Chapter 3

Implementing Neural Networks

Content of this Chapter

0. Recap
1. Neural Network Libraries
2. Introduction to PyTorch
    1. From Numpy to PyTorch
    2. Implementing a Neural Network
3. Vectorisation of Neural Networks

3.0 Recap

- Classification
- Neural Networks
- Backpropagation
- Gradient Descent

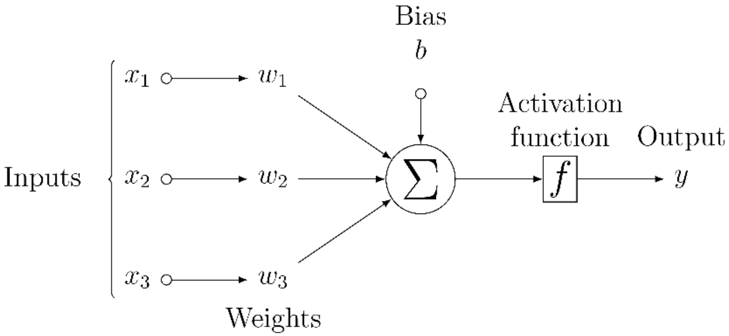
Recap: Classification

- Classification:
  Given data, select a label from a set of possibilities

- Linear classifiers:

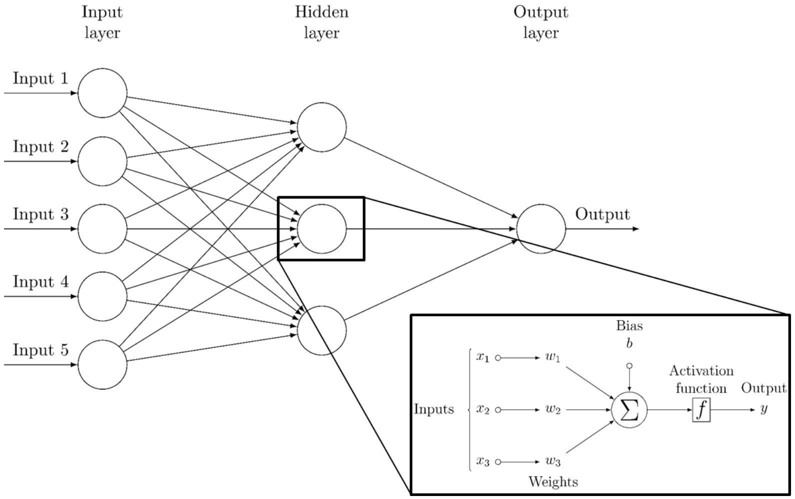
$$y = Wx + b$$

→ Can only model linear relations!



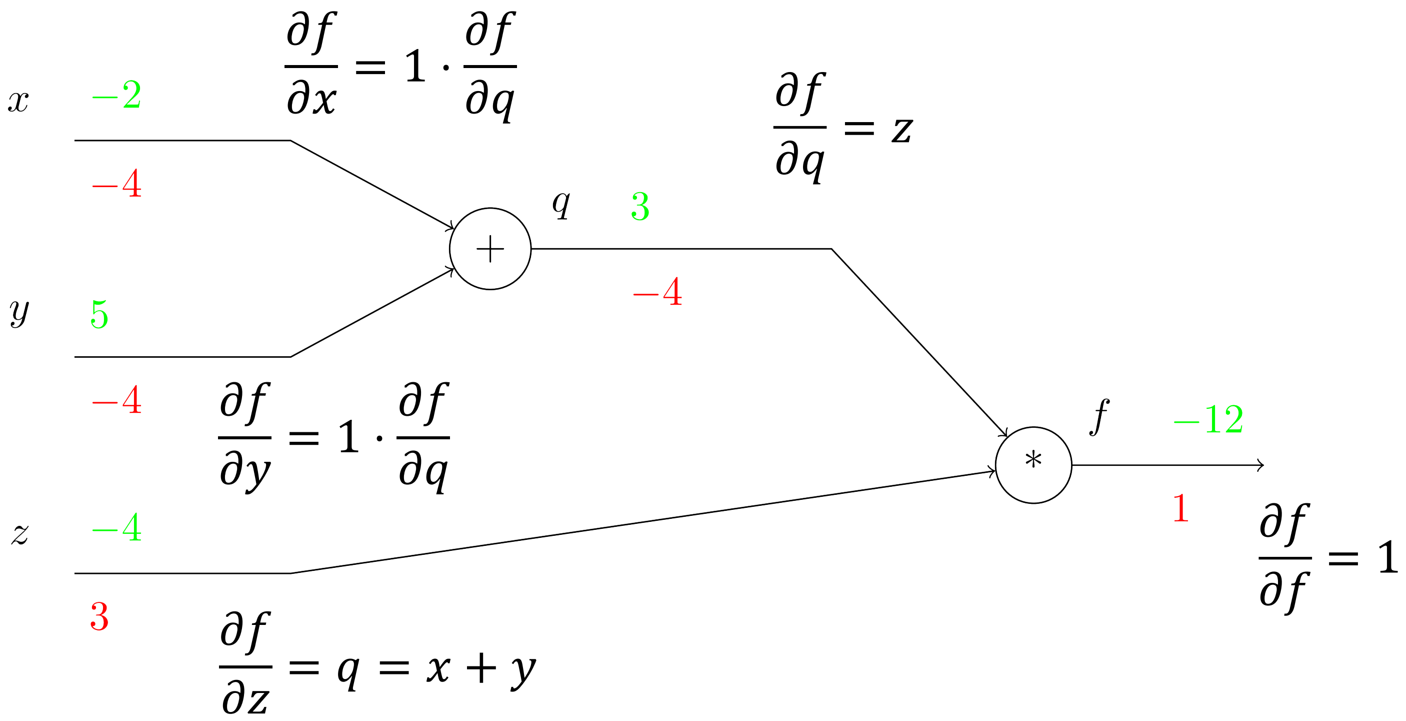
Recap: Neural Networks

- Neural network:
  Multiple linear classifiers with non-linearities in between
- Much more expressive:
  Enables learning (arbitrary) non-linear dependencies
- Loss function:
  How good are the current weights?



Recap: Backpropagation

- Backpropagation ≈ Chain rule with dynamic programming

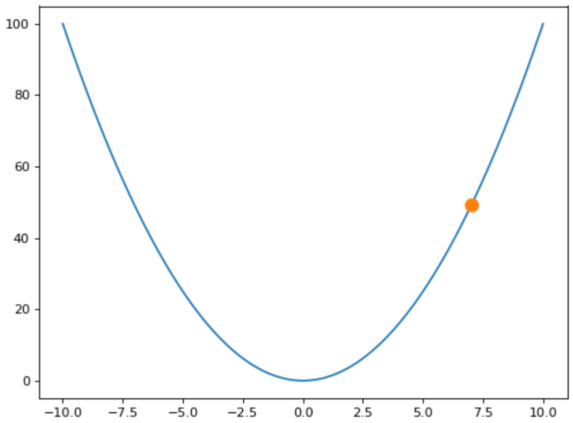

$\frac{\partial f}{\partial x} = 1 \cdot \frac{\partial f}{\partial q}$
x
-2
-4
$\frac{\partial f}{\partial q} = z$
+
q
3
-4
y
5
-4
$\frac{\partial f}{\partial y} = 1 \cdot \frac{\partial f}{\partial q}$
*
f
-12
1
$\frac{\partial f}{\partial f} = 1$
z
-4
3
$\frac{\partial f}{\partial z} = q = x + y$

Recap: Gradient Descent

- Gradient Descent:
  Going in the direction of the steepest descent

- Some optimised versions
  - Less dependent on parameters
  - Faster convergence

$$x \mathrel{+}= -learningrate \cdot \frac{df}{dx}(x)$$

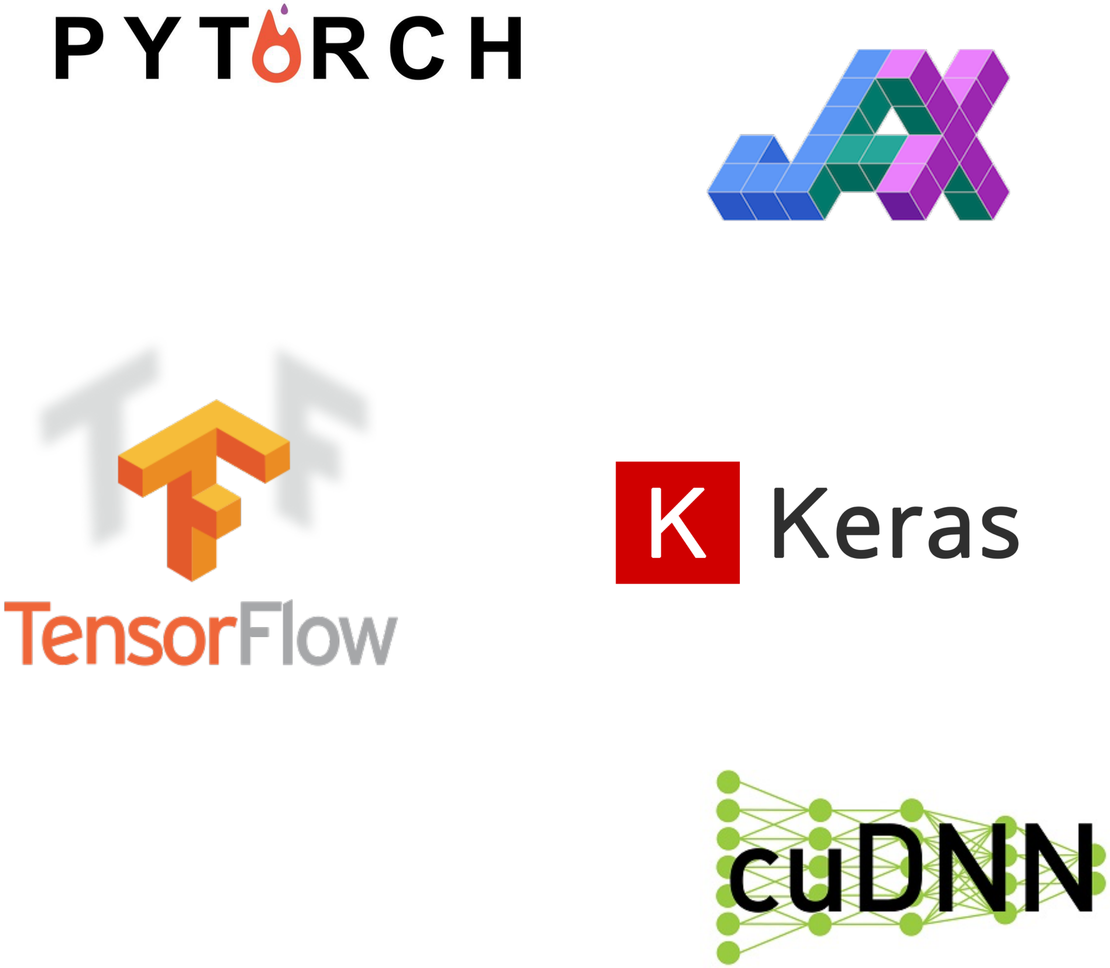


3.1 Neural Network Libraries

- What libraries exist?
- What are their features?
- Why is GPU support so important?
- What do we use here?

Deep Learning Package Zoo

- **PyTorch**
- Tensorflow
- Keras
- JAX
- …



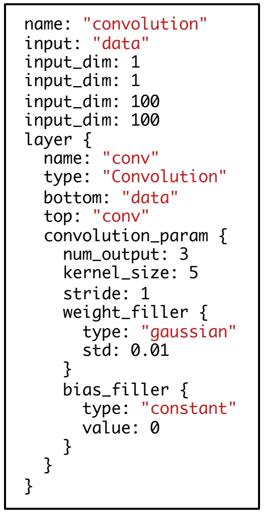
Deep Learning Package Design Choices

- Model specification?
- Computational graph?
- High-level programming language?

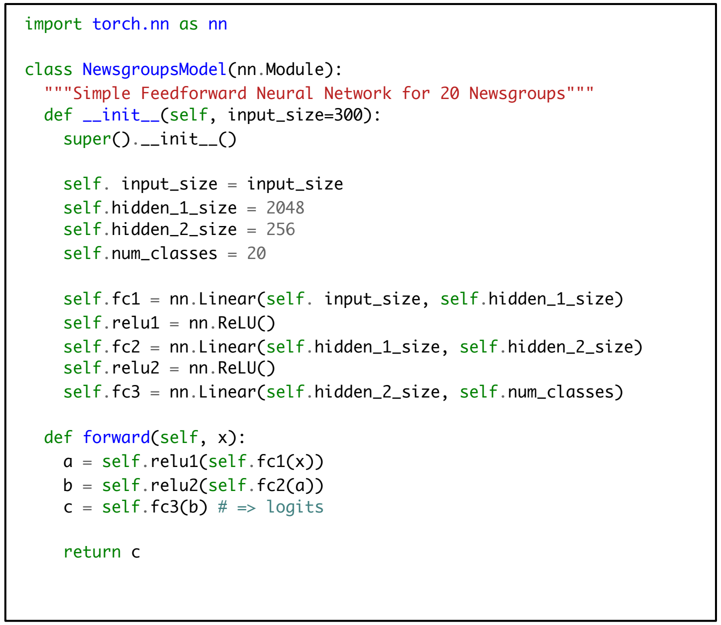
Deep Learning Package Design Choices

Model specification

Configuration file	Programmatic generation
e.g. - Caffe - DistBelief - CNTK	e.g. - PyTorch - Theano - Tensorflow



```
name: "convolution"
input: "data"
input_dim: 1
input_dim: 1
input_dim: 100
input_dim: 100
layer {
  name: "conv"
  type: "Convolution"
  bottom: "data"
  top: "conv"
  convolution_param {
    num_output: 3
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

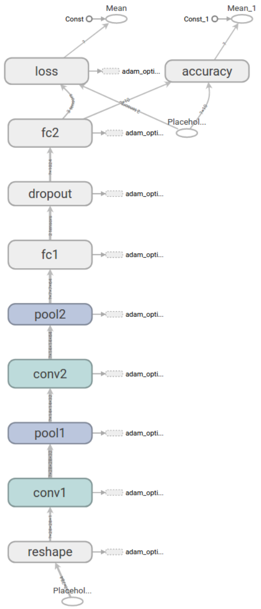


```
import torch.nn as nn

class NewsgroupsModel(nn.Module):
  """Simple Feedforward Neural Network for 20 Newsgroups"""
  def __init__(self, input_size=300):
    super().__init__()

    self. input_size = input_size
    self.hidden_1_size = 2048
    self.hidden_2_size = 256
    self.num_classes = 20

    self.fc1 = nn.Linear(self. input_size, self.hidden_1_size)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(self.hidden_1_size, self.hidden_2_size)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(self.hidden_2_size, self.num_classes)

  def forward(self, x):
    a = self.relu1(self.fc1(x))
    b = self.relu2(self.fc2(a))
    c = self.fc3(b) # => logits

    return c
```

Deep Learning Package Design Choices

Computational graph

Static	Dynamic
- set before execution - can be optimized up front - e.g. default in Tensorflow 1	- set during forward pass - can be changed dynamically during execution - e.g. in PyTorch & Tensorflow 2



computational graph of a convolutional neural network:

- contains the program‘s operations and variables as a directed acyclic graph
- looks a lot like the circuit diagram from last lecture

Deep Learning Package Design Choices

High-level programming language:

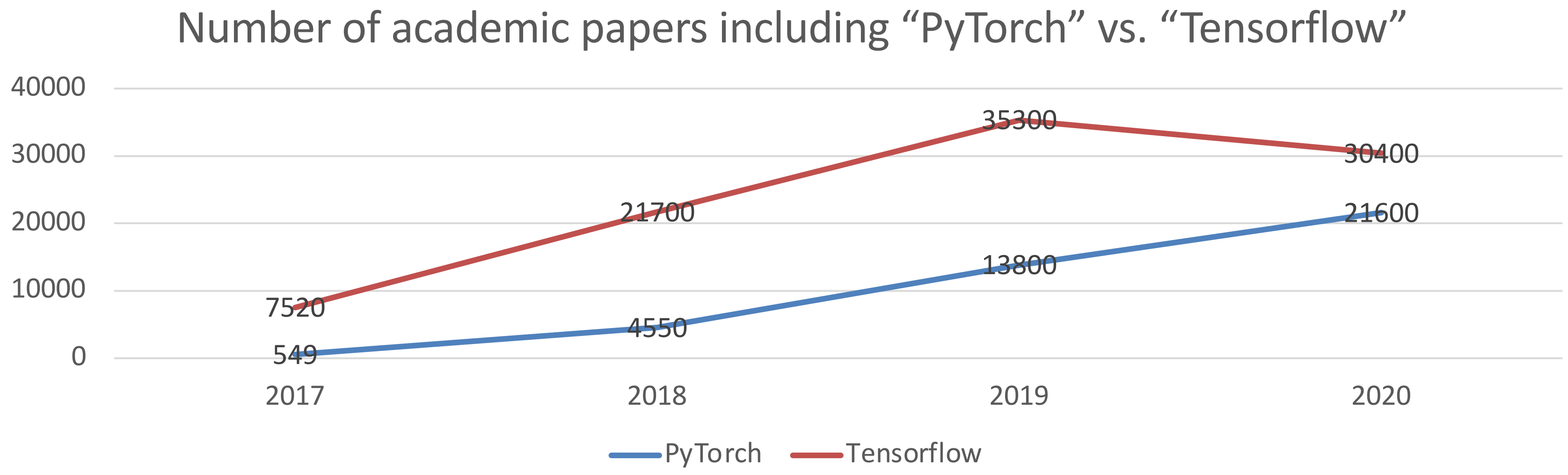
- Lua (Torch)
- Python (Theano, Tensorflow, PyTorch, JAX)
- …

- We choose **PyTorch** because it is easy to understand for Python users

PyTorch

https://pytorch.org

- PyTorch is
  - A deep learning library (but not only that)
  - Written in Python, but based on Torch that was written in Lua
  - Developed by Facebook AI research group
  - Open Source since February 2017 ☺
  - Since then, steadily growing, especially in NLP research!



PyTorch

- PyTorch provides
  - Predefined function to compute gradients, optimisers, …
  - Simple, object-oriented interface
  - ONNX (Open Neural Network Exchange) support to easily use trained models in other frameworks (e.g. on mobile devices, on the web, ...)
  - **GPU support!**

Using GPUs

- Why GPU support?

→ Neural network operations are mostly matrix multiplications

→ Matrix multiplications can be parallelized very effectively!

→ GPUs excell at tasks where the same operation is done on many data points

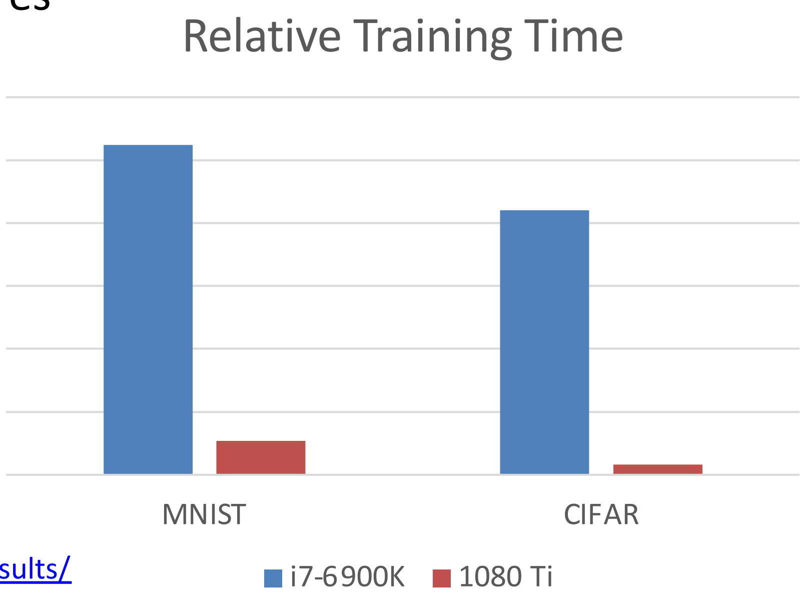
$$BE = \begin{bmatrix} 8 & 1 & 2 \\ -5 & 6 & 7 \end{bmatrix} \begin{bmatrix} -5 & 1 \\ 0 & 2 \\ -11 & 7 \end{bmatrix}$$

$$= \begin{bmatrix} (8)(-5)+(1)(0)+(2)(-11) & (8)(1)+(1)(2)+(2)(7) \\ (-5)(-5)+(6)(0)+(7)(-11) & (-5)(1)+(6)(2)+(7)(7) \end{bmatrix}$$

$$= \begin{bmatrix} -62 & 24 \\ -52 & 56 \end{bmatrix}$$

GPU vs CPU

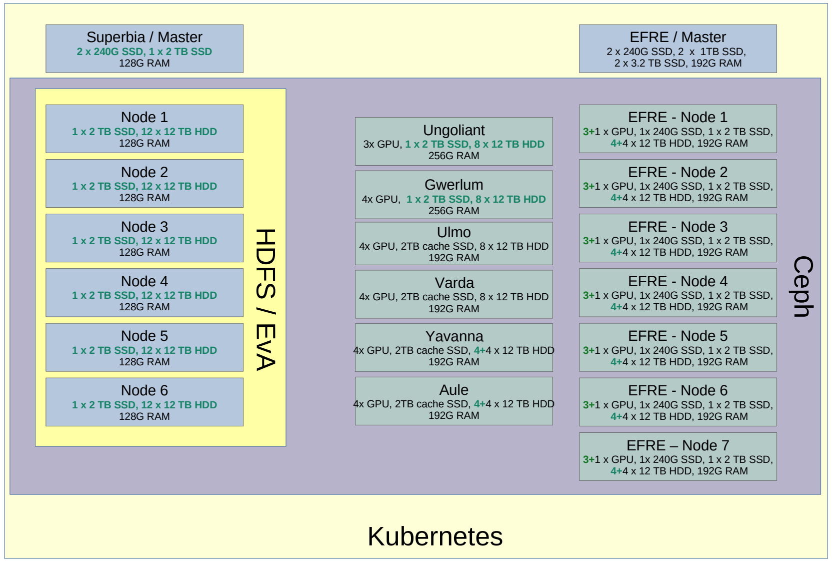
- CPUs
  - Very few (~2 – 64) complex cores
  - Great for complex tasks
  - Not so great for simple, parallel tasks
- GPUs
  - Have a lot (~2500 – 3800) less complex cores
  - Can do huge matrix mults in parallel!



For full results see https://mlperf.org/results/

Deep Learning at our group

- Steadily growing server infrastructure
- **71 GPUs** in 22 deep learning servers, planning on adding further GPUs in the future
- Using Kubernetes to automatically manage resources


Superbia / Master
2 x 240G SSD, 1 x 2 TB SSD
128G RAM

EFRE / Master
2 x 240G SSD, 2 x 1TB SSD,
2 x 3.2 TB SSD, 192G RAM

Node 1
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

Node 2
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

Node 3
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

Node 4
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

Node 5
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

Node 6
1 x 2 TB SSD, 12 x 12 TB HDD
128G RAM

HDFS / EvA

Ungoliant
3x GPU, 1 x 2 TB SSD, 8 x 12 TB HDD
256G RAM

Gwerlum
4x GPU, 1 x 2 TB SSD, 8 x 12 TB HDD
256G RAM

Ulmo
4x GPU, 2TB cache SSD, 8 x 12 TB HDD
192G RAM

Varda
4x GPU, 2TB cache SSD, 8 x 12 TB HDD
192G RAM

Yavanna
4x GPU, 2TB cache SSD, 4+4 x 12 TB HDD
192G RAM

Aule
4x GPU, 2TB cache SSD, 4+4 x 12 TB HDD
192G RAM

EFRE - Node 1
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE - Node 2
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE - Node 3
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE - Node 4
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE - Node 5
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE - Node 6
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

EFRE – Node 7
3+1 x GPU, 1x 240G SSD, 1 x 2 TB SSD,
4+4 x 12 TB HDD, 192G RAM

Ceph

Kubernetes

3.2 PyTorch

- How to use PyTorch?

From Numpy to PyTorch

http://www.numpy.org

- You may know Numpy
  - Math library for Python
  - Provides efficient implementation (C backend) for many common operations

- A simple program in Numpy:

```
In [23]: import numpy as np

In [24]: a = np.zeros((2,2)); b = np.ones((2,2))
In [25]: np.sum(b, axis=1)
Out[25]: array([ 2., 2.])

In [26]: a.shape
Out[26]: (2, 2)

In [27]: np.reshape(a, (1,4))
Out[27]: array([[ 0., 0., 0., 0.]])
```

From Numpy to PyTorch

- PyTorch was intended as a Numpy replacement with GPU support
- Therefore very similar concepts and code:

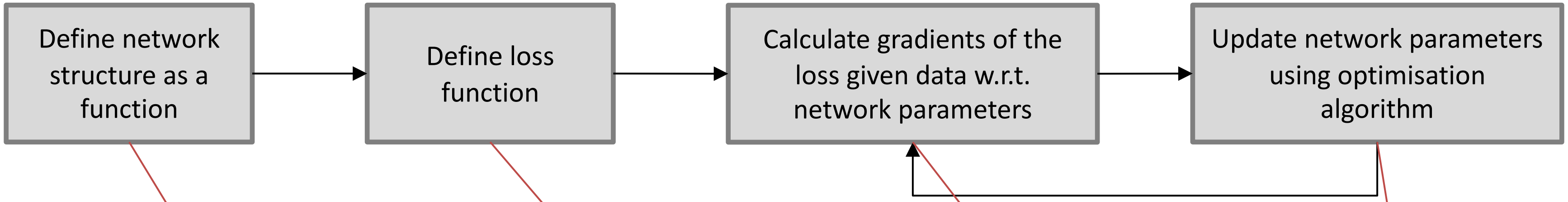
```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.ones((2, 2))

np.sum(b, axis=1)
# array([2., 2.])

a.shape
# (2, 2)

np.reshape(a, (1, 4))
# array([[1, 2, 3, 4]])
```

```
import torch

a = torch.tensor([[1, 2], [3, 4]])
b = torch.ones((2, 2))

torch.sum(b, dim=1)
# tensor([2., 2.])

a.shape
# torch.Size([2, 2])

torch.reshape(a, (1, 4))
# tensor([[1, 2, 3, 4]])

a.numpy()
# array([[1, 2],
#        [3, 4]])
```

possible to extract a numpy
array from a torch tensor

Neural Networks with PyTorch

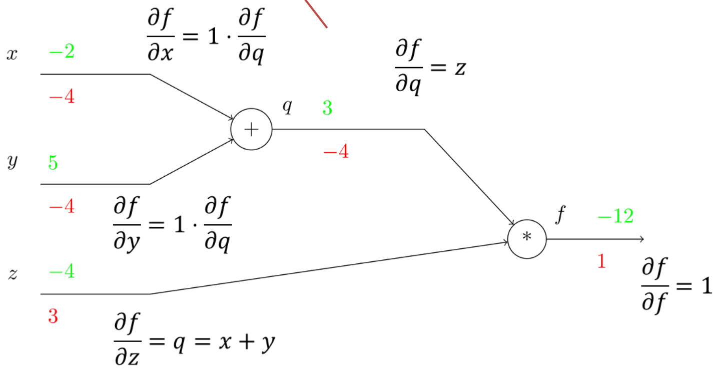
You already know this


Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

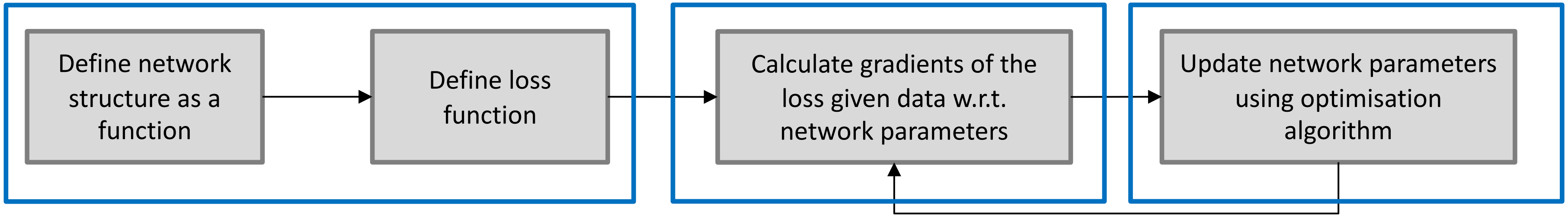
$$x \mathrel{+}= -learningrate \cdot \frac{df}{dx}(x)$$

$$L_\theta(l, x) = -\sum_i l[i] \cdot \log y_\theta(x)[i]$$

$$y = softmax(W_3 \tanh(W_2 \tanh(W_1 x + b_1) + b_2) + b_3)$$


$\frac{\partial f}{\partial x} = 1 \cdot \frac{\partial f}{\partial q}$
x
-2
-4
$\frac{\partial f}{\partial q} = z$
q
3
-4
$+$
y
5
-4
$\frac{\partial f}{\partial y} = 1 \cdot \frac{\partial f}{\partial q}$
f
-12
1
$*$
$\frac{\partial f}{\partial f} = 1$
z
-4
3
$\frac{\partial f}{\partial z} = q = x + y$

Neural Networks with PyTorch

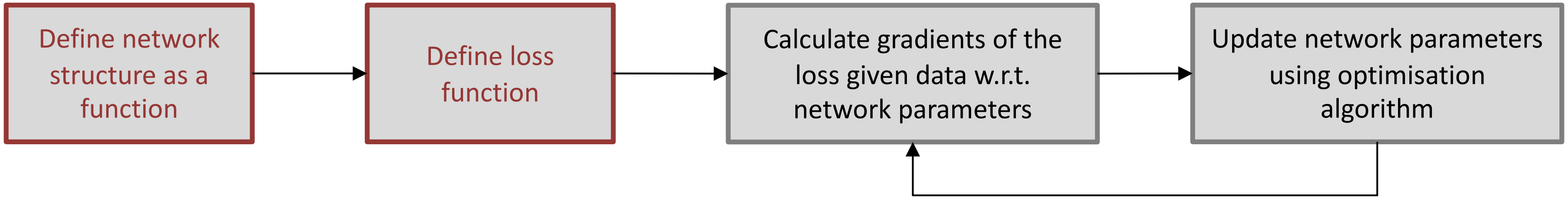

Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

- How to do this in PyTorch?

→ Three modules corresponding to these steps:

  – nn Module
  – Autograd module
  – Optim module

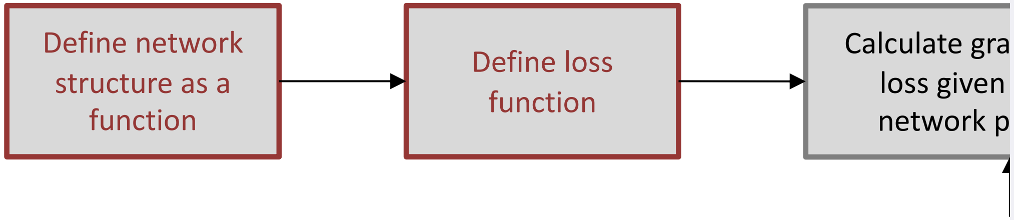
PyTorch Modules


Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

nn module

- Collection of neural network related building blocks, e.g.
  - fully connected layers
  - common activation functions
  - common loss functions
  - …
- Handles weights and biases internally
  → hides complexity!

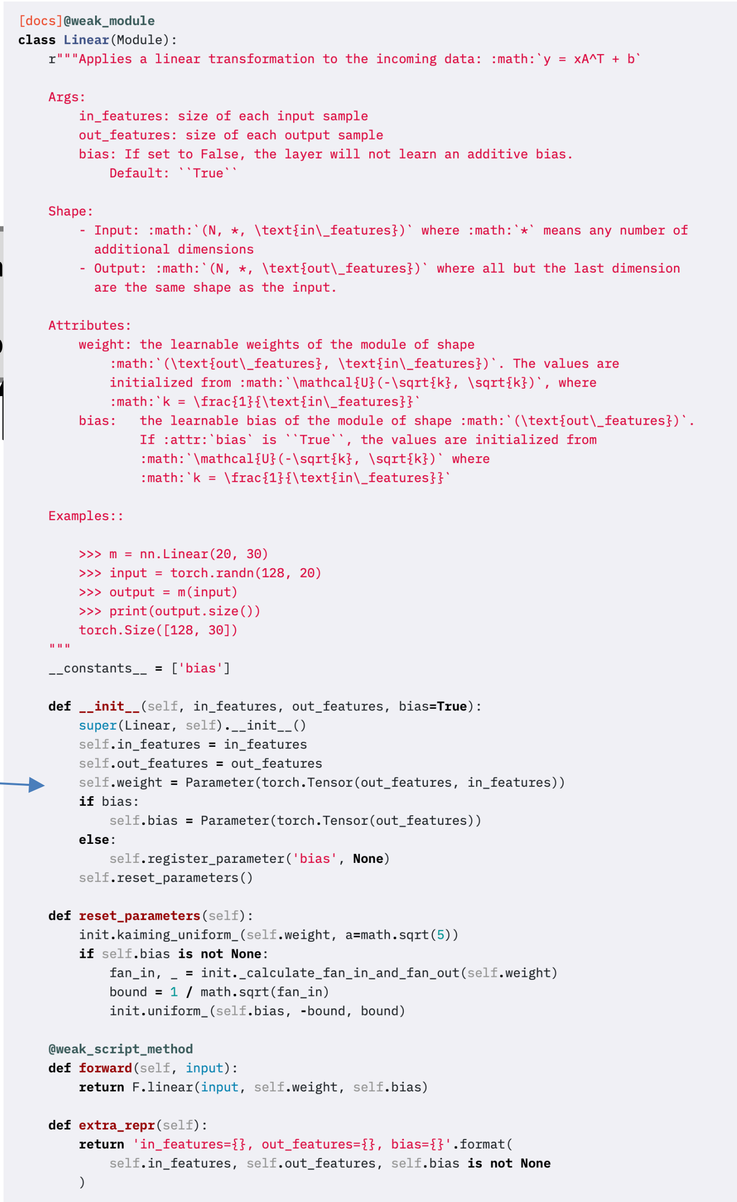
PyTorch Modules — Example



```
# Feedforward layer
torch.nn.Linear(...)
# Convolutional layer
torch.nn.Conv2d(...)
# ReLU activation
torch.nn.ReLU()
# Cross Entropy Loss
torch.nn.CrossEntropyLoss()
```

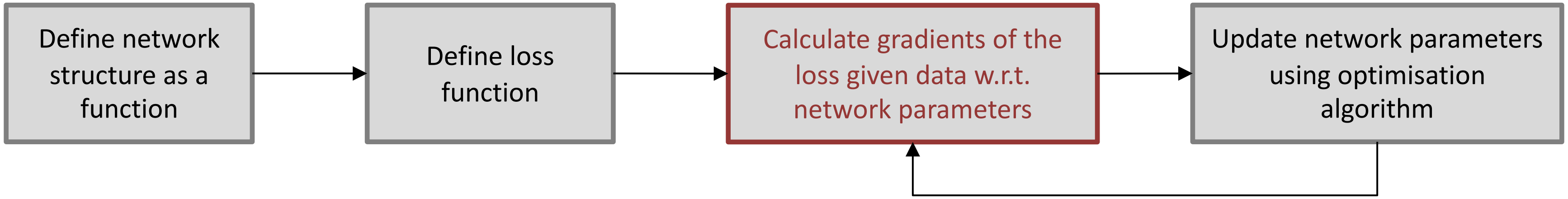
weights and bias

- Tested code
- Object-oriented
- Automatically handles things like weight initialisation
- Easy to switch to other layers



```
[docs]@weak_module
class Linear(Module):
    r"""Applies a linear transformation to the incoming data: :math:`y = xA^T + b`

    Args:
        in_features: size of each input sample
        out_features: size of each output sample
        bias: If set to False, the layer will not learn an additive bias.
            Default: ``True``

    Shape:
        - Input: :math:`(N, *, \text{in\_features})` where :math:`*` means any number of
          additional dimensions
        - Output: :math:`(N, *, \text{out\_features})` where all but the last dimension
          are the same shape as the input.

    Attributes:
        weight: the learnable weights of the module of shape
            :math:`(\text{out\_features}, \text{in\_features})`. The values are
            initialized from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})`, where
            :math:`k = \frac{1}{\text{in\_features}}`
        bias:   the learnable bias of the module of shape :math:`(\text{out\_features})`.
                If :attr:`bias` is ``True``, the values are initialized from
                :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                :math:`k = \frac{1}{\text{in\_features}}`

    Examples::

        >>> m = nn.Linear(20, 30)
        >>> input = torch.randn(128, 20)
        >>> output = m(input)
        >>> print(output.size())
        torch.Size([128, 30])
    """
    __constants__ = ['bias']

    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)

    @weak_script_method
    def forward(self, input):
        return F.linear(input, self.weight, self.bias)

    def extra_repr(self):
        return 'in_features={}, out_features={}, bias={}'.format(
            self.in_features, self.out_features, self.bias is not None
        )
```

PyTorch Modules


Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

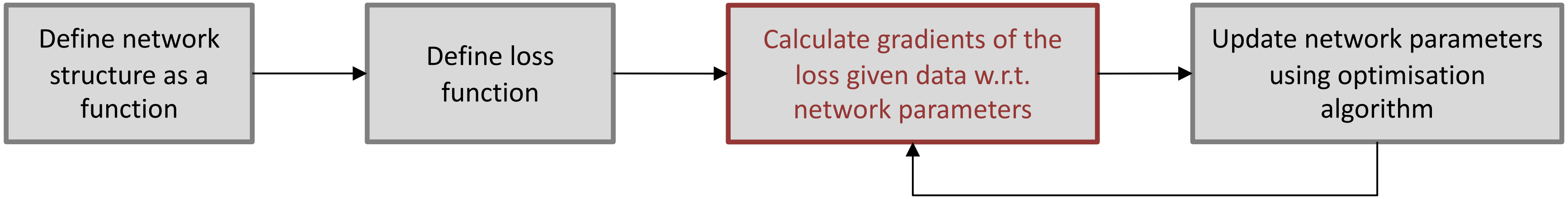
Autograd module

- Gradient computation necessary for gradient descent

- Autograd abstracts backpropagation away
  - Builds computation graph
  - Performs differentiation by chain rule

→ No differentiation by hand ☺

PyTorch Modules — Example



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

Update network parameters using optimisation algorithm

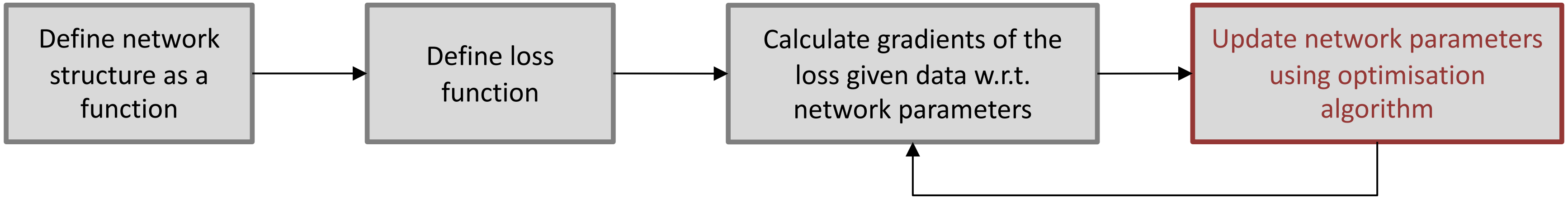
```
import torch

a = torch.tensor([[1,2],[3,4]], requires_grad=True)
# 1  2
# 3  4
y = torch.sum(a**2)
# => y = 1 + 4 + 9 + 16 = 30


# compute gradients of y w.r.t. to all Variables
y.backward()


a.grad # contains the gradient of y w.r.t. every entry of a
# 2  4
# 6  8
```

We want to calculate gradients w.r.t. this tensor

$$a = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

$$y = \sum_{a_{ij}} a_{ij}^2 = a_{00}^2 + a_{10}^2 + a_{01}^2 + a_{11}^2$$

PyTorch Modules


Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

Optim module

- Defines multiple optimisation algorithms
  - SGD
  - Adam optimiser
  - …
- Consistent interface → easily interchangeable

These modules make it easy to build neural networks. Let‘s do this!

20 Newsgroups Text Classification

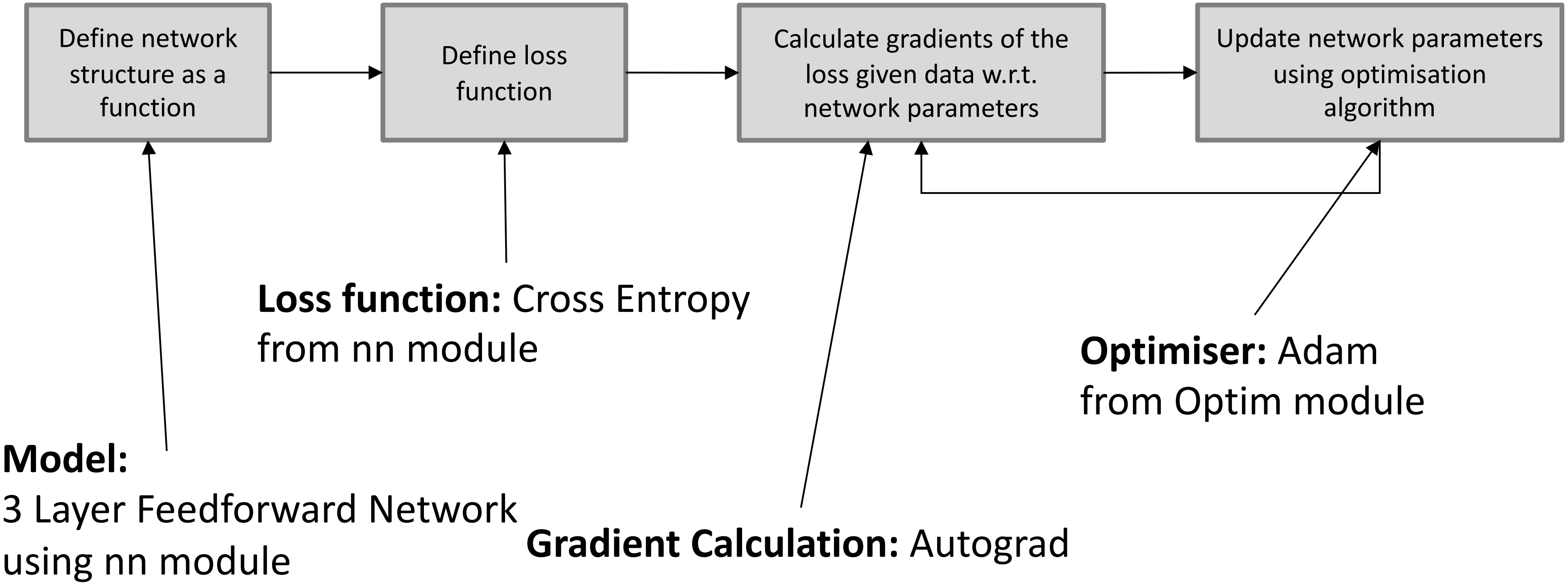
You know this already!

- 20 Newsgroups is a dataset of online discussion
  - 18,828 documents total
  - 20 forums, some closely related (pc.hardware vs mac.hardware), some highly unrelated (misc.forsale vs religion.christian)

  → Classify, which forum each document originates from
- Popular evaluation set in Natural Language Processing
- Not completely solved (Error rate ~ 11.4%)

Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr., Christopher Fifty, Tao Yu, & Kilian Q. Weinberger. (2019). *Simplifying Graph Convolutional Networks*

PyTorch – Building a Neural Network



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

Update network parameters using optimisation algorithm

Loss function: Cross Entropy from nn module

Optimiser: Adam from Optim module

Model:
3 Layer Feedforward Network using nn module

Gradient Calculation: Autograd

→ Let‘s build a classifier for 20 Newsgroups!

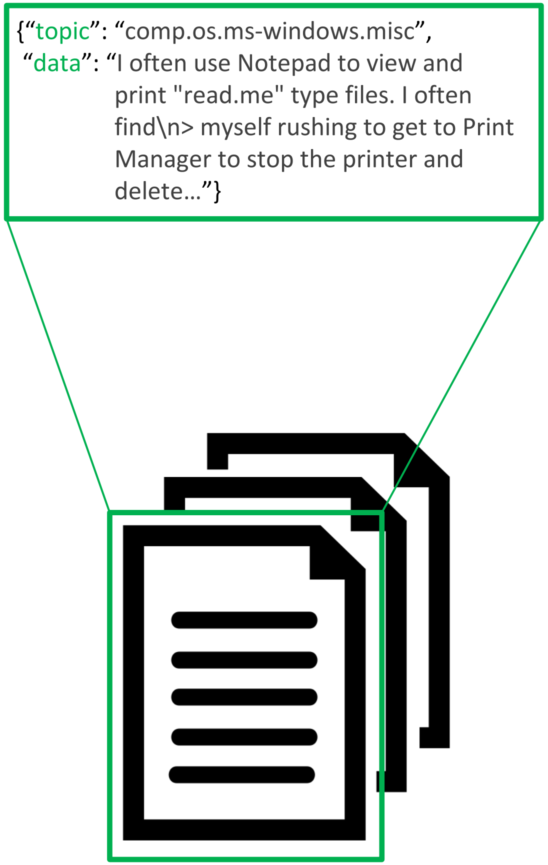
But first: How do we get the data into the network?

Data Input and Preprocessing

PyTorch – Dataset

- Given:
  - A dataset of documents
  - For each document i
    - An attribute „data“ containing the raw unprocessed text
    - An attribute „topic“ containing the label

- We will build a Dataset class for 20 Newsgroups



```
{“topic”: “comp.os.ms-windows.misc”,
 “data”: “I often use Notepad to view and
             print "read.me" type files. I often
             find\n> myself rushing to get to Print
             Manager to stop the printer and
             delete…”}
```

PyTorch – Dataset class

- PyTorch provides an abstract class representing a dataset
- Our Dataset class should inherit from it and overwrite the following methods:
  - `__len__` returns the length of our dataset
  - `__getitem__` returns a data point given an index

- `__getitem__` can load a specific data point on demand
→ no need to load the entire dataset

```
import numpy as np
from torch.utils.data import Dataset


class NewsgroupsDataset(Dataset):
    """20 Newsgroups Dataset"""
    def __init__(self , data: list, labels: dict):
```

initialise everything

```
    def __len__(self):
        """Returns the size of the dataset"""
```

get the number of examples

```
    def __getitem__(self, idx: int):
        """Returns a data point (text and label) given an index"""
```

get an example and label
for a given index

```
import numpy as np
from torch.utils.data import Dataset


class NewsgroupsDataset(Dataset):
    """20 Newsgroups Dataset"""
    def __init__(self , data: list, labels: dict):
        super().__init__()
        self.data = data  # [{"data": "Hello there …", "label": "misc.forsale"}, {"data": "…", "label":
"…"}, …]
        self.labels = labels  # {"misc.forsale": 0, "sci.space": 1, …}
```

initialise everything

```
    def __len__(self):
        """Returns the size of the dataset"""
```

get the number of examples

```
    def __getitem__(self, idx: int):
        """Returns a data point (text and label) given an index"""
```

get an example and label
for a given index

```
import numpy as np
from torch.utils.data import Dataset


class NewsgroupsDataset(Dataset):
    """20 Newsgroups Dataset"""
    def __init__(self , data: list, labels: dict):
        super().__init__()
        self.data = data  # [{"data": "Hello there …", "label": "misc.forsale"}, {"data": "…", "label":
"…"}, …]
        self.labels = labels  # {"misc.forsale": 0, "sci.space": 1, …}
```

initialise everything

```
    def __len__(self):
        """Returns the size of the dataset"""
        return len(self.data)
```

get the number of examples

```
    def __getitem__(self, idx: int):
        """Returns a data point (text and label) given an index"""
```

get an example and label
for a given index

lass indices

```
import numpy as np
from torch.utils.data import Dataset


class NewsgroupsDataset(Dataset):
    """20 Newsgroups Dataset"""
    def __init__(self , data: list, labels: dict):
        super().__init__()
        self.data = data  # [{"data": "Hello there …", "label": "misc.forsale"}, {"data": "…", "label":
"…"}, …]
        self.labels = labels  # {"misc.forsale": 0, "sci.space": 1, …}
```

initialise everything

```
    def __len__(self):
        """Returns the size of the dataset"""
        return len(self.data)
```

get the number of examples

```
    def __getitem__(self, idx: int):
        """Returns a data point (text and label) given an index"""
        text = self.data[idx]["data"]  # load the raw text from the document with the given id
        text = self.preprocess(text)  # clean text (e.g. lowercasing, …) and create embedding vector
        text = torch.from_numpy(text).float()  # network inputs need to be float

        label = self.data[idx]["topic"] # load the true label of the document with the given id
        label = self.labels[label]  # lookup integer value of the string label
        label = torch.tensor(label).long()  # label is not a continuous value but class indices

        return text, label
```

get an example and label
for a given index

PyTorch – Dataset

- Now we can retrieve an element using an index from `0` to `len(dataset)-1`

- One more thing:
  Neural Networks use *batches* for training:
  - Concatenate multiple examples to a higher dimensional matrix
  - Train on multiple examples at once

→ This is handled by a DataLoader in PyTorch

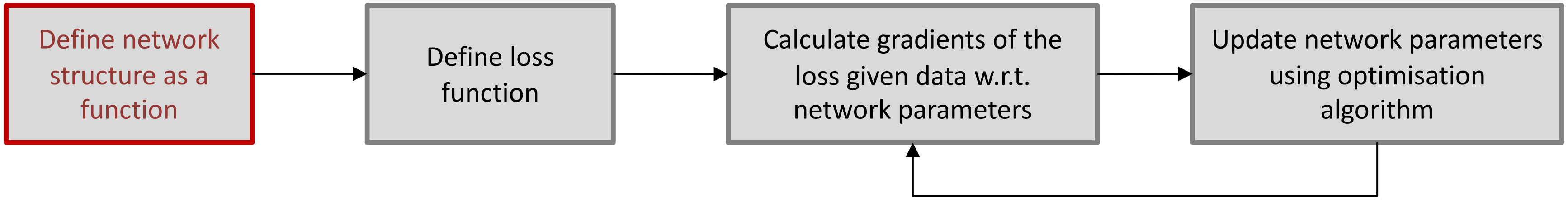
- Using a DataLoader also allows for:
  - Shuffling: Shuffle a list of all available indices and iterate over it
  - Parallel loading of items: Query the `__getitem__` method from multiple threads

```
# gensim provides a module for downloading datasets/models
import gensim.downloader as api
from torch.utils.data import DataLoader


data = list()
label_mask = set()
for document in api.load("20-newsgroups"):
    data.append(document)
    label_mask.add(document["topic"])
# assign an unique integer to every string label
label_mask = {label: index for index, label in enumerate(label_mask)}

dataset = NewsgroupsDataset(data, label_mask)


data_loader = torch.utils.data.DataLoader(dataset, batch_size=512,
shuffle=True, num_workers=2)


for data in data_loader:
    inputs, labels = data

    # feed inputs through network
    # calculate loss based on output and labels
    # ...
```

create a DataLoader by handing in our dataset

iterable that yields data batches and their labels that can be used in model training and testing

Building the Model



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

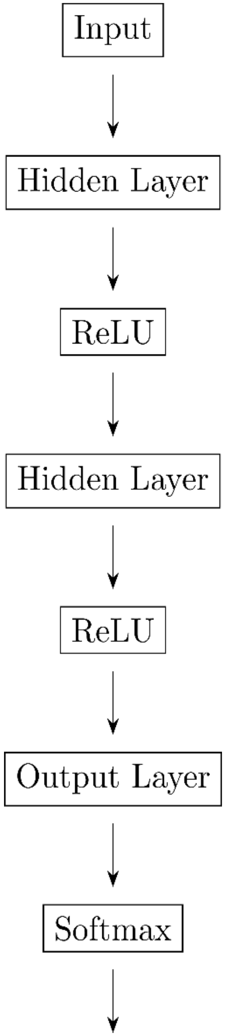
Update network parameters using optimisation algorithm

Two ways:

- The **hard/low-level** (but educational 😉) way
- The **easy/high-level** way

PyTorch – Building the Model

- PyTorch provides an abstract class `Module`
  (nearly everything that modifies tensors is a Module)
- Mainly two methods to override:
  - `forward`: feed incoming values through the network
  - `backward`: defines the gradient of the operation → Autograd handles this for us (when only using PyTorch operations), so usually no need to implement this
- We will build a simple feedforward neural network
  - With 2 hidden layers
  - ReLU activation
  - And a softmax output layer



Building Models:
The **Hard/Low-Level** Way

Useful when:

- PyTorch does not (yet) provide layers you need
- you want to do other kinds of calculations

Low-Level Model Building

- Recall: A simple feedforward layer can be written as

$$y = \sigma(Wx + b)$$

- We can use PyTorch‘s functions that are inspired by Numpy to implement this!
- We need:
  - A matrix W
  - A bias term b
  - Operations
    - matrix multiplication
    - addition
    - element-wise operations for the activation function

Low-Level Model Building

- **One addition:** Up until now, we always used *column vectors*
- PyTorch, however, uses *row vectors*

- Therefore, the linear classifier

$$y = \sigma(Wx + b)$$

needs to be rewritten as

$$y = \sigma(xW + b)$$

to be calculable.

```
class NewsgroupsModelLowLevel(nn.Module):
    """Simple Feedforward Neural Network for 20 Newsgroups"""
    def __init__(self, input_size=300):
        super().__init__()
        self.input_size = input_size
        self.hidden_1_size = 2048
        self.hidden_2_size = 256
        self.num_classes = 20

        self.W1 = nn.Parameter(torch.randn(self. input_size, self.hidden_1_size, requires_grad=True))
        self.b1 = nn.Parameter(torch.randn(1, self.hidden_1_size, requires_grad=True))
        self.relu1 = nn.ReLU()
        self.W2 = nn.Parameter(torch.randn(self.hidden_1_size, self.hidden_2_size, requires_grad=True))
        self.b2 = nn.Parameter(torch.randn(1, self.hidden_2_size, requires_grad=True))
        self.relu2 = nn.ReLU()
        self.W3 = nn.Parameter(torch.randn(self.hidden_2_size, self.num_classes, requires_grad=True))
        self.b3 = nn.Parameter(torch.randn(1, self.num_classes, requires_grad=True))

    def forward(self, x):

        # first hidden layer
        a = x @ self.W1 + self.b1
        a = self.relu1(a)
        # second hidden layer
        b = a @ self.W2 + self.b2
        b = self.relu2(b)
        # output layer
        c = b @ self.W3 + self.b3

        return c # => logits
```

a

b

c

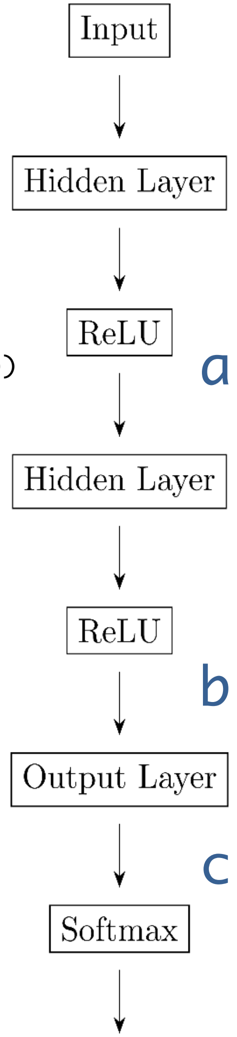
weights and biases are randomly initialised

wrapped by Parameter to be later picked up by the optimisation

data input

no softmax

@ is the shorthand for matrix multiplication



Building Models:
The **Easy/High-Level** Way

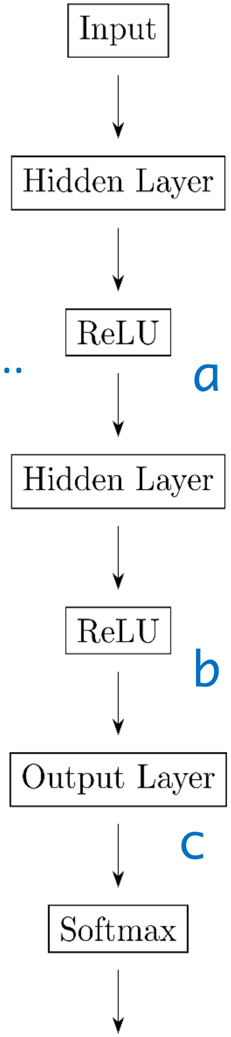
Most layers are already implemented in PyTorch‘s nn Module!

```
import torch.nn as nn

class NewsgroupsModel(nn.Module):
  """Simple Feedforward Neural Network for 20 Newsgroups"""
  def __init__(self, input_size=300):
    super().__init__()

    self.image_size = input_size
    self.hidden_1_size = 2048
    self.hidden_2_size = 256
    self.num_classes = 20

    self.fc1 = nn.Linear(self.image_size, self.hidden_1_size)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(self.hidden_1_size, self.hidden_2_size)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(self.hidden_2_size, self.num_classes)

  def forward(self, x):
    a = self.relu1(self.fc1(x))
    b = self.relu2(self.fc2(a))
    c = self.fc3(b) # => logits

    return c
```

takes care of weights, biases, initialisation, ...

data input

still no softmax



Short slide-in: Initialising Neural Networks

- The weights in a network need some initial values
- PyTorch layers handle this for us
- But what is the best option for initialisation?

- **Our low-level approach was:** Take a normal distribution around zero and pick some values…
- … was that a smart idea?

→ Maybe. It depends

Initialising Neural Networks

- Many possible initialisers
- Problem: You never know which works best in your context

→ Try different initialisers

→ Some common choices on the following slides

Initialising Neural Networks – Random

- The easiest initialiser: Just sample random values

- Two variants:
  - Pick numbers from a **uniform** distribution, usually close to zero

    ```
    w = np.random.uniform(-0.01,0.01)
    ```

  - Pick numbers from a **normal** distribution, usually around zero

    ```
    w = 0.01* np.random.randn()
    ```

Initialising Neural Networks – Calibrating the Variance

- Problem with purely random initialisation:
  More inputs
  → Higher (variance of the) output of the neuron
  → Possible problem with high gradients!

- Can be fixed by „normalising“ the initialisation
- For each weight w of a neuron N:
  `w = np.random.randn() / sqrt(n)`
- n is the number of inputs to the neuron N

Some more detail: http://cs231n.github.io/neural-networks-2/#init

Initialising Neural Networks – Glorot/Xavier

- Introduced by Xavier Glorot and Yoshua Bengio
- Names **Glorot-** and **Xavier-Initialiser** used interchangeably

- Complex analysis of gradient flow in networks
  → Recommend to normalise the variance to

$$Var(w) = \frac{2}{n_{in} + n_{out}}$$

Number of inputs/outputs of the neuron

```
w = np.random.randn() / sqrt(2/(n_in + n_out))
```

- Works best for layers with sigmoid or tanh activation functions

Glorot, Xavier ; Bengio, Yoshua ; Teh, Yee Whye (Bearb.) ; Titterington, D. Mike (Bearb.): Understanding the difficulty of training deep feedforward neural networks.. 9. In: *AISTATS* : JMLR.org, 2010 (JMLR Proceedings), S. 249-256

Initialising Neural Networks – He/Kaiming

- Introduced by Kaiming He et al.
- Names **He-** and **Kaiming-Initialiser** used interchangeably

$$Var(w) = \frac{gain}{n_{in}}$$

- $gain$ depends on activation function, e.g. for ReLU: $gain = 2$

```
w = np.random.randn() / sqrt(2/n_in)
```

- Was initially introduced for the ReLU activation function, thus works best for it

HE, Kaiming, et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: *Proceedings of the IEEE international conference on computer vision*. 2015. S. 1026-1034.

Initialising Neural Networks - PyTorch

- PyTorch contains many common initialisers
  - Random uniform/normal
  - Glorot/Xavier
  - He/Kaiming
  - …

- He/Kaiming initialisation is the default for most layers

→ Use He/Kaiming first. Try others, too!

Initialising Neural Networks - PyTorch

How to initialise weights in PyTorch?

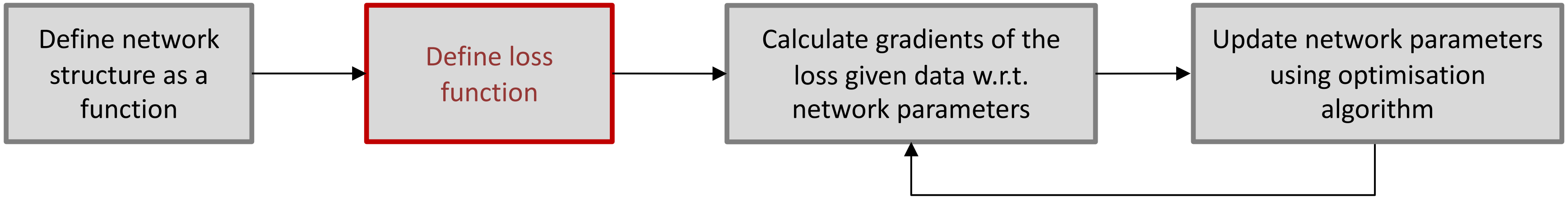
1. Initialise only one layer after creating it:

```
torch.nn.init.xavier_uniform(layer.weight)
```

2. Initialise the whole model after creating it

```
def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform(m.weight)

model.apply(init_weights)
```

Loss Function



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

Update network parameters using optimisation algorithm

PyTorch – Loss Function

- Given the output of the network, calculate the error of the prediction w.r.t. the correct label → Loss function / „criterion“

```python
import torch.nn as nn

# Loss function
criterion = nn.CrossEntropyLoss()

# calculate the loss
loss = criterion(logits, labels)
```

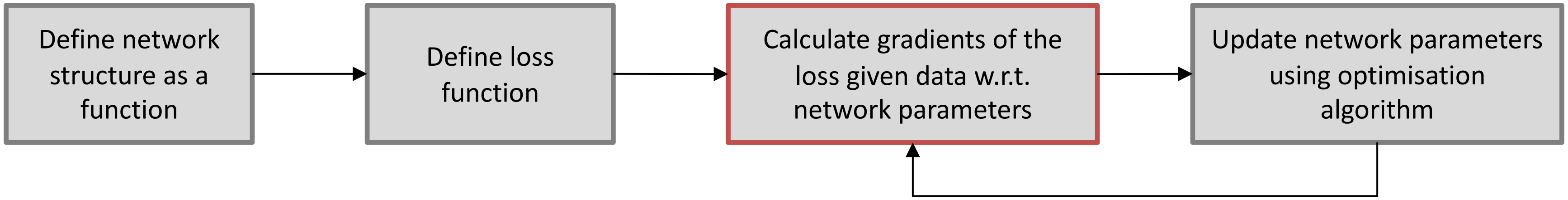
nn module provides common loss functions

applys softmax for us as gradient calculation is faster this way

Unnormalised output of the network

Correct label

Gradient Calculation



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

Update network parameters using optimisation algorithm

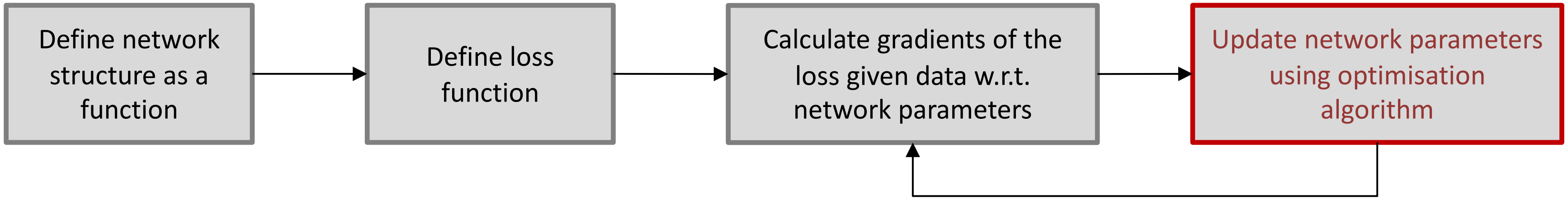
PyTorch – Gradient Calculation

- Given the value of the loss function and the current weights, calculate the gradient for all parameters

```
import torch.nn as nn

# Loss function
criterion = nn.CrossEntropyLoss()

# calculate the loss
loss = criterion(logits, labels)
[…]
# calculate the gradients w.r.t. the network parameters
loss.backward()
```

That was easy ☺

Optimiser



Define network structure as a function

Define loss function

Calculate gradients of the loss given data w.r.t. network parameters

Update network parameters using optimisation algorithm

PyTorch – Optimiser

- Alter the weights of the network to minimise the error
  → Optimiser

optim provides common optimisers

```
import torch.optim as optim
```

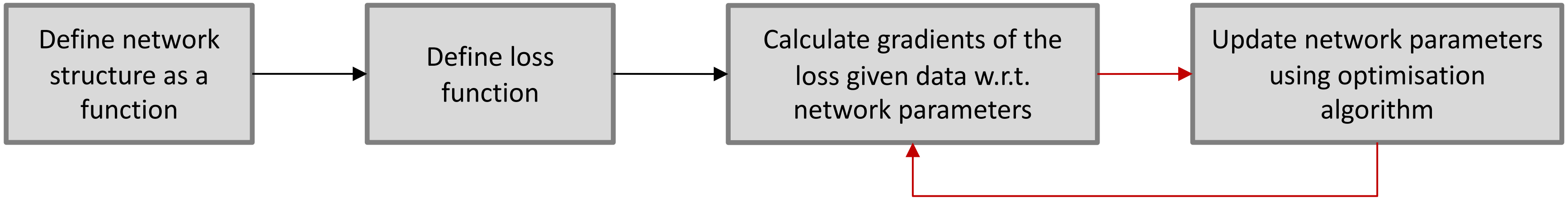
learning rate

```
# Optimiser
optimiser = optim.Adam(model.parameters(), lr=0.001)
```

all weights and biases of the NewsgroupsModel instance

That was also easy ☺

Putting things together: The Training Loop


Define network structure as a function
Define loss function
Calculate gradients of the loss given data w.r.t. network parameters
Update network parameters using optimisation algorithm

PyTorch — Training Loop

```
for epoch in range(10):  # loop over the dataset multiple times
    for data in data_loader:
        # get the data points
        inputs, labels = data


        # zero the parameter gradients
        # (else, they are accumulated)
        optimiser.zero_grad()


        # forward the data through the network
        logits = model(inputs)

        # calculate the loss
        loss = criterion(logits, labels)

        # calculate the gradients w.r.t. the network parameters
        loss.backward()

        # let the optimiser take an optimization step
        optimiser.step()
```

```
# Data
dataset = NewsgroupsDataset(data)
data_loader =
torch.utils.data.DataLoader(datas
et, batch_size=512, shuffle=True,
num_workers=2)

# Model
model = NewsgroupsModel()

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser
optimiser =
optim.Adam(model.parameters(),
lr=0.001)
```

PyTorch – A Model for 20 Newsgroups

We‘ve set up everything, it‘s time to let it run

→ Demo Time!

Combatting Overfitting – Regularisation

- Our network is adequate on the training set (78.96 % Accuracy)…
- … but worse on the test set (70.17 % Accuracy)!

- **Overfitting** is very common for neural networks
- Deep networks → Lots of parameters → Memorising the training set

- **Regularisation** is designed to prevent this
- Two main tactics:
  - Penalising high weights (L1/L2-norm)
  - Dropout

Combatting Overfitting – L2 Regularisation

- **Idea**:
  Force the network to use all inputs rather than focusing on some
- Modify the loss to penalise „peaky“ weights
- Add a regularisation term:

$$L_{reg} = L + \frac{1}{2}\lambda\sum_{w} w^2$$

Loss without regularisation

Regularisation „strength“

All weights in the network

- $\boldsymbol{L_2}$**-norm** is smaller if there are no „outliers“ in the weights

Combatting Overfitting – L2 Regularisation

- L2-Regularisation in PyTorch:

$$L_{reg} = L + \frac{1}{2}\lambda \sum_{w} w^2$$

- You could add the loss term by hand:

weight or bias tensor from the defined layers

```
for param in model.parameters():
    loss += 0.5 * lamb * torch.sum(param**2)
```

- But most optimisers already have a `weight_decay` parameter that is closely related and mostly equivalent to L2 regularisation:

```
optimiser = optim.Adam(model.parameters(), lr=0.001, weight_decay=lamb)
```

More information about the similarities and differences between L2 regularisation and weight decay can be found at https://bbabenko.github.io/weight-decay/

L2 Regularisation vs. Weight Decay

- L2 Regularisation adds a term to **the loss function**:

$$L_{reg} = L + \underbrace{\frac{1}{2}\lambda\sum_{w} w^2}_{\text{that's new}}$$

- Deriving the regularised loss function:

$$\frac{\partial}{\partial w} L_{reg} = \frac{\partial}{\partial w} L + \lambda w$$

vs

- Weight decay adds a term to **the derivative of the loss function**:

$$\frac{\partial}{\partial w} L + \underbrace{\lambda w}_{\text{that's new}}$$

- Then, in the optimisation step (SGD):

$$w = w \ - lr \cdot (\frac{\partial}{\partial w} L + \lambda w)$$

looks like they are equivalent!

L2 Regularisation vs. Weight Decay

Question: But are they? When do they obtain different results?

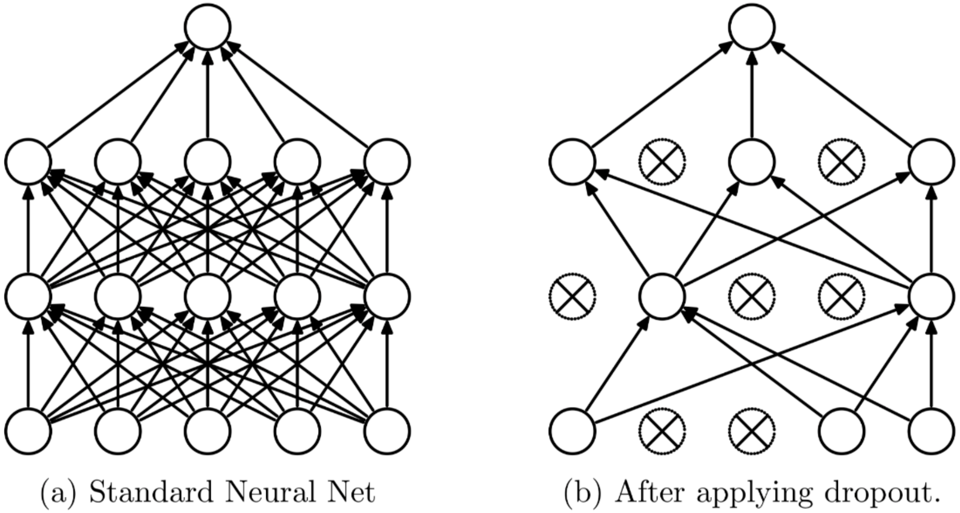
Answer:

- Adding a term to the optimisation step (**Weight Decay**)
  vs.
  modifying the loss gradient (**L2**)
- Equivalent for optimisers that do not reuse previous gradients (e.g. SGD)
- Not equivalent for optimisers that reuse previous loss gradients (e.g. Adam or SGD with Momentum)
  - L2 affects the previous gradient
  - The Weight Decay term is not reused

Combatting Overfitting – Dropout

- Dropout: More recent method (2014)

- **Idea**: During **training**, „remove" a portion $0 \leq p < 1$ of neurons from the net
  → Force the net to rely on many features instead of few!



Srivastava, Nitish ; Hinton, Geoffrey E ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: a simple way to prevent neural networks from overfitting.. In: *Journal of machine learning research*, 15 (2014), Nr. 1, S. 1929--1958

Combatting Overfitting – Dropout

- Implementation by a **mask**:
  Random vector with zeros (probability **p**) and ones

Fully Connected layer with ReLU

```
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) > p
```

→ Mask with zeros and ones

```
H1 *= U1
```

→ Keep the neuron if the position in the mask has a one, set it to zero otherwise
→ Equivalent to removing the neuron

Combatting Overfitting – Dropout

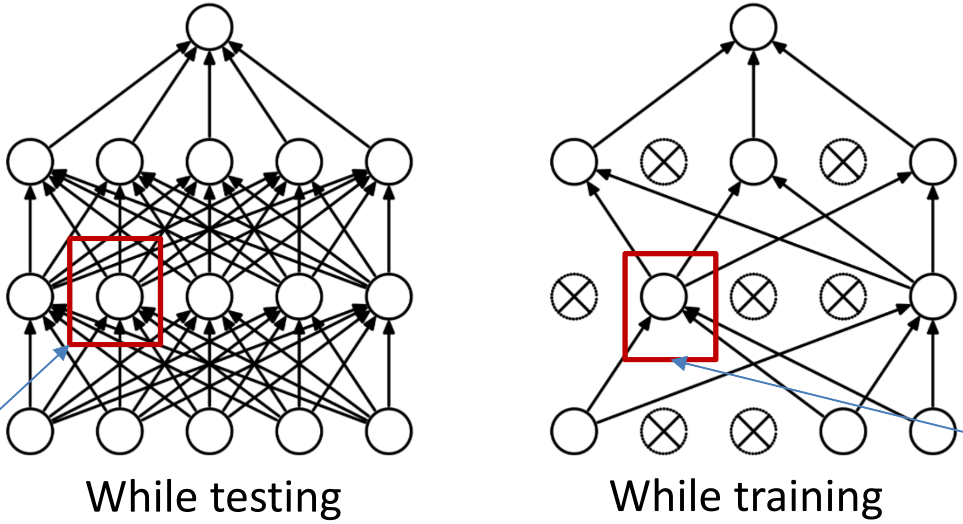
- Dropout only applied in **training** phase!
- When **testing**, dropout is **disabled** ($p = 0$)

→ Roughly equivalent to training an **ensemble** of smaller networks

- Empirically shown to improve generalisation on many tasks

Combatting Overfitting – Dropout

Problem: Dropout changes the expected output of a neuron!



Sum over **5** inputs

Sum over **3** inputs

→ Larger expected output when testing!
→ Scaling needed!

Combatting Overfitting – Dropout

- Scaling needed because of different expected output during training/testing
- Two possible solutions:
  - Scale **down** while **testing**

```
H1 = np.maximum(0, np.dot(W1, X) + b1) * p
```

  - Scale **up** while **training**

```
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) > p) / p
H1 *= U1
```

Combatting Overfitting – Dropout

- Scaling needed because of different expected output during training/testing
- Two possible solutions:
  - Scale **down** while **testing**

```
H1 = np.maxim                              1) * p
```

This slows down the prediction step! We **don‘t** want that!

  - Scale **up** while **training**

```
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) > p) / p
H1 *= U1
```

Combatting Overfitting – Dropout

- PyTorch provides a Dropout module

```
# Model __init__
self.dropout = nn.Dropout(p=0.4)

# Model forward
x = self.dropout(x)
```

to drop values with a **probability** of 40% during training.

Important:

```
model.train() # default
model.eval()
```

sets a variable in the model to indicate training model,
so dropout should be applied

sets the model training variable to False, so dropout is disabled;
call this before testing!

Attention!
Libraries use **keep probability** and **drop probability** inconsistently!

PyTorch – A Model for 20 Newsgroups

Dropout Demo

PyTorch – A Model for 20 Newsgroups

→ Better results on test set (71.35 % vs 70.17 % without Dropout)!

This may not sound like much, but:
Additional **>2%** often means a huge gain!
All the more so if we consider the small effort involved.

- Curious effect of Dropout:
  - Results on the training set sometimes worse than on the test set
  - Caused by aforementioned „ensemble“ of smaller networks:
    All neurons available for testing, only some for training
- Nice to have:
  - Dropout usually makes training faster due to more multiplications with zeros

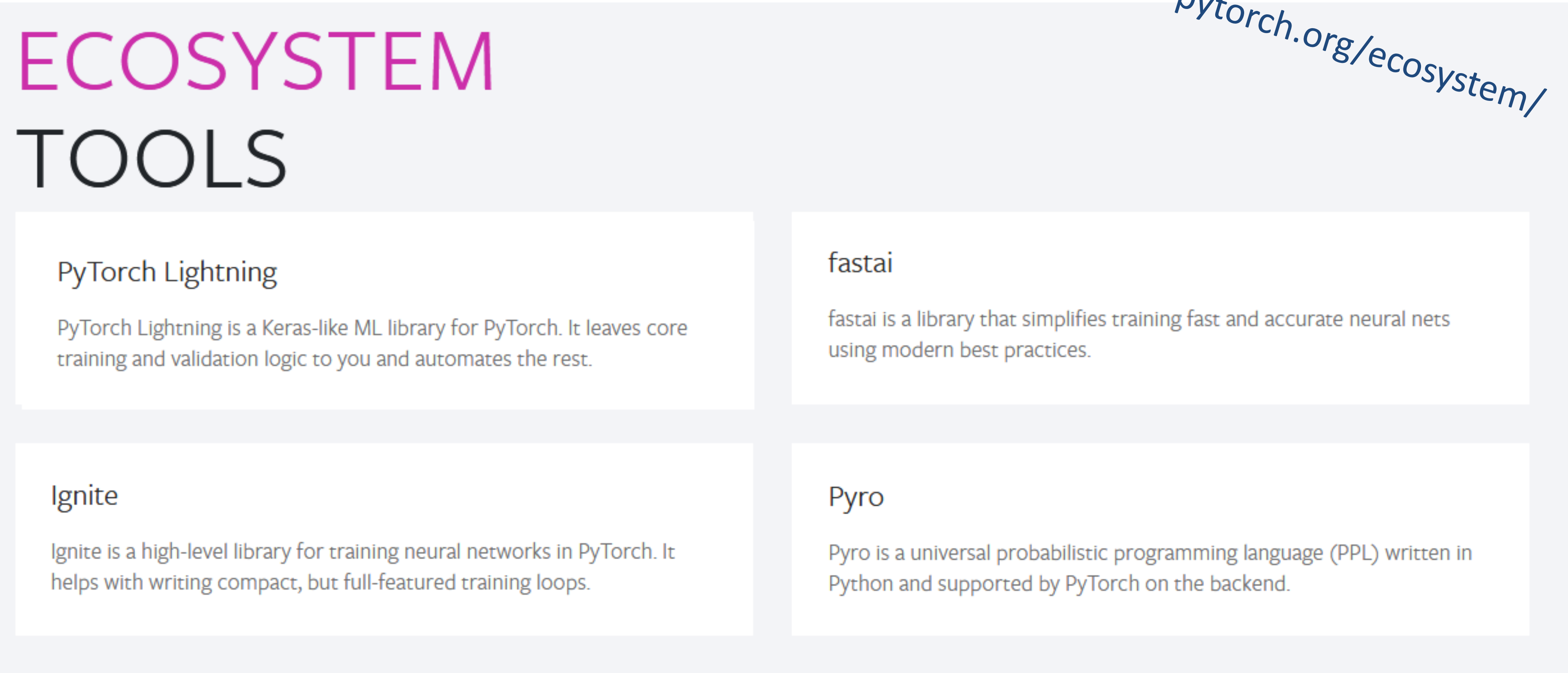
PyTorch – Bringing everything to the GPU

- Currently, everything was computed on the CPU
- In PyTorch, we have to move all values to the GPU if we want to work with it
- Only a few changes are necessary to run on the GPU

```
# Automatically choose GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


# Move the data tensors (inputs and outputs) to the correct device
input_tensor.to(device)
output_tensor.to(device)


# Move the model parameters to the correct device
model.to(device)
```

Making PyTorch development easier/faster

pytorch.org/ecosystem/



ECOSYSTEM

TOOLS

PyTorch Lightning

PyTorch Lightning is a Keras-like ML library for PyTorch. It leaves core training and validation logic to you and automates the rest.

fastai

fastai is a library that simplifies training fast and accurate neural nets using modern best practices.

Ignite

Ignite is a high-level library for training neural networks in PyTorch. It helps with writing compact, but full-featured training loops.

Pyro

Pyro is a universal probabilistic programming language (PPL) written in Python and supported by PyTorch on the backend.

… and many more

3.3 Vectorisation of Neural Networks

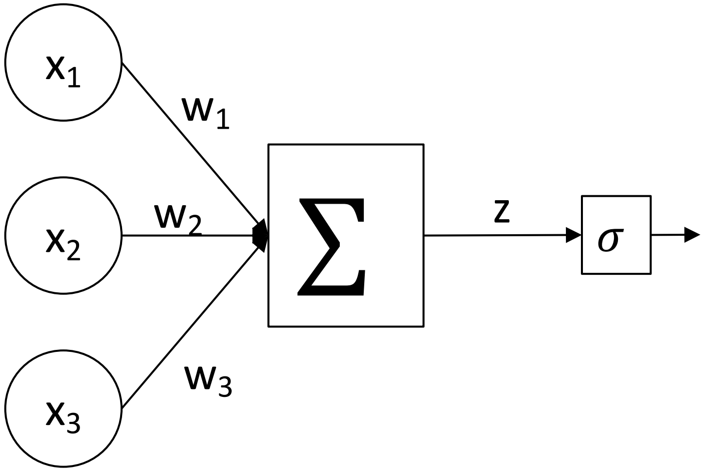
Why Vectorisation?

- **Recall:**
  - Matrix multiplications can be computed very efficiently
  - Element-wise operations can be parallelised (good for GPUs)
- **Idea:** Make your network faster to compute by using as many matrix multiplications and element-wise operations as possible

→ Vectorise the forward pass and backpropagation algorithm

Vectorising the Forward Pass

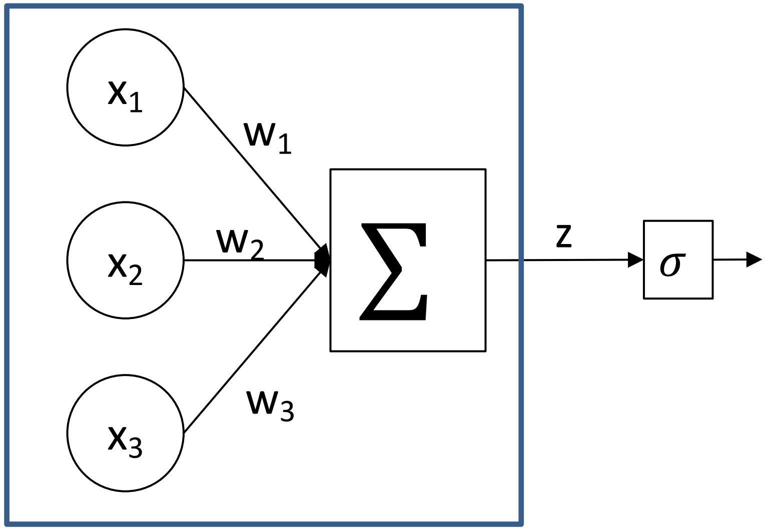
Vectorising the Forward Pass



You already know this

- Two steps:
  - $z = \sum_{i=1}^{3} w_i x_i$
  - $y = \sigma(z)$

Vectorising the Forward Pass



You already know this

Instead of calculating

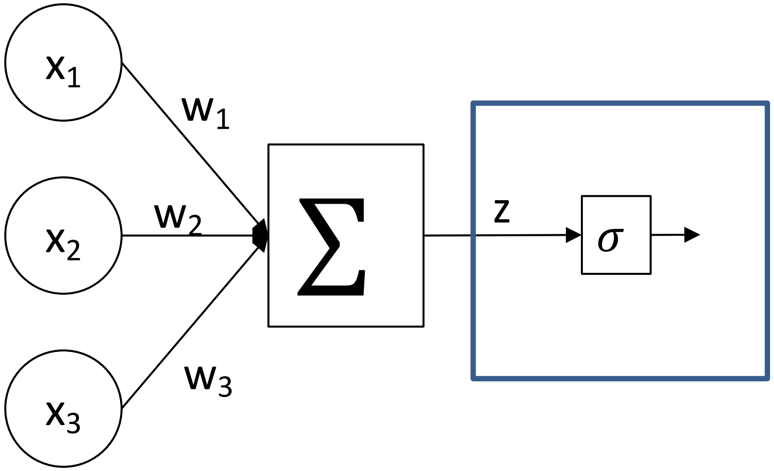
$$z = \sum_{i=1}^{3} w_i x_i$$

using a for-loop, we can compute

$$z = w^T x$$

using vector/matrix multiplication.

Vectorising the Forward Pass



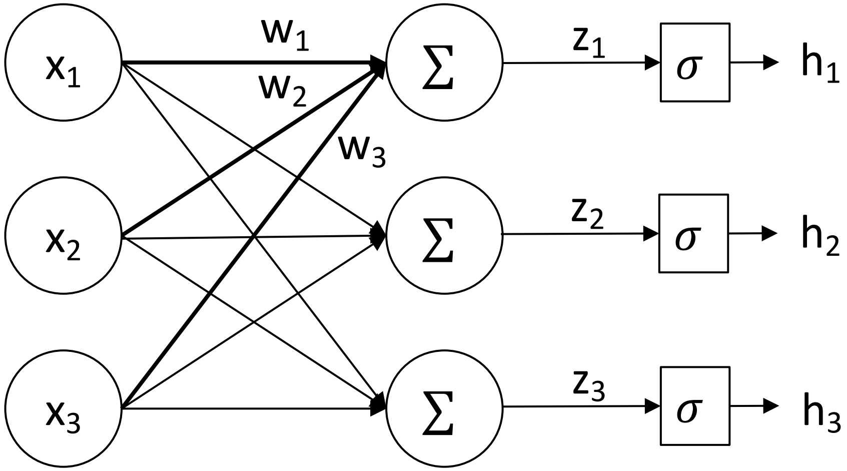
You already know this

- $\sigma(z)$ is just a function working on one scalar

→ Forward pass is easily parallelisable for one neuron ☺

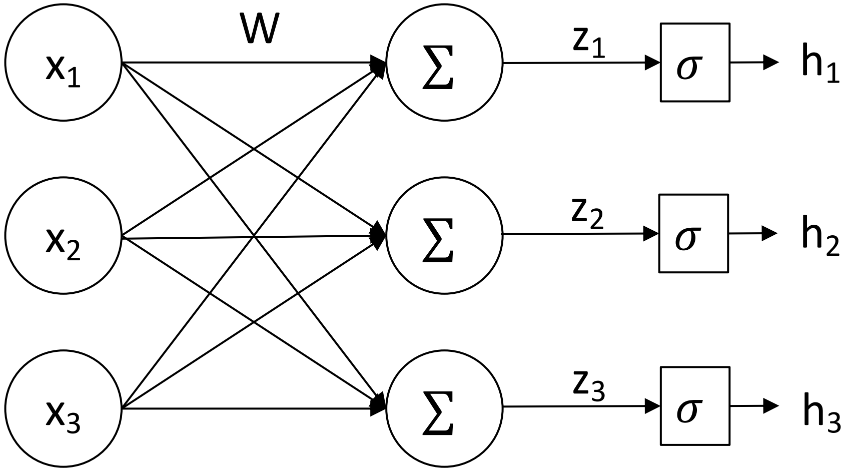
→ But wait! Usually, we have more than one neuron per layer!

Vectorising the Forward Pass



- Fortunately, all neurons are the same, except the weights
- Combine all weight vectors w to matrix W:
  $W_{i,j}$ is the weight for the connection from neuron x_j to h_i

Vectorising the Forward Pass



Again two steps:

this is now a vector

$$z = Wx$$

this is now a matrix

$$h = \sigma(z)$$

element-wise operation (parallelisable!)

this is now a vector

very similar to before, but fully vectorised! ☺

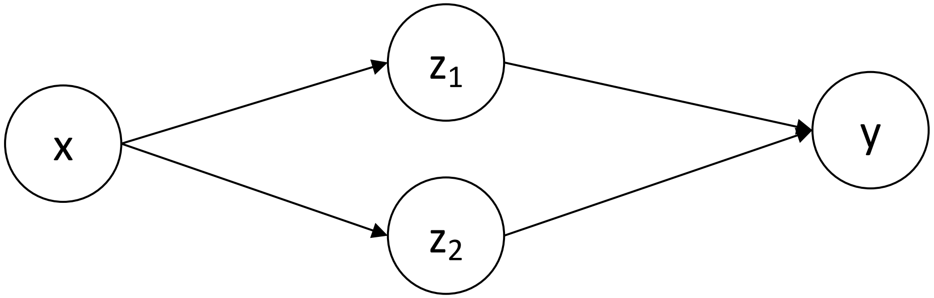
Vectorising Backpropagation

Backpropagation

- Recall: Backpropagation =
  - calculating **partial derivatives**…
  - … of the network‘s **loss function**…
  - … w.r.t. the **weights and biases**…
  - … by applying the **chain rule**:

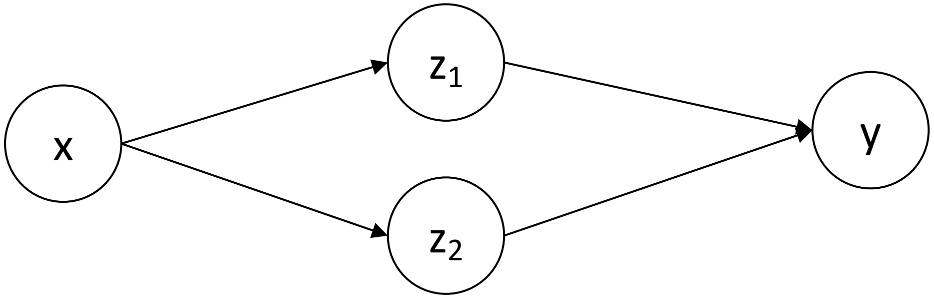
$$\text{If } f(x) = p\big(q(x)\big)\text{, then } \frac{df}{dx} = \frac{df}{dq}\,\frac{dq}{dx}$$

Multivariable Chain Rule



- In neural networks, it is typical for a value to „flow“ through multiple paths to a neuron (here: x → y)
- When calculating gradients w.r.t. x, we can go two paths:
  - via z_1: $\frac{dy}{dz_1}\frac{dz_1}{dx}$
  - via z_2: $\frac{dy}{dz_2}\frac{dz_2}{dx}$
- How do we calculate the final gradient $\frac{dy}{dx}$?

Multivariable Chain Rule



- The multivariable chain rule states:

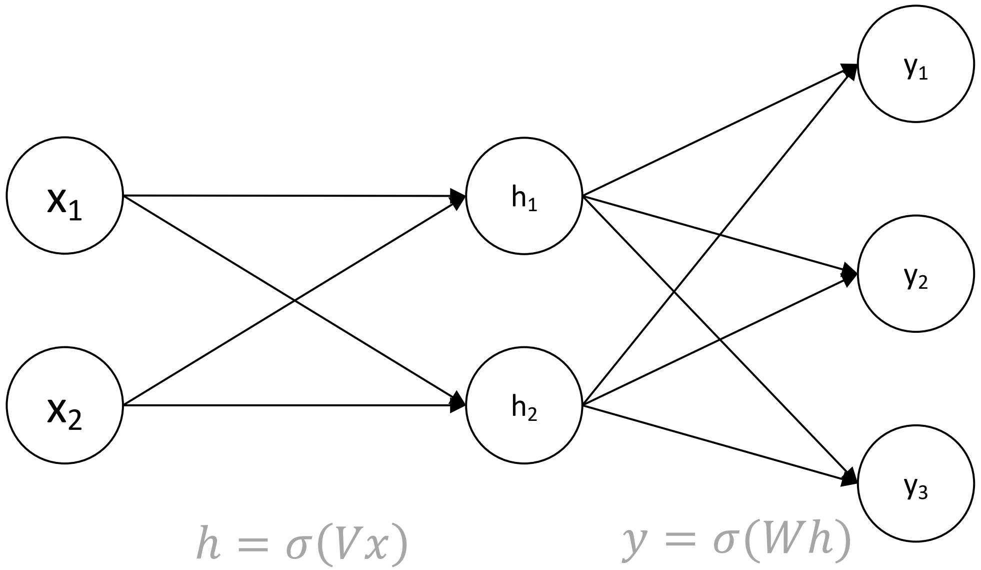
If $f\big(g(x), q(x)\big)$, then $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx} + \frac{df}{dq}\frac{dq}{dx}$

- In this case:

$$\frac{dy}{dx} = \underbrace{\frac{dy}{dz_1}\frac{dz_1}{dx}}_{\text{path via } z_1} + \underbrace{\frac{dy}{dz_2}\frac{dz_2}{dx}}_{\text{path via } z_2}$$

Okay, let‘s vectorise the backpropagation in an example neural network!

Example Neural Network



$$L(y, t) = \frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

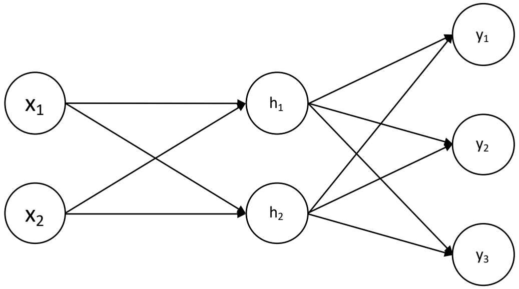
output vector

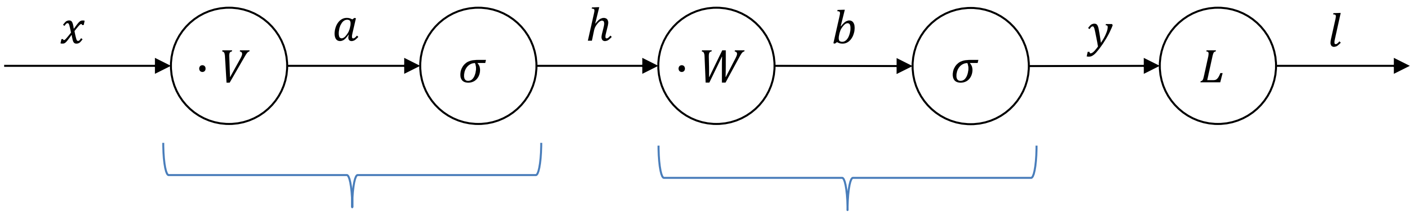
target vector

- Simple Feedforward Network with one hidden layer
- No bias, but possible to add using the bias trick (see exercises)
- Sigmoid activation function
- Loss function: squared error loss

First, let‘s visualise this network as a circuit diagram with vectors

Example Neural Network





x to hidden layer

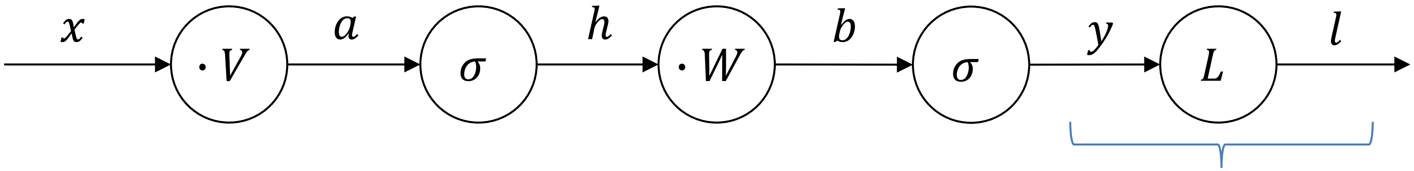
$h = \sigma(Vx)$

hidden layer to output

$y = \sigma(Wh)$

Backpropagation: Use the chain rule and go back every step in the diagram

Backpropagation Step 1



what is $\frac{\partial l}{\partial y}$?

$$L(y,t) = \frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

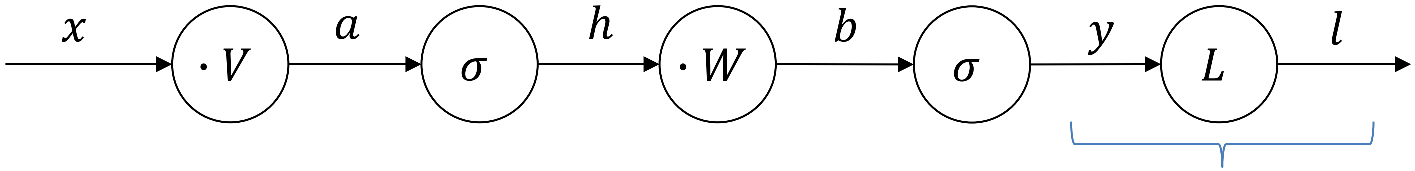
$$\frac{\partial l}{\partial y_1} = \frac{\partial}{\partial y_1}\frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

$$\frac{\partial l}{\partial y} = \begin{bmatrix} \frac{\partial l}{\partial y_1} \\ \frac{\partial l}{\partial y_2} \\ \frac{\partial l}{\partial y_3} \end{bmatrix}$$

$$\frac{\partial l}{\partial y_2} = \frac{\partial}{\partial y_2}\frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

$$\frac{\partial l}{\partial y_3} = \frac{\partial}{\partial y_3}\frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

Backpropagation Step 1



what is $\frac{\partial l}{\partial y}$?

$$\frac{\partial l}{\partial y_1} = \frac{\partial}{\partial y_1}\frac{1}{2}\sum_{i=1}^{3}(t_i - y_i)^2$$

$$= \frac{\partial}{\partial y_1}\left(\frac{1}{2}(t_1 - y_1)^2 + \frac{1}{2}(t_2 - y_2)^2 + \frac{1}{2}(t_3 - y_3)^2\right)$$

sum rule

$$= \frac{\partial}{\partial y_1}\frac{1}{2}(t_1 - y_1)^2 + \cancel{\frac{\partial}{\partial y_1}\frac{1}{2}(t_2 - y_2)^2} + \cancel{\frac{\partial}{\partial y_1}\frac{1}{2}(t_3 - y_3)^2}$$

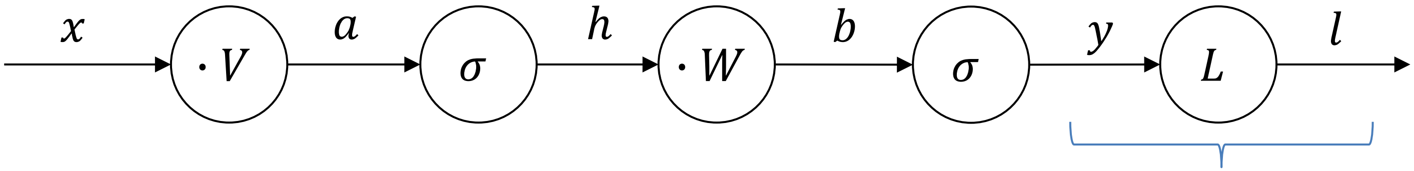
$$= \frac{\partial}{\partial y_1}\boxed{\frac{1}{2}(\boxed{t_1 - y_1})^2}$$

chain rule: $p(q(y_1))$ with

$$\boxed{q(y_1, t_1) = t_1 - y_1}$$

$$\boxed{\text{and } p(q) = \frac{1}{2}(q)^2}$$

Backpropagation Step 1



what is $\frac{\partial l}{\partial y}$?

chain rule: $p(q(y_1, t_1))$ with

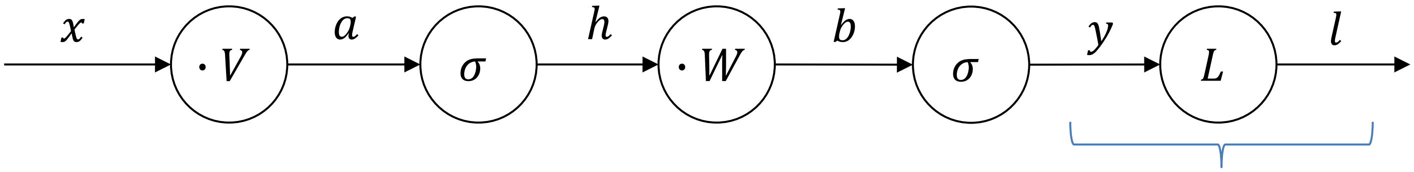
$q(y_1, t_1) = t_1 - y_1$

and $p(q) = \frac{1}{2}(q)^2$

Previous slide

$$\frac{\partial l}{\partial y_1} = \frac{\partial}{\partial y_1}\frac{1}{2}(t_1 - y_1)^2 = \underbrace{2\frac{1}{2}(t_1 - y_1)}_{\frac{dp}{dq}} \cdot \underbrace{-1}_{\frac{dq}{dy_1}} = y_1 - t_1$$

Backpropagation Step 1



what is $\frac{\partial l}{\partial y}$?

$$\frac{\partial l}{\partial y_1} = y_1 - t_1$$

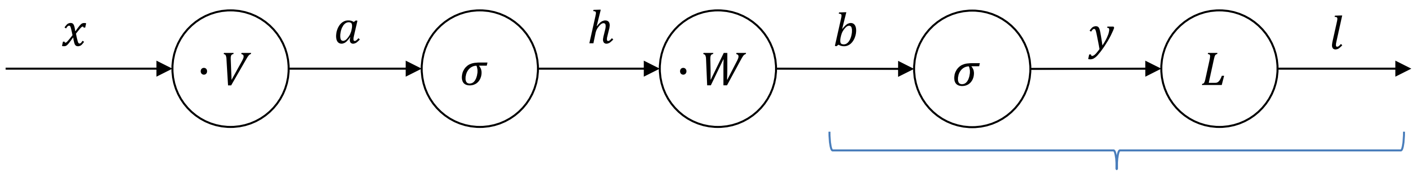
$$\frac{\partial l}{\partial y_2} = y_2 - t_2$$

$$\frac{\partial l}{\partial y_3} = y_3 - t_3$$

$$\frac{\partial l}{\partial y} = \begin{bmatrix} \frac{\partial l}{\partial y_1} \\ \frac{\partial l}{\partial y_2} \\ \frac{\partial l}{\partial y_3} \end{bmatrix} \boxed{= y - t}$$

fast vector operation ☺

Backpropagation Step 2



what is $\frac{\partial l}{\partial b}$?

σ is an element-wise operation, so we only need to look at the i^{th} element

chain rule!

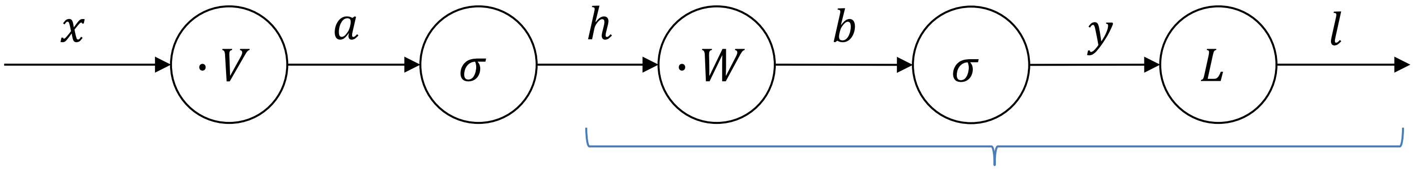
$$\frac{\partial l}{\partial b} = \begin{bmatrix} \frac{\partial l}{\partial b_1} \\ \frac{\partial l}{\partial b_2} \\ \frac{\partial l}{\partial b_3} \end{bmatrix}$$

$$\frac{\partial l}{\partial b_1} = \frac{\partial l}{\partial y_1} \cdot \frac{\partial y_1}{\partial b_1} = (y_1 - t_1) \cdot \sigma(b_1) \cdot (1 - \sigma(b_1))$$

$$\frac{\partial l}{\partial b_2} = \frac{\partial l}{\partial y_2} \cdot \frac{\partial y_2}{\partial b_2} = (y_2 - t_2) \cdot \sigma(b_2) \cdot (1 - \sigma(b_2))$$

$$\frac{\partial l}{\partial b_3} = \frac{\partial l}{\partial y_3} \cdot \frac{\partial y_3}{\partial b_3} = (y_3 - t_3) \cdot \sigma(b_3) \cdot (1 - \sigma(b_3))$$

Backpropagation Step 3



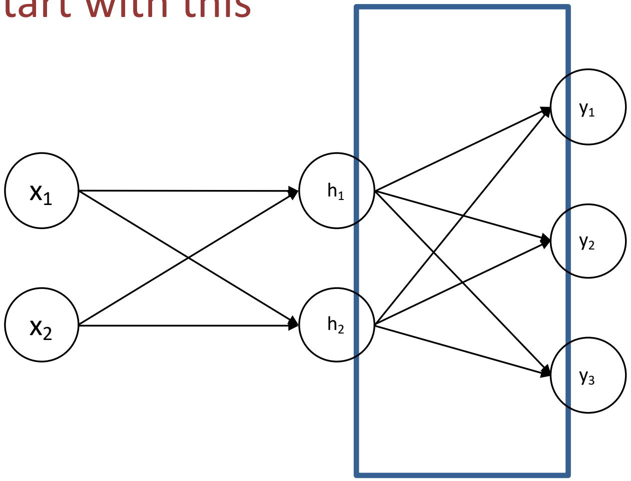
what are $\frac{\partial l}{\partial h}$ and $\frac{\partial l}{\partial W}$?

$$\frac{\partial l}{\partial h} = \begin{bmatrix} \frac{\partial l}{\partial h_1} \\ \frac{\partial l}{\partial h_2} \end{bmatrix}$$

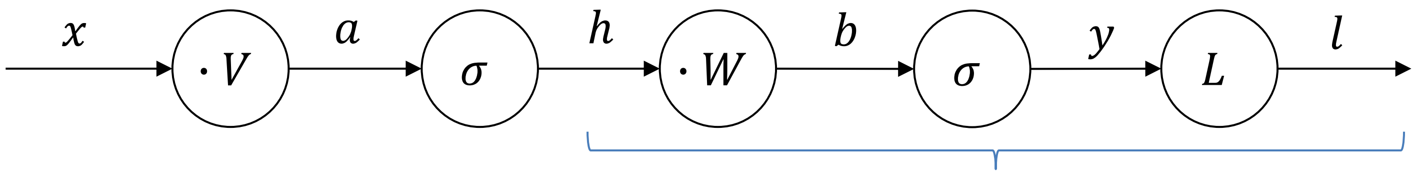
We need this for calculating the derivatives of the previous layer

let‘s start with this

$$\frac{\partial l}{\partial W} = \begin{bmatrix} \frac{\partial l}{\partial W_{1,1}} & \frac{\partial l}{\partial W_{1,2}} \\ \frac{\partial l}{\partial W_{2,1}} & \frac{\partial l}{\partial W_{2,2}} \\ \frac{\partial l}{\partial W_{3,1}} & \frac{\partial l}{\partial W_{3,2}} \end{bmatrix}$$



Backpropagation Step 3 — h



what is $\frac{\partial l}{\partial h}$?

$$\frac{\partial l}{\partial h} = \begin{bmatrix} \frac{\partial l}{\partial h_1} \\ \frac{\partial l}{\partial h_2} \end{bmatrix}$$

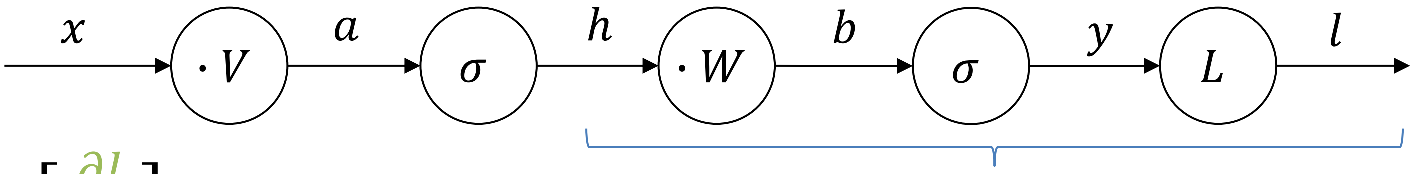
$$b = Wh = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} W_{1,1}h_1 + W_{1,2}h_2 \\ W_{2,1}h_1 + W_{2,2}h_2 \\ W_{3,1}h_1 + W_{3,2}h_2 \end{bmatrix}$$

multivariable chain rule!

$$\frac{\partial l}{\partial h_1} = \frac{\partial l}{\partial b_1} \cdot \frac{\partial b_1}{\partial h_1} + \frac{\partial l}{\partial b_2} \cdot \frac{\partial b_2}{\partial h_1} + \frac{\partial l}{\partial b_3} \cdot \frac{\partial b_3}{\partial h_1} = \frac{\partial l}{\partial b_1} \cdot W_{1,1} + \frac{\partial l}{\partial b_2} \cdot W_{2,1} + \frac{\partial l}{\partial b_3} \cdot W_{3,1}$$

$$\frac{\partial l}{\partial h_2} = \frac{\partial l}{\partial b_1} \cdot \frac{\partial b_1}{\partial h_2} + \frac{\partial l}{\partial b_2} \cdot \frac{\partial b_2}{\partial h_2} + \frac{\partial l}{\partial b_3} \cdot \frac{\partial b_3}{\partial h_2} = \frac{\partial l}{\partial b_1} \cdot W_{1,2} + \frac{\partial l}{\partial b_2} \cdot W_{2,2} + \frac{\partial l}{\partial b_3} \cdot W_{3,2}$$

Backpropagation Step 3 — h



what is $\frac{\partial l}{\partial h}$?

$$\frac{\partial l}{\partial b} = \begin{bmatrix} \frac{\partial l}{\partial b_1} \\ \frac{\partial l}{\partial b_2} \\ \frac{\partial l}{\partial b_3} \end{bmatrix}$$

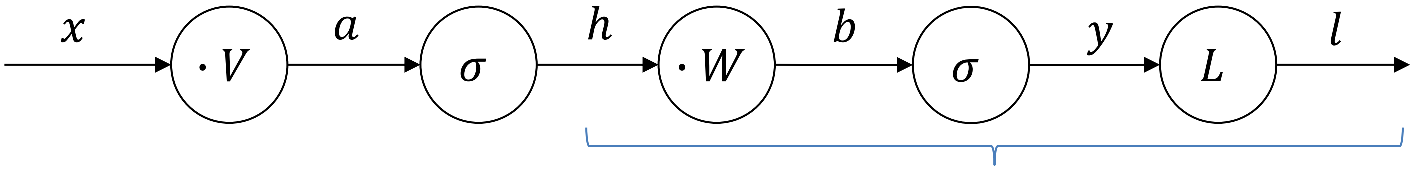
From previous slide:

$$\frac{\partial l}{\partial h} = \begin{bmatrix} \frac{\partial l}{\partial h_1} \\ \frac{\partial l}{\partial h_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial b_1} \cdot W_{1,1} + \frac{\partial l}{\partial b_2} \cdot W_{2,1} + \frac{\partial l}{\partial b_3} \cdot W_{3,1} \\ \frac{\partial l}{\partial b_1} \cdot W_{1,2} + \frac{\partial l}{\partial b_2} \cdot W_{2,2} + \frac{\partial l}{\partial b_3} \cdot W_{3,2} \end{bmatrix} \boxed{= W^T \frac{\partial l}{\partial b}}$$

vectorised version ☺

Backpropagation Step 3

You already know this



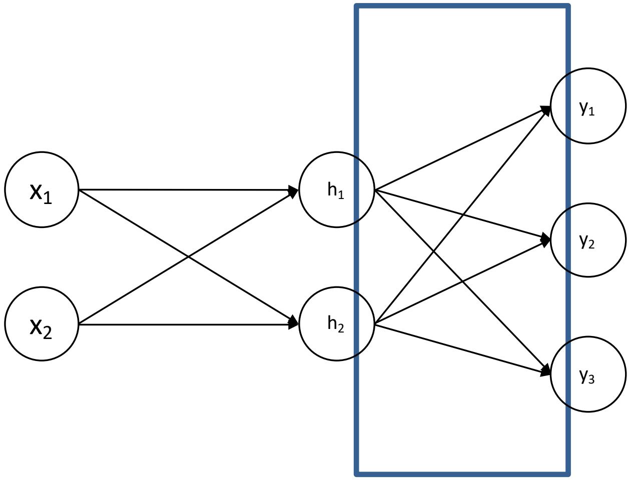
what are $\frac{\partial l}{\partial h}$ and $\frac{\partial l}{\partial W}$?

$$\frac{\partial l}{\partial h} = \begin{bmatrix} \frac{\partial l}{\partial h_1} \\ \frac{\partial l}{\partial h_2} \end{bmatrix}$$

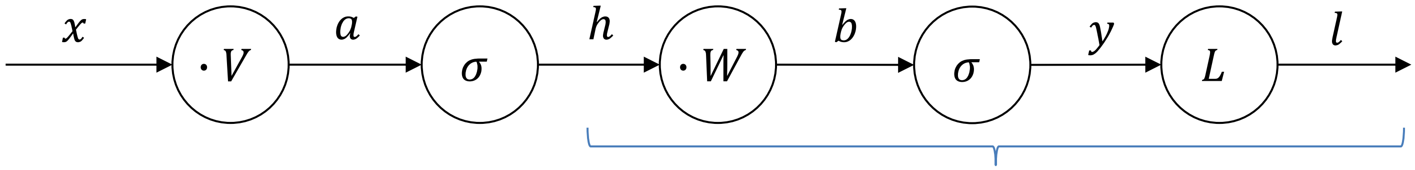
We need this for calculating the derivatives of the previous layer

$$\boxed{\frac{\partial l}{\partial W} = \begin{bmatrix} \frac{\partial l}{\partial W_{1,1}} & \frac{\partial l}{\partial W_{1,2}} \\ \frac{\partial l}{\partial W_{2,1}} & \frac{\partial l}{\partial W_{2,2}} \\ \frac{\partial l}{\partial W_{3,1}} & \frac{\partial l}{\partial W_{3,2}} \end{bmatrix}}$$

now this one



Backpropagation Step 3 — W



what is $\frac{\partial l}{\partial W}$?

$$\frac{\partial l}{\partial W} = \begin{bmatrix} \frac{\partial l}{\partial W_{1,1}} & \frac{\partial l}{\partial W_{1,2}} \\ \frac{\partial l}{\partial W_{2,1}} & \frac{\partial l}{\partial W_{2,2}} \\ \frac{\partial l}{\partial W_{3,1}} & \frac{\partial l}{\partial W_{3,2}} \end{bmatrix}$$

$$b = Wh = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} W_{1,1}h_1 + W_{1,2}h_2 \\ W_{2,1}h_1 + W_{2,2}h_2 \\ W_{3,1}h_1 + W_{3,2}h_2 \end{bmatrix}$$

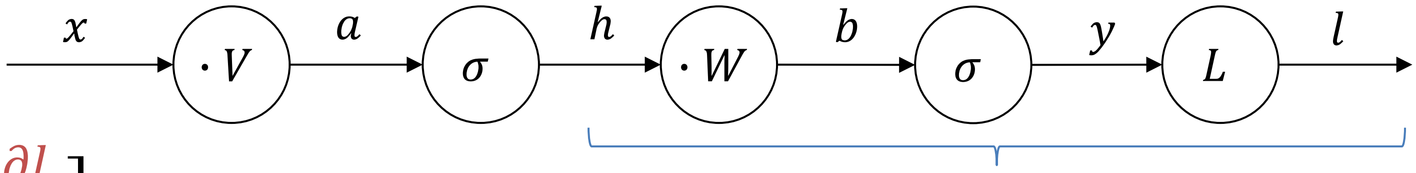
multivariable chain rule!

$$\frac{\partial l}{\partial W_{1,1}} \overset{\downarrow}{=} \frac{\partial l}{\partial b_1} \cdot \frac{\partial b_1}{\partial W_{1,1}} + \frac{\partial l}{\partial b_2} \cdot \frac{\partial b_2}{\partial W_{1,1}} + \frac{\partial l}{\partial b_3} \cdot \frac{\partial b_3}{\partial W_{1,1}} = \frac{\partial l}{\partial b_1} \cdot h_1 + \frac{\partial l}{\partial b_2} \cdot 0 + \frac{\partial l}{\partial b_3} \cdot 0$$

$$\vdots$$

$$\frac{\partial l}{\partial W_{3,2}} = \frac{\partial l}{\partial b_1} \cdot \frac{\partial b_1}{\partial W_{3,2}} + \frac{\partial l}{\partial b_2} \cdot \frac{\partial b_2}{\partial W_{3,2}} + \frac{\partial l}{\partial b_3} \cdot \frac{\partial b_3}{\partial W_{3,2}} = \frac{\partial l}{\partial b_1} \cdot 0 + \frac{\partial l}{\partial b_2} \cdot 0 + \frac{\partial l}{\partial b_3} \cdot h_2$$

Backpropagation Step 3 — W



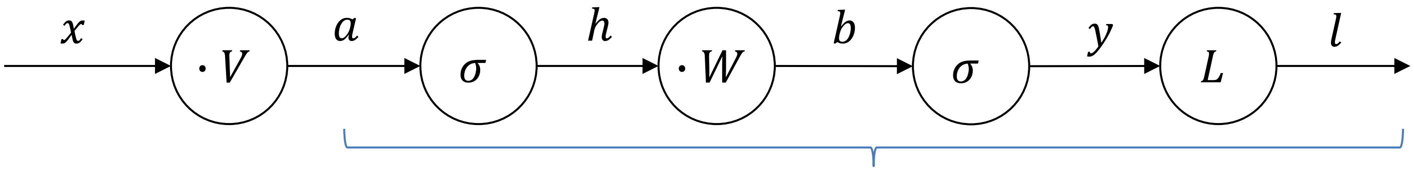
what is $\frac{\partial l}{\partial W}$?

$$\frac{\partial l}{\partial b} = \begin{bmatrix} \frac{\partial l}{\partial b_1} \\ \frac{\partial l}{\partial b_2} \\ \frac{\partial l}{\partial b_3} \end{bmatrix}$$

$$\frac{\partial l}{\partial W} = \begin{bmatrix} \frac{\partial l}{\partial W_{1,1}} & \frac{\partial l}{\partial W_{1,2}} \\ \frac{\partial l}{\partial W_{2,1}} & \frac{\partial l}{\partial W_{2,2}} \\ \frac{\partial l}{\partial W_{3,1}} & \frac{\partial l}{\partial W_{3,2}} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial b_1} \cdot h_1 & \frac{\partial l}{\partial b_1} \cdot h_2 \\ \frac{\partial l}{\partial b_2} \cdot h_1 & \frac{\partial l}{\partial b_2} \cdot h_2 \\ \frac{\partial l}{\partial b_3} \cdot h_1 & \frac{\partial l}{\partial b_3} \cdot h_2 \end{bmatrix} = \boxed{\frac{\partial l}{\partial b} h^T}$$

vectorised version ☺

Backpropagation Step 4



what is $\frac{\partial l}{\partial a}$?

σ is an element-wise operation, so we only need to look at the ith element

chain rule!

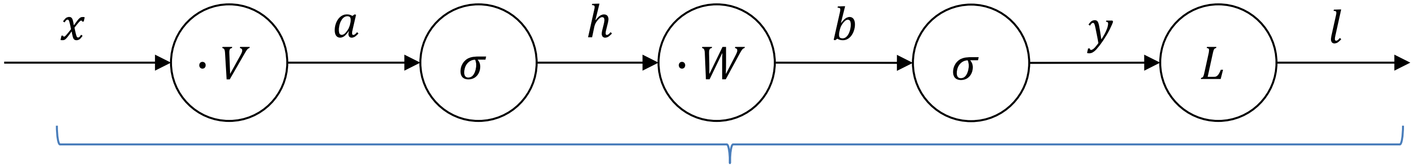
$$\frac{\partial l}{\partial a} = \begin{bmatrix} \frac{\partial l}{\partial a_1} \\ \frac{\partial l}{\partial a_2} \end{bmatrix}$$

$$\frac{\partial l}{\partial a_1} = \frac{\partial l}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} = \frac{\partial l}{\partial h_1} \cdot \sigma(a_1) \cdot (1 - \sigma(a_1))$$

$$\frac{\partial l}{\partial a_2} = \frac{\partial l}{\partial h_2} \cdot \frac{\partial h_2}{\partial a_2} = \frac{\partial l}{\partial h_2} \cdot \sigma(a_2) \cdot (1 - \sigma(a_2))$$

same structure as in Step 2, when we calculated $\frac{\partial l}{\partial W}$

Backpropagation Step 5

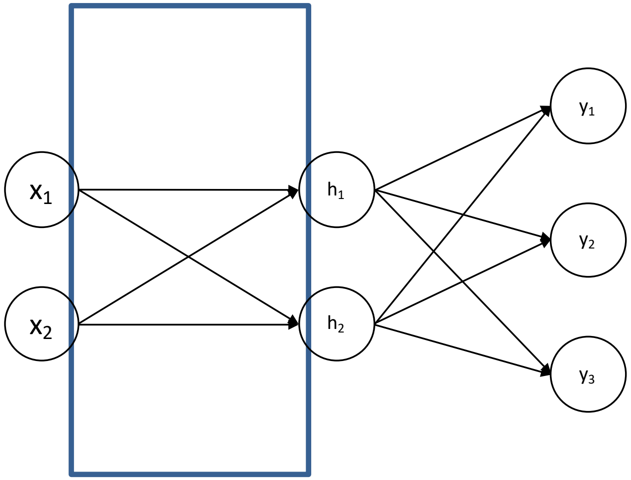


what are $\frac{\partial l}{\partial x}$ and $\frac{\partial l}{\partial V}$?

$$\frac{\partial l}{\partial x} = \begin{bmatrix} \frac{\partial l}{\partial x_1} \\ \frac{\partial l}{\partial x_2} \end{bmatrix}$$

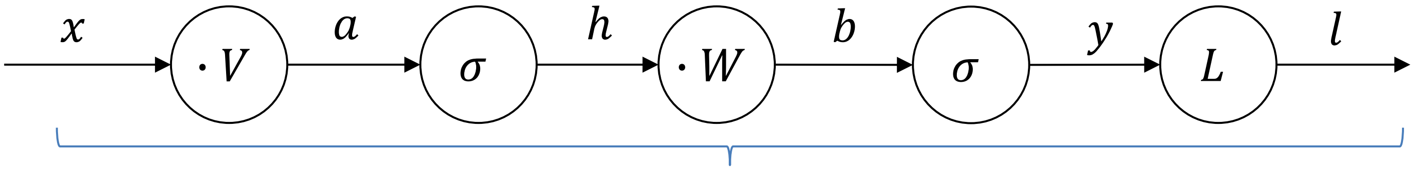
We don‘t need these for weight optimisation (but we could)
→ Don‘t calculate them

$$\frac{\partial l}{\partial V} = \begin{bmatrix} \frac{\partial l}{\partial V_{1,1}} & \frac{\partial l}{\partial V_{1,2}} \\ \frac{\partial l}{\partial V_{2,1}} & \frac{\partial l}{\partial V_{2,2}} \end{bmatrix}$$



same structure as in Step 3, when we calculated $\frac{\partial l}{\partial W}$

Backpropagation Step 5 — V



what is $\frac{\partial l}{\partial V}$?

Step 3 result:

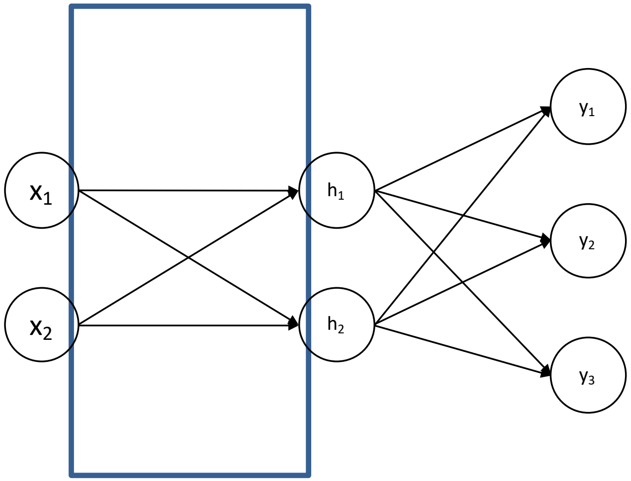
$$\frac{\partial l}{\partial W} = \frac{\partial l}{\partial b} h^T \qquad \text{with } b = Wh$$

Same calculation works here:

$$\mathrm{a} = Vx$$

$$\frac{\partial l}{\partial V} = \begin{bmatrix} \frac{\partial l}{\partial V_{1,1}} & \frac{\partial l}{\partial V_{1,2}} \\ \frac{\partial l}{\partial V_{2,1}} & \frac{\partial l}{\partial V_{2,2}} \end{bmatrix} = \boxed{\frac{\partial l}{\partial a} x^T}$$

vectorised calculation ☺



Summary Vectorising Neural Networks

- How can we speed up neural network calculations?
  1. Backpropagation algorithm: reuse calculated derivatives
  2. Vectorisation: use highly optimised vector and matrix multiplications

  made computations feasible in 1970s

  3. Faster hardware: GPUs, parallelize computations, …

  made computing big models possible (since 2009)

- Trends since then:
  - Faster hardware
  - More efficient models than feedforward networks (e.g. CNN, that reuses weights)
  - Optimisation algorithms that converge faster (less training iterations → faster training)

Next Week:

- Convolutional Neural Networks!