VL Deep Learning for Natural Language Processing

4. Neural Networks Recap

Prof. Dr. Ralf Krestel

AG Information Profiling and Retrieval

Lerning Goals for this Chapter

- Train a simple neural network in Python using Keras
- Understand gradient-based optimization
- Describe the components of deep neural networks
- Understand and apply the backpropagation algorithm

- Relevant chapters
  - P2, P3

Topics Today

1. **A First Neural Network**
2. Tensor-Based Operations
3. Gradient-Based Optimization
4. Backprop(agation Algorithm)

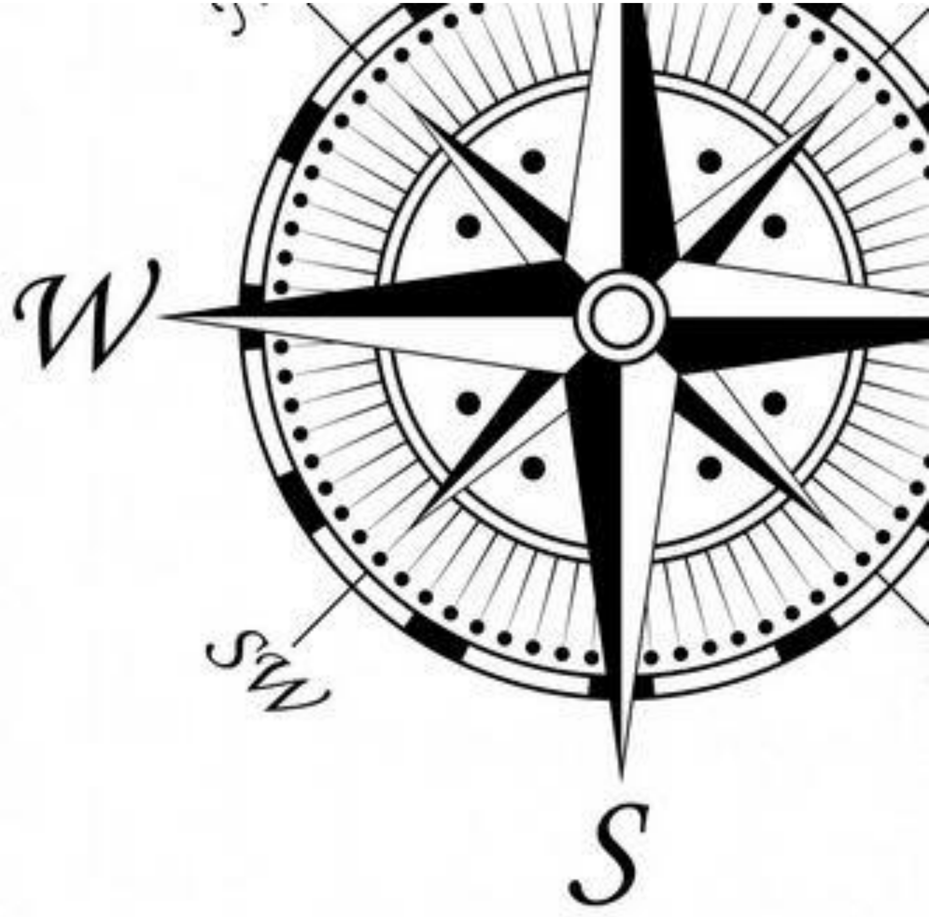


Image Classification Example I

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

- In machine learning, a **category** in a classification problem is called a **class**.
- Data points are called **samples**.
- The class associated with a specific sample is called a **label**.

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

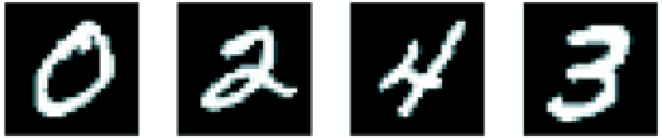
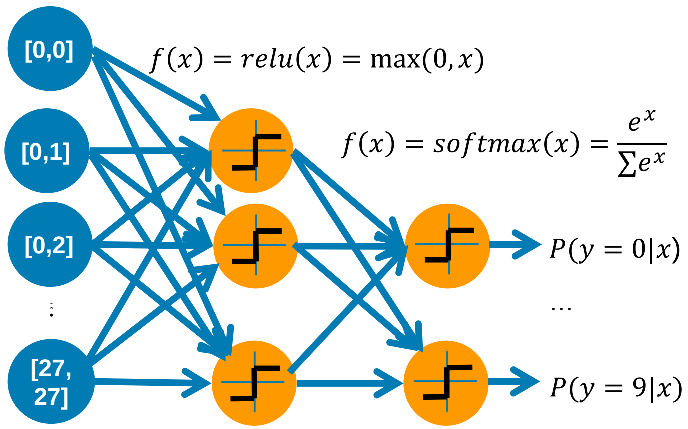


Image Classification Example II

```
from keras import models
from keras import layers
network = models.Sequential()
network.add(layers.Dense(512,activation='relu',input_shape=(28*28,)))
network.add(layers.Dense(10,activation='softmax'))
```



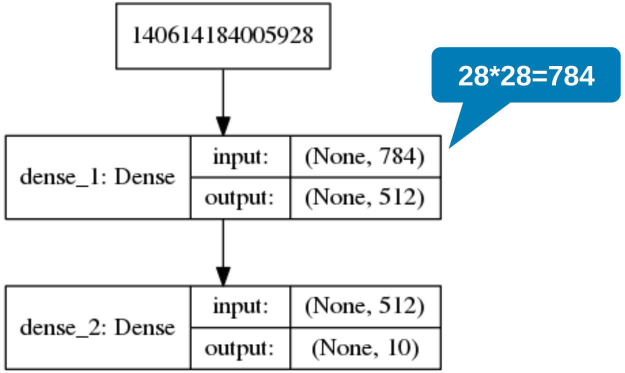


Image Classification Example III

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

- **Loss function**
  - Defines how the network measures improvement
- **Optimizer**
  - Defines how the network improves based on the input data and the loss function.
- **Success metric**
  - Defines how success for a given task is measured during training and testing
  - In this example: proportion of correctly classified images

Image Classification Example IV

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

- Transforming the data (reshaping)
- Scaling the data (scaling)

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

- Labels become categories

Image Classification Example III

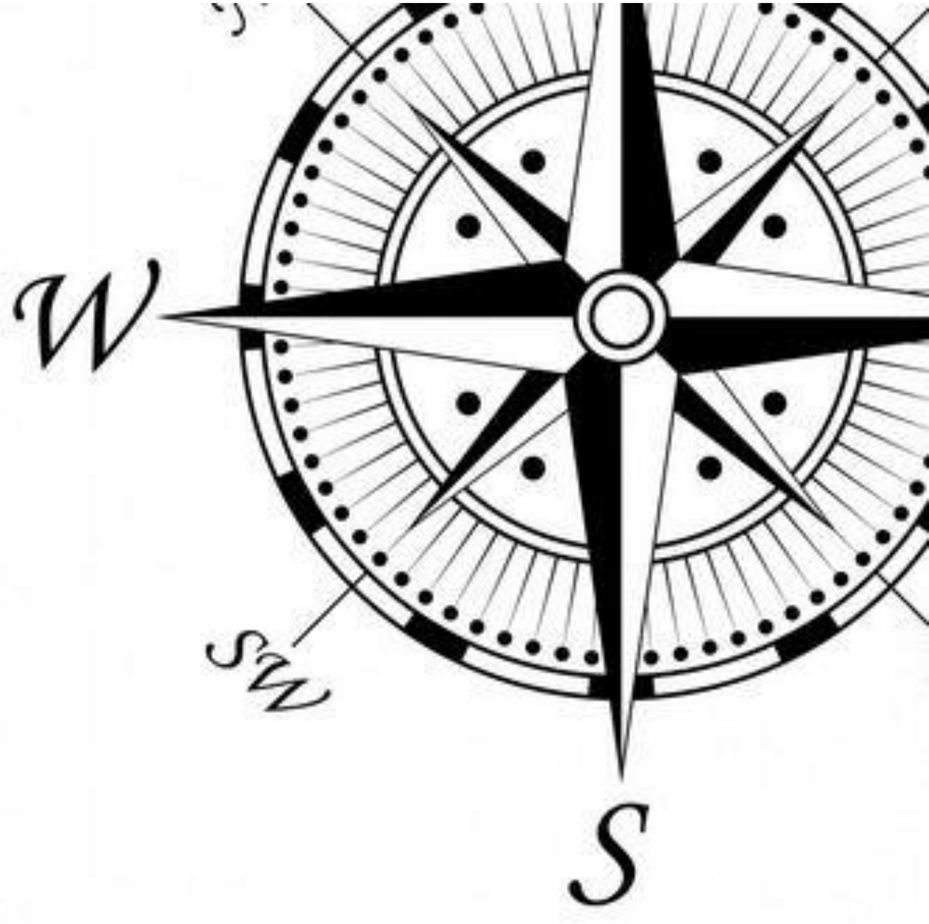
```
>>> network.fit(train_images,train_labels,epochs=5,batch_size=128)
Epoch 1/5
60000/60000 [==============] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

- Training the model (**fitting**)
- After 5 **epochs**: 0.989 accuracy

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

- Accuracy on test set significantly lower than on training set
  - -> **Overfitting**

Topics Today



1. A First Neural Network
2. **Tensor-Based Operations**
3. Gradient-Based Optimization
4. Backprop(agation Algorithm)

Tensor Operations

```
network.add(layers.Dense(512,activation='relu'))

output = relu(dot(W, input) + b)
```

$relu(x) = \max(0, x)$
$dot(x, y)$ = dot product (scalar product) of x and y
W = tensor (holding the weights)
input = tensor (holding the input data)
$x + y$ = addition between tensor and vector
b = vector (holding the bias terms)

Element-Wise Operations

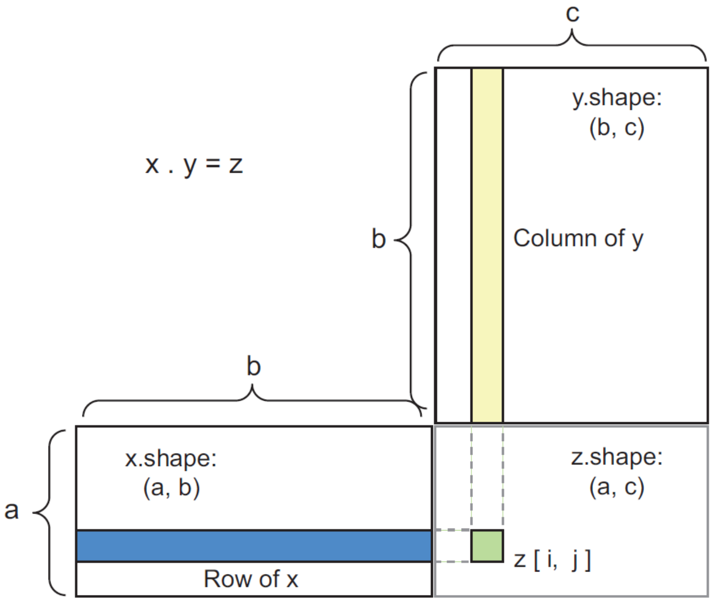
```
def naive_add(x, y):
        assert len(x.shape) == 2
        assert x.shape == y.shape
        x = x.copy()
        for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                        x[i, j] += y[i, j]
        return x
def naive_relu(x):
        assert len(x.shape) == 2
        x = x.copy()
        for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                        x[i, j] = max(x[i, j], 0)
        return x

import numpy as np
z = x + y
z = np.maximum(z, 0.)
```

Optimized implementations, e.g., in numpy

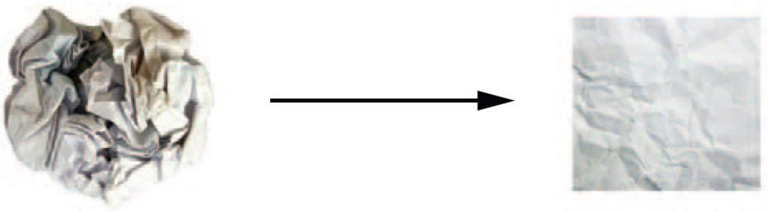
Dot Product

```
def naive_vector_dot(x, y):
  assert len(x.shape) == 1
  assert len(y.shape) == 1
  assert x.shape[0] == y.shape[0]
  z = 0.
  for i in range(x.shape[0]):
    z += x[i] * y[i]
  return z
def naive_matrix_dot(x, y):
  assert len(x.shape) == 2
  assert len(y.shape) == 2
  assert x.shape[1] == y.shape[0]
  z = np.zeros((x.shape[0], y.shape[1]))
  for i in range(x.shape[0]):
    for j in range(y.shape[1]):
        row_x = x[i, :]
        column_y = y[:, j]
        z[i,j] = naive_vector_dot(row_x, column_y)
  return z
import numpy as np
z = np.dot(x, y)
```



Geometric Interpretation

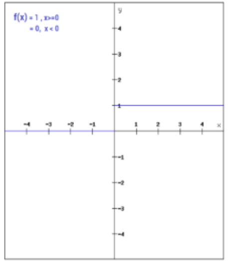
- Neural networks consist entirely of chains of tensor operations.
- All of these tensor operations are just geometric transformations of the input data.
- You can interpret a neural network as a very complex geometric transformation.
  - In a high-dimensional space
  - implemented via a long series of simple steps.



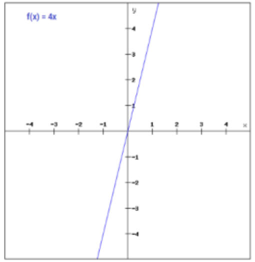
Activation Functions

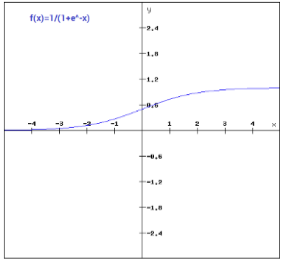
- Binary Step
- Linear
- Sigmoid
- Tanh
- ReLU
- Leaky ReLU
- Parameterised ReLU
- Exponential Linear Unit
- Swish
- Softmax

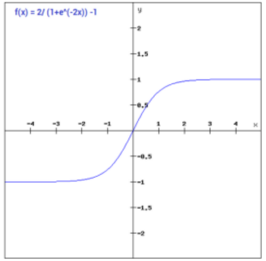
= 0, x<0

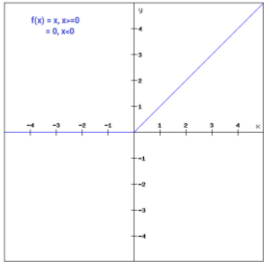


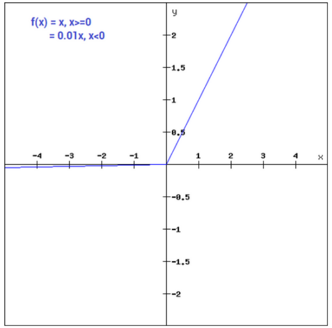
f(x)=ax

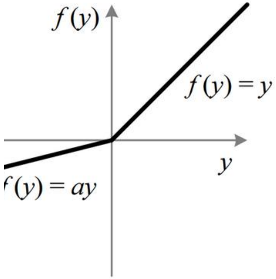


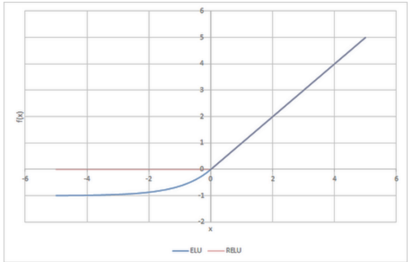


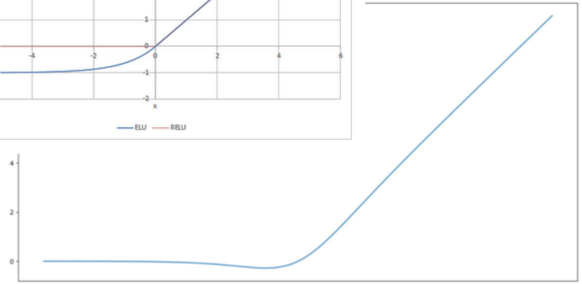












https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

Activation Functions

- $softmax(\boldsymbol{x}): \mathbb{R}^K \rightarrow \mathbb{R}^K$
  - Transforms unnormalized log probabilities over k classes to individual probabilities for exclusive classes

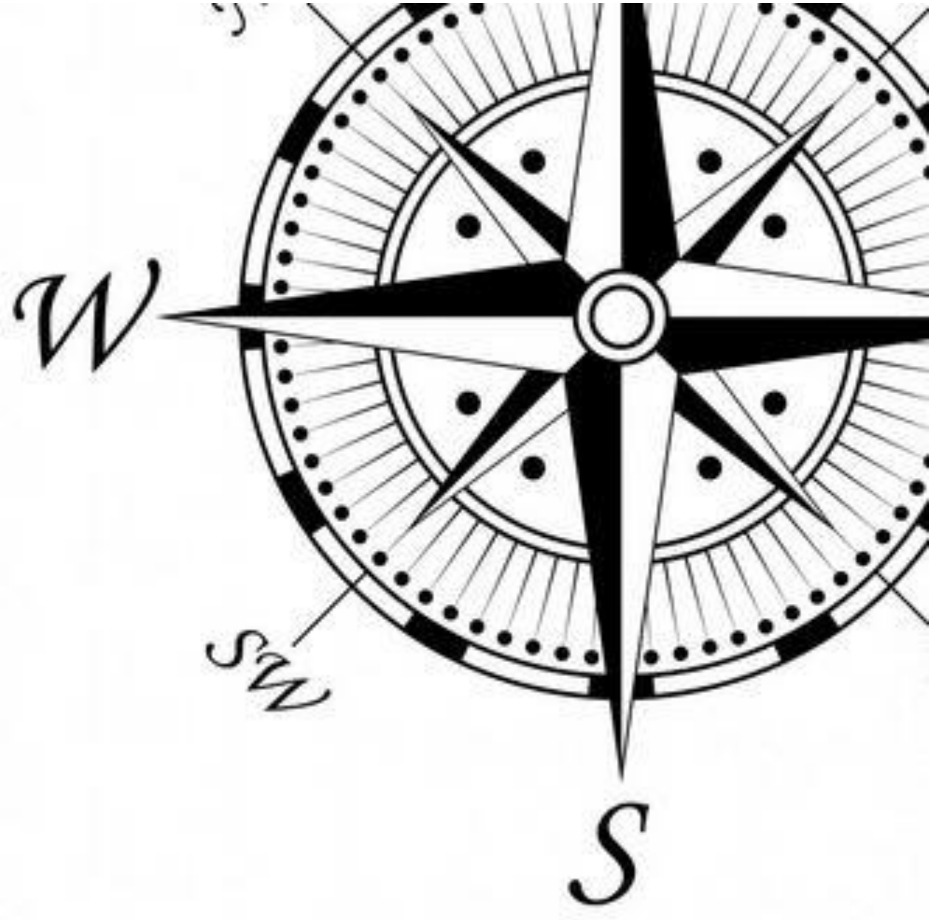
$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} = P(y = i|\boldsymbol{x})$$

  - Example: $\boldsymbol{x} = [2.0\ 1.0\ 0.1]^T \quad softmax(\boldsymbol{x}) = [0.7\ 0.2\ 0.1]^T$
- Binary step and linear functions are not suited for hidden layers in deep neural networks!
  - Why?
- Which activation functions (and how many ouput neurons) in output layer for

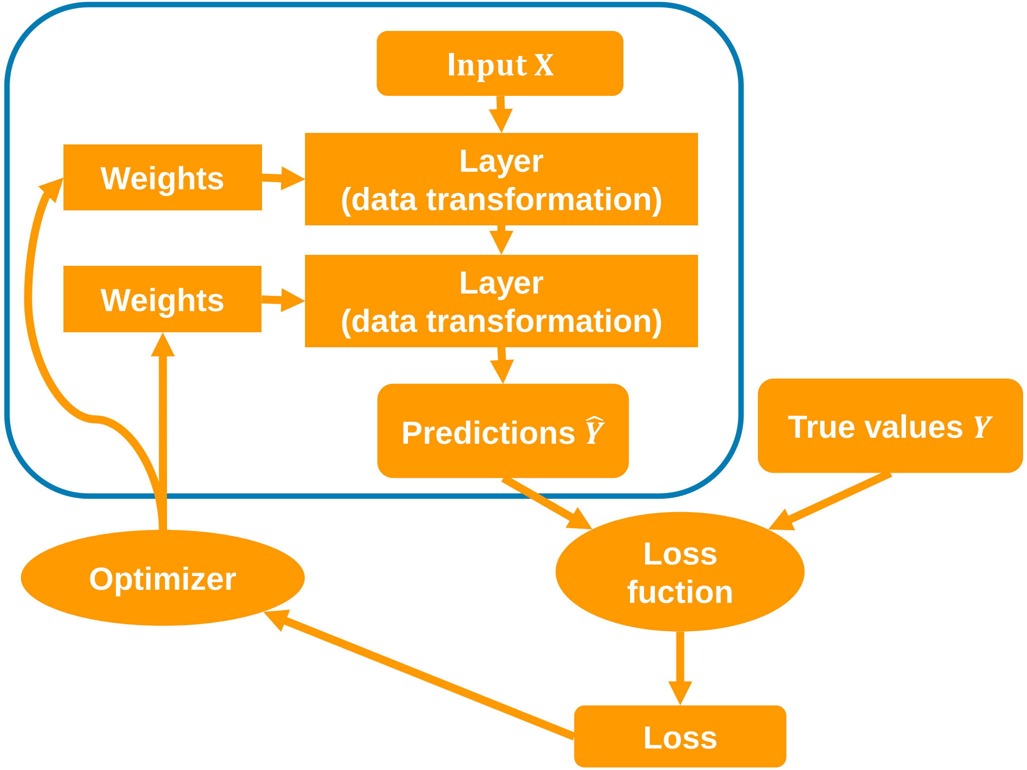
1. Regression problem
   - Input tensor $\boldsymbol{x} \rightarrow \mathbb{R}$
2. Binary classification
   - Input tensor $\boldsymbol{x} \rightarrow [0,1]$ (can also be $\mathrm{x} \rightarrow [-1,1]$)
3. (Single-label), multi-class classification
   - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector and $|y| = 1$
4. Multi-label, (multi-class) classification
   - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector

Topics Today

1. A First Neural Network
2. Tensor-Based Operations
3. **Gradient-Based Optimization**
4. Backprop(agation Algorithm)



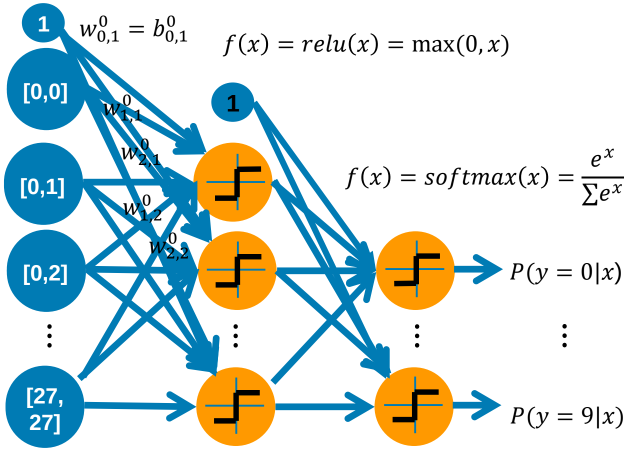
What Happens in a Neural Network?



What Happens in a Layer?

```
output = f(x)= f(dot(W, input) + b)
```

- W and b are tensors.
  - W are the **weights** or **parameters** of a layer and need to be trained / learned.
  - b can be merged into W by using a constant 1 as pseudo input
- At the beginning, weights are randomly initialized.
  - The network has not learned any meaningful representation or transormation and therefore no good mapping from input to output.



$$W^0 = \begin{bmatrix} w_{0,1}^0 & \cdots & w_{28,1}^0 \\ \vdots & \ddots & \vdots \\ w_{0,512}^0 & \cdots & w_{28,512}^0 \end{bmatrix}$$

How to Train a Network?

1. Randomly choose k training samples x (**mini batch**).
   – Alternative:
     - Batch (all training samples)
     - Stochastisic (one training sample)

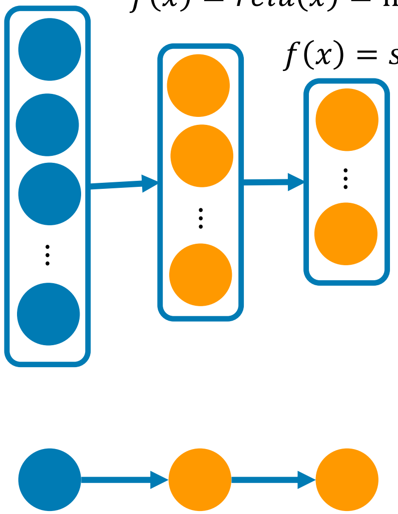
2. Compute the network‘s output $\hat{y}$ for input x.

3. Compute the **loss** of the network on the batch, i.e. the discrepancy between the predicition $\hat{y}$ and the actual value y

$$Loss = L(\hat{y}, y)$$

$$f(x) = relu(x) = \max(0, x)$$

$$f(x) = softmax(x) = \frac{e^x}{\sum e^x}$$



4. Update the weights in a way that reduces the loss a little

Loss Function

- Mean absolute error
  - $MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$
- Mean squared error
  - $MSE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|^2}{n}$
- Binary cross entropy
  - $BCE = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$
- Hinge loss
  - $Hinge = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
- Categorical cross entropy
  - $CE = -\sum_C \boldsymbol{y}_i \log(\hat{\boldsymbol{y}}_i)$
- Kullback Leibler divergence
  - $KL(P, Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}$

- Which loss functions for
1. Regression problem
   - Input tensor $\boldsymbol{x} \rightarrow \mathbb{R}$
2. Binary classification
   - Input tensor $\boldsymbol{x} \rightarrow [0,1]$ (can also be $\mathrm{x} \rightarrow [-1,1]$)
3. (Single-label), multi-class classification
   - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector and $|y| = 1$
4. Multi-label, (multi-class) classification
   - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector

How to Update the Weights I?

- Naive appraoch:

1. Choose a training **sample** randomly
2. Compute the **loss**
3. Choose a weight randomly
4. Update this weight **randomly**, keep all other weights fixed
5. Compute the **loss** again
   - If loss lower, keep weight and go to step 3.
   - If loss higher, go back to step 4.
   - If all weights are updated, go back to step 1.
   - If all samples were used for training, repeat whole process (i times)

- Way too **inefficient**!
  - Theoretically possible
  - Possible: Finding only **local minimum**

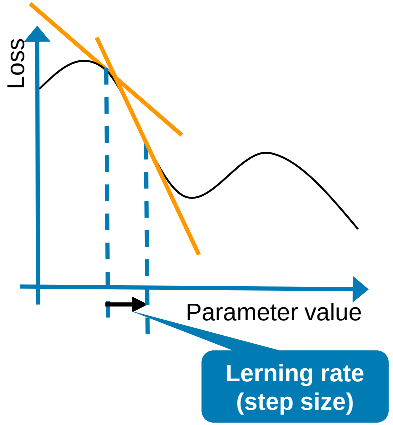
How to Update the Weights II?

- Analytical appraoch:
  1. Compute the derivative / gradient of the loss function $L(W)$
  2. Identify all W^* with $L'(W^*) = 0$
  3. Compute the loss for all W^*
  4. Select the W^* for which $L(W^*)$ is minimal
     - Globle minimum

- Not **efficiently** solvable!
  - If more than a few weights are involved
  - Typically: millions of weights!

Stochastic Gradient Descent (SGD)

- (true) **SGD**
  - Just like naive approach, only updates of weights not random
  - Updates are based on derivative / gradient
  - Stochastic, since samples are chosen randomly from training set



Batch and Mini-Batch SGD

- **Batch SGD**
  - Similar to SGD, but instead of using only one sample, compute loss on all training samples
    - Updates of weights much more accurate
    - Computation much more **expensive**
- **Mini-batch SGD**
  - Compromise between looking at all samples and only one sample
  - Simulaneously evaluating a small set of samples
    - Typically 8, 16, 32, 64, 128 or 256
- **Learning rate** is an important (hyper-) parameter
  - Variations:
    - Adaptive learning rate
      - Higher order derivatives (**momentum**)

The Backpropagation Algorithm

- SGD needs the derivative / gradient of the loss function for each weight
- Typically, a NN consists of many tensor operations

```
f(W1, W2, W3) = a(W1, b(W2, c(W3)))
```

  - For each single one it is easy to compute the gradient
- **Chain rule**:

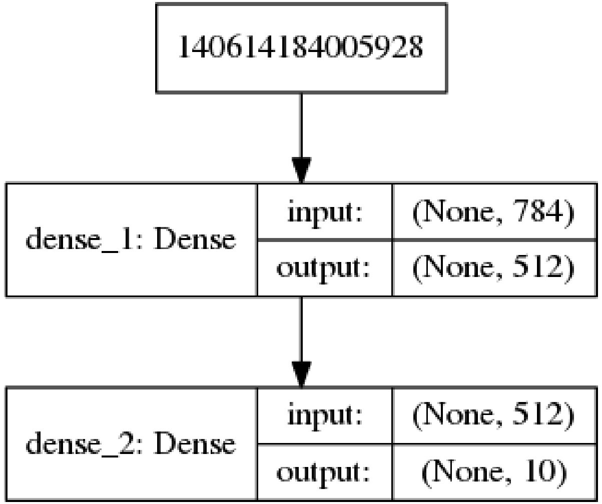
$$\mathrm{f} = u \circ v : V \rightarrow \mathbb{R}$$

$$(u \circ v)'(x_0) = u'\big(v(x_0)\big) \cdot v'(x_0)$$

- **Backpropagation Algoritm**
  - Application of chain rule to compute gradient of NN
  - Start with loss at the last (output) layer of the network and compute backwards the proportion that each weight contributed to this loss (backpropagation).
  - Implemented in Keras using symbolic differentiation
    - A gradient function for the chain of derivatives maps network parameter values (weights) to the respective gradients.

Runtime Estimation

- How many gradient updates?
- How large are the 2 weight matrices?

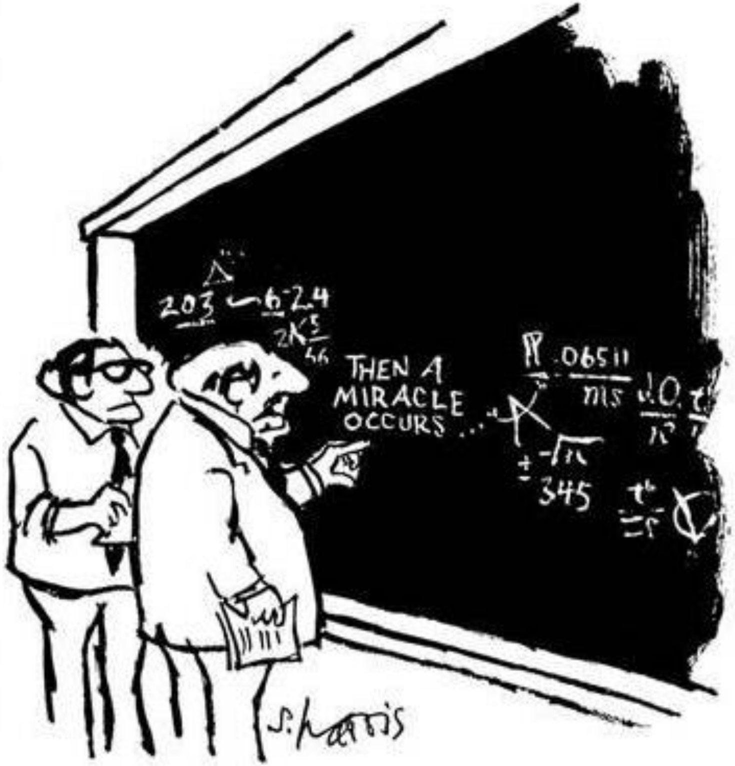



```
>>> train_images.shape
(60000, 28, 28)
>>>
network.fit(train_images,
train_labels, epochs=5,
batch_size=128)
```

Start 3 2 1 Ende

Topics Today

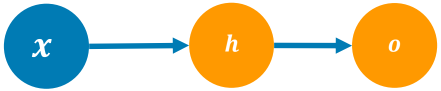
1. A First Neural Network
2. Tensor-Based Operations
3. Gradient-Based Optimization
4. **Backprop(agation Algorithm)**



"I think you should be more explicit here in step two."

Computation of Gradients

- **Forward pass**
  - For feedforward neural nets
    - Computation of $\hat{y}$ given input $\boldsymbol{x}$
  - During training: Additionally computation of
    - Error / Loss function $J(\boldsymbol{\theta})$
  - Batch processing possible
    - Simultaneously computing $J(\boldsymbol{\theta})$ for multiple input samples $\boldsymbol{X}$

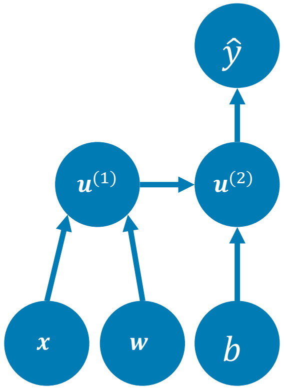


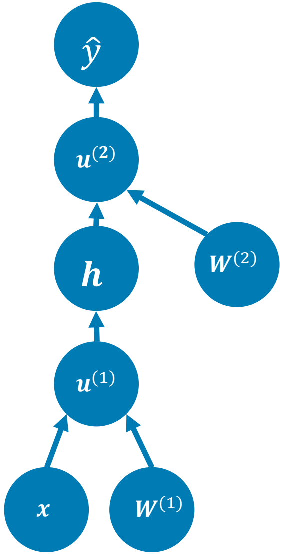
$\boldsymbol{\theta} = \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}$

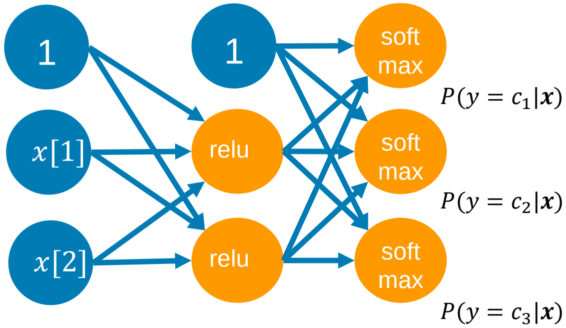
- **Backpropagation algorithm** (Rumelhart et al., 1986)
  - short: backprop
  - Propagation of the error back through the network to compute the gradients
  - **Backward pass**
  - Actual learning is done via gradient descent

Computational Graphs

Logistic regression $\hat{y} = \sigma(\boldsymbol{x}^T \boldsymbol{w} + b)$







Fully connected feed forward network

$$\hat{\boldsymbol{y}} = softmax\{\boldsymbol{h}\boldsymbol{W}^{(2)}\}$$
$$= softmax\{\max\{0, \boldsymbol{x}\boldsymbol{W}^{(1)}\}\boldsymbol{W}^{(2)}\}$$

Input: 2d, i.e. 2 features

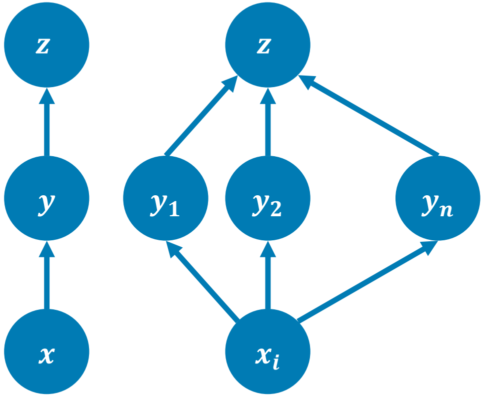
output: Probabilities for each of the three exclusive classes

Chain Rule

- Given $g, f\colon \mathbb{R} \rightarrow \mathbb{R}$
  - $y = g(x)$
  - $z = f\big(g(x)\big) = f(y)$

- Chain rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



- In case of vectors $\quad g\colon \mathbb{R}^m \rightarrow \mathbb{R}^n \qquad f\colon \mathbb{R}^n \rightarrow \mathbb{R}$
  - $\boldsymbol{x} \in \mathbb{R}^m;\ \boldsymbol{y} \in \mathbb{R}^n; \boldsymbol{y} = g(\boldsymbol{x});\ z = f(\boldsymbol{y})$

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

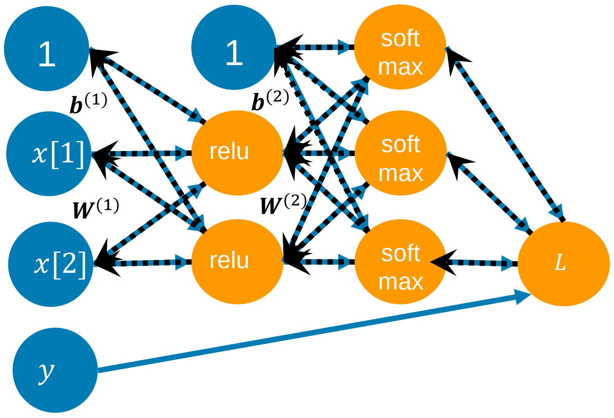
$n \times m$ Jacobi-Matrix

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

$n \times m$ Jacobi-Matrix

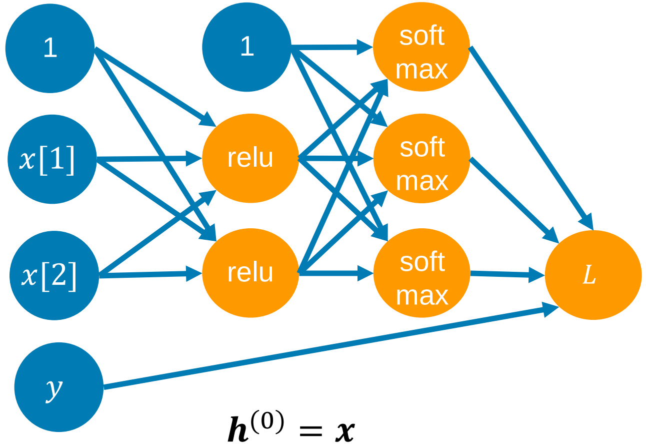
Graphical Representation

- For epoche 1 to k:
  - For each training sample / batch of samples:
    - Forward pass
    - Computation of loss function
    - Backward pass
    - Update of weights (gradient descent)



The Algorithm: Forward Pass

- Input:
  - Network depth l
  - $\boldsymbol{W}^{(i)}, i \in \{1, \dots, l\}$
  - $\boldsymbol{b}^{(i)}, i \in \{1, \dots, l\}$
  - $\boldsymbol{x}$ input data
  - $\boldsymbol{y}$ target data
- Output
  - Value of the loss function at position $\boldsymbol{x}$



$$
\begin{aligned}
&\boldsymbol{h}^{(0)} = \boldsymbol{x} \\
&\boldsymbol{for}\ k = 1, \dots, l\ \boldsymbol{do} \\
&\qquad \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)} \\
&\qquad \boldsymbol{h}^{(k)} = f\left(\boldsymbol{a}^{(k)}\right) \\
&\boldsymbol{end\ for} \\
&\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)} \\
&J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\boldsymbol{\theta})
\end{aligned}
$$

Regularization;
$\boldsymbol{\theta} = \textbf{all weights} + \textbf{bias terms}$

The Algorithm: Backward Pass

- Output: The gradients of all activations $\boldsymbol{a}^{(k)}$
- Afterwards, update the weights
  - E.g. using gradient descent

$$
\begin{aligned}
&\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) \\
&\boldsymbol{for}\ k = l, l-1, \dots, 1\ \boldsymbol{do} \\
&\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right) \\
&\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g} \\
&\boldsymbol{end\ for}
\end{aligned}
$$

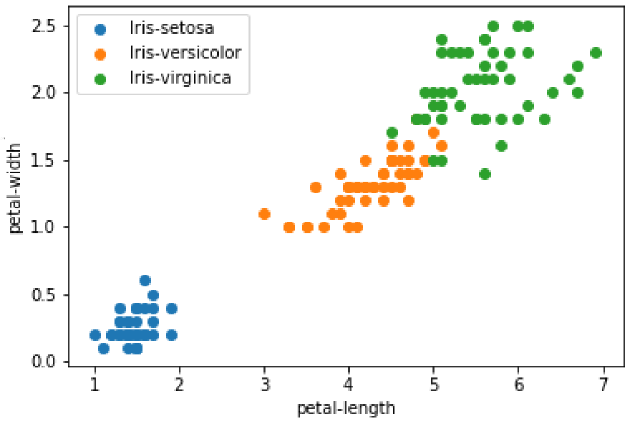
Gradient of output layer

Gradient before non-linear activation; ⊙ element-wise multiplication

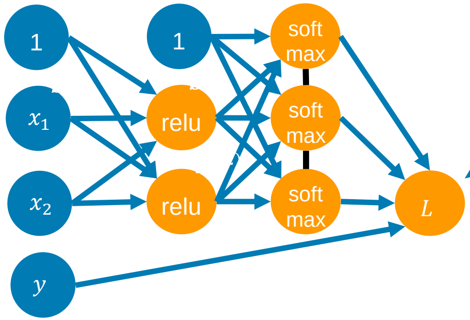
Gradients of weights and bias terms of layer k

Propagation of the gradients to the activations of the next lower layer

Walk-Through Example: Iris Dataset




```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```



Binary cross-entropy:

$$L = BCE(\hat{y}, y) = -(y log(\hat{y}) + (1 - y) log(1 - \hat{y}))$$

Cross-entropy:

$$L = CE(\hat{y}, y) = -\sum_{C} y_i log(\hat{y}_i)$$

Cross-entropy considering also negative samples:

$$L = CE(\hat{y}, y) = -\sum_{C} y_i log(\hat{y}_i)\,(1 - y_i) log(1 - \hat{y}_i)$$

Walk-Through Example: Forward Pass

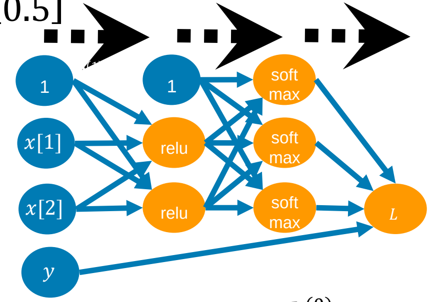
- Initializing the weights

Different Initializations possible

$$W^{(1)} = \begin{bmatrix} 0.1 & -0.2 \\ 0.4 & 0.6 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0.2 \\ -0.3 \end{bmatrix} \quad W^{(2)} = \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} -0.1 \\ -0.5 \\ -0.4 \end{bmatrix}$$

- First training sample $\boldsymbol{x}^{(1)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix} \quad \boldsymbol{y}^{(1)} = \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} \quad \boldsymbol{h}^{(0)} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix}$
  - $\boldsymbol{a}^{(1)} = \begin{bmatrix} 0.2 \\ -0.3 \end{bmatrix} + \begin{bmatrix} 0.1 & -0.2 \\ 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}$
  - $\boldsymbol{h}^{(1)} = relu\left(\begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}\right) = \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}$
  - $\boldsymbol{a}^{(2)} = \begin{bmatrix} -0.1 \\ -0.5 \\ -0.4 \end{bmatrix} + \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix} = \begin{bmatrix} -0.23 \\ -0.37 \\ -0.46 \end{bmatrix}$
  - $\boldsymbol{o}^{(2)} = \text{softmax}\left(\begin{bmatrix} -0.23 \\ -0.37 \\ -0.46 \end{bmatrix}\right) = \begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix} = \hat{\boldsymbol{y}}$
  - $J = CE\left(\begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix}, \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}\right) = -(\log(0.38) + \log(1-0.32) + \log(1-0.30)) = 0.74$

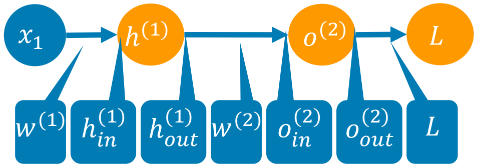
Considers also negative samples



$$\begin{aligned} &\boldsymbol{h}^{(0)} = \boldsymbol{x} \\ &\boldsymbol{for}\ k = 1, \dots, l\ \boldsymbol{do} \\ &\quad \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)} \\ &\quad \boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)}) \\ &\boldsymbol{end\ for} \\ &\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)} = \boldsymbol{o}^{(l)} \\ &J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\boldsymbol{\theta}) \end{aligned}$$

https://github.com/Prakashvanapalli/TensorFlow/blob/master/Blogposts/Backpropogation_with_Images.ipynb

Walk-Through Example: Backward Pass I

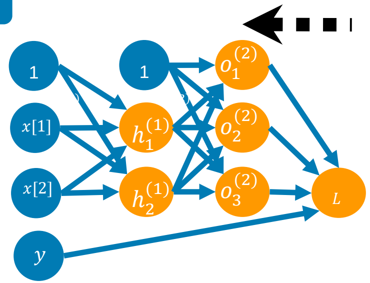


- Loss for $\boldsymbol{x}^{(1)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix}$

$$L = CE\left(\begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix}, \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}\right) = 0.74$$

- Reminder: chain rule

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial o_{out}} \frac{\partial o_{out}}{\partial o_{in}} \frac{\partial o_{in}}{\partial h_{out}} \frac{\partial h_{out}}{\partial h_{in}} \frac{\partial h_{in}}{\partial W^{(1)}}$$



- Gradient of loss function:

- $$\frac{\partial L}{\partial o_{out}} = \begin{bmatrix} \frac{\partial L}{\partial o_{1,out}} \\ \frac{\partial L}{\partial o_{2,out}} \\ \frac{\partial L}{\partial o_{3,out}} \end{bmatrix} = \begin{bmatrix} -1 \cdot (1 \cdot \frac{1}{0.38} + (1-1) \cdot \frac{1}{1-0.38}) \\ -1 \cdot (0 \cdot \frac{1}{0.32} + (1-0) \cdot \frac{1}{1-0.32}) \\ -1 \cdot (0 \cdot \frac{1}{0.30} + (1-0) \cdot \frac{1}{1-0.30}) \end{bmatrix} = \begin{bmatrix} -2.63 \\ -1.47 \\ -1.43 \end{bmatrix}$$

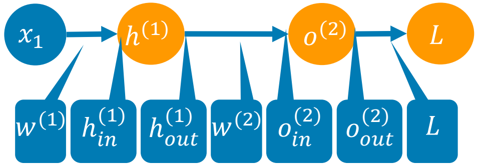
Partial derivative of cross-entropy:

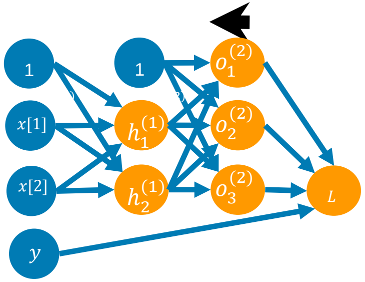
$$\frac{\partial L}{\partial \hat{y}_i} = -1 \cdot (y_i \frac{1}{\hat{y}_i} + (1 - y_i) \frac{1}{1 - \hat{y}_i})$$

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$\boldsymbol{for}\ k = l, l-1, \ldots, 1\ \boldsymbol{do}$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
$\boldsymbol{end\ for}$

Hadamard product

Walk-Through Example: Backward Pass II





- Gradient of the output of the output layer

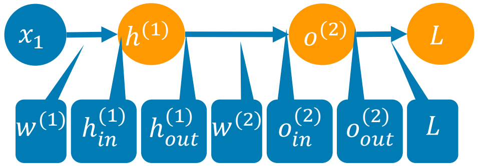
$$\frac{\partial o_{out}}{\partial o_{in}} = \begin{bmatrix} \frac{\partial o^{(2)}_{1,out}}{\partial o^{(2)}_{1,in}} \\ \frac{\partial o^{(2)}_{2,out}}{\partial o^{(2)}_{2,in}} \\ \frac{\partial o^{(2)}_{3,out}}{\partial o^{(2)}_{3,in}} \end{bmatrix} = \begin{bmatrix} \frac{e^{-0.23} \cdot (e^{-0.37} + e^{-0.46})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \\ \frac{e^{-0.37} \cdot (e^{-0.23} + e^{-0.46})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \\ \frac{e^{-0.46} \cdot (e^{-0.23} + e^{-0.37})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \end{bmatrix} = \begin{bmatrix} 0.23 \\ 0.22 \\ 0.21 \end{bmatrix}$$

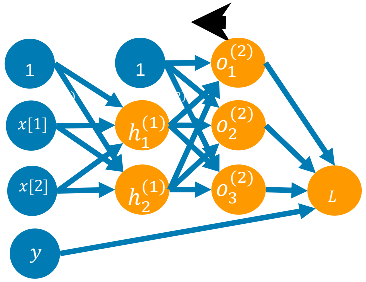
Partrial derivative of softmax:

$$\frac{\partial o^{(2)}_{i,out}}{\partial o^{(2)}_{i,in}} = \frac{e^{o^{(2)}_{i,in}} \cdot \sum_{j \neq i} e^{o^{(2)}_{j,in}}}{\left(\sum_j e^{o^{(2)}_{j,in}}\right)^2}$$

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$\boldsymbol{for}\ k = l, l-1, \dots, 1\ \boldsymbol{do}$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
$\boldsymbol{end\ for}$

Walk-Through Example: Backward Pass III





- Gradient of the input of the output layer with respect to weights $W^{(2)}$

$$\frac{\partial o_{1,in}^{(2)}}{\partial W_{1,1}^{(2)}} = \frac{\partial o_{2,in}^{(2)}}{\partial W_{1,2}^{(2)}} = \frac{\partial o_{3,in}^{(2)}}{\partial W_{1,3}^{(2)}} = 0.28 \qquad \frac{\partial o_{1,in}^{(2)}}{\partial W_{2,1}^{(2)}} = \frac{\partial o_{2,in}^{(2)}}{\partial W_{2,2}^{(2)}} = \frac{\partial o_{3,in}^{(2)}}{\partial W_{2,3}^{(2)}} = 0.72$$

Partial derivative:

$$\frac{\partial o_{1,in}^{(2)}}{\partial W_{1,1}^{(2)}} = \frac{\partial\left(h_1^{(1)} W_{1,1}^{(2)} + h_2^{(1)} W_{2,1}^{(2)} + b_1^{(2)}\right)}{\partial W_{1,1}^{(2)}} = h_{1,out}^{(1)}$$

$$\begin{aligned}
&\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) \\
&\textbf{\textit{for}}\ k = l, l-1, \dots, 1\ \textbf{\textit{do}} \\
&\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right) \\
&\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g} \\
&\textbf{\textit{end for}}
\end{aligned}$$

Walk-Through Example: Backward Pass IV

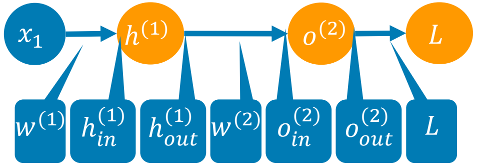
$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial o_{out}} \frac{\partial o_{out}}{\partial o_{in}} \frac{\partial o_{in}}{\partial W^{(2)}} = \begin{bmatrix} \frac{\partial L}{\partial o_{1,out}} \frac{\partial o_{1,out}}{\partial o_{1,in}} \frac{\partial o_{1,in}}{\partial W^{(2)}_{1,1}} & \frac{\partial L}{\partial o_{1,out}} \frac{\partial o_{1,out}}{\partial o_{1,in}} \frac{\partial o_{1,in}}{\partial W^{(2)}_{2,1}} \\ \frac{\partial L}{\partial o_{2,out}} \frac{\partial o_{2,out}}{\partial o_{2,in}} \frac{\partial o_{2,in}}{\partial W^{(2)}_{1,2}} & \frac{\partial L}{\partial o_{2,out}} \frac{\partial o_{2,out}}{\partial o_{2,in}} \frac{\partial o_{2,in}}{\partial W^{(2)}_{2,2}} \\ \frac{\partial L}{\partial o_{3,out}} \frac{\partial o_{3,out}}{\partial o_{13in}} \frac{\partial o_{3,in}}{\partial W^{(2)}_{1,3}} & \frac{\partial L}{\partial o_{3,out}} \frac{\partial o_{3,out}}{\partial o_{13in}} \frac{\partial o_{3,in}}{\partial W^{(2)}_{2,3}} \end{bmatrix}$$

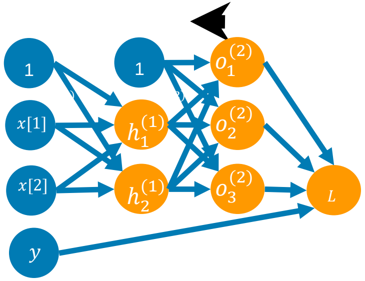
$$\frac{\partial L}{\partial W^{(2)}} = \begin{bmatrix} -2.63 \cdot 0.23 \cdot 0.28 & -2.63 \cdot 0.23 \cdot 0.72 \\ -1.47 \cdot 0.22 \cdot 0.28 & -1.47 \cdot 0.22 \cdot 0.72 \\ -1.43 \cdot 0.21 \cdot 0.28 & -1.43 \cdot 0.21 \cdot 0.72 \end{bmatrix}$$

- Gradient descent with learning rate $\lambda = 0.5$ results in new weights:

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \qquad \nabla_{\boldsymbol{W}^{(2)}} \boldsymbol{L} = \begin{bmatrix} -0.17 & -0.44 \\ -0.09 & -0.23 \\ -0.08 & -0.22 \end{bmatrix}$$

$$\boldsymbol{W}'^{(2)} = \boldsymbol{W}^{(2)} - \lambda \nabla_{\boldsymbol{W}^{(2)}} \boldsymbol{L} = \begin{bmatrix} 0.39 & -0.08 \\ 0.25 & 0.22 \\ 0.34 & -0.09 \end{bmatrix}$$





$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \dots, 1$ ***do***
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
end for

Computational Graph

$$x \rightarrow h \rightarrow L$$

- Draw a computational graph for a MLP with one hidden layer, cross-entropy as loss function, and L-2 Regularization for both weight matrices

Start 5 4 3 2 1 End

Lerning Goals for this Chapter

- Train a simple neural network in Python using Keras
- Understand gradient-based optimization
- Describe the components of deep neural networks
- Understand and apply the backpropagation algorithm

- Relevant chapters
  - P2, P3