



FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Neural Networks

F. Thamm, Z. Yang, K. Breininger, S. Jaganathan, C. Liu, L. Rist, N. Maul, L. Folle, N. Gourmelon, Z. Pan, M. Zinnen, K. Packhäuser, B. Geissler, S. Sun, M. Vornehm, J. Wolf, E. Wittmann, W. Karbole, N. Panchi

Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg

April 26, 2024



## Flexibility vs. Abstraction

Low level

High level



- Linear Algebra operations
- Bare metal

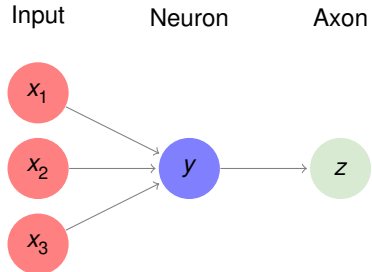
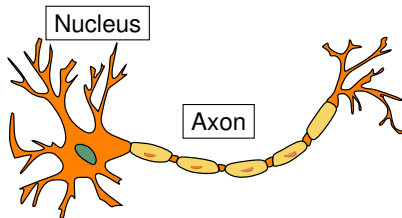


- Compiles graphs of Tensor operations
- High flexibility



- Stacks together elementary layers
- Reduced flexibility

# Artificial Neural Networks



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



FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# 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**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - 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**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - 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**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - 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**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - 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**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - 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 stores the loss over iterations, while in other frameworks this is commonly separated into an optimizer class

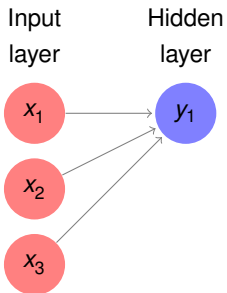


FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

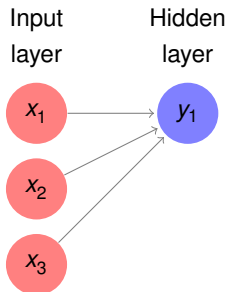
# Fully Connected Layer



## Forward



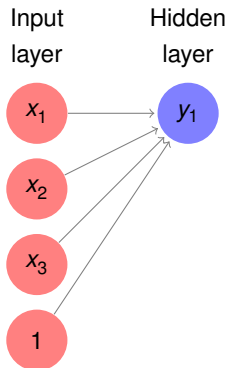
## Forward



$$(w_1 \quad \dots \quad w_n) \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + w_{n+1} = \hat{y}$$

$$\mathbf{w}\mathbf{x} + w_{n+1} = \hat{y}$$

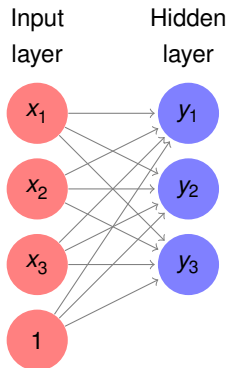
## Forward



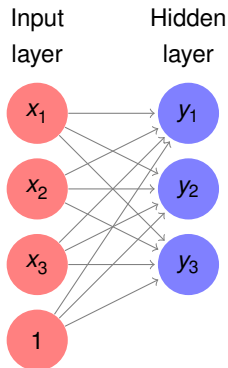
$$(w_1 \quad \dots \quad w_n \quad w_{n+1}) \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \hat{y}$$

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

## Forward



## Forward

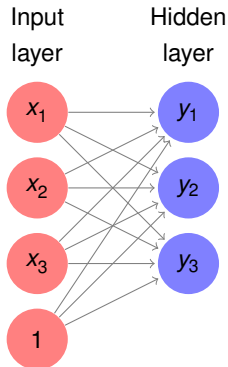


$$\begin{pmatrix} w_{1,1} & \dots & w_{n,1} & w_{n+1,1} \\ \vdots & \ddots & \vdots & \vdots \\ w_{1,m} & \dots & w_{n,m} & w_{n+1,m} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \begin{pmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_m \end{pmatrix}$$

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



## Forward



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

$$\mathbf{WX} = \hat{\mathbf{Y}} \quad (1)$$

## Backward

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

## Backward

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

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

- **E<sub>n</sub>**: **error\_tensor** passed downward

## Backward

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

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

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

- **E<sub>n</sub>**: **error\_tensor** passed downward

## Backward

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

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

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

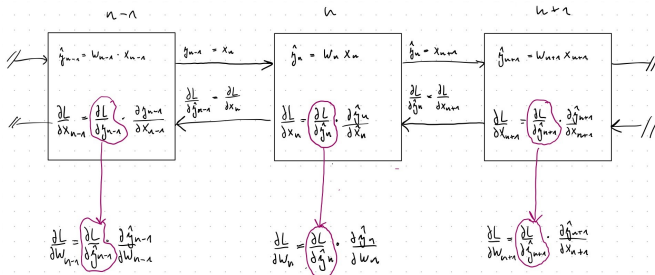
$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta \cdot \mathbf{E}_n \mathbf{X}^T \quad (3)$$

**Note:** Dynamic programming part of Backpropagation

- $\mathbf{E}_n$ : **error\_tensor** passed downward
- $\eta$ : learning rate

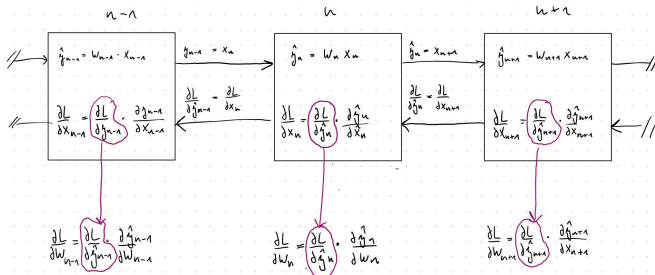
## But what is $E_n$ ?

- $L$  denotes the loss and



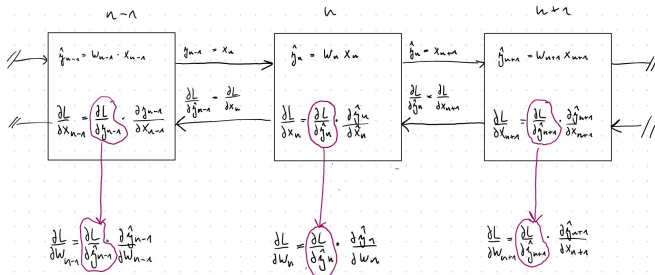
## But what is $E_n$ ?

- $L$  denotes the loss and
- $E_n$  is  $\frac{\partial L}{\partial \hat{y}_n}$  of a layer  $n$  (center box down below in purple).



## But what is $E_n$ ?

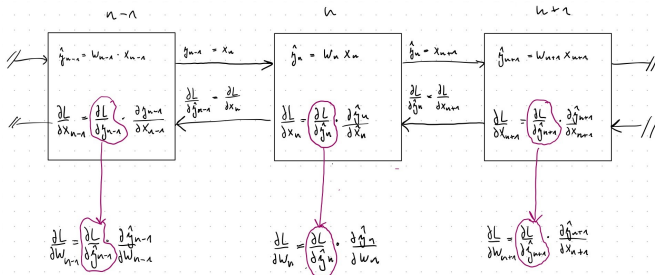
- $L$  denotes the loss and
- $E_n$  is  $\frac{\partial L}{\partial \hat{y}_n}$  of a layer  $n$  (center box down below in purple).
- When backwarding, it is used to compute  $\frac{\partial L}{\partial x_n}$





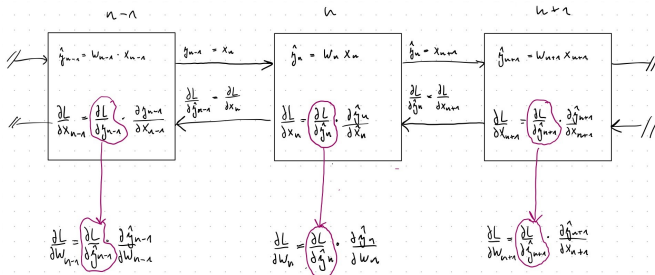
## But what is $E_n$ ?

- $L$  denotes the loss and
- $E_n$  is  $\frac{\partial L}{\partial \hat{y}_n}$  of a layer  $n$  (center box down below in purple).
- When backwarding, it is used to compute  $\frac{\partial L}{\partial x_n}$
- which is  $E_{n-1} = \frac{\partial L}{\partial \hat{y}_{n-1}}$  of the next upper layer  $n - 1$



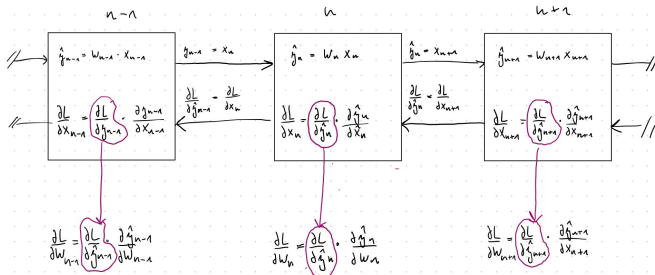
## But what is $E_n$ ?

- $L$  denotes the loss and
- $E_n$  is  $\frac{\partial L}{\partial \hat{y}_n}$  of a layer  $n$  (center box down below in purple).
- When backwarding, it is used to compute  $\frac{\partial L}{\partial x_n}$
- which is  $E_{n-1} = \frac{\partial L}{\partial \hat{y}_{n-1}}$  of the next upper layer  $n - 1$
- because the output of the layer  $n - 1$  is the input of layer  $n$ :  $\hat{y}_{n-1} = x_n$ .



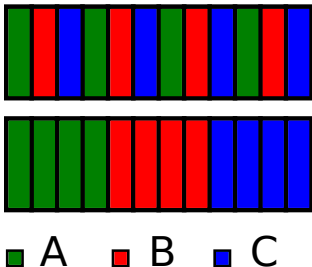
## But what is $E_n$ ?

- $L$  denotes the loss and
- $E_n$  is  $\frac{\partial L}{\partial \hat{y}_n}$  of a layer  $n$  (center box down below in purple).
- When backwarding, it is used to compute  $\frac{\partial L}{\partial x_n}$
- which is  $E_{n-1} = \frac{\partial L}{\partial \hat{y}_{n-1}}$  of the next upper layer  $n - 1$
- because the output of the layer  $n - 1$  is the input of layer  $n$ :  $\hat{y}_{n-1} = x_n$ .
- Thus  $\frac{\partial L}{\partial \hat{y}_{n-1}} = \frac{\partial L}{\partial x_n}$ . This is **Backpropagation**!

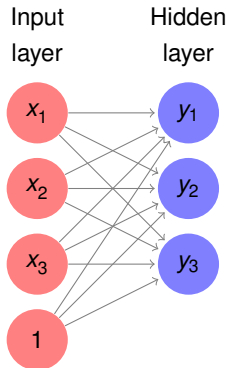


## Memory Layout

- We don't want to have  $X[:, 0]$  but  $X[0]$  to access the batch
- We want the batch size to be the outermost loop  
→ We have to adjust our formulas for the implementation
- We achieve it by transposition!



## Forward - Our Memory Layout



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

$$(\mathbf{W}\mathbf{X})^T = \hat{\mathbf{Y}}^T \quad (4)$$

$$\mathbf{X}^T \mathbf{W}^T = \hat{\mathbf{Y}}^T \quad (5)$$

## Forward - Our Memory Layout

We transposed our equations

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

$$\mathbf{X}^T \mathbf{W}^T = \hat{\mathbf{Y}}^T \quad (7)$$

but to benefit in our code from this new layout, we need to store our variables also in the transposed version. To differentiate the new and the old layout, the transposed versions of  $\mathbf{X}$ ,  $\mathbf{W}$ ,  $\mathbf{E}$  and  $\hat{\mathbf{Y}}$  are now denoted with primes:

$$\mathbf{X}' = \mathbf{X}^T, \mathbf{W}' = \mathbf{W}^T, \hat{\mathbf{Y}}' = \hat{\mathbf{Y}}^T \quad (8)$$

E.g. your python variable for the weights is now  $\mathbf{W}'$ , so we store our variables already in the transposed layout and compute everything in the new layout, like the forward pass:

$$\mathbf{X}' \mathbf{W}' = \hat{\mathbf{Y}}' \quad (9)$$

## Backward - Our Memory Layout

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

$$\mathbf{E}'_{n-1} = \mathbf{E}'_n \mathbf{W}'^T \quad (10)$$

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

$$\mathbf{W}'^{t+1} = \mathbf{W}'^t - \eta \cdot \mathbf{X}'^T \mathbf{E}'_n \quad (11)$$

**Note:** Dynamic programming part of Backpropagation

- $\mathbf{E}'_n$ : **error\_tensor** passed downward
- $\mathbf{E}'_n$  has always the same shape as  $\mathbf{Y}$
- $\mathbf{E}'_{n-1}$  has always the same shape as  $\mathbf{X}$
- $\eta$ : learning rate



FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Basic Optimization





## SGD

- In order to perform the aforementioned weight update we make use of a dedicated optimizer.
- In the first exercise we implement the Stochastic Gradient Descent Algorithm

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

where  $\eta$  denotes the learning rate.

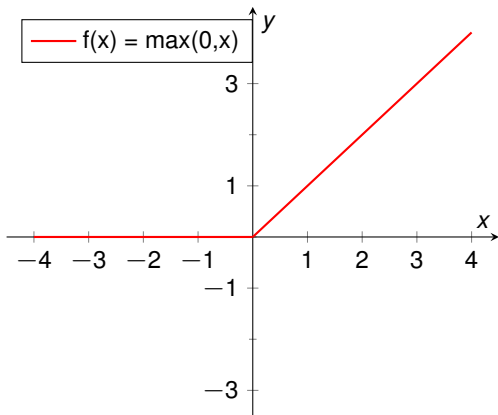


FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# ReLU Activation Function



# Forward



## Backward

**ReLU is not continuously differentiable!**

## Backward

# ReLU is not continuously differentiable!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \quad (12)$$

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

## Backward

# ReLU is not continuously differentiable!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \quad (12)$$

**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!

$$e_{n-1} = \begin{cases} 0 & \text{if } x \leq 0 \\ e_n & \text{else} \end{cases} \quad (12)$$

**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  $\underbrace{\frac{\partial L}{\partial \hat{\mathbf{y}}}}_{\mathbf{E}} \cdot \underbrace{\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{x}}}_{\text{ReLU}}$



FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# 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  $B$ :

$$\hat{y}_k = \frac{\exp(x_k)}{\sum_{j=1}^N \exp(x_j)} \quad (13)$$

## 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:

$$\mathbf{E}_{n-1} = \hat{y} \left( \mathbf{E}_n - \sum_{j=1}^N \mathbf{E}_{n,j} \hat{y}_j \right) \quad (14)$$

## Backward

- Compute for every element of the batch:

$$\mathbf{E}_{n-1} = \hat{y} \left( \mathbf{E}_n - \sum_{j=1}^N \mathbf{E}_{n,j} \hat{y}_j \right) \quad (14)$$

- All operations are element-wise

## Backward

- Compute for every element of the batch:

$$\mathbf{E}_{n-1} = \hat{y} \left( \mathbf{E}_n - \sum_{j=1}^N \mathbf{E}_{n,j} \hat{y}_j \right) \quad (14)$$

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



FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Cross Entropy Loss



## Forward

$$loss = \sum_{b=1}^B -\ln(\hat{y}_k + \epsilon) \text{ where } y_k = 1 \quad (15)$$

- $\epsilon$  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

$$loss = \sum_{b=1}^B -\ln(\hat{y}_k + \epsilon) \text{ where } y_k = 1 \quad (15)$$

- $\epsilon$  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 Cross Entropy Loss requires predictions to be greater than 0,
- thus the Cross Entropy Loss works most stable with SoftMax predictions.

## Backward

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \quad (16)$$

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

## Backward

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \quad (16)$$

- $\epsilon$  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  $\mathbf{E}$ .  
→ it's the starting point of the recursive computation of gradients.



Thanks for listening.  
**Any questions?**