Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 8: Neural Networks from Scratch

Motivation

- Today we finally implement a neural network from scratch
- We look at two examples to get intuition
  - Digit recognition as in 3Blue1Brown Video
  - Even tinier problem to get intuition
- You will understand things we have already worked with
  - What are these "last hidden states"
  - What is a "classification head"
- Many pretty plots!

Topics

- What is a neural network
- What are the trainable parameters
- What are nonlinearities and why do we need them
- What does a Network do before it is trained
- Training loops from scratch

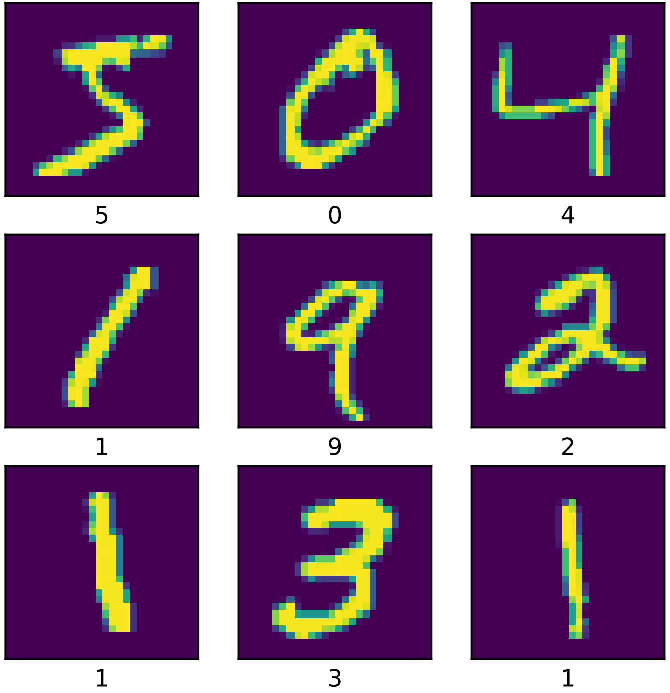
Technical topics

- Randomness in pytorch
- More advanced automatic differentiation
- Indexing tricks

Disclaimer

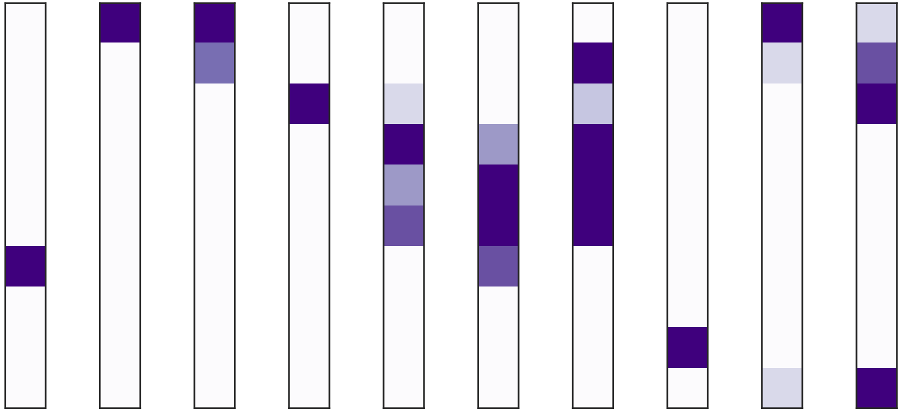
- Today we implement many things from absolute scratch
- This is so you gain understanding, how things works
- In practice, you should use higher level interfaces
- We will learn how to do that next week

Task 1: Digit recognition



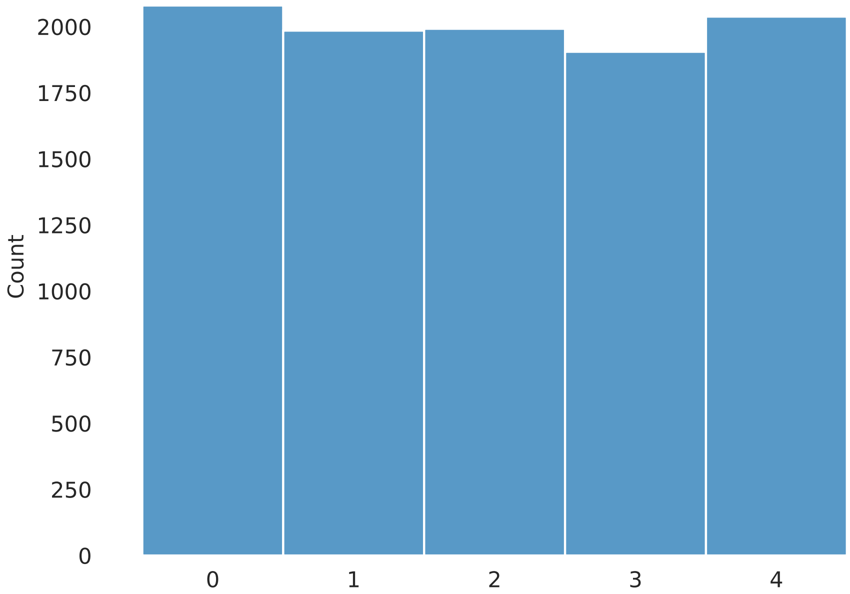
- 10 different digits
- Standard introductory example
- 28 x 28 pixels
- 255 shades of brightness
- Can be solved with very simple models
- 60 000 images with balanced class distribution

Task 2: Line length recognition

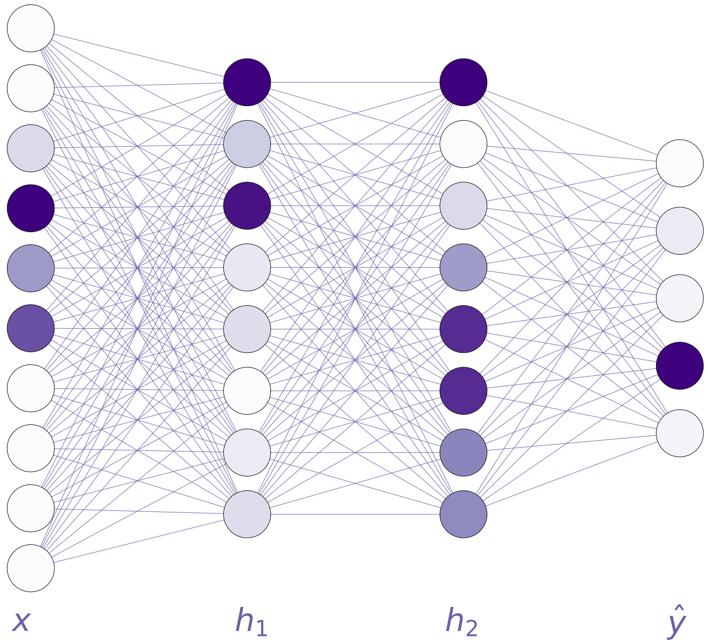


- Images of 10 x 1 Pixels with lines of length 1 to 5
- Task: Estimate the length of the line
- Can see correct result from flattened image
- Simulated sample of 10 000 images with balanced class distribution

Class balance

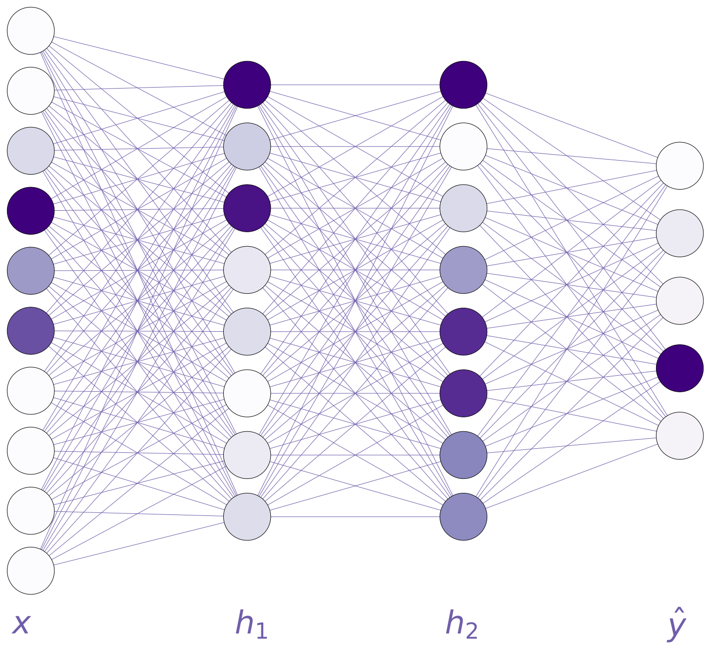


What is a neural network



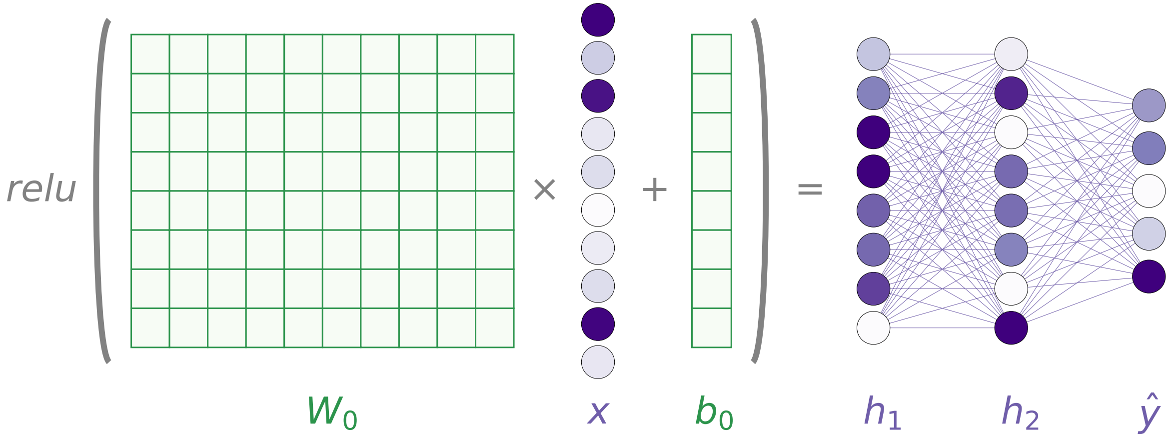
- A network consists of multiple layers of neurons
- Layer of neuron = vector of numbers
  - Input layer: just the flat image
  - hidden: transformations of image
  - Output: class probabilities
- High number -> active neuron
- Values in one layer determine values in next layer
- This model is already trained

Network architecture



- This is a Feed-Forward Network or Multi Layer Perceptron (MLP)
- Fully determined by:
  - Number of layers
  - Their dimensions
  - Activation functions
- Only dimensions of hidden layers are a choice

From x to h_1

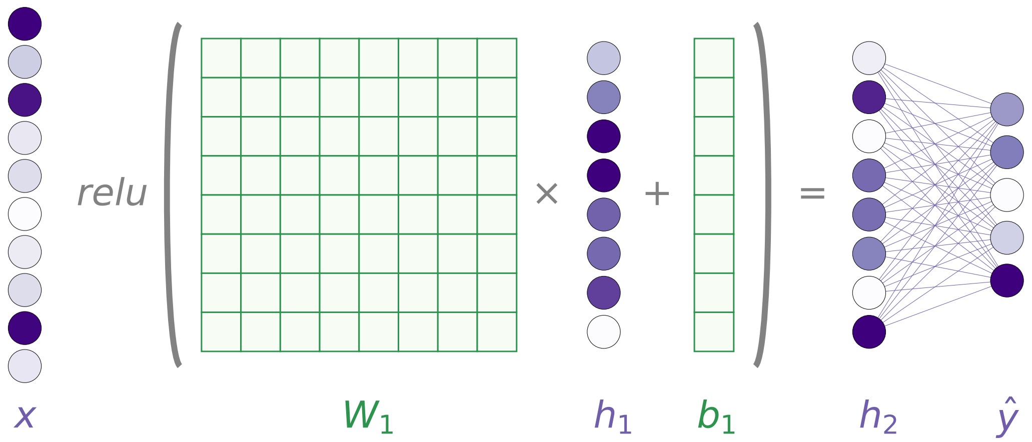


$$h_1 = relu(W_0 x + b_0)$$

Weights and biases

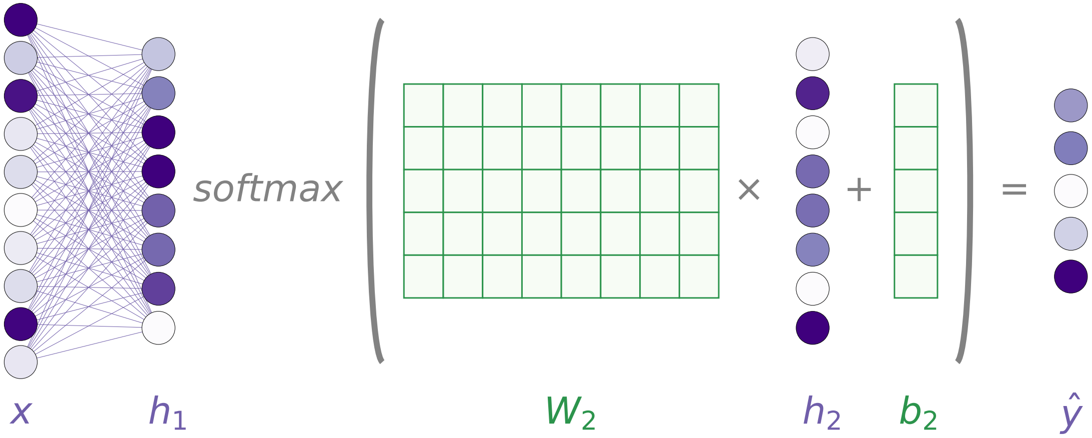
- Weights and biases are the trainable parameters
- Like β in a linear model but many more!
  - Weights are slope parameters
  - Biases are intercepts
- Weights are a matrix because as in regression with multiple outcomes

From h_1 to h_2



$$h_2 = relu(W_1 h_1 + b_1)$$

From h_2 to probabilities



$$\hat{y} = softmax(W_2 h_2 + b_2)$$

Relationship to before

- h_2 are the last hidden states we extracted in lectures 5 and 6
- The step from h_2 to $\hat{y}$ is a classification head, as in the model from lecture 7
- Of course, the models we had were:
  - deeper: Bert has 12 layers
  - larger: Bert has an input dimension of 30k and hidden dimension of 768
  - different: Transformer architecture instead of feed-forward

Task 1

5 min

Initializing parameters

- Paramters will be trained via SGD
- Need start parameters
- Simple approach: Random values close to zero, e.g. \simU[-0.5, .5]
    - Small $\rightarrow$ nothing explodes
    - Centered at zero $\rightarrow$ area of strong gradients
- There are more sophisticated methods (blogpost)

Randomness in pytorch

```
>>> torch.rand(size=(1, 3))
```

```
tensor([[0.8509, 0.1596, 0.9954]])
```

```
>>> torch.rand(size=(1, 3))
```

```
tensor([[0.6959, 0.8918, 0.3364]])
```

```
torch.manual_seed(1234)
torch.rand(size=(1, 3))
```

```
tensor([[0.0290, 0.4019, 0.2598]])
```

```
torch.rand(size=1, 3)
```

```
tensor([[0.3666, 0.0583, 0.7006]])
```

```
torch.manual_seed(1234)
torch.rand(size=(1, 3))
```

```
tensor([[0.0290, 0.4019, 0.2598]])
```

- `` `torch.rand` `` draws from $U[0, 1]$
- Each time you call it, you get different numbers
- Setting a seed, puts you in a reproducible random state
- The seed has a global effect, i.e. not just on the next drawn number but on all following numbers until the next seed

Transforming random variables

- Linear transformations of uniform random variables are uniform random variables with other bounds
- $x \sim U[0,1] \rightarrow ax + b \sim U[b, a+b]$
- Use this to generate start parameters between -0.5 and 0.5

Task 2

Why do we need nonlinearities

- Without nonlinearities, our model would be:

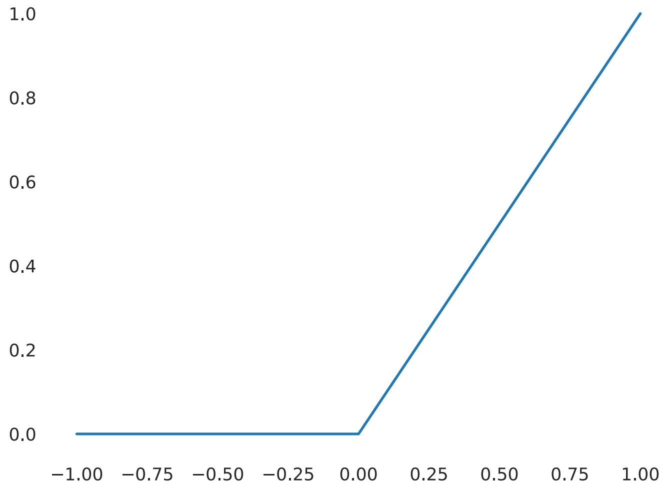
$$W_2(W_1(W_0x + b_0) + b_1) + b_2$$

- This could be simplified to:

$$Wx + b$$

- Thus we would end up with one linear model!

What is relu



- relu = rectified linear unit
- $r(x) = max(0, x)$
- Most commonly used non-linearity in neural networks
- You will see why we need non-linearities in a second

Other activation functions

- Sigmoid was used a lot historically
- Most common today is relu
- Many options out there!
- Blogpost

A brain analogy

- Neurons in the brain have a certain level of excitment
- If the excitement is above a certain level, the neuron fires
- Firing means, the neuron sends information to another neuron
- Receiving information changes that neuron’s excitement

What is softmax

- Converts any real-valued vector into a vector of valid probabilities

$$\sigma : \mathbb{R}^K \to \left\{ z \in \mathbb{R}^K | z_i \geq 0, \sum_{i=1}^K z_i = 1 \right\}$$

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{i=1}^K e^{z_i}}$$

- Our probabilities do not have frequentist interpretation
- You know this as choice probabilities in a logit model

It should be called soft argmax!

```
softmax([1, 1, 1, 1])
```

```
array([0.25, 0.25, 0.25, 0.25])
```

```
softmax([1, 1, 2, 1])
```

```
array([0.17, 0.17, 0.48, 0.17])
```

```
softmax([1, 1, 5, 1])
```

```
array([0.02, 0.02, 0.95, 0.02])
```

```
softmax([1, 1, 10, 1])
```

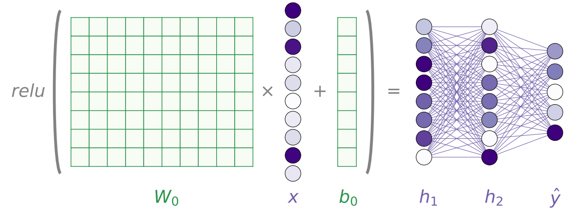
```
array([0., 0., 1., 0.])
```

- Softmax approximates an argmax
- If sizes of vector elements vary strongly, it becomes an indicator function for the largest element

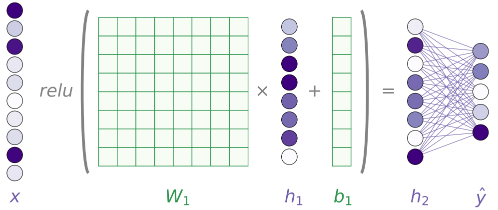
Task 3

Summary

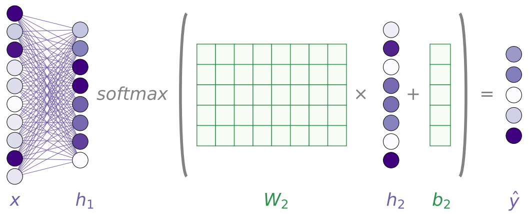
$$h_1 = relu(W_0 x + b_0)$$



$$h_2 = relu(W_1 h_1 + b_1)$$

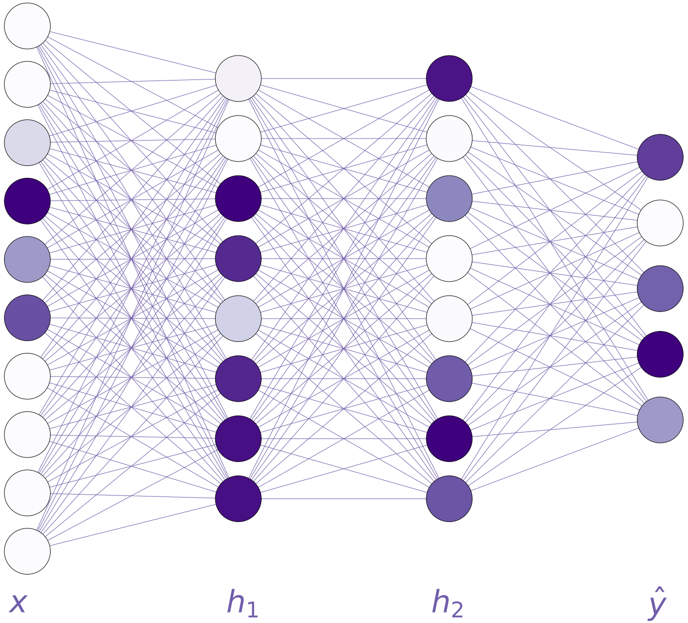


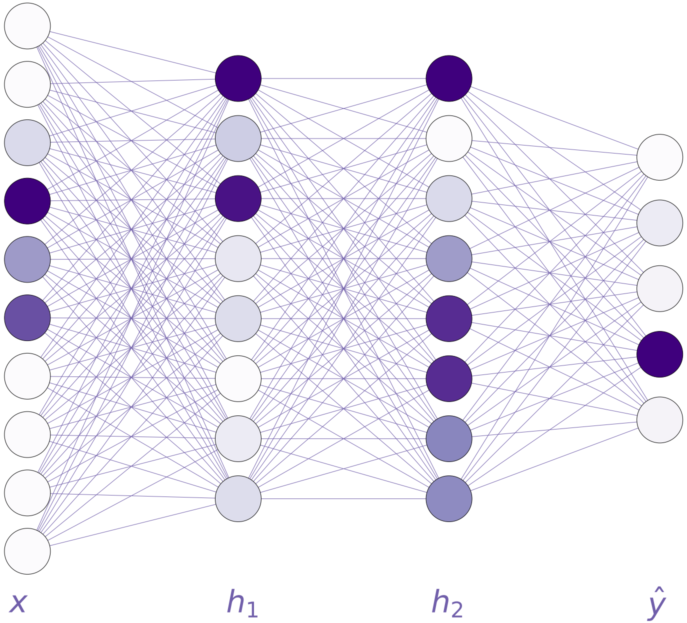
$$\hat{y} = softmax(W_2 h_2 + b_2)$$



Task 4

Goal of training





- Untrained network
- Multiple outcomes have relatively high scores

- Trained network
- Correct outcome has high score

NLL-Loss

- Negative log-likelihood loss
  - Likelihood is the score of the correct label
  - Take the log for numerical stability
  - Make negative to get something we can minimize
- Equivalent to cross-entropy loss but cross-entropy works on scores that are not yet softmaxed
- Blogpost

Batch model

```python
def batch_model(batch, weights, biases):
    n_out = len(biases[-1])
    out = torch.zeros((len(batch), n_out))
    for i, x in enumerate(batch):
        out[i] = model(x, weights, biases)
    return out
```

- For optimization we want to evaluate the model on a batch of parameters
- Since we do not care about performance, we can do this in a loop
- I give you this function in the notebook
- The result of this will go into the loss function

Indexing trick

```
>>> a = torch.arange(12).reshape(4, 3)
>>> a
```

```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

```
>>> a[torch.arange(4), torch.tensor([0, 1, 2, 1])]
```

```
tensor([ 0,  4,  8, 10])
```

- You will need this to compute the loss function
- Replacing the arange by `:` would not work!
- Numpy has `np.choose` for this, but torch does not

Task 5

When does the magic start?

- So far, the Neural Network was disappointing
- Just three lines of code, and not even difficult ones!
- Magic comes from training!
- Analogy: It is not the linear model $y = x\beta$ that is useful but OLS!
- Now we write a training loop to make our model work!

Refresher: Pseudo code for SGD

```
for i in range(nb_epochs):
  shuffle_data(data)
  for batch in data:
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

- We will implement this from scratch using pytorch's autograd ability
- We look at three key ingredients before you do it!

1. Loop over random batches of data
2. Doing calculations with gradients
3. Zeroing gradients!

Looping over random batches of data

```
data = torch.arange(12).reshape(4, 3)
data
```

```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

```
batch_size = 2
batch_indices = torch.randperm(len(data)).reshape(-1, batc
batch_indices
```

```
tensor([[3, 1],
        [2, 0]])
```

```
for idxs in batch_indices:
    batch = data[idxs]
    print(batch)
```

```
tensor([[ 9, 10, 11],
        [ 3,  4,  5]])
tensor([[6, 7, 8],
        [0, 1, 2]])
```

- Draw random permutations of an arange as long as the data
- Reshape them so rows become batch indices
- Loop over batch indices and create batches

Calculations with gradients

```
>>> def f(x):
...     return (x ** 2).sum()


>>> a = torch.tensor(
...     [1., 2., 3.],
...     requires_grad=True,
... )
>>> b = f(a)
>>> b.backward()
>>> a.grad
```

```
tensor([2., 4., 6.])
```

```
>>> a.data = a.data - 0.1 * a.grad.data
>>> a
```

```
tensor([0.8000, 1.6000, 2.4000],
       requires_grad=True)
```

- You cannot just subtract the `a.grad` from a
- Doing so, leads to wrong results
- In the second iteration it also leads to an error

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[227], line 1
----> 1 a = a - 0.1 * a.grad

TypeError: unsupported operand type(s) for *: 'float' and 'NoneType'
```

Important: Zero the gradient

```
a = torch.tensor([1., 2., 3.], requires_grad=True)
b = f(a)
b.backward()
a.grad
```

```
tensor([2., 4., 6.])
```

```
c = f(a)
c.backward()
a.grad
```

```
tensor([ 4.,  8., 12.])
```

```
a = torch.tensor([1., 2., 3.], requires_grad=True)
b = f(a)
b.backward()
a.grad
```

```
tensor([2., 4., 6.])
```

```
a.grad.zero_()
c = f(a)
c.backward()
a.grad
```

```
tensor([2., 4., 6.])
```

When to zero

- There are two possible positions
  - Right before you do the forward pass (i.e. call the model)
  - Right after you do the gradient update
- Need to do this even if you use pre-implemented pytorch models and optimizers
- Gets a bit more convenient then but stays equally dangerous!

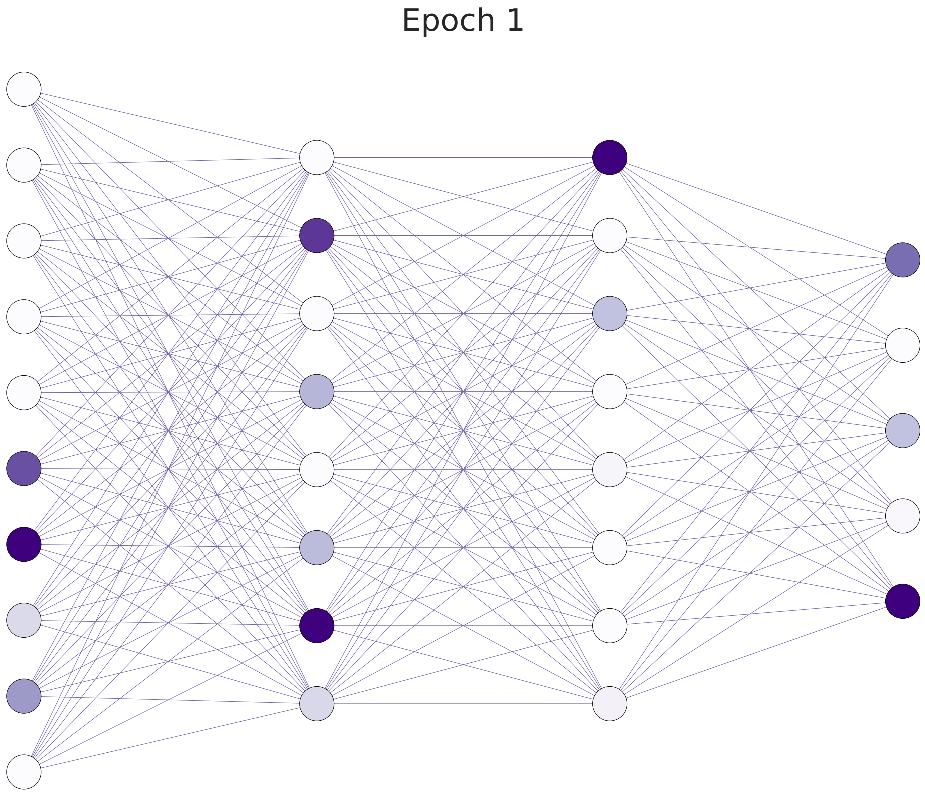
footgun:

any feature whose addition to a product results in the user shooting themselves in the foot

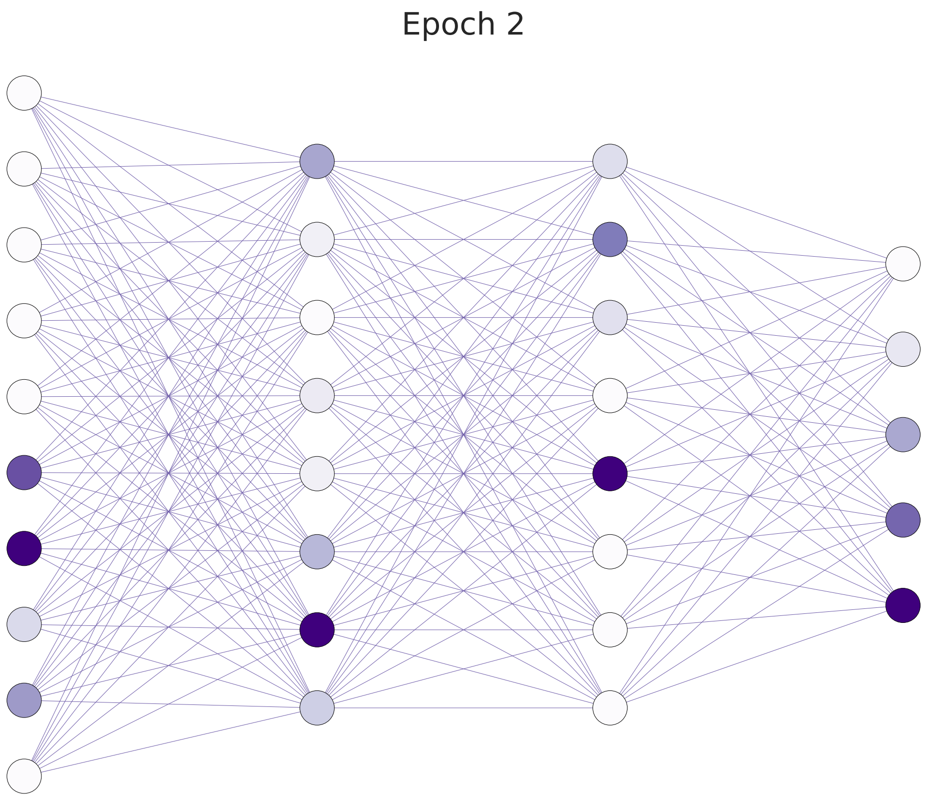
@RobTatman

Task 6

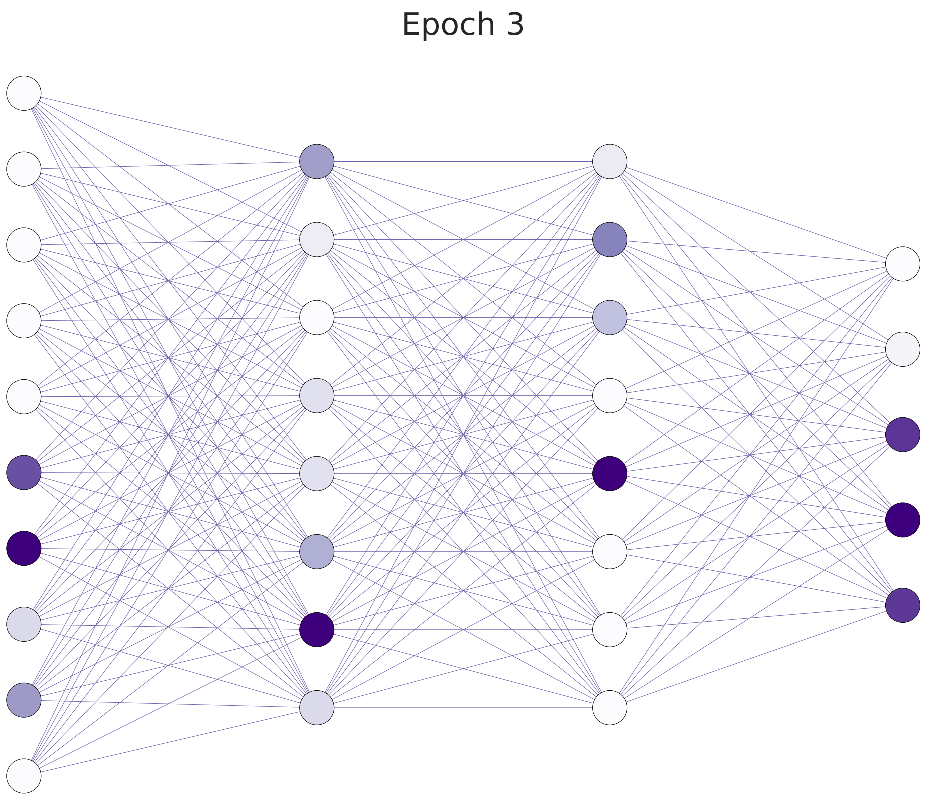
What happens during training



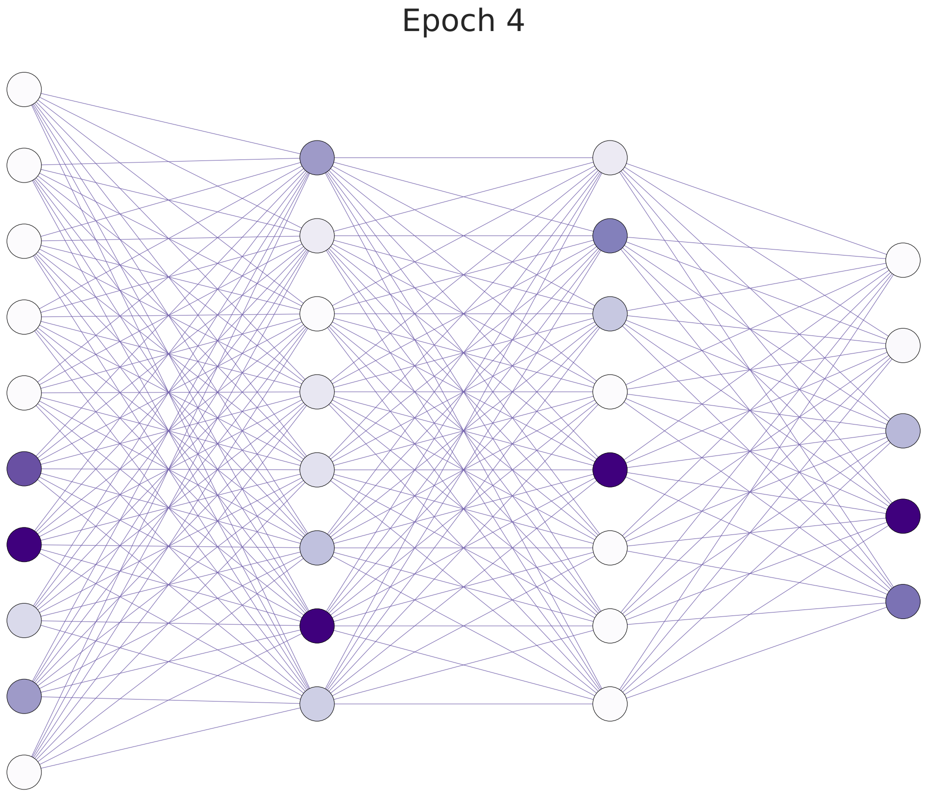
What happens during training



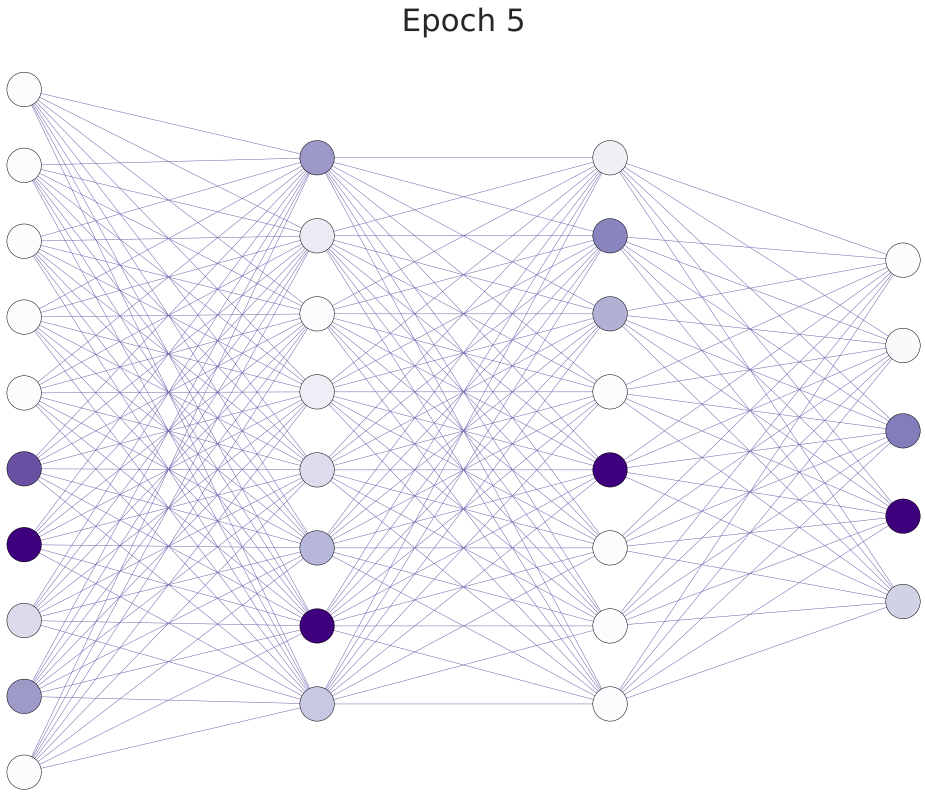
What happens during training



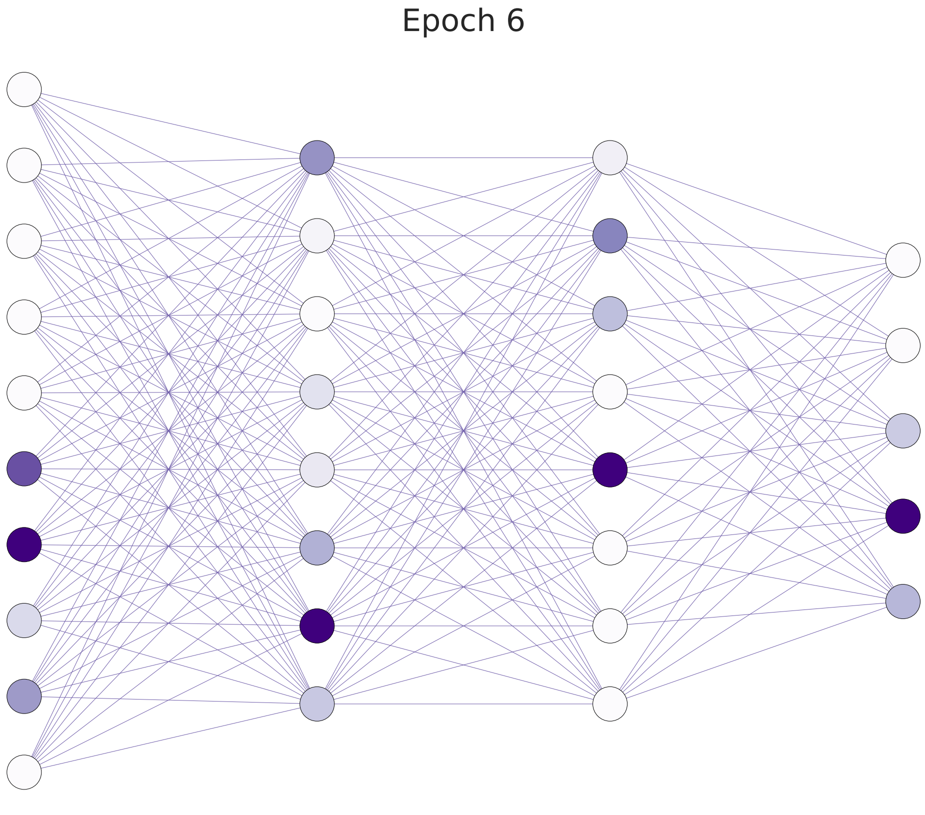
What happens during training



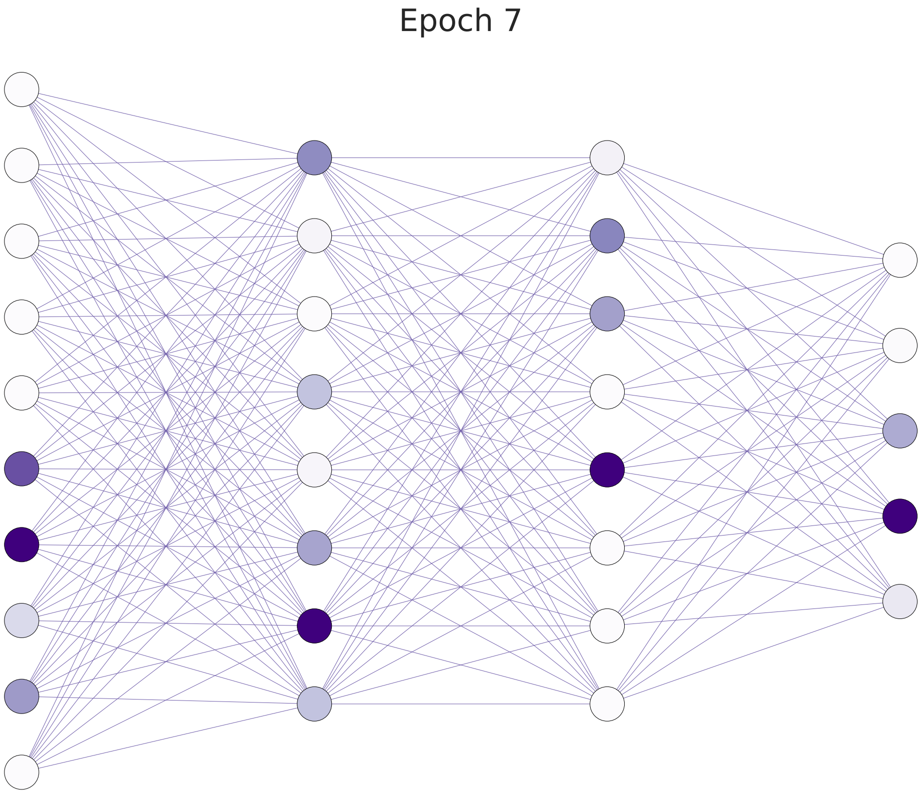
What happens during training



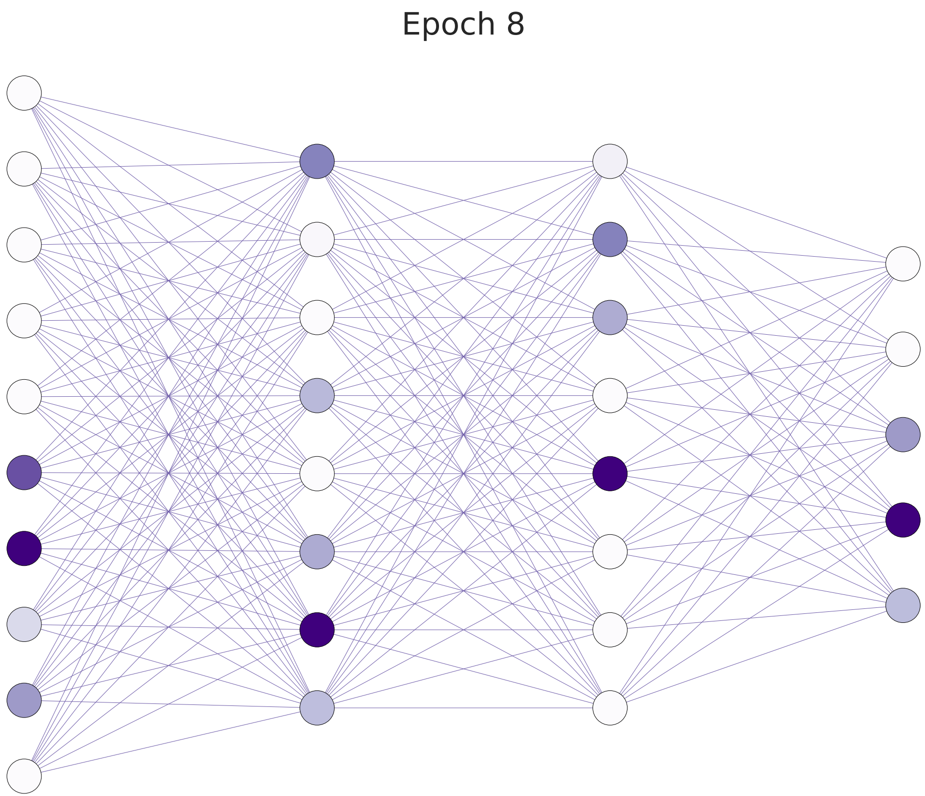
What happens during training



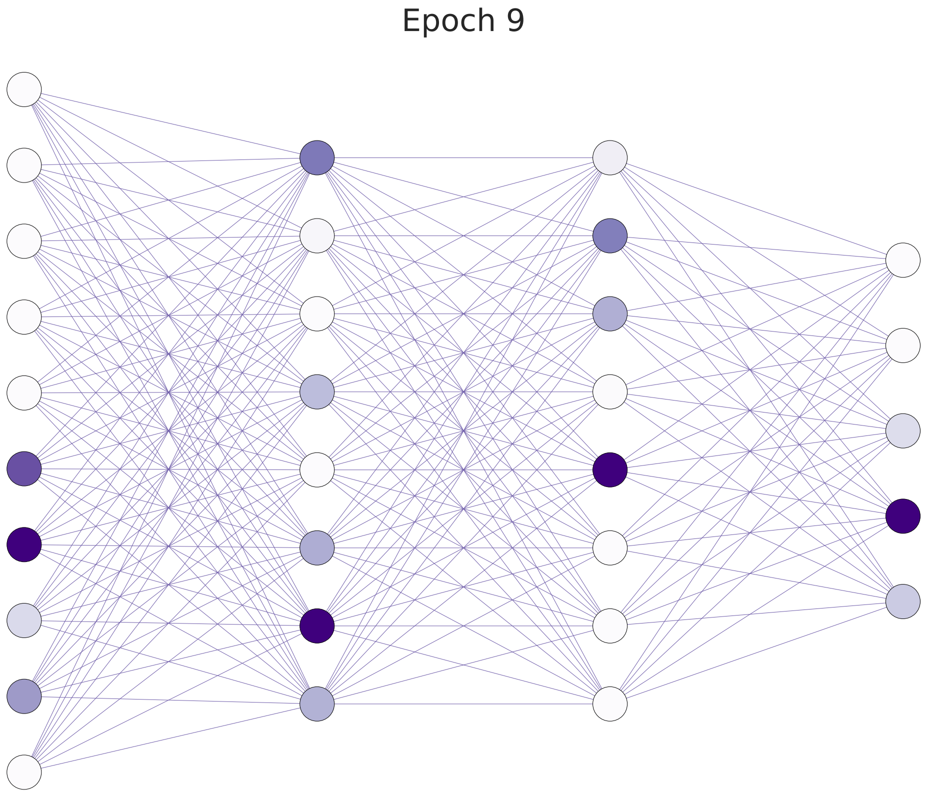
What happens during training



What happens during training



What happens during training



Limits of intuition

- Even in this small network, we only see that the network learns, not how it does that
- Different starting values lead to similar performance but different networks
- The network can switch out where it stores information
- No reason to think that bottom entries are related to bottom pixels or longer lines

Need some monitoring

- After task 7, your output of running the training should look somthing like this

```
Accuracy after epoch 0: 0.4580000042915344
Accuracy after epoch 1: 0.5734999775886536
```

- Numbers can vary, this was on the digits example

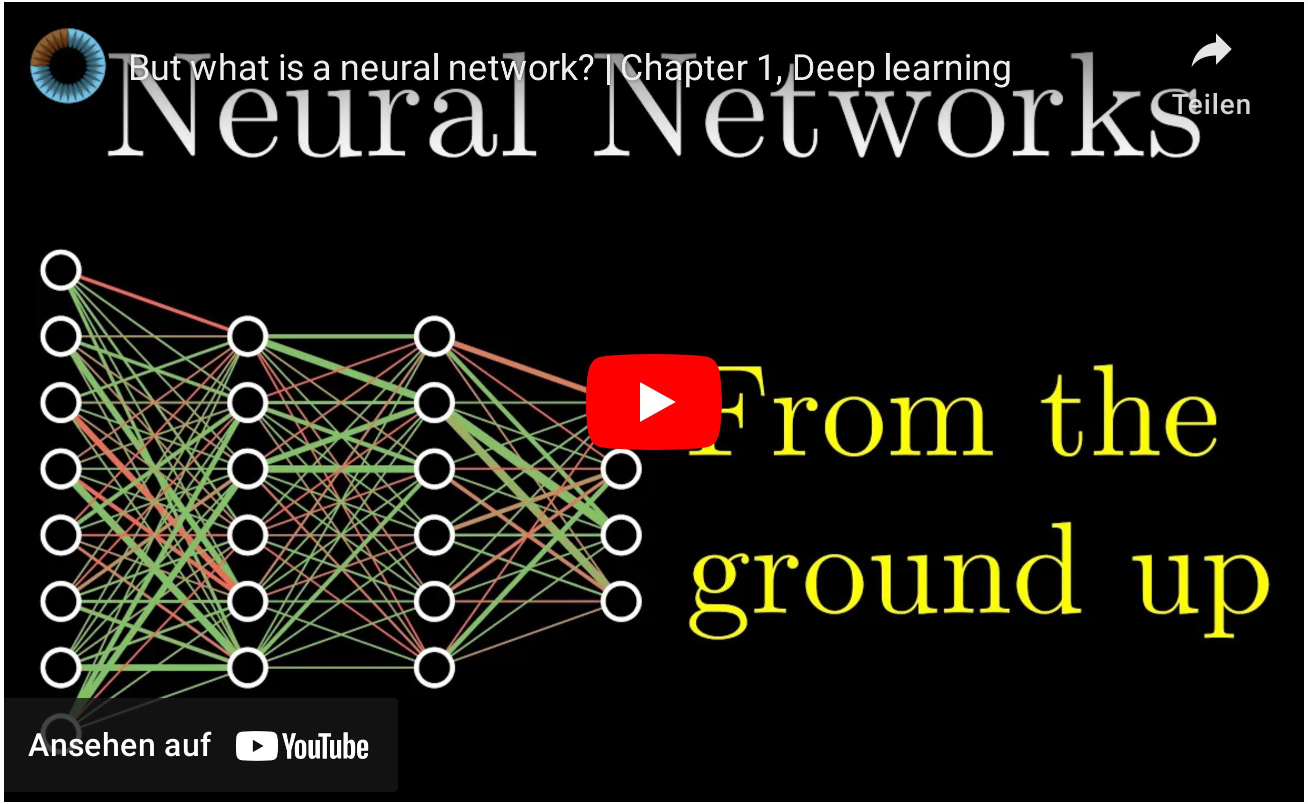
Task 7

The hyperparameters

- Learning rates
  - Want it to be as large as possible for speed
  - Too large can lead to NaNs and non-convergence
  - 0.01 is very small for such a tiny network
- Batch size
  - Want to make it as small as possible for speed
  - Too small means that update become erratic
- Number of epochs
  - For this tiny model we are not very worried about overfitting
  - Thus, more is better but do not exaggerate

Task 8

Video


But what is a neural network? | Chapter 1, Deep learning
Teilen
Neural Networks
From the
ground up
Ansehen auf YouTube