"

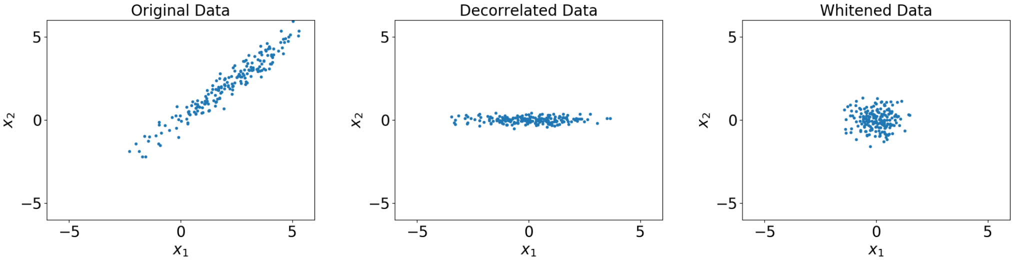


Data Preprocessing



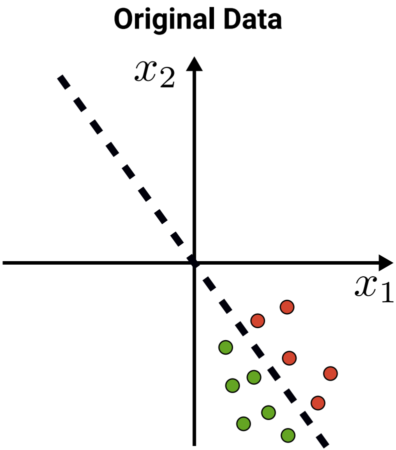
- **Zero-center:** $x_{i,j} \leftarrow x_{i,j} - \mu_j$ with $\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$
- **Normalization:** $x_{i,j} \leftarrow x_{i,j}/\sigma_j$ with $\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$

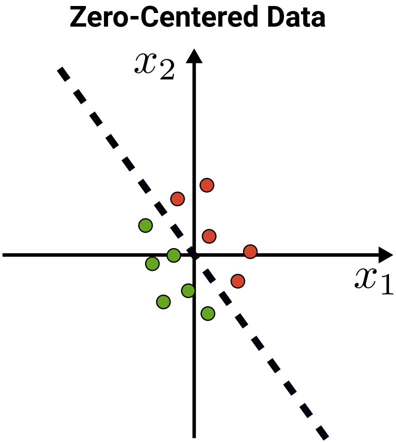
Data Preprocessing



- **Decorrelate:** Multiply with eigenvectors of covariance matrix
- **Whiten:** Divide by square root of eigenvalues of covariance matrix

Data Preprocessing





- Classification loss becomes less sensitive to changes in the weight matrix

Data Preprocessing

Common Practices for Images:

- AlexNet: Subtract mean image
  (mean image: $W \times H \times 3$ numbers)
- VGGNet: Subtract per-channel mean
  (mean along each channel: 3 numbers)
- ResNet: Subtract per-channel mean and divide by per-channel std. dev.
  (mean along each channel: 3 numbers)
- Whitening is less common

Weight Initialization

Recap: Stochastic Gradient Descent

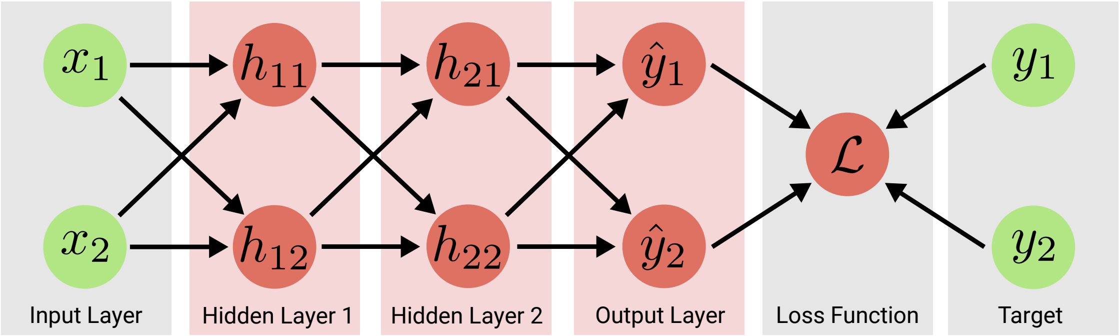
Algorithm for training an MLP using (stochastic) gradient descent:

1. Initialize weights $\mathbf{w}$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   3.1 Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   3.2 Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

Question:

- How to best initialize the weights $\mathbf{w}$?

Constant Initialization



- How to initialize the parameters $\mathbf{w}$ of all network layers?
- Simple solution: set all network parameters to a constant (i.e., $\mathbf{w} = 0$)
- Learning not be possible (all units of each layer are learning the same)

Weight Initialization

Consider a layer in a Multi-Layer Perceptron:

$$g(x) = g\left(\sum_i a_i x_i + b\right)$$

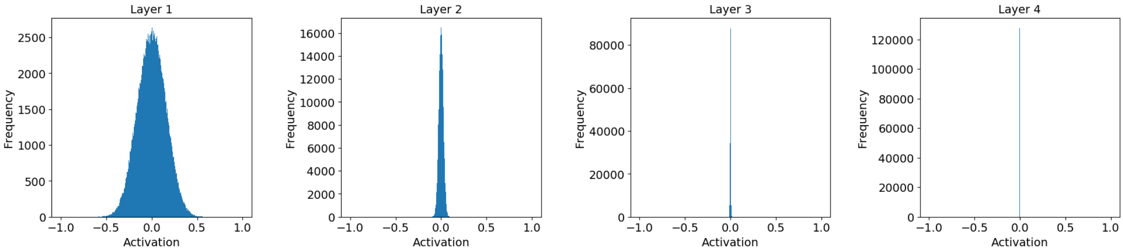
The gradient wrt. parameter a_i is given by:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial x} x_i$$

Remark:

- For $g(\cdot)$, we will use Tanh and ReLU in the following

Small Random Numbers

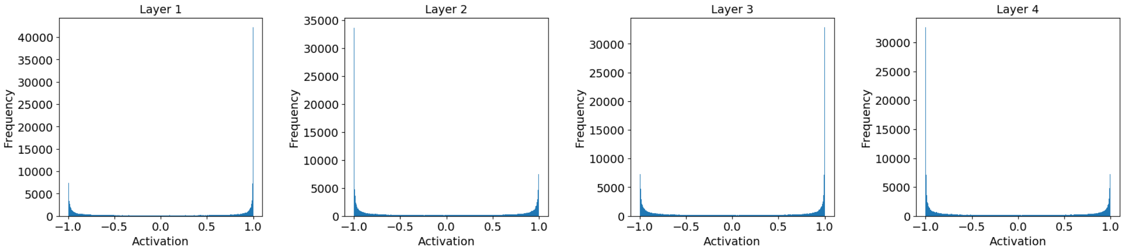


Tanh Activation Function:

- Draw weights independently from Gaussian with small std. dev ($\sigma = 0.01$)
- Activations (=activation function outputs) in deeper layers tend towards zero
- Gradients wrt. weights thus also tend towards zero $\Rightarrow$ no learning:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial x} x_i = \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial x} 0 = 0$$

Large Random Numbers

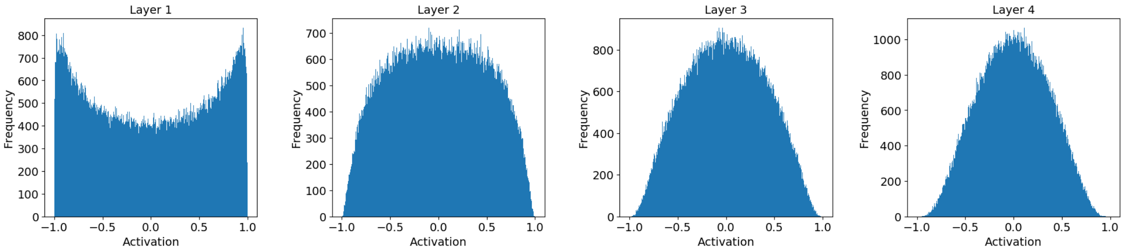


Tanh Activation Function:

- Draw weights independently from Gaussian with large std. dev ($\sigma = 0.2$)
- All activation functions saturate
- Local gradients are all becoming zero $\Rightarrow$ no learning:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g} \frac{\partial g}{\partial x} x_i = \frac{\partial \mathcal{L}}{\partial g} \, 0 \, x_i = 0$$

Xavier Initialization



Tanh Activation Function:

- Glorot et al. draw weights independently from Gaussian with $\sigma^2 = 1/D_{in}$
- D_{in} denotes the dimension of the input to the layer, may vary across layers
- Activation distribution now well scaled across all layers

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

Xavier Initialization

Why $\sigma = 1/\sqrt{D_{in}}$? Let us consider $y = g(\mathbf{w}^\top \mathbf{x})$ and assume that all x_i and w_i are independent and identically (i.i.d.) distributed with zero mean. Let further $g'(0) = 1$. Then:

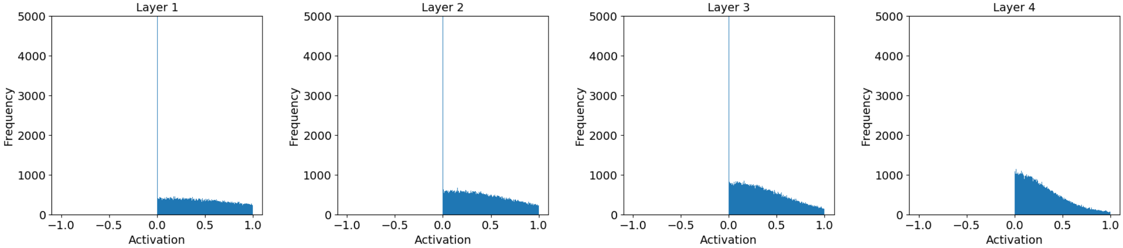
$$\begin{aligned}
\mathrm{Var}(y) \approx \mathrm{Var}(\mathbf{w}^\top \mathbf{x}) &= D_{in}\,\mathrm{Var}(x_i\, w_i) \\
&= D_{in}(\mathbb{E}[x_i^2 w_i^2] - \mathbb{E}[x_i w_i]^2) \\
&= D_{in}(\mathbb{E}[x_i^2]\,\mathbb{E}[w_i^2] - \mathbb{E}[x_i]^2\,\mathbb{E}[w_i]^2) \\
&= D_{in}\mathbb{E}[x_i^2]\,\mathbb{E}[w_i^2] \\
&= D_{in}\,\mathrm{Var}(x_i)\,\mathrm{Var}(w_i)
\end{aligned}$$

Thus:

$$\mathrm{Var}(w_i) = 1/D_{in} \Rightarrow \mathrm{Var}(y) = \mathrm{Var}(x_i)$$

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

Xavier Initialization

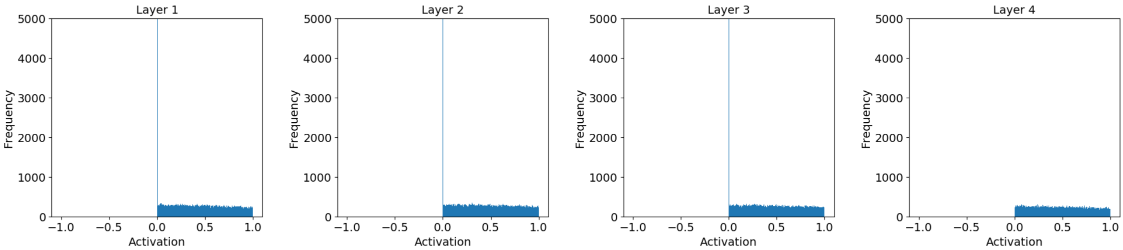


ReLU Activation Function:

- Xavier initialization assumes zero centered activation function
- For ReLU, activations again start collapsing to zero for deeper layers

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

He Initialization



ReLU Activation Function:

- Since ReLU is restricted to positive outputs, variance must be doubled
- He et al. draw weights independently from Gaussian with $\sigma^2 = 2/D_{in}$
- Activation distribution now well scaled across all layers

He, Zhang, Ren and Sun: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. ICCV, 2015.

Summary

Data Preprocessing:

- Zero-centering the network inputs is important for efficient learning
- Decorrelation and whitening used less frequently

Weight Initialization:

- Proper initialization important for ensuring a good “gradient flow”
- For zero-centered activation functions, use Xavier initialization
- For ReLU activation functions, use He initialization
- Initialization is a research topic, much more literature on this topic