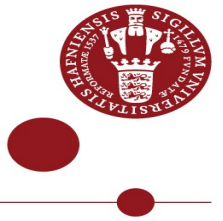
Deep Learning

Today: deep neural networks, convolutional nets

The Rectified Linear Unit Activation Function



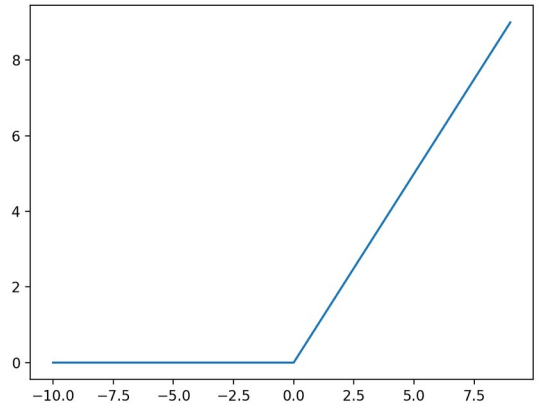
The Rectified Linear Unit Activation Function

What complex idea could this name possibly encompass?

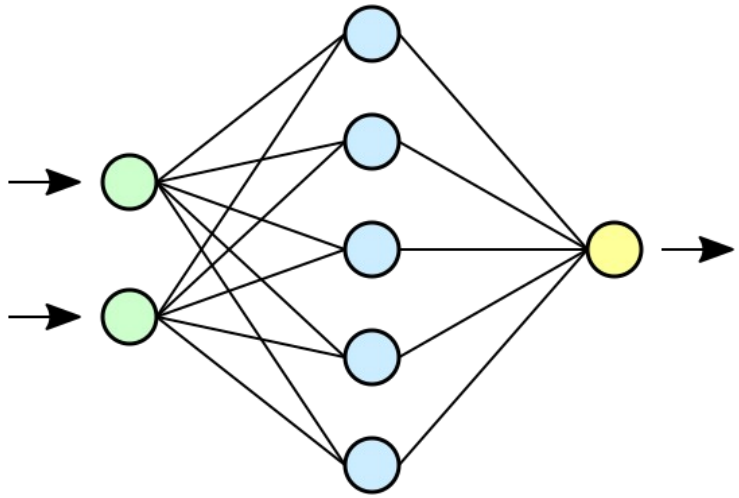
The Rectified Linear Unit Activation Function

What complex idea could this name possibly encompass?

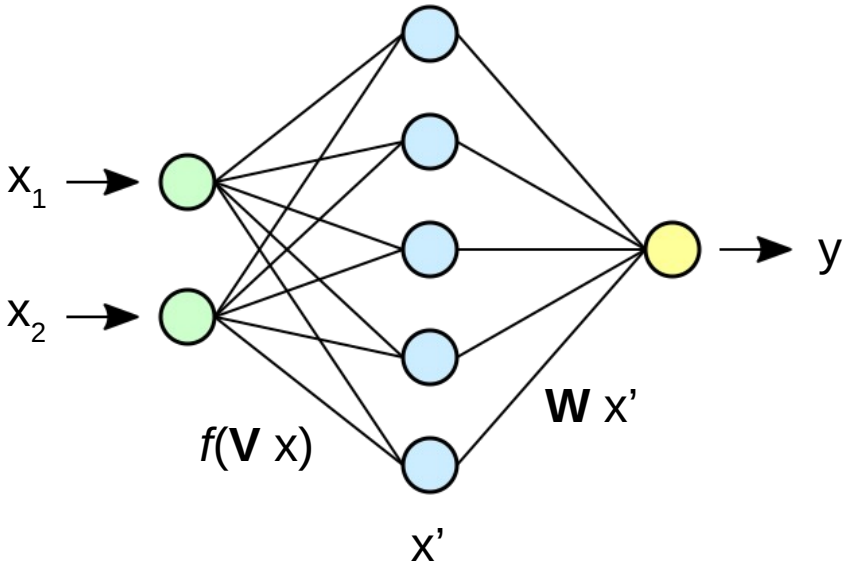
$$f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$



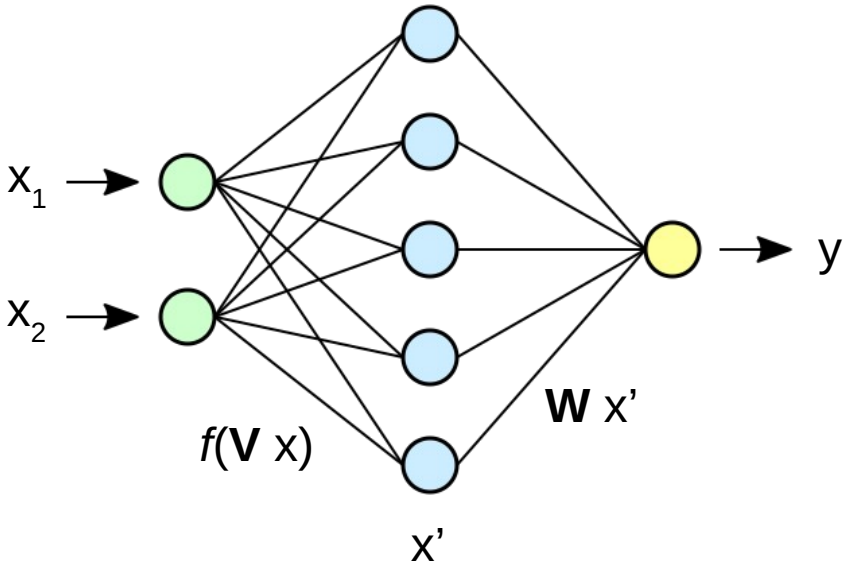
What are neural networks?



What are neural networks?



What are neural networks?

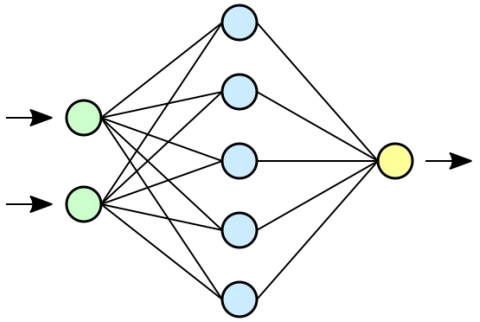


$$x' = [f(V_{11}\ x_1 + V_{12}\ x_2),\ f(V_{21}\ x_1 + V_{22}\ x_2),\ f(V_{21}\ x_1 + V_{22}\ x_2),\ .....]$$

$$y = W_{11}\ x'_1 + W_{12}\ x'_2 + W_{13}\ x'_3 + ...$$

where e.g. $f(x) = \tanh(x)$.

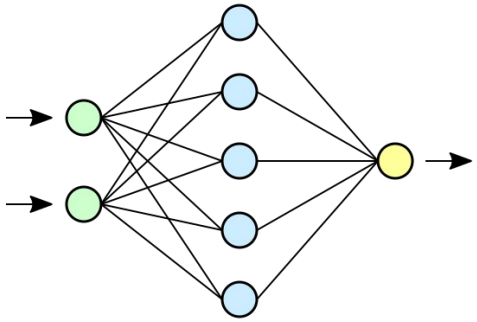
What are neural networks?



- A simple neural network

- It’s clear why it’s called a ***neural*** network

What are neural networks?



- A simple neural network

- It's clear why it's called a ***neural*** network

Slightly more complex:

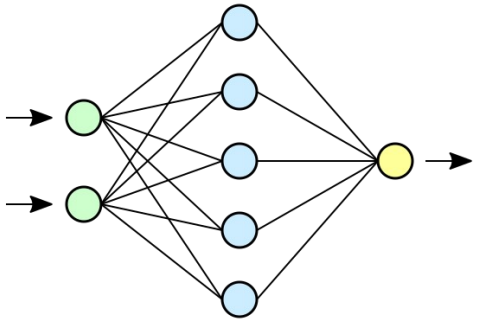
$$f(x;p) = \mathbf{a}_1\, g(\mathbf{a}_2\, g(\mathbf{a}_3 x + \mathbf{b}_3) + \mathbf{b}_2)) + \mathbf{b}_1$$

Some matrix

Some non-linear function
"Activation Function"

Some vector

What are neural networks?



- A simple neural network

- It's clear why it's called a ***neural*** network

Slightly more complex:

$$f(x;p) = \mathbf{a}_1\, g(\mathbf{a}_2\, g(\mathbf{a}_3 x + \mathbf{b}_3) + \mathbf{b}_2)) + \mathbf{b}_1$$

Some matrix

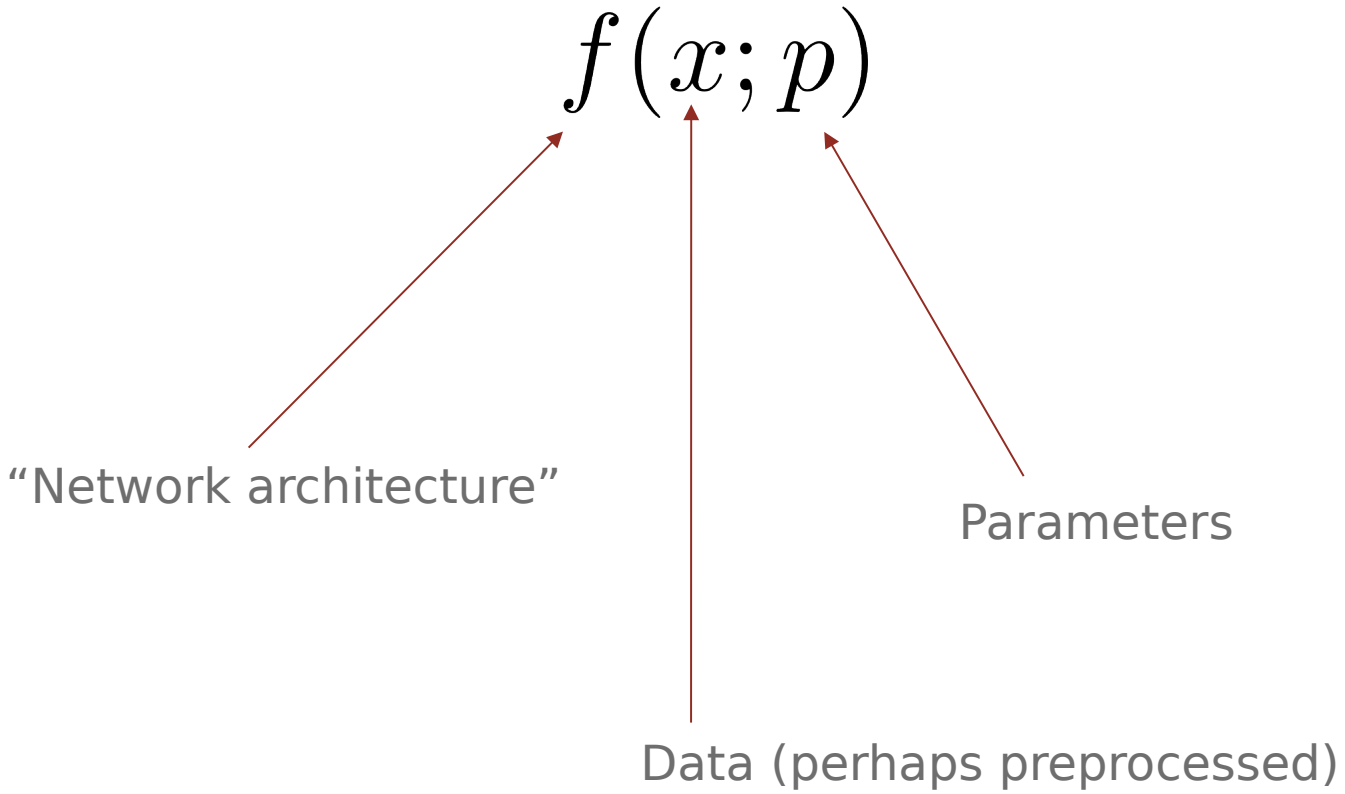
Some non-linear function
"Activation Function"

Some vector

Can be made much more complex...

Neural networks

In fact, the term “neural network” nowadays can be used about almost any function:



Neural networks

In fact, the term “neural network” nowadays can be used about almost any function:

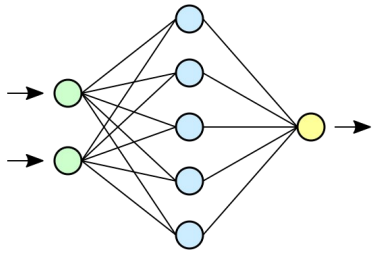
$$f(x; p)$$

“Network architecture”

Parameters

Data (perhaps preprocessed)

This:



is a special-case called a *feed-forward* neural network

Neural networks

Neural networks is just (“glorified”) **function fitting**!

Neural networks

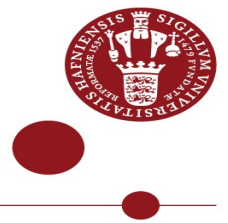
Neural networks is just (“glorified”) **function fitting**!

- The main difference is that functions chosen for machine learning application (“neural networks”) have **many** parameters that can be tuned (“trained”)

Neural networks

If neural network is just function fitting, what do I need to learn?

- Architectures that work well (designing the function) ← The core of these lectures
- How to fit/train them
- How to train them fast, with lots of data
- How to validate that you've actually "learned" what you think
- How do understand the "reasoning" behind predictions

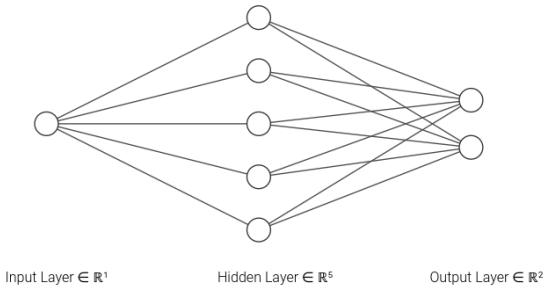


What is deep learning?

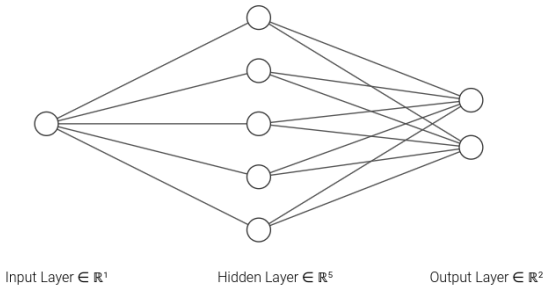
Deep learning is simply an expression used to indicate that the architecture of our networks are "*deep".*

This is an *architecture* choice that happens to often be a good idea.

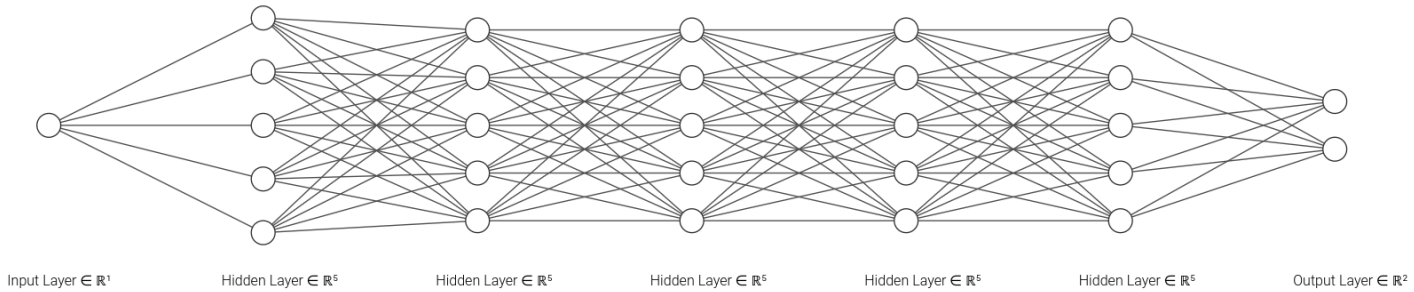
What is deep learning?



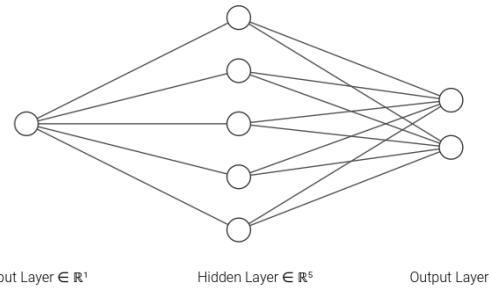
What is deep learning?



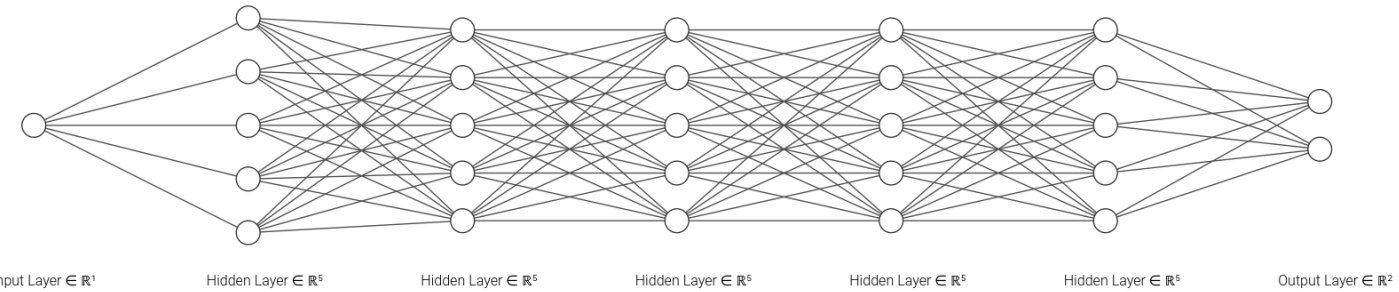
Deep network



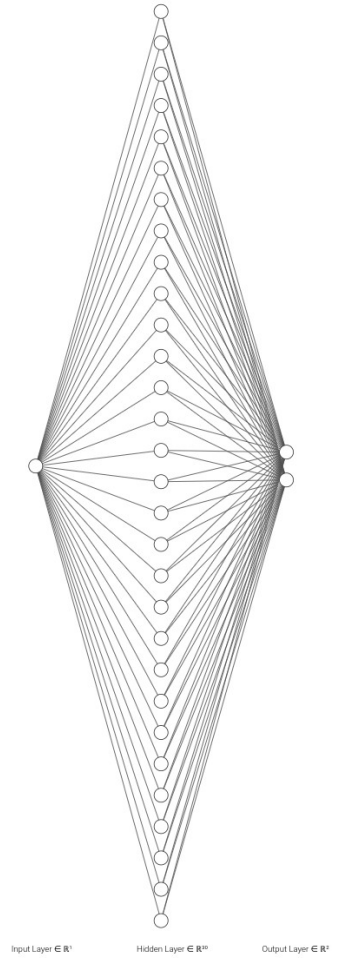
What is deep learning?



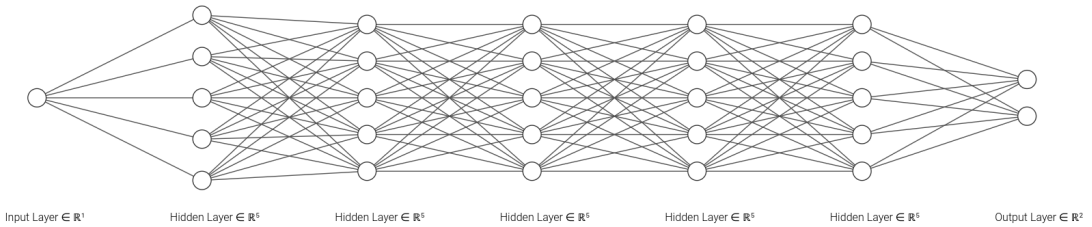
Deep network



Broad network



How to do deep learning



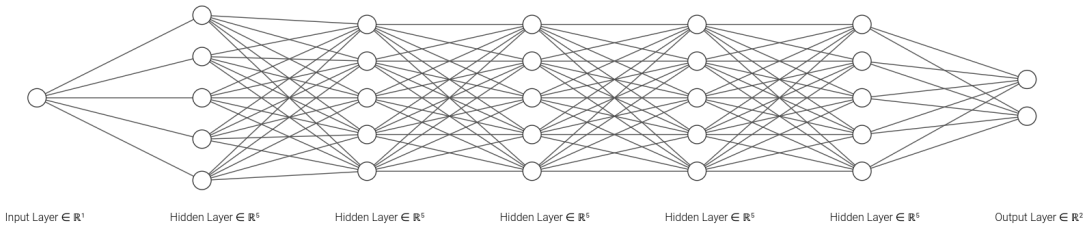
NumPy

“I can do this implement such a function in numpy!”

“… and I can use scipy to fit the function to data!”

“… so can do deep learning!”

How to do deep learning



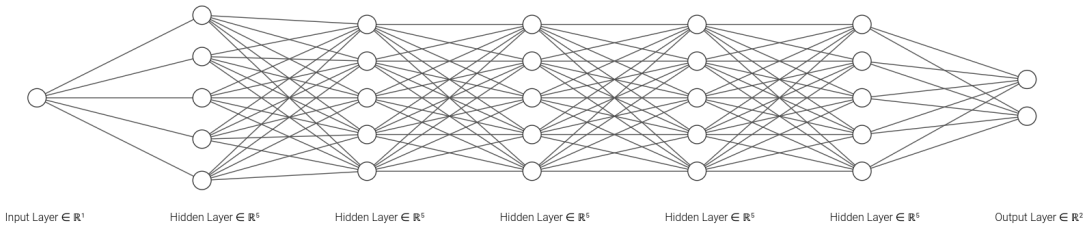
NumPy

"I can do this implement such a function in numpy!"

"… and I can use scipy to fit the function to data!"

"… so can do deep learning!"

How to do deep learning



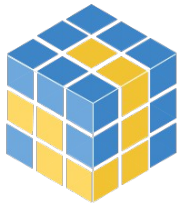
NumPy

"I can do this implement such a function in numpy!"

"… and I can use scipy to fit the function to data!"

"… so can do deep learning!"

But there are easier ways! (and tons of tricks!)



NumPy

PYTORCH

TensorFlow

Keras

+ Haiku

Ease of use for NNs

(your opinion may differ!)

NumPy

PYTORCH

+ Haiku

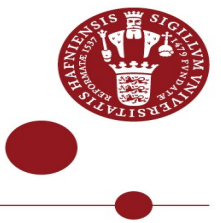
Ease of doing non-standard stuff

(your opinion may differ!)

PyTorch

- GPU acceleration
- Automatic Error-Backpropagation (chain rule through operations)
- Tons of functionality built-in

Deep Learning with PyTorch

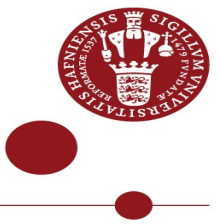


Deep Learning with PyTorch

What does PyTorch (or other frameworks) do for you?)

Framework feature #1: Automatic gradient calculations

Minimize $Loss(f(x, p), y)$

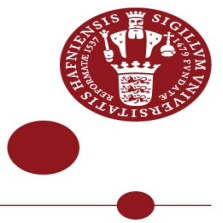


Framework feature #1: Automatic gradient calculations

Minimize $Loss(f(x,p),y)$

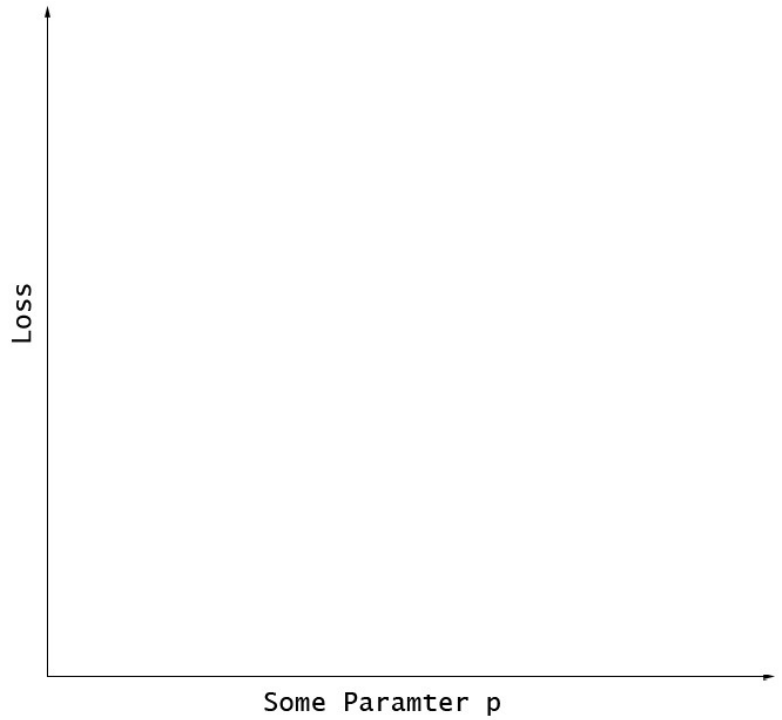
Only feasible way to minimize this if there are many parameters, is if can calculate

$$\nabla_p Loss(f(x,p),y)$$



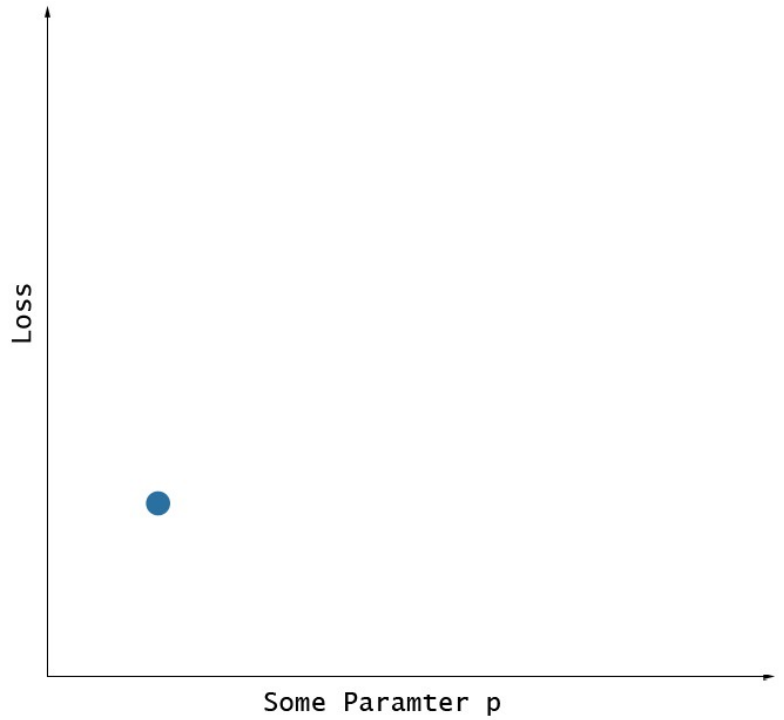
Training a Network

Minimize $Loss(f(x, p), y)$



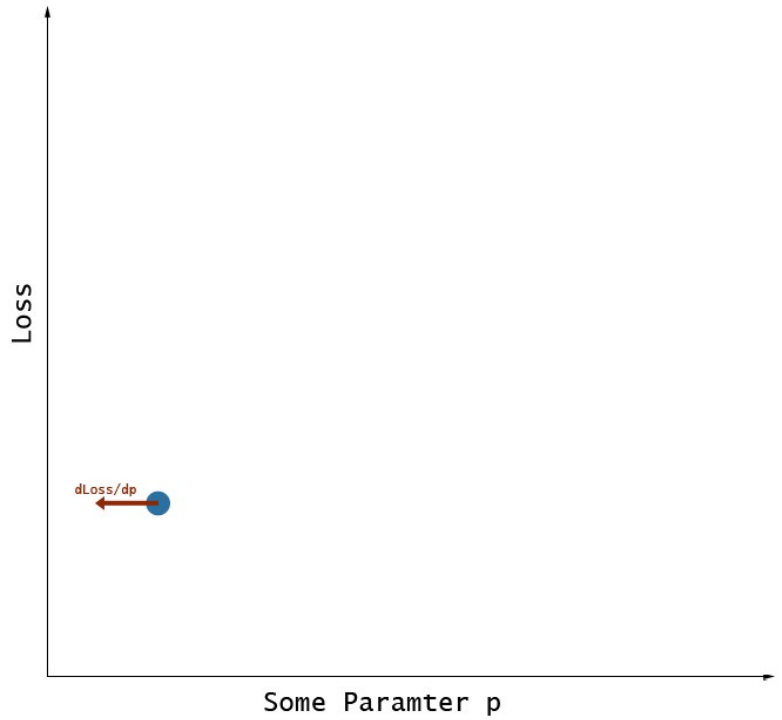
Training a Network

Minimize $Loss(f(x, p), y)$



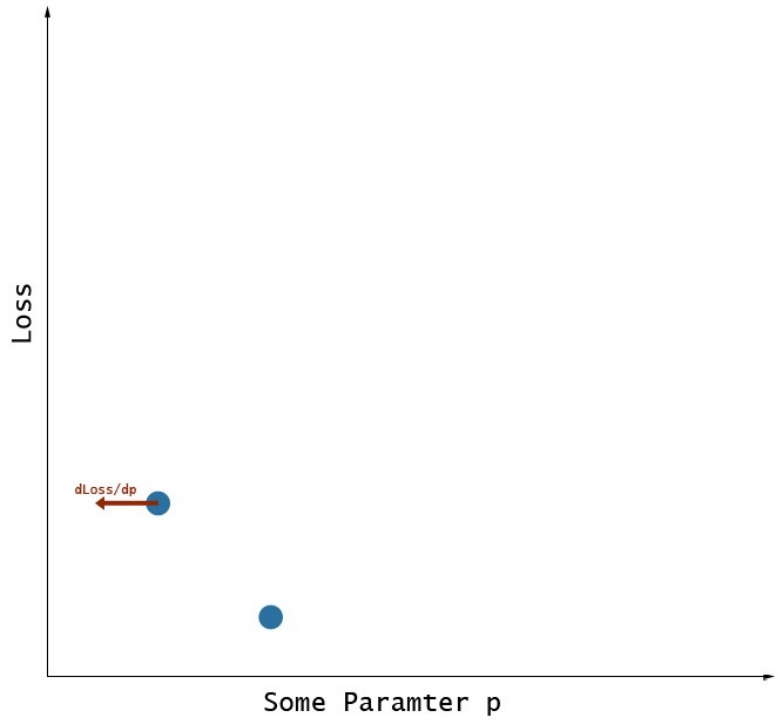
Training a Network

Minimize $Loss(f(x, p), y)$



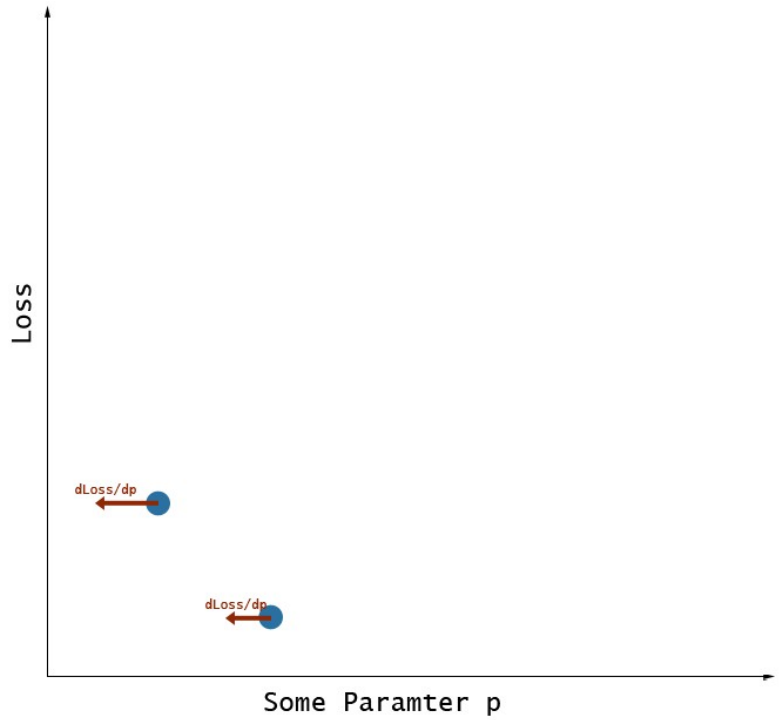
Training a Network

Minimize $Loss(f(x, p), y)$



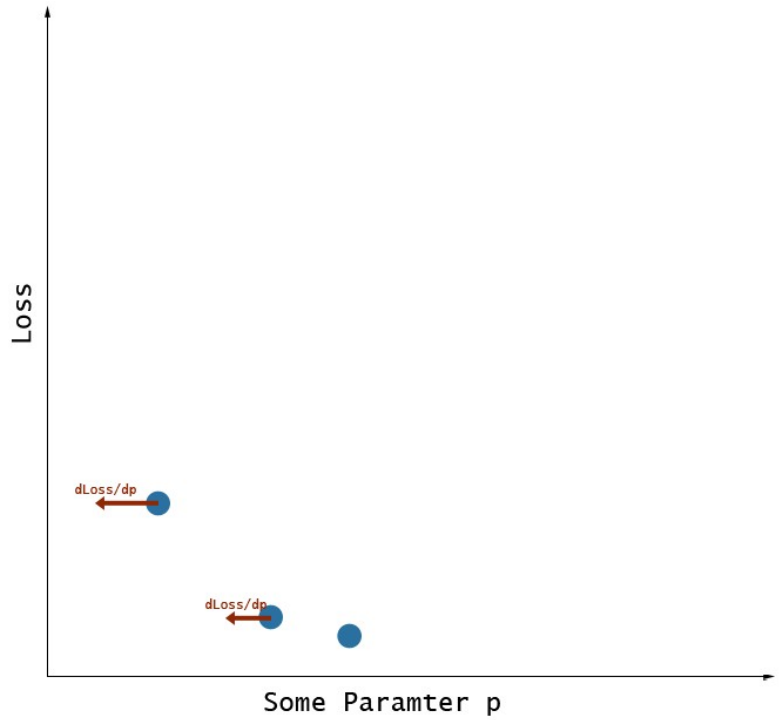
Training a Network

Minimize $Loss(f(x, p), y)$



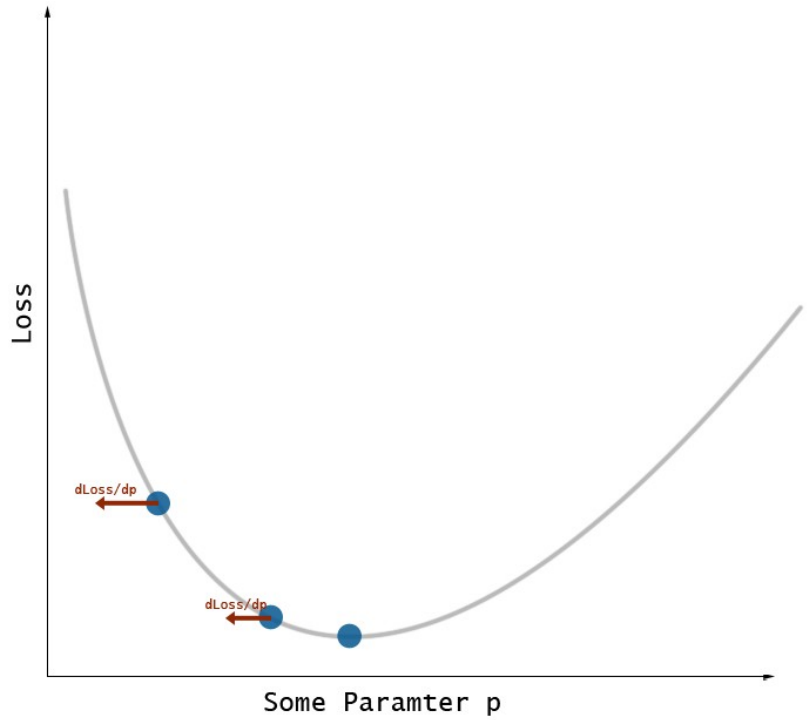
Training a Network

Minimize $Loss(f(x, p), y)$



Training a Network

Minimize $Loss(f(x,p),y)$



Requirement 1: Calculate gradients

```python
import torch

x = torch.arange(10, dtype=torch.float, requires_grad=True)
print(x)  # >>> tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], requires_grad=True)

y = torch.sum(x**3)
print(y) # >>> tensor(2025., grad_fn=<SumBackward0>)

y.backward()
print(x.grad) # >>> tensor([  0.,   3.,  12.,  27.,  48.,  75., 108., 147., 192., 243.])

```

$$\frac{\partial y}{\partial \mathbf{x}}$$

$$f(x) = x^3 \Rightarrow f'(x) = 3x^2 \Rightarrow f'(2) = 12$$

Requirement 1: Calculate gradients

```python
import torch

x = torch.arange(10, dtype=torch.float, requires_grad=True)
print(x)  # >>> tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], requires_grad=True)

y = torch.sum(x**3)
print(y) # >>> tensor(2025., grad_fn=<SumBackward0>)

y.backward()
print(x.grad) # >>> tensor([  0.,   3.,  12.,  27.,  48.,  75., 108., 147., 192., 243.])

```

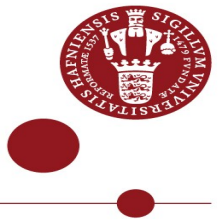
$$\frac{\partial y}{\partial \mathbf{x}}$$

$$f(x) = x^3 \Rightarrow f'(x) = 3x^2 \Rightarrow f'(2) = 12$$

This works for basically any function (and it's not magic!)

Framework feature #2: Use GPUs

Graphical processing units are much faster for deep learning than using CPUs!



Framework feature #2: Use GPUs

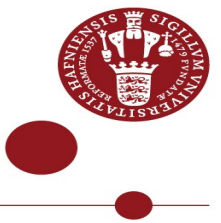
Graphical processing units are much faster for deep learning than using CPUs!

A big **thanks** to:

Requirement 2: GPU

```python
import torch

x = torch.arange(10, dtype=torch.float, requires_grad=True, device="cuda")
print(x)  # >>> tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], device='cuda:0', requires_grad=True)

y = torch.sum(x**3)
print(y) # >>> tensor(2025., device='cuda:0', grad_fn=<SumBackward0>)

y.backward()
print(x.grad) # >>> tensor([  0.,   3.,  12.,  27.,  48.,  75., 108., 147., 192., 243.],
              #         device='cuda:0')

```

The only change



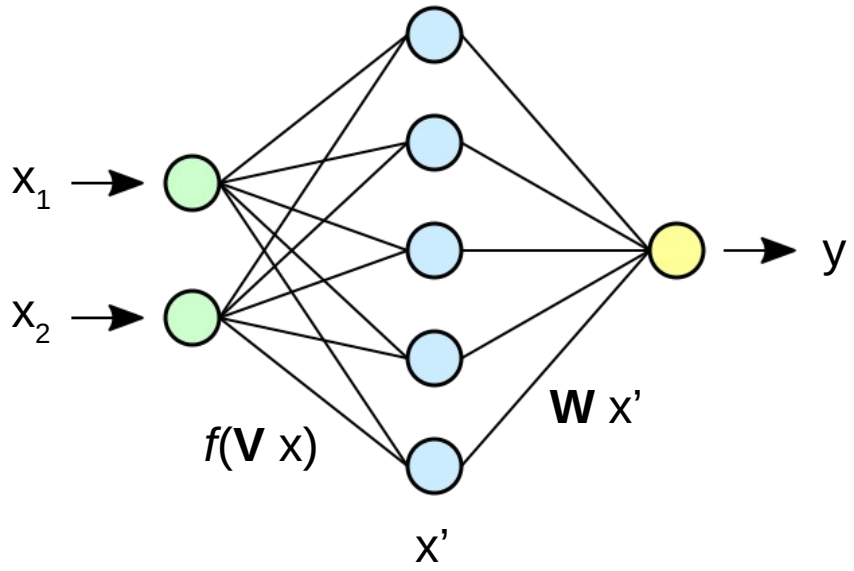
Requirements for DNN Frameworks

$$f(x; p)$$

- Optimisation of parameters *p*
  - Take first order derivatives
  - Chain rule (backpropagation)
- Process large amounts of data fast
  - Exploit GPUs
- Nice to haves:
  - Standard functions and operations built-in
  - Built-in optimizers
  - Spread training across network
  - Compile for fast inference
  - ...

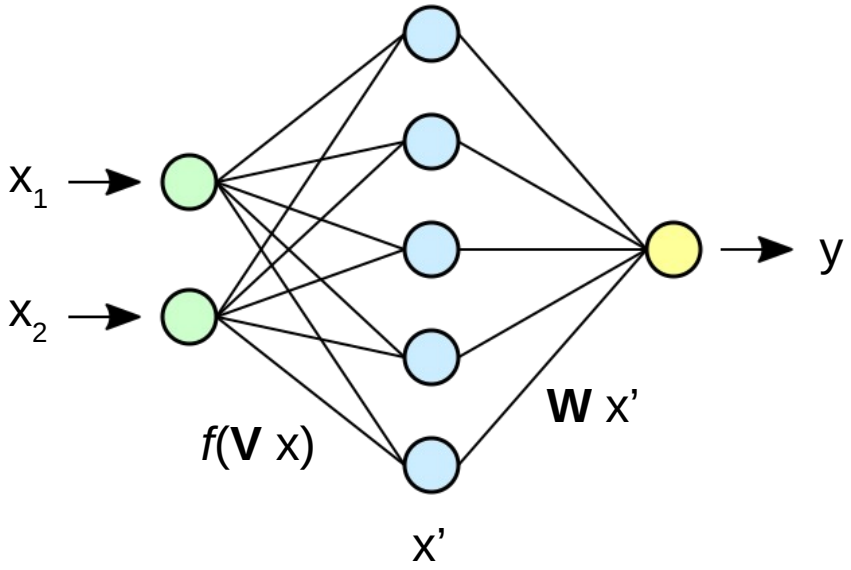
PYTORCH

Simple PyTorch Example



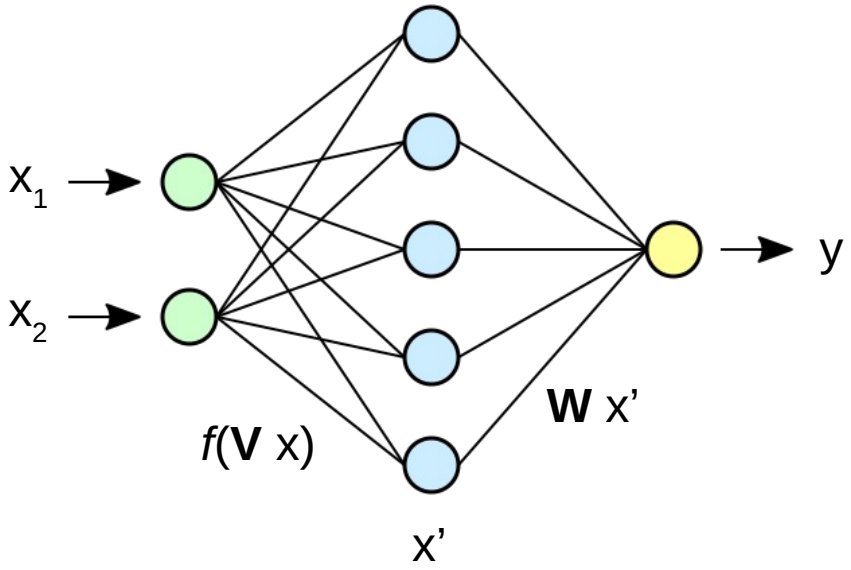

```python
import torch

V = torch.randn((5, 2))
W = torch.randn((1, 5))


def net(x):
    xp = torch.tanh(V @ x)
    return W @ xp


x = torch.tensor([0.1, 1.3])
net(x)
```

Simple PyTorch Example



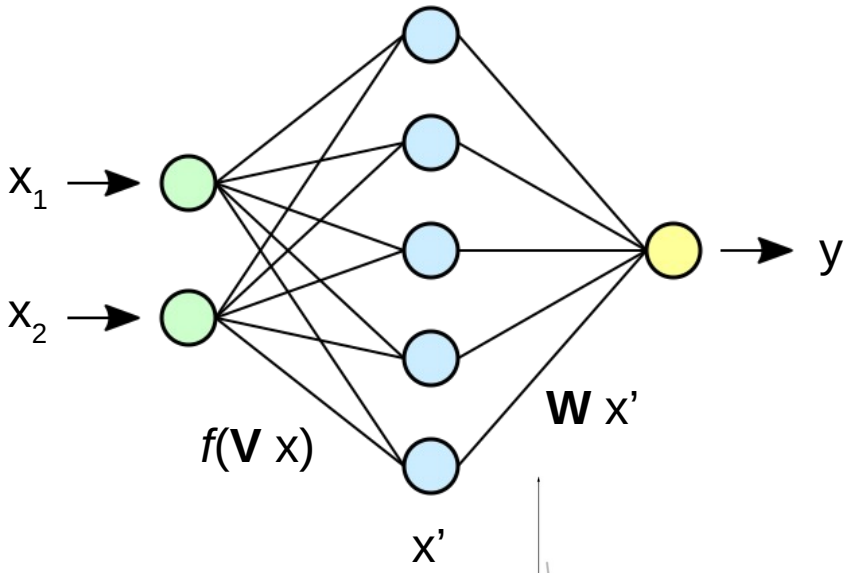

```python
import torch
from torch import nn

net = nn.Sequential(nn.Linear(2, 5),
                    nn.Tanh(),
                    nn.Linear(5, 1))

x = torch.tensor([0.1, 1.3])
net(x)
```

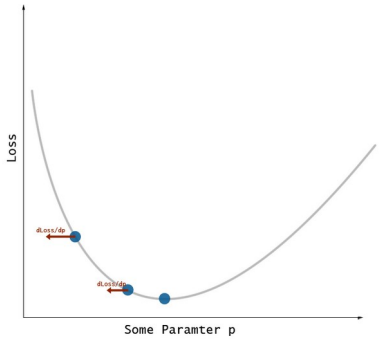
More control...




```python
import torch
from torch import nn


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_1 = nn.Linear(2, 5)
        self.linear_2 = nn.Linear(5, 1)

    def forward(self, x):
        xp = torch.tanh(self.linear_1(x))
        return self.linear_2(xp)


net = Net()
x = torch.tensor([0.1, 1.3])
net(x)

```

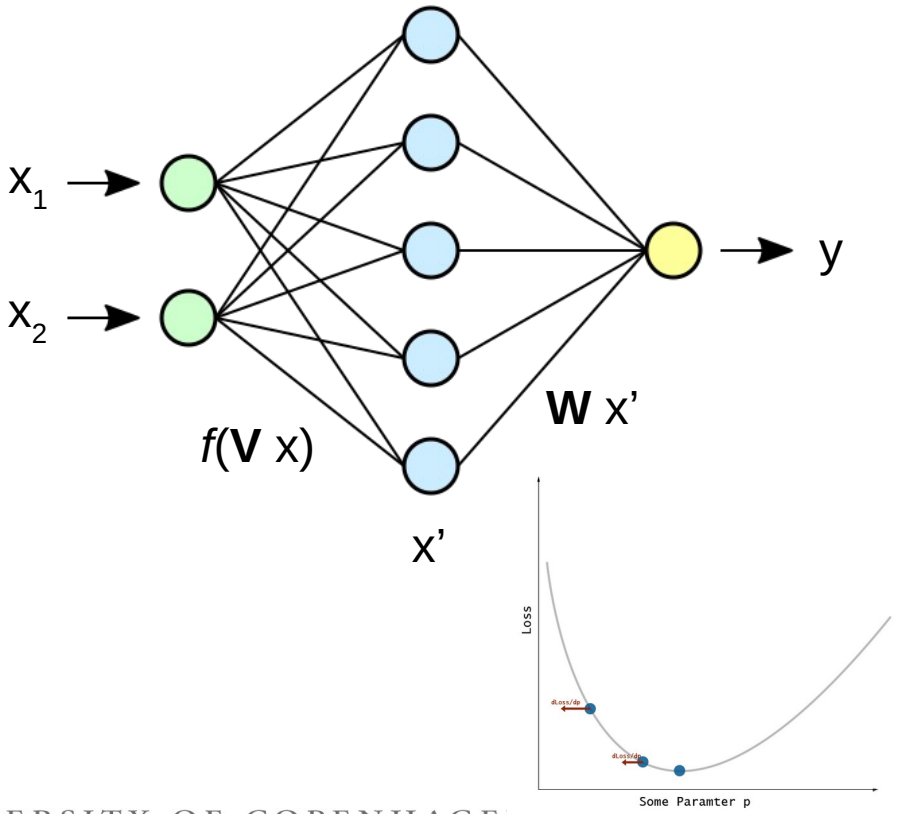
How to train in PyTorch





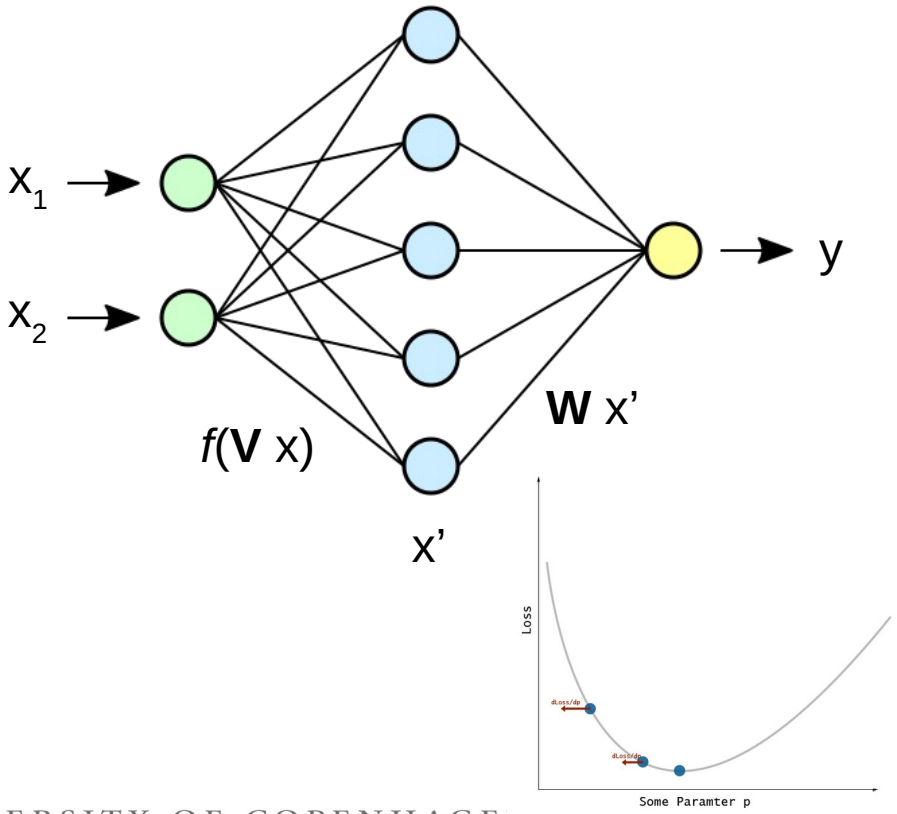

```python
import torch
from torch import nn


net = nn.Sequential(nn.Linear(2, 5),
                    nn.Tanh(),
                    nn.Linear(5, 1))

x_data = torch.randn((1000, 2))
y_data = torch.sum(x_data**2, dim=1)

```

How to train in PyTorch




```python
import torch
from torch import nn


net = nn.Sequential(nn.Linear(2, 5),
                    nn.Tanh(),
                    nn.Linear(5, 1))

x_data = torch.randn((1000, 2))
y_data = torch.sum(x_data**2, dim=1)

lr = 0.01  # learning rate
iterations = 250
for _ in range(iterations):
    prediction = net(x_data)
    loss = torch.mean((prediction - y_data)**2)
    loss.backward()  # calculate gradients
    for p in net.parameters():
        with torch.no_grad():
            p -= p.grad * lr
            p.grad.zero_()
```

How to train in PyTorch

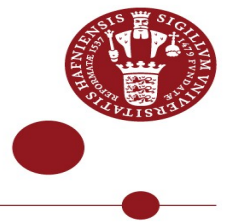



```python
import torch
from torch import nn


net = nn.Sequential(nn.Linear(2, 5),
                    nn.Tanh(),
                    nn.Linear(5, 1))

x_data = torch.randn((1000, 2))
y_data = torch.sum(x_data**2, dim=1)

lr = 0.01  # learning rate
iterations = 250
opt = torch.optim.SGD(net.parameters(), lr)
for _ in range(iterations):
    prediction = net(x_data)
    loss = torch.mean((prediction - y_data)**2)
    loss.backward()  # calculate gradients
    opt.step()
    opt.zero_grad()

```

Better optimiser stepping

$$p_{n+1} = p_n - \nabla_{p_n} \mathcal{L} \cdot \text{learning rate}$$

- What if some gradients are much smaller than others?
- What happens when gradients disappear when loss is small?



Better optimiser stepping

$$p_{n+1} = p_n - \nabla_{p_n} \mathcal{L} \cdot \text{learning rate}$$

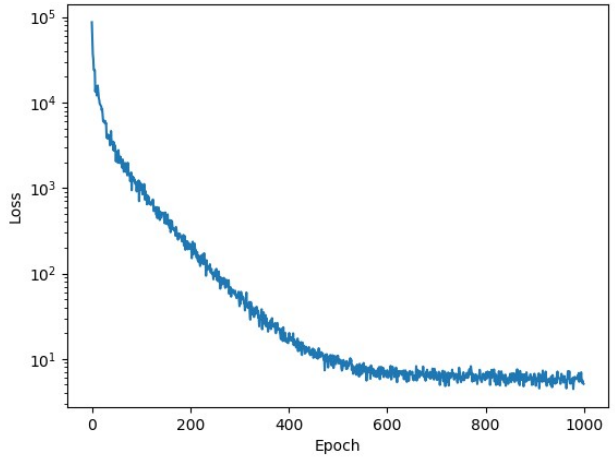
- What if some gradients are much smaller than others?
- What happens when gradients disappear when loss is small?

Solution : Variable learning rates and momentum

- Many algorithms exists, perhaps most popular: “Adam”

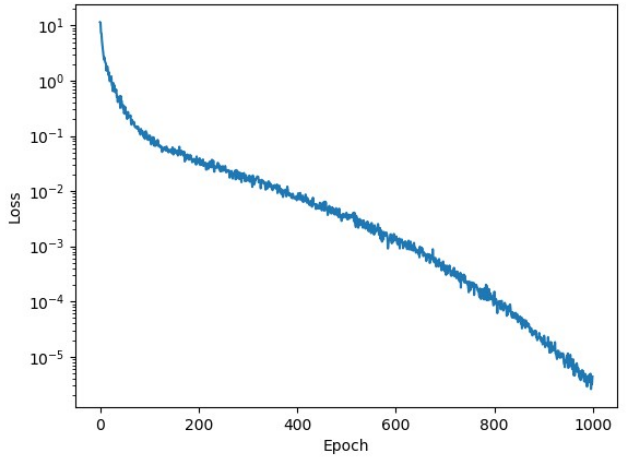
Better optimiser stepping

SGD (Stochastic gradient descent)




```python
opt = torch.optim.SGD(net.parameters(), lr)
for _ in range(iterations):
    prediction = net(x_data)
    loss = torch.mean((prediction - y_data)**2)
    loss.backward()  # calculate gradients
    opt.step()
    opt.zero_grad()
```

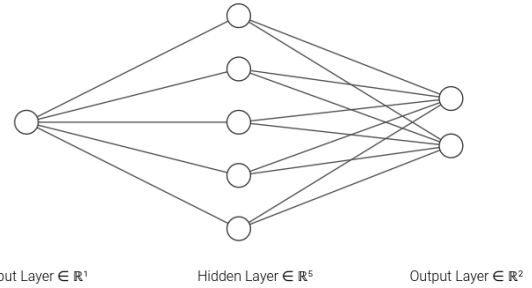
Adam (Adaptive Moment Estimation)



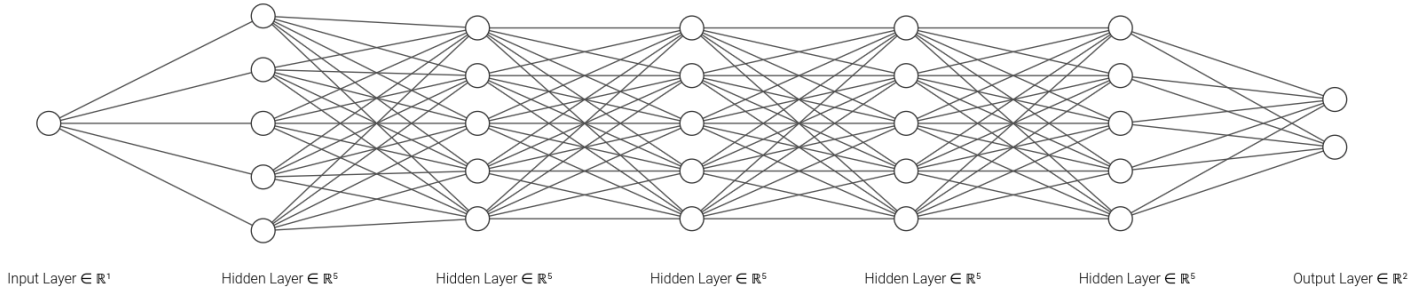

```python
opt = torch.optim.Adam(net.parameters(), lr)
for _ in range(iterations):
    prediction = net(x_data)
    loss = torch.mean((prediction - y_data)**2)
    loss.backward()  # calculate gradients
    opt.step()
    opt.zero_grad()
```

A more complex example:

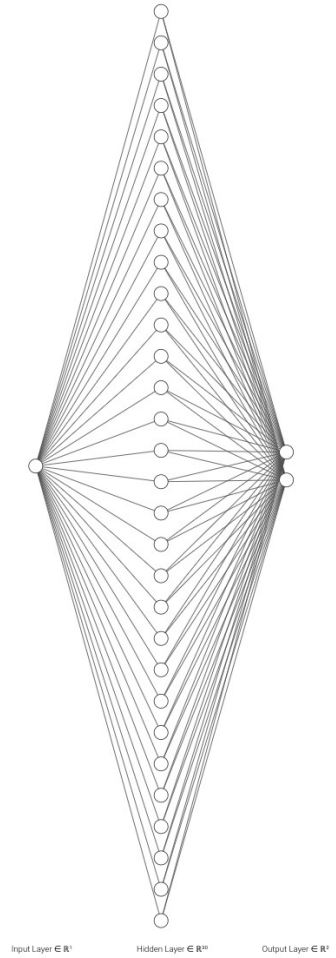
Neural networks are *universal function approximators*



Deep network



Broad network



Loss functions

Regression

"Output of network is a number that should be close to label *y*

L2 loss:

$$L(x, y) = (\mathrm{NN}(x) - y)^2$$

Loss functions

Regression

"Output of network is a number that should be close to label *y*

L2 loss:

$$L(x, y) = (\mathrm{NN}(x) - y)^2$$

Classification

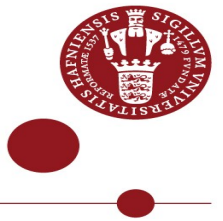
"Output of network is a probability distribution over labels (y_i = 1 for true label and = 0 otherwise).

Cross-entropy loss:

$$\mathbf{p} = \mathrm{NN}(x)$$

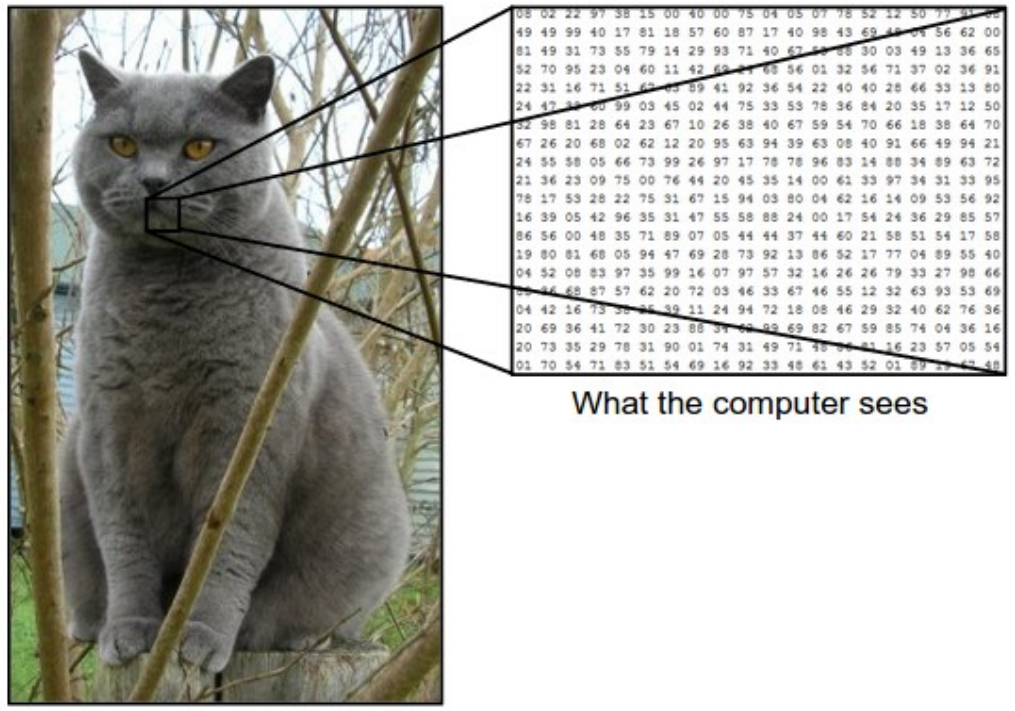
$$L(\mathbf{p}, \mathbf{y}) = -\sum_i y_i \log(p_i)$$

Convolutional Neural Networks



Convolutional Neural Networks

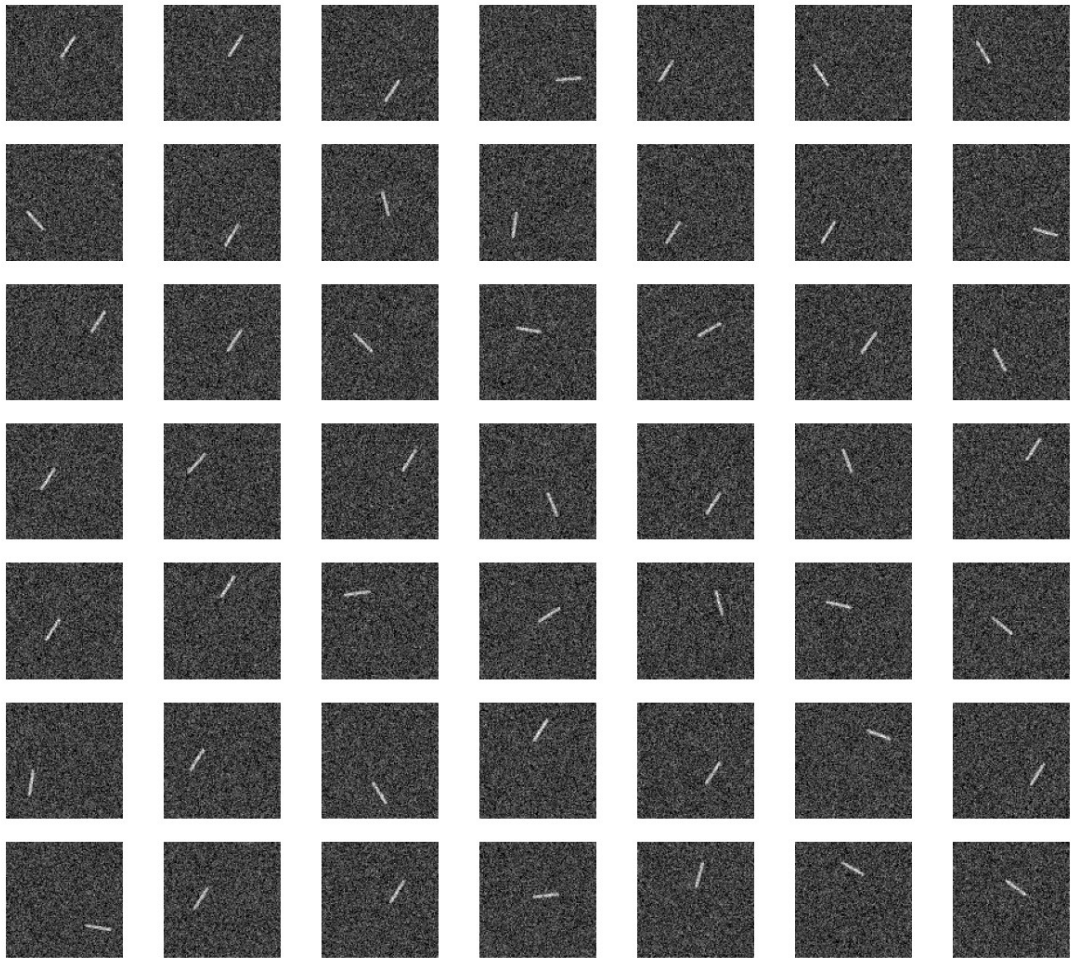
Exemplified using image classification..



What the computer sees

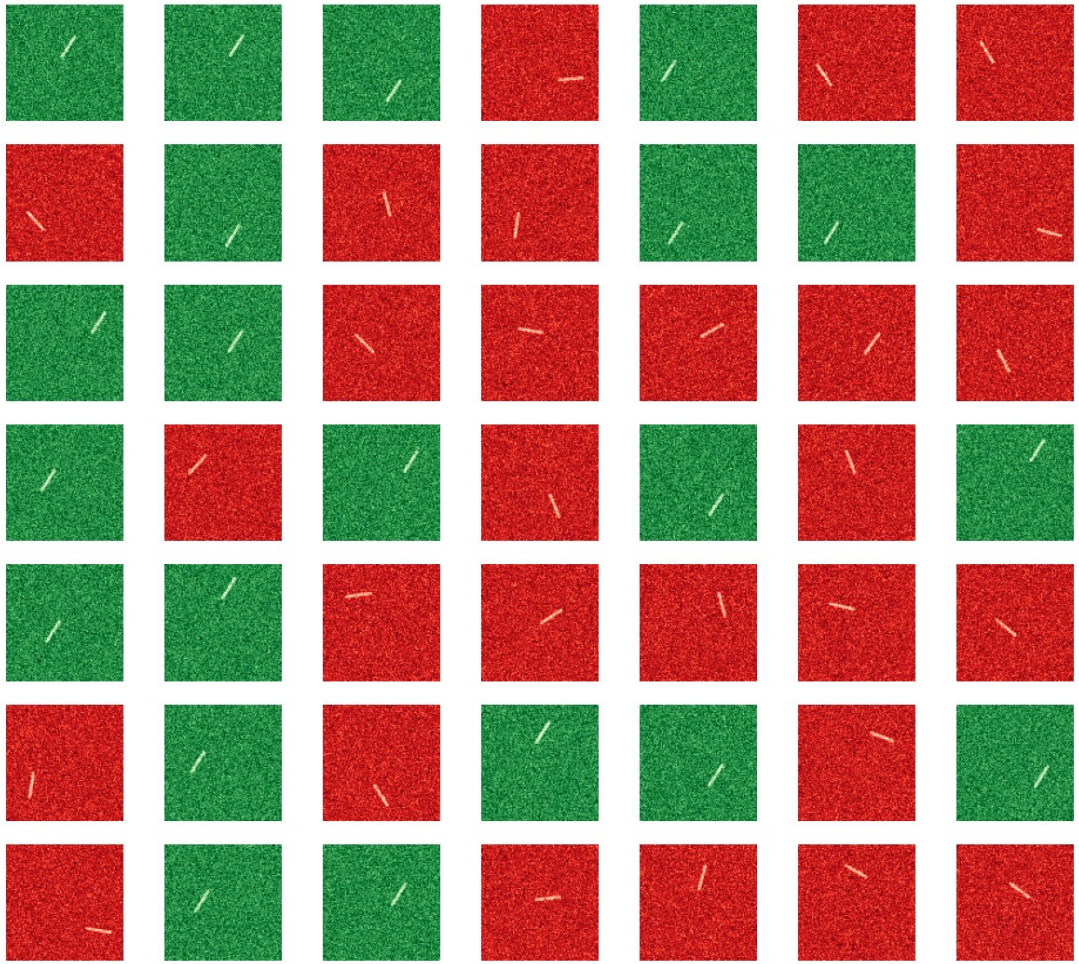
Convolutional Neural Networks

Find images with angle:



Convolutional Neural Networks

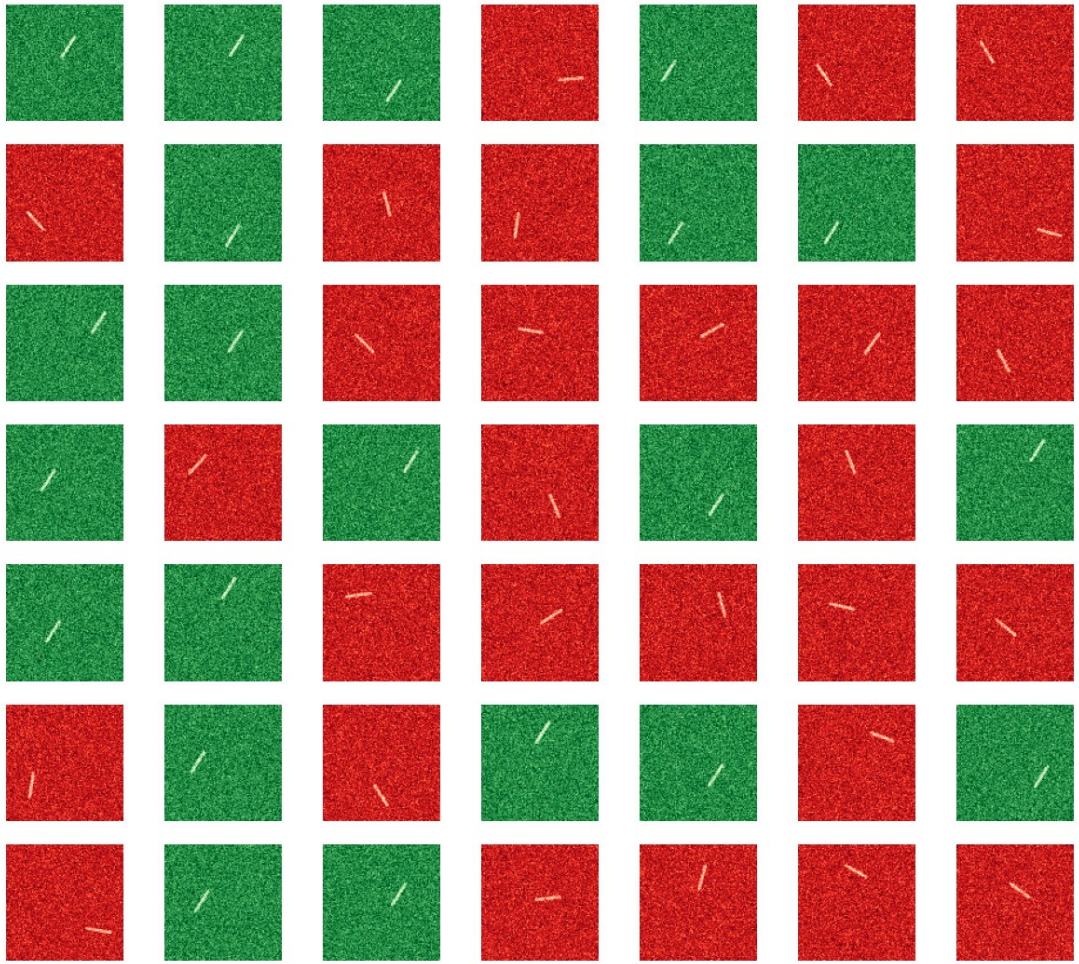
Find images with angle:



Convolutional Neural Networks

Find images with angle:

We don’t care where in the images the line is! (like: *“is it a cat or not?”*)



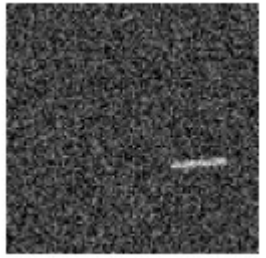
Convolutional Neural Networks

What we want:

NN() = *large number (high probability)*

NN() = *small number (low probability)*

NN() = *large number (high probability)*

NN() = *small number (low probability)*

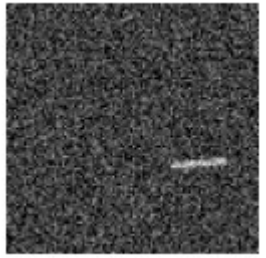
Convolutional Neural Networks

What we want:

NN() = *large number (high probability)*

NN() = *small number (low probability)*

NN() = *large number (high probability)*

NN() = *small number (low probability)*

How can we design such a function?

Convolutional Neural Networks

Define a *kernel, k* = 

Convolutional Neural Networks

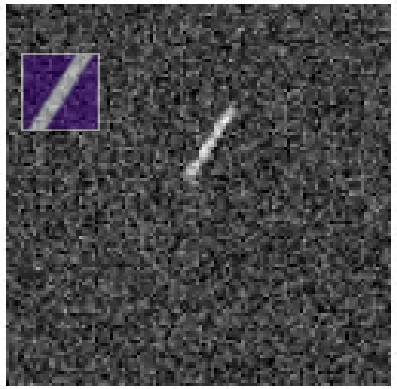
Define a *kernel, k* = 

(color just for visualisation purposes. Think of this a matrix of numbers)

Convolutional Neural Networks

Define a *kernel, k* = 

Place kernel somewhere on image, multiply and sum:

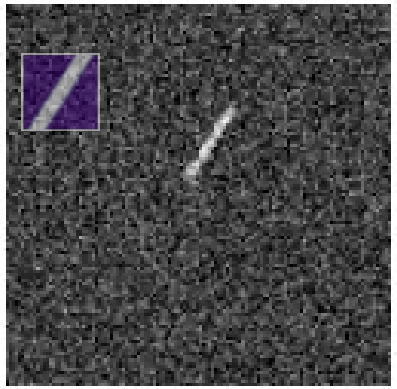


= small number

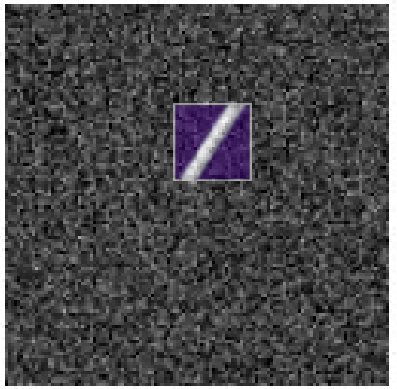
Convolutional Neural Networks

Define a *kernel, k* = 

Place kernel somewhere on image, multiply and sum:



= small number

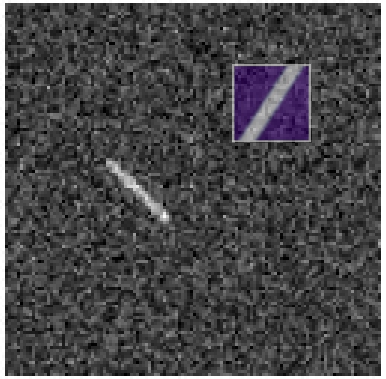


= large number

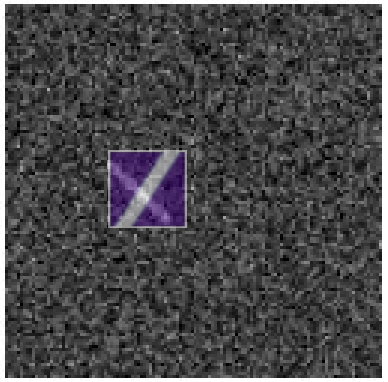
Convolutional Neural Networks

Define a *kernel, k* = 

What happens for a “wrong” image?



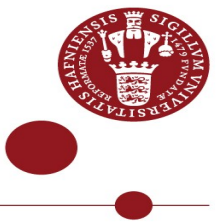
= small number



= small(ish) number

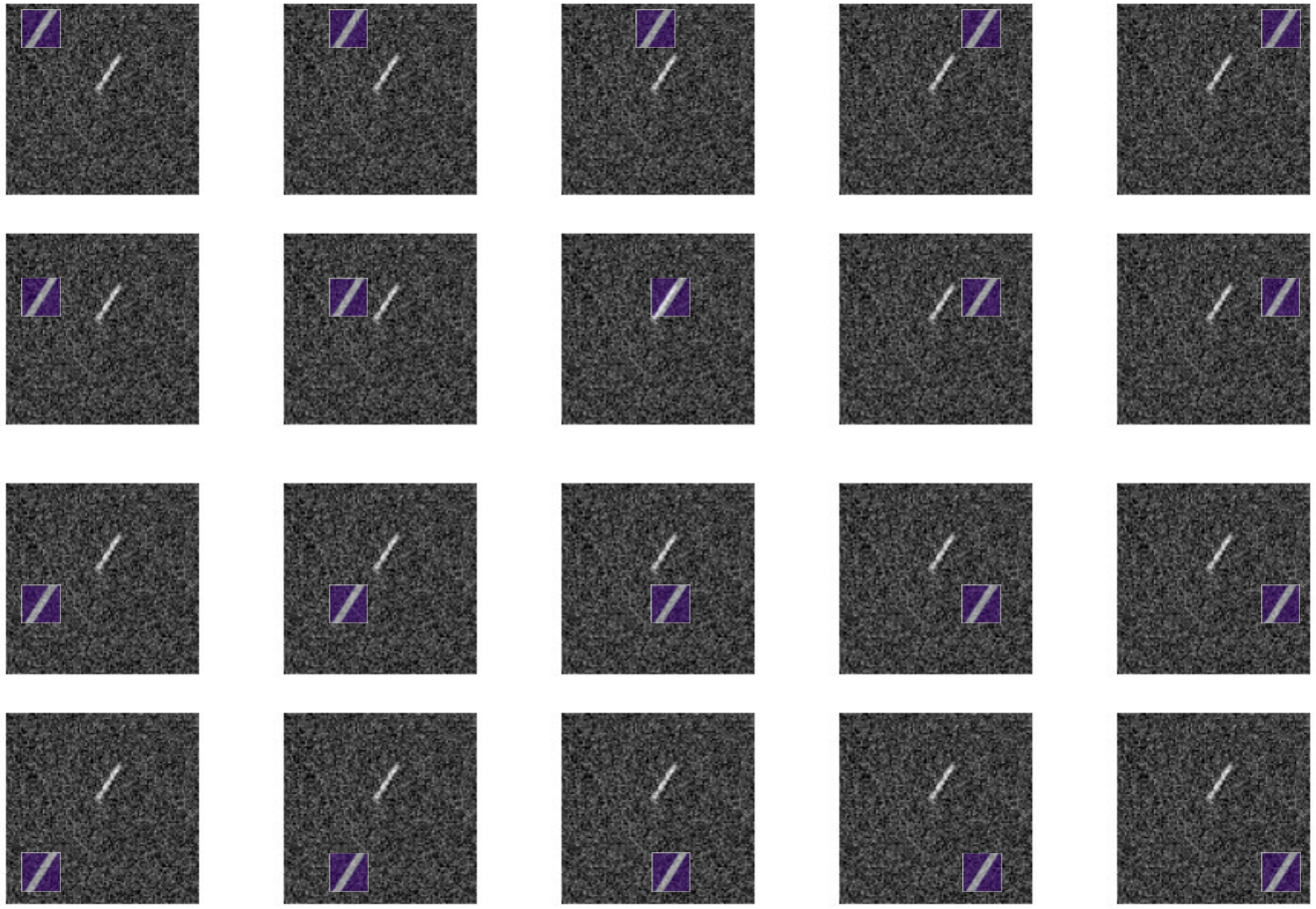
Convolutional Neural Networks

Idea: try all locations of kernel and find maximum:



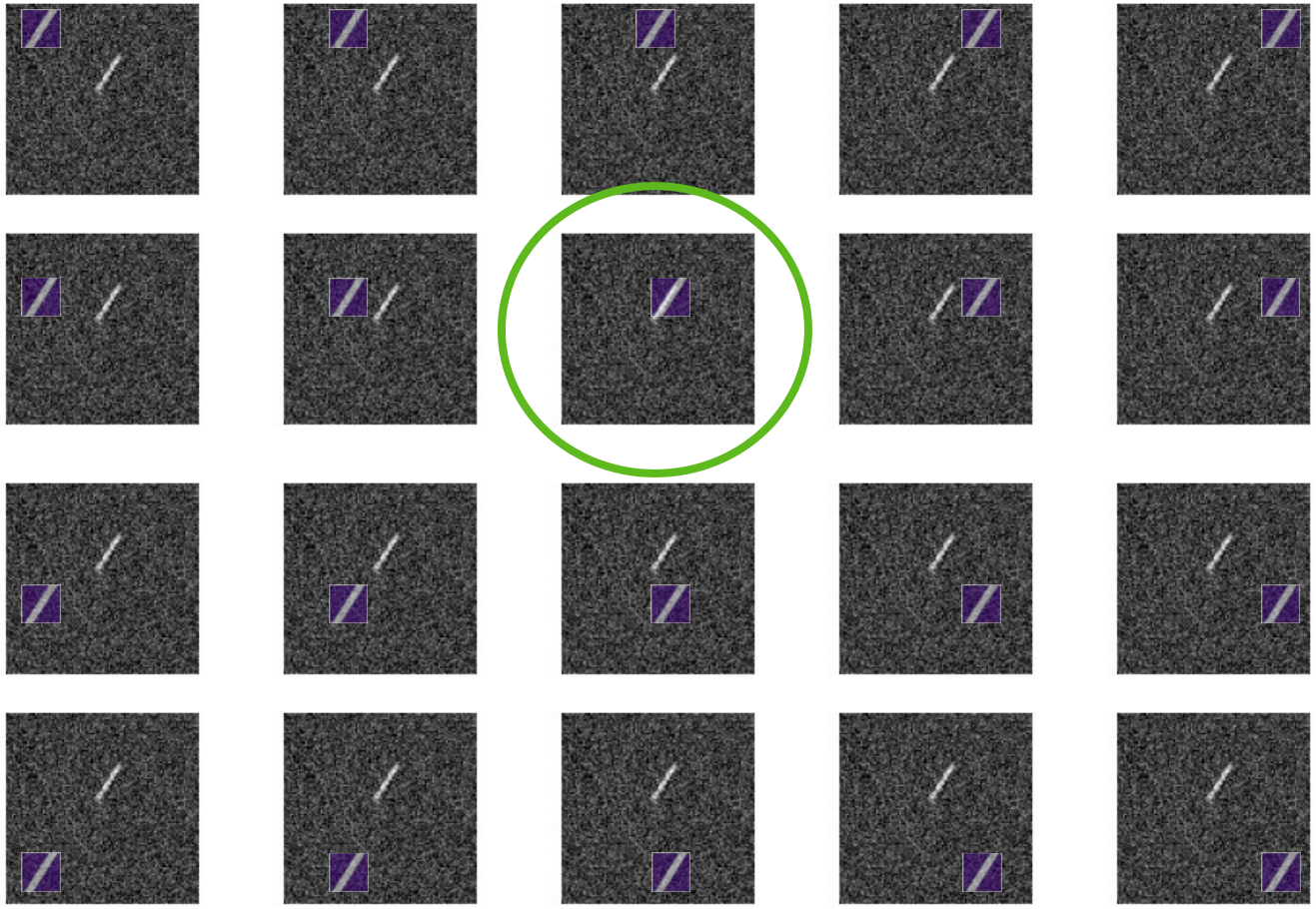
Convolutional Neural Networks

Idea: try all locations of kernel and find maximum:

Max()

Convolutional Neural Networks

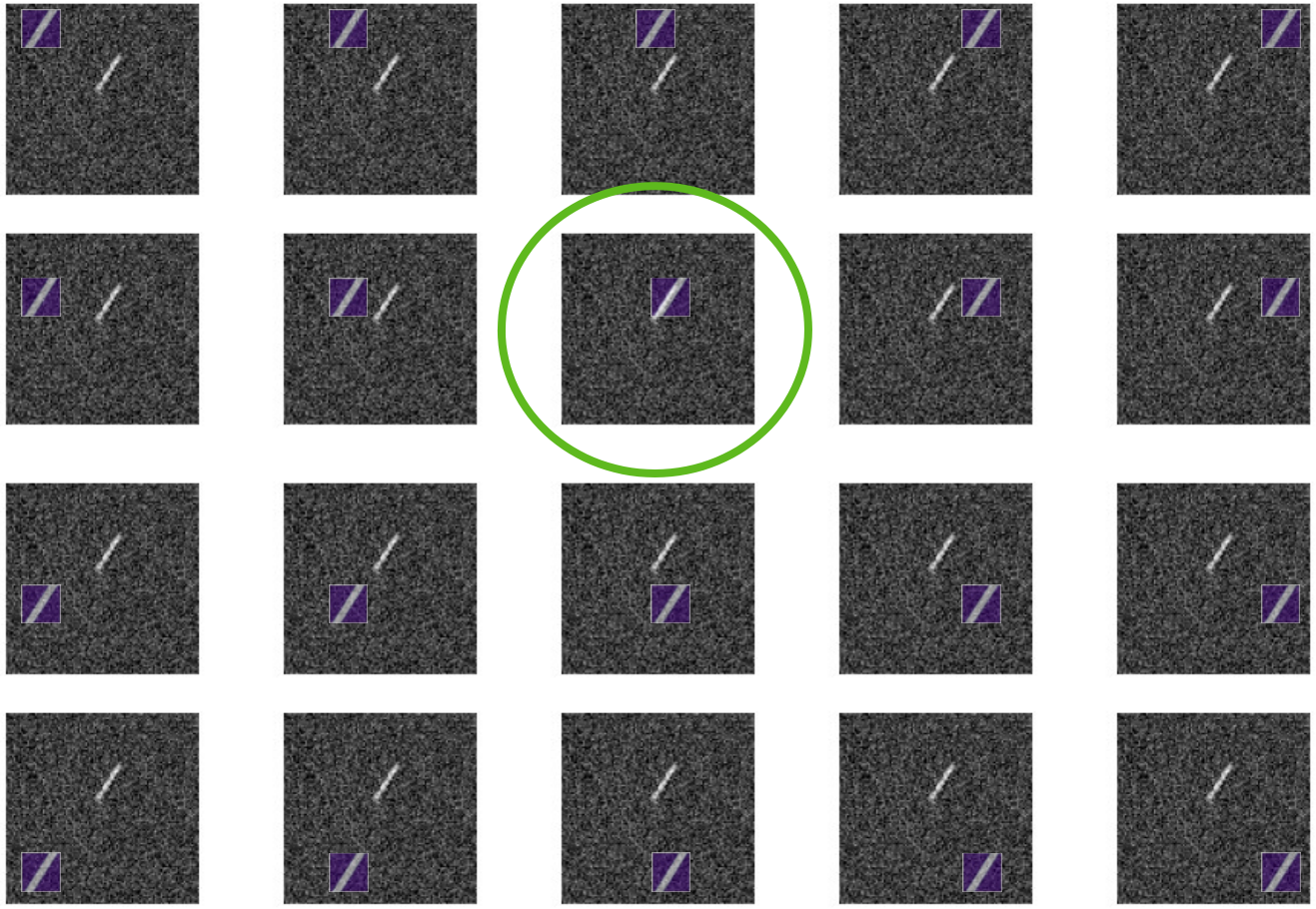
Idea: try all locations of kernel and find maximum:

Max()

Convolutional Neural Networks

Idea: try all locations of kernel and find maximum:

Max(



)

This is called *Max-Pooling*, we could also do *Average-pooling*

Convolutional Neural Networks

This is precisely what a convolution does!

NN() = max(conv2D( )

Convolutional Neural Networks

This is precisely what a convolution does!

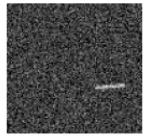
NN() = max(conv2D( )

We have our function!
Indeed:

NN() = *large*

NN() = *large*

NN() = *small*

NN() = *small*

Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

We handcrafted the kernel. In CNNs we train to choose the best kernels

Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

We handcrafted the kernel. In CNNs we train to choose the best kernels

```
net = nn.Sequential(nn.Conv2d(1, 1, 11),
                    nn.AdaptiveMaxPool2d(1),
                    nn.Linear(1, 1))
```

```
# Soft-max probabilities -> cross entropy
p = torch.exp(pred) / (torch.exp(pred) + torch.exp(-pred))
loss = -torch.mean(labels * torch.log(p) + (1 - labels) * torch.log(1 - p))  # (this can be done smarter!,
                                                                              # no reason to take exp, then log)
```

Run code!

Convolutional Neural Networks

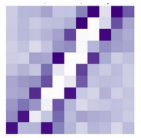
But we didn't train anything?

NN() = max(conv2D( )

We handcrafted the kernel. In CNNs we train to choose the best kernels

```
net = nn.Sequential(nn.Conv2d(1, 1, 11),
                    nn.AdaptiveMaxPool2d(1),
                    nn.Linear(1, 1))
```

```
# Soft-max probabilities -> cross entropy
p = torch.exp(pred) / (torch.exp(pred) + torch.exp(-pred))
loss = -torch.mean(labels * torch.log(p) + (1 - labels) * torch.log(1 - p))  # (this can be done smarter!,
                                                                              # no reason to take exp, then log)
```

Run code! 

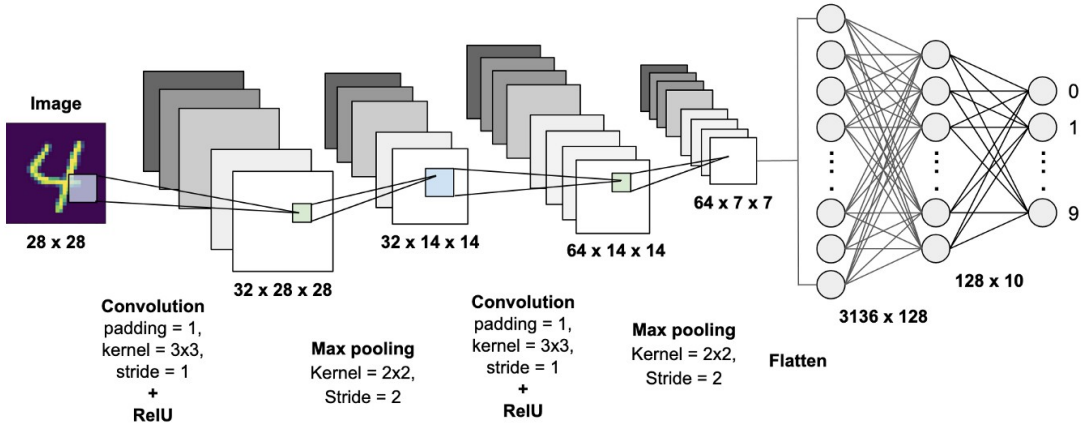
(slightly smarter than ours!, albeit noisy)

Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

We then make it deep, so kernels can be combined:

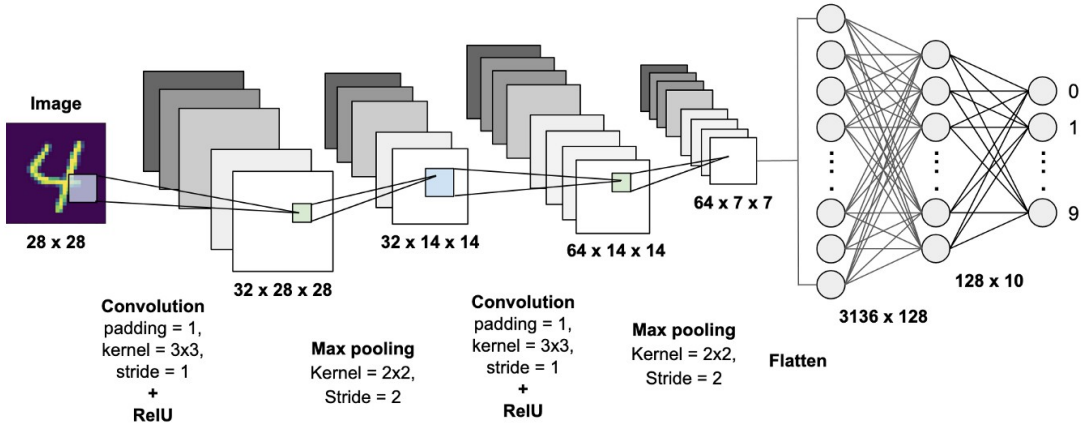


Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

We then make it deep, so kernels can be combined:



The number "4" is made up of four small lines

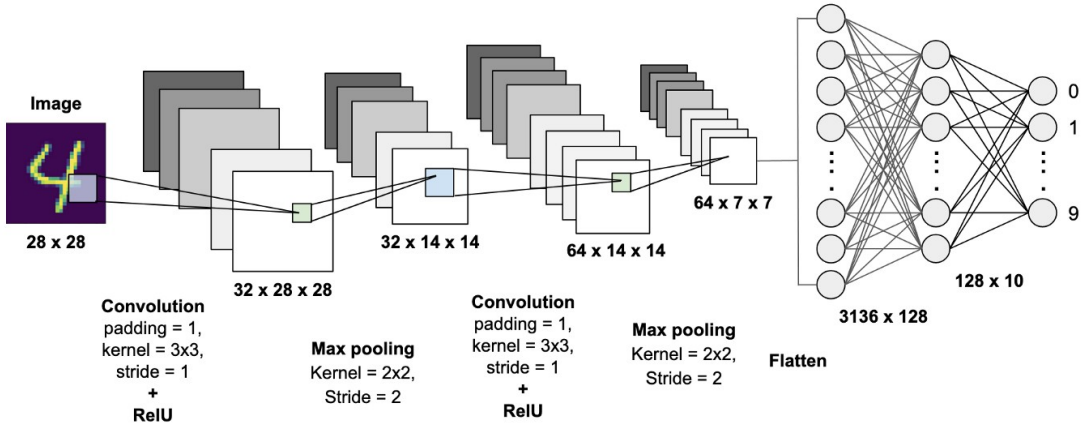
Convolutional Neural Networks

But we didn't train anything?

NN() = max(conv2D( )

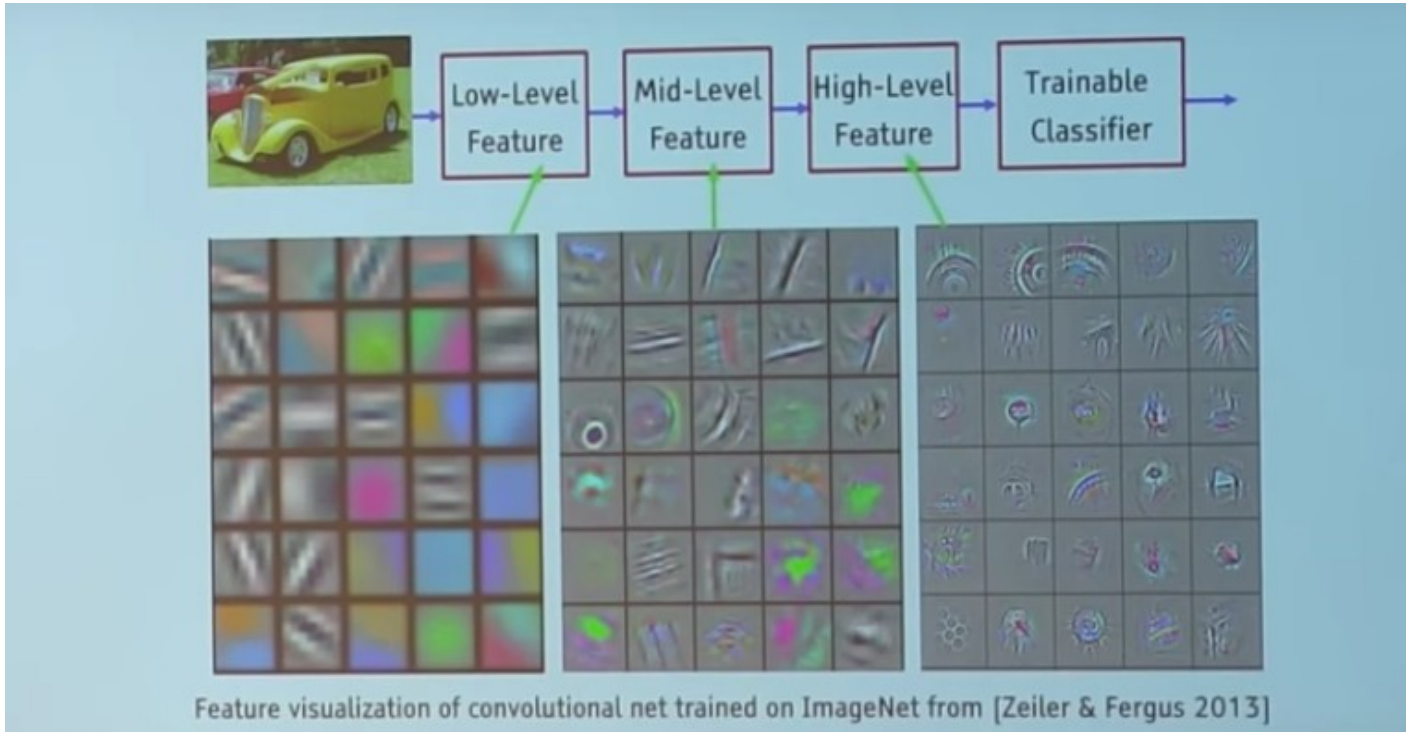
For deep networks, this max is taken over a few neighbors at a time, not entire image.

We then make it deep, so kernels can be combined:

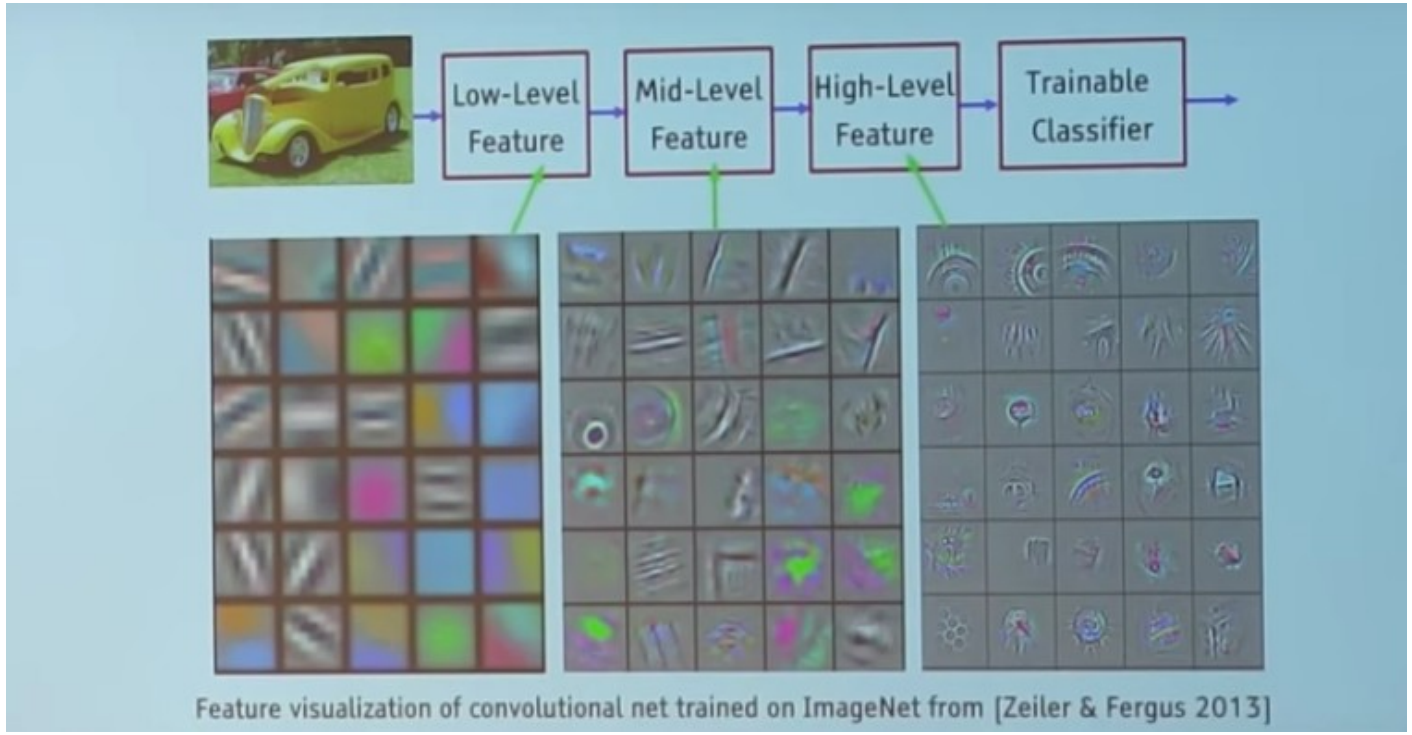


The number "4" is made up of four small lines

Convolutional Neural Networks

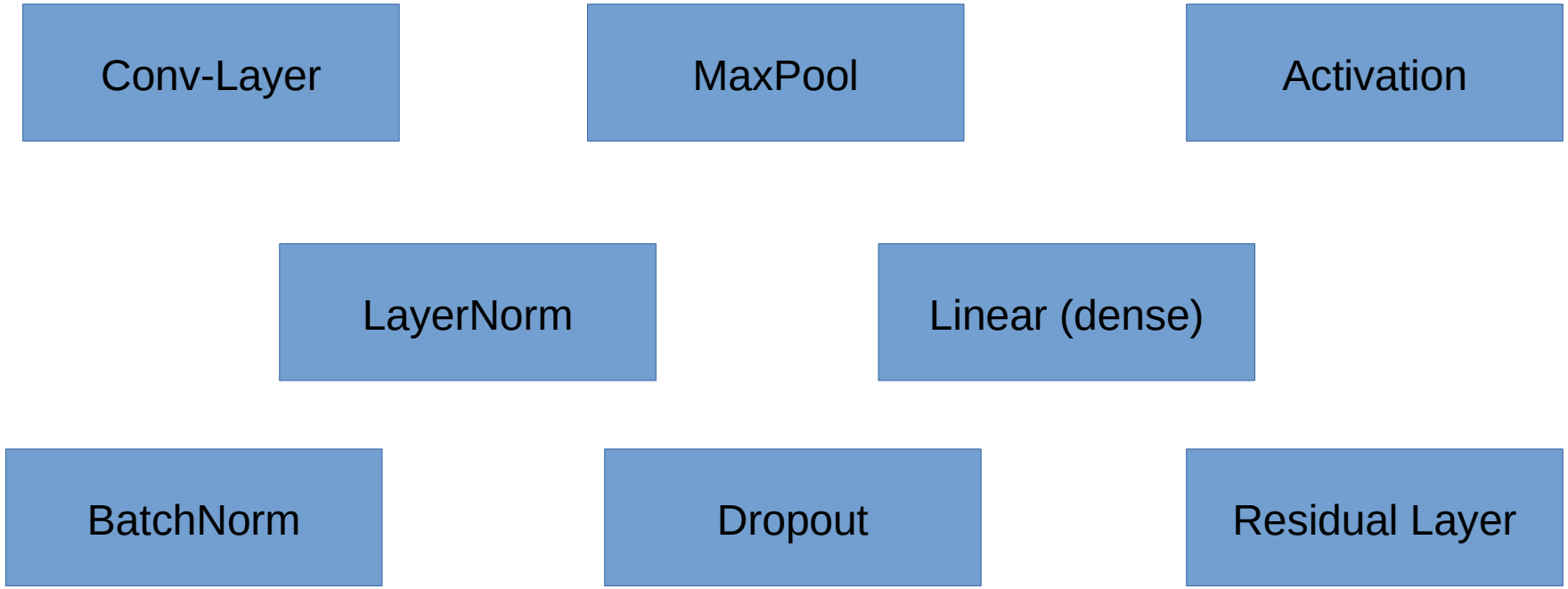


Convolutional Neural Networks

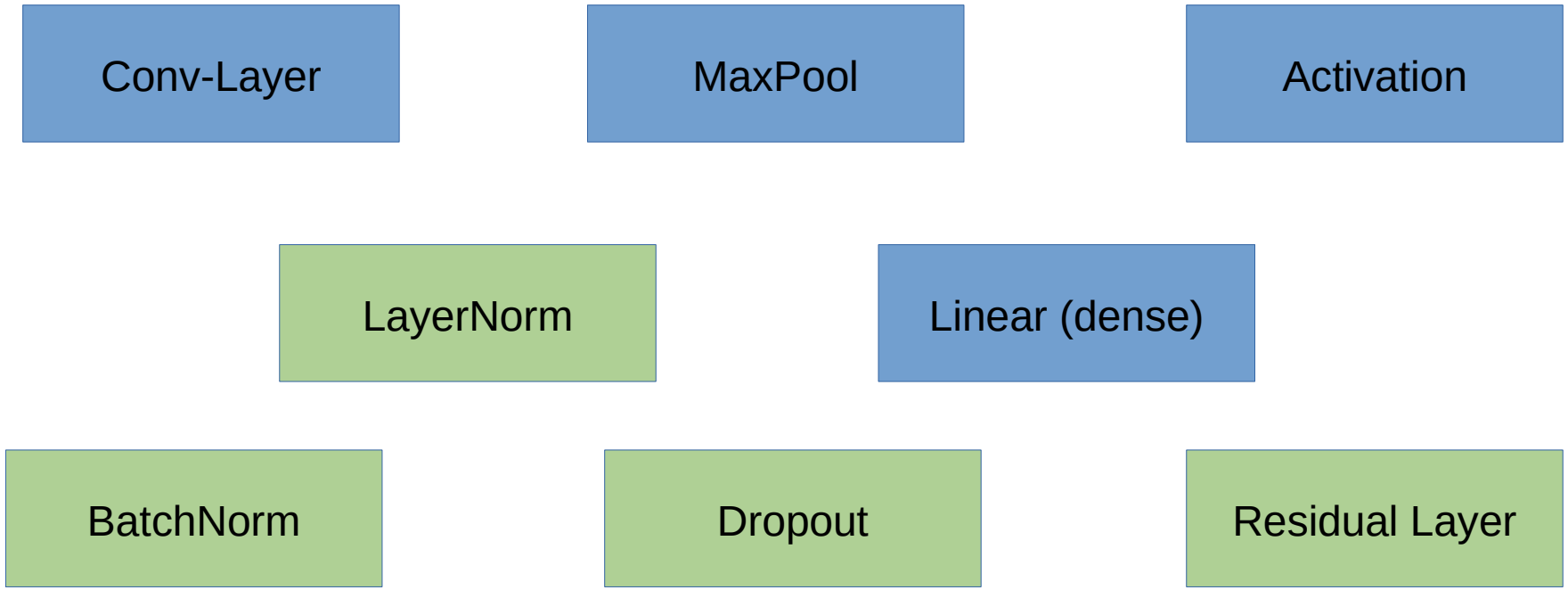


https://poloclub.github.io/cnn-explainer/
http://www.cs.cmu.edu/~aharley/nn_vis/cnn/2d.html

CNN: The Building Blocks

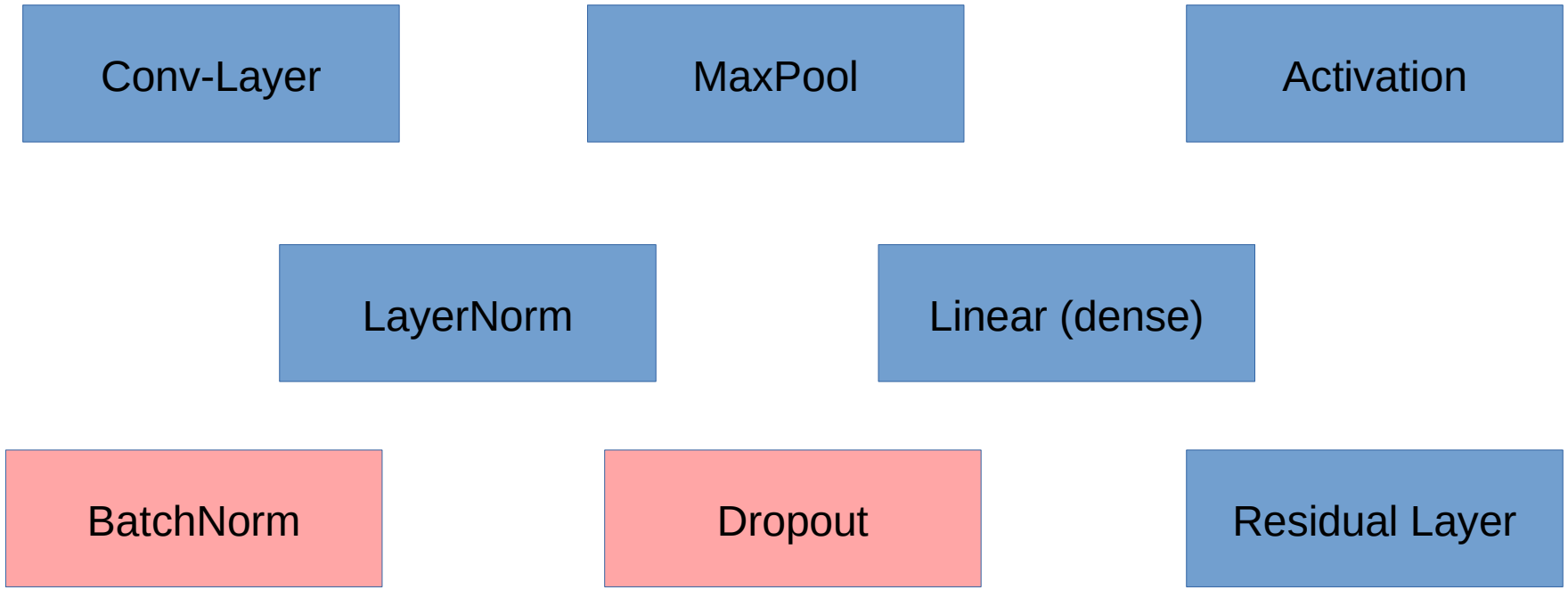


CNN: The Building Blocks



These layers can make training easier (but do not in fact change the type of functions that the NN can fit).

CNN: The Building Blocks



These layers behave differently during training and evaluation:
net.train()
net.eval()

CNN: A *deep* example:

```
nn.Sequential(nn.Conv2d(1, 32, 7),
              nn.BatchNorm2d(32),
              nn.ReLU(),
              nn.MaxPool2d(3),
              nn.Conv2d(32, 32, 3),
              nn.BatchNorm2d(32),
              nn.ReLU(),
              nn.Conv2d(32, 3, 3),
              nn.MaxPool2d(3),
              nn.Flatten(),
              nn.Linear(432, 20),
              nn.ReLU(),
              nn.Linear(20, 1),
              nn.Sigmoid())
```