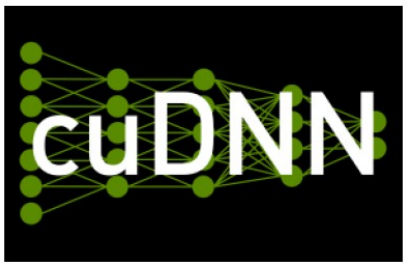

# Neural Networks

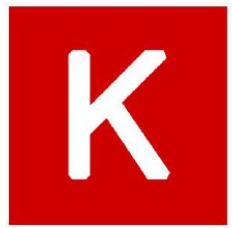
Katharina Breininger, Mingxuan Gu, Noah Maul, Zhaoya Pan, Luca Reeb, Florian Thamm, Sulaiman Vesal, Tobias Würfl, Zijin Yang
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
October 13, 2019

# Flexibility vs. abstraction

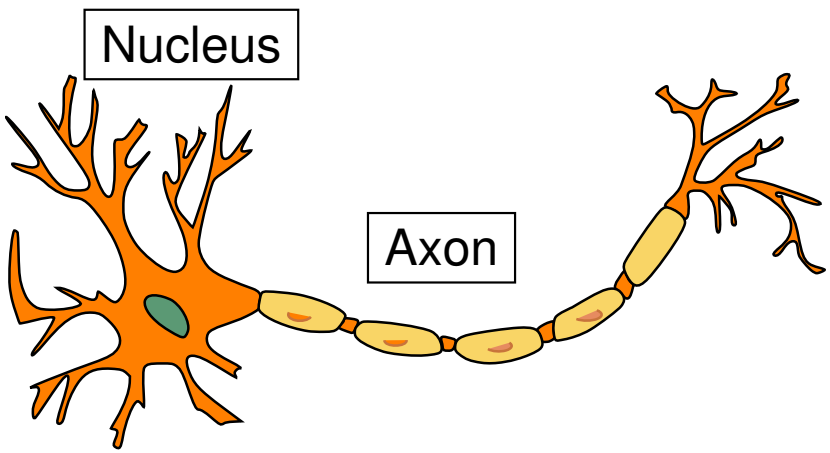

- Linear Algebra operations
- Bare metal

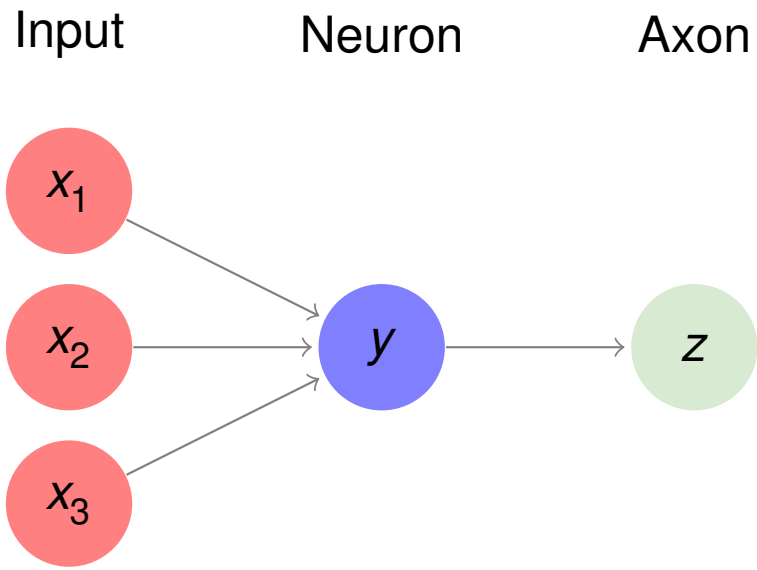

- Compiles graphs of Tensor operations
- High flexibility




- Stacks together elementary layers
- Reduced flexibility

# Artifical Neural Networks




$$\mathbf{y} = f\left(\sum_{N} w_i x_i\right)$$

# Terminology

- We will call $\frac{\partial L}{\partial \hat{\mathbf{y}}}$ the **error E** in the exercises
- "Layer" it is now a technical term. Layers must not be present in graphical depictions. E.g. activation functions become "layers"


# Neural Network

# Neural Network

In layer-oriented frameworks we typically have a neural network object which

# Neural Network

In layer-oriented frameworks we typically have a neural network object which

- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function

# Neural Network

In layer-oriented frameworks we typically have a neural network object which

- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**

# Neural Network

In layer-oriented frameworks we typically have a neural network object which

- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**
- has **no explicit knowledge** about the graph of layers it contains

# Neural Network

In layer-oriented frameworks we typically have a neural network object which

- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**
- has **no explicit knowledge** about the graph of layers it contains
- **recursively calls forward** on its layers passing the input-data
- **recursively calls backward** on its layers passing the error

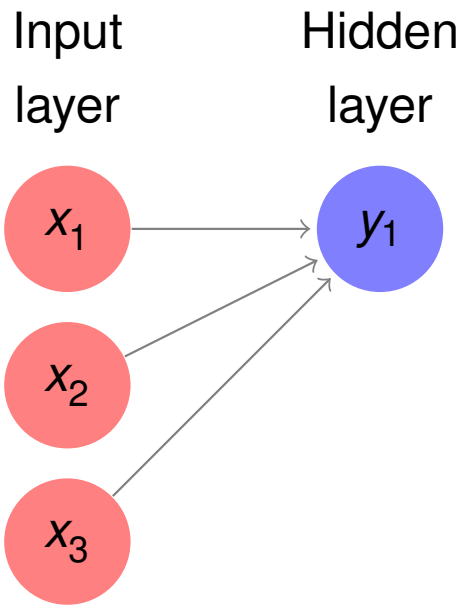
# Neural Network

In layer-oriented frameworks we typically have a neural network object which

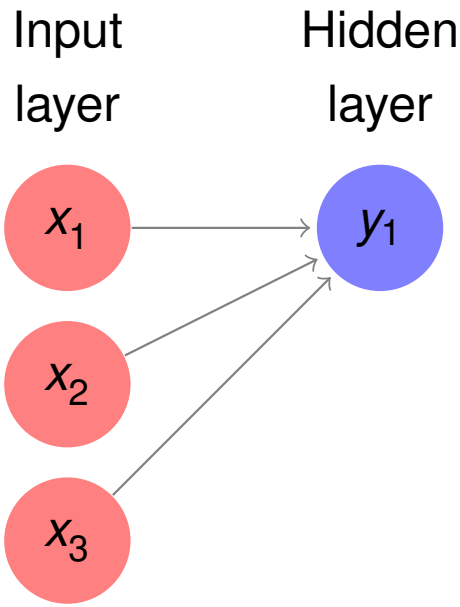
- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**
- has **no explicit knowledge** about the graph of layers it contains
- **recursively calls forward** on its layers passing the input-data
- **recursively calls backward** on its layers passing the error
- in our case it stores the loss over iterations, while in other frameworks this is commonly separated into an optimizer class


# Fully Connected Layer

# Forward



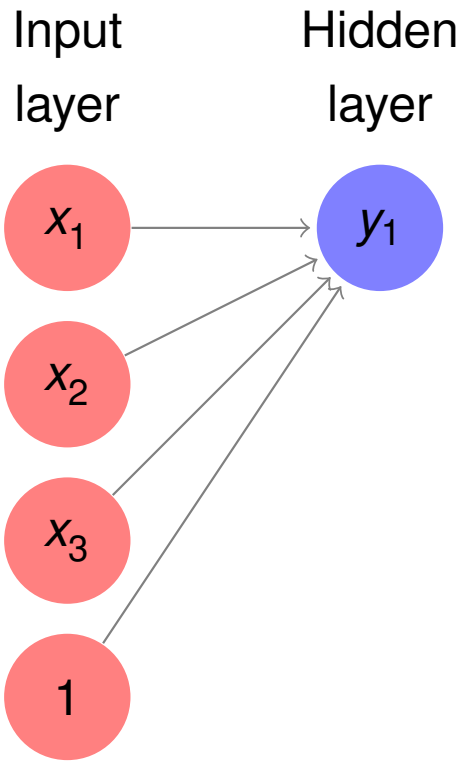
# Forward




$$\begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}^T \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + w_{n+1} = \hat{y}$$

$$\mathbf{w}^T\mathbf{x} = \hat{y}$$

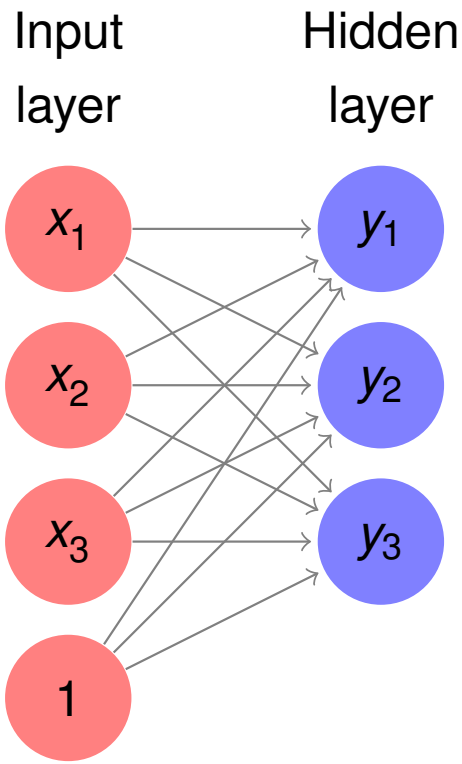
# Forward



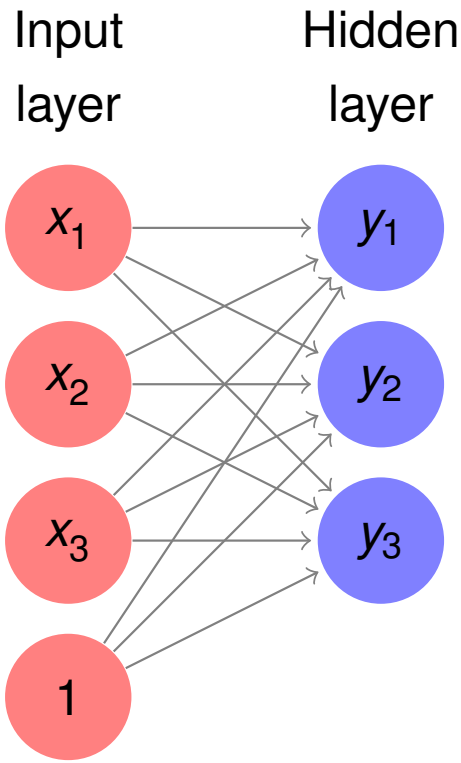

$$\begin{pmatrix} w_1 \\ \vdots \\ w_n \\ w_{n+1} \end{pmatrix}^T \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \hat{y}$$

$$\mathbf{w}^T\mathbf{x} = \hat{y}$$

# Forward



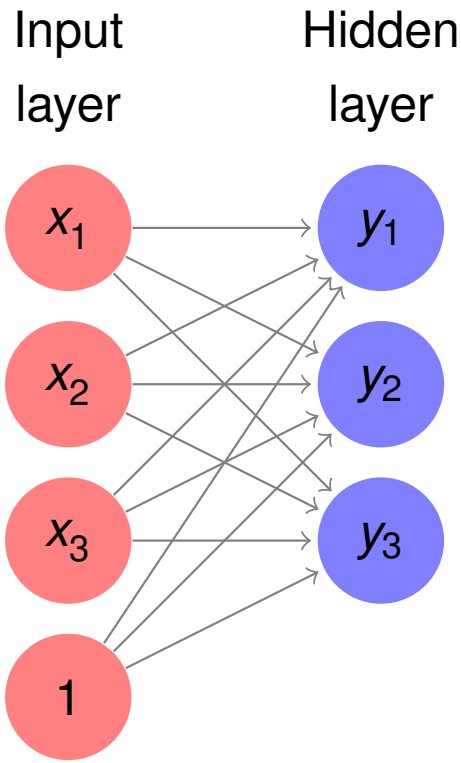
# Forward




$$\begin{pmatrix} w_{1,1} & \cdots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m} \\ w_{n+1,1} & \cdots & w_{n+1,m} \end{pmatrix}^T \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \begin{pmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_m \end{pmatrix}$$

$$\mathbf{W}\mathbf{x} = \hat{\mathbf{y}}$$

# Forward




$$\begin{pmatrix} w_{1,1} & \cdots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m} \\ w_{n+1,1} & \cdots & w_{n+1,m} \end{pmatrix}^T \begin{pmatrix} x_{1,1} & \cdots & x_{1,b} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,b} \\ 1 & \cdots & 1 \end{pmatrix}$$

$$\mathbf{WX} = \hat{\mathbf{Y}} \tag{1}$$

# Backward

- Return gradient with respect to $\mathbf{X}$:

# Backward

- Return gradient with respect to **X**:

$$\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n \tag{2}$$

- $\mathbf{E_n}$: **error_tensor** passed downward

# Backward

- Return gradient with respect to **X**:

$$\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n \tag{2}$$

- Update **W** using gradient with respect to **W**:

- $\mathbf{E_n}$: **error_tensor** passed downward

# Backward

- Return gradient with respect to **X**:

$$\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n \tag{2}$$

- Update **W** using gradient with respect to **W**:

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta \cdot \mathbf{E_n}\mathbf{X}^T \tag{3}$$

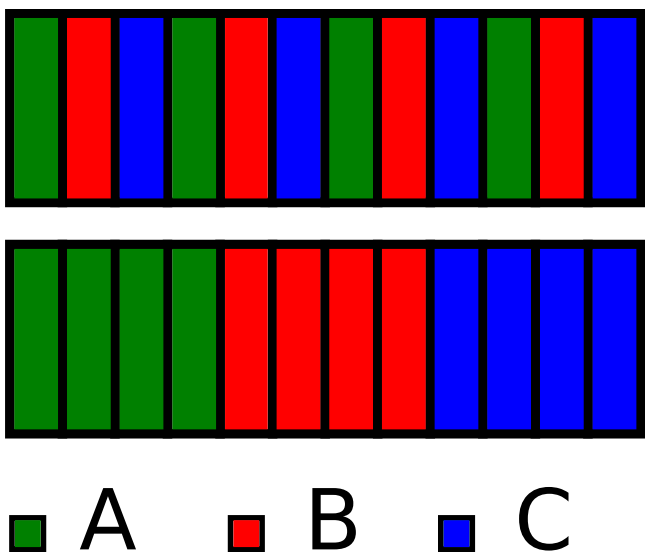
**Note**: Dynamic programming part of Backpropagation

- $\mathbf{E_n}$: **error_tensor** passed downward
- $\eta$: learning rate

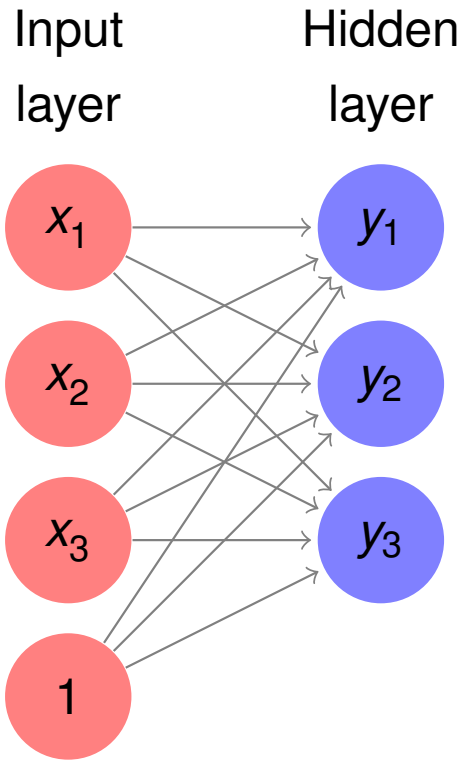
# Memory Layout

- Numpy uses C ordering by default
- Wrong ordering will cause strided data access
- We want the batch size to be the outermost loop

  $\rightarrow$ We have to adjust our formulas for the implementation



# Forward - Our Memory Layout




$$\begin{pmatrix} x_{1,1} & \dots & x_{1,b} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,b} \\ 1 & \dots & 1 \end{pmatrix}^T \begin{pmatrix} w_{1,1} & \dots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \dots & w_{n,m} \\ w_{n+1,1} & \dots & w_{n+1,m} \end{pmatrix}$$

$$\mathbf{X}'\mathbf{W}' = \hat{\mathbf{Y}}' \tag{4}$$

with

$$\mathbf{X}' = \mathbf{X}^\mathbf{T},\ \mathbf{W}' = \mathbf{W}^\mathbf{T},\ \hat{\mathbf{Y}}' = \hat{\mathbf{Y}}^\mathbf{T} \tag{5}$$

$$\hat{\mathbf{Y}}^\mathbf{T} = (\mathbf{W}\mathbf{X})^\mathbf{T} = \mathbf{X}^\mathbf{T}\mathbf{W}^\mathbf{T} \tag{6}$$

# Backward - Our Memory Layout

- Return gradient with respect to **X**:

$$\mathbf{E'_{n-1}} = \mathbf{E'_n}\mathbf{W'}^{\mathbf{T}} \tag{7}$$

- Update $\mathbf{W'}$ using gradient with respect to $\mathbf{W'}$:

$$\mathbf{W'}^{t+1} = \mathbf{W'}^{t} - \eta \cdot \mathbf{X'}^{\mathbf{T}}\mathbf{E'_n} \tag{8}$$

**Note**: Dynamic programming part of Backpropagation

- $\mathbf{E'_n}$: **error_tensor** passed downward
- $\eta$: learning rate


# Basic Optimization

# SGD

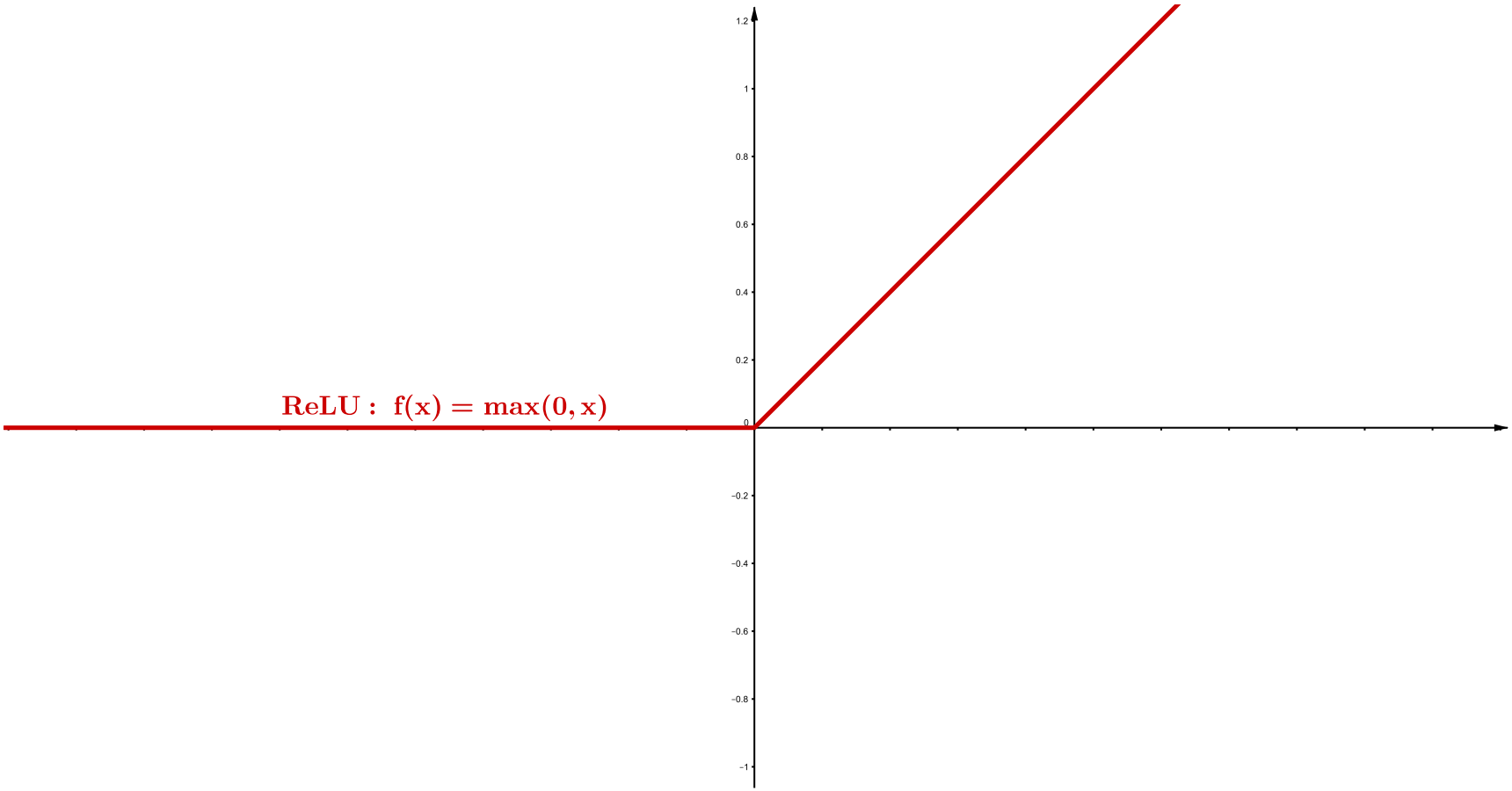
- In order to perform the update as in equation (8) we make use of a dedicated optimizer.
- In the first exercise we implement the Stochastic Gradient Descent Algorithm

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \underbrace{\nabla L(\mathbf{w}^{(k)})}_{\textit{Gradient}}$$

where $\eta$ denotes the learning rate.

# ReLU Activation Function

# Forward



## Backward

**ReLU is not continuously differentiable!**

## Backward

# ReLU is not continuously differentiable!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \tag{9}$$

**Note**: DP part of Backpropagation yet again

## Backward

# ReLU is not continuously differentiable!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \tag{9}$$

**Note**: DP part of Backpropagation yet again

- The scalar $e$ is because activation functions operate elementwise on $\mathbf{E}$

## Backward

# ReLU is not continuously differentiable!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \tag{9}$$

**Note**: DP part of Backpropagation yet again

- The scalar $e$ is because activation functions operate elementwise on $\mathbf{E}$
- If you wonder about $e_n$ instead of 1 consider that this is $\underbrace{\frac{\partial L}{\partial \hat{\mathbf{y}}}}_{\mathbf{E}} \cdot \underbrace{\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{x}}}_{\text{ReLU}}$


# SoftMax Activation Function

# Forward

Labels as $N$-dimensional **one hot** vector $\mathbf{y}$: $\begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix}$

# Forward

Labels as $N$-dimensional **one hot** vector $\mathbf{y}$: $\begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix}$

- Activation(Prediction) $\hat{\mathbf{y}}$ for every element of the batch of size $B$:

$$\hat{y}_k = \frac{\exp(x_k)}{\sum_{j=1}^{N} \exp(x_j)} \tag{10}$$

# Numeric

- If $x_k > 0 \rightarrow e^{x_k}$ might become very large
- To increase numerical stability $x_k$ can be shifted
- $\tilde{x}_k = x_k - \max(\mathbf{x})$
- This leaves the scores unchanged!

# Backward

- Compute for every element of the batch:

$$\mathbf{E}_{n-1} = \hat{y}\left(\mathbf{E}_n - \sum_{j=1}^{N} \mathbf{E}_{n,j}\hat{y}_j\right) \tag{11}$$

# Backward

- Compute for every element of the batch:

$$\mathbf{E}_{n-1} = \hat{y}\left(\mathbf{E}_n - \sum_{j=1}^{N} \mathbf{E}_{n,j}\hat{y}_j\right) \tag{11}$$

- All operations are element-wise

# Backward

- Compute for every element of the batch:

$$\mathbf{E}_{n-1} = \hat{y}\left(\mathbf{E}_n - \sum_{j=1}^{N} \mathbf{E}_{n,j}\hat{y}_j\right) \tag{11}$$

- All operations are element-wise
- Notice the similarity to the sigmoid gradient $\hat{y}(1-\hat{y})$


# CrossEntropy Loss

# Forward

$$loss = \sum_{b=1}^{B} -\ln\left(\hat{y}_k + \epsilon\right) \textbf{ where } y_k = 1 \tag{12}$$

- $\epsilon$ represents the smallest representable number. Take a look into *np.finfo.eps*
- $\epsilon$ increases stability for very wrong predictions to prevent values close to $log(0)$

# Forward

$$loss = \sum_{b=1}^{B} -\ln\left(\hat{y}_k + \epsilon\right) \textbf{ where } y_k = 1 \tag{12}$$

- $\epsilon$ represents the smallest representable number. Take a look into *np.finfo.eps*
- $\epsilon$ increases stability for very wrong predictions to prevent values close to $log(0)$
- Notice: the CrossEntropy Loss requires predictions to be greater than 0,
- thus the CrossEntropyLoss works most stable with softmax predictions.

# Backward

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \tag{13}$$

- $\epsilon$ cancels out due to derivation. An additional $\epsilon$ would distort the gradient dramatically!
- The gradient prohibits predictions of 0 as well.

# Backward

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \tag{13}$$

- $\epsilon$ cancels out due to derivation. An additional $\epsilon$ would distort the gradient dramatically!
- The gradient prohibits predictions of 0 as well.
- Notice that this does **not** depend on an error **E**
- Because it's the starting point of the recursive computation of gradients

Thanks for listening.

**Any questions?**