FOUNDATIONS OF NEURAL NETWORKS

OSCAR LLORENTE GONZALEZ

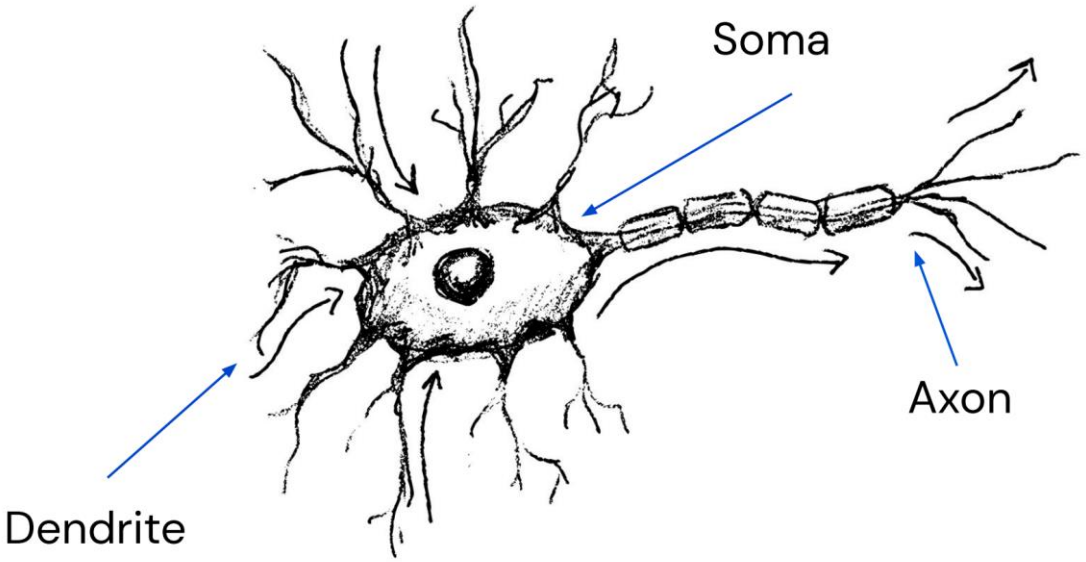
OLLORENTE@COMILLAS.EDU

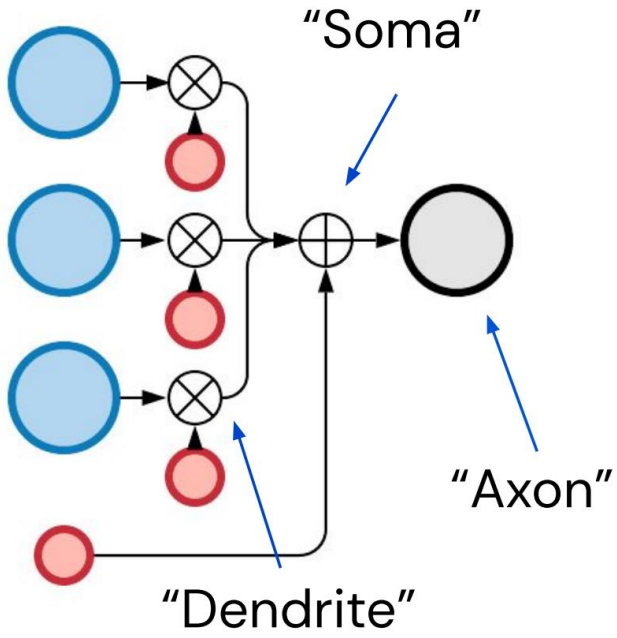
INDEX

- Introduction
- Feedforward Layer
- Regression and Classification
- Loss Functions
- Non Linearities
- Initialization
- Optimization

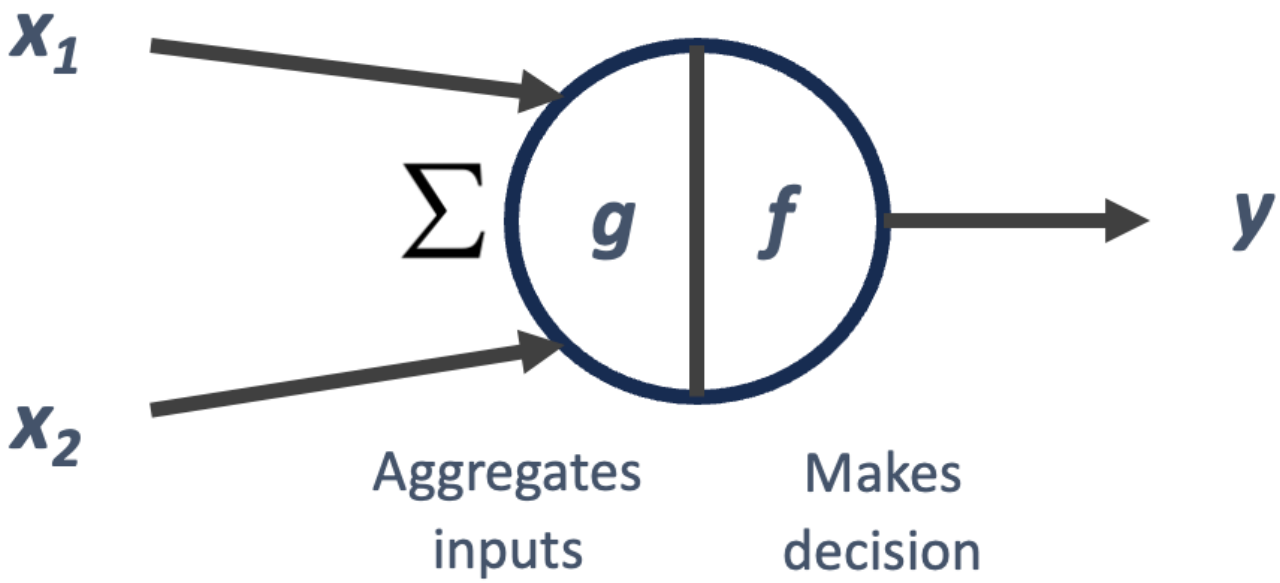
1 - INTRODUCTION

REAL VS ARTIFICIAL NEURON

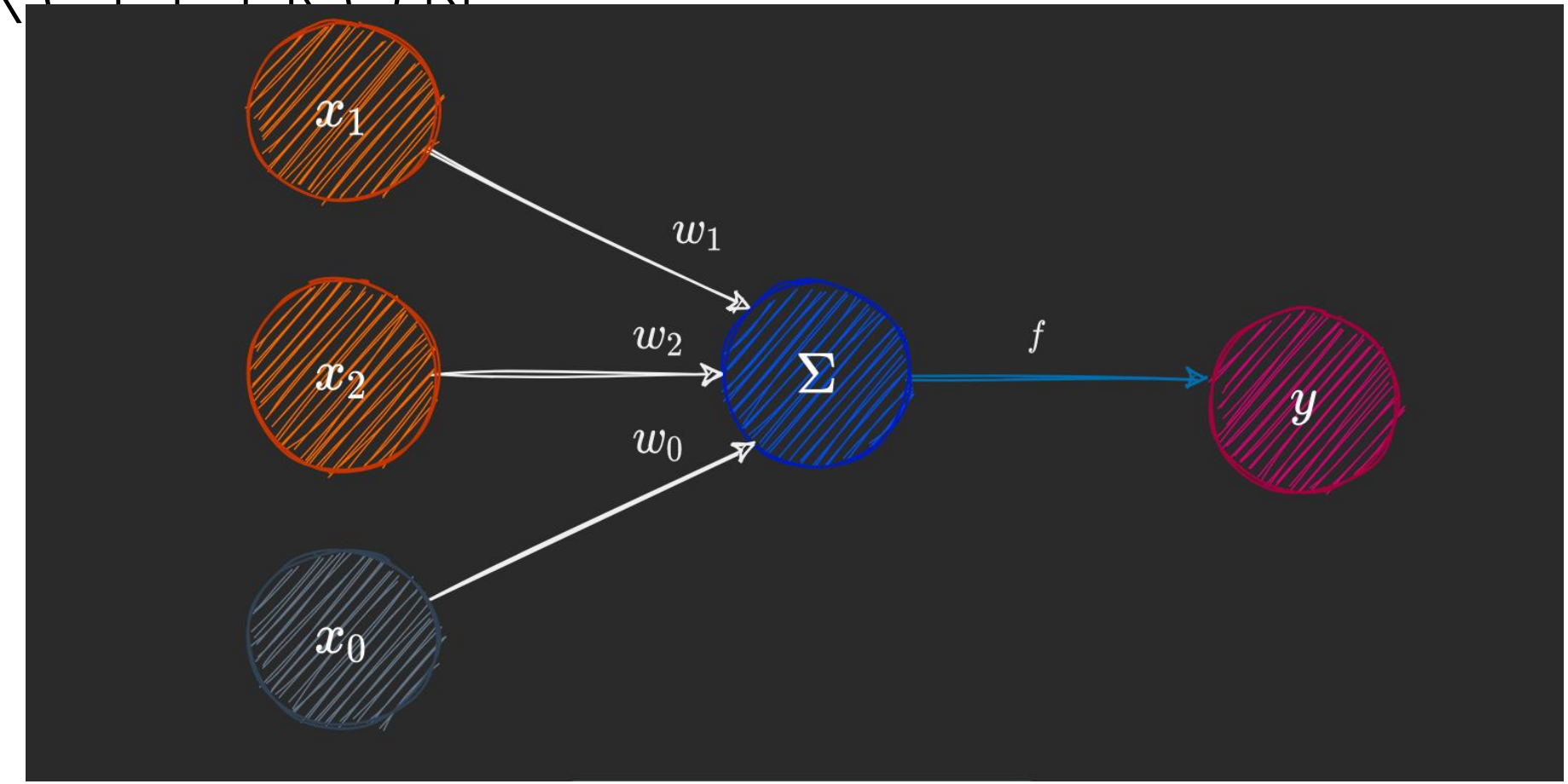




ARTIFICIAL NEURON



PERCEPTRON



PERCEPTRON EQUATIONS

$$y = f(x_1 \cdot w_1 + x_2 \cdot w_2 + b)$$

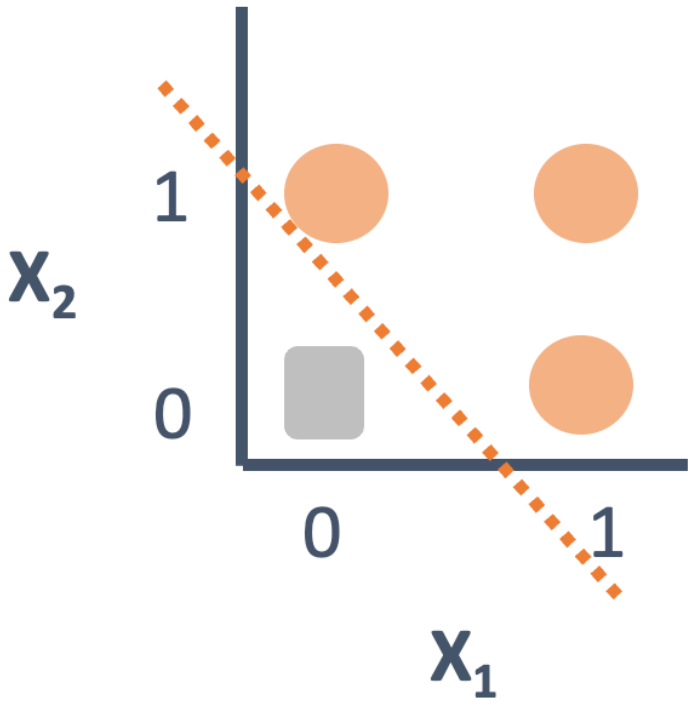
$$y = f(w \cdot X + b)$$

$$Class_1 : w \cdot X + b \geq 0$$

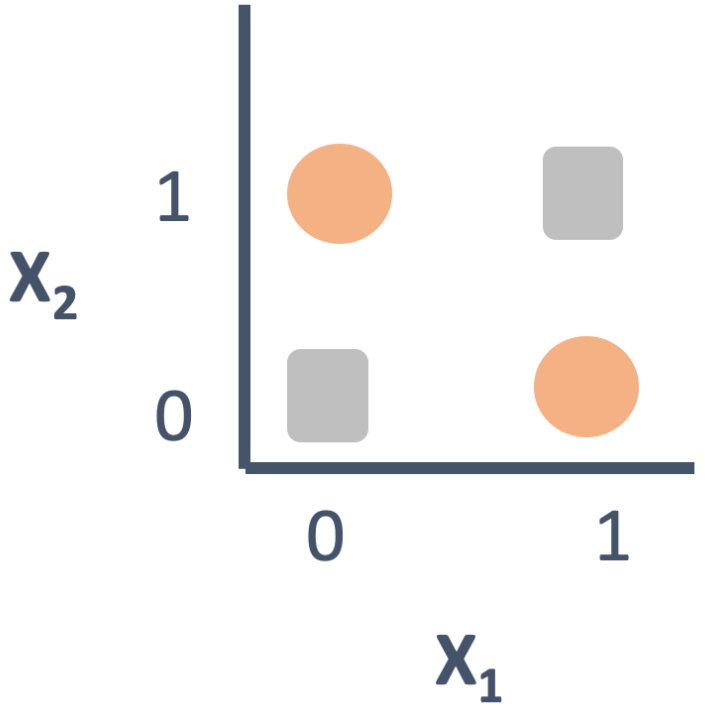
$$Class_2 : w \cdot X + b < 0$$

FIRST AI WINTER: PERCEPTRON PROBLEM

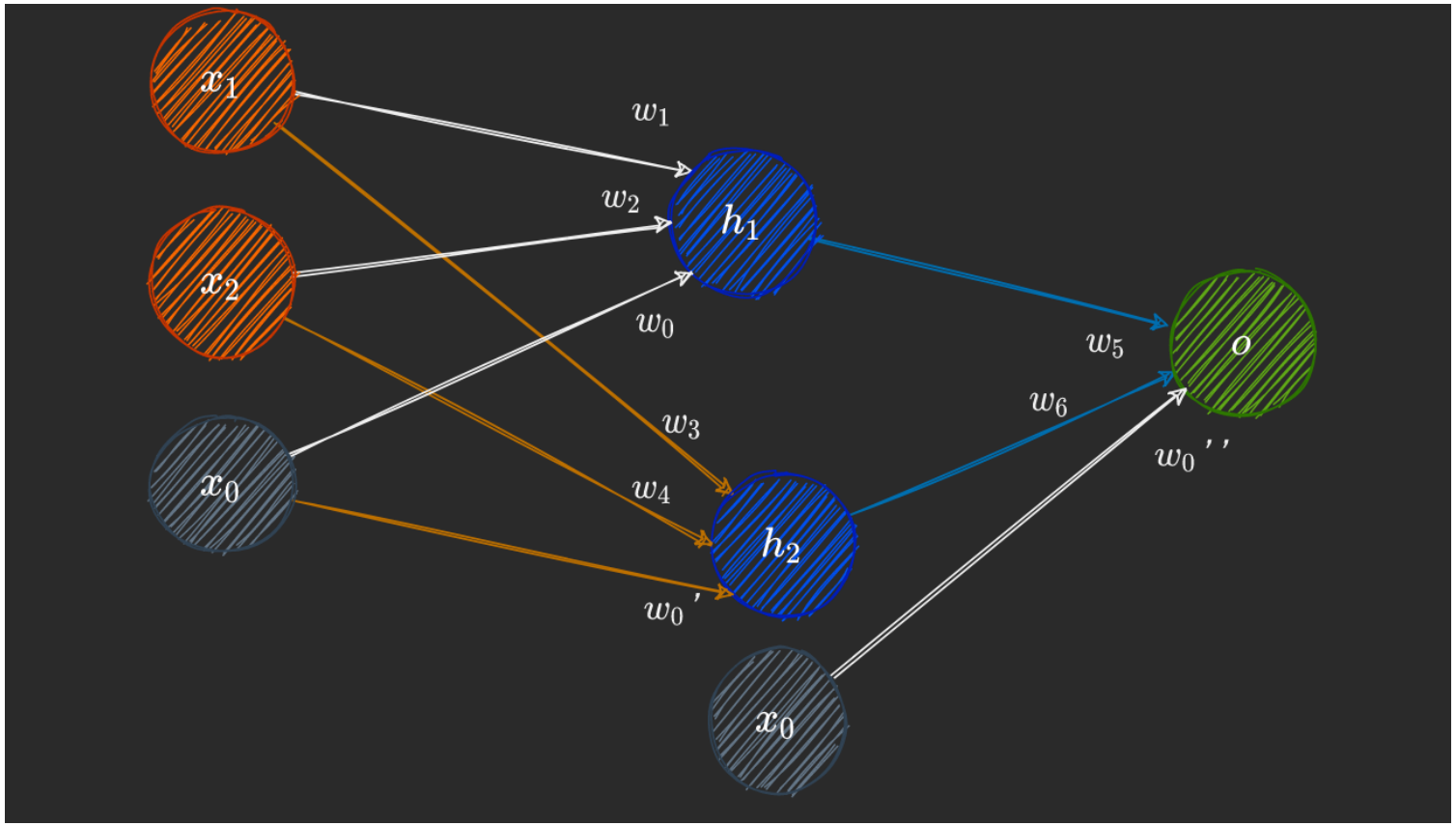
AND



XOR



SOLUTION TO NON-LINEAR PROBLEMS



PERCEPTRON LEARNING ALGORITHM

Let

$$\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$$

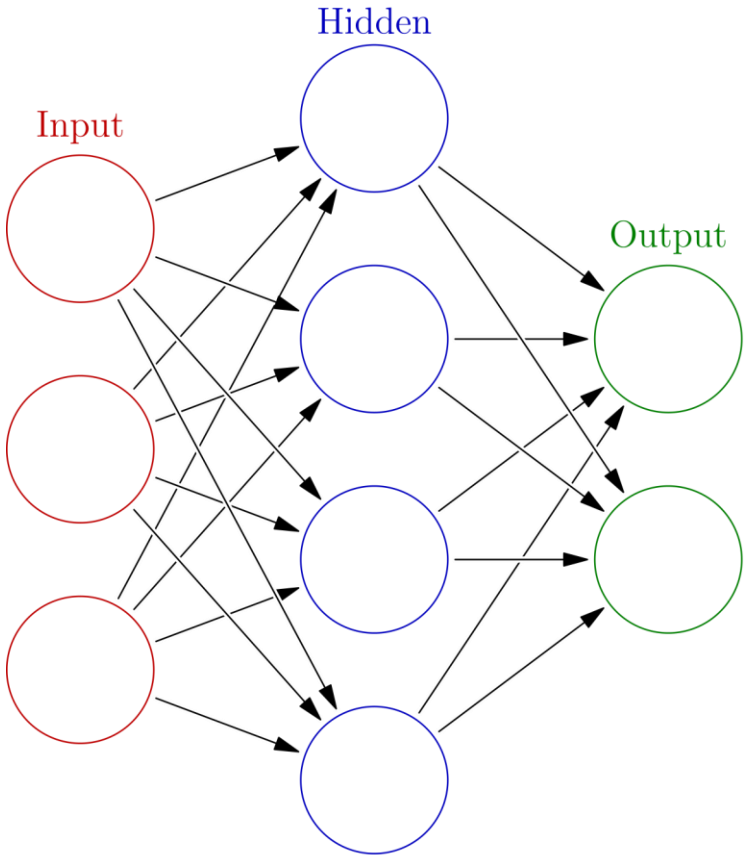
1. Initialize $\mathbf{w} := 0^m$ (assume notation where weight incl. bias)
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
      (a) $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]\top} \mathbf{w}\right)$
      (b) $\text{err} := (y^{[i]} - \hat{y}^{[i]})$
      (c) $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

SECOND A

2 - FEEDFORWARD LAYER

WHAT IS A FEEDFORWARD LAYER?

$$\underbrace{\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix}}_{z \in \mathbb{R}^{m \times 1}} = \underbrace{\begin{bmatrix} — & w_1^{[1]\top} & — \\ — & w_2^{[1]\top} & — \\ & \vdots & \\ — & w_m^{[1]\top} & — \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{m \times d}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}}_{x \in \mathbb{R}^{d \times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{m \times 1}}$$



FEEDFORWARD LAYER – BATCH AS INPUT

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]} \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & \dots & w_{h,m} \end{bmatrix}$$

$$Y = (WX^T + b)^T$$

$$Y = XW^T + b$$

FEEDFORWARD LAYER - IMPLEMENTATION

```python
class MyModel(torch.nn.Module):

    def __init__(self, input_dim, output_dim) -> None:

        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:

        outputs = self.linear(inputs)

        return outputs
```

LINEAR

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Examples:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

3 - REGRESSION & CLASSIFICATION

LINEAR REGRESSION - EXAMPLE

- $1,200,000

- 3200 sq. ft.
- 5 beds
- 3 baths

LINEAR REGRESSION - PROCESS

Input: $\mathbf{x}$ (tensor)

Output: $y \in \mathbb{R}$

Parameters: $\mathbf{w}, b$

Regression: $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$

Loss: $\ell(\hat{y}, y) = |\hat{y} - y|$

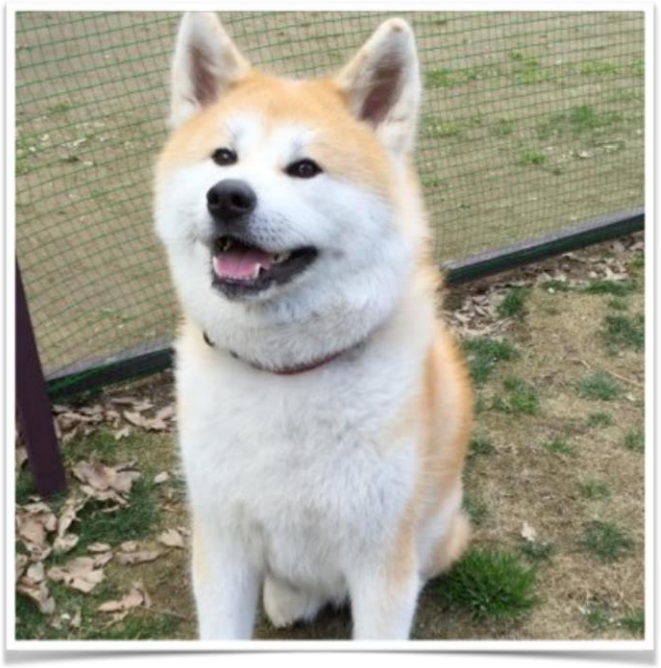
or $\ell(\hat{y}, y) = (\hat{y} - y)^2$

LINEAR REGRESSION - OPTIMIZATION

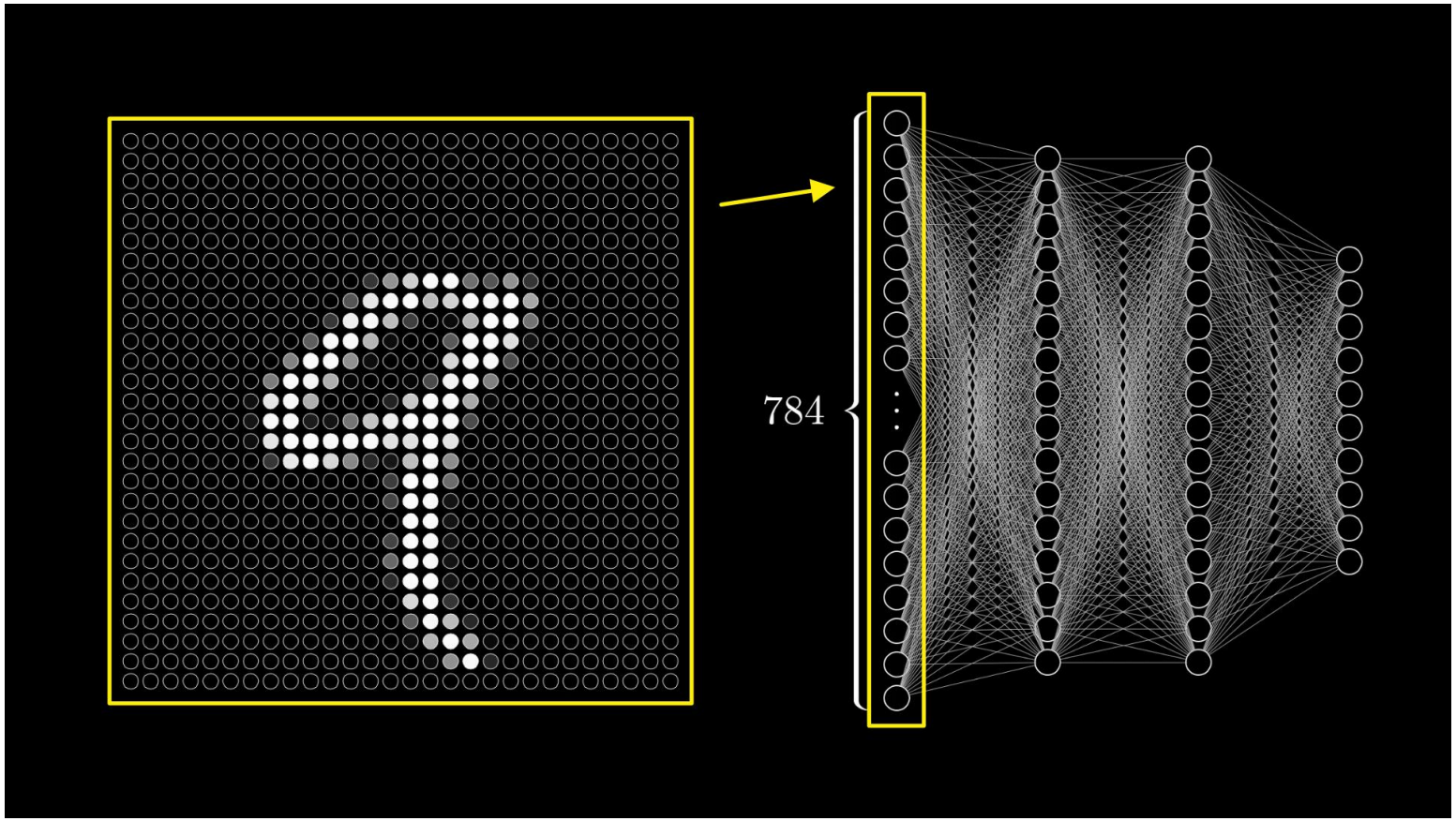
Loss: $\ell(\hat{y}, y) = (\hat{y} - y)^2 = (\mathbf{w}^\top \mathbf{x} + b - y)^2$

BINARY CLASSIFICATION - EXAMPLE

VS



FLATTEN INPUT



FLATTEN INPUT

FLATTEN

CLASS torch.nn.Flatten(*start_dim=1, end_dim=-1*) [SOURCE]

Flattens a contiguous range of dims into a tensor. For use with `Sequential`. See `torch.flatten()` for details.

Shape:

- Input: $(*, S_{\text{start}}, ..., S_i, ..., S_{\text{end}}, *)$, where S_i is the size at dimension i and $*$ means any number of dimensions including none.
- Output: $(*, \prod_{i=\text{start}}^{\text{end}} S_i, *)$.

Parameters

- **start_dim** (*int*) – first dim to flatten (default = 1).
- **end_dim** (*int*) – last dim to flatten (default = -1).

Examples::

```
>>> input = torch.randn(32, 1, 5, 5)
>>> # With default parameters
>>> m = nn.Flatten()
>>> output = m(input)
>>> output.size()
torch.Size([32, 25])
>>> # With non-default parameters
>>> m = nn.Flatten(0, 2)
>>> output = m(input)
>>> output.size()
torch.Size([160, 5])
```

TORCH.FLATTEN

torch.flatten(*input, start_dim=0, end_dim=-1*) → Tensor

Flattens `input` by reshaping it into a one-dimensional tensor. If `start_dim` or `end_dim` are passed, only dimensions starting with `start_dim` and ending with `end_dim` are flattened. The order of elements in `input` is unchanged.

Unlike NumPy's flatten, which always copies input's data, this function may return the original object, a view, or copy. If no dimensions are flattened, then the original object `input` is returned. Otherwise, if input can be viewed as the flattened shape, then that view is returned. Finally, only if the input cannot be viewed as the flattened shape is input's data copied. See `torch.Tensor.view()` for details on when a view will be returned.

NOTE

Flattening a zero-dimensional tensor will return a one-dimensional view.

Parameters

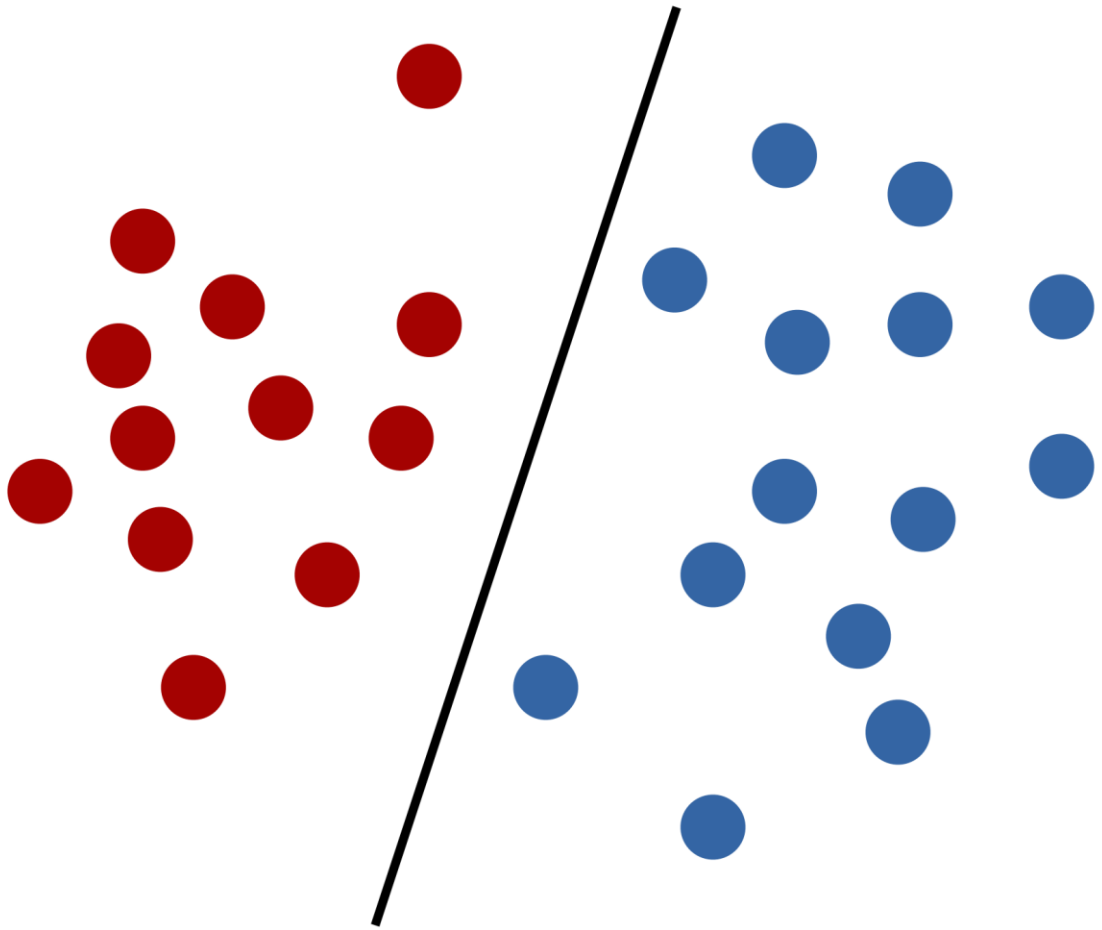
- **input** (*Tensor*) – the input tensor.
- **start_dim** (*int*) – the first dim to flatten
- **end_dim** (*int*) – the last dim to flatten

Example:

```
>>> t = torch.tensor([[[1, 2],
...                    [3, 4]],
...                   [[5, 6],
...                    [7, 8]]])
>>> torch.flatten(t)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> torch.flatten(t, start_dim=1)
tensor([[1, 2, 3, 4],
        [5, 6, 7, 8]])
```

```
mytensor = torch.rand(3, 3)
flatten_layer: torch.nn.Module = torch.nn.Flatten(0)
flatten_tensor: torch.Tensor = flatten_layer(mytensor)

flatten_tensor = torch.flatten(mytensor)
```

BINARY CLASSIFICATION - EXAMPLE



BINARY CLASSIFICATION - PROCESS

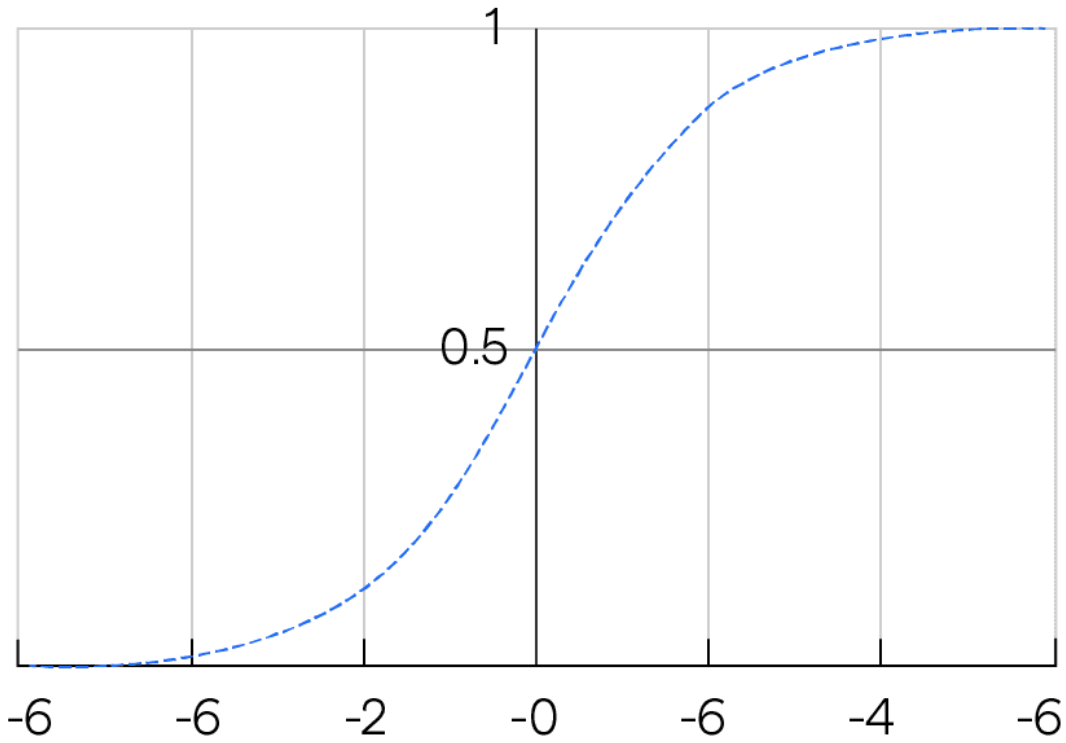
Input: $\mathbf{x}$ (tensor)

Label: $y \in \{0,1\}$

Parameters: $\mathbf{w}, b$

Prediction $\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$

SIGMOID



$$\sigma(o) = \frac{1}{1 + e^{-o}}$$

```
# torch.nn
sigmoid = torch.nn.Sigmoid()
outputs = sigmoid(inputs)

# torch.nn.functional
outputs = F.sigmoid(inputs)
```

LOGISTIC REGRESSION

Input: $\mathbf{x}$ (tensor)

Label: $y \in \{0,1\}$

Parameters: $\mathbf{w}, b$

$$o = \mathbf{w}^\top \mathbf{x} + b$$

$$p(y=1) = \sigma(o)$$
$$p(y=0) = 1 - \sigma(o)$$

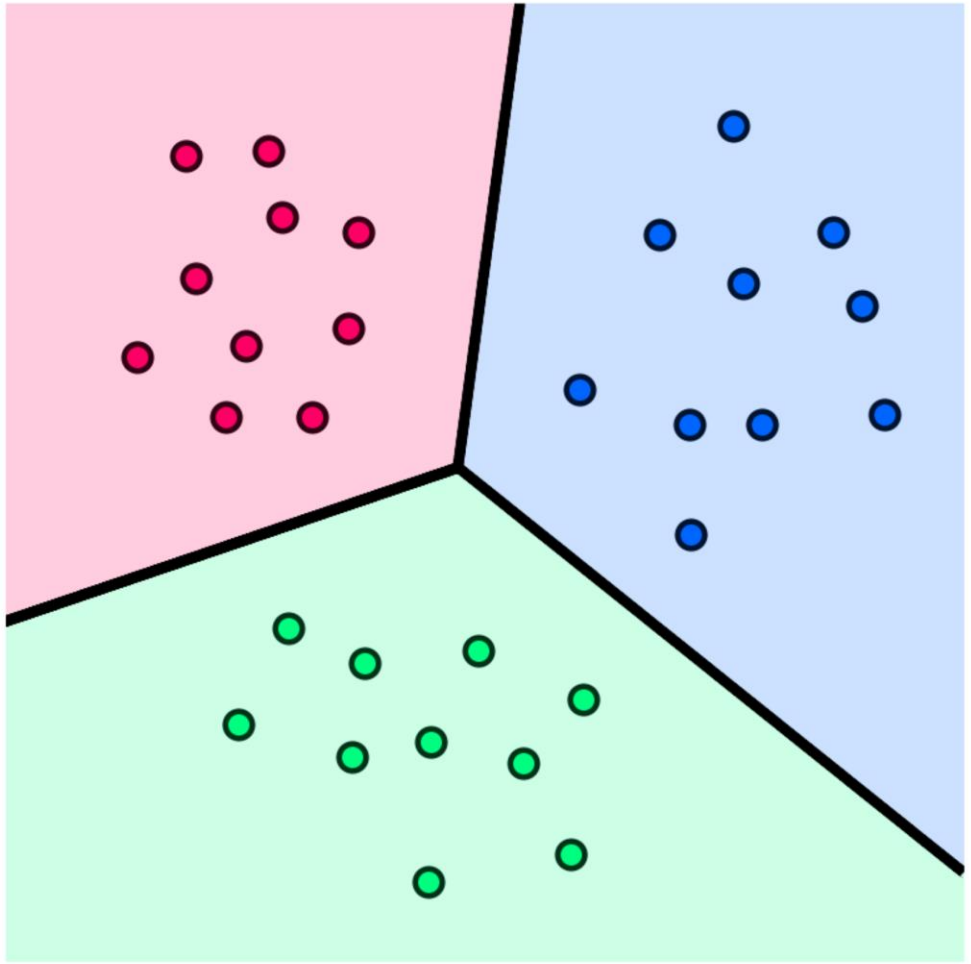
$$\Leftrightarrow \quad p(y) = \sigma(o)^y (1-\sigma(o))^{1-y}$$

Loss (negative log likelihood):

$$-\log p(y) = -y \log(\sigma(o)) - (1-y)\log(1-\sigma(o))$$

MULTICLASS CLASSIFICATION - EXAMPLE

MULTICLASS CLASSIFICATION - EXAMPLE



MULTICLASS CLASSIFICATION - PROCESS

Input: $\mathbf{x} \in \mathbb{R}^d$ (tensor)

Label: $y \in \{0,1,\ldots,k\}$

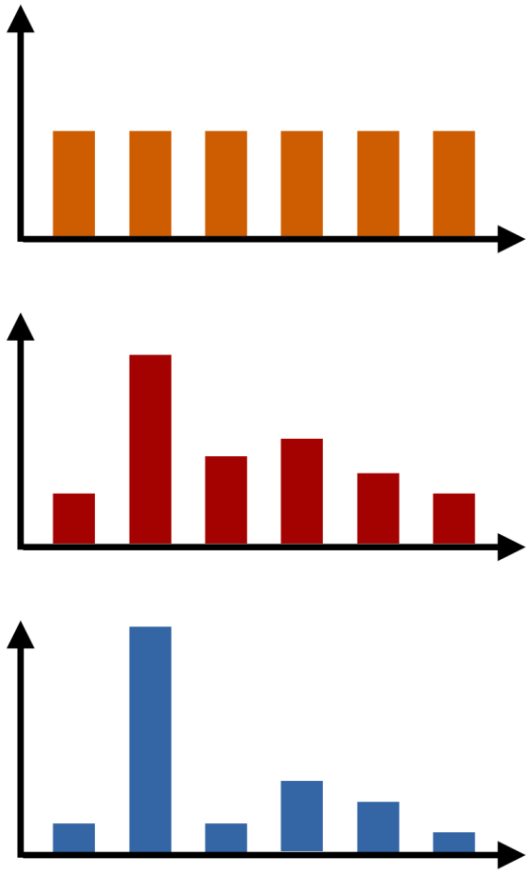
Parameters: $\mathbf{W} \in \mathbb{R}^{d \times k}, \mathbf{b} \in \mathbb{R}^k$

$$P(y) = \text{softmax}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})_y$$

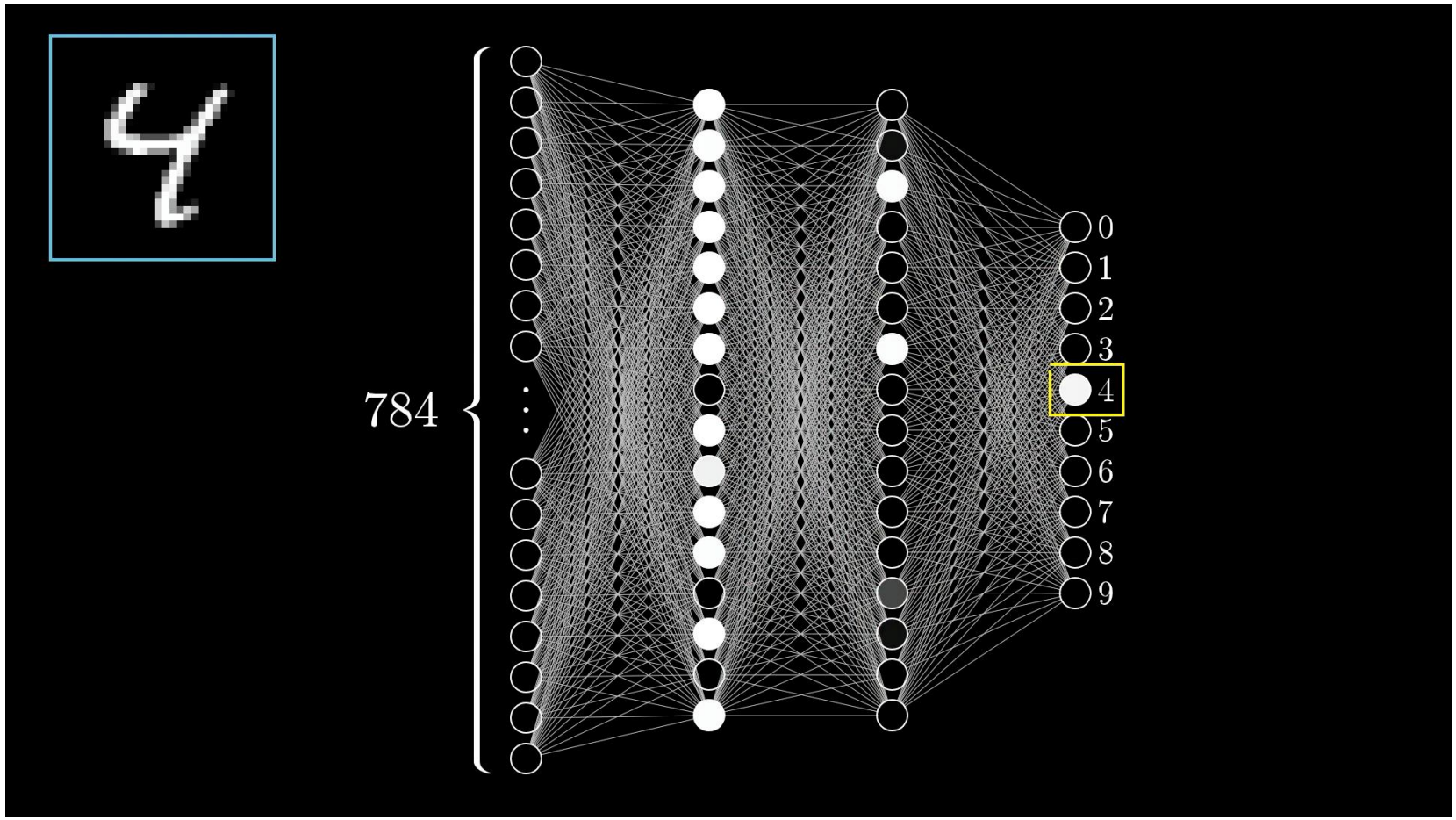
SOFTMAX

$$\text{softmax}(\mathbf{o})_i = \frac{e^{o_i}}{\sum_{i'} e^{o_{i'}}}$$

```
# torch.nn
softmax = torch.nn.Softmax(dim=1)
outputs = softmax(inputs)

# torch.nn.functional
outputs = F.softmax(inputs, dim=1)
```



EXAMPLE OUTPUT



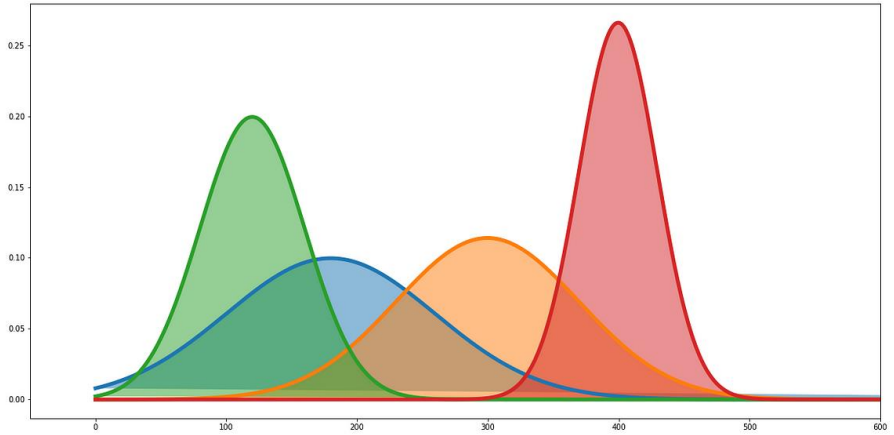
4 - LOSS FUNCTIONS

MAXIMUM LIKELIHOOD ESTIMATION

$$p_{\text{data}}(\mathbf{x})$$

$$\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$$

$$p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$$



MAXIMUM LIKELIHOOD ESTIMATION (MLE)

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\mathrm{model}}(\mathbb{X}; \boldsymbol{\theta}),$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

LOG LIKELIHOOD

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta}).$$

MLE AS KL-DIVERGENCE

$$D_{\mathrm{KL}}\left(\hat{p}_{\mathrm{data}} \| p_{\mathrm{model}}\right) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}}\left[\log \hat{p}_{\mathrm{data}}(\boldsymbol{x}) - \log p_{\mathrm{model}}(\boldsymbol{x})\right].$$

$$- \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}}\left[\log p_{\mathrm{model}}(\boldsymbol{x})\right],$$

CONDITIONAL LOG-LIKELIHOOD

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}).$$

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

PROPERTIES MAXIMUM LIKELIHOOD ESTIMATION

CONSISTENCY

STATISTICAL EFFIENCIENCY

CONDITIONS FOR CONSISTENCY

- The true distribution p_{data} must lie within the model family $p_{\text{model}}(\cdot; \boldsymbol{\theta})$. Otherwise, no estimator can recover p_{data}.
- The true distribution p_{data} must correspond to exactly one value of $\boldsymbol{\theta}$. Otherwise, maximum likelihood can recover the correct p_{data} but will not be able to determine which value of $\boldsymbol{\theta}$ was used by the data-generating process.

MLE-MSE

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{y}(\boldsymbol{x}; \boldsymbol{w}), \sigma^2)$$

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\left\| \hat{y}^{(i)} - y^{(i)} \right\|^2}{2\sigma^2},$$

$$\text{MSE}_{\text{train}} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2 ,$$

MLE-SIGMOID

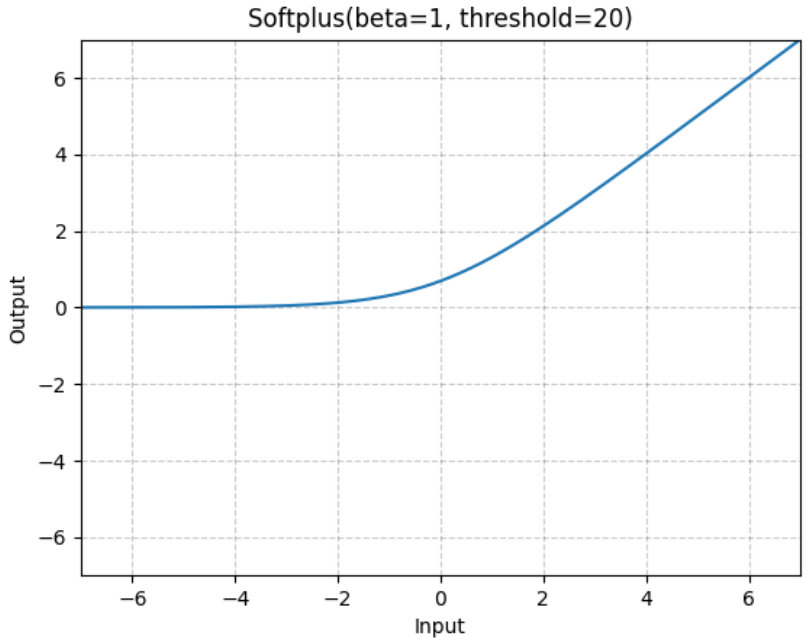
$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right),$$

$$\begin{aligned}
\log \tilde{P}(y) &= yz, \\
\tilde{P}(y) &= \exp(yz), \\
P(y) &= \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)}, \\
P(y) &= \sigma\left((2y-1)z\right).
\end{aligned}$$

$$\begin{aligned}
J(\boldsymbol{\theta}) &= -\log P(y \mid \boldsymbol{x}) \\
&= -\log \sigma\left((2y-1)z\right) \\
&= \zeta\left((1-2y)z\right).
\end{aligned}$$

SOFTPLUS

```
softplus: torch.nn.Module = torch.nn.Softplus()
outputs: torch.Tensor = softplus(inputs)

outputs = F.softplus(inputs)
```



MLE-SOFTMAX

$$z_i = \log \tilde{P}(y = i \mid \boldsymbol{x}).$$

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j).$$

NUMERICAL STABILITY ON THE SOFTMAX

$$\hat{y}_j = \frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp(o_j - \bar{o}) \exp \bar{o}}{\sum_k \exp(o_k - \bar{o}) \exp \bar{o}} = \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})}.$$

$$\log \hat{y}_j = \log \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})} = o_j - \bar{o} - \log \sum_k \exp(o_k - \bar{o}).$$

L1LOSS

```
loss = torch.nn.L1Loss()
loss_value = loss(outputs, targets)
```

L1LOSS

CLASS torch.nn.L1Loss(*size_average=None, reduce=None, reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The sum operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

Supports real-valued and complex-valued inputs.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when `reduce` is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*str, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

MSELOSS

```
loss = torch.nn.MSELoss()
loss_value = loss(outputs, targets)
```

MSELOSS

CLASS torch.nn.MSELoss(*size_average=None, reduce=None, reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The mean operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when `reduce` is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*str, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

HUBERLOSS

```python
loss = torch.nn.HuberLoss()
loss_value = loss(outputs, targets)
```

HUBERLOSS

CLASS torch.nn.HuberLoss(*reduction='mean', delta=1.0*) [SOURCE]

Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise. This loss combines advantages of both `L1Loss` and `MSELoss`; the delta-scaled L1 region makes the loss less sensitive to outliers than `MSELoss`, while the L2 region provides smoothness over `L1Loss` near 0. See Huber loss for more information.

For a batch of size N, the unreduced loss can be described as:

$$\ell(x, y) = L = \{l_1, ..., l_N\}^T$$

with

$$l_n = \begin{cases} 0.5(x_n - y_n)^2, & \text{if } |x_n - y_n| < delta \\ delta * (|x_n - y_n| - 0.5 * delta), & \text{otherwise} \end{cases}$$

If *reduction* is not *none*, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

NOTE

When delta is set to 1, this loss is equivalent to `SmoothL1Loss`. In general, this loss differs from `SmoothL1Loss` by a factor of delta (AKA beta in Smooth L1). See `SmoothL1Loss` for additional discussion on the differences in behavior between the two losses.

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`
- **delta** (*float, optional*) – Specifies the threshold at which to change between delta-scaled L1 and L2 loss. The value must be positive. Default: 1.0

BINARY CROSS ENTROPY

```
loss = torch.nn.BCEWithLogitsLoss()
loss_value = loss(outputs, targets)
```

BCEWITHLOGITSLOSS

CLASS torch.nn.BCEWithLogitsLoss(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*, *pos_weight=None*) [SOURCE]

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets *t[i]* should be numbers between 0 and 1.

It's possible to trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$\ell_c(x, y) = L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x$$

where c is the class number ($c > 1$ for multi-label binary classification, $c = 1$ for single-label binary classification), n is the number of the sample in the batch and p_c is the weight of the positive answer for the class c.

$p_c > 1$ increases the recall, $p_c < 1$ increases the precision.

For example, if a dataset contains 100 positive and 300 negative examples of a single class, then *pos_weight* for the class should be equal to $\frac{300}{100} = 3$. The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

CROSS ENTROPY

```
loss = torch.nn.CrossEntropyLoss()
loss_value = loss(outputs, targets)
```

CROSSENTROPYLOSS

CLASS torch.nn.CrossEntropyLoss(*weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0*) [SOURCE]

This criterion computes the cross entropy loss between input logits and target.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain the unnormalized logits for each class (which do *not* need to be positive or sum to 1, in general). *input* has to be a Tensor of size (C) for unbatched input, $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The last being useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C)$ where C is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Note that this case is equivalent to applying `LogSoftmax` on an input, followed by `NLLLoss`.

- Probabilities for each class; useful when labels beyond a single class per minibatch item are required, such as for blended labels, label smoothing, etc. The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

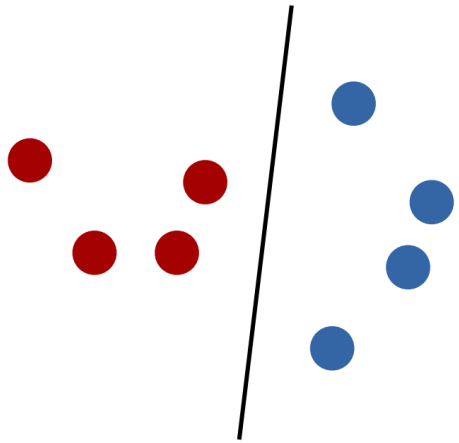
$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

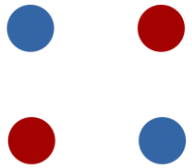
• NOTE

The performance of this criterion is generally better when *target* contains class indices, as this allows for optimized computation. Consider providing *target* as class probabilities only when a single class label per minibatch item is too restrictive.

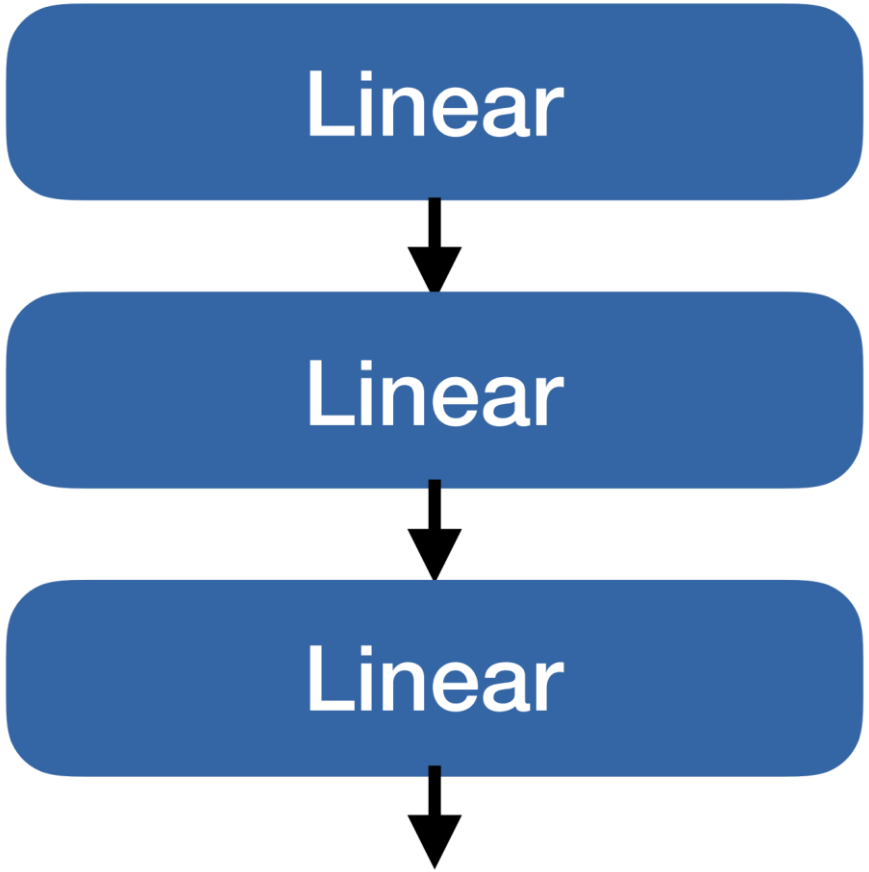
5 - NON LINEARITIES

LIMITATIONS OF LINEAR MODELS

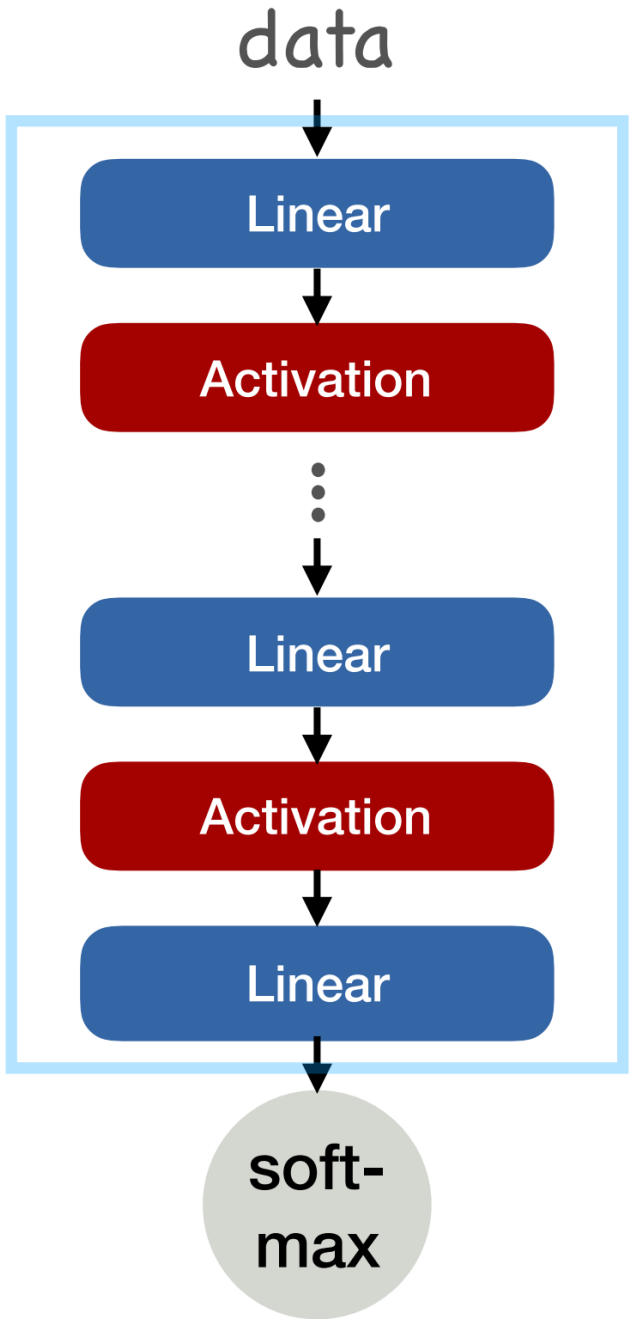




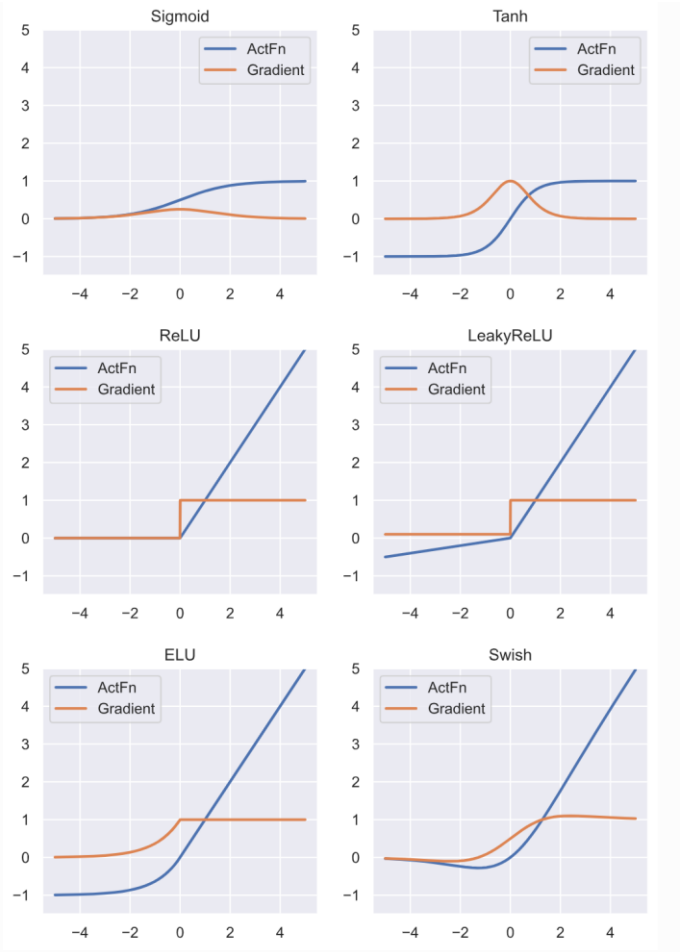
MULTIPLE LINEAR LAYERS = LINEAR LAYER



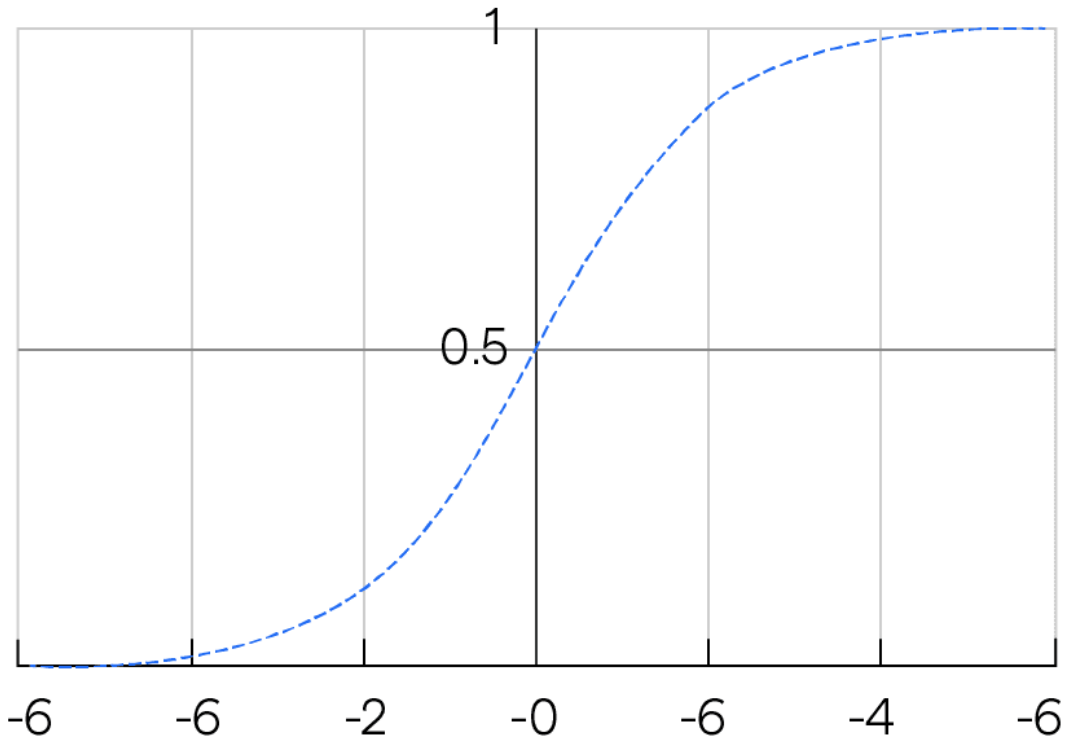
DEEP NETWORKS



ACTIVATIONS FUNCTIONS



SIGMOID

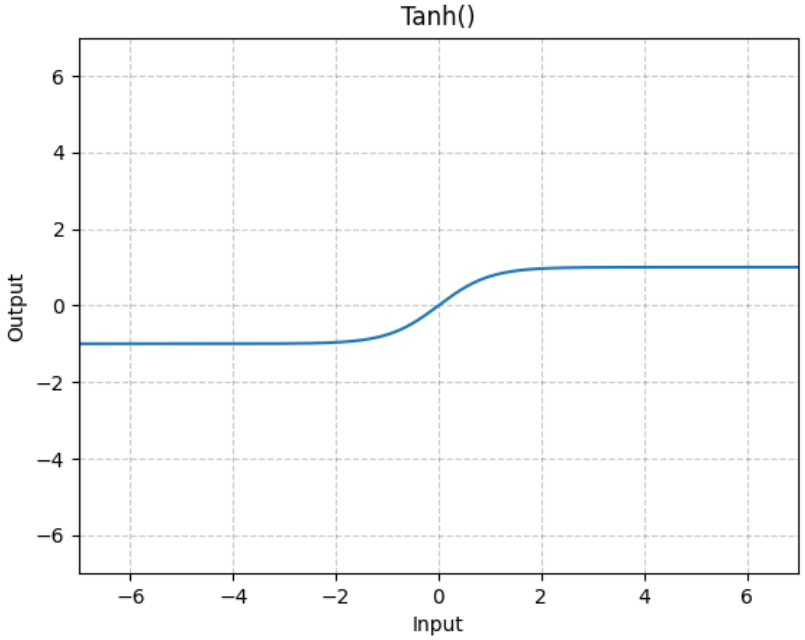


$$\sigma(o) = \frac{1}{1 + e^{-o}}$$

```
# torch.nn
sigmoid = torch.nn.Sigmoid()
outputs = sigmoid(inputs)

# torch.nn.functional
outputs = F.sigmoid(inputs)
```

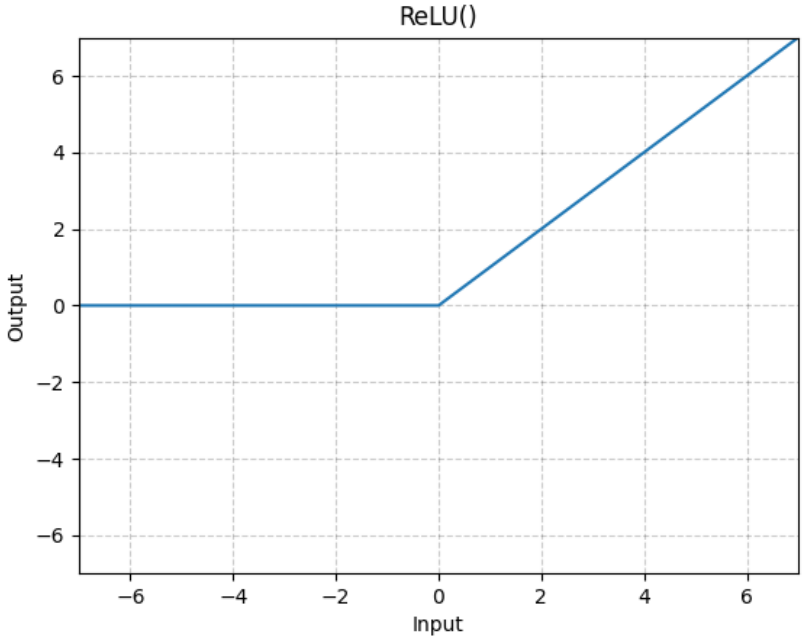
TANH

```
tanh: torch.nn.Module = torch.nn.Tanh()
outputs: torch.Tensor = tanh(inputs)

outputs = F.tanh(inputs)
```



RELU

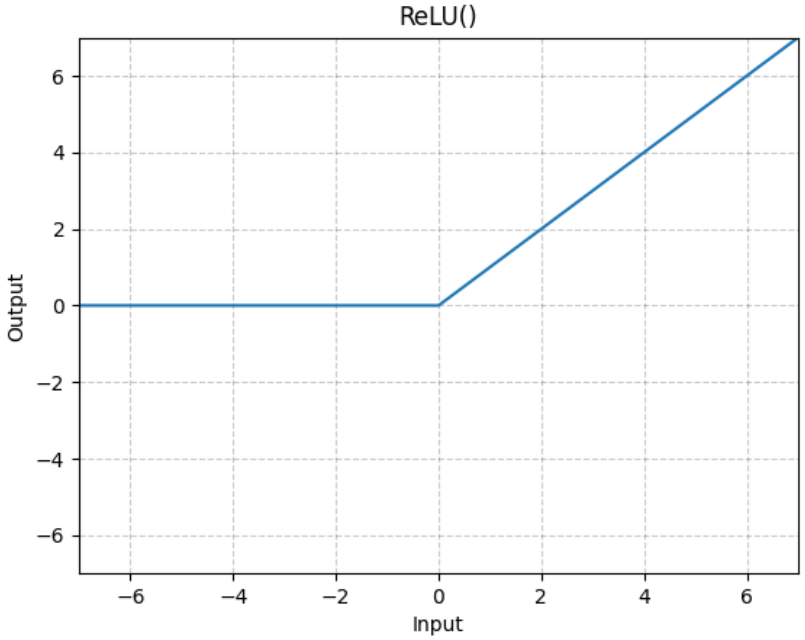
```
relu = torch.nn.ReLU()
outputs: torch.Tensor = relu(inputs)

outputs = F.relu(inputs)
```



DEAD RELU PROBLEM

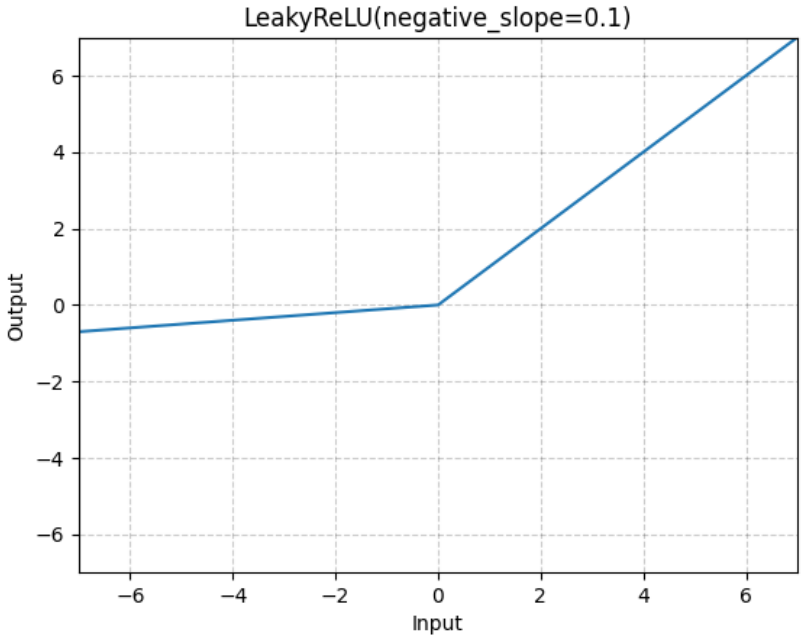
To prevent it:

- Initialize carefully
- Decrease learning rate



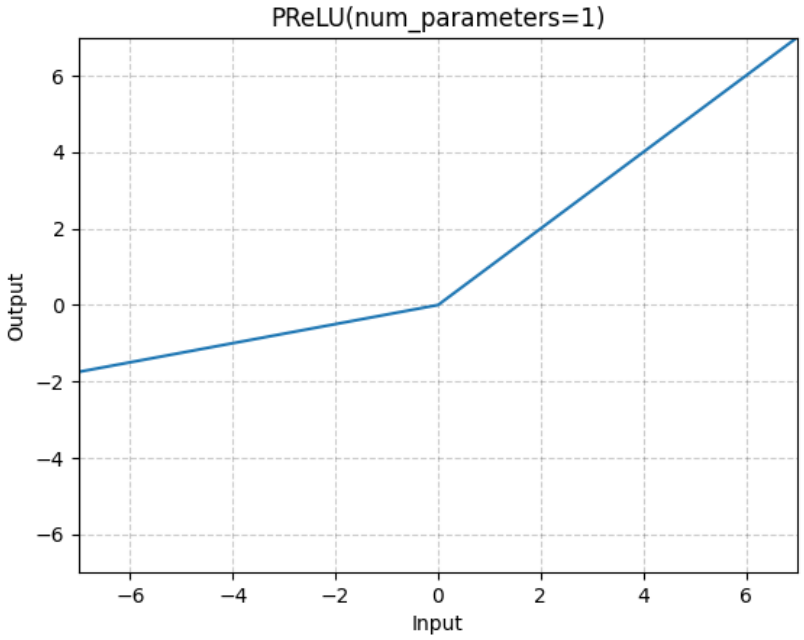
LEAKY RELU

```python
non_linearity: torch.nn.Module = torch.nn.LeakyReLU()
outputs: torch.Tensor = non_linearity(inputs)

outputs = F.leaky_relu(inputs)
```



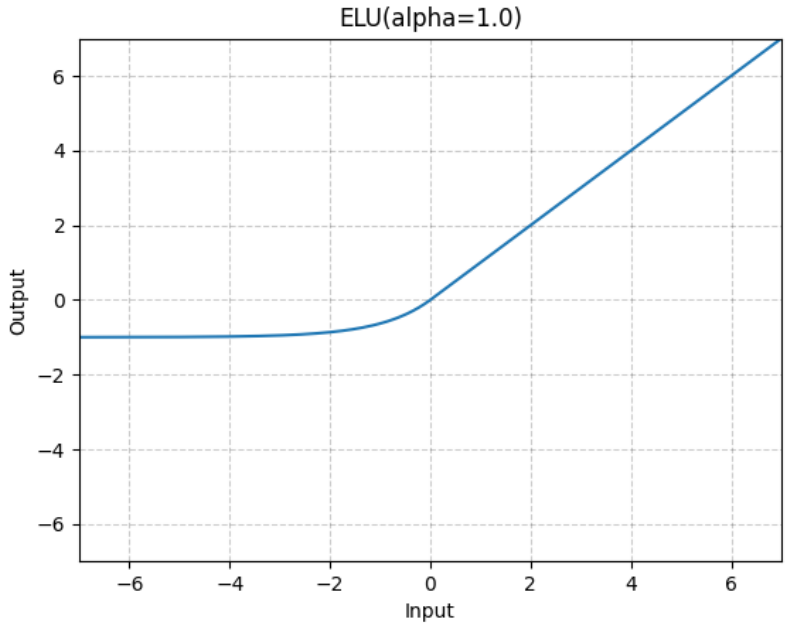
PRELU

```
non_linearity: torch.nn.Module = torch.nn.LeakyReLU()
outputs: torch.Tensor = non_linearity(inputs)

outputs = F.leaky_relu(inputs)
```



ELU

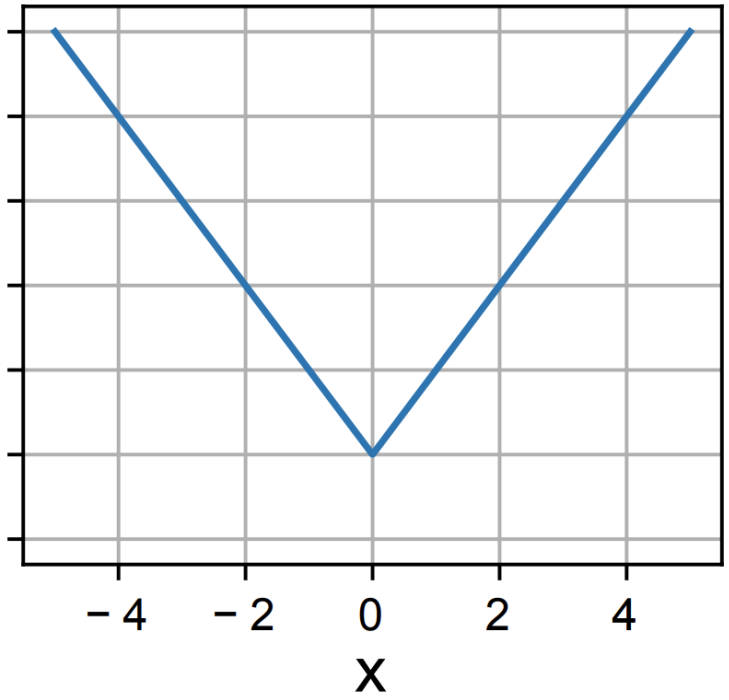
```
non_linearity: torch.nn.Module = torch.nn.LeakyReLU()
outputs: torch.Tensor = non_linearity(inputs)

outputs = F.leaky_relu(inputs)
```



ABSOLUTE VALUE

```
outputs = torch.abs(inputs)
```

```
@article{valles2023empirical,
  title={Empirical study of the modulus as activation function in computer vision applications},
  author={Vall{\'e}s-P{\'e}rez, Iv{\'a}n and Soria-Olivas, Emilio and Mart{\'\i}nez-Sober, Marcelino and Serrano-L{\'o}pez, Antonio J and Vila-Franc{\'e}s, Joan and G{\'o}mez-Sanch{\'\i}s, Juan},
  journal={Engineering Applications of Artificial Intelligence},
  volume={120},
  pages={105863},
  year={2023},
  publisher={Elsevier}
}
```



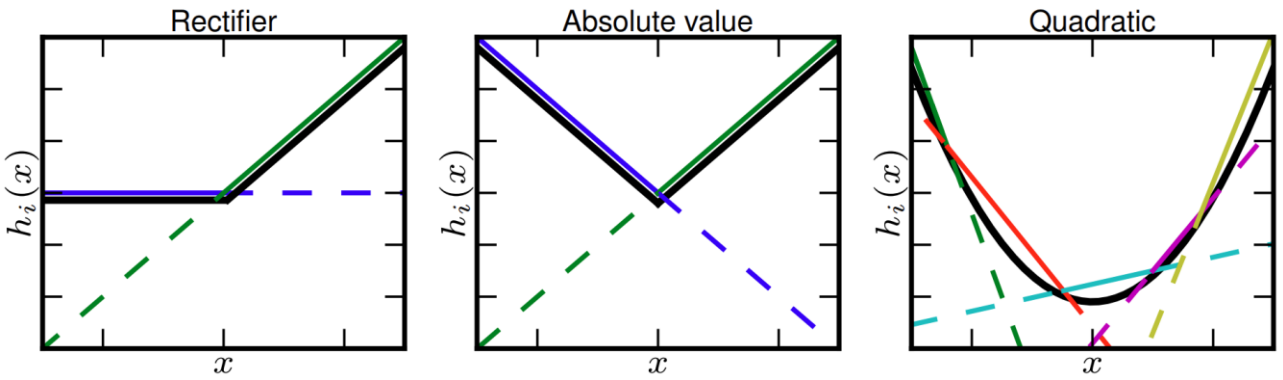
MAXOUT

$$W \in \mathbb{R}^{d \times m \times k}$$
$$b \in \mathbb{R}^{m \times k}$$

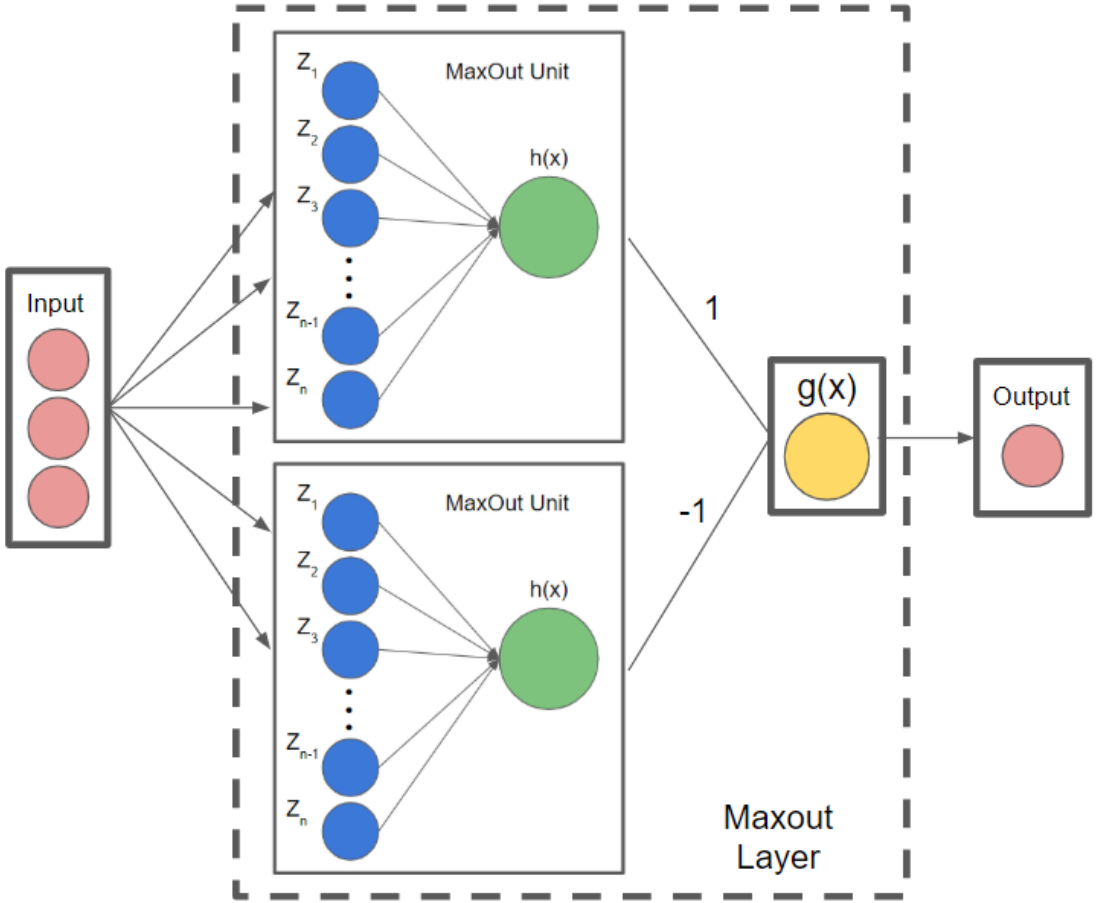
$$z_{ij} = x^T W_{...ij} + b_{ij}$$

$$h_i(x) = \max_{j \in [1,k]} z_{ij}$$

$$f(x) = \max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

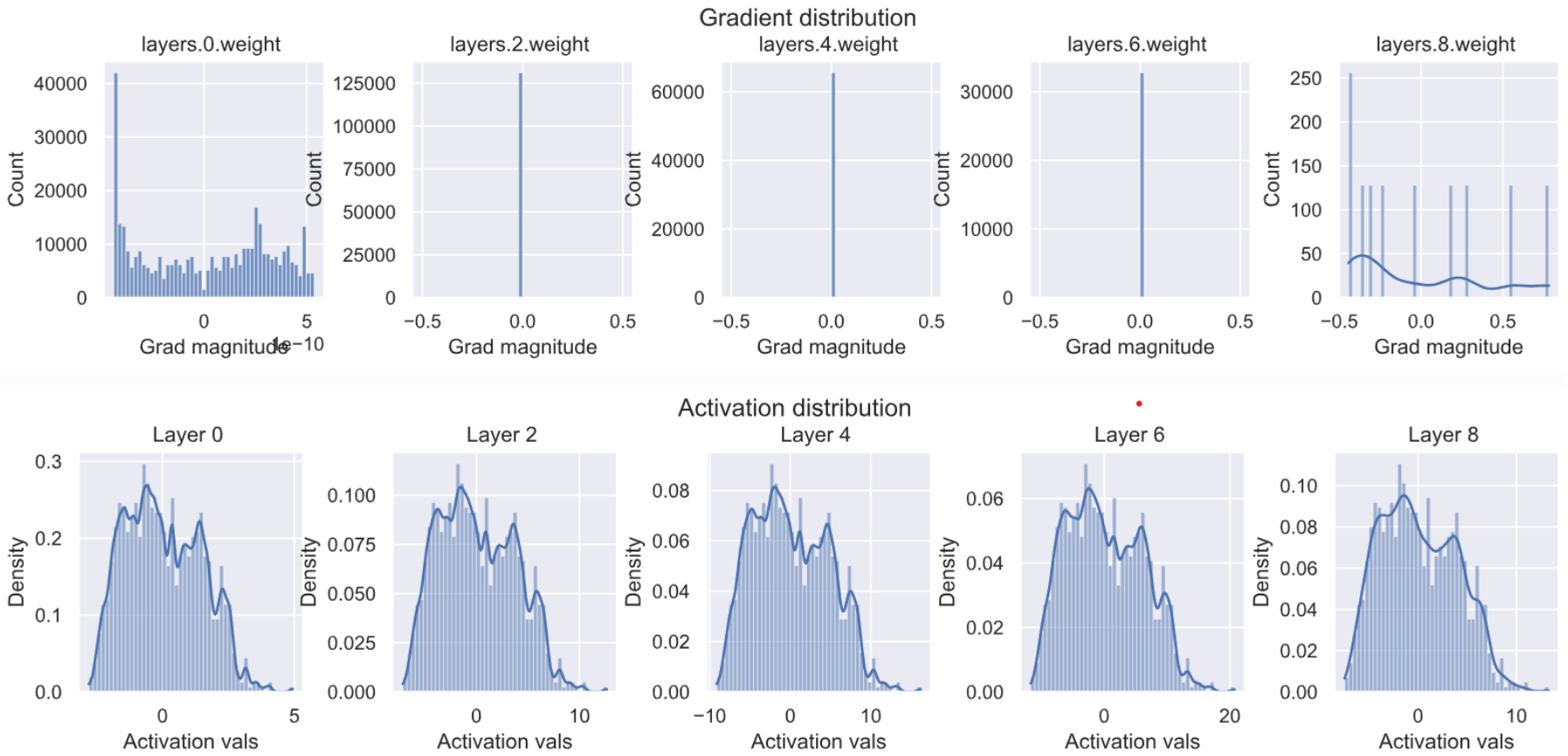


MAXOUT - ORDER 2

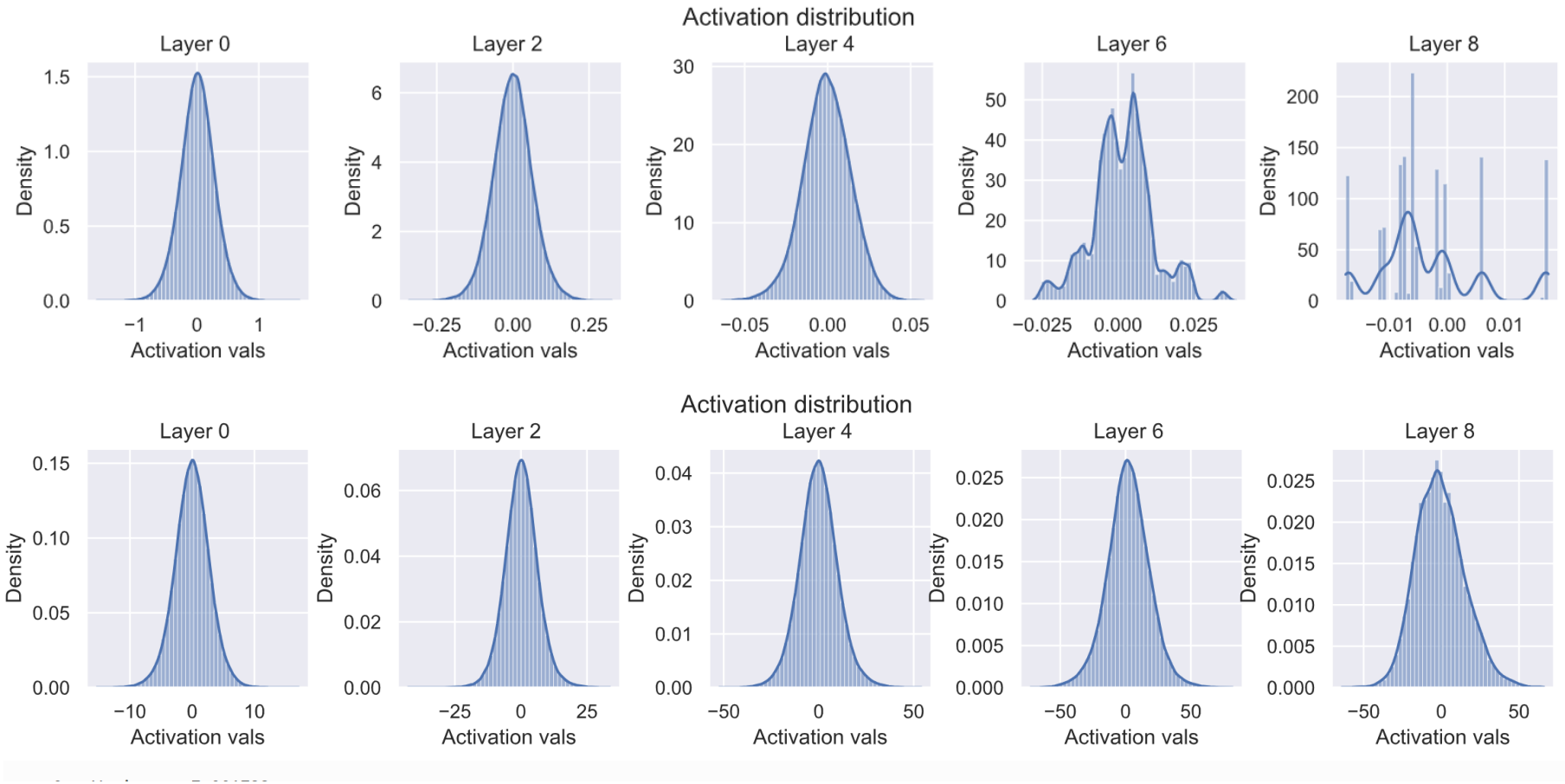


6 - INITIALIZATION

CONSTANT INITIALIZATION



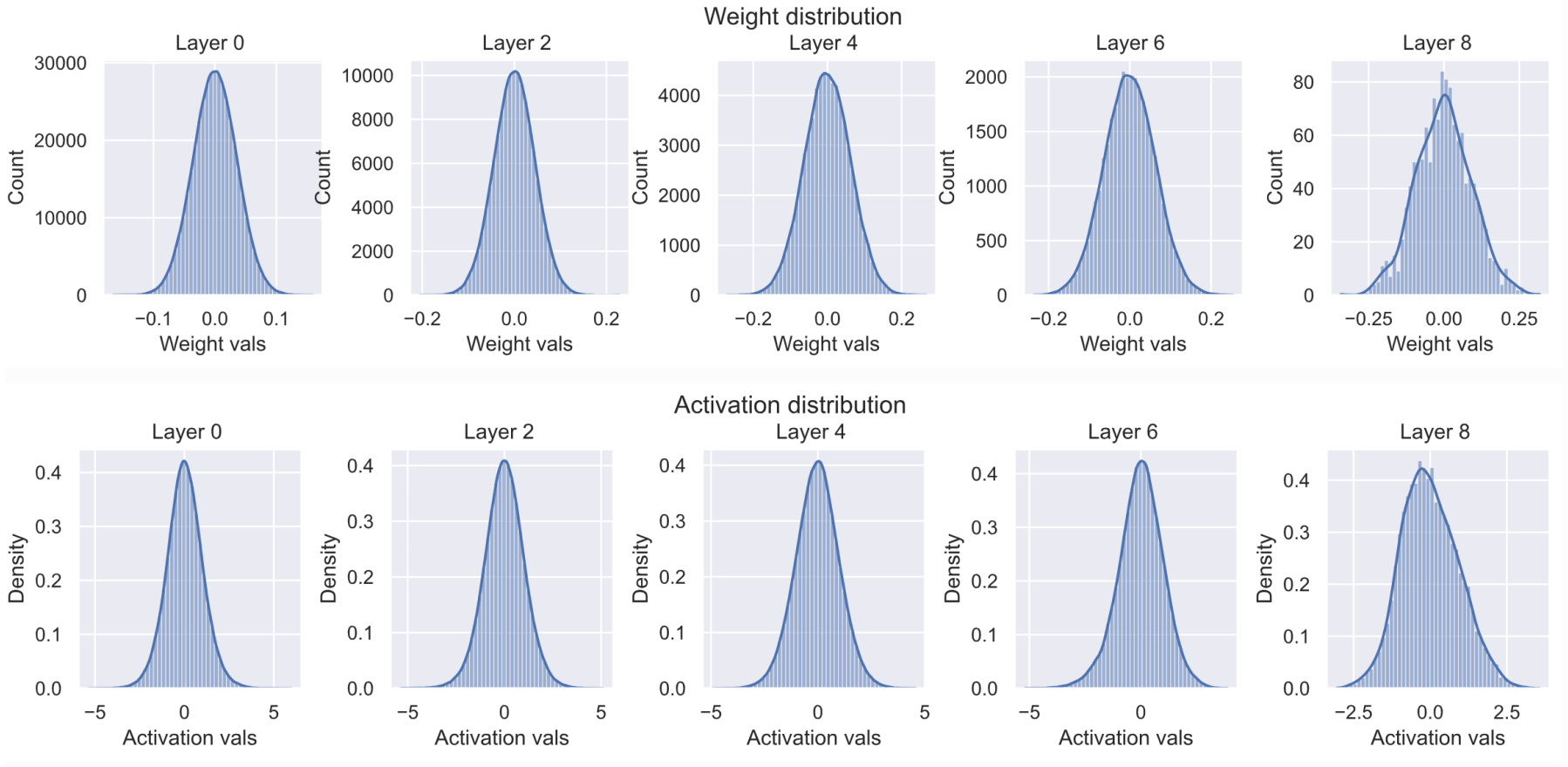
CONSTANT VARIANCE



CONSTANT ACTIVATIONS VARIANCE

$$
\begin{aligned}
y_i &= \sum_j w_{ij} x_j && \text{Calculation of a single output neuron without bias} \\
\mathrm{Var}(y_i) = \sigma_x^2 &= \mathrm{Var}\left(\sum_j w_{ij} x_j\right) \\
&= \sum_j \mathrm{Var}(w_{ij} x_j) && \text{Inputs and weights are independent of each other} \\
&= \sum_j \mathrm{Var}(w_{ij}) \cdot \mathrm{Var}(x_j) && \text{Variance rule (see above) with expectations being zero} \\
&= d_x \cdot \mathrm{Var}(w_{ij}) \cdot \mathrm{Var}(x_j) && \text{Variance equal for all } d_x \text{ elements} \\
&= \sigma_x^2 \cdot d_x \cdot \mathrm{Var}(w_{ij}) \\
\Rightarrow \mathrm{Var}(w_{ij}) = \sigma_W^2 &= \frac{1}{d_x}
\end{aligned}
$$

CONSTANT ACTIVATIONS VARIANCE

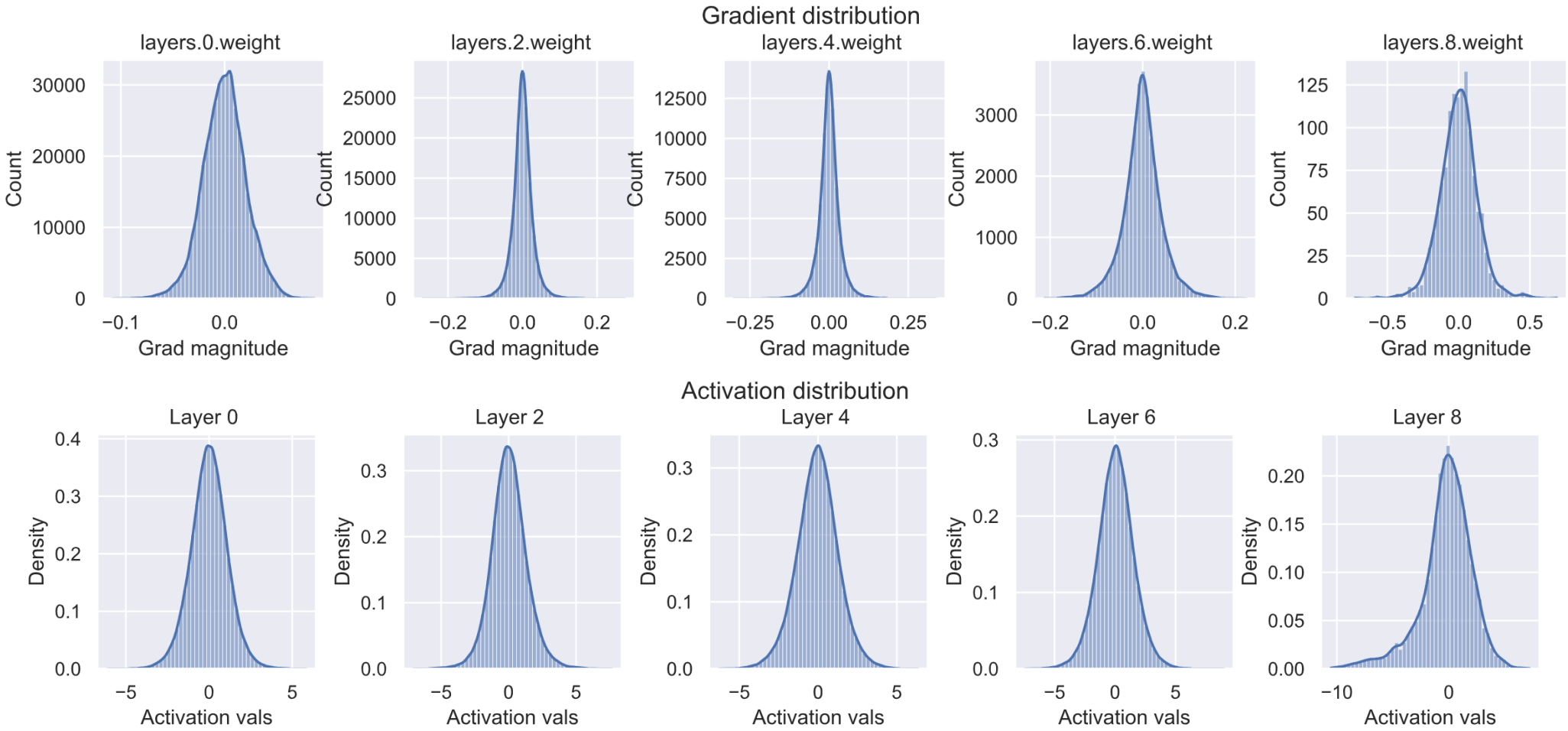


XAVIER INITIALIZATION

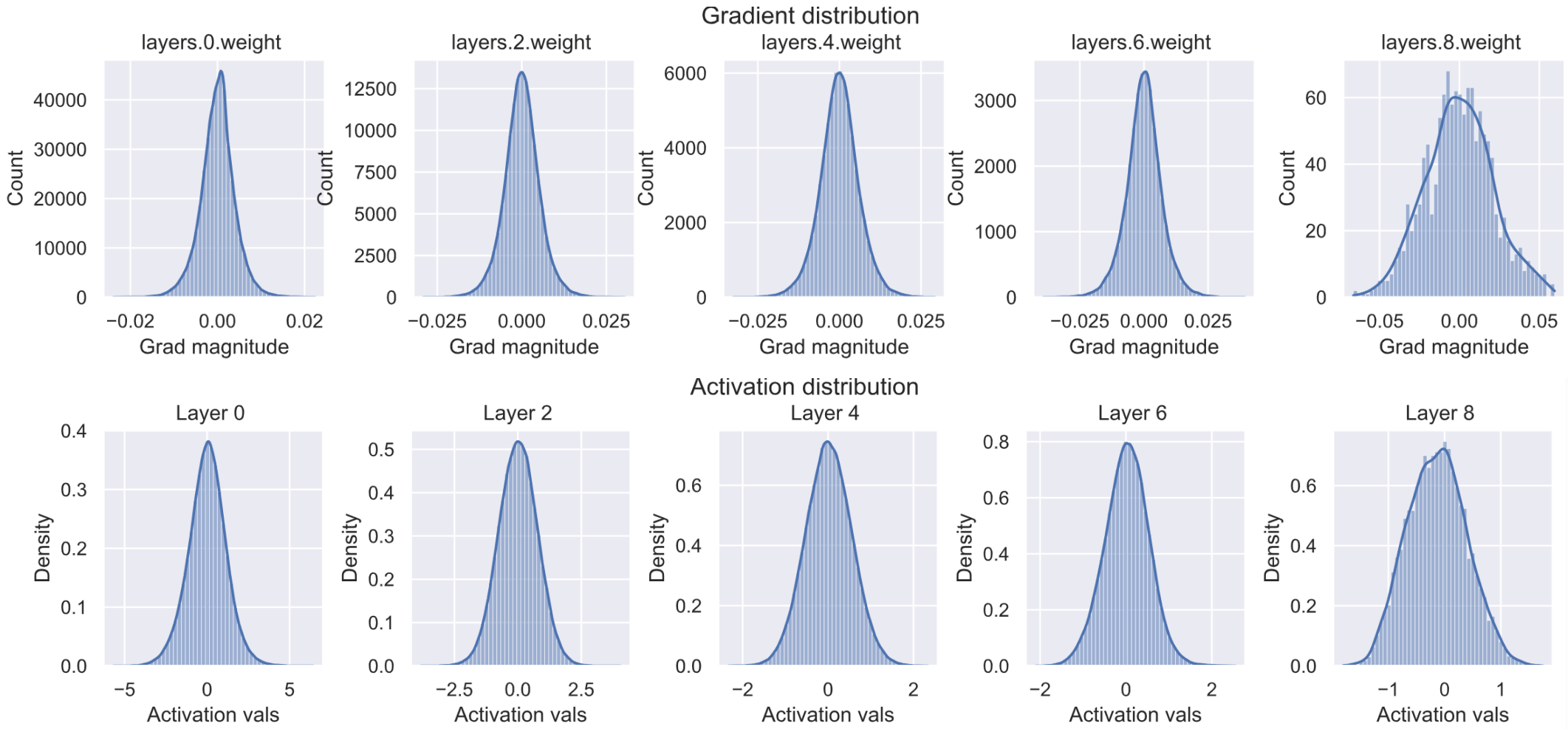
$$W \sim \mathcal{N}\left(0, \frac{2}{d_x + d_y}\right)$$

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}}\right]$$

XAVIER INITIALIZATION



XAVIER WITH TANH

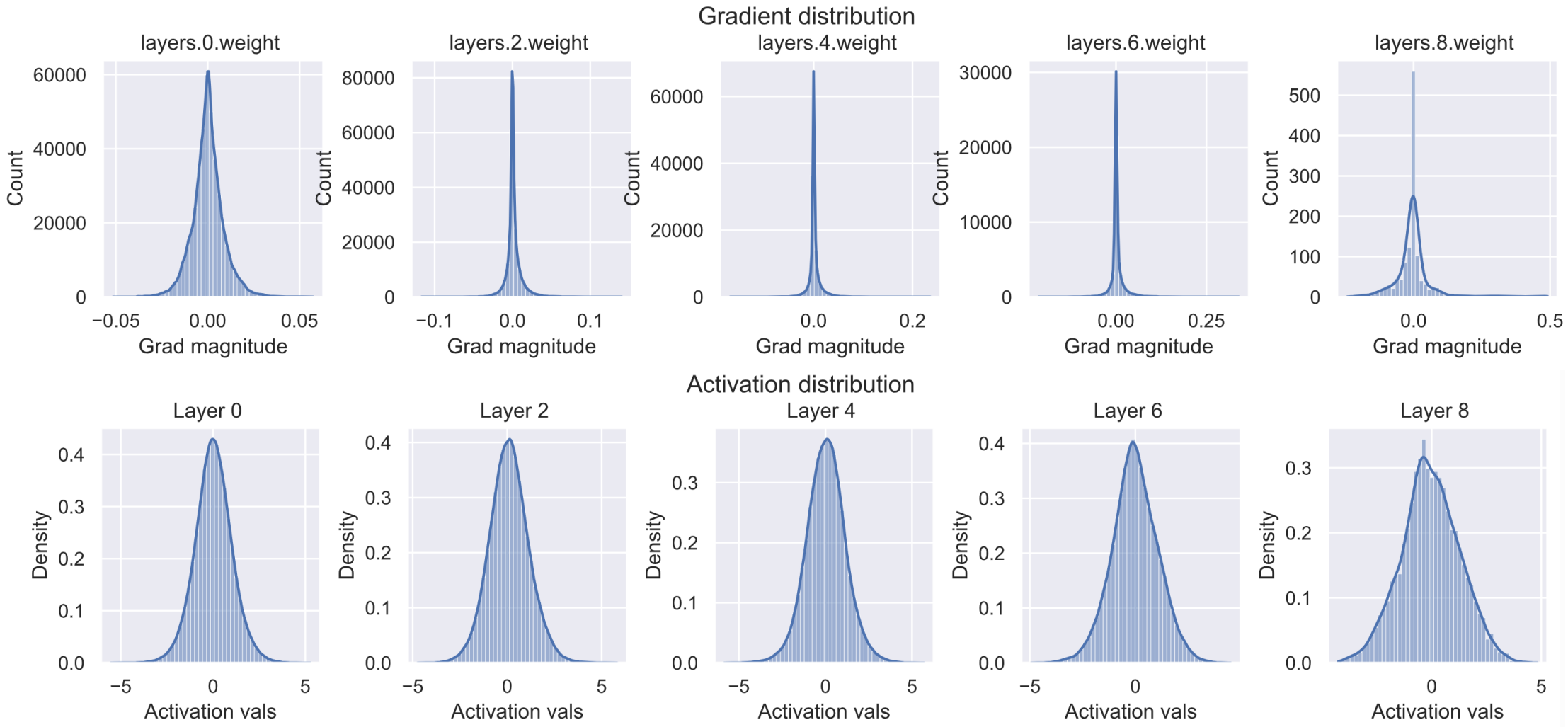


KAIMING INITIALIZATION

$$\mathrm{Var}(w_{ij}x_j) = \underbrace{\mathbb{E}[w_{ij}^2]}_{=\mathrm{Var}(w_{ij})}\mathbb{E}[x_j^2] - \underbrace{\mathbb{E}[w_{ij}]^2}_{=0}\mathbb{E}[x_j]^2 = \mathrm{Var}(w_{ij})\mathbb{E}[x_j^2]$$

$$\begin{aligned}\mathbb{E}[x^2] &= \mathbb{E}[\max(0,\tilde{y})^2] \\ &= \frac{1}{2}\mathbb{E}[\tilde{y}^2] \qquad\qquad \tilde{y} \text{ is zero-centered and symmetric} \\ &= \frac{1}{2}\mathrm{Var}(\tilde{y})\end{aligned}$$

KAIMING INITIALIZATION



7 - OPTIMIZATION

OPTIMIZE MODEL

Differentiable $\ell(\mathbf{o}, \mathbf{y})$

Regression

Distance norm

$$\ell(\mathbf{o}, \mathbf{y}) = \|\mathbf{o} - \mathbf{y}\|$$

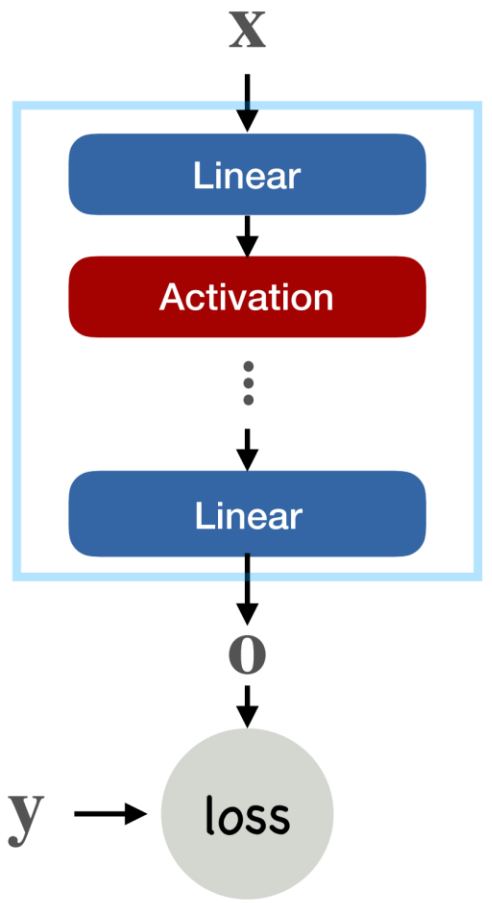
Classification

Cross Entropy

$$\ell(\mathbf{o}, y) = -\log p(y)$$

Over training dataset

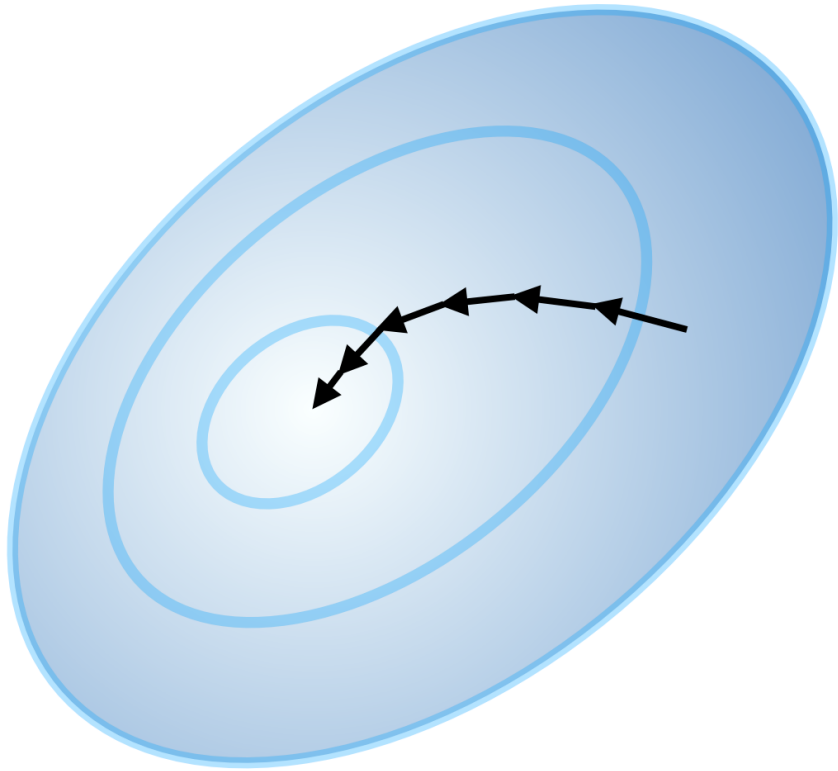
$$L(\theta) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim D}[\ell(f(\mathbf{x}, \theta), \mathbf{y})]$$



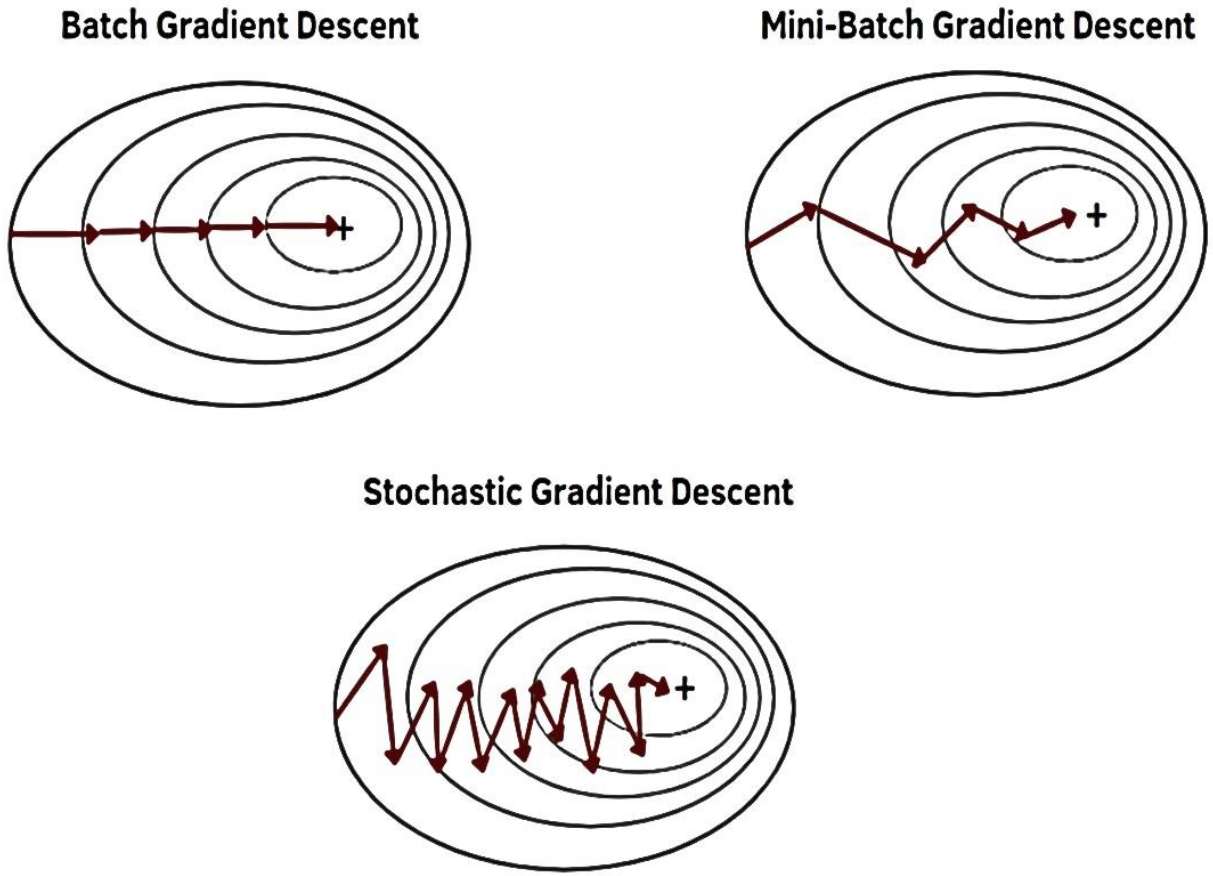
GRADIENT DESCENT

Repeat until convergence:

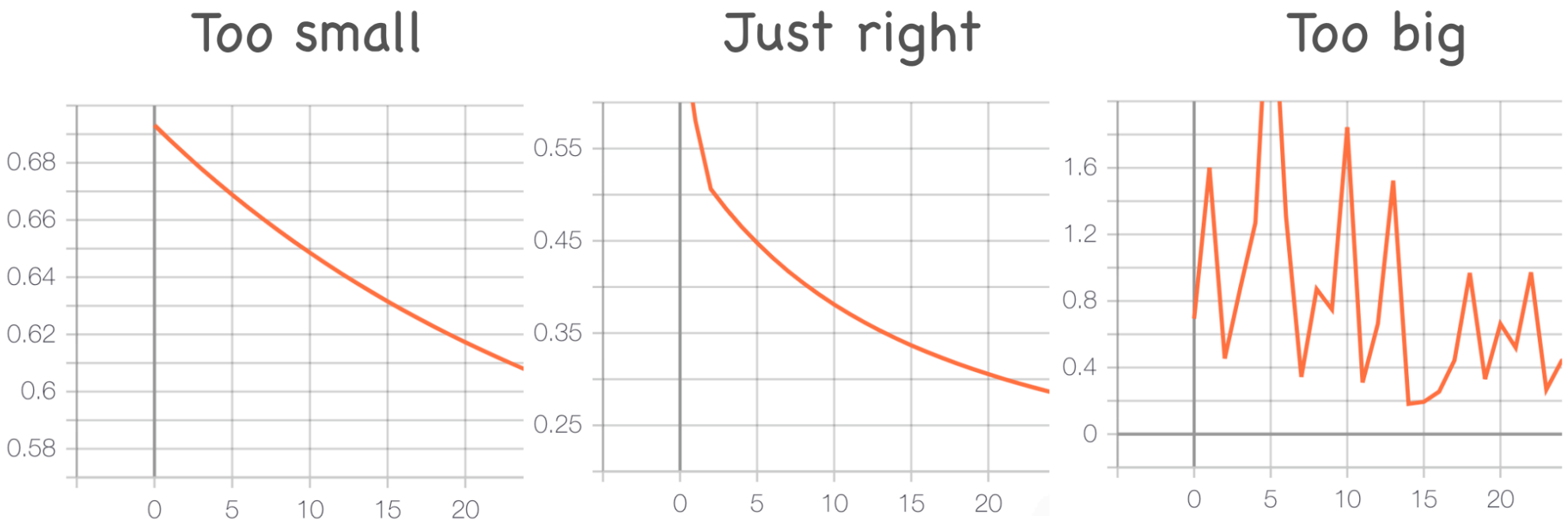
$$\theta := \theta - \epsilon \frac{dL(\theta)}{d\theta}$$



STOCHASTIC AND MINI-BATCH GRADIENT DESCENT



LEARNING RATE MATTERS



REFERENCES

- https://www.philkr.net/cs342
- https://storage.googleapis.com/deepmind-media/UCLxDeepMind_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf
- https://www.deeplearningbook.org
- https://www.youtube.com/@3blue1brown
- https://medium.com/@hunter-j-phillips/a-simple-introduction-to-broadcasting-db8e581368b3