FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Neural Networks

Z. Yang, L. Rist, M. Nau, S. Jaganathan, C. Liu, N. Maul, L. Folle, M. Zinnen, K. Packhäuser, J. Wolf, E. Wittmann, W. Karbole, N. Panchi, M. Reimann, J. Yao, V. Rabas, P. Sharma
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
October 16, 2021

Flexibility vs. Abstraction

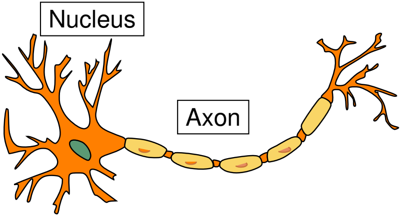
Low level → High level

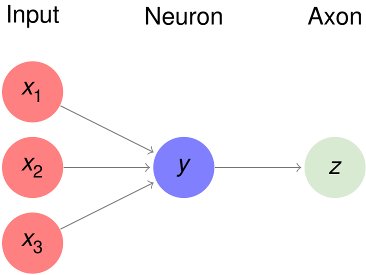
- Linear Algebra operations
- Bare metal

- Compiles graphs of Tensor operations
- High flexibility

- Stacks together elementary layers
- Reduced flexibility

Artifical Neural Networks





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

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

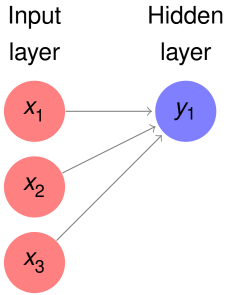
Neural Network

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

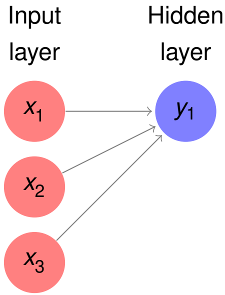
- is responsible for holding a **graph of layers**, whereas a "layer" represents a function (e.g. ReLU) or operation (e.g. convolution)
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**
- 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

Fully Connected Layer

Forward



Forward

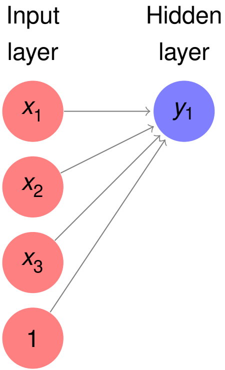


$$\begin{pmatrix} w_1 & \dots & w_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + w_{n+1} = \hat{y}$$

$$\mathbf{wx} + \underbrace{w_{n+1}}_{\text{bias}} = \hat{y}$$

- Including the bias into the weight matrix results in a single matrix multiplication

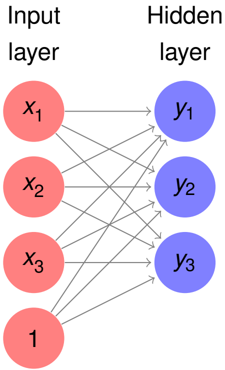
Forward



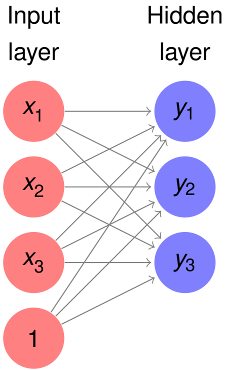
$$\begin{pmatrix} w_1 & \dots & w_n & w_{n+1} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \hat{y}$$

$$\mathbf{wx} = \hat{y}$$

Forward



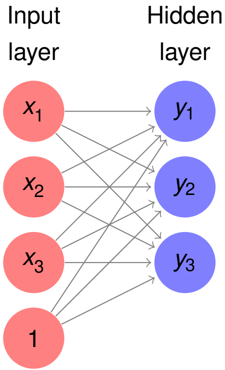
Forward



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

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

Forward



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

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

Backward

- Return gradient with respect to **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**:

- $\mathbf{E}_\mathbf{n}$: **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**:

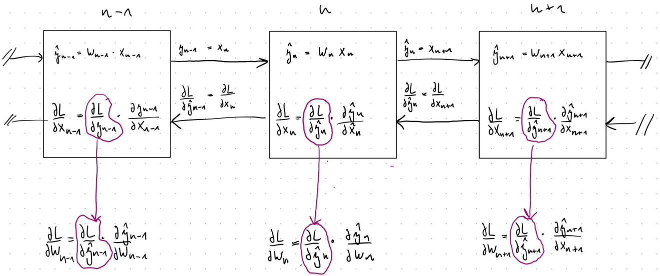
$$\mathbf{W}^{t+1} = \mathbf{W}^{t} - \eta \cdot \mathbf{E_n}\mathbf{X}^{T} \tag{3}$$

  Note: Dynamic programming part of Backpropagation

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

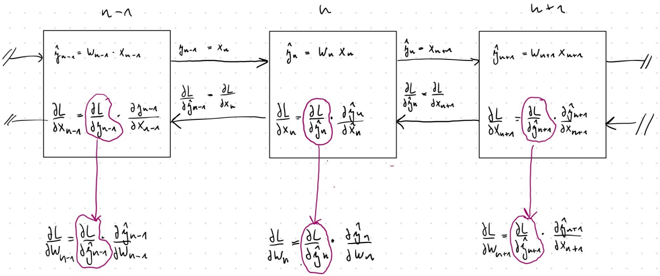
But what is E_n?

- *L* denotes the loss and



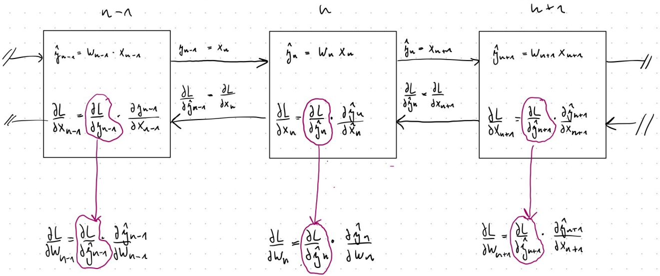
But what is E_n?

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



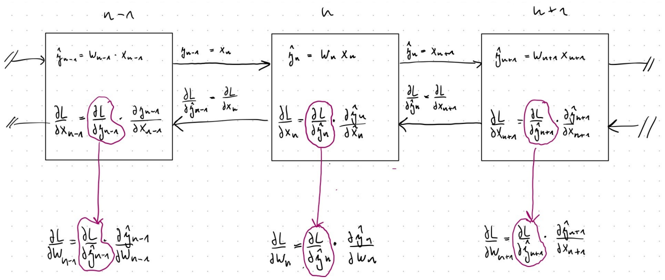
But what is $\mathbf{E_n}$?

- L denotes the loss and
- $\mathbf{E_n}$ 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 $\frac{\partial L}{\partial x_n}$



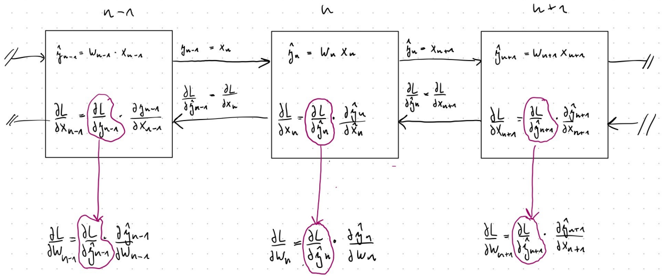
But what is $\mathbf{E_n}$?

- L denotes the loss and
- $\mathbf{E_n}$ 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 $\mathbf{E_{n-1}} = \frac{\partial L}{\partial \hat{y}_{n-1}}$ of the next upper layer $n-1$



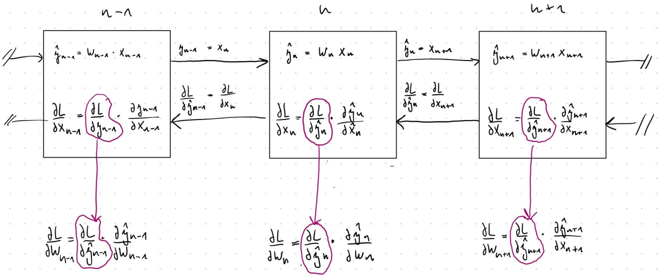
But what is $\mathbf{E_n}$?

- L denotes the loss and
- $\mathbf{E_n}$ 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 $\mathbf{E_{n-1}} = \frac{\partial L}{\partial \hat{y}_{n-1}}$ of the next upper layer $n-1$
- because the output of the layer $n-1$ is the input of layer n: $\hat{y}_{n-1} = x_n$.



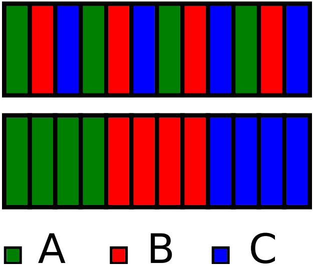
But what is $\mathbf{E_n}$?

- L denotes the loss and
- $\mathbf{E_n}$ 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 $\mathbf{E_{n-1}} = \frac{\partial L}{\partial \hat{y}_{n-1}}$ 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**!

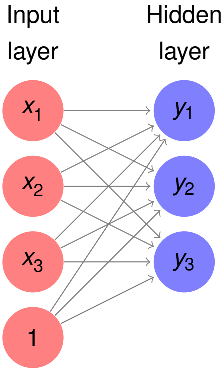


Memory Layout

- 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
  $\rightarrow$ We have to adjust our formulas for the implementation
- We achieve it by transposition!



Forward - Our Memory Layout



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

$$(\mathbf{WX})^{\mathbf{T}} = \hat{\mathbf{Y}}^{\mathbf{T}} \tag{4}$$

$$\mathbf{X}^{\mathbf{T}}\mathbf{W}^{\mathbf{T}} = \hat{\mathbf{Y}}^{\mathbf{T}} \tag{5}$$

Forward - Our Memory Layout

We transposed our equations

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

$$\mathbf{X}^{\mathbf{T}}\mathbf{W}^{\mathbf{T}} = \hat{\mathbf{Y}}^{\mathbf{T}} \tag{7}$$

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:

$$\mathbf{X}' = \mathbf{X}^{\mathbf{T}},\ \mathbf{W}' = \mathbf{W}^{\mathbf{T}},\ \mathbf{E}' = \mathbf{E}^{\mathbf{T}},\ \hat{\mathbf{Y}}' = \hat{\mathbf{Y}}^{\mathbf{T}} \tag{8}$$

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:

$$\mathbf{X}'\mathbf{W}' = \hat{\mathbf{Y}}' \tag{9}$$

Backward - Our Memory Layout

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

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

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

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

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 $\mathbf{X}$
- η: learning rate

Basic Optimization

SGD

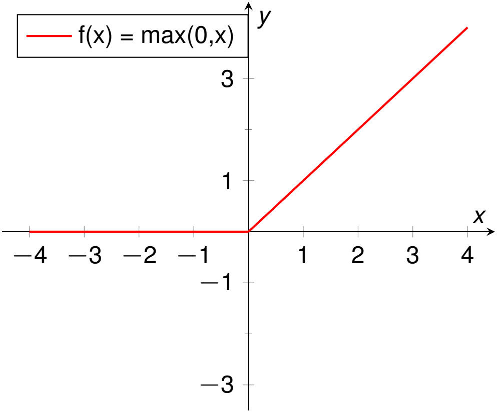
- In order to perform the aforementioned weight update we make use of a dedicated optimizer.
- In the first exercise we implement the Stochastic Gradient Descent Algorithm

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

where η denotes the learning rate.

ReLU Activation Function

Forward



Backward

ReLU is not continuously differentiable!

Backward

ReLU is not continuously differentiable!

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

Note: DP part of Backpropagation yet again

Backward

ReLU is not continuously differentiable!

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

Note: DP part of Backpropagation yet again

- The scalar *e* is because activation functions operate elementwise on **E**

Backward

ReLU is not continuously differentiable!

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

Note: DP part of Backpropagation yet again

- The scalar e is because activation functions operate elementwise on **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 y**: $\begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix}$

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

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

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:

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

Backward

- Compute for every element of the batch:

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

- All operations are element-wise

Backward

- Compute for every element of the batch:

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

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

Cross Entropy Loss

Forward

$$loss = \sum_{b=1}^{B} - \ln\left(\hat{y}_k + \epsilon\right) \textbf{ where } y_k = 1 \tag{15}$$

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

Forward

$$loss = \sum_{b=1}^{B} - \ln\left(\hat{y}_k + \epsilon\right) \ \textbf{where}\ y_k = 1 \tag{15}$$

- ϵ represents the smallest representable number. Take a look into *np.finfo.eps*
- ϵ 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

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \tag{16}$$

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

Backward

$$\mathbf{E}_n = -\frac{y}{\hat{y}} \tag{16}$$

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

Thanks for listening.

Any questions?