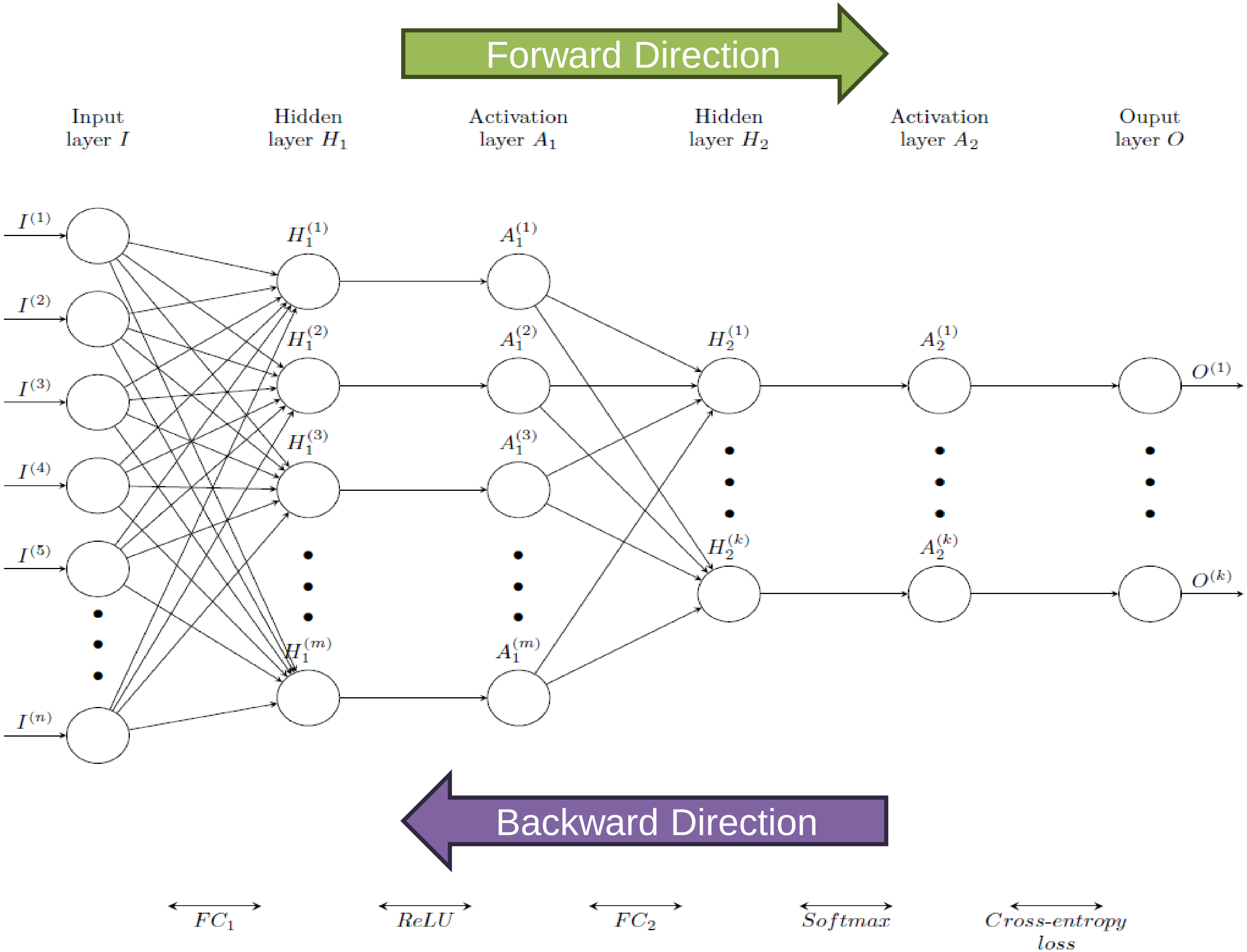
AdvPT Project

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Proposed FCNN Network



General Program Flow

Program workflow can be split into three parts:

1. **Initialization**
2. **Training**
3. **Evaluation**

Each part will be explained in the following, but also take a look at the PyTorch reference for more insights!

Initialization

Tasks at hand are:

- Parse input configuration file
- Load and preprocess dataset
  - Normalize pixel input data via linear mapping (cf. 1a)
  - One-hot encoding of label data (cf. 1b)
- Set up model and layers with
  - Fixed input size $n = 28^2 = 784$ and output size $k = 10$
  - Hidden layer size m specified in input configuration file
- Initialize weights and biases of layers <u>randomly</u>
  - Fix the random seed for reproducibility
  - **Avoid zero-value initialization** (*Problem*: exploding gradients)
  - ➢ Use other initialization techniques for better network convergence
    - Uniform distribution with negative values, e.g. in range [-1/n, 1/n] (*Problem*: vanishing gradients)
    - Better: Kaiming or Xavier initialization

Training

Repeatedly process input data (repititions are called epochs):

- Feed input data in forward direction to obtain prediction
- Calculate errors through a loss function
- Traverse network in backward direction and compute error tensors used for optimizing the trainable parameters

Pseudo-Code

```
- Repeat for a number of epochs
    - For each batch/sample in training dataset
        1. Perform forward pass
        2. Compute loss (and potentially log it to console)
        3. Perform backward pass
        4. Optimization: update trainable parameters
```

Evaluation

Trained model is tested on unseen data to measure generalizability to new inputs

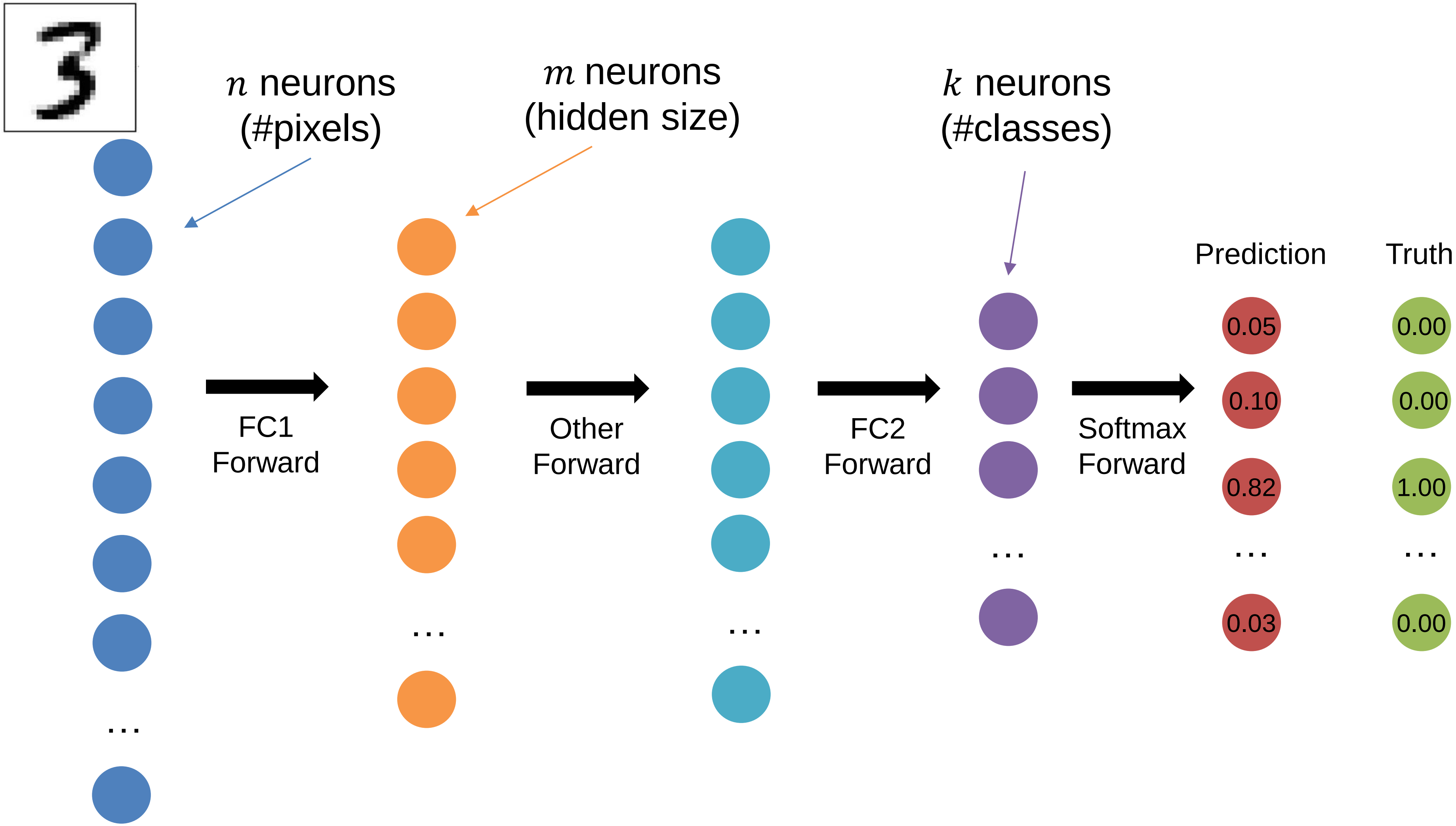
- Feed input data in forward direction to obtain prediction
- Compare predictions to ground truth (true label value) and print using the prediction logger for review
- Determine accuracy metric (i.e. percentage of correct predictions)

Pseudo-Code

```
- For each batch/sample in testing dataset
    1. Perform forward pass
    2. Compare predictions to true label values and log
    3. Compute accuracy
```

Forward Pass

Idea: Feed input data through network to obtain prediction that can be compared to ground truth



Loss

Quantifies difference between predicted and true class probability distributions

- Shows how well the network performs at prediction task
- Quantity to be minimized ➔ Optimizer often make use of loss gradients

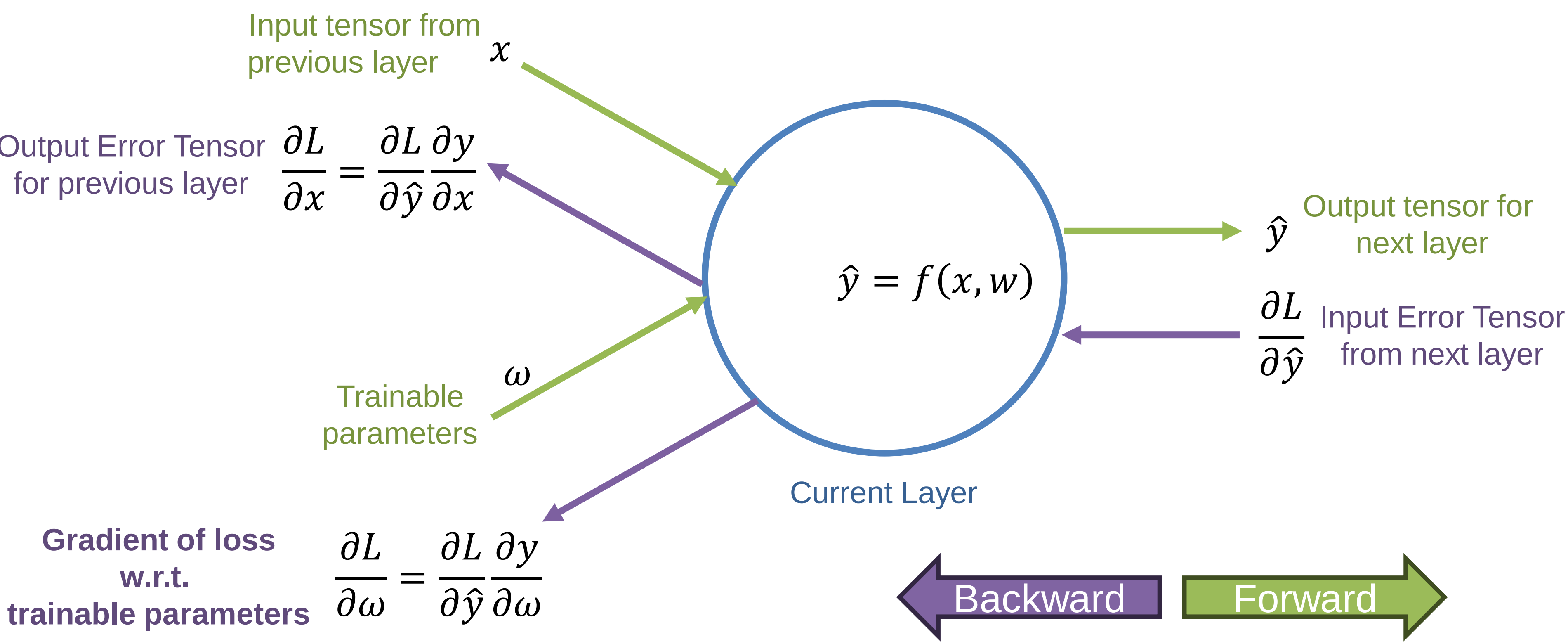
Proposed loss function: **cross-entropy loss**

$$L(y, \hat{y}) = \sum_{i=1}^{k} y_i \log \widehat{y_i}$$

- k: number of labels (here: 10)
- y_i: true class probability (one-hot encoded)
- $\widehat{y_i}$: predicted class probability output by previous softmax layer

Backward Pass

- Idea: Compute gradient of loss w.r.t. trainable parameters ω for optimization realized by recursive application of chain rule
- **Remember to store input tensors in your layer classes**



Optimizer

Adjust trainable parameters (weights and biases) to minimize loss

➔ Essential for improving performance of networks during training

Proposed optimizer: **stochastic gradient descent (SGD)**

$$\omega_{new} = \omega - \eta \frac{\partial L}{\partial \omega}$$

- η: Learning rate, provided in configuration file
- $\frac{\partial L}{\partial \omega}$: Gradient of loss w.r.t. ω. Computed in backward pass

Layer Description

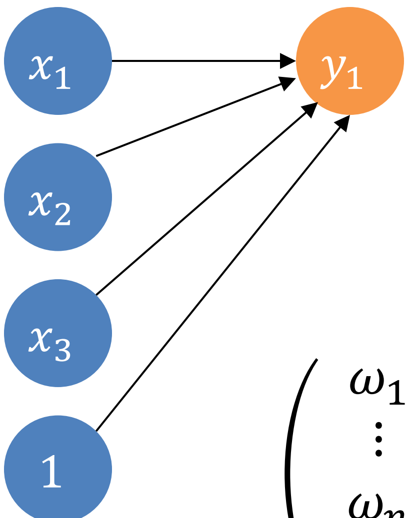
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Fully Connected Layer

Forward pass with n input neurons and m output neurons (no batching)

For single neuron

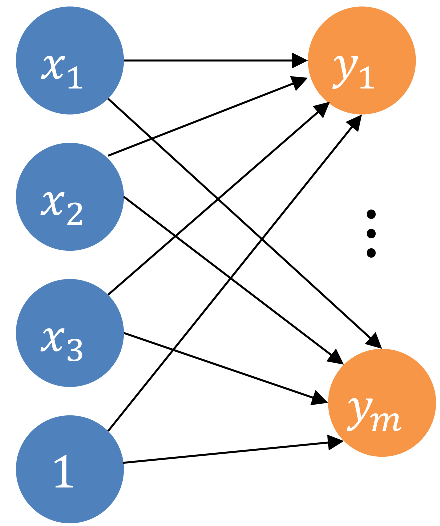


$$\begin{pmatrix} \omega_1 \\ \vdots \\ \omega_n \\ \omega_{n+1} \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \hat{y}$$

$$\longrightarrow \quad \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} = \hat{y}$$

- $\boldsymbol{w}^{\boldsymbol{T}}$: Vector with train params
- $\omega_1 \ldots \omega_n$: weights
- ω_{n+1}: bias
- $\boldsymbol{x}$: input tensor

For all neurons



$$\begin{pmatrix} \omega_{1,1} & \cdots & \omega_{1,m} \\ \vdots & \ddots & \vdots \\ \omega_{n,1} & \cdots & \omega_{n,m} \\ \omega_{n+1,1} & \cdots & \omega_{n+1,m} \end{pmatrix}^{T} \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}$$

$$\longrightarrow \quad \boldsymbol{W} \boldsymbol{x} = \hat{y}$$

Fully Connected Layer

Forward pass with batching

$$\begin{array}{ccc} M \times (N+1) & (N+1) \times B & M \times B \\ \begin{pmatrix} \omega_{1,1} & \cdots & \omega_{1,m} \\ \vdots & \ddots & \vdots \\ \omega_{n,1} & \cdots & \omega_{n,m} \\ \omega_{n+1,1} & \cdots & \omega_{n+1,m} \end{pmatrix}^T & \begin{pmatrix} x_{1,1} & \cdots & x_{1,b} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,b} \\ 1 & \cdots & 1 \end{pmatrix} = & \begin{pmatrix} y_{1,1} & \cdots & y_{1,b} \\ \vdots & \ddots & \vdots \\ y_{m,1} & \cdots & y_{m,b} \end{pmatrix} \end{array}$$

$$\longrightarrow \boldsymbol{W}\boldsymbol{X} = \hat{\boldsymbol{Y}}$$

But: inefficient memory layout causes strided data access since values are stored row-wise in C/C++

Fully Connected Layer

Backward pass:

- Return gradient with respect to (w.r.t.) $\boldsymbol{X}$:

$$\boldsymbol{E_{n-1}} = \boldsymbol{W^T E_n}$$

- Update $\boldsymbol{W}$ using gradient w.r.t. $\boldsymbol{W}$ using SGD:

$$\boldsymbol{W_{new}} = \boldsymbol{W} - \eta \boldsymbol{E_n X^T}$$

- $\boldsymbol{E_n}$: Error tensor received from previous layer (backward direction)
- $\boldsymbol{E_{n-1}}$: Error tensor passed to next layer (backward direction)
- η: Learning rate

Fully Connected Layer

Forward Pass *with improved memory layout* for batching caused adjustment to formulas

$$\begin{array}{ccc} B \times (N+1) & (N+1) \times M & B \times M \\ \begin{pmatrix} x_{1,1} & \cdots & x_{1,b} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,b} \\ 1 & \cdots & 1 \end{pmatrix}^T & \begin{pmatrix} \omega_{1,1} & \cdots & \omega_{1,m} \\ \vdots & \ddots & \vdots \\ \omega_{n,1} & \cdots & \omega_{n,m} \\ \omega_{n+1,1} & \cdots & \omega_{n+1,m} \end{pmatrix} = & \begin{pmatrix} y_{1,1} & \cdots & y_{1,m} \\ \vdots & \ddots & \vdots \\ y_{b,1} & \cdots & y_{b,m} \end{pmatrix} \end{array}$$

$$\longrightarrow \boldsymbol{X'W'} = \boldsymbol{\hat{Y}'}$$

With:

- $\boldsymbol{X'} = \boldsymbol{X^T}$
- $\boldsymbol{W'} = \boldsymbol{W^T}$
- $\boldsymbol{\hat{Y}'} = \boldsymbol{\hat{Y}^T}$
- $\boldsymbol{\hat{Y}^T} = (\boldsymbol{WX})^{\boldsymbol{T}} = \boldsymbol{X^T W^T}$

Fully Connected Layer

Backward pass *with improved memory layout*

- Return gradient with respect to (w.r.t.) $\boldsymbol{X}$:

$$\boldsymbol{E}'_{\boldsymbol{n-1}} = \boldsymbol{E}'_{\boldsymbol{n}} \boldsymbol{W}'^{\boldsymbol{T}}$$

- Update $\boldsymbol{W}$ using gradient w.r.t. $\boldsymbol{W}$ using SGD:

$$\boldsymbol{W}_{\boldsymbol{new}} = \boldsymbol{W} - \eta \boldsymbol{X}'^{\boldsymbol{T}} \boldsymbol{E}'_{\boldsymbol{n}}$$

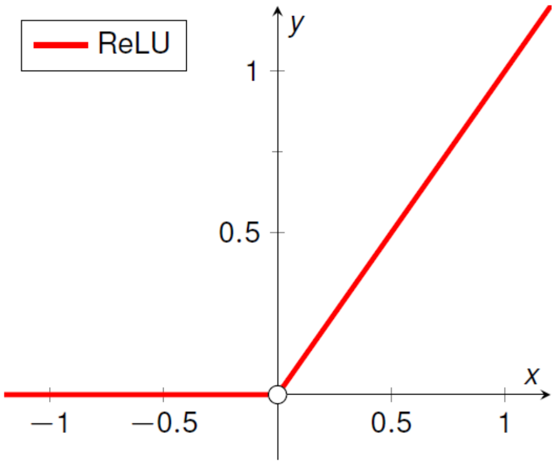
- $\boldsymbol{E}'_{\boldsymbol{n}}$: Error tensor received from previous layer (backward direction)
- $\boldsymbol{E}'_{\boldsymbol{n-1}}$: Error tensor passed to next layer (backward direction)
- η: Learning rate

ReLU Activation Function

Rectified Linear Unit (ReLU), introduces non-linearity to the model

Forward Pass

- Governing formula: $\boldsymbol{f(x) = max(0, x)}$



From [1]

Backward Pass

- ReLU is not continuously differentiable
- Requires information about input tensor from forward pass
- Note: Activation function operate elementwise for each x ($\odot$ operator)

$$E_{n-1} = E_n \odot \begin{cases} 0, \text{ if } x \leq 0 \\ 1, \text{ else} \end{cases}$$

Softmax Activation Function

Forward Pass encodes input neurons x to a probability function

$$\hat{y} = \frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{k} \mathrm{e}^{x_j}}$$

Properties:

- k: Number of classes (here: 10)
- $\sum_{i=1}^{k} \hat{\boldsymbol{y}}_i = 1$
- $\hat{\boldsymbol{y}}_i \geq 0$
- If $x_i > 0$, exponent might become very large ➔ x_i can be shifted to increase numerical stability by using:

$$\tilde{x}_i = x_i - \max(\boldsymbol{x})$$

Softmax Activation Function

Backward Pass

$$E_{n-1} = \hat{y}\left(E_n - \sum_{i=1}^{k} E_{n,i}\,\hat{y}_i\right)$$

- Note again: elementwise operations
- Remember to store $\hat{y}_i$

Cross Entropy Loss Function

Forward Pass (cf. previous slide)

Backward Pass

$$E_n = \frac{-y}{\hat{y}}$$

- First layer in backward direction to compute error tensor
- $\hat{y}$: Predicted class probability
- y: True class probability

References

[1]: Andreas Meier, Activation Functions and Convolutional Neural Networks. Deep Learning Lecture Notes, 2020.

Thank you and good luck!