Neural Networks

Leonid Mill, Tobias Würfl, Katharina Breininger, Nishant Ravikumar, Sebastian Gündel, Mathis Hoffmann, Florian Thamm, Felix Denzinger
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
October 25, 2018

Flexibility vs. abstraction

Low level → High level

cuDNN

- Linear Algebra operations
- Bare metal

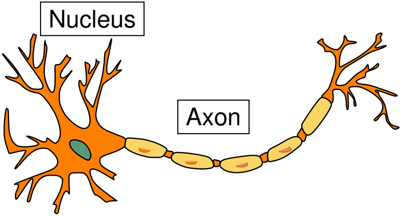
TensorFlow

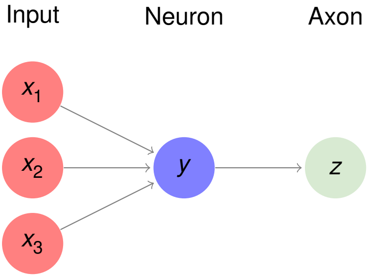
- Compiles graphs of Tensor operations
- High flexibility

K

- 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

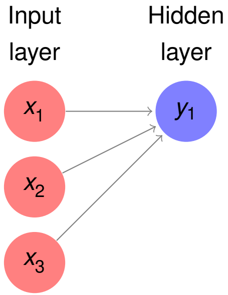
Neural Network

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

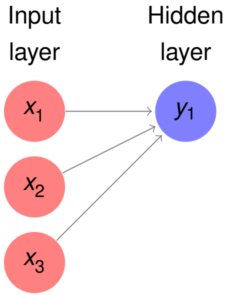
- is responsible for holding a **graph of layers**
  - we allow only extremely simple graphs
  - with a list of layers
  - and only one data source
  - and one loss function
- is responsible to hold **access to data**
- has **no explicit knowledge** about the graph of layers it contains
- **recursively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



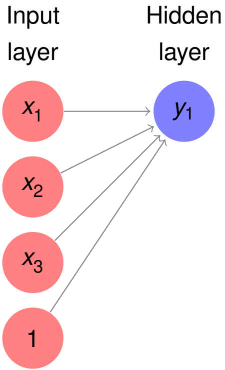
Forward



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

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

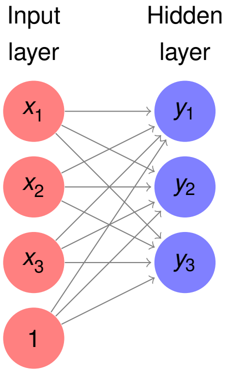
Forward



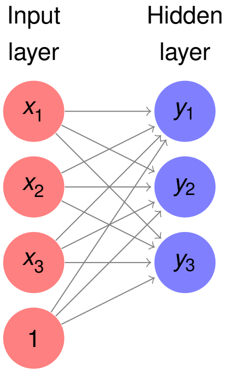
$$\begin{pmatrix} w_1 \\ \vdots \\ w_n \\ w_{n+1} \end{pmatrix}^T \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \hat{y}$$

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

Forward



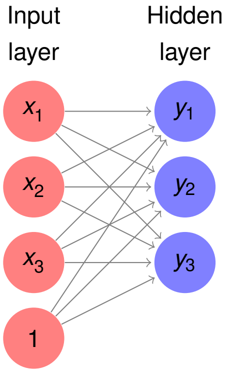
Forward



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

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

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 **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 $\mathbf{X}$:

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

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

- $\mathbf{E_n}$: **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**:

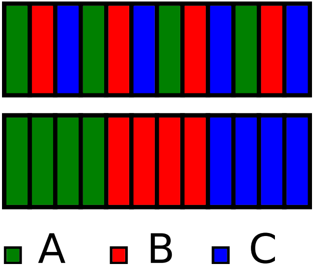
$$\mathbf{W}^{t+1} = \mathbf{W}^{t} - \delta \cdot \mathbf{E_n}\mathbf{X}^{T} \tag{3}$$

Note: Dynamic programming part of Backpropagation

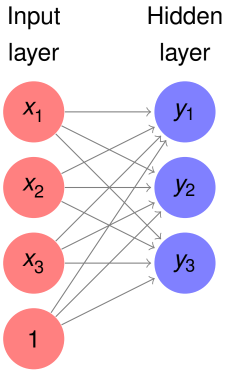
- $\mathbf{E_n}$: **error_tensor** passed downward
- δ : learning rate **delta** individual to this layer

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

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

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

Backward - Our Memory Layout

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

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

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

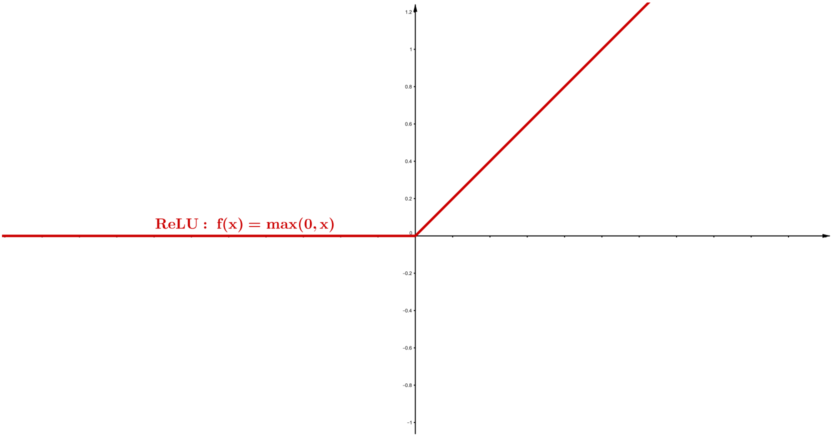
$$\mathbf{W}'^{t+1} = \mathbf{W}'^{t} - \delta \cdot \mathbf{X}'^{\mathbf{T}}\mathbf{E}'_{\mathbf{n}} \tag{8}$$

Note: Dynamic programming part of Backpropagation

- $\mathbf{E}'_{\mathbf{n}}$: **error_tensor** passed downward
- δ : learning rate **delta** individual to this layer

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 Loss" 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:

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

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:

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

- Loss:

$$loss = \sum_{b=1}^{B} -\log \hat{y}_k \textbf{ where } y_k = 1 \tag{11}$$

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

For every element of the batch:

$$e_k = \begin{cases} \hat{y}_k - 1 & \text{where } y_k = 1 \\ \hat{y}_k & \text{else} \end{cases} \tag{12}$$

Backward

For every element of the batch:

$$e_k = \begin{cases} \hat{y}_k - 1 & \text{where } y_k = 1 \\ \hat{y}_k & \text{else} \end{cases} \tag{12}$$

- We simply decrease the activation of the correct class
- And increase all others

Backward

For every element of the batch:

$$e_k = \begin{cases} \hat{y}_k - 1 & \text{where } y_k = 1 \\ \hat{y}_k & \text{else} \end{cases} \tag{12}$$

- We simply decrease the activation of the correct class
- And increase all others
- 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?