University of Zurich^UZH

Institute of Computational Linguistics

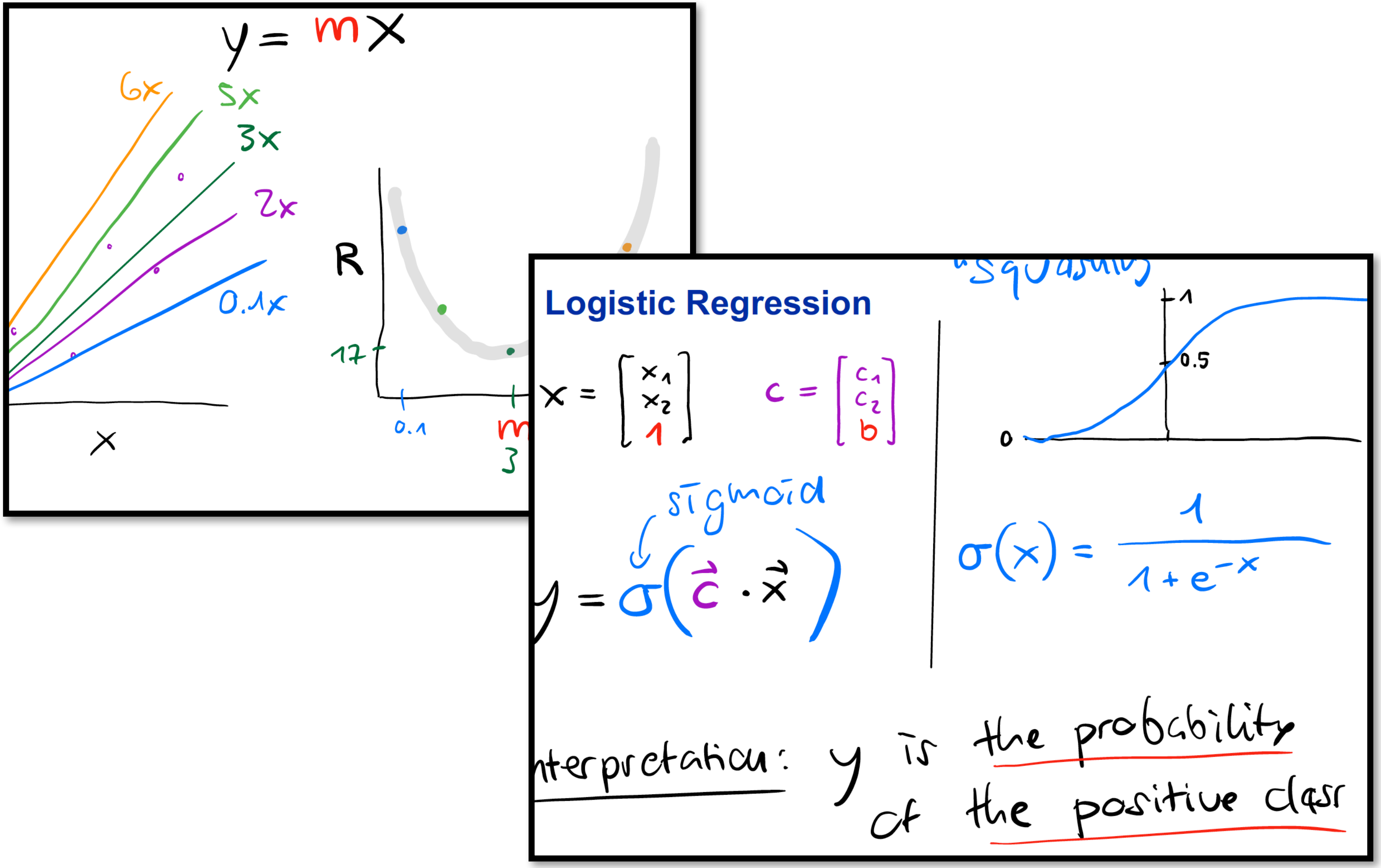
Machine Translation

7 Neural Networks

Mathias Müller

How was Exercise 2 for you?

Last time



Logistic Regression

$$x = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \qquad c = \begin{bmatrix} c_1 \\ c_2 \\ b \end{bmatrix}$$

sigmoid

$$y = \sigma(\vec{c} \cdot \vec{x})$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

nterpretation: y is the probability of the positive class

Topics of today

FFNN

- Neural networks: feed-forward neural networks or “multi-layer perceptrons”

MLP

- Backpropagation
- Gradient Descent

Remember: logistic regression

training data

x	y
$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	"good"
$\begin{bmatrix} 3 \\ 4 \end{bmatrix}$	"bad"
	"awesome"

$$\vec{w} = \begin{bmatrix} 0.1 \\ -2 \end{bmatrix}$$

probability of positive class $= \sigma\left(\vec{w} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$

Multi-class classification with linear models

training data

x	y
$\begin{bmatrix}1\\2\\3\end{bmatrix}$	"dog"
$\begin{bmatrix}4\\5\\6\end{bmatrix}$	"cat"
$\begin{bmatrix}7\\8\\9\end{bmatrix}$	"frog"

$\begin{bmatrix}1\\0\\0\end{bmatrix}$

$\begin{bmatrix}0\\1\\0\end{bmatrix}$

$\begin{bmatrix}0\\0\\1\end{bmatrix}$

Multi-class classification with linear models

training data

x	y
$\begin{bmatrix}1\\2\\3\end{bmatrix}$	"dog"
$\begin{bmatrix}4\\5\\6\end{bmatrix}$	"cat"
$\begin{bmatrix}7\\8\\9\end{bmatrix}$	"frog"

$$W = \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

vector with probability distribution $= s\left(W\vec{x}\right) = \begin{bmatrix}-1.6\\0.4\\1.2\end{bmatrix}$

Softmax

$$\boxed{s(\vec{z}) = \frac{e^{z_j}}{\sum_{k=1} e^{z_k}}}$$

$$\text{dog} \mathrel{\hat{=}} \begin{bmatrix}1\\0\\0\end{bmatrix}$$

$$\text{cat} \mathrel{\hat{=}} \begin{bmatrix}0\\1\\0\end{bmatrix}$$

without softmax

$$W\vec{x} = \begin{bmatrix}-1.6\\0.4\\1.2\end{bmatrix}$$

with softmax

$$s(W\vec{x}) = \begin{bmatrix}0.04\\0.30\\0.66\end{bmatrix}$$

Feed-forward Neural Networks

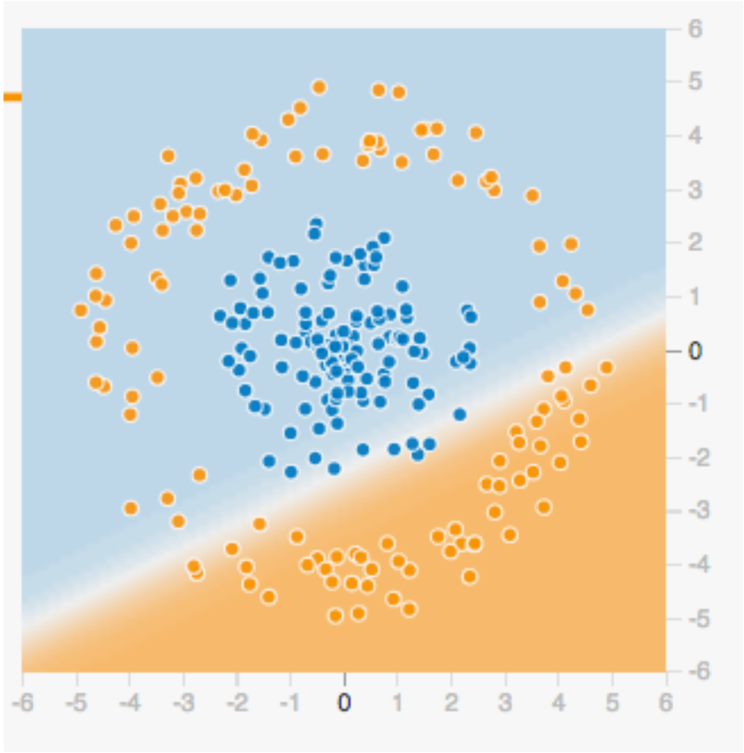
Feed-forward neural networks

Are like **logistic regression extended**:

- multiple layers, each with its own weights
- the activation function does not have to be sigmoid

Why would we want to extend log reg?

- Linear models, can only solve linear problems
- Need more complicated model families



Nesting linear transformations

$$\vec{b} = W\vec{a}$$

$$\vec{c} = U\vec{b}$$

$$\vdots$$

$$\vec{c} = U(W\vec{a})$$

FFNN structure, vector notation

- Several layers, each with their own weight matrix and activation function

$$\vec{b} = \sigma_1\left(W\vec{a}\right)$$

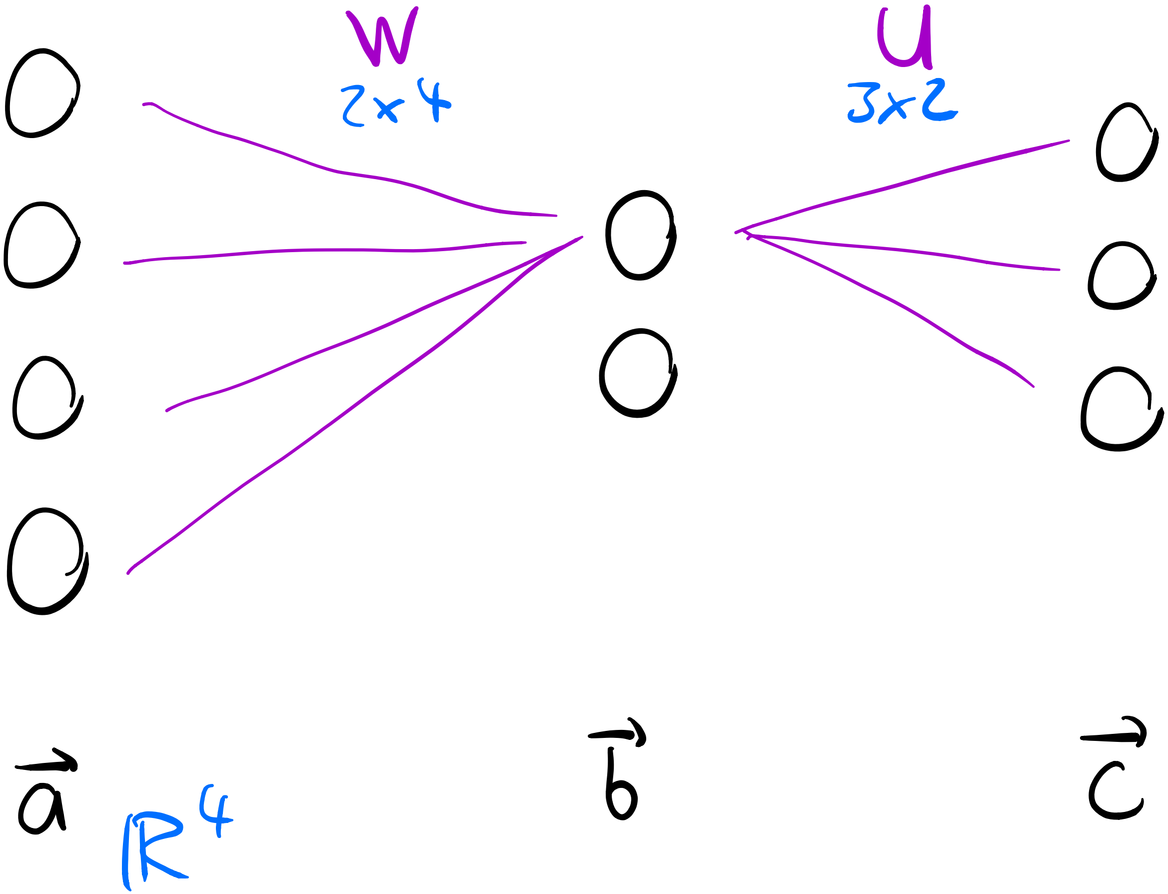
$$\vec{c} = \sigma_2\left(U\vec{b}\right)$$

sigmoid

relu

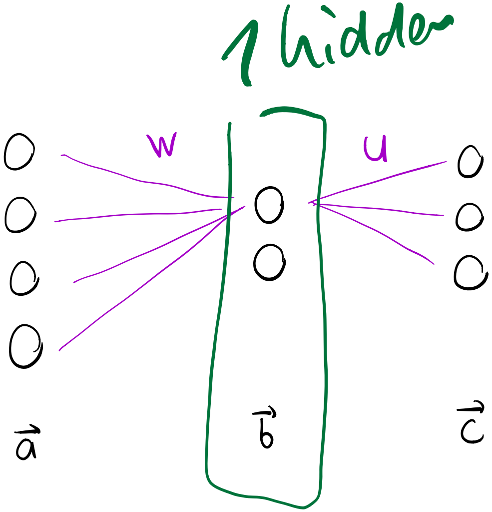
tanh

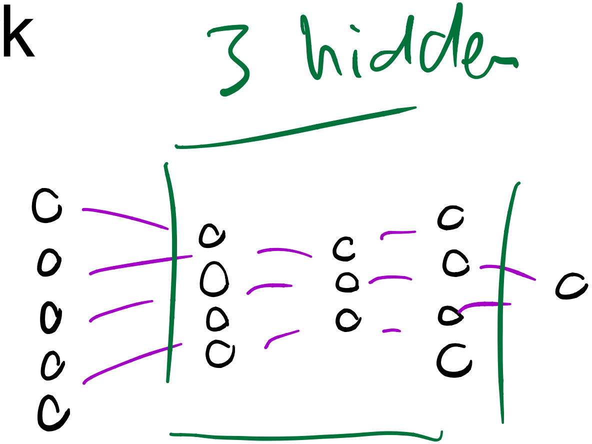
How to understand drawings of FFNNs



Layers

- “hidden” layers,
- “deep” neural network





Activation functions (“non-linearities”)

- actually crucial for non-linear behaviour!
- applied element-wise

$$\vec{e} = \sigma(\vec{d})$$

$$\text{RELU} = \max(0, x)$$

$$\text{RELU}\left(\begin{bmatrix} -2 \\ 0 \\ 3 \\ -2 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

When to use which activation function

- hidden layer: always use RELU

- output layer:
  - regression problem: ~~identity function~~

    no function

  - classification problem: softmax

Summary FFNN structure

- have several layers,
- a layer
  - takes a vector as input $\vec{a}$
  - computes a matrix-vector product with a weight matrix $W\vec{a}$
  - and applies an activation function element-wise $RELU(W\vec{a})$
  - output: a vector

W U

Learning optimal parameters

How to learn optimal weights?

- define how to measure error, a **loss function**

- Two-step procedure:
  - find partial derivative of a loss function with respect to each weight

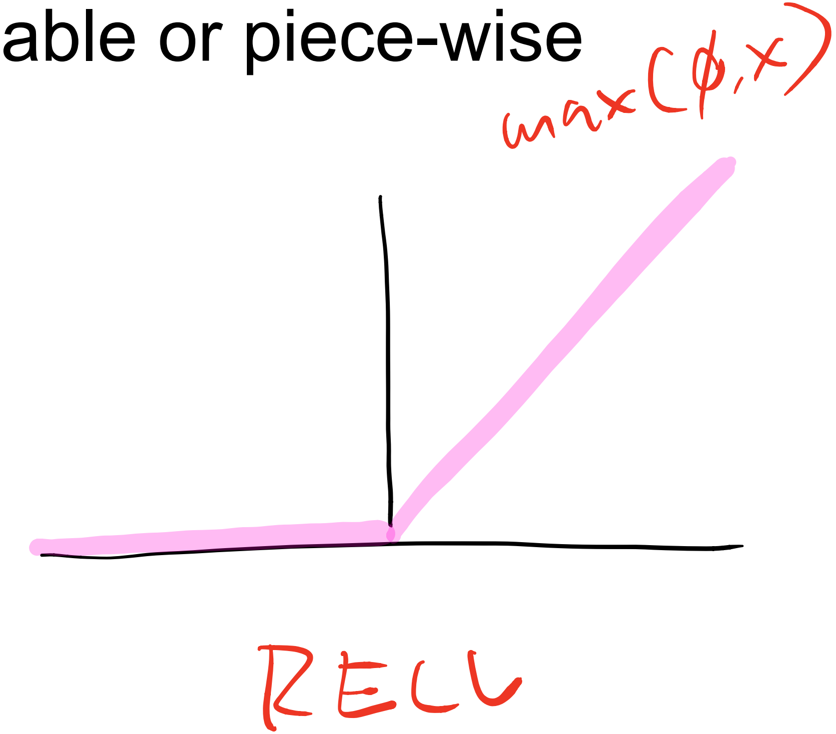
    backpropagation

  - ② update each parameter using its partial derivative

    gradient descent

Prerequisite: differentiability

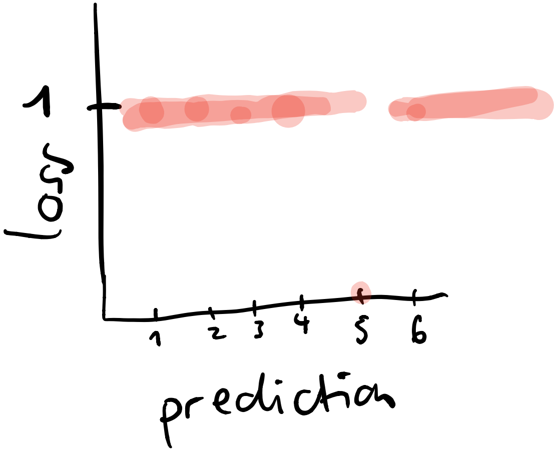
- Entire computation must be smooth enough: differentiable or piece-wise differentiable



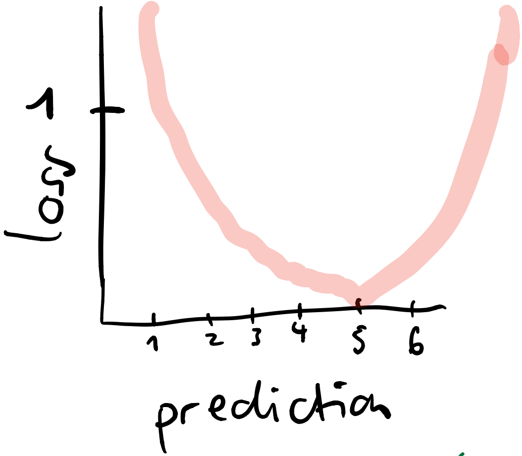
Loss functions

- Our actual metric may **not be differentiable,** so we have to define a surrogate loss function

correct y:
5



0-1-loss



squared loss

Typical loss functions

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

- For regression: MSE

→ for any vectors

- For classification: cross-entropy

→ for distributions!

Mean squared error loss (MSE)

$$\frac{1}{N} \sum_{x,y}^{N} \left(\text{correct\_y} - \text{prediction} \right)^2$$

Cross-entropy loss (CE)

prediction

truth

$$p = \begin{bmatrix} 0.7 \\ 0.1 \\ 0.2 \end{bmatrix} \qquad q = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$CE(p, q) = -\sum_{k=1}^{K} q_k * \log(p_k)$$

Reverse Differentiation: Backpropagation

Backpropagation

- **Want**: influence of each parameter on current loss

$$\frac{\partial L}{\partial w}$$

- Writing out analytical gradient too complicated for nested functions
- Instead: staged computation by applying the **chain rule of calculus**

Chain rule of calculus

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

$y = 3x = 12$

$z = y^2$

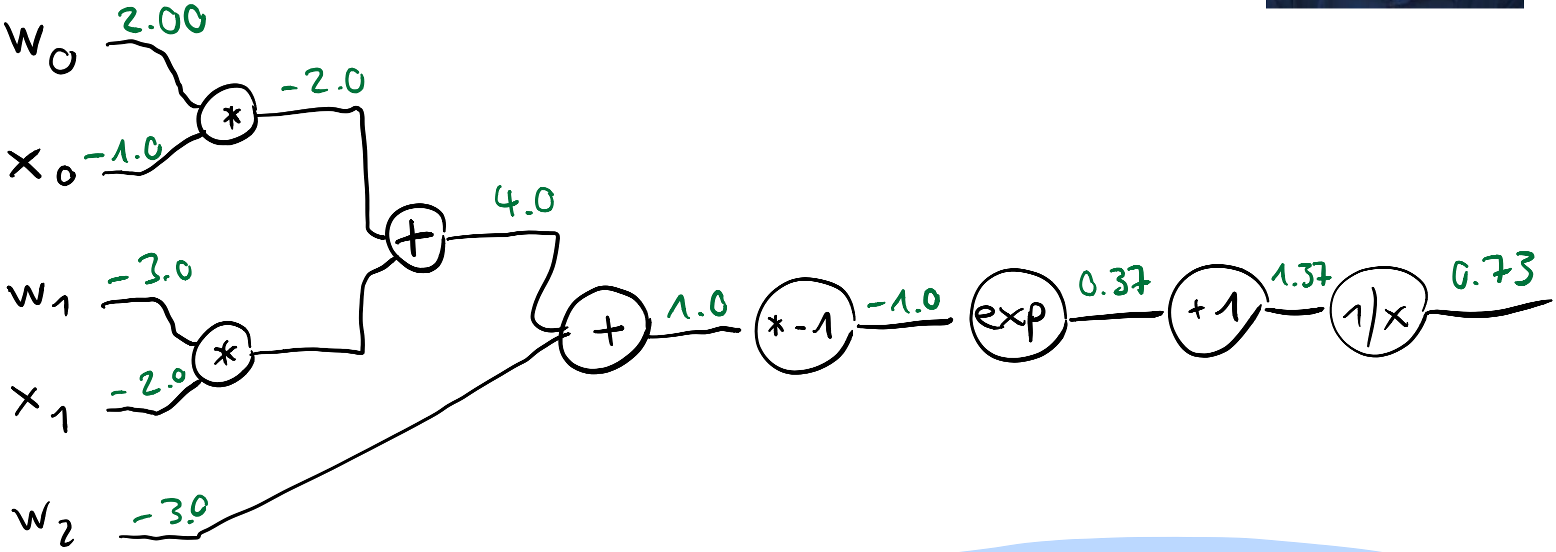
$x = 4$

$\frac{dz}{dy} = 2y = 24$

$\frac{dy}{dx} = 3$

$\frac{dz}{dx} = 24 * 3$

View FFNN as a computational graph



$$y = w_0 x_0 + w_1 x_1 + w_2$$

$$\sigma(y) = \frac{1}{1+e^{-y}} = 0.73 \,!$$

Backpropagation

$x^2 \quad 2x$

What backpropagation sees:

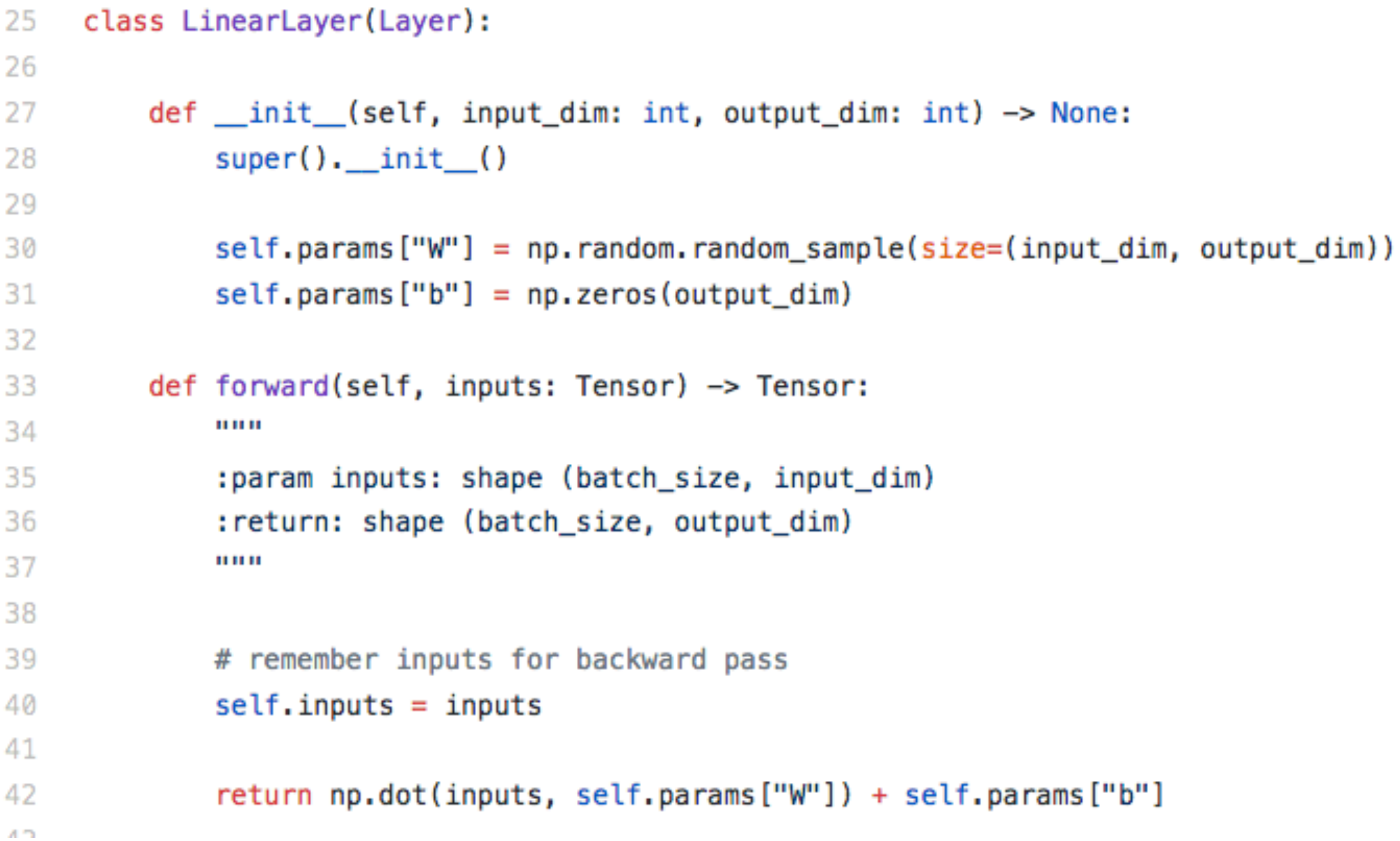
- graph of nodes
- every node can perform **forward** pass and remember its inputs

output of n = n.forward(inputs)

- every node can perform **backward** pass and knows its local gradient

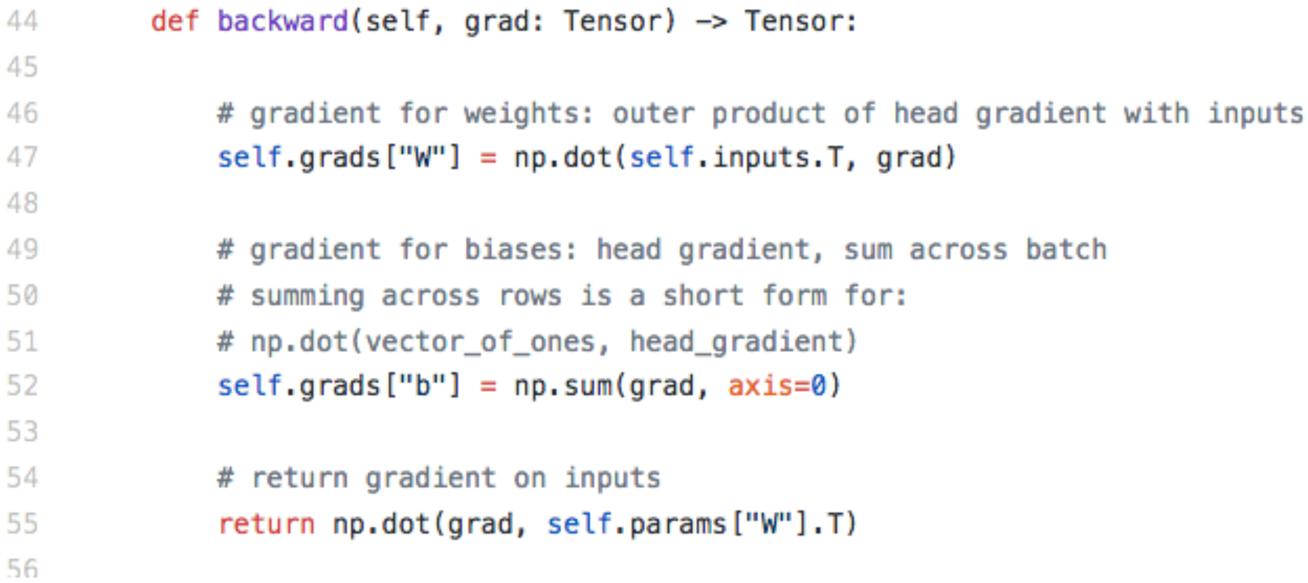
$\frac{\partial L}{\partial n.w}$ = n.backward(head gradient)

Every node can perform forward pass and remember its inputs



```
class LinearLayer(Layer):

    def __init__(self, input_dim: int, output_dim: int) -> None:
        super().__init__()

        self.params["W"] = np.random.random_sample(size=(input_dim, output_dim))
        self.params["b"] = np.zeros(output_dim)

    def forward(self, inputs: Tensor) -> Tensor:
        """
        :param inputs: shape (batch_size, input_dim)
        :return: shape (batch_size, output_dim)
        """


        # remember inputs for backward pass
        self.inputs = inputs

        return np.dot(inputs, self.params["W"]) + self.params["b"]
```

Every node can perform backward pass and knows its local gradient



```python
    def backward(self, grad: Tensor) -> Tensor:

        # gradient for weights: outer product of head gradient with inputs
        self.grads["W"] = np.dot(self.inputs.T, grad)

        # gradient for biases: head gradient, sum across batch
        # summing across rows is a short form for:
        # np.dot(vector_of_ones, head_gradient)
        self.grads["b"] = np.sum(grad, axis=0)

        # return gradient on inputs
        return np.dot(grad, self.params["W"].T)

```

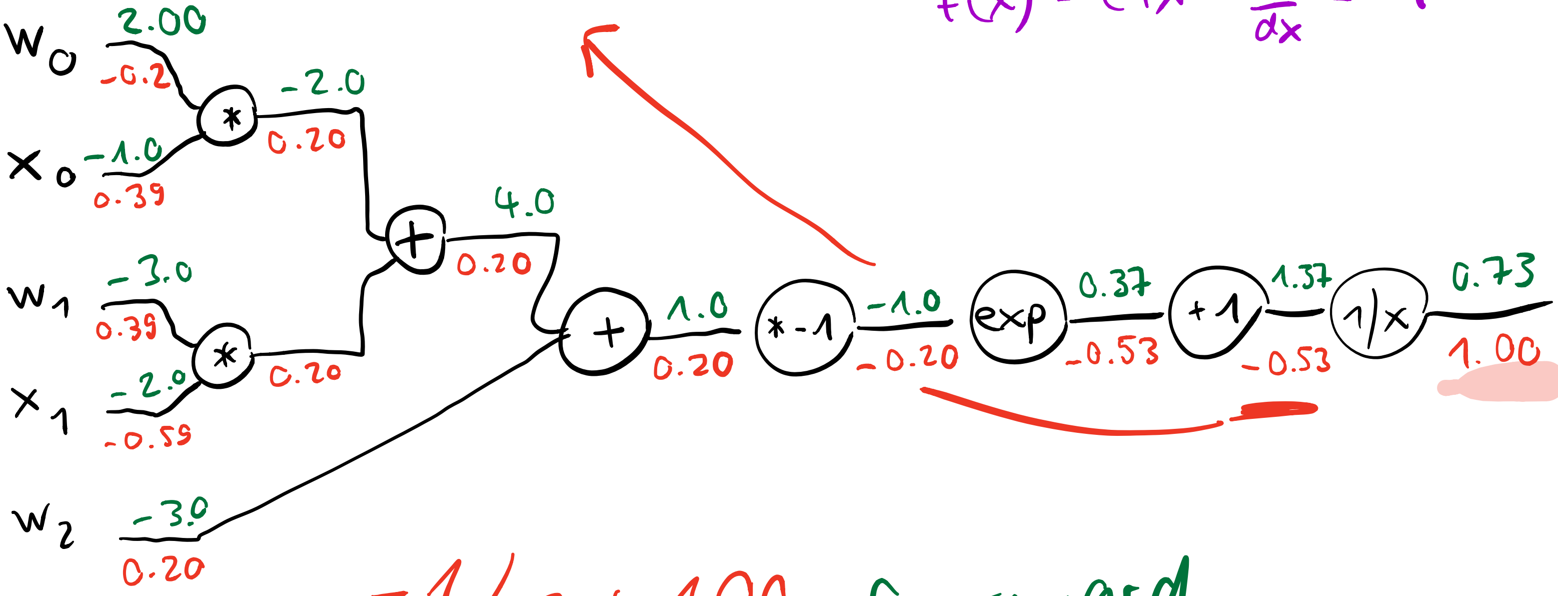
Computational graph view

RULES

$f(x) = \frac{1}{x} \quad \frac{df}{dx} = -1/x^2$

$f(x) = c + x \quad \frac{df}{dx} = 1$

$\frac{1}{x} = -1/x^2$

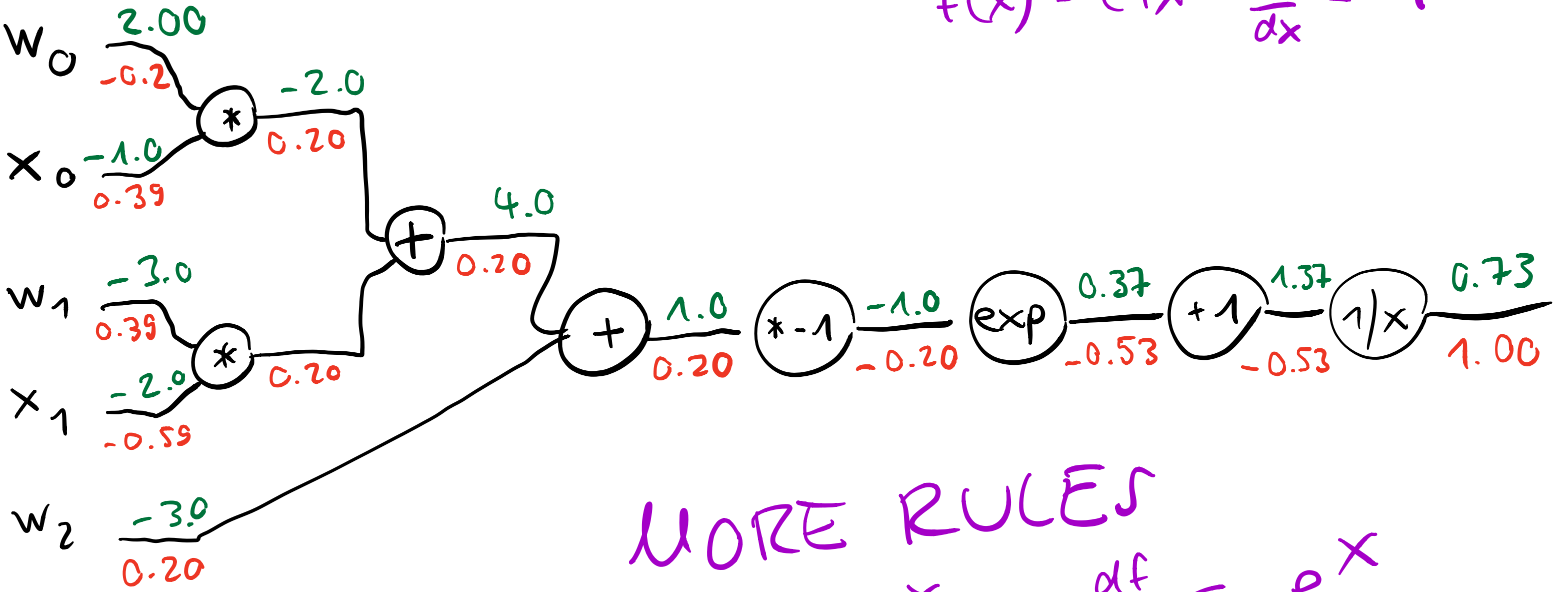


$-1/x^2 * 1.00$

forward

backward

Computational graph view



RULES

$f(x) = \frac{1}{x} \quad \frac{df}{dx} = -1/x^2$

$f(x) = c + x \quad \frac{df}{dx} = 1$

MORE RULES

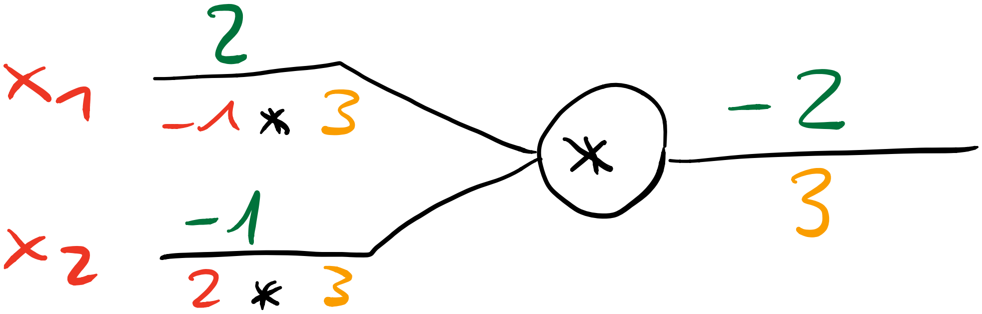
$f(x) = e^x \quad \frac{df}{dx} = e^x$

$f(x) = ax \quad \frac{df}{dx} = a$

Backpropagation

$f(x) = ax$ $\quad \frac{df}{dx} = a$

- (receive some data **(X, y)**)
- (forward pass: compute output)
- (compute loss)
- in reverse order, compute local gradient and multiply with head gradient



Summary Backpropagation

- Compute partial derivatives of loss function with respect to all network parameters
- View FFNN as computational graph with operations
- Each operation must know how to do forward and backward passes

Optimization: Gradient Descent

Gradient descent

- Purpose: given a forward and backward pass, find better parameters
- Iterative optimization algorithm

What gradient descent sees

① list of weights

$$\begin{bmatrix} W \\ U \\ V \end{bmatrix}$$

② list of gradients

$$\begin{bmatrix} \frac{\partial L}{\partial W} \\ \frac{\partial L}{\partial U} \\ \frac{\partial L}{\partial V} \end{bmatrix}$$

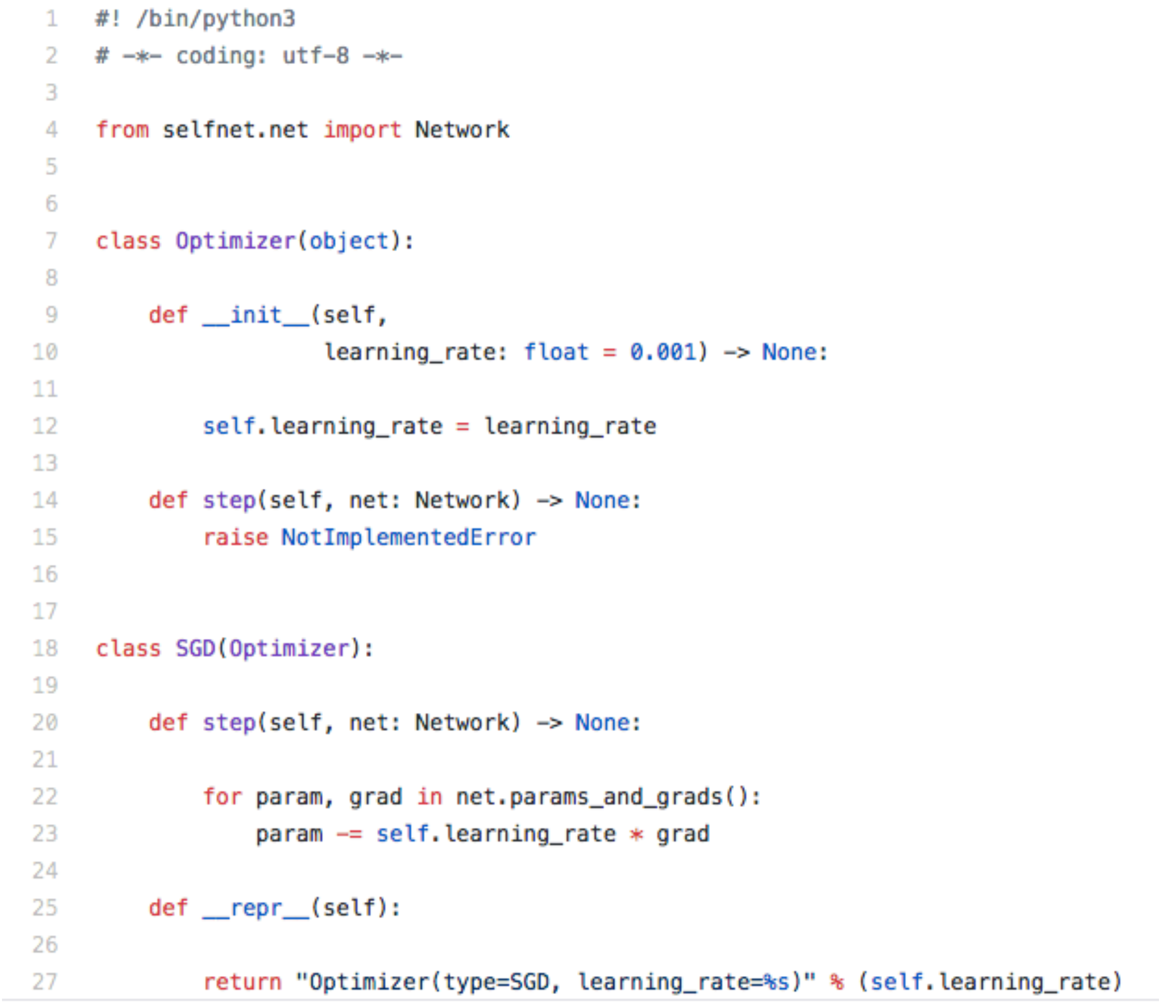
③ learning rate 0.0001

Gradient descent procedure

$$\begin{array}{ccc}
\text{current params} & \text{gradients} & \text{new parameters} \\
\begin{bmatrix}1.7\\2\\4\\3\end{bmatrix} & \begin{bmatrix}-3\\24\\11.7\\0.1\end{bmatrix} & \begin{bmatrix}1.7-\alpha*-3\\2-\alpha*24\\4-\alpha*11.7\\3-\alpha*0.1\end{bmatrix}
\end{array}$$

$$\alpha = 0.01$$

Gradient descent in code



```python
#! /bin/python3
# -*- coding: utf-8 -*-

from selfnet.net import Network


class Optimizer(object):

    def __init__(self,
                 learning_rate: float = 0.001) -> None:

        self.learning_rate = learning_rate

    def step(self, net: Network) -> None:
        raise NotImplementedError


class SGD(Optimizer):

    def step(self, net: Network) -> None:

        for param, grad in net.params_and_grads():
            param -= self.learning_rate * grad

    def __repr__(self):

        return "Optimizer(type=SGD, learning_rate=%s)" % (self.learning_rate)
```

Gradient descent variants

- **Batch gradient descent**: estimate gradients once and compute 1 update using the entire training set

- **Stochastic gradient descent**: use one single training example for 1 update

SGD

- **Minibatch stochastic gradient descent**: use several examples for 1 update ✓

Gradient descent hyperparameters

- Learning rate ***alpha*** α
- Minibatch size $\# (x, y)$
- How many updates

Summary

- **FFNNs** consist of layers, each layer is a linear transformation followed by an activation function
- **Backpropagation** is used to obtain gradients of the loss function with respect to each parameter
- **Gradient descent** updates parameters iteratively, using previously computed gradients

Links / Further Reading (there are LOTS)

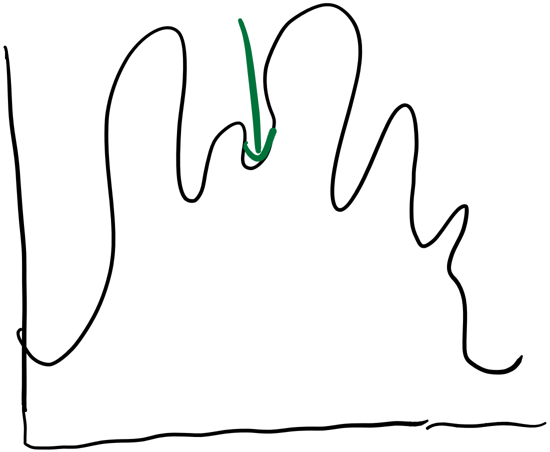
- cs231n specific lecture about FFNNs with Andrej Karpathy: https://www.youtube.com/watch?v=i94OvYb6noo&index=4&list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC
- Here is a neural network library written only in numpy: https://github.com/bricksdont/selfnet
- Chris Olah's blog post on backprop: http://colah.github.io/posts/2015-08-Backprop/
- "Deep Learning with Python" book by F. Chollet (on OLAT)
- "Deep Learning" book by Goodfellow et al (on OLAT)
- http://neuralnetworksanddeeplearning.com/ online book by Michael Nielsen
- Section 1 of Koehn's draft of NMT Chapter (on OLAT)
- www.playground.tensorflow.org

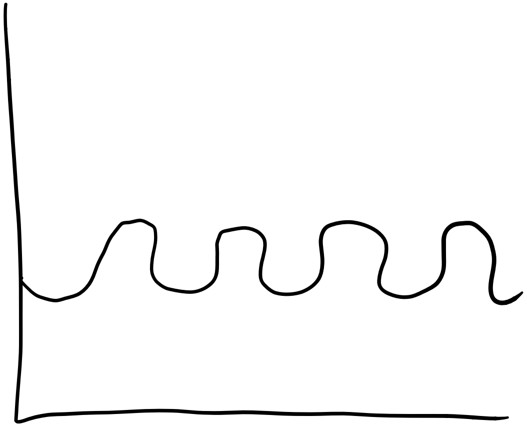
Bonus slides: local minima and non-convex optimization in FFNNs

Local minima

- Convex optimization procedure for non-convex problems?
- Local minima not a problem in high-dimensional spaces
- more common: saddle points

Local minima





Saddle points