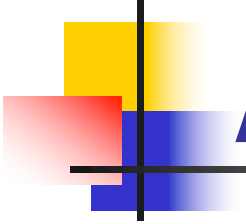
Deep Learning

Dr. Tom Arodz



Agenda

Automated Differentiation (AD)

PyTorch example of Automated Differentiation

Problems in training Deep Networks and how to overcome them

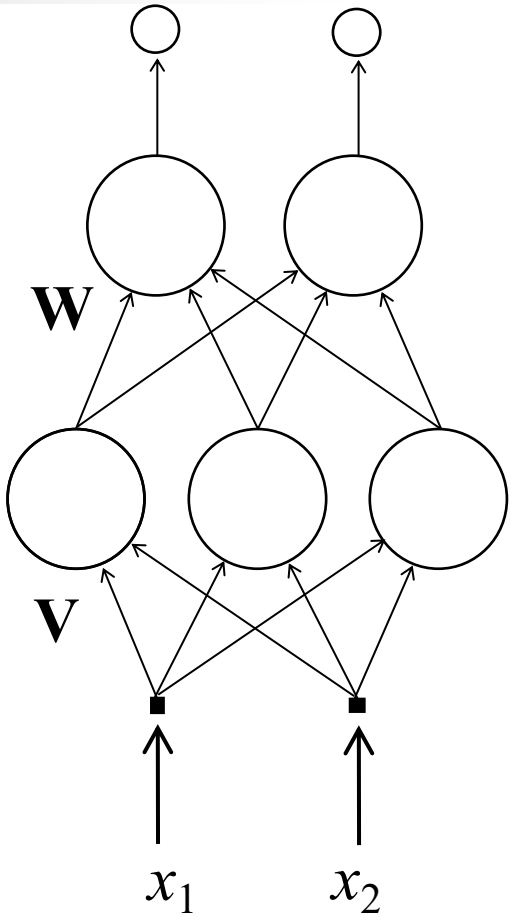
Convolutional Neural Networks (CNNs)

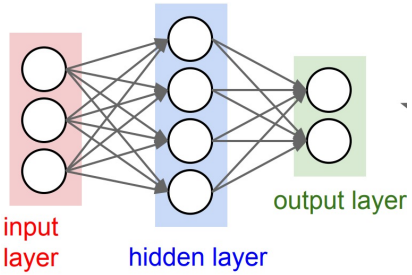
Residual Networks (ResNets)

Self-attention-based Networks (Transformers)

Deep Learning in a Nutshell

- Choose architecture
  - How input data flows towards the output
  - What the trainable parameters (aka weights) are
- Choose the objective function
  - How to measure if training is successful
- Choose the optimization method
  - How to update the weights





Optimization method

- Choose the optimization method
  - First-order method: gradient descent

$$\nabla f(\mathbf{z}) = \left(\frac{\partial}{\partial x_1} f(\mathbf{x})|_{\mathbf{x}=\mathbf{z}}, \ldots, \frac{\partial}{\partial x_n} f(\mathbf{x})|_{\mathbf{x}=\mathbf{z}} \right)$$

Gradient descent:

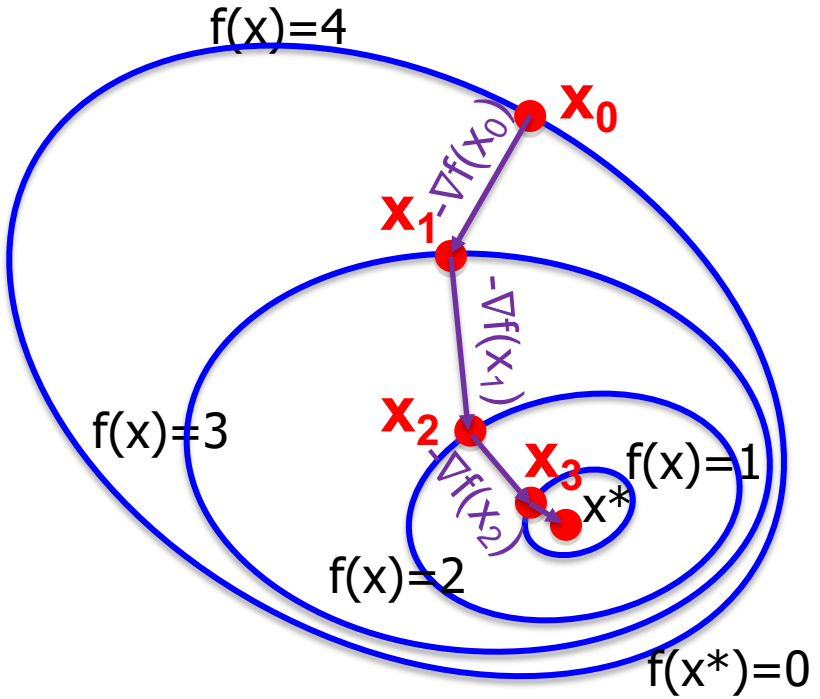
We start from x_0

We calculate $x_1 = x_0 - \nabla f(x_0)/L$

We calculate $x_2 = x_1 - \nabla f(x_1)/L$

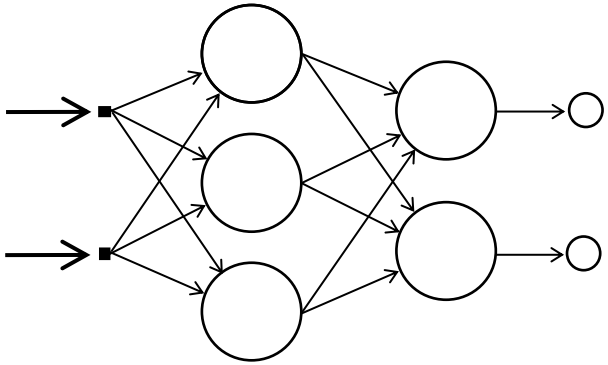
$x_{n+1} = x_n - \nabla f(x_n)/L$

If we choose L large enough g.d. goes down in each step, converging towards local minimum



Deep Learning in a Nutshell

- Choose the optimization method
  - First-order method: gradient descent



- Many architectures, many objective functions, many gradient-based learning techniques

- One framework to rule them all:
  - **Automated differentiation**

Automatic differentiation

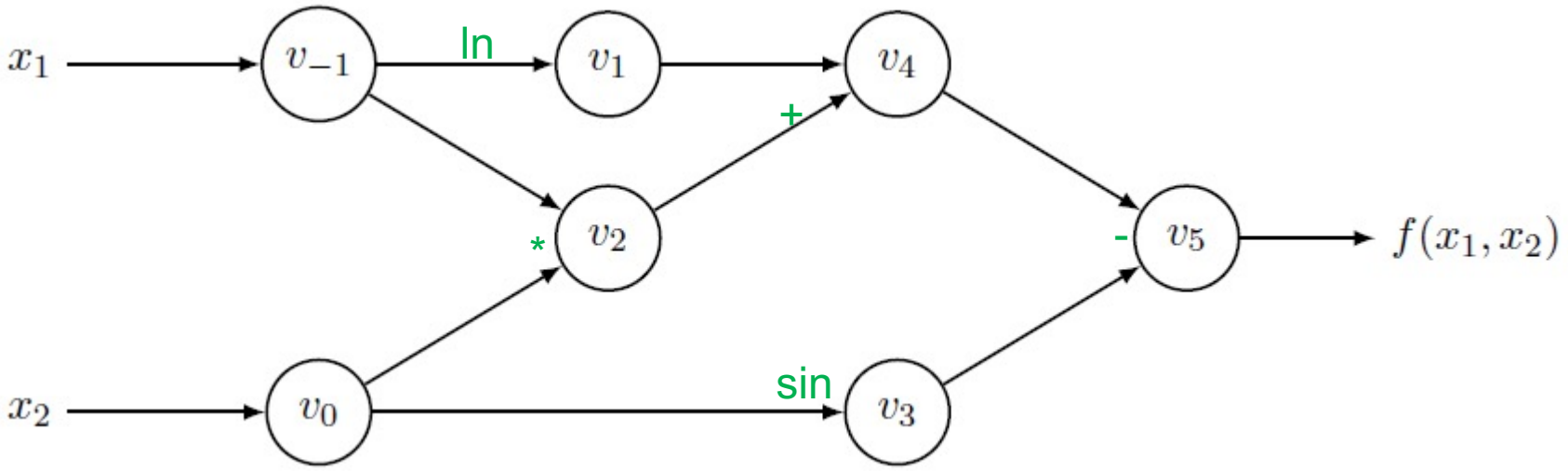
Performing calculations and derivatives on a *computational graph*

Example: $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$

We want to compute y and dy/dx_1

e.g. y is the prediction, x_1 is the weight, x_2 is input data

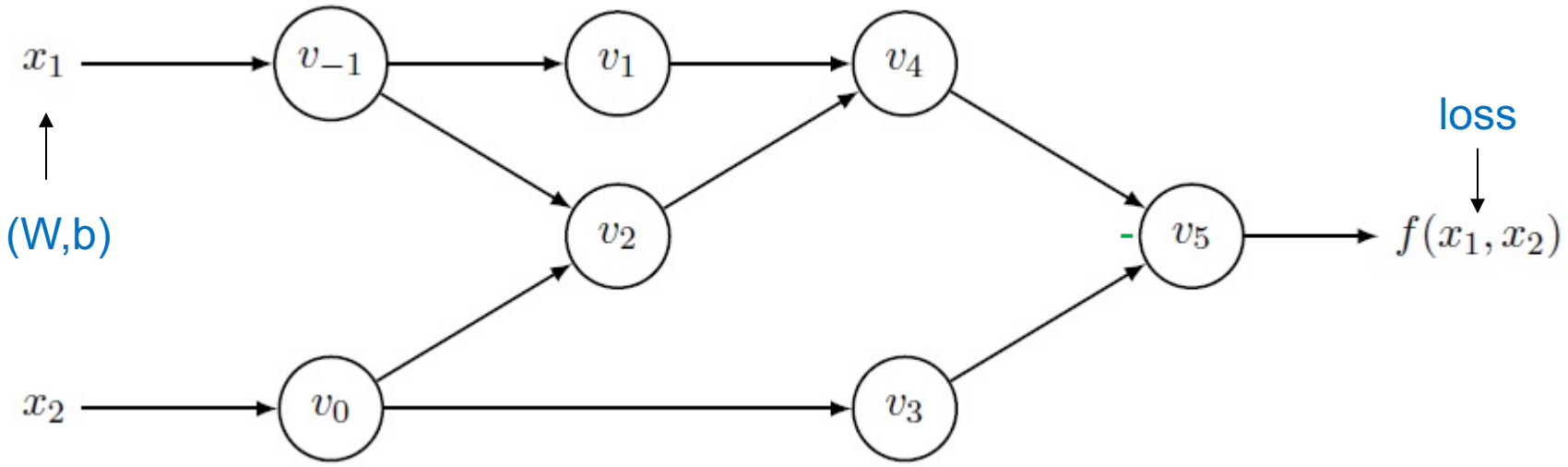
$$(x_1, x_2) = (2, 5)$$



Automatic differentiation

In real deep network, we will have model weights (W,b) as the variables in the computational graph for which we want gradients (like x_1 in this example)

- training data x,y would just be constants on input to the graph (like x_2 here)



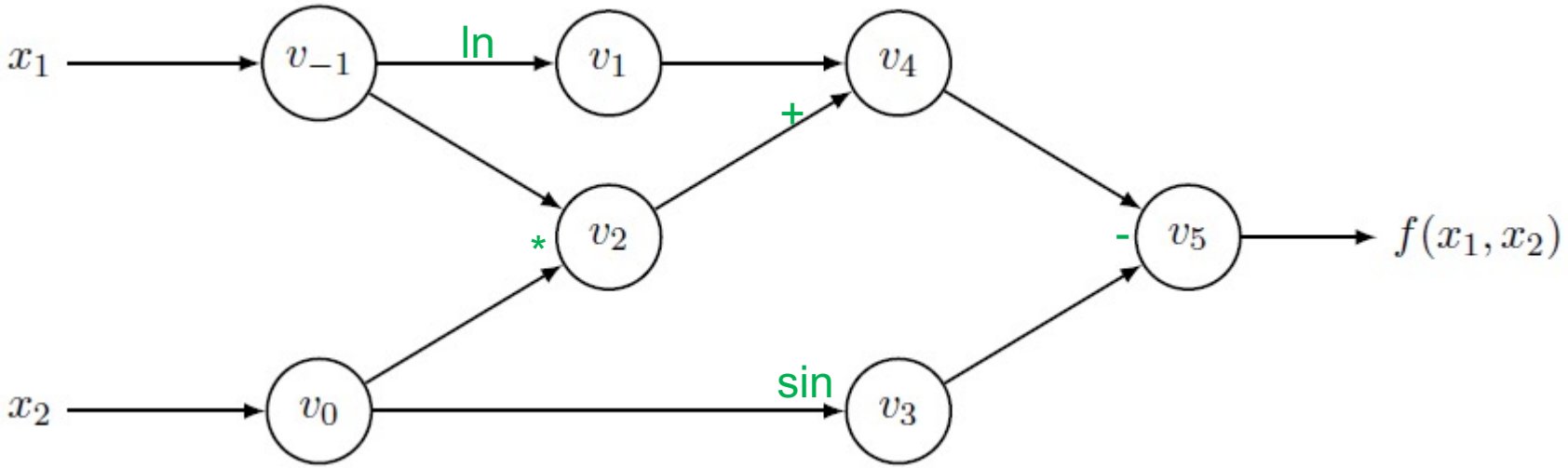
Automatic differentiation (AD)

Forward Primal Trace

v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Go forward in the graph doing the calculations of all variables for specific values of x_1 & x_2

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \quad (x_1, x_2) = (2, 5)$$



AD

Dot over a variable represents a derivative
Over what? Below, it's the derivative of any "v" over x_1

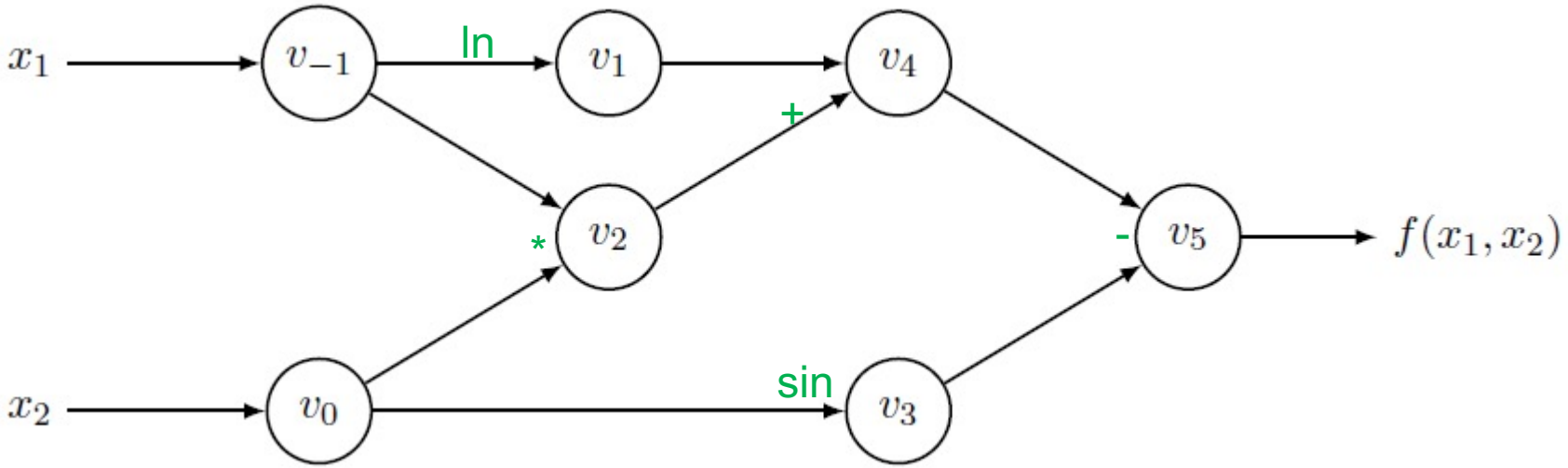
Forward Primal Trace

v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace

$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= 5.5$

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \quad (x_1, x_2) = (2, 5)$$



AD

Dot over a variable represents a derivative
Over what? Below, over x_1

Forward Primal Trace

v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace

$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \quad (x_1, x_2) = (2, 5)$$

We do not get derivatives in a form of mathematical formulas

We just get the value of the derivative value ($dy / dx_1 = 5.5$) for specific x_1,x_2

That's what we need for gradient descent - a specific value (or values) that tell us how to change weights and biases in the net!

AD

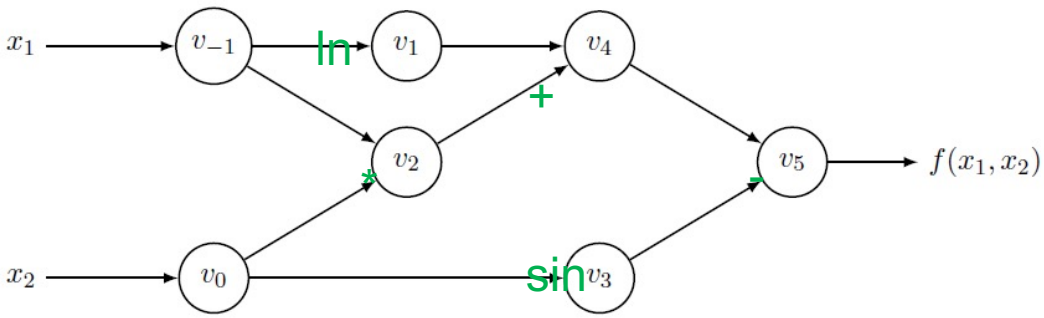
Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= 5.5$

Above, we had forward differentiation

Calculate d v_i / d x for increasing i

Until we get to d v_5 / d x = d y / d x



AD

Above, we had forward differentiation

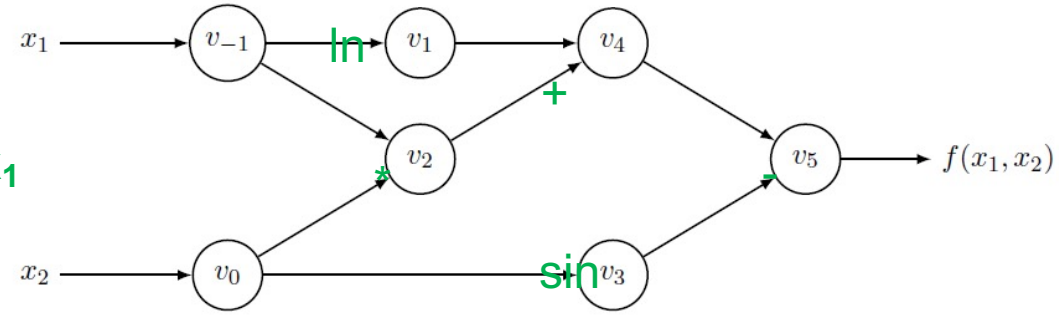
Calculate d v_i / d x for increasing i

Until we get to d v_5 / d x = **d y / d x_1**

There is an alternative way, closer to backpropagation

Calculate d y / d v_i for decreasing i

Until we get to d y / d v_0 = **d y / d x_1**



Forward Primal Trace

$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace — alternative way (faster)

$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$		$= 1$

Automated Differentiation

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import torch as torch;

# create a tensor variable, this is a constant parameter, we do not need gradient
w.r.t. to it
minimum_w=torch.tensor(np.array([1.0,3.0]),requires_grad=False);

#define some function using pytorch operations (note torch. instead of np.)
# this function is f(w)=||w-minimum||^2, and so has minimum at minimum_w, i.e. at
vector [1.0,3.0]
# it is a convex function so has one minimum, no other local minima
def f(w):
    shiftedW=w-minimum_w;
    return torch.sum(torch.mul(shiftedW,shiftedW));
```

Automated Differentiation

```
#define starting value of W for gradient descent
#here, W is a 2D vector
initialW=np.random.rand(2)

#create a PyTorch tensor variable for w.
# we will be optimizing over w, finding its best value using gradient descent (df /
dw) so we need gradient enabled
w = torch.tensor(initialW,requires_grad=True);
```

Automated Differentiation

```
# create a tensor variable, this is a constant parameter, we do not need gradient
w.r.t. to it
minimum_w=torch.tensor(np.array([1.0,3.0]),requires_grad=False);

#define some function using pytorch operations (note torch. instead of np.)
# this function is f(w)=||w-minimum||^2, and so has minimum at minimum_w, i.e. at
vector [1.0,3.0]
# it is a convex function so has one minimum, no other local minima
def f(w):
    shiftedW=w-minimum_w;
    return torch.sum(torch.mul(shiftedW,shiftedW));
```

```
#define starting value of W for gradient descent
#here, W is a 2D vector
initialW=np.random.rand(2)

#create a PyTorch tensor variable for w.
# we will be optimizing over w, finding its best value using gradient descent (df /
dw) so we need gradient enabled
w = torch.tensor(initialW,requires_grad=True);
```

```
# this will do gradient descent (fancy, adaptive learning rate version called Adam)
for us
optimizer = torch.optim.Adam([w],lr=0.001)

for i in range(10000):
    # clear previous gradient calculations
    optimizer.zero_grad();
    # calculate f based on current value
    z=f(w);
    if (i % 100 == 0 ):
        print("Iter: ",i," w: ",w.data.cpu().numpy()," f(w): ",z.item())
    # calculate gradient of f w.r.t. w
    z.backward();
    # use the gradient to change w
    optimizer.step();
```

AD for learning y=ax+b

- Another example, closer to deep learning
- We have training set of pairs (x,y)
- We want to predict y based on x
- It's a very simple relationship: y=ax+b (no noise)
- We start from random a,b, and let gradient descent improve them, until they converge to the unknown, true a,b

AD for learning y=ax+b

```
import torch as torch;
import numpy as np

np.random.seed(42)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

#generate simple set of training (x,y) pairs for y=ax+b
true_a = 2.7
true_b = -0.3
x_train = np.random.rand(100,1);
y_train = true_a * x_train + true_b;

# create tensor variables for data, we do not need gradient w.r.t. to them
t_x_train=torch.tensor(x_train,requires_grad=False,device=device);
t_y_train=torch.tensor(y_train,requires_grad=False,device=device);

# create a PyTorch tensor variable for a and b.
# we will be optimizing over a, b using gradient descent, so we need gradient enabled
init_std_dev = 0.01;
random_initial_a=init_std_dev*np.random.randn(1,1)
a = torch.tensor(random_initial_a,requires_grad=True,device=device);
b = torch.zeros((1,1),requires_grad=True,device=device);
```

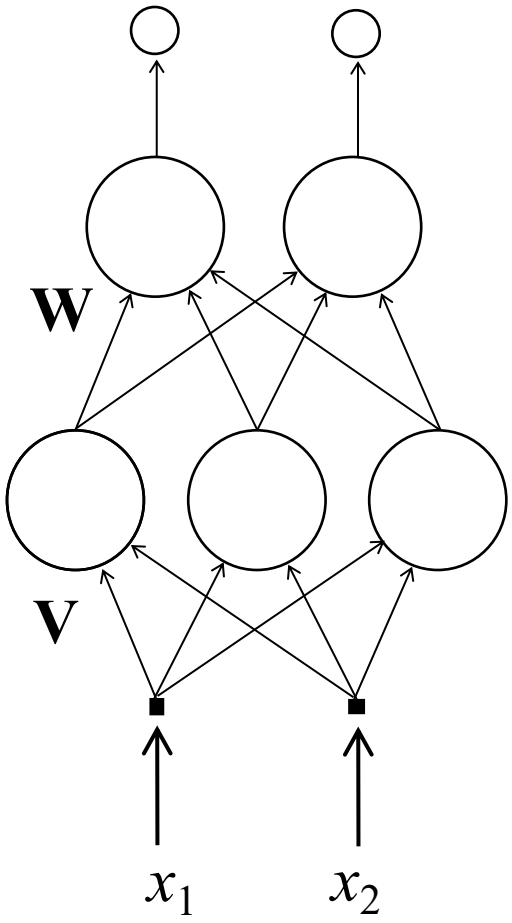
```
# the optimizer will do gradient descent for us
learning_rate = 0.01;
optimizer = torch.optim.SGD([a,b],lr=learning_rate)
#optimizer = torch.optim.Adam([a,b],lr=learning_rate)

n_epochs = 50000;
for i in range(n_epochs):
    # clear previous gradient calculations
    optimizer.zero_grad();

    # calculate model predictions
    linear_predictions = torch.matmul(t_x_train,a)+b

    #calculate loss/risk
    prediction_error = t_y_train-linear_predictions
    squared_error = torch.pow(prediction_error,2)
    risk = torch.mean(squared_error);

    #calculate gradients of risk w.r.t. a,b and propagate them back
    risk.backward();

    # use the gradient to change a, b
    optimizer.step();

    print(i,risk.item(),a.detach().cpu().numpy(),b.detach().cpu().numpy())
```

Summary so far

- Gradient descent can be encapsulated inside packages for automated differentiation
  - We just write what the model is, no need to worry about coding the details of the math needed to train it

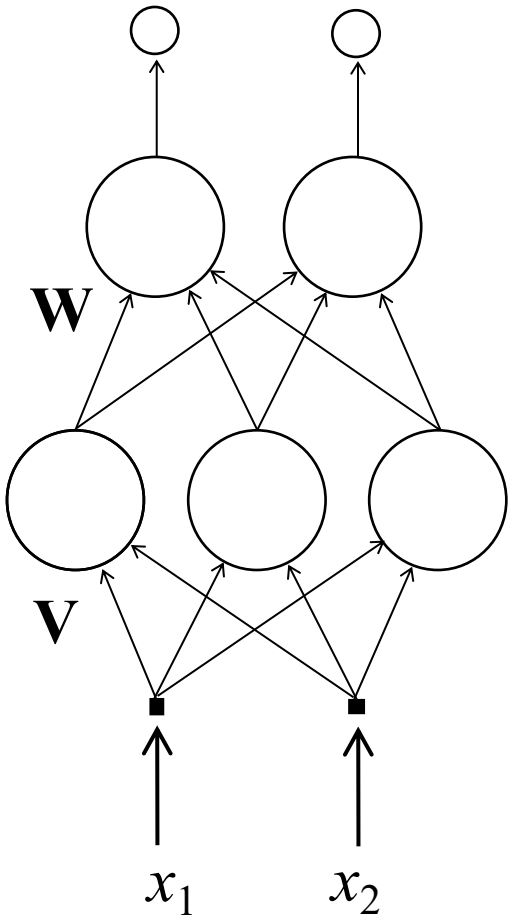
Deep Learning in a Nutshell

- Choose architecture
  - Standard approach:
    fully connected feed-forward layers
- Choose the objective function
  - Standard approach:
    mean squared error
    over sigmoid output
- Choose the optimization method
  - Standard approach:
    - Gradient descent



Design problems and solutions

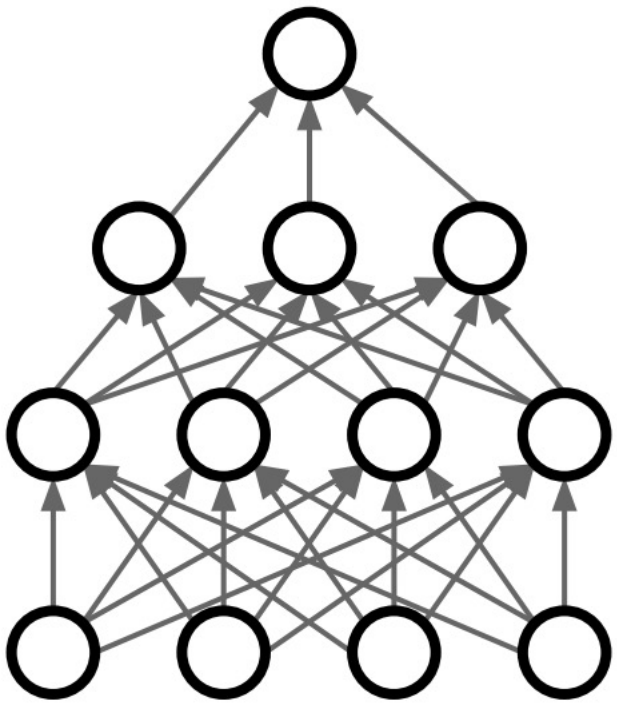
- Choose the objective function
  - Standard approach: mean squared error over sigmoid output

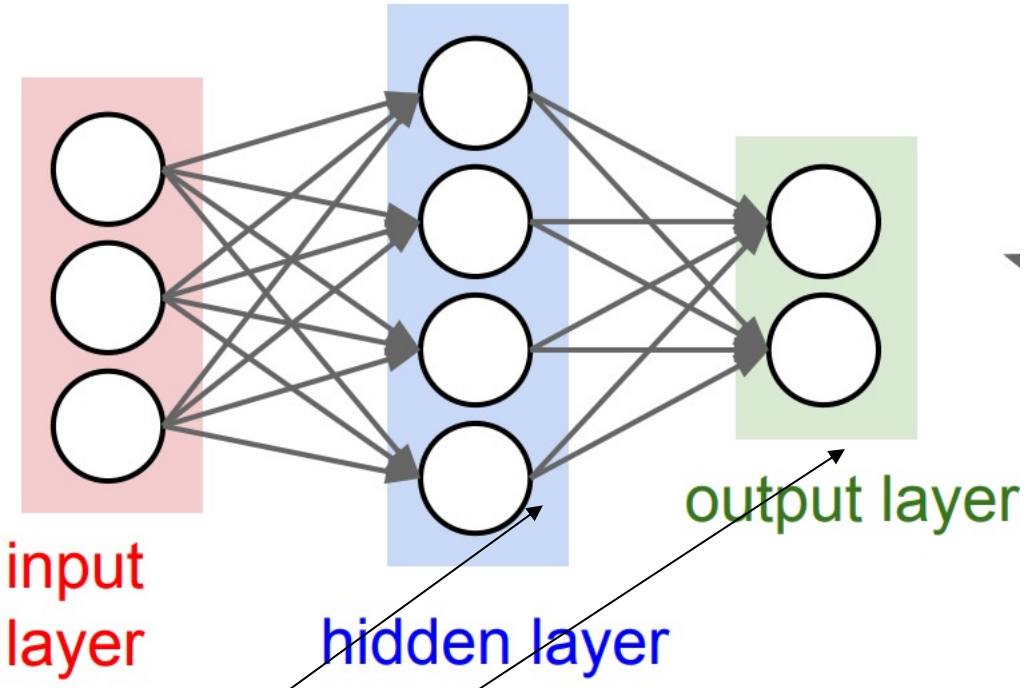


- Problems with the standard approach:
  - Vanishing gradients
- Partial Solutions:
  - Cross-entropy loss
  - Alternative Nonlinearities

Nonlinear classification

- A feed forward neural network





- Typically:
  - Multiple classes / output neurons
  - Multiple nonlinear layers: linear z=Wx followed by nonlinear y=a(z)

Mean squared error over sigmoid

- Prediction=a($w^T x$) $$a(u) = \frac{2}{1+e^{-u}} - 1$$
- Quadratic loss on sigmoid predictions:

$$\ell(h, \mathbf{z}) = \left(y - a\left(\mathbf{w}^{\dagger T}\mathbf{x}^{\dagger}\right)\right)^2 = \left(1 - ya\left(\mathbf{w}^{\dagger T}\mathbf{x}^{\dagger}\right)\right)^2$$

- Gradient-based update:

$$\mathbf{w}^{\dagger}{}_{t+1} = \mathbf{w}^{\dagger}{}_t - \frac{c}{2}\left.\frac{\partial \ell(h, \mathbf{z})}{\partial \mathbf{w}^{\dagger}}\right|_{\mathbf{w}^{\dagger}=\mathbf{w}^{\dagger}{}_t} = \mathbf{w}^{\dagger}{}_t + c\left[y - a\left(\mathbf{w}^{\dagger}{}_t^T\mathbf{x}^{\dagger}\right)\right]a'\left(\mathbf{w}^{\dagger}{}_t^T\mathbf{x}^{\dagger}\right)\mathbf{x}^{\dagger}$$

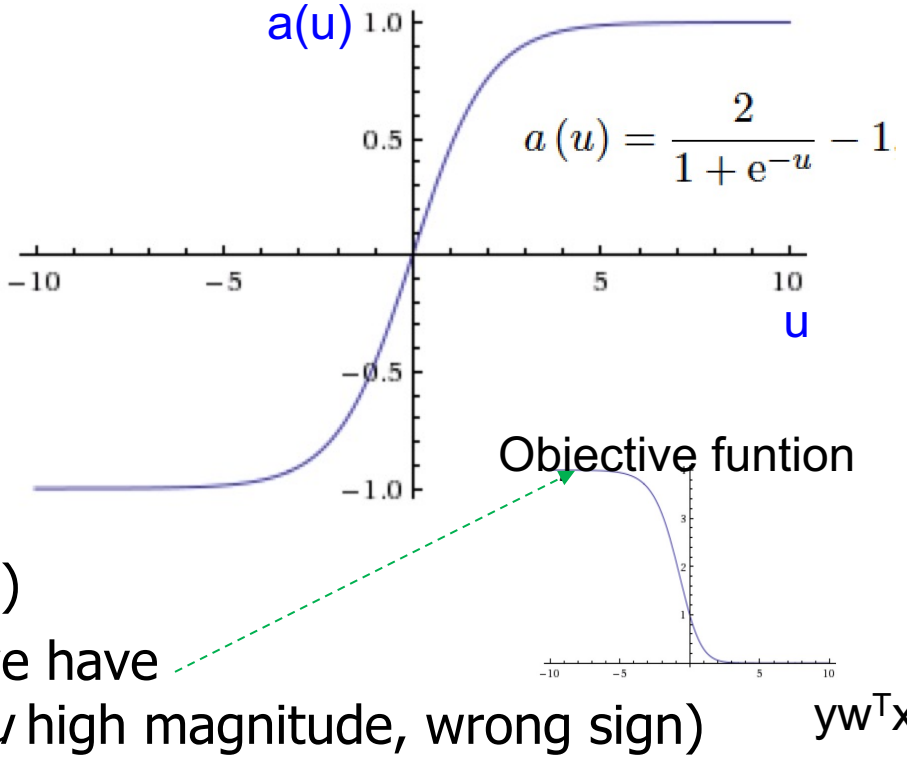
  - Note that there are two terms multiplied, y-a, and a’

Mean squared error over sigmoid

- Weight update rule:

$$\mathbf{w}^{\dagger}{}_{t+1} = \mathbf{w}^{\dagger}{}_{t} - \frac{c}{2} \left. \frac{\partial \ell\left(h, \mathbf{z}\right)}{\partial \mathbf{w}^{\dagger}} \right|_{\mathbf{w}^{\dagger}=\mathbf{w}^{\dagger}{}_{t}} = \mathbf{w}^{\dagger}{}_{t} + c\left[y - a\left(\mathbf{w}^{\dagger}{}_{t}^{T}\mathbf{x}^{\dagger}\right)\right] a'\left(\mathbf{w}^{\dagger}{}_{t}^{T}\mathbf{x}^{\dagger}\right)\mathbf{x}^{\dagger}$$

- The update depends on:
  - (y-a(u))
    - Smaller as we approach correct prediction
  - a'(u)
    - Much smaller as we approach correct prediction (*u* v.large)
    - And also very small when we have really incorrect prediction (*u* high magnitude, wrong sign)



- Very slow learning for large |u|

Mean squared error over sigmoid

$$a'(u) = a(u)(1 - a(u))$$

- Weight update:

$$\mathbf{w}^{\dagger}{}_{t+1} = \mathbf{w}^{\dagger}{}_{t} - \frac{c}{2} \frac{\partial \ell\left(h, \mathbf{z}\right)}{\partial \mathbf{w}^{\dagger}}\bigg|_{\mathbf{w}^{\dagger}=\mathbf{w}^{\dagger}{}_{t}} = \mathbf{w}^{\dagger}{}_{t} + c\left[y - a\left(\mathbf{w}^{\dagger}{}_{t}^{T}\mathbf{x}^{\dagger}\right)\right] a'\left(\mathbf{w}^{\dagger}{}_{t}^{T}\mathbf{x}^{\dagger}\right)\mathbf{x}^{\dagger}$$

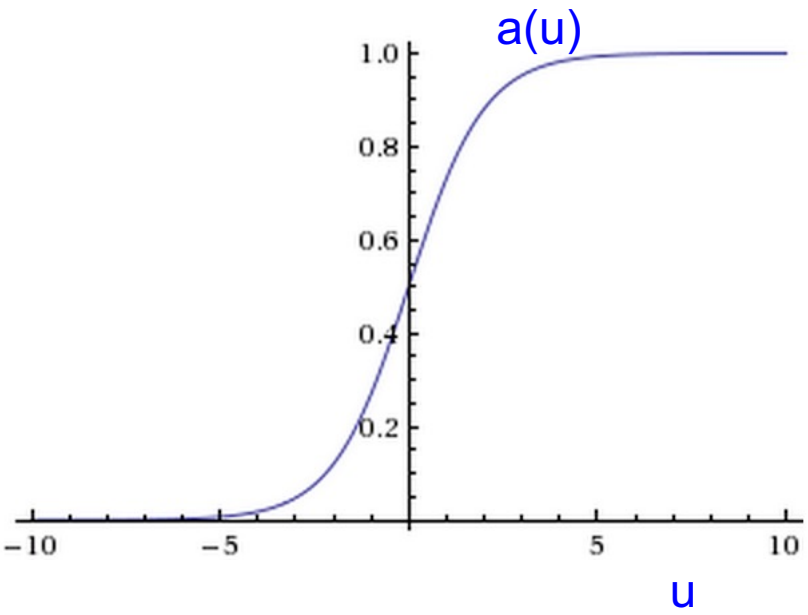
- Let's get rid of the a′ term and have:

$$-\frac{\partial \ell(a,y)}{\partial w_i} = (y - a)x_i$$

- We can find loss function to have equality above

- The solution is to have loss of this form:

$$\ell(a, y) = -[y \log(a) + (1 - y) \log(1 - a)]$$



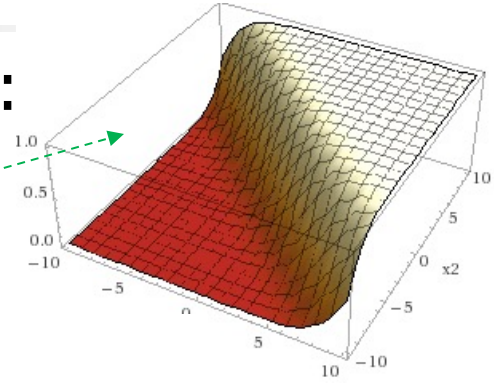
Cross-entropy loss

- Our derived "better" loss on result of $a(w^T x)$:

$$\ell(a, y) = -[y \log(a) + (1 - y) \log(1 - a)]$$

- We can rewrite it as:

$$P_{\text{data}}(y = 1|x) = y \qquad P_{\text{data}}(y = 0|x) = 1 - y$$
$$P_{\text{model}}(y = 1|x) = a(w^T x) \qquad P_{\text{model}}(y = 0|x) = 1 - a(w^T x)$$



$$\ell(a, y) = -\sum_{i=\{0,1\}} P_{\text{data}}(y = i|x) \log P_{\text{model}}(y = i|x)$$

- P_{data} is a distribution over y={0,1}, and takes values {0,1}
  - a sample has 100% probability of being class 1, or 100% of class 0
- For P_{model}, this is the probability that the model/network produces for the sample.

Cross-entropy loss

- For P_{model}, this is the probability that the model/network produces for the sample. We see a term $-\log_2 p_i = \log_2 1/p_i$
- What is the meaning of this term
  - Surprise $I(P_A)$ associated with seeing an event A that is supposed to happen with probability P_A:
    - Certainty = no surprise: $P_A=1 \Rightarrow I(P_A)=0$
    - Lower probability => higher surprise: $P_A < P_B \Rightarrow I(P_A) > I(P_B)$
  - Surprise $I(P_{A+B})$ associated with seeing events A and B that are supposed to happen with probabilities P_A, P_B
    - $I(P_{A+B}) \Rightarrow I(P_A)+I(P_B)$ if events A, B are independent
      - $P_{A+B}=P_AP_B \Rightarrow I(P_AP_B)=I(P_A)+I(P_B)$
    - What form can the function I(P) take?
      - Is 0 for argument of 1
      - Is decreasing
      - Turns multiplication into addition

Cross-entropy loss

- Surprise $I(P_A)$ associated with seeing an event A that is supposed to happen with probability P_A:
  - $P_A=1 \Rightarrow I(P_A)=0$
- Lower probability => higher surprise
  - $P_A< P_B \Rightarrow I(P_A) > I(P_B)$
- Surprise $I(P_{A+B})$ associated with seeing events A and B that are supposed to happen with probabilities P_A, P_B
  - $I(P_{A+B}) \Rightarrow I(P_A)+I(P_B)$ if events A, B are independent
    - $P_{A+B}=P_AP_B \Rightarrow I(P_AP_B)=I(P_A)+I(P_B)$
  - What form can the function I(P) take?
    - I(p)=-log(p)
      - -log(1)=0
      - -log(ab)=-log(a) + -log(b)
      - -log is decreasing, since log is increasing

Cross-entropy loss

- Interpretation of $-\log_2 p_i = \log_2 1/p_i$

Let's use *log-base-2* here.
In the code, we use *ln*

- Measure of surprise: $I(p)=-\log_2(p)$

- We see event with probability p_i – how surprised are we?
  - Sun rising in the morning: $p_i = 1$, surprise = 0
  - Heads after a fair coin toss: $p_i = 1/2$, surprise = 1 bit
  - Two heads from two fair coins: $p_i = 1/2*1/2$, surprise = 2 bits
  - "3" on a 4-sided dice: $p_i = 1/4$, surprise = 2 bits
  - Once-in-a-hundred-years heat-wave: $p_i = 1/100$, surprise = 6.64 bits
  - Win on a "1:1,048,576 chance" lottery, $p_i = 2^{-20}$, surprise = 20 bits
  - Sun NOT rising in the morning: $p_i = 0$, surprise = ∞ bits

Cross-entropy loss

0 log0 =0
log is log2

- **Entropy** of distribution *p*:

$$H(p) = E_p[\log \tfrac{1}{p}] = E_p[-\log p] = -\sum_i p_i \log p_i$$

  - **Expected surprise** from samples from distr. p
    - $p_0=p_1=1/2$ => high surprise H=1,
      $p_0=1$ no surprise, $p_1=0$ infinite surprise (but times 0), H=0

- Example: hotter than median summer
  - heat wave $p_1=1/2$ (surprise=1bit) or not $p_0=1/2$ (surprise=1bit)
  - We observe 128 summers, we get 64 heat waves
    - H(p)=H(p,p)=-1/2 log2(1/2) - 1/2 log2(1/2) = 1
  - Highest possible entropy = expected surprise for "this-or-that"
- Example: once-in-a-(CS)-century heat wave
  - heat wave $q_1=1/128$ (surprise=7bits) or not $q_0=127/128$ (surprise=0.011 bits)
  - We observe 128 summers, we get 1 heat wave
    - H(q)=H(q,q)=-1/128 log2(1/128) - 127/128 log2(127/128) = 0.0659
  - Much lower entropy = expected surprise

Cross-entropy loss

0 log0 =0
log is log2

- Entropy of distribution *p*:

$$H(p) = E_p[\log \tfrac{1}{p}] = E_p[-\log p] = -\sum_i p_i \log p_i$$

  - Amount of surprise from samples from distr. P

- Cross-entropy of *p* and *q*:

$$H(p,q) = E_p[-\log q] = -\sum_i p_i \log q_i$$

  - Surprise when assuming samples came from distrib. q (use q to calc. surprise of each even)
    but they actually come from distrib. p (use p to calc. the mean/expect.)

- Example: climate change
  - heat wave q_1=1/128 (surprise=7bits) or not q_0=127/128 (surprise=0.011 bits)
  - We expect that if we see 128 summers, we get 1 heat wave
    - H(q)=H(q,q)=-1/128 log2(1/128) - 127/128 log2(127/128) = 0.0659
  - We observe 128 summers, we get 64 heat waves
    - So we have observed p_1=1/2 p_0=1/2
    - Our expectation was that it came from q_1=1/128 q_0=127/128
    - H(p,q) = -1/2 log2(1/128) – 1/2 log2(127/128) = **3.5** >> 0.0659 =H(q)

Cross-entropy loss

0 log0 =0
log is log2

- Entropy of distribution *p*:

$$H(p) = E_p[\log \tfrac{1}{p}] = E_p[-\log p] = -\sum_i p_i \log p_i$$

  - Amount of surprise from samples from distr. p
    - E.g. $p_0=p_1=1/2$ high surprise H=1, $p_0=1$ no surprise H=0
- Cross-entropy of *p* and *q*:

$$H(p,q) = E_p[-\log q] = -\sum_i p_i \log q_i$$

  - Surprise when assuming samples came from distrib. q but they actually come from distrib. p

- Cross-entropy as a loss:
  assume model is correct, surprise we have from seeing training data

Cross-entropy loss

0 log0 =0
log is log2

- Cross-entropy of *p* and *q*:

$$H(p,q) = E_p[-\log q] = -\sum_i p_i \log q_i$$

- Loss = $H(P_{data}, P_{model})$

- We observe data where class is certain (e.g. *P(class 1) for given x* = 1 or 0)

- Model says data came from distrib. $a(w^T x)$ = *P(class 1) for given x* is

- How surprised we are by data (y) if model y=h(x) was true?

Cross-entropy loss

- Can we extend it to more than 2 classes?
- Easy:

$$\ell(a, y) = -\sum_{i=\{0,1\}} P_{\text{data}}(y = i|x) \log P_{\text{model}}(y = i|x)$$

i=1 to num_classes

$a(w_1^T x) = P(\text{class } 1 \mid x)$
$a(w_2^T x) = P(\text{class } 2 \mid x)$
$a(w_3^T x) = P(\text{class } 3 \mid x)$
$a(w_4^T x) = P(\text{class } 4 \mid x)$
…

- But: $a(w_1^T x)$, $a(w_2^T x)$, $a(w_3^T x)$, … must be a probability distribution over classes
  - >=0
  - Add up to 1

Soft-max

$$\ell(a, y) = -\sum_{i=\{0,1\}} P_{\text{data}}(y = i|x) \log P_{\text{model}}(y = i|x)$$

i=1 to num_classes

$a(w_1^T x) = P(\text{class } 1 \mid x)$
$a(w_2^T x) = P(\text{class } 2 \mid x)$
$a(w_3^T x) = P(\text{class } 3 \mid x)$
$a(w_4^T x) = P(\text{class } 4 \mid x)$
…

- But: $a(w_1^T x)$, $a(w_2^T x)$, $a(w_3^T x)$, …
  must be a probability distribution over classes

- Softmax:
  - $a(w_1^T x) = \exp(w_1^T x) / \Sigma_k \exp(w_k^T x)$
    - exp(…) makes values > 0
    - Then we normalize to add up to 1

Multi-class predictions

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

- One output neuron per class: P_{model}
  - K classes = K *logits*: $w_1^T x$, $w_2^T x$,…,$w_k^T x$
  - Then we do softmax on the logits to get probabilities
  - output j represents the probability $P_{model}(y=j|x)$
- True class: encode as one-hot vector P_{data}
  - P_{data}(y=wrong class|x)=0
  - P_{data}(y=true class|x)=1
- Loss = $H(P_{data}, P_{model})$
- P_{model}=[0.1, 0.8, 0.06, 0.04]
  - P_{data}=[0, 1, 0, 0]
  - Loss = - [0*log(0.1) + 1*log(0.8) + 0*log(0.06) +0*log(0.04)]

Multi-class predictions

- Softmax vs sigmoid

WolframAlpha

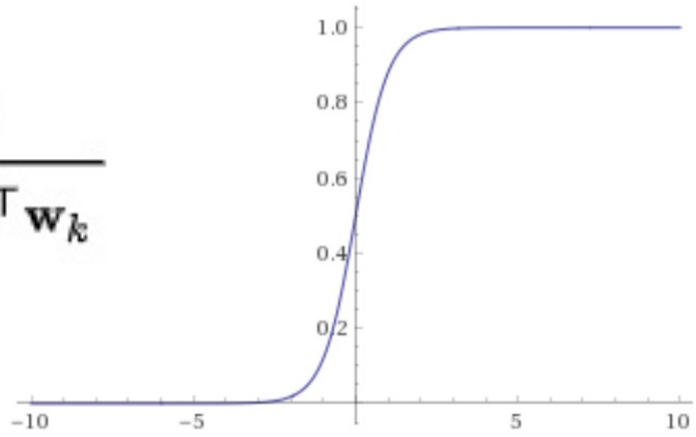
exp(x) / (exp(x)+exp(-x)) plot x from -10 to 10

Extended Keyboard Upload

Input interpretation:

plot	$\frac{\exp(x)}{\exp(x)+\exp(-x)}$	$x = -10$ to 10

Plot:



$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Deep neural networks

- Sigmoid has the problem of "vanishing gradient"
- Softmax is a generalization of sigmoid, it has the same problem

- But:

- Cross-Entropy Loss=$H(P_{data}, P_{model})=-\Sigma P_{data}\log(P_{model})$

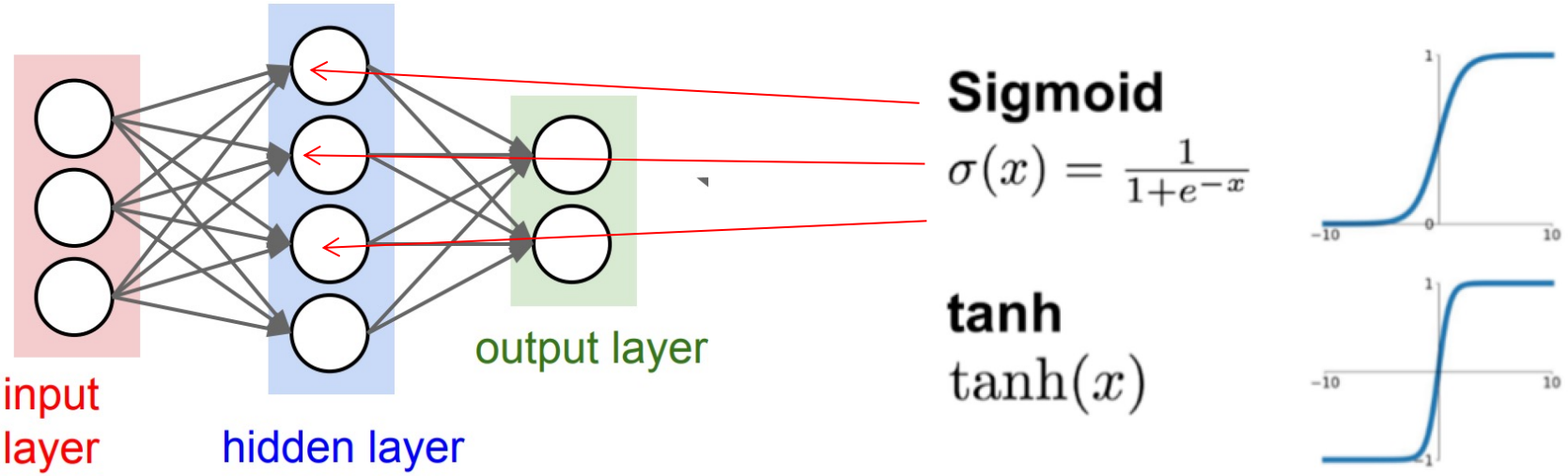
$$P(y=j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

- We will be taking $\log(\exp(x^T w))$
  - Reducing the problem

Deep neural networks

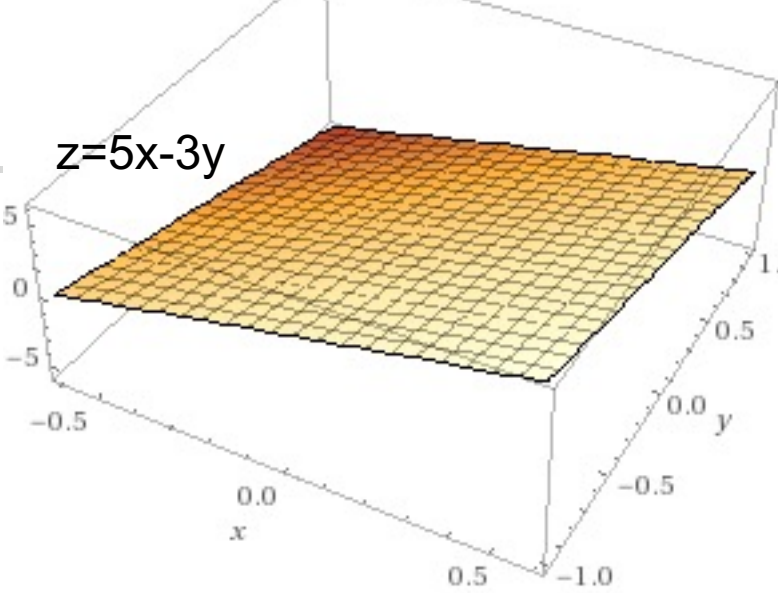
- With Cross-Entropy after softmax, the last layer does not lead to flat regions (vanishing gradients)
  - unlike Mean Square Error loss

- We do not have that option for hidden layers
  - A sigmoid as activation function in hidden layers may get saturated: "vanishing gradients"

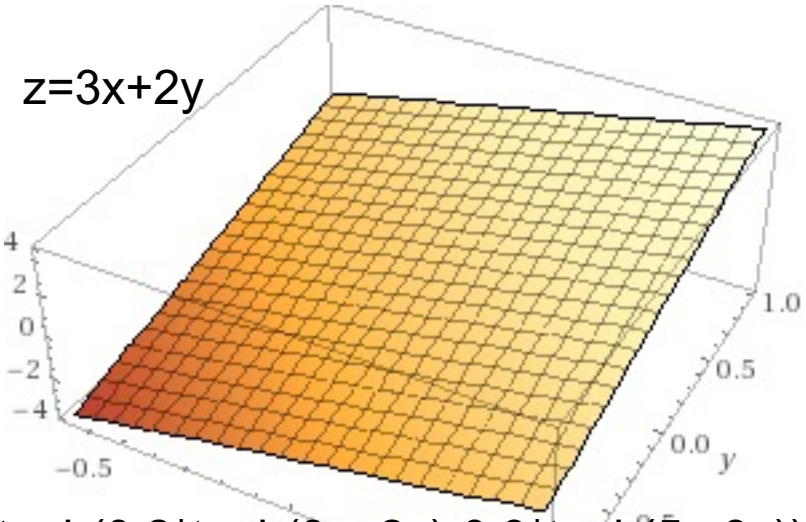


Vanishing gradients

z=5x-3y

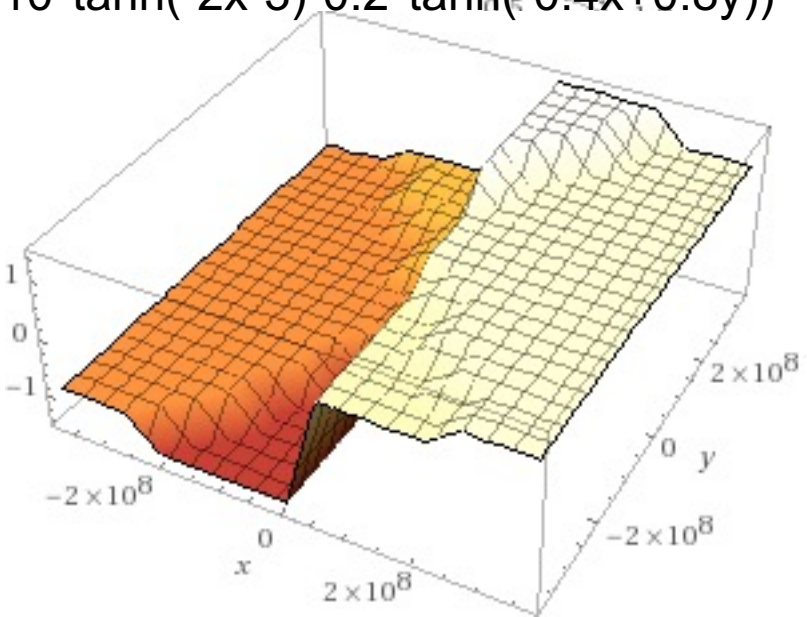


z=3x+2y

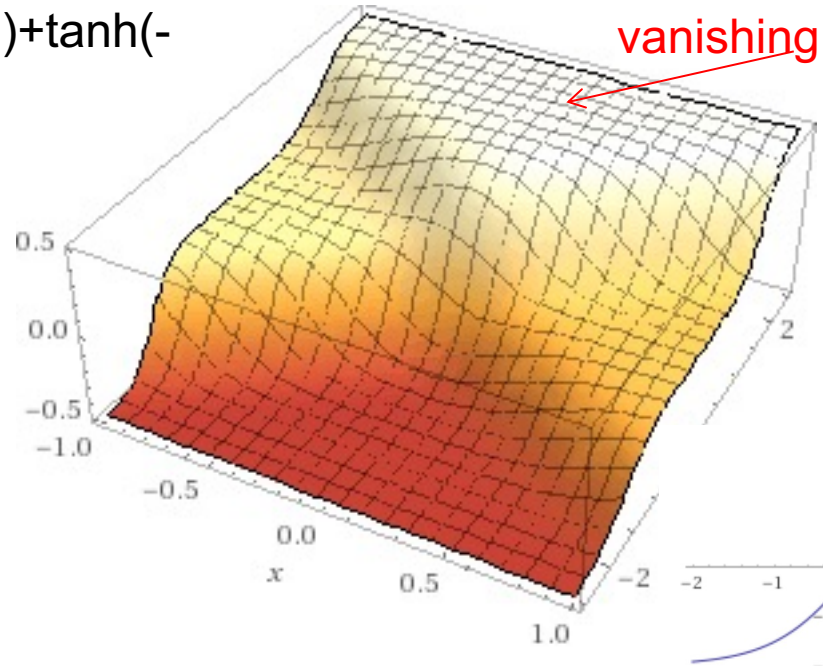


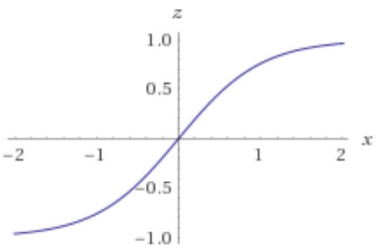
z=0.2*tanh(3x+2y)-0.3*tanh(5x-3y)

z=tanh(0.2*tanh(3x+2y)-0.3*tanh(5x-3y))+tanh(-10*tanh(-2x-3)-0.2*tanh(-0.4x+0.8y))



vanishing gradients





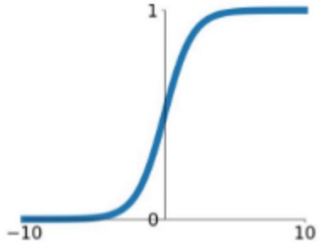
Modern activation functions

- Rectified Linear Unit (and similar):

Activation Functions

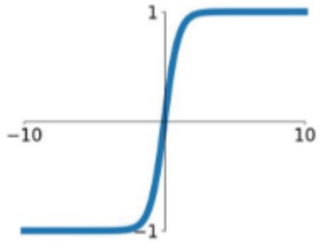
Sigmoid

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



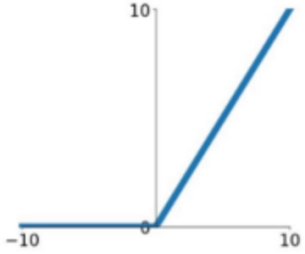
tanh

$$\tanh(x)$$



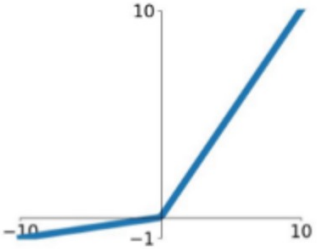
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

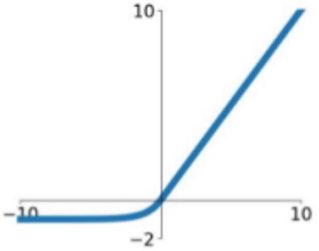


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

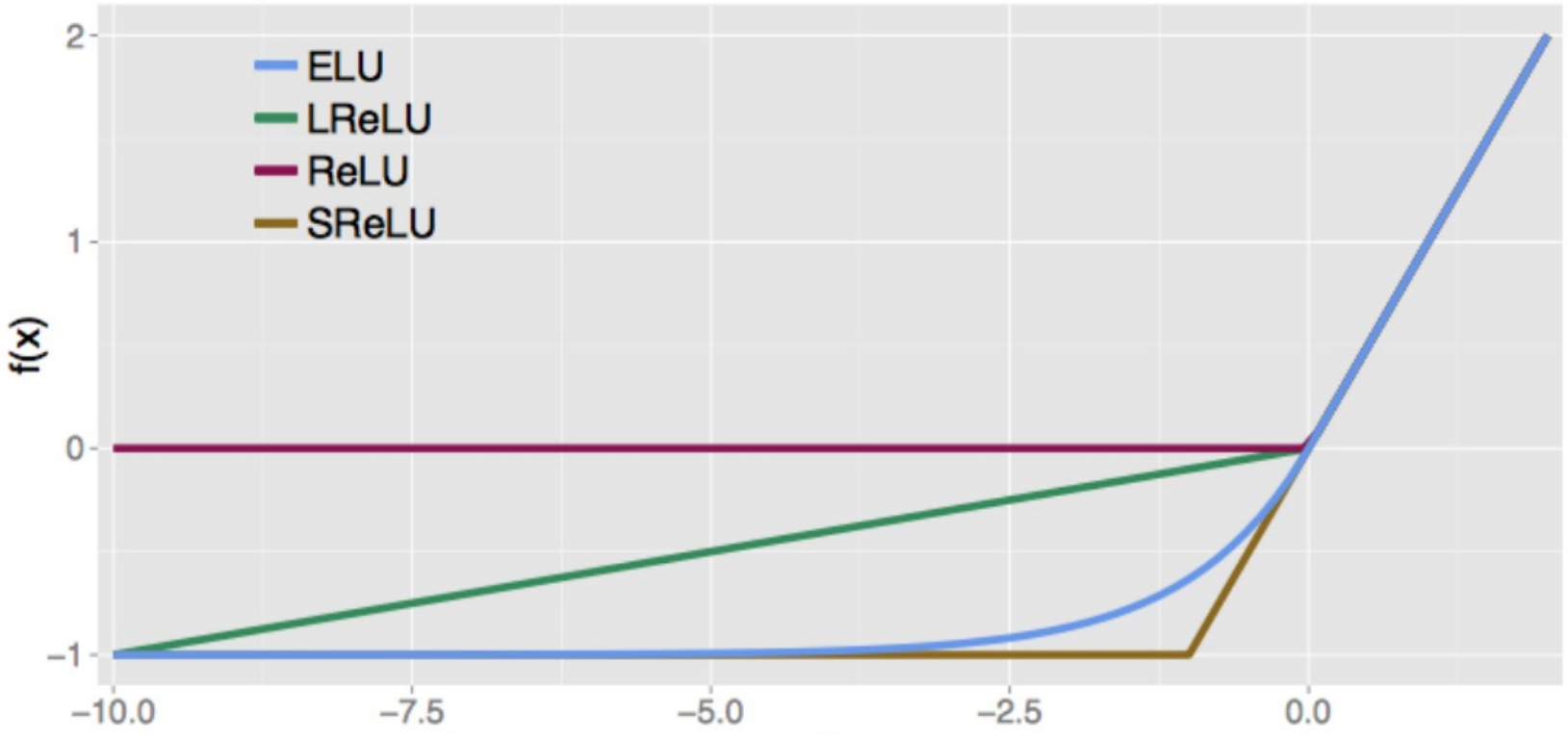
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Deep neural networks

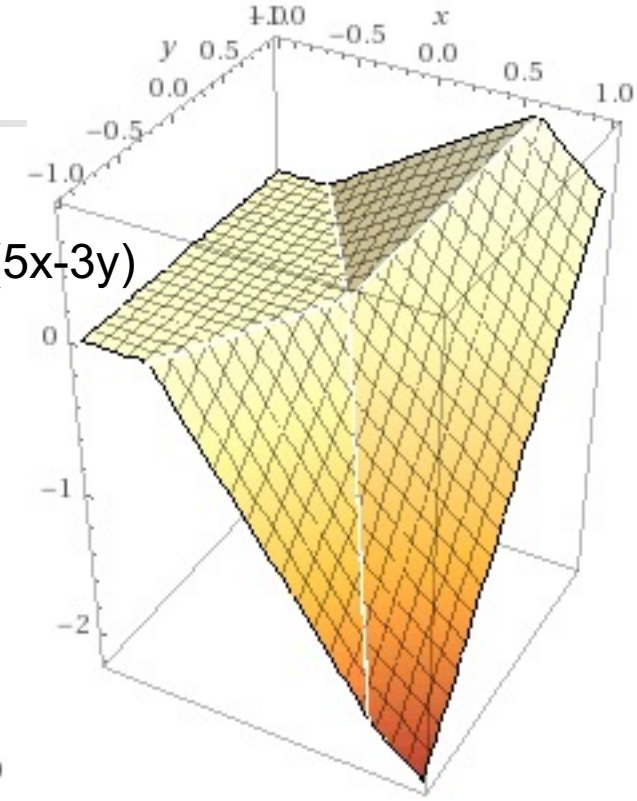
- Rectified Linear Unit (and similar):

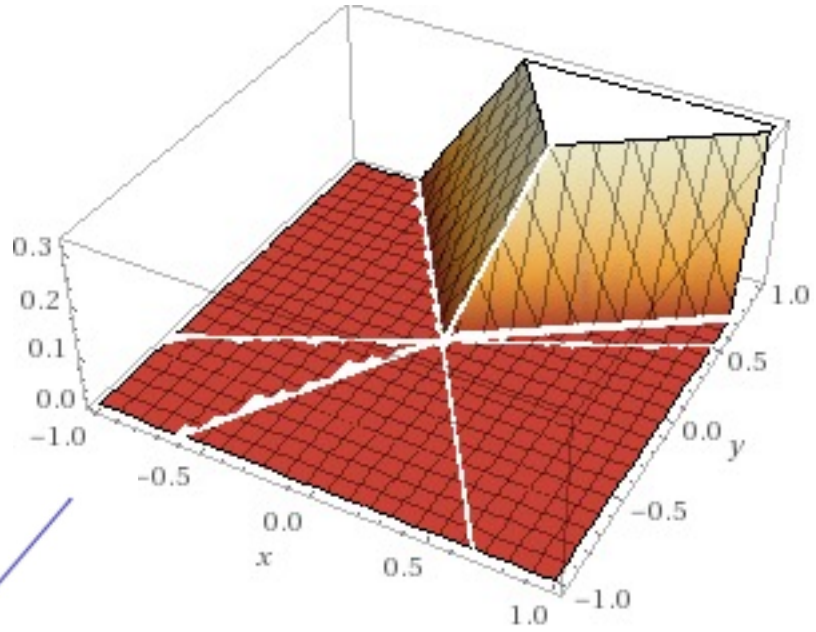


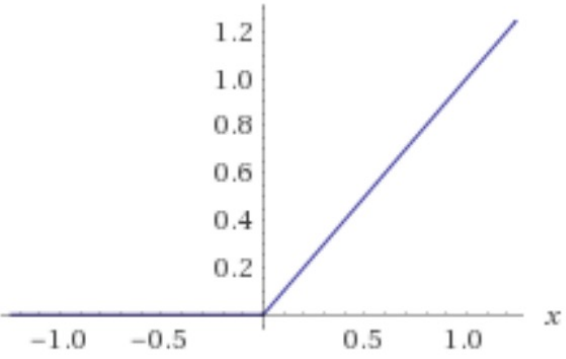
Nonlinearity

- ReLU is capable of nonlinear classification

0.2*ReLU(3x+2y)-0.3*ReLU(5x-3y)



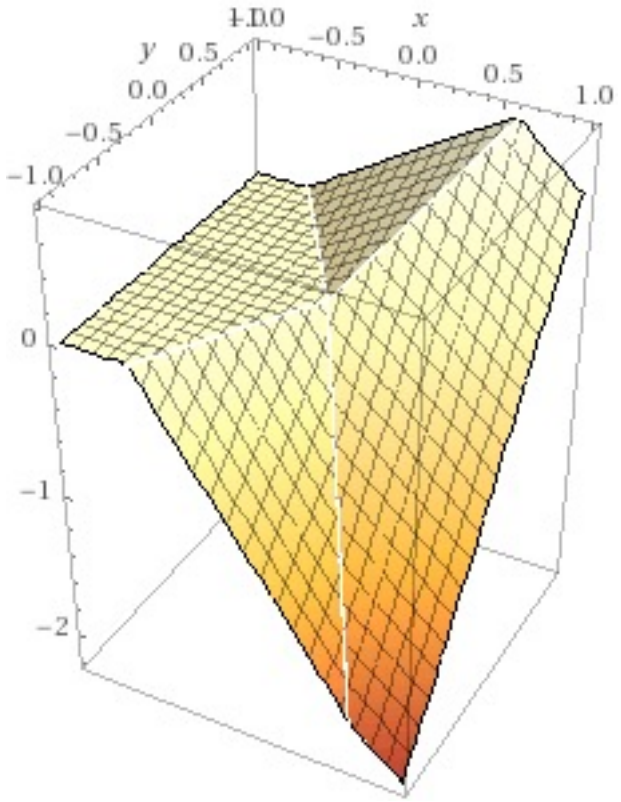




ReLU(0.2*ReLU(3x+2y)-0.3*ReLU(5x-3y))+
ReLU(-1*ReLU(-2x-3)-0.2*ReLU(-0.4x+0.8y))

Deep neural networks

- We change the activation function:
  - ReLU
- These still have flat regions ("dying ReLU problem"), but only on one side, it is easier to manage



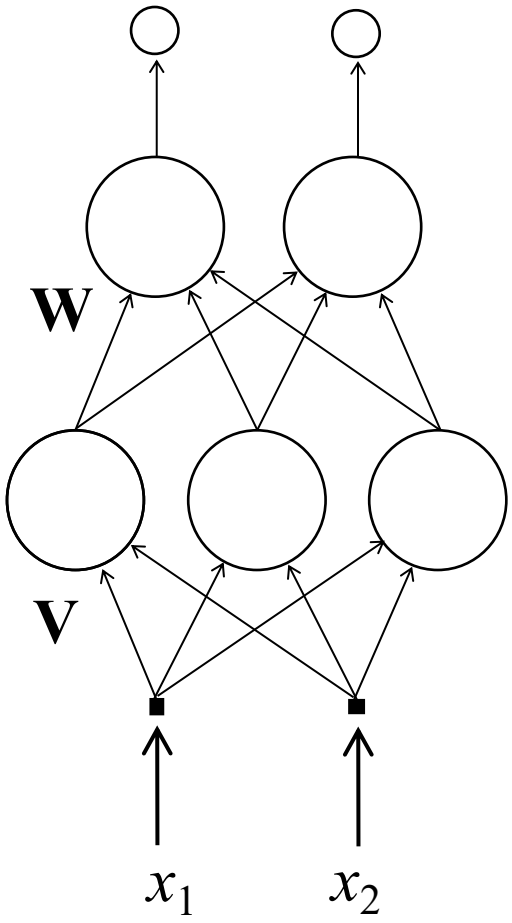
Summary so far

- Gradient descent can be encapsulated inside packages for automated differentiation
  - We just write what the model is, no need to worry about coding the details of the math needed to train it
- MSE loss may cause problems, use Cross Entropy loss
- Sigmoid activation may cause problems, use ReLU (or ELU, or GELU, or …) instead

Design problems and solutions

- Choose architecture
  - Standard approach:
    fully connected feed-forward layers

- Problems with the standard approach:
  - Huge number of weights
    -> overtraining
- Partial solution:
  - **Specialized architectures**
    e.g. convolutional networks

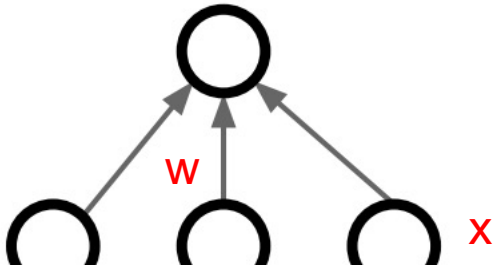


Convolutional networks

Linear model:

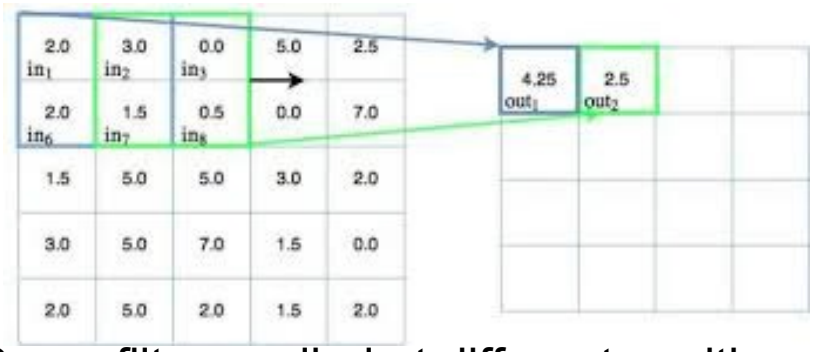
$h(x) = wx+b$,

we have a number of pre-determined features x_i

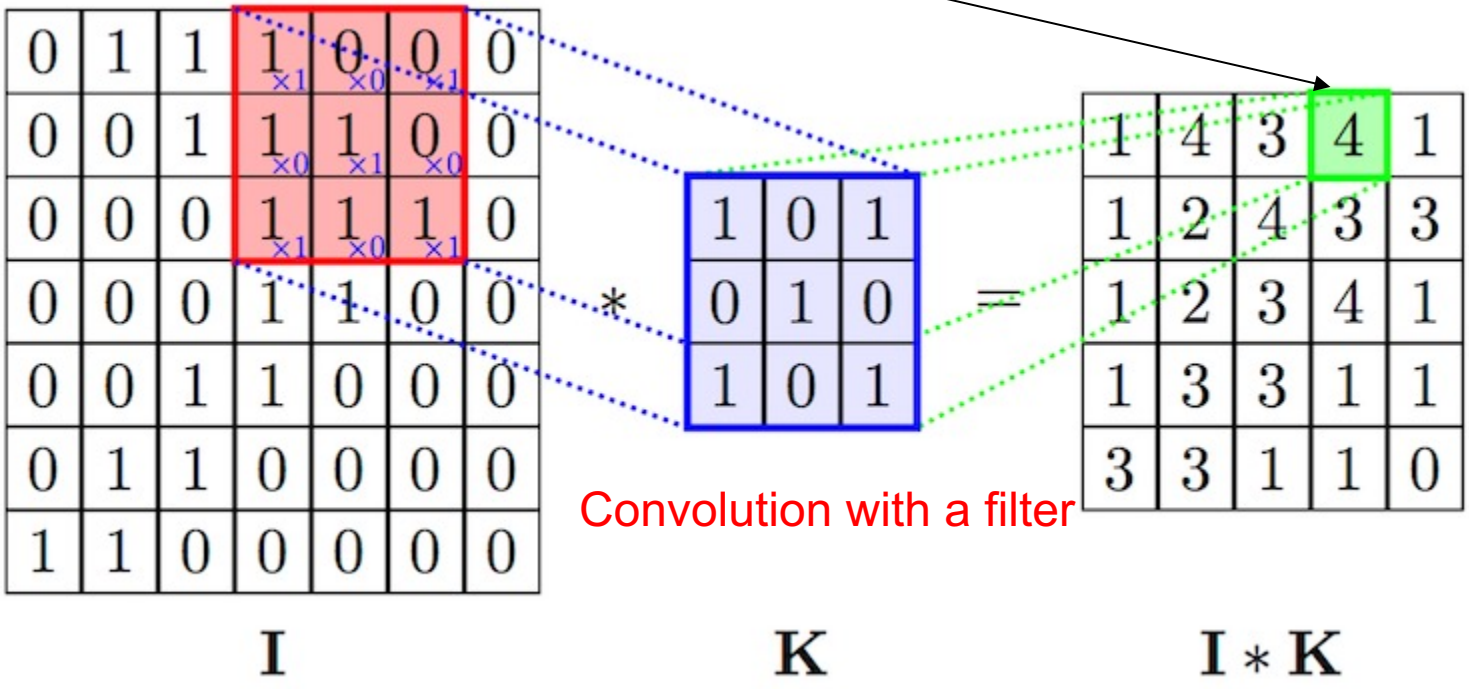


Convolutional networks

- We can try to pre-construct some "new features" and use linear classifier on those
- $h(x)=\sum_k w_k \text{ new\_features}_k(x)$



Same filter applied at different positions



Convolution with a filter

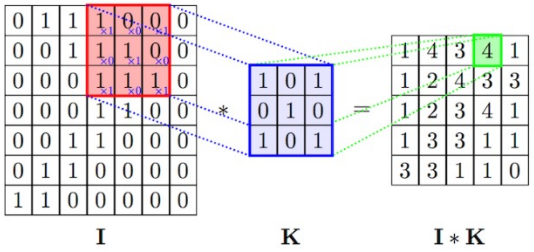
Convolutional networks

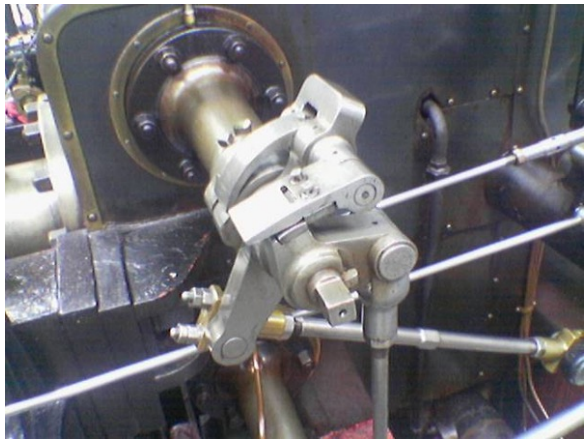
- We can try to pre-construct some "new features" and use linear classifier in those
- $h(x)=\sum_j w_j$ "new features"

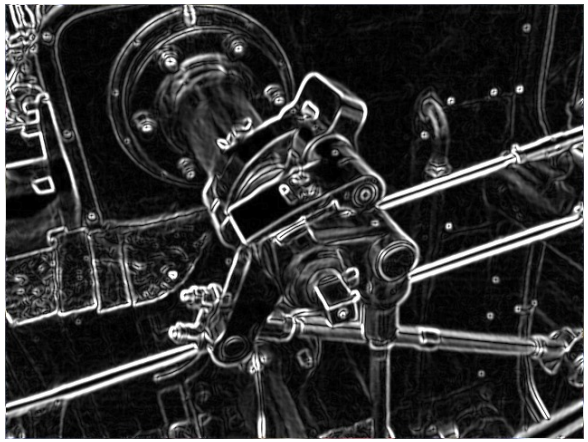
-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Basically, some clever "photoshop" operation



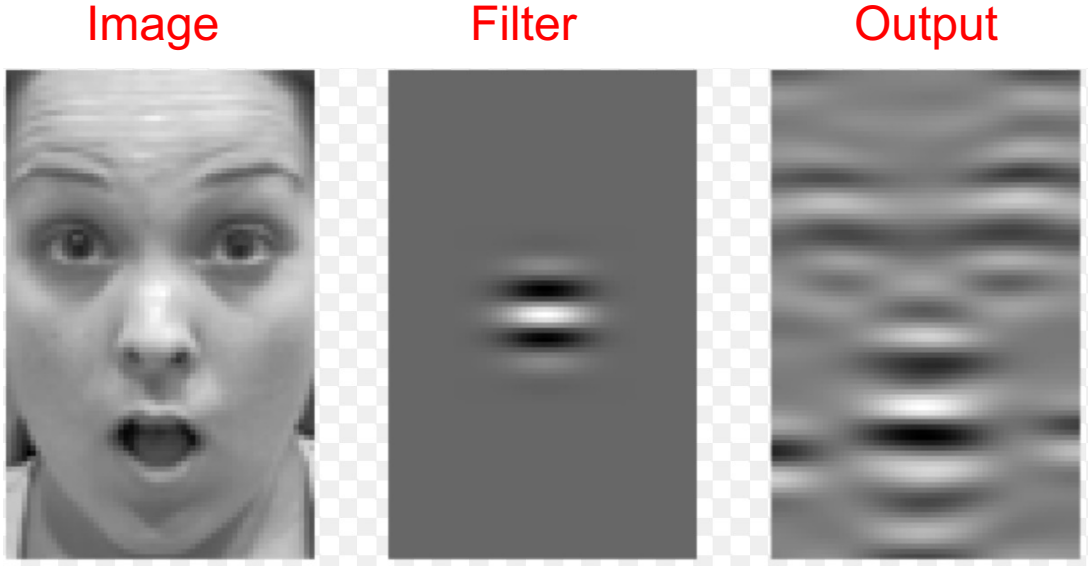




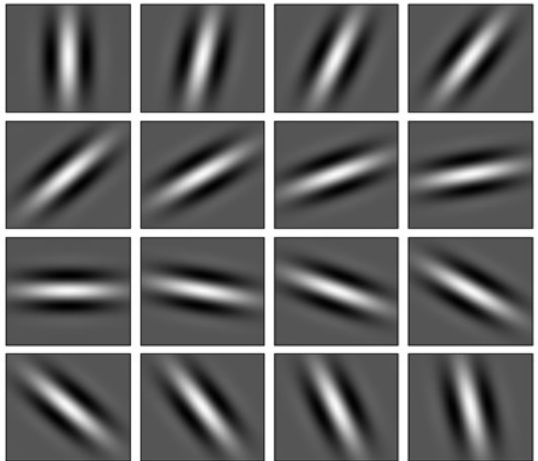
Convolutional networks

- We can try to pre-construct some "new features" and use linear classifier in those
- $h(x)=\sum_j w_j$ "new features"

Basically, some clever "photoshop" operation

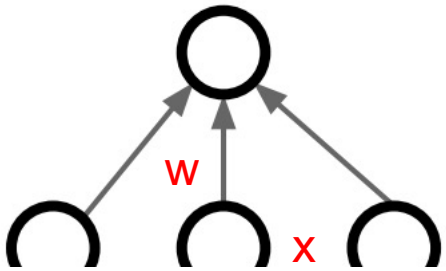


- Drawback: We have to do a good job designing the features (e.g. Gabor filters)

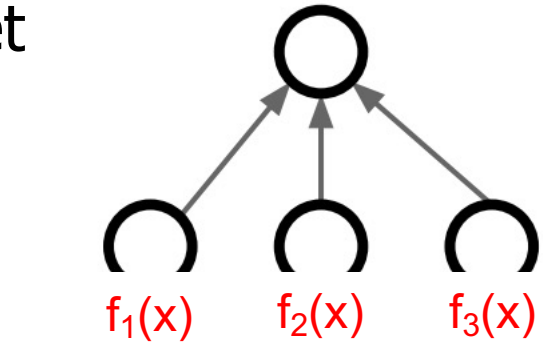


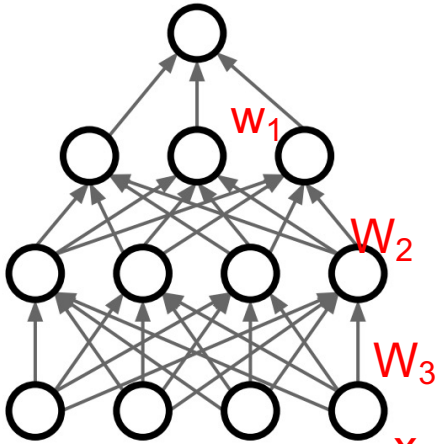
Convolutional networks

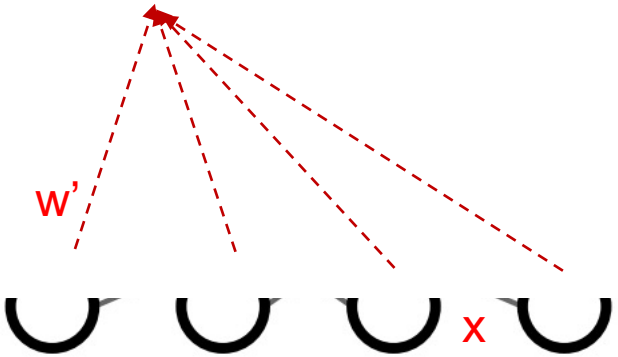
- $h(x)=\sum_j w_j$ "new features"



- Instead of figuring out how to design "new features"
- We can try learning them from the dataset
  - Specify some parameters describing the filters, and learn them, just as we learn weights
  - Lots and lots of weights => overtraining

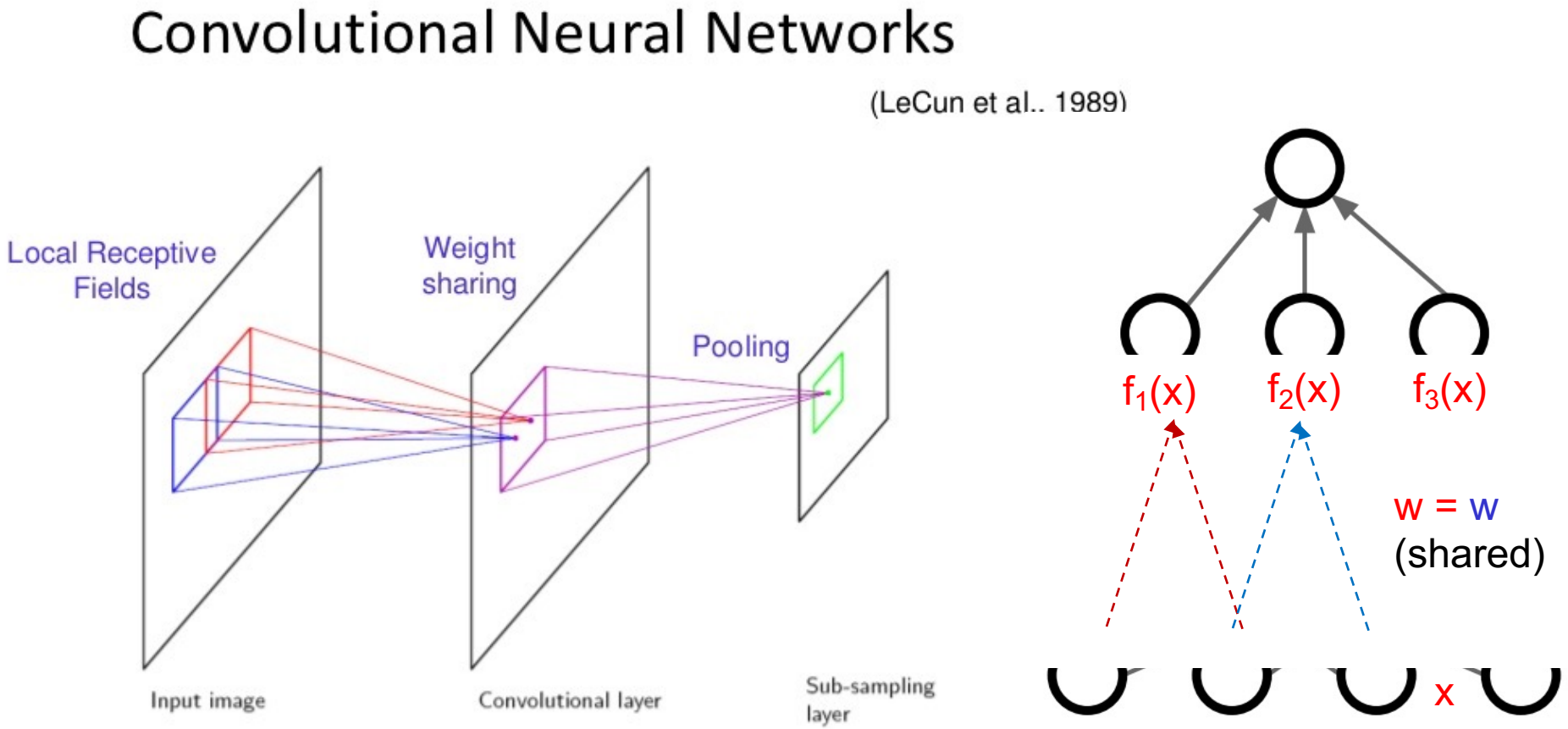






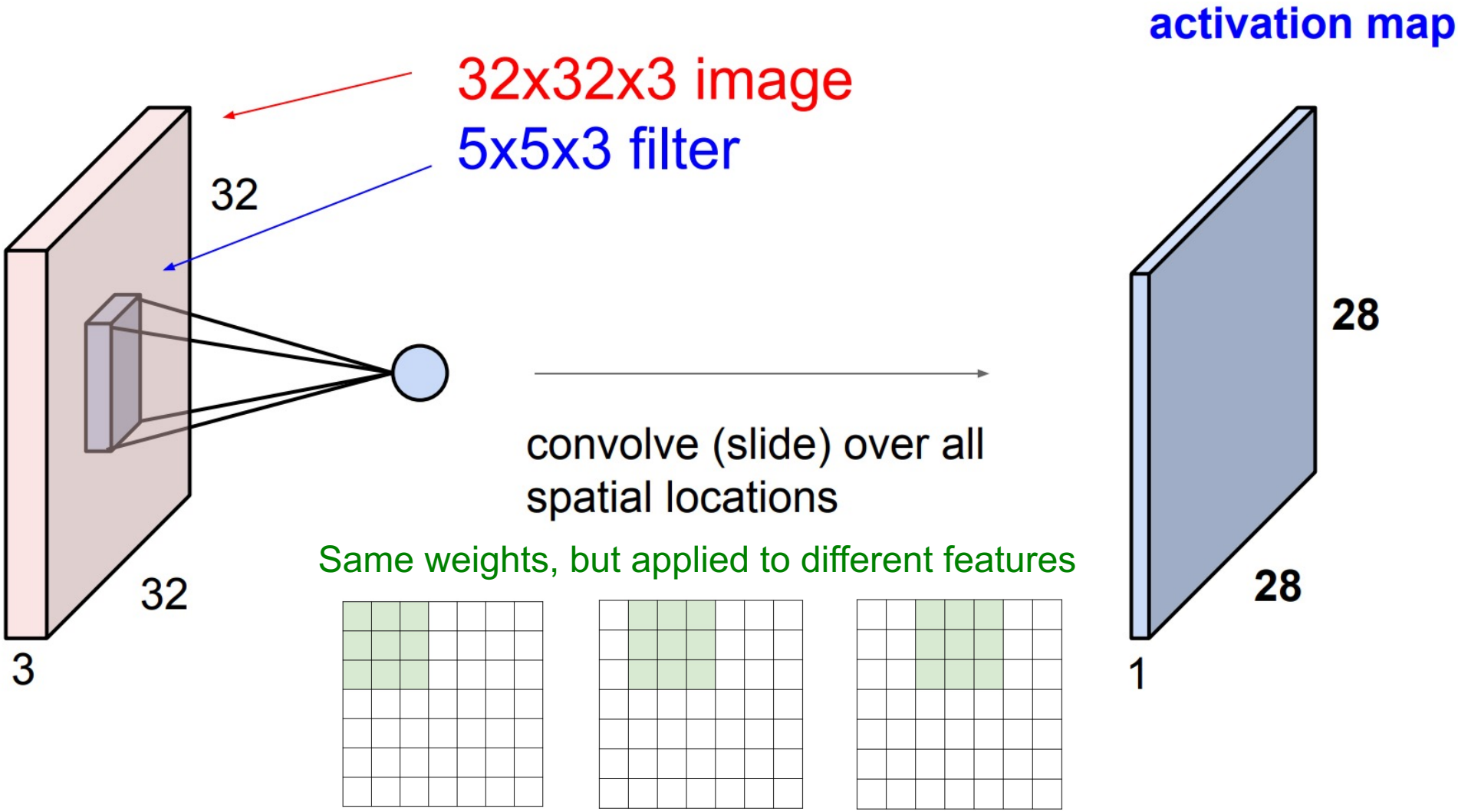
Convolutional networks

- Solution: Weight sharing
  - Red and blue weights are the same
  - We reduce the number of parameters significantly



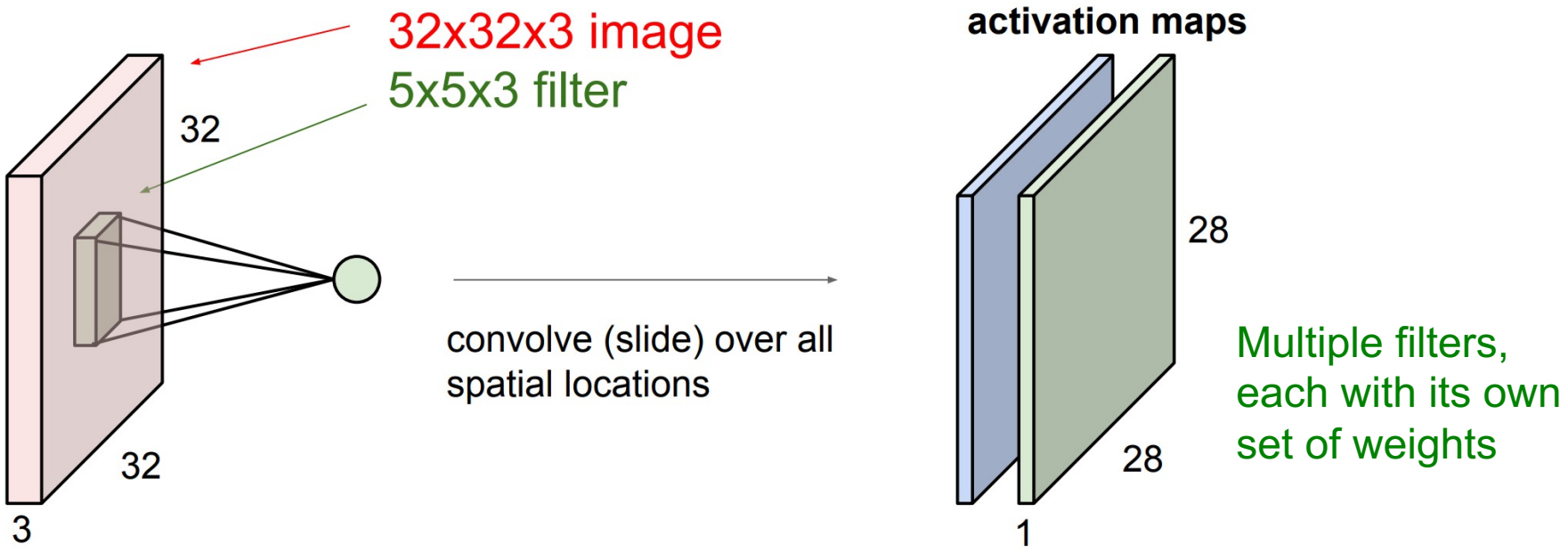
Convolutional networks

- Weight sharing



Convolutional networks

- Reducing the number of "features" for next layer



Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

Max-pooling

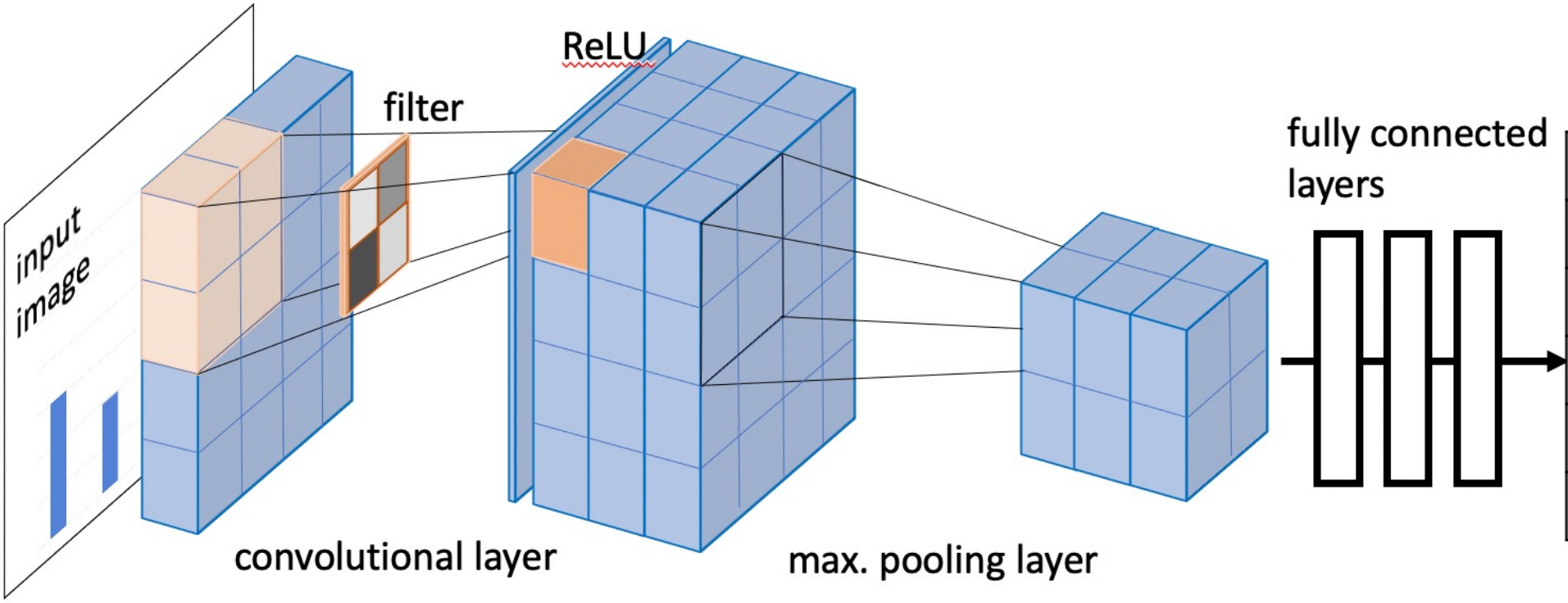
max pool with 2x2 filters and stride 2

6	8
3	4

Pooling in ConvNets

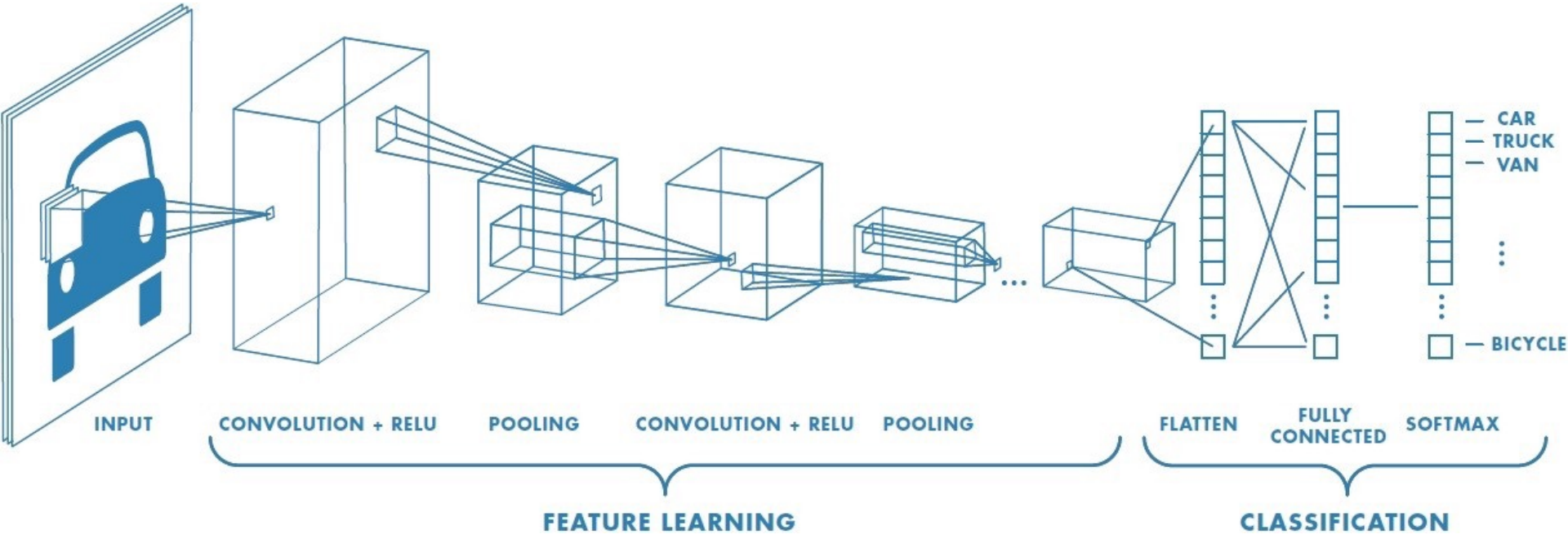
ConvNet:

downsampling via maxpooling



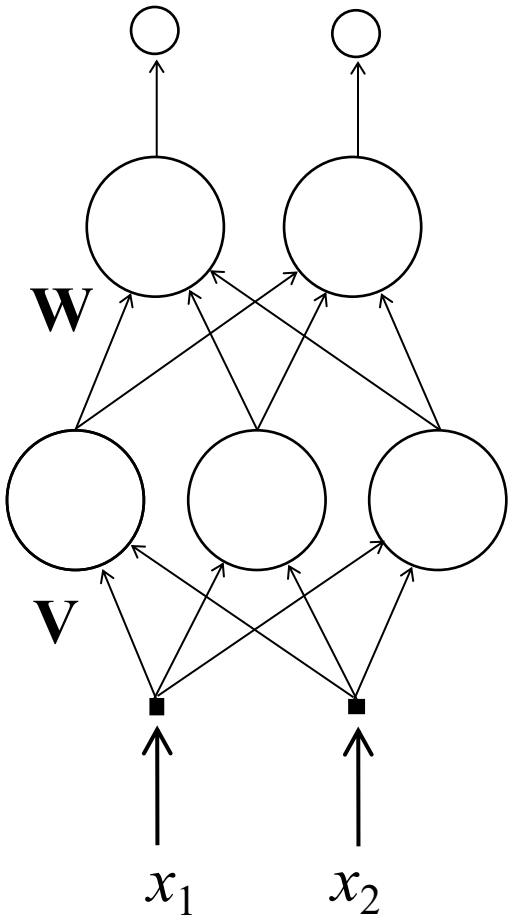
Typical architecture of CNNs

- Multiple layers of convolution + pooling, followed by dense/fully-connected layer



Design problems and solutions

- Choose architecture
  - Deep ConvNet
- Problems:
  - Vanishing gradients: passing the training signal to earlier layers
- Partial solution:
  - **Skip connections / residual layers**



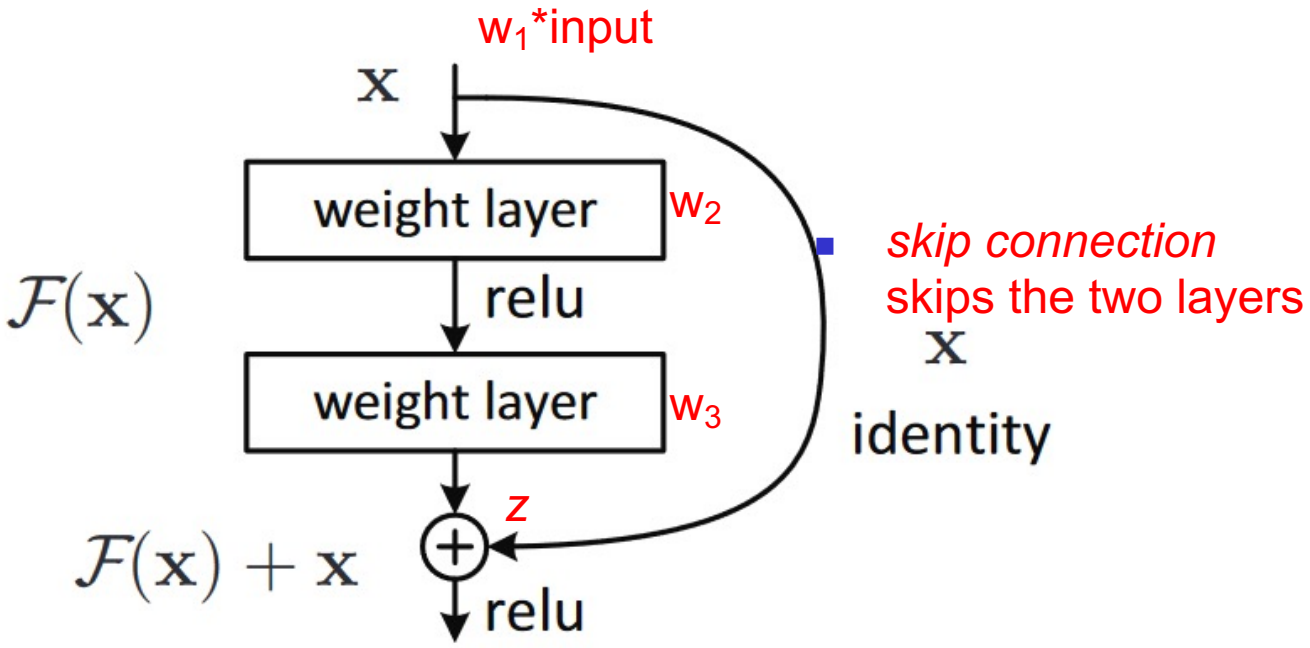
Skip connections

- A neural network with ReLU activation is essentially:
  - $Y=ReLU(\ W_3\ ReLU(\ W_2\ ReLU(\ W_1X\ )\ )\ )$
- We see terms like
  $z=w_{3ij}*w_{2kl}*w_{1mn}*input$
- The derivative of z over w_{1mn} is
  $w_{3ij}*\ w_{2kl}\ *input$
- If w_{3ij} is small
  the gradient "signal" telling w_{1mn} in which way to change does not reach w_{1mn}

Skip connections

- $z = w_{3ij} * w_{2kl} * w_{1mn} * \text{features}$
  - If w_{3ij} or w_{2kl} is small
    the gradient "signal" telling w_{1mn} in which way to change
    does not reach w_{1mn}

- Solution: link w_{1mn} to *z* directly – via a *skip connection*

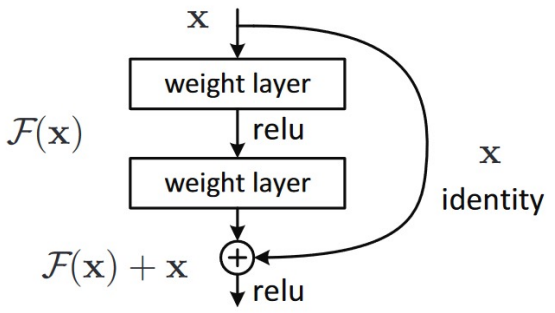


Skip connections

- Solution: link w_{1mn} to z directly – via a skip connection
- Without skip connection:
  - We see terms like $z=w_{3ij}*w_{2kl}*w_{1mn}*input$
  - The derivative of z over w_{1mn} is $w_{3ij}* w_{2kl} *input$
- With skip connection:
  - Connect $w_{1mn}*x$ directly to z, in addition to going through w_3 and w_2
  - We have $z=w_{3ij}*w_{2kl}*w_{1mn}*input + w_{1mn}*input$

    $z=(w_{3ij}*w_{2kl}+1)*w_{1mn}*input$
  - The derivative of z over w_{1mn} is

    $w_{3ij}* w_{2kl} *input + input$

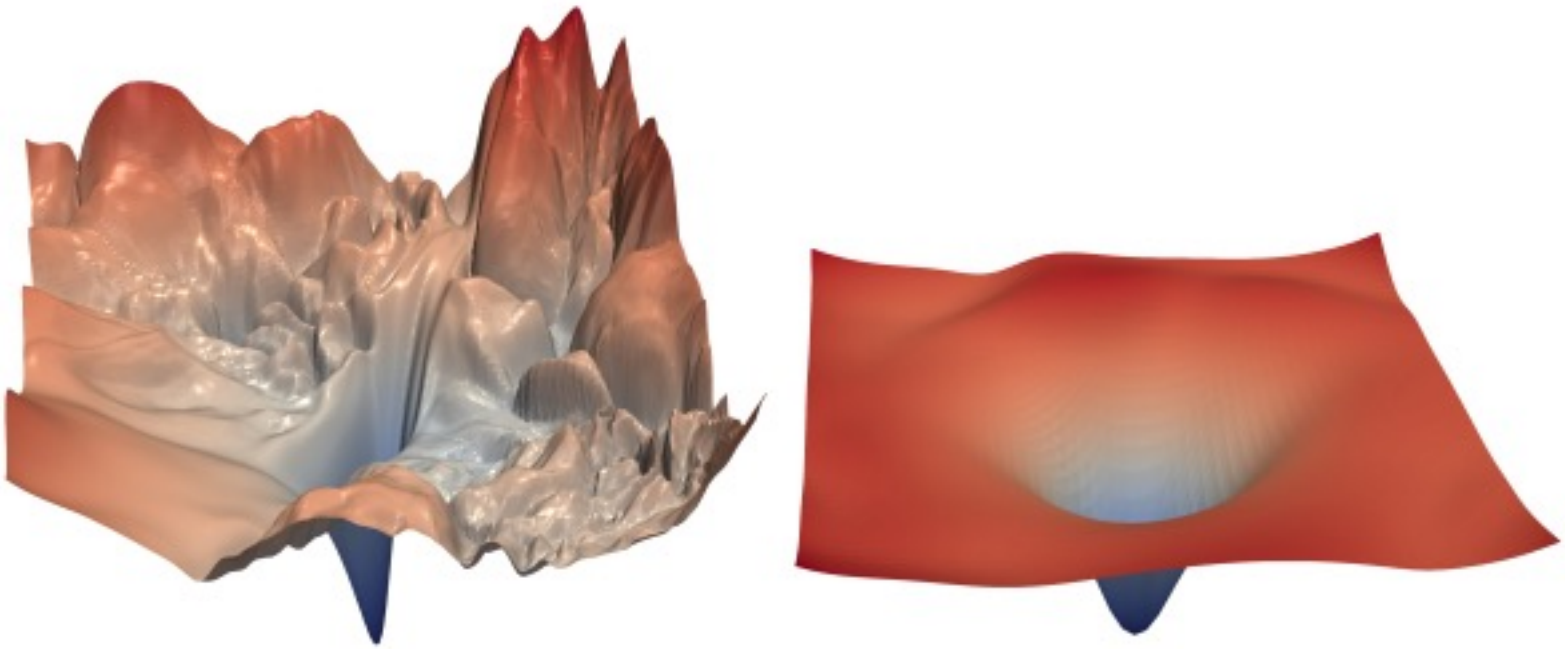


ResNet

- Adding skip connections makes the loss landscape smoother

$$z=(w_{3ij}*w_{2kl}+1)*w_{1mn}*x$$

less reliance just on multiplication

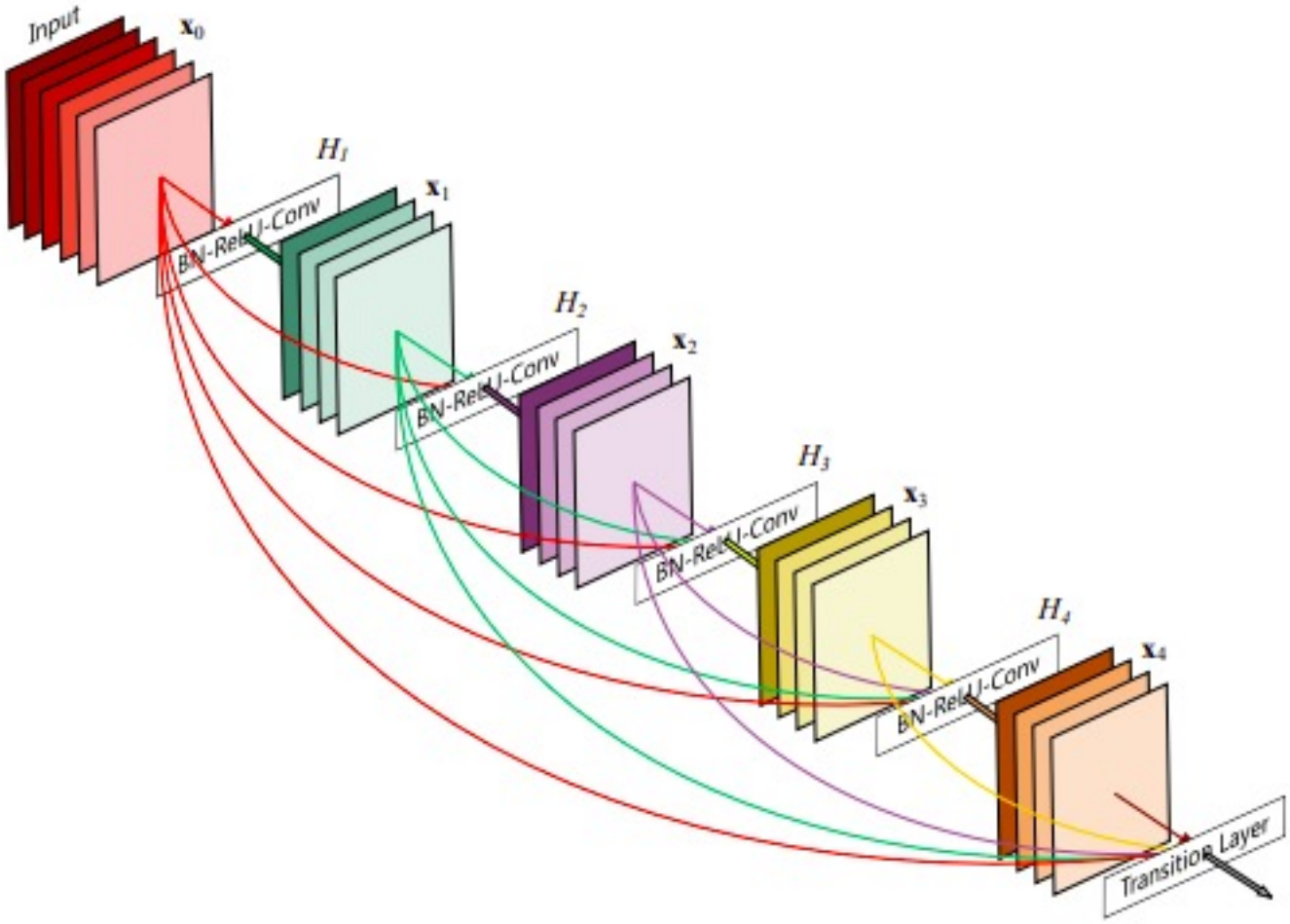


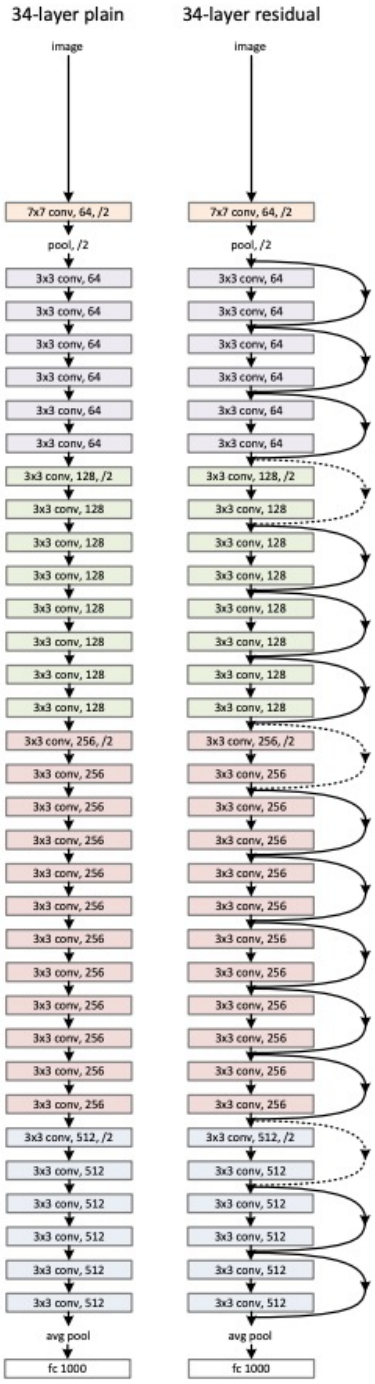
(a) without skip connections

(b) with skip connections

Skip connections

- Modern network architectures often involve many skip connections

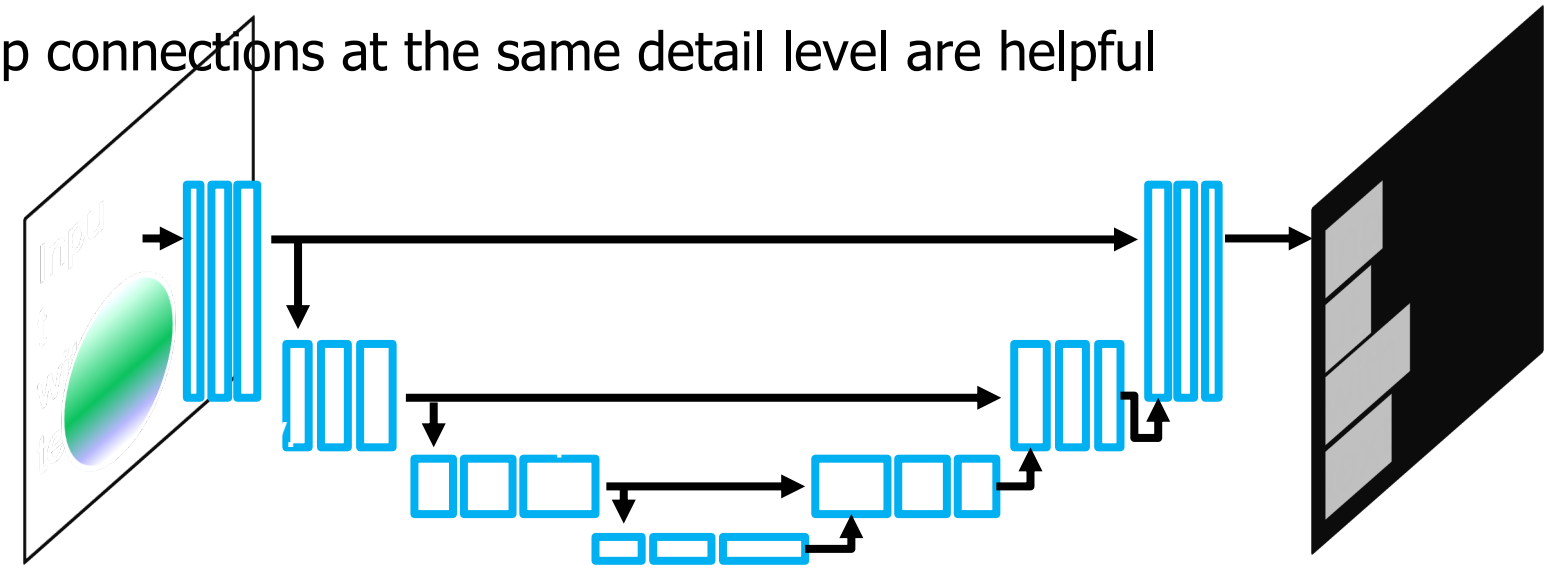




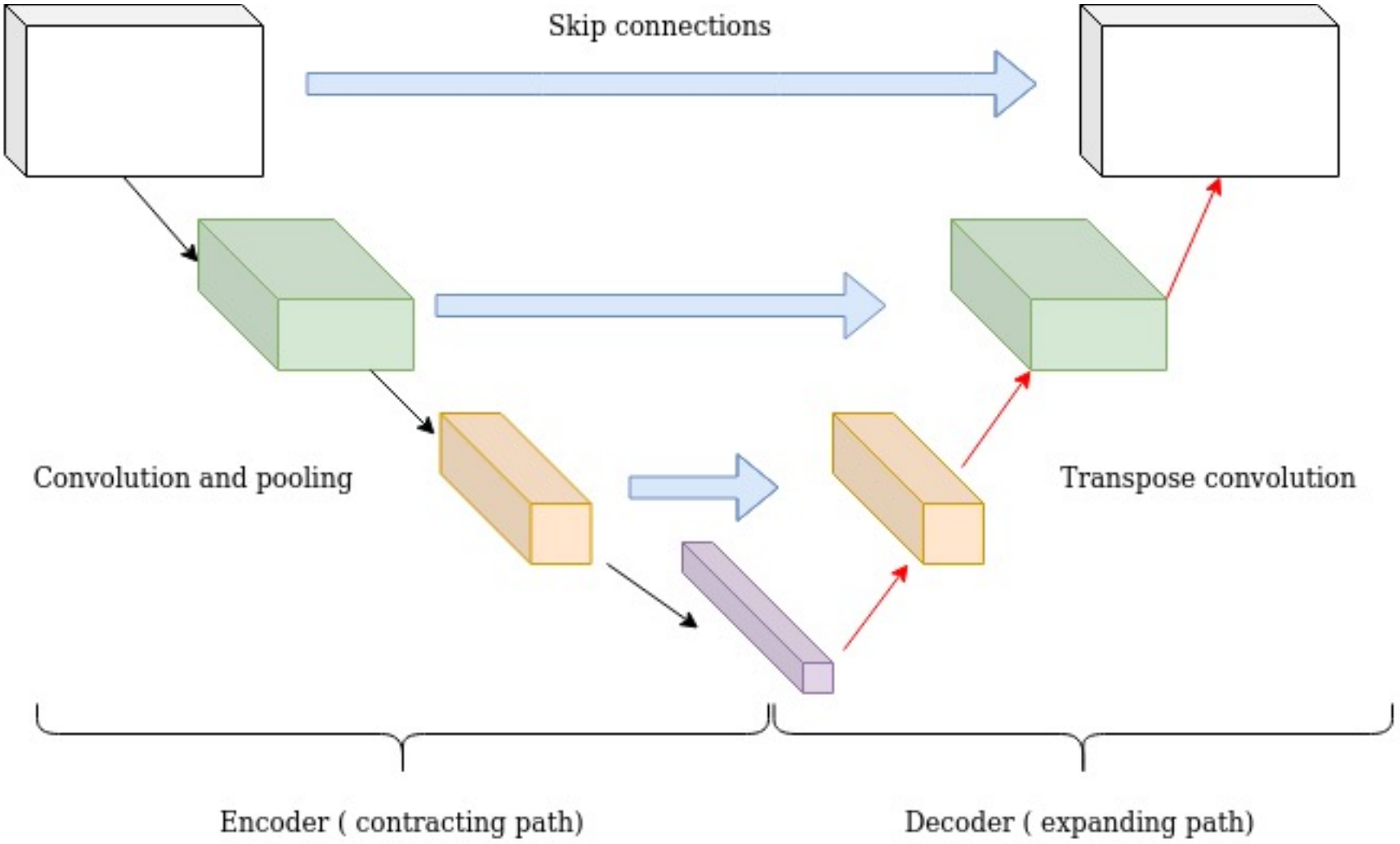
U-Net architecture

- Skip connections are used in image-to-image networks

- E.g. U-Net architecture
  - Convnet downsampling (via max pooling)
    - From detail to coarse-grained
  - Upsampling
    - Generate detail from coarse-grained
  - Skip connections at the same detail level are helpful

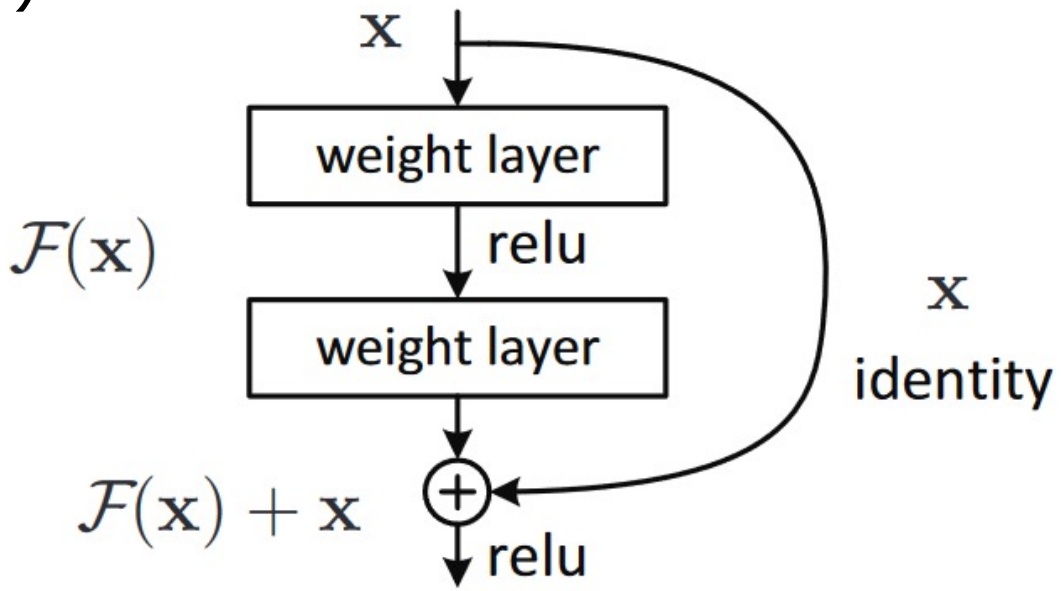


U-Net architecture



Skip connections - summary

- Skip connection can help with optimization
- They can also help with preserving details (information) from earlier layers, e.g. in U-Net architecture
- Networks with skip connections are often called residual networks (ResNets)
  - They only need to learn the residual (the "delta") f(x)-x not the whole f(x) from scratch



ODE-Net – infinite # of layers

A simple residual block

$$x_{t+1} = x_t + f_\Theta(x_t, t)$$

trainable net

Another view of ResNet

discrete layers

$$x_{t+1} - x_t = f_\Theta(x_t, t)$$

trainable net

ODE-Net

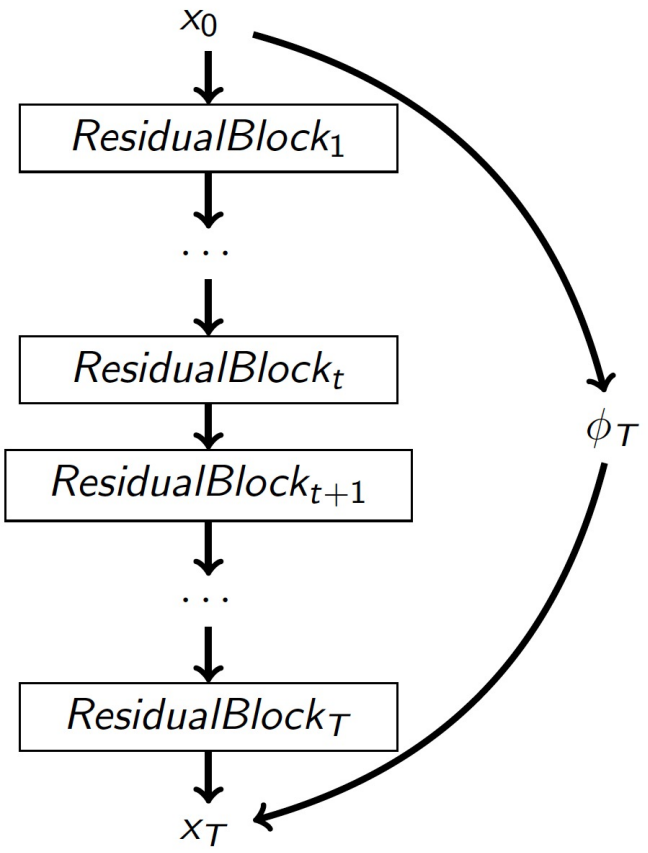
continuous time evolution through t instead of layers

$$\frac{dx_t}{dt} = \lim_{\delta_t \to 0} \frac{x_{t+\delta_t} - x_t}{\delta_t} = f_\Theta(x_t, t)$$

$$x_T = \phi_T(x_0) = x_0 + \int_0^T f_\Theta(x_t, t)dt$$

trainable net

$\phi_T : x_0 \to x_T$



ODE-Net – infinite # of layers

Another view of ResNet

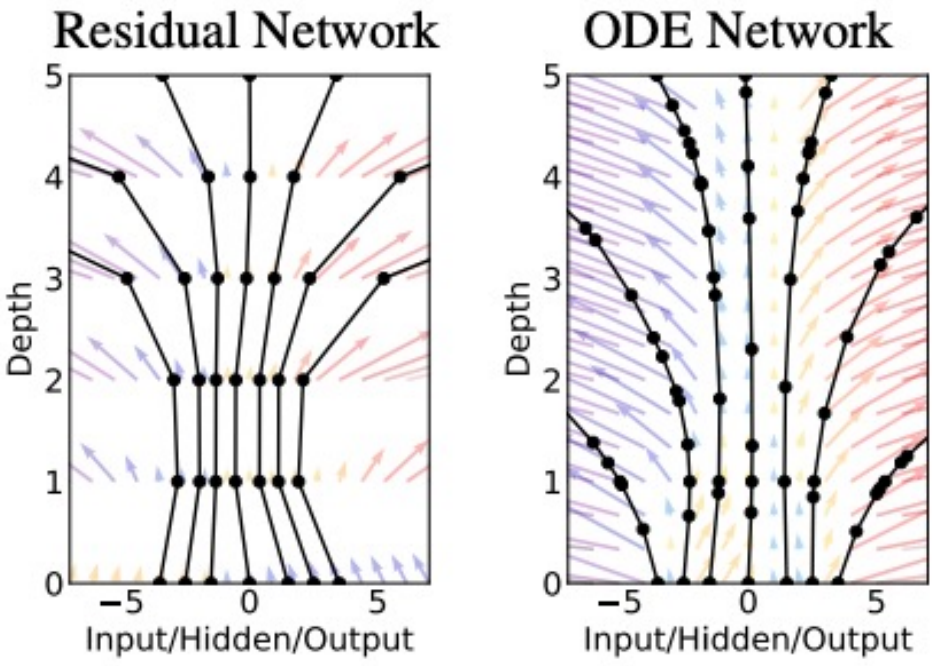
discrete layers

$$x_{t+1} - x_t = f_\Theta(x_t, t)$$

ODE-Net

continuous time evolution
through t instead of layers

$$\frac{dx_t}{dt} = \lim_{\delta_t \to 0} \frac{x_{t+\delta_t} - x_t}{\delta_t} = f_\Theta(x_t, t)$$

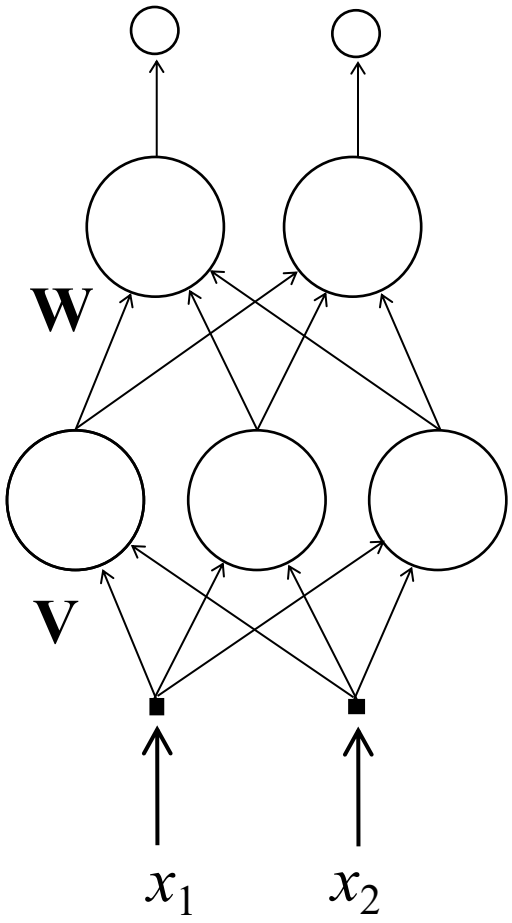


- Benefit: ODE-Nets are invertible
(like any ordinary differential equations)

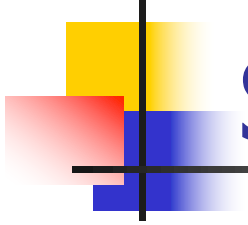
Design problems and solutions

- Choose architecture
  - Standard approach:
    fully connected feed-forward layers

- Problems with the standard approach:
  - Huge number of weights
    -> overtraining
- Partial solution:
  - **Specialized architectures**
    e.g. self-attention



Self-attention / Transformers

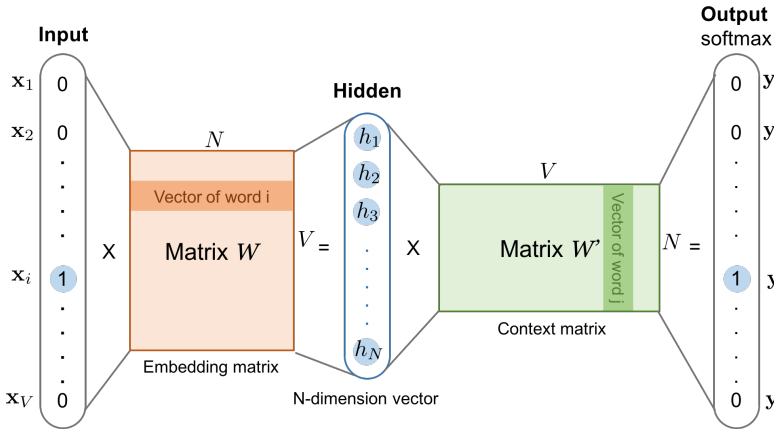


- Transformers are a new architecture the improves deep networks for natural language data
- How to represent text on input?
  - Word embeddings
- Transformers = context-depended word embeddings

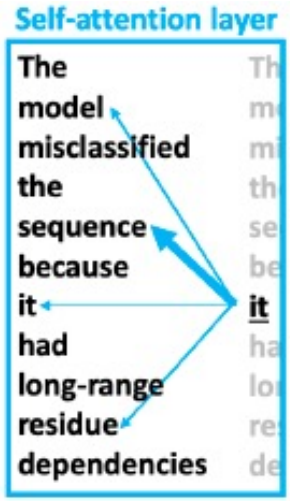
Word embeddings

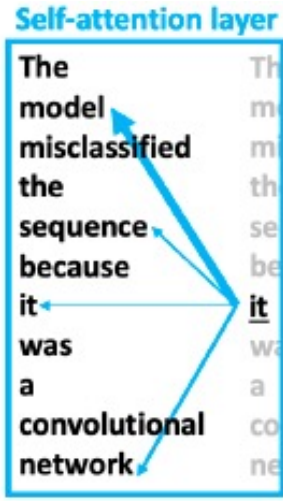
- Represent each word in a vocabulary using a vector
- Dimensionality of all vectors is the same
  e.g. 300
- Words that are similar have similar vector representation

cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

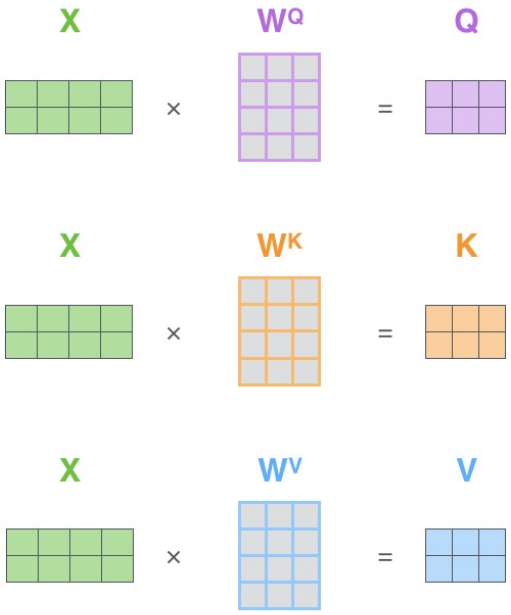


Self-attention



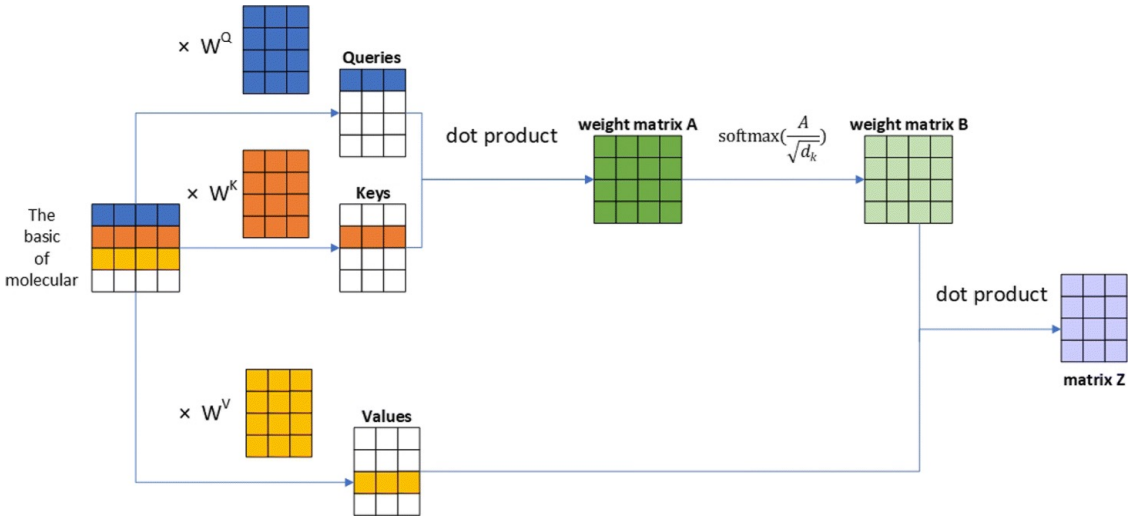


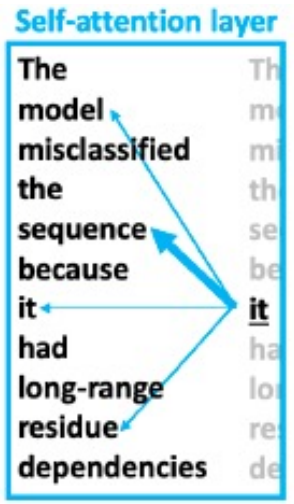
- Self-attention:
  - For each word, we calculate the attention to all other words
- Calculating self-attention for a pair of words A & B (Query and Key)
  - We take embedding of A (x^Q)
  - Do a linear projection (using trainable matrix W^Q)
  - We take embedding of B (x^K)
  - Do a linear projection (using trainable matrix W^K)
  - Unnormalized Attention(Q,K)
    = inner product of the two linear projections
  - We then apply softmax: attention for K is a probability over all Q in the sequence of words

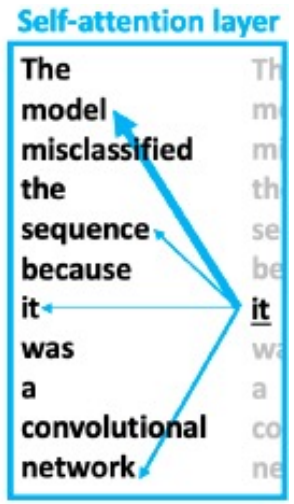


Contextual word embeddings

- Once we have attentions for each Key to all Queries, we use it to produce "new" (next layer) embeddings of each Key
  - We use previous embeddings of Queries, projected by matrix W^V, and weighted by the attention
    - We also have some trainable transformations on top, to increase flexibility
- The new embedding depends on other words in the sentence – it's contextual
- Self-attention: specialized to capture pair-wise relations





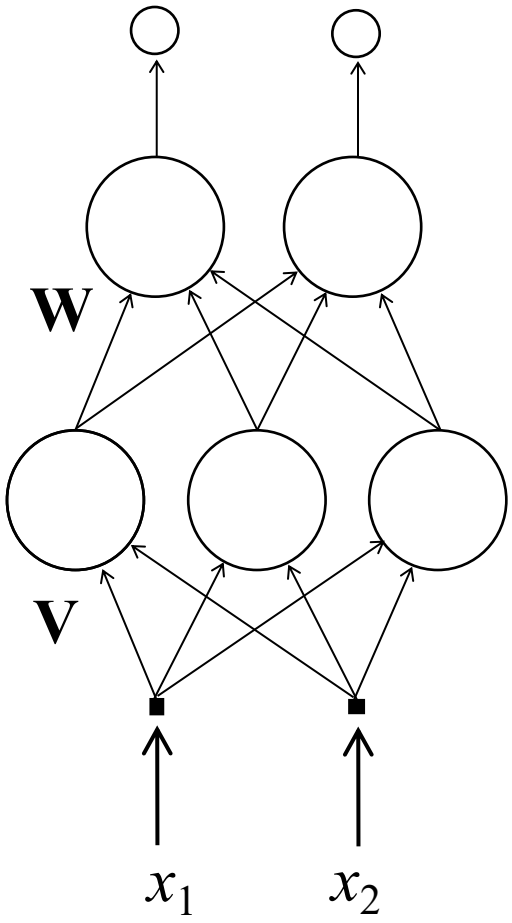


Summary so far

- Gradient descent can be encapsulated inside packages for automated differentiation
  - We just write what the model is, no need to worry about coding the details of the math needed to train it
- MSE loss and sigmoids may cause problems, use Cross Entropy loss and ReLU or similar
- Deep networks have many parameters layer after layer, can lead to problems with convergence to minimum, or to overtraining
  - Use specialized architectures: ConvNets, ResNets, Transformers

Design problems and solutions

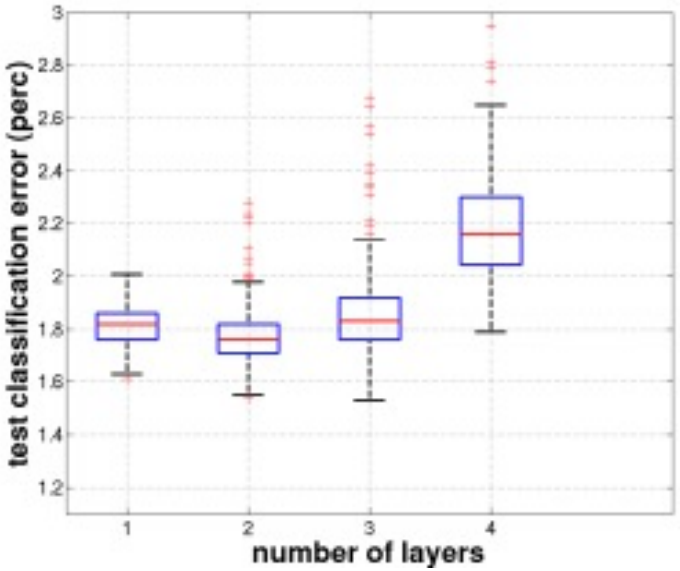
- Choose architecture
  - Standard approach:
    fully connected feed-forward layers



- Problems with the standard approach:
  - Huge number of weights
    -> overtraining
- Partial solution:
  - **Pre-training and transfer learning**

Training deep nets

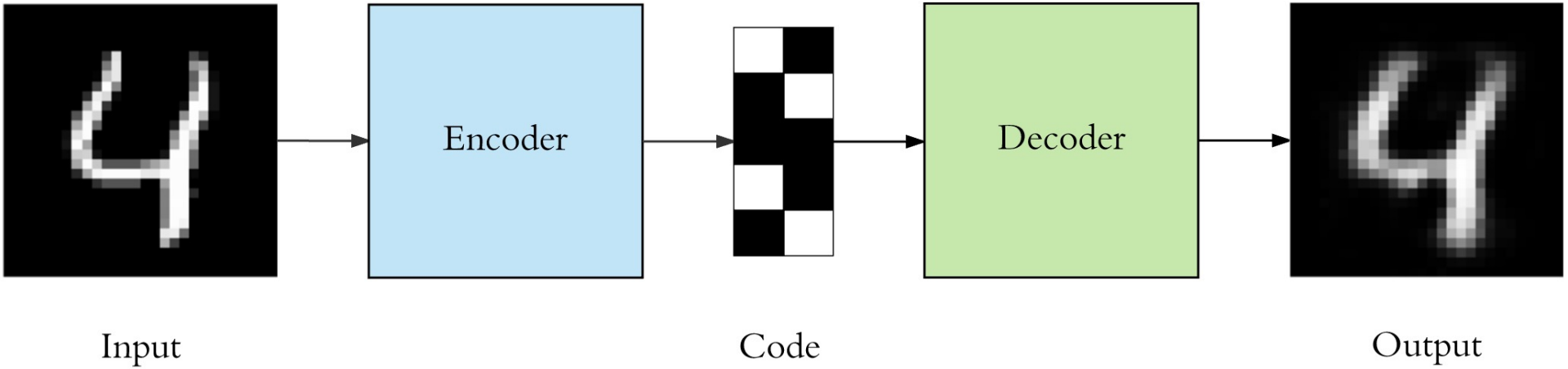
- Historically, deep networks (with many layers) were difficult to train



- A lot effort in recent years went into making training easier
  - ReLU etc. instead of sigmoid
  - Unsupervised pre-training
  - Normalization techniques

Pre-training

- Pre-training for dealing with depth of the network
  - Greedy layer-wise pretraining
- First, training an autoencoder unsupervised network



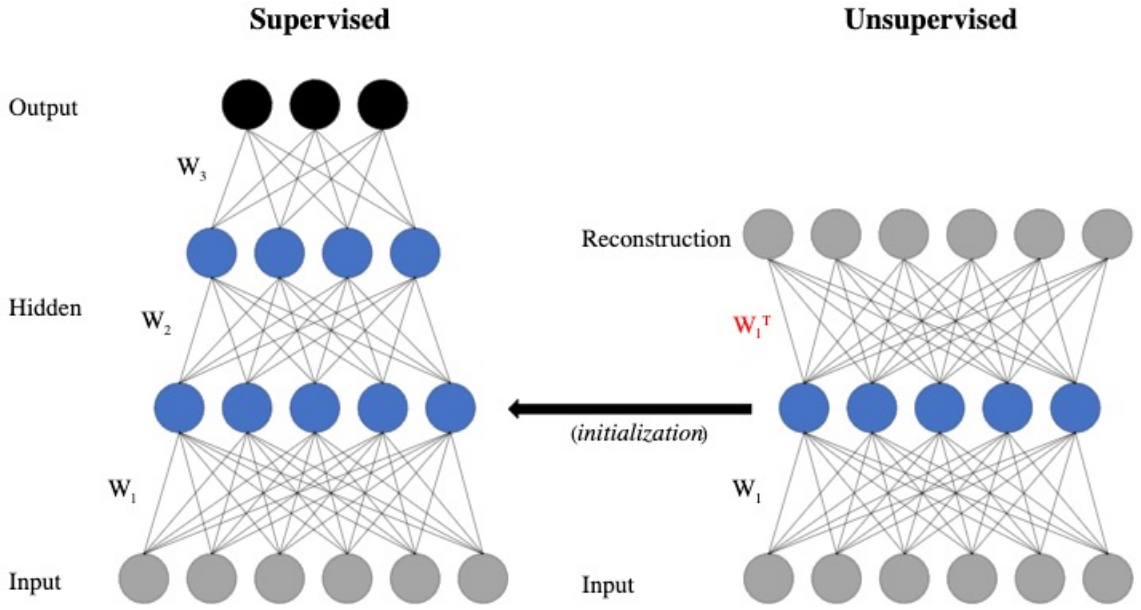
  - Two networks: an encoder + a decoder
  - Unsupervised training (no classes)
    - MSE: $(x - \text{decoder}(\text{encoder}(x))^2$

No sigmoid – no problem

https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

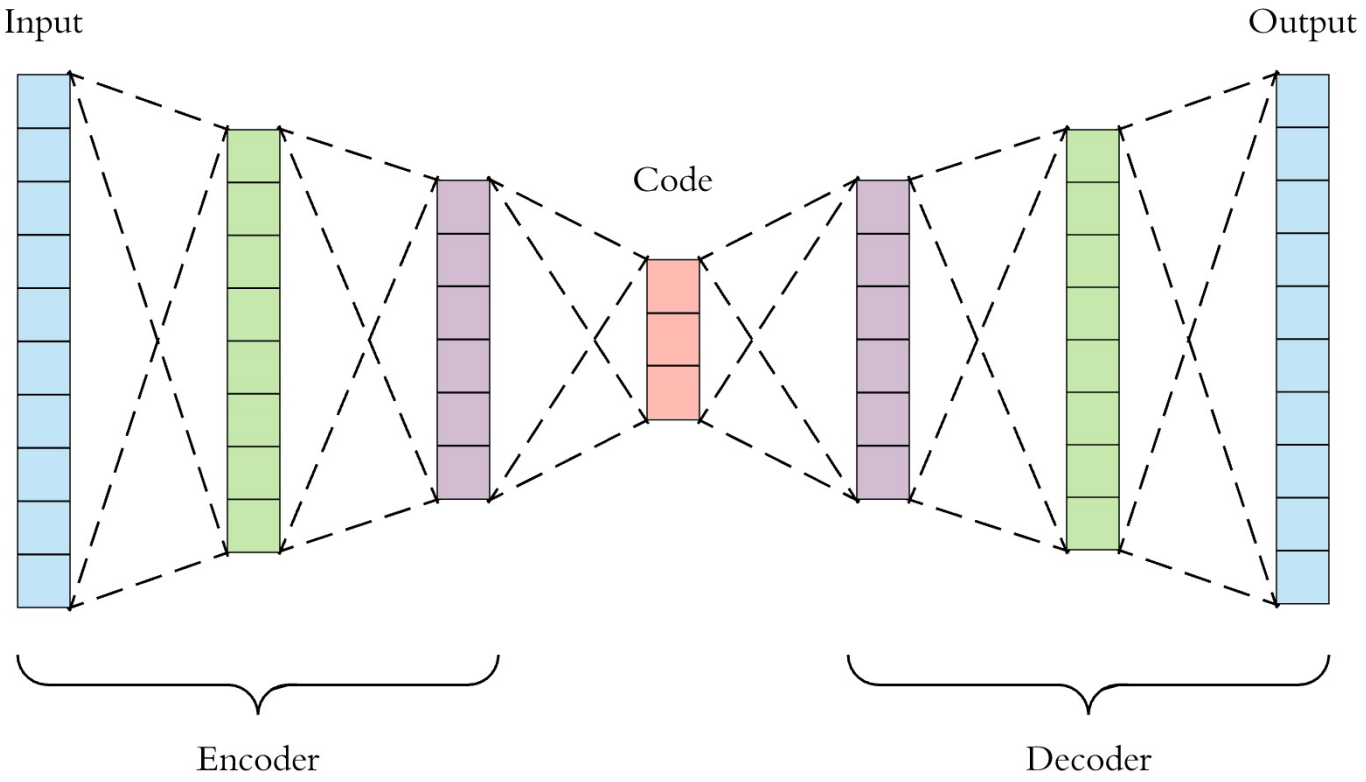
Pre-training

- Pre-training for dealing with depth of the network
  - Greedy layer-wise pretraining
- Second:
  - eliminate the decoder
  - take the encoder part
  - add a layer or two
  - softmax on top
  - train for classification (e.g. cross-entropy)



Pre-training

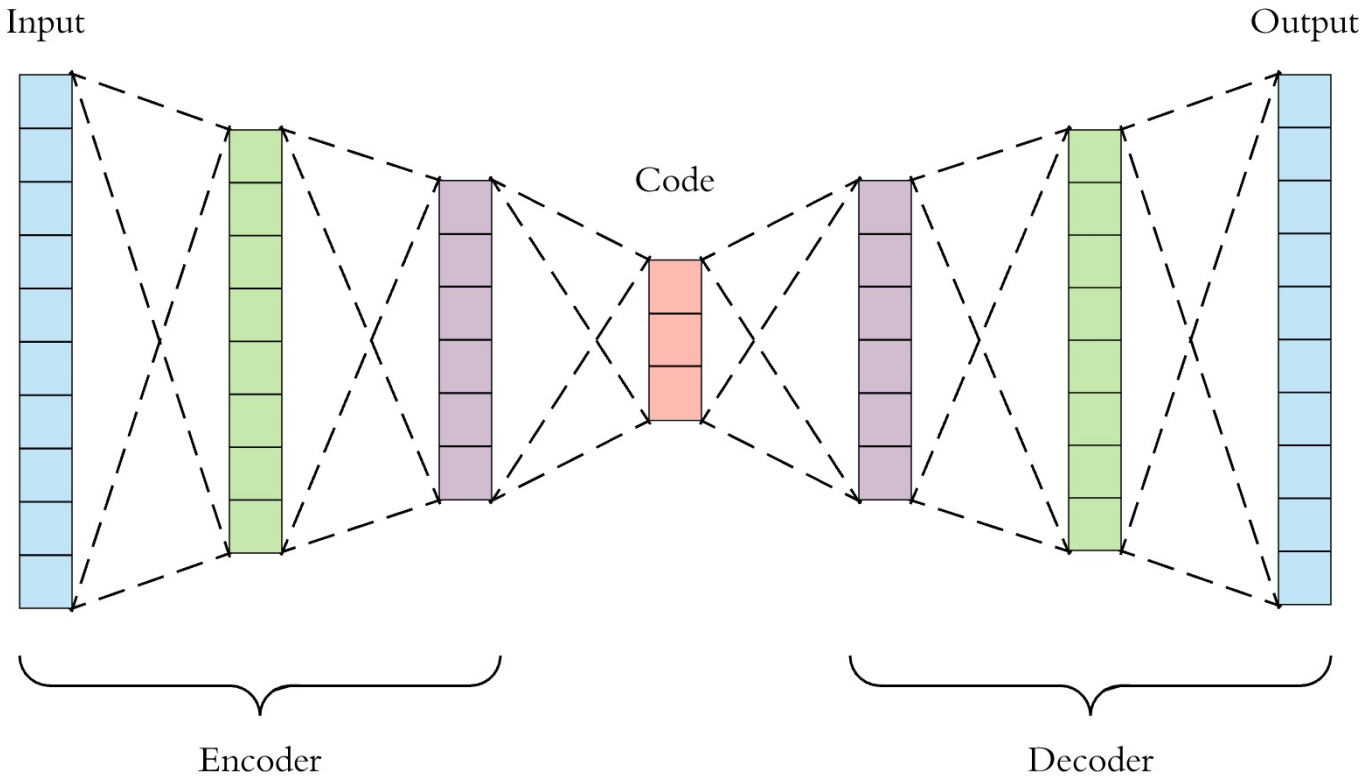
- training an autoencoder unsupervised network



- If the final supervised network needs to be large (many layers) the autoencoder also needs to be large (many layers)

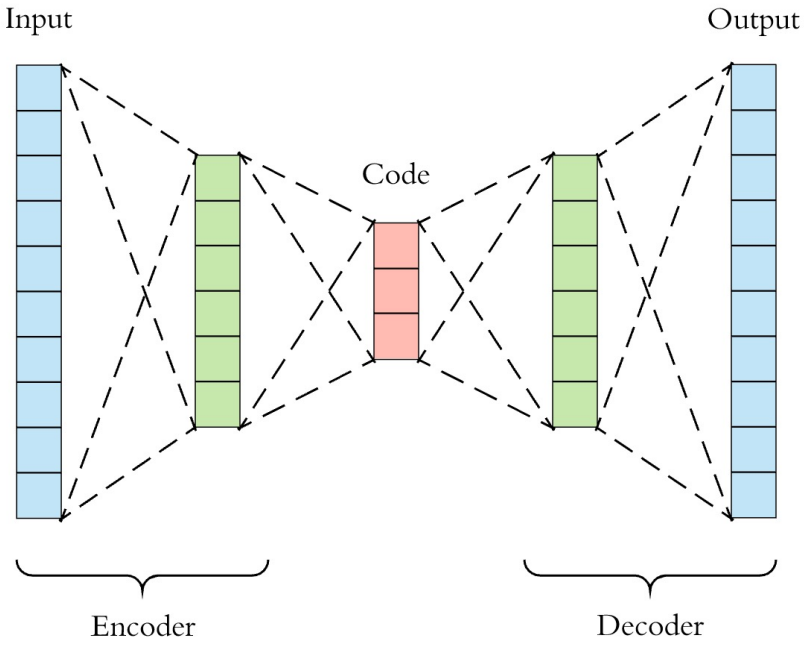
Pre-training

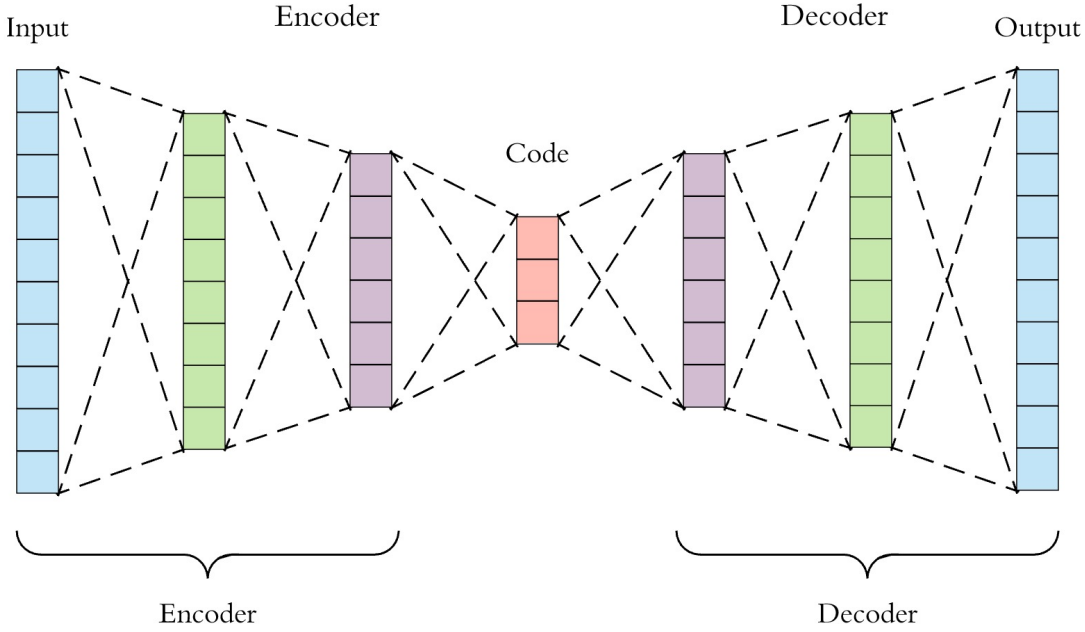
- Autoencoder with many layers
  - *n* layers - may be as difficult to train as a deep supervised network with *n* layers



Pre-training

- Greedy layer-wise pretraining
  - First train a shallow autoencoder (one hidden layer – green)
  - Next, add another hidden layer (violet)
    - Keep the previous weights as initialization
  - Repeat, adding one layer at a time
    - Actually, two: one in encoder one in decoder

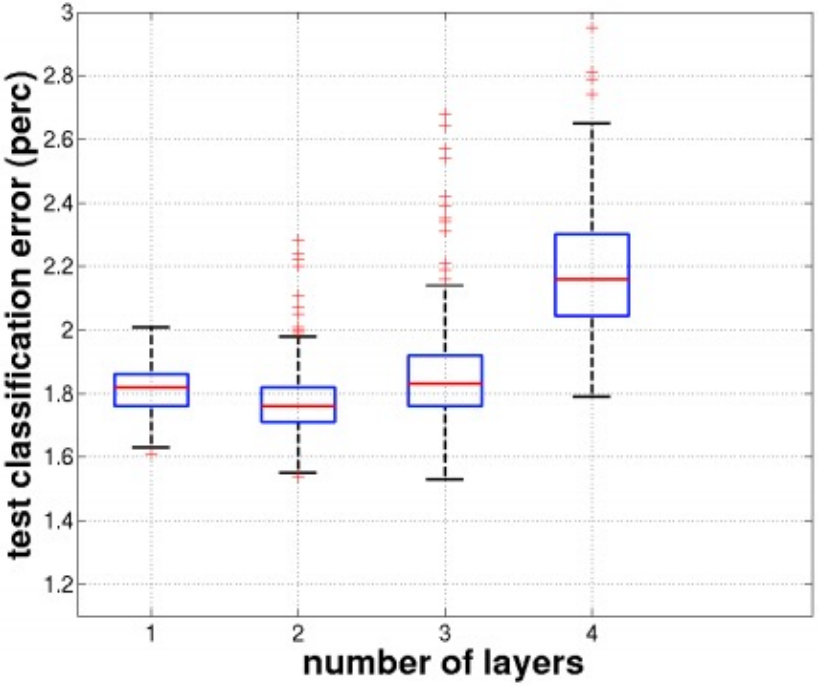


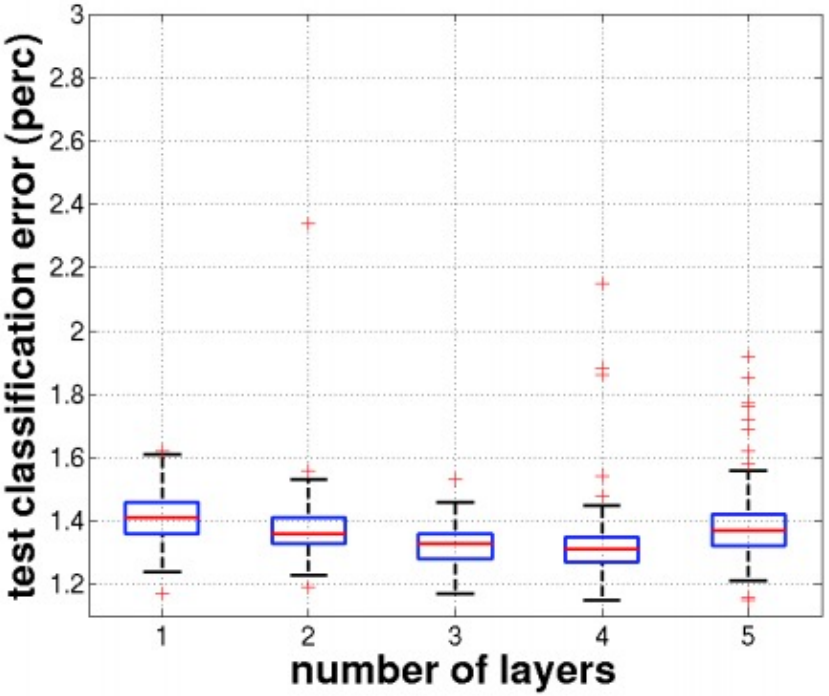


Training deep nets

- Pre-training can help in training deep nets

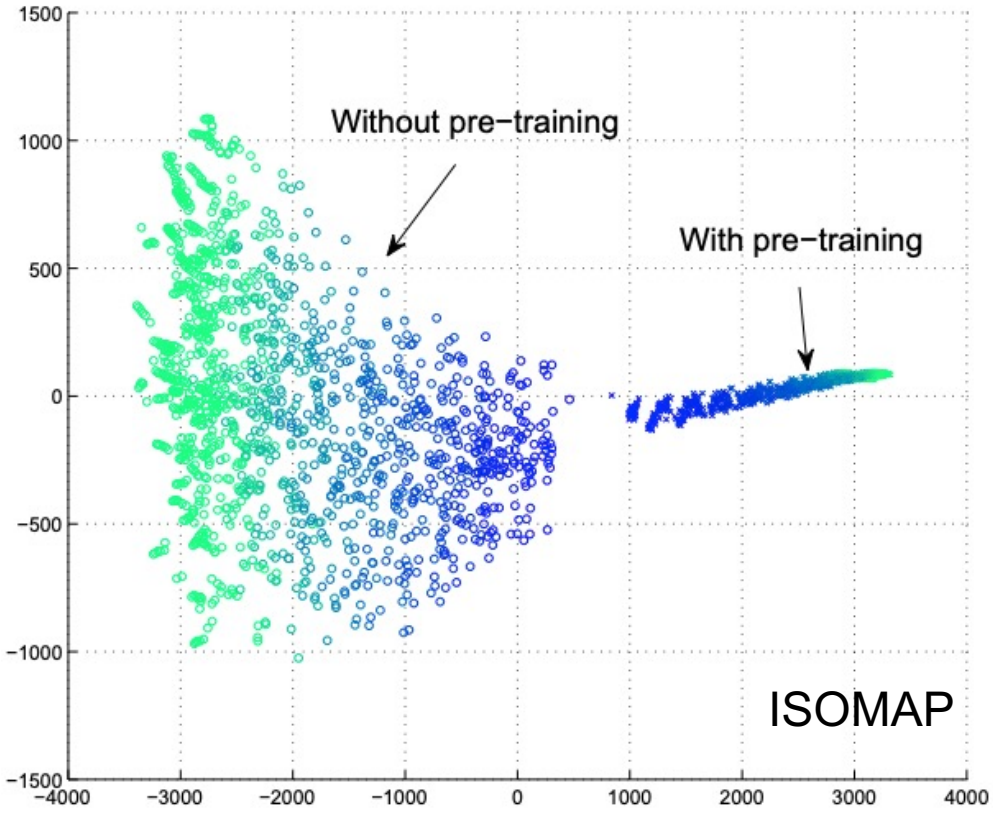
(no pre-training) (with pre-training)





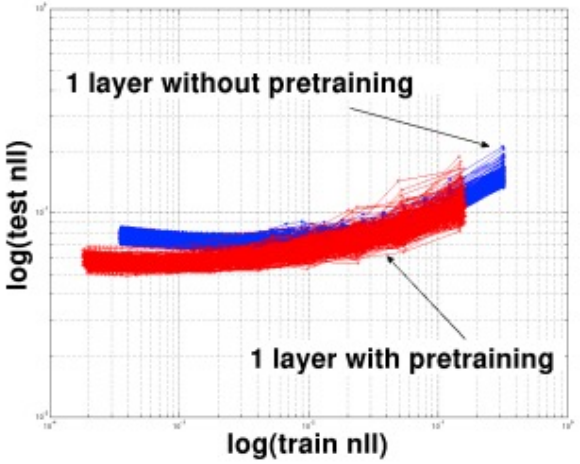
Training deep nets

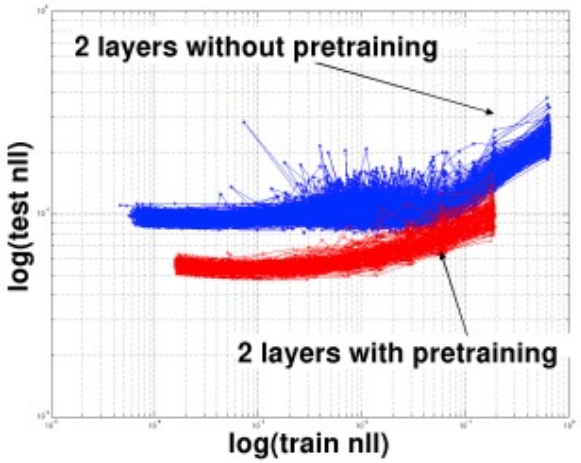
- Pre-training serves as a regularizer
  - Different runs of pre-trained nets are similar, and converge during training
  - Different runs of non-pre-trained nets progress in different directions during training (blue to green)

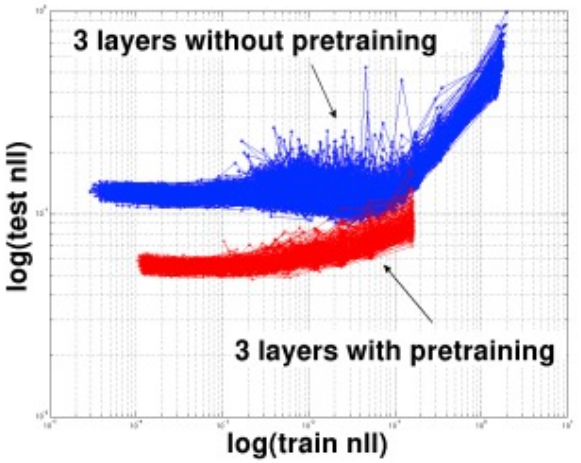


Training deep nets

- Pre-trained networks overtrain less
  - For the same training loss test loss is lower for pre-trained net

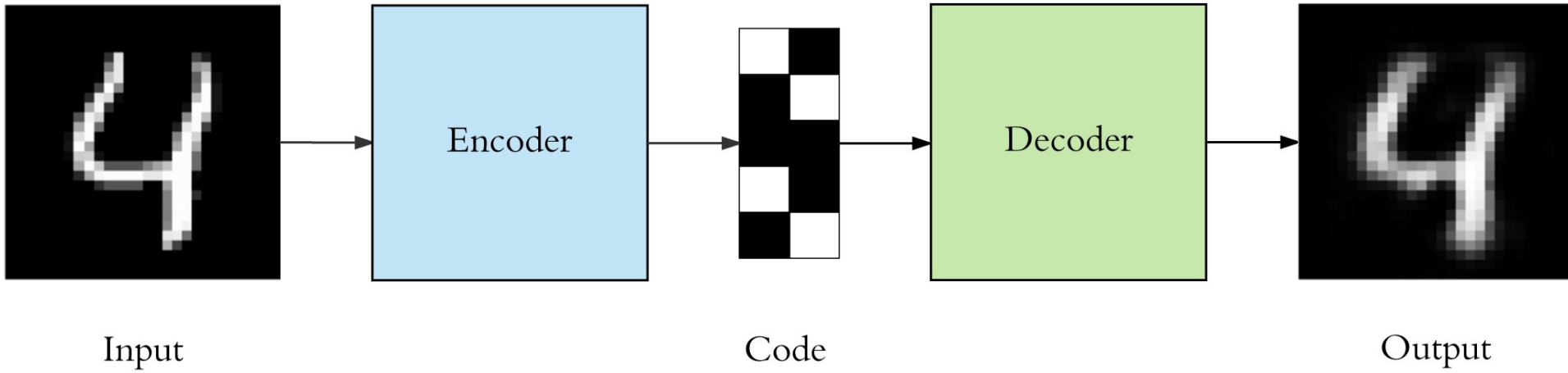




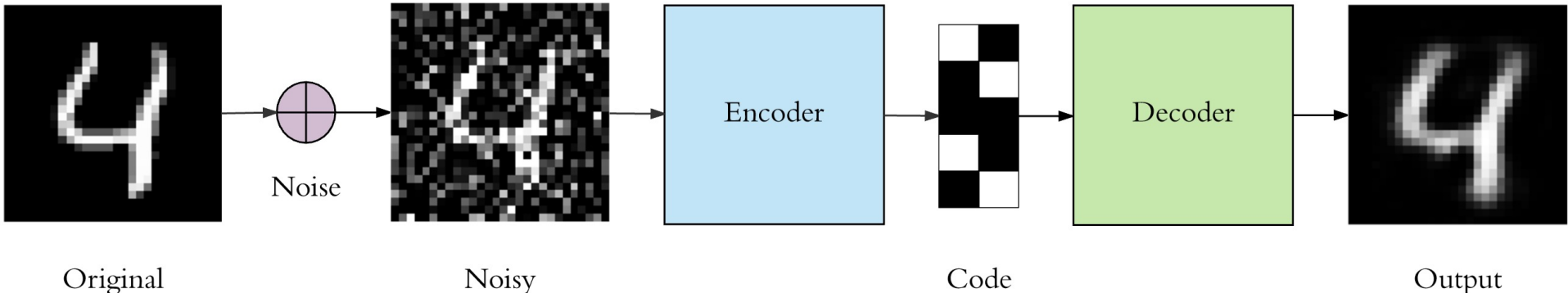


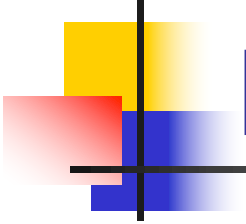
Pre-training

- De-noising autoencoder:



  - We can make the network more resistant to minor variations in input



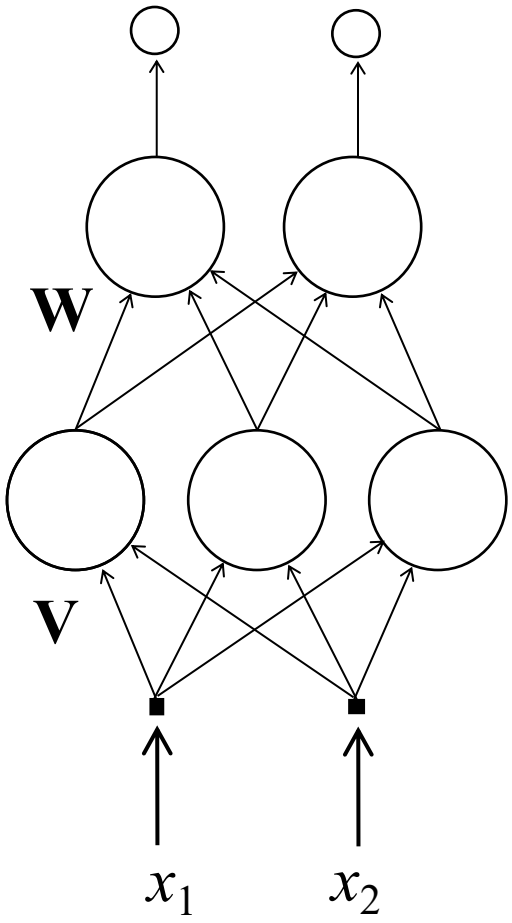


Pre-training

- Pre-training for dealing with depth of the network
  - Greedy layer-wise pretraining
  - Is not as much needed these days – there are other techniques for helping with optimization of deep nets
- Pre-training (not greedy, not layer-wise) is still very useful for dealing with limited training data
  - Transfer learning
    - E.g. training Transformer on the whole Wikipedia to predict some "redacted" words
    - Then fine-tuning for the actual task (e.g. sentiment analysis) on a much smaller dataset
  - Teacher-student learning

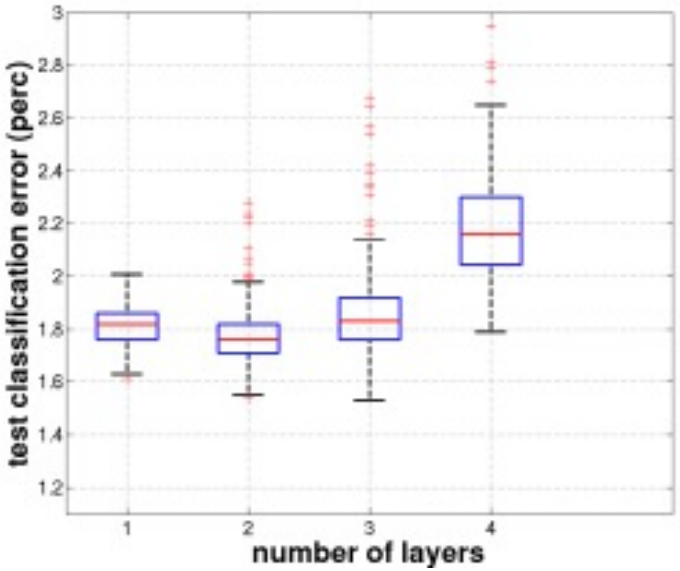
Design problems and solutions

- Choose architecture
  - Standard approach:
    fully connected feed-forward layers
- Choose the optimization method
  - Standard approach:
    - Gradient descent
- Problems with the standard approach:
  - Vanishing gradients
  - Exploding gradients
- Partial solution:
  - **Normalization**



Training deep nets

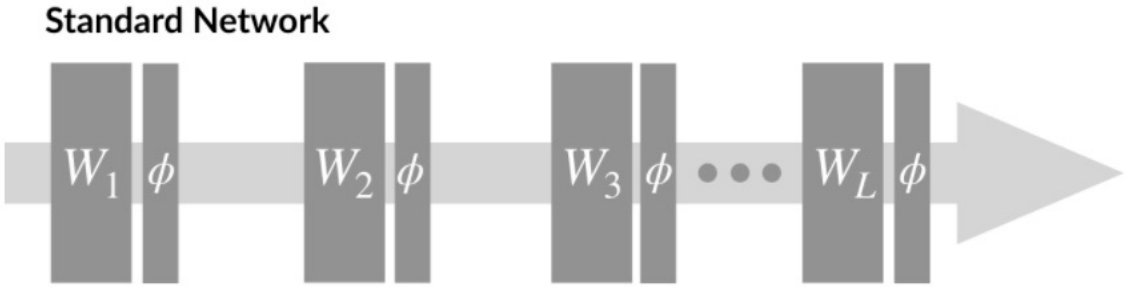
- Historically, deep networks (with many layers) were difficult to train



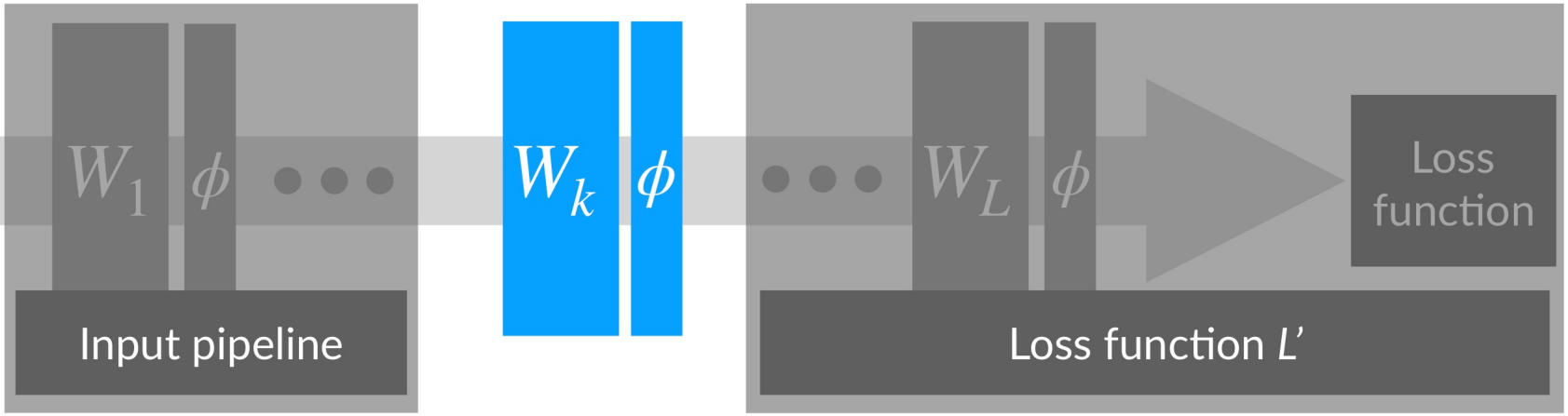
- A lot effort in recent years went into training deep nets easier
  - ReLU instead of sigmoid
  - Unsupervised pre-training
  - Normalization techniques

Layer normalization

- A standard deep net has layers, where each layer is
  - a linear transformation W$\boldsymbol{\phi}$, where $\boldsymbol{\phi}$ is input from previous layer
  - a nonlinear activation function acting on W$\boldsymbol{\phi}$, e.g. ReLU(W $\boldsymbol{\phi}$)

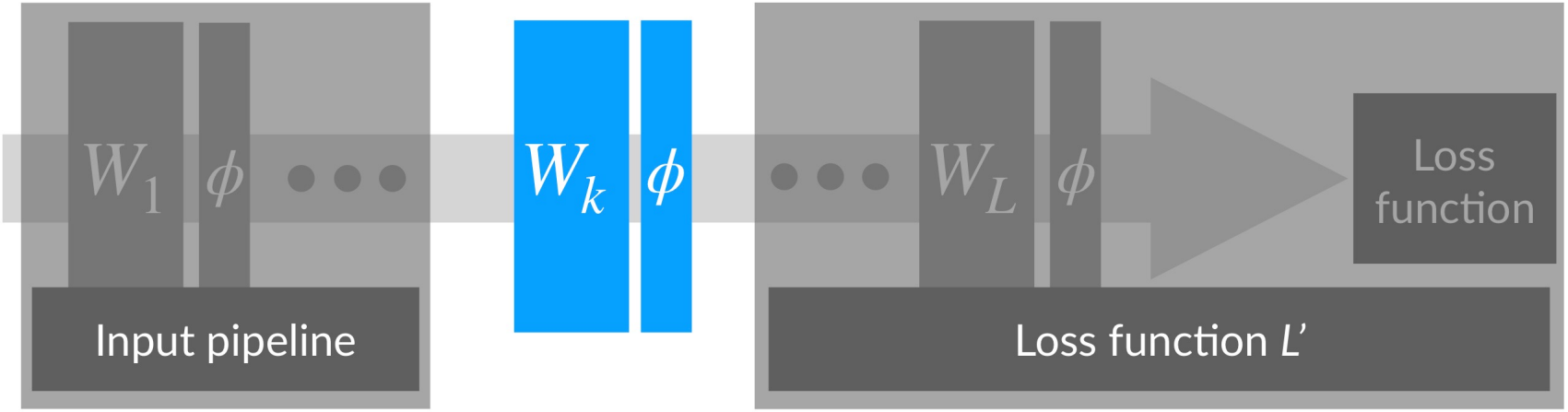


- We can look at the training from the perspective of a single layer – as if the other layers were constant



Layer normalization

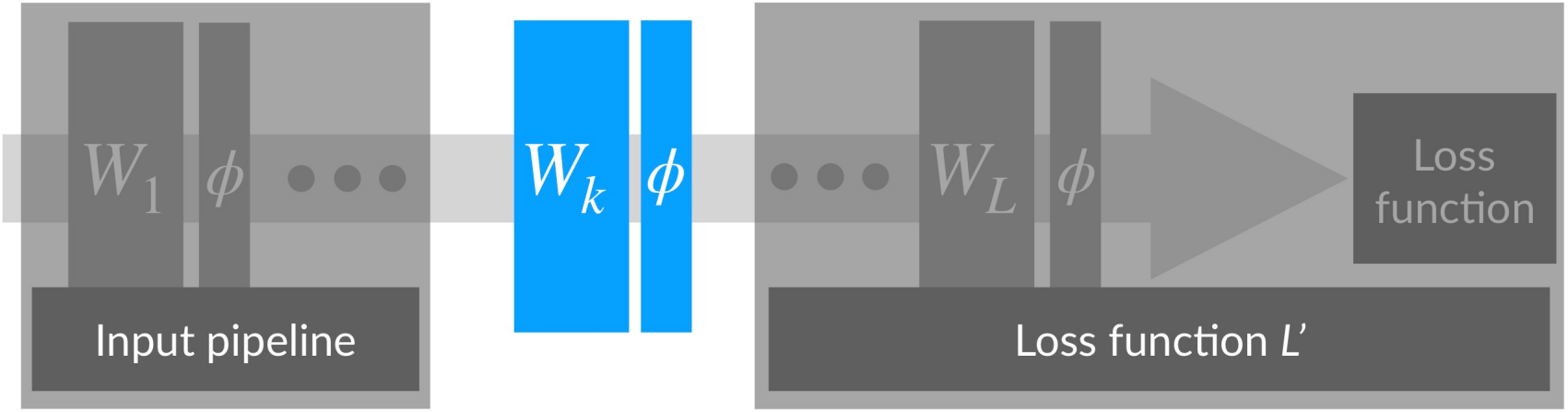
- We can look at the training from the perspective of a single layer – as if the other layers were constant



- Layer k will learn how to transform its input $\boldsymbol{\phi}$ (output of layers 1,…,k-1) into something that minimizes the ”new loss” (layers k+1,…,L + original loss).

Layer normalization

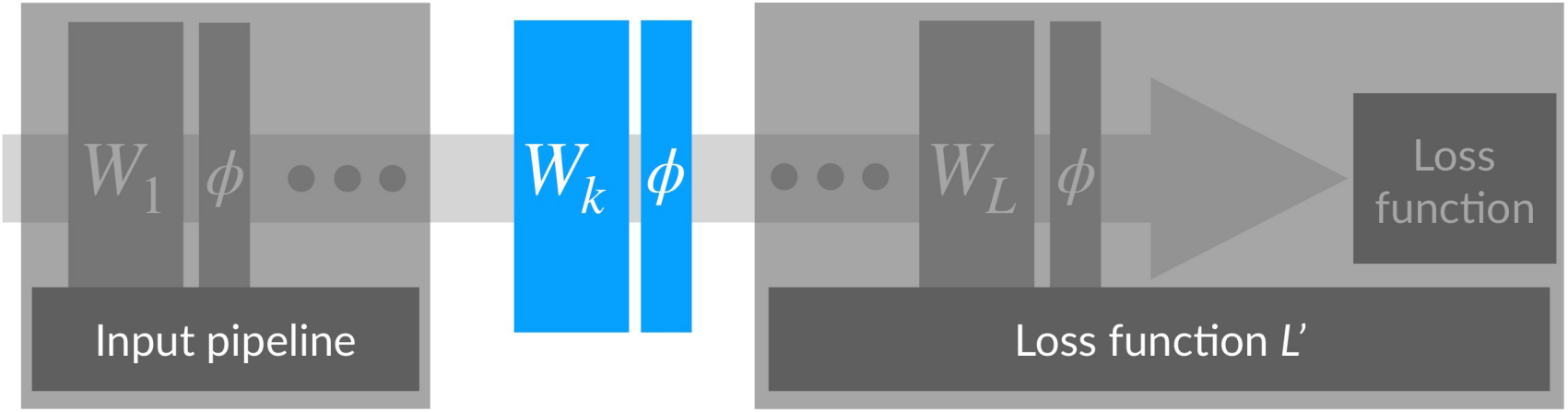
- We can look at the training from the perspective of a single layer – as if the other layers were constant



- If layers 1,…,k-1 are not constant (are trained), the distribution of input to layer k changes all the time
- If layers k+1,…,L are not constant (are trained), the "new loss" also changes all the time

Layer normalization

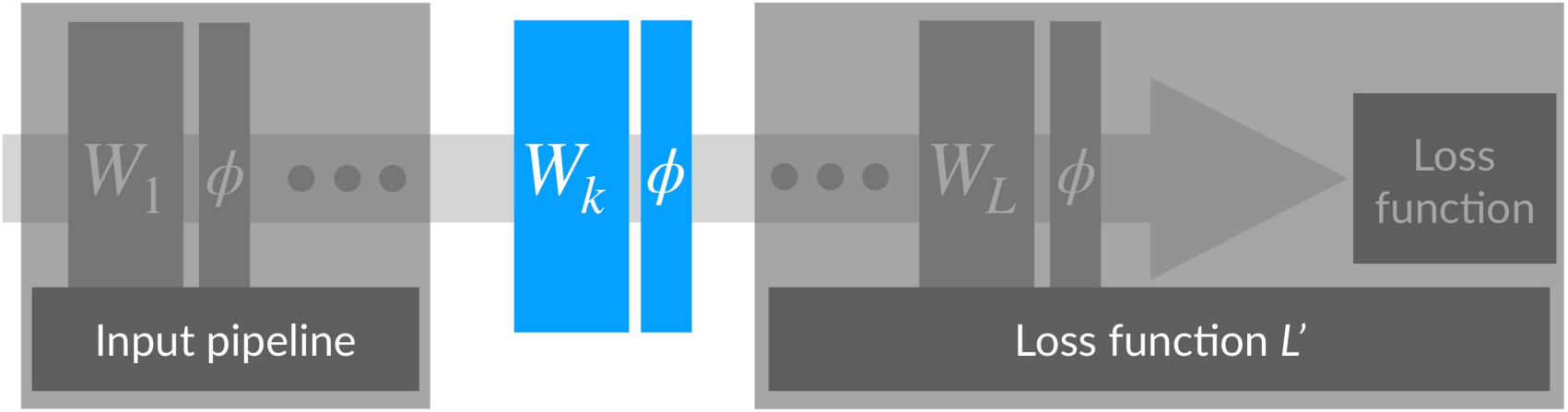
- We can look at the training from the perspective of a single layer – as if the other layers were constant



- If layers 1,…,k-1 are not constant (are trained), the distribution of input to layer k changes all the time
- We can't fix the layers, to have the same distribution/loss
  - We do want all layers to learn, i.e., change what they're producing
- We can fix some general property of what a layer produces

Layer normalization

- We can look at the training from the perspective of a single layer – as if the other layers were constant



- We can fix some general property of what it produces

- Normalization:
  - Make some statistic (e.g. mean) of the outputs of a layer constant, even if the actual output vectors change during training

Layer normalization

- Normalization:
  - Make some statistic (e.g. mean, or std.dev) of the outputs of a layer constant, even if the actual output vectors change during training
- Output of a layer with 6 neurons, presented with a batch of 4 samples
  - a 4x6 matrix $W\phi$
    - Rows represent samples
    - Columns represent activations / neuron outputs from the layer

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

- Possible options:
  - LayerNorm: Normalize rows (separately each sample, across the neurons)
    - To have 0 mean, unit std.dev.
  - BatchNorm: Normalize columns (separately each neuron, across the batch):
    - To have 0 mean, unit std.dev.

Batch normalization

- Normalization:
  - Make some statistic (e.g. mean, or std.dev) of the outputs of a layer constant, even if the actual output vectors change during training

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

- BatchNorm
  - Normalize columns (separately each neuron, across the batch):
    - Squares add up to 1, i.e. Std.Dev=1
    - Mean = 0

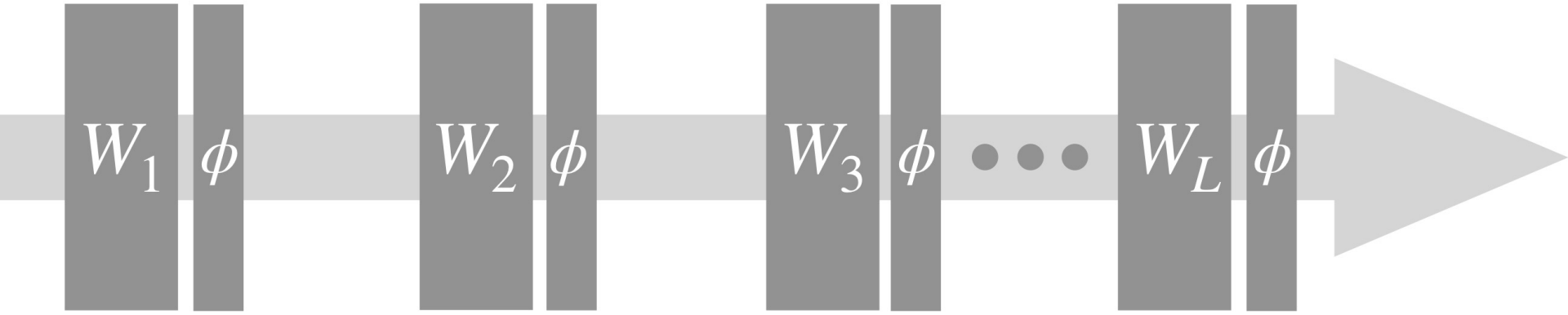
$$BN(y_j)^{(b)} = \gamma \cdot \left(\frac{y_j^{(b)} - \mu(y_j)}{\sigma(y_j)} \right) + \beta$$

    - Add scale and off na)
  - These are normal computations in the computational graph
    - i.e. chain rule applies to the operation of calculating mean/std.dev

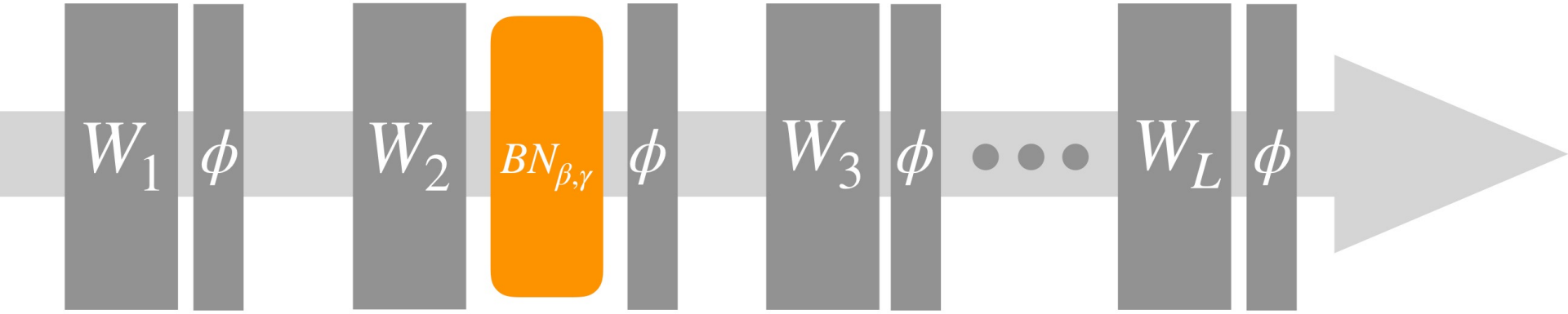
BatchNorm

$$BN(y_j)^{(b)} = \gamma \cdot \left(\frac{y_j^{(b)} - \mu(y_j)}{\sigma(y_j)} \right) + \beta$$

Standard Network



Adding a BatchNorm layer (between weights and activation function)

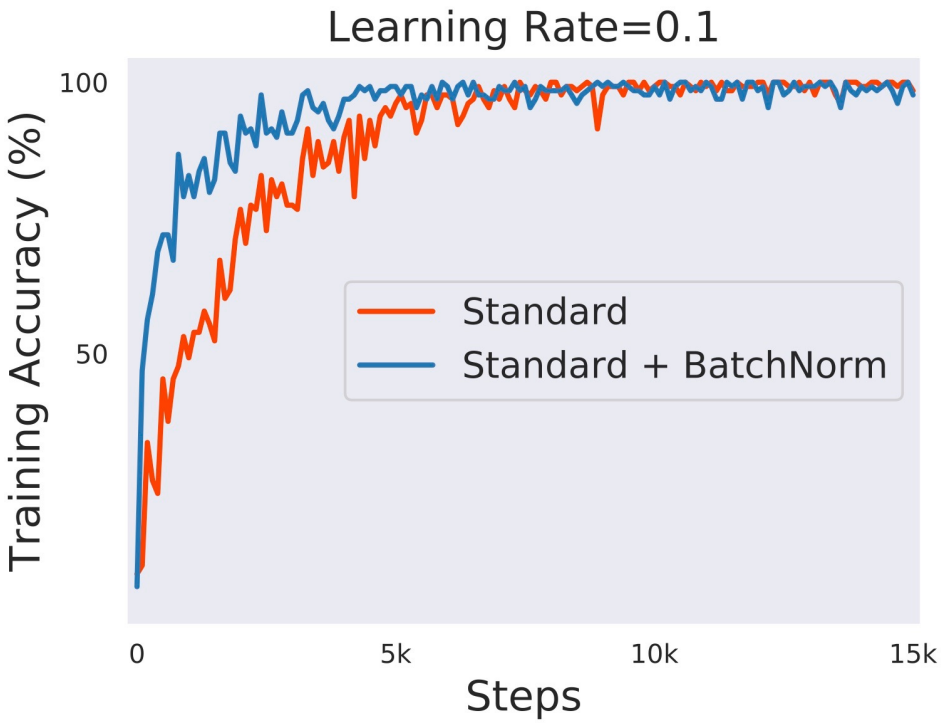


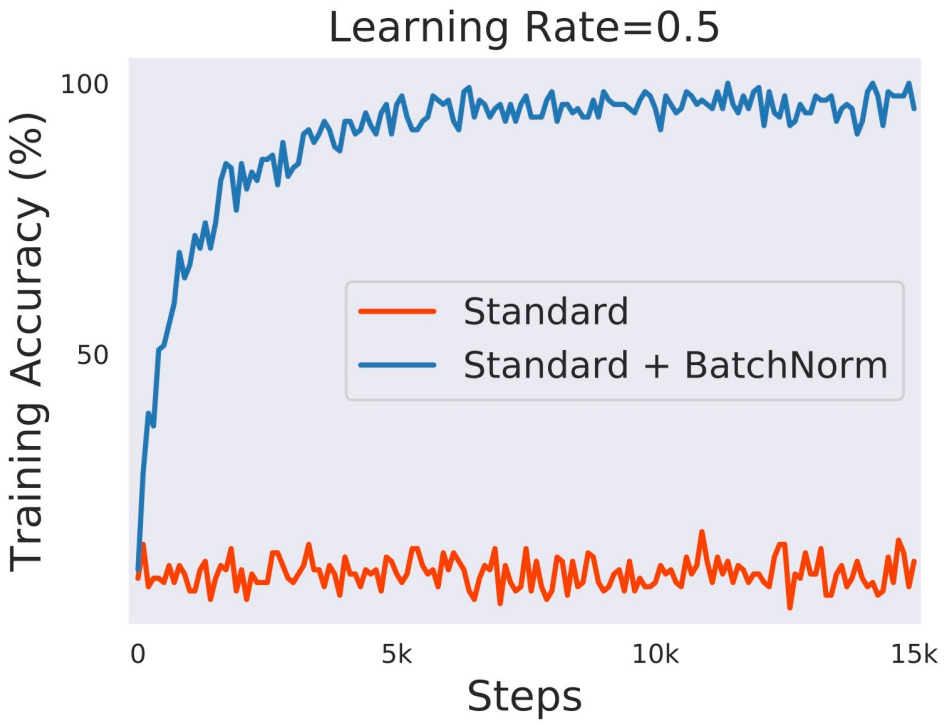
Batch normalization

- A three-layer neural network with ReLU activation is:
  - $Y=ReLU(\ W_3\ ReLU(\ W_2\ ReLU(\ W_1X\ )\ )\ )$
- Expanding ReLU(wx)=max(0,wx), we see either wx, or 0
  - $ReLU(w_2ReLu(w_1x))$ can be w_2*w_1*x
- Jointly over three layers, we see terms like $z=w_{3ij}*w_{2kl}*w_{1mn}*x$
  - The derivative of z over w_{1mn} is $w_{3ij}*w_{2kl}*x$
- The derivative can quickly get large even if individual w's are not that much larger than 1
- Or can get very small if individual w's are close to zero
- Vanishing/exploding gradient!

Batch normalization

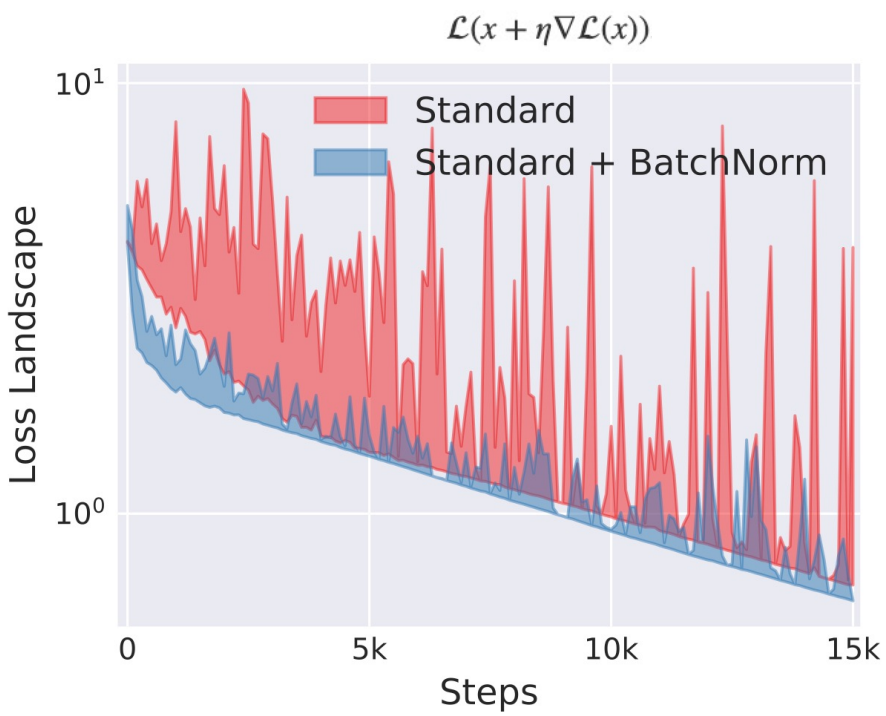
- Vanishing gradients slow learning
- Exploding gradients prevent learning

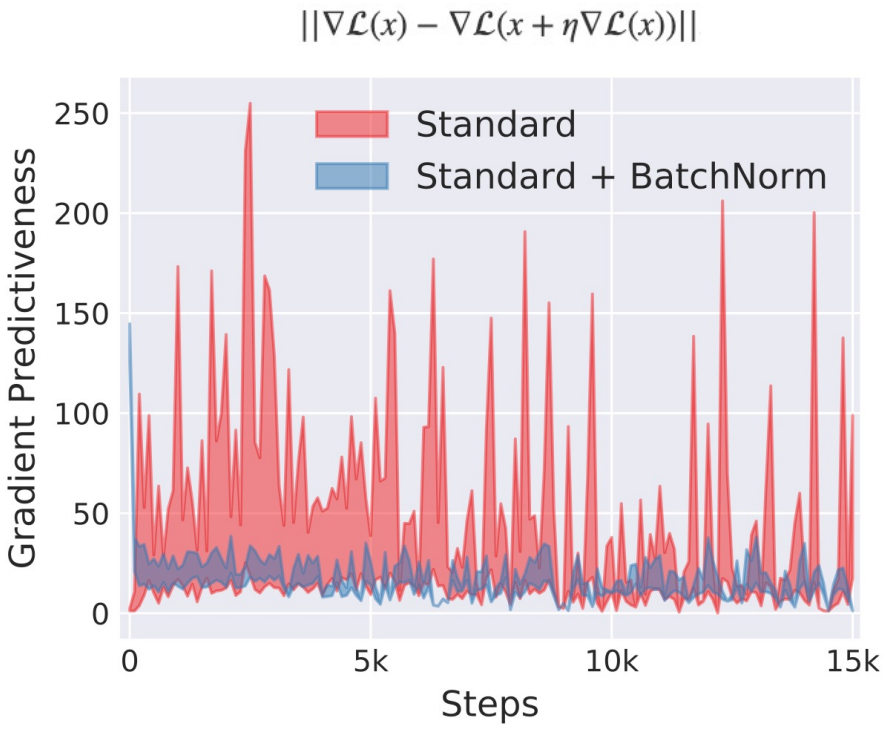




Batch normalization

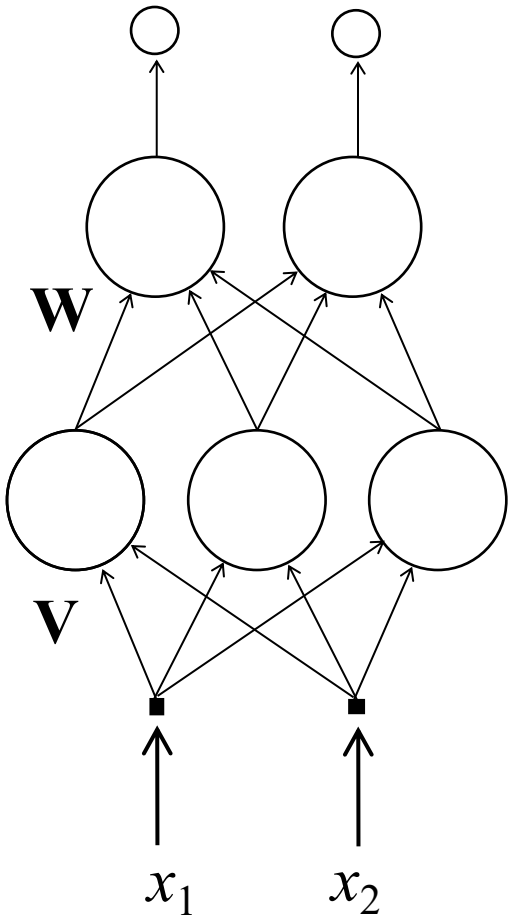
- The derivative over w_{2kl} is w_{3ij}* w_{1mn} *x
  ~ w_{3ij}*activation(prev.layer)
- Solution: make activations "just right"
  - E.g. make them roughly follow a Gaussian with 0-mean, unit norm
- Helps keep loss stable:





Design problems and solutions

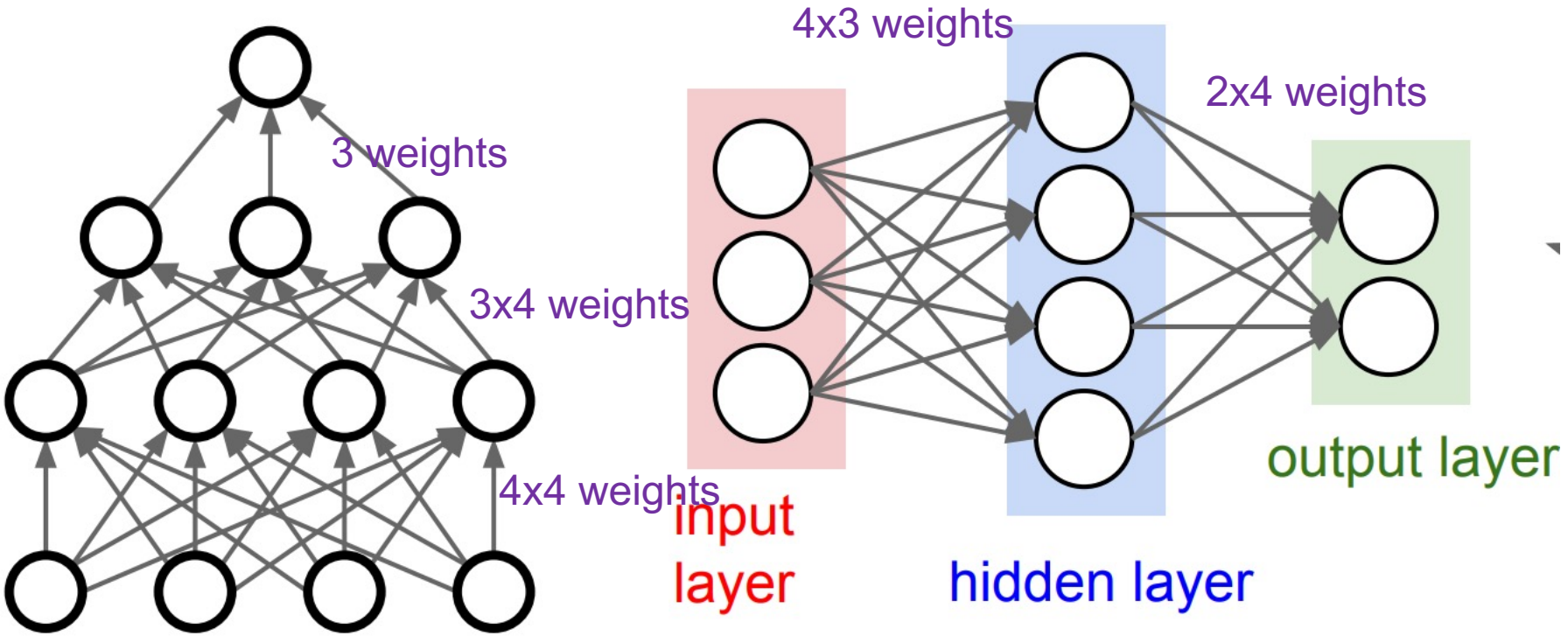
- Choose architecture
  - Standard approach:
    fully connected feed-forward layers



- Problems with the standard approach:
  - Huge number of weights
    -> overtraining
- Partial solution:
  - **Regularization via weight sharing**
    - **E.g. dropout**

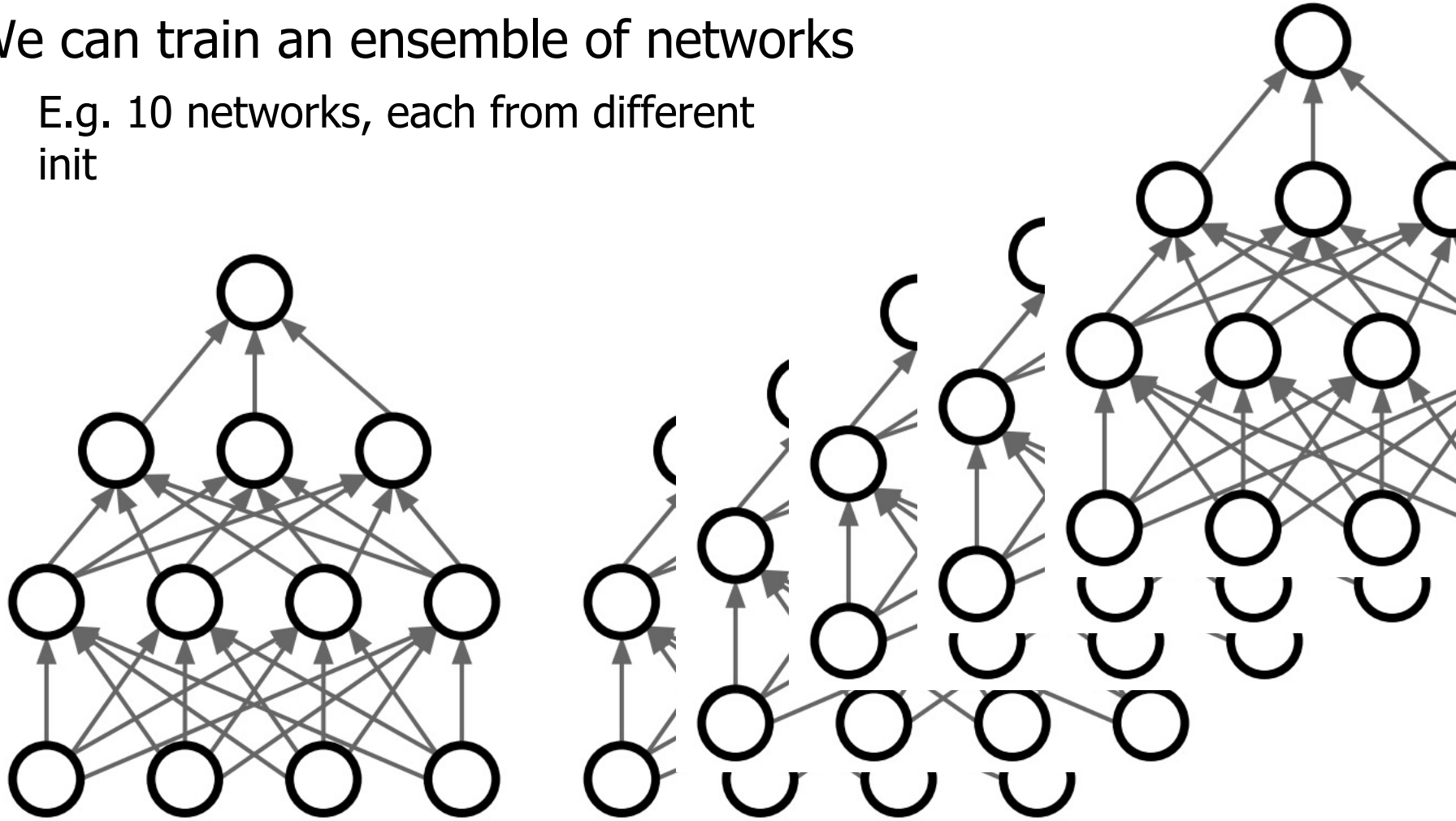
Regularization in deep nets

- A feedforward network has many many weights



Regularization in deep nets

- Each different initialization from random weights will lead to a different network
- We can train an ensemble of networks
  - E.g. 10 networks, each from different init

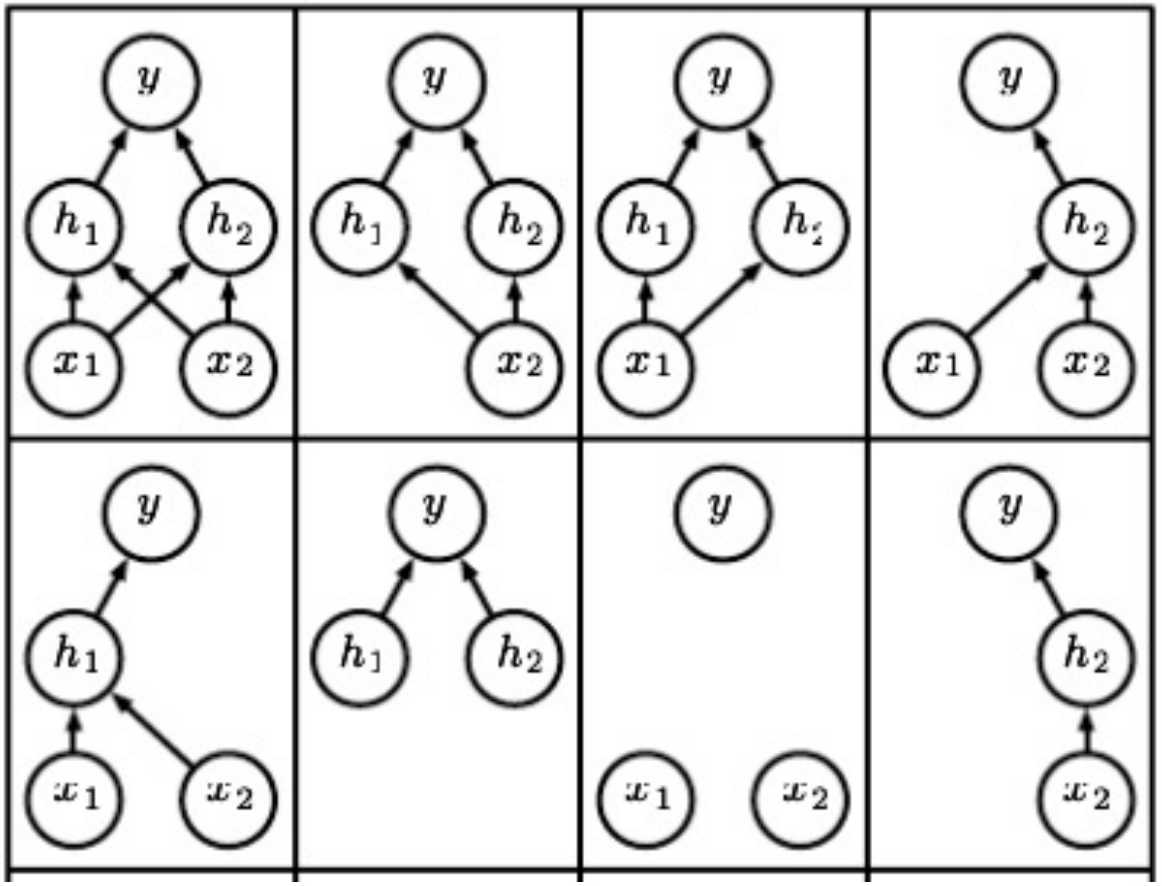


Regularization in deep nets

- We can also train an ensemble of networks with slightly different architectures

- A large ensemble:

=> a lot of weights!



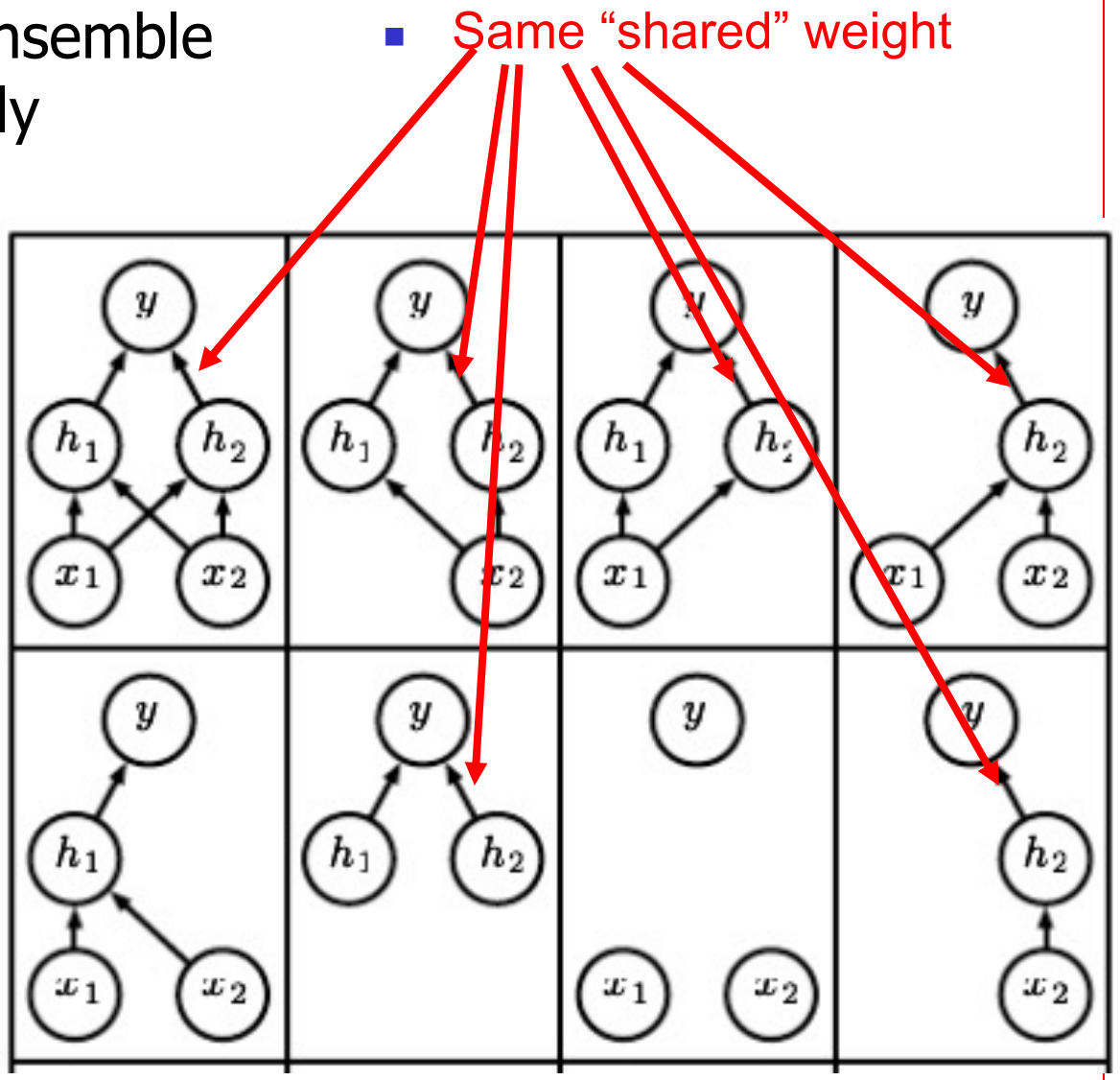
Regularization in deep nets

- We can also train an ensemble of networks with slightly different architectures

- A large ensemble:

=> a lot of weights!

- How to reduce number of weights?
  - Weight sharing



Regularization in deep nets

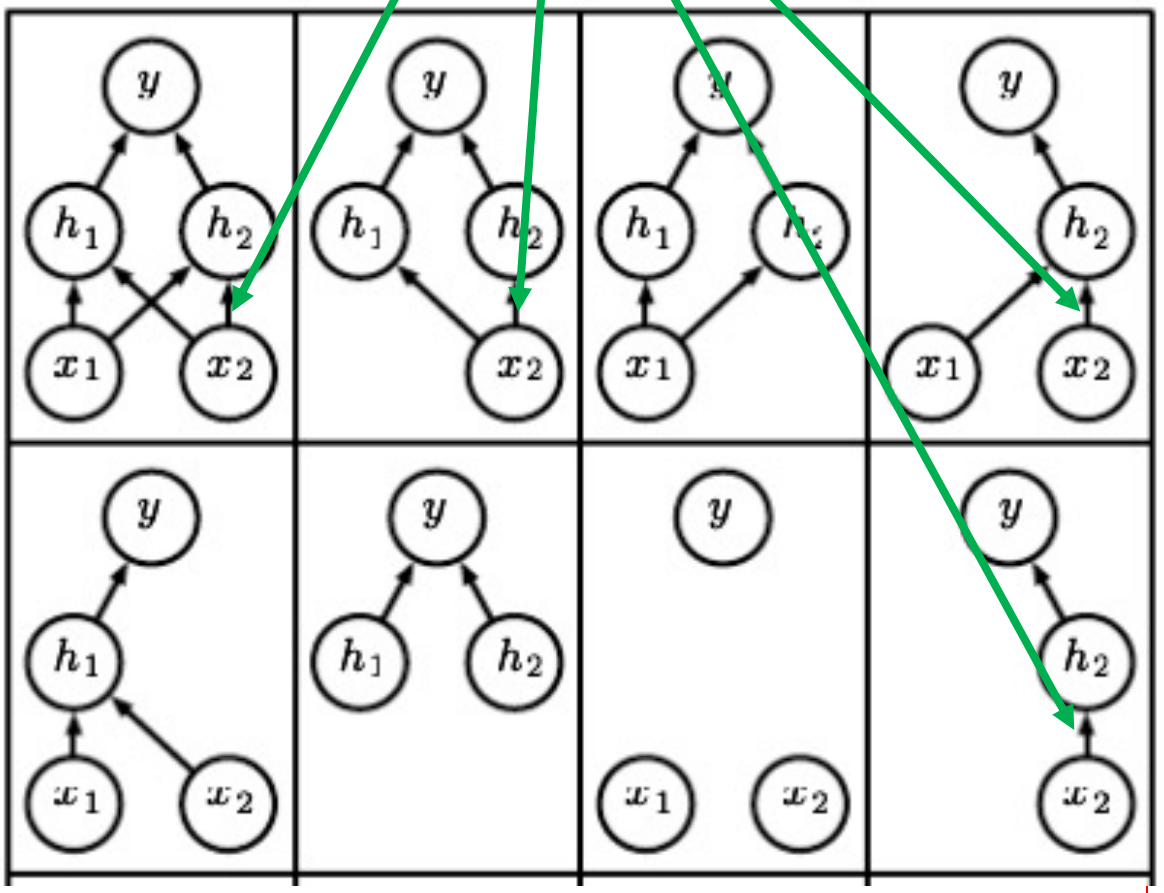
- We can also train an ensemble of networks with slightly different architectures

- A large ensemble:

=> a lot of weights!

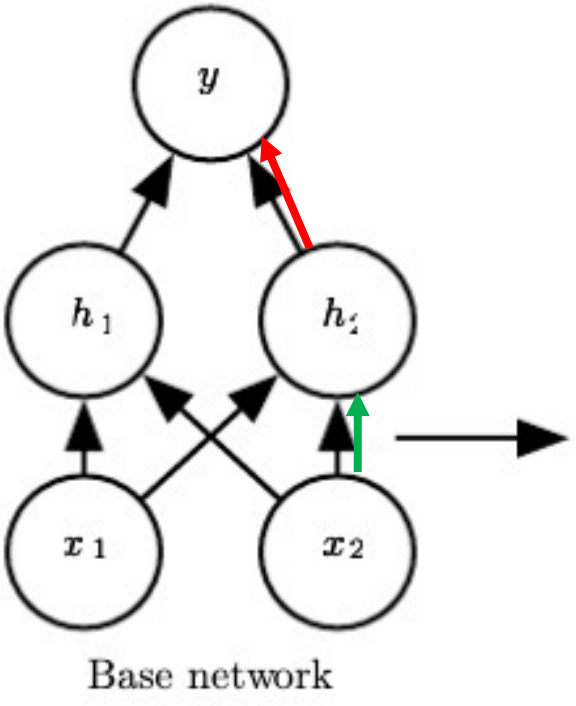
- How to reduce number of weights?
  - Weight sharing

- Same “shared” weight

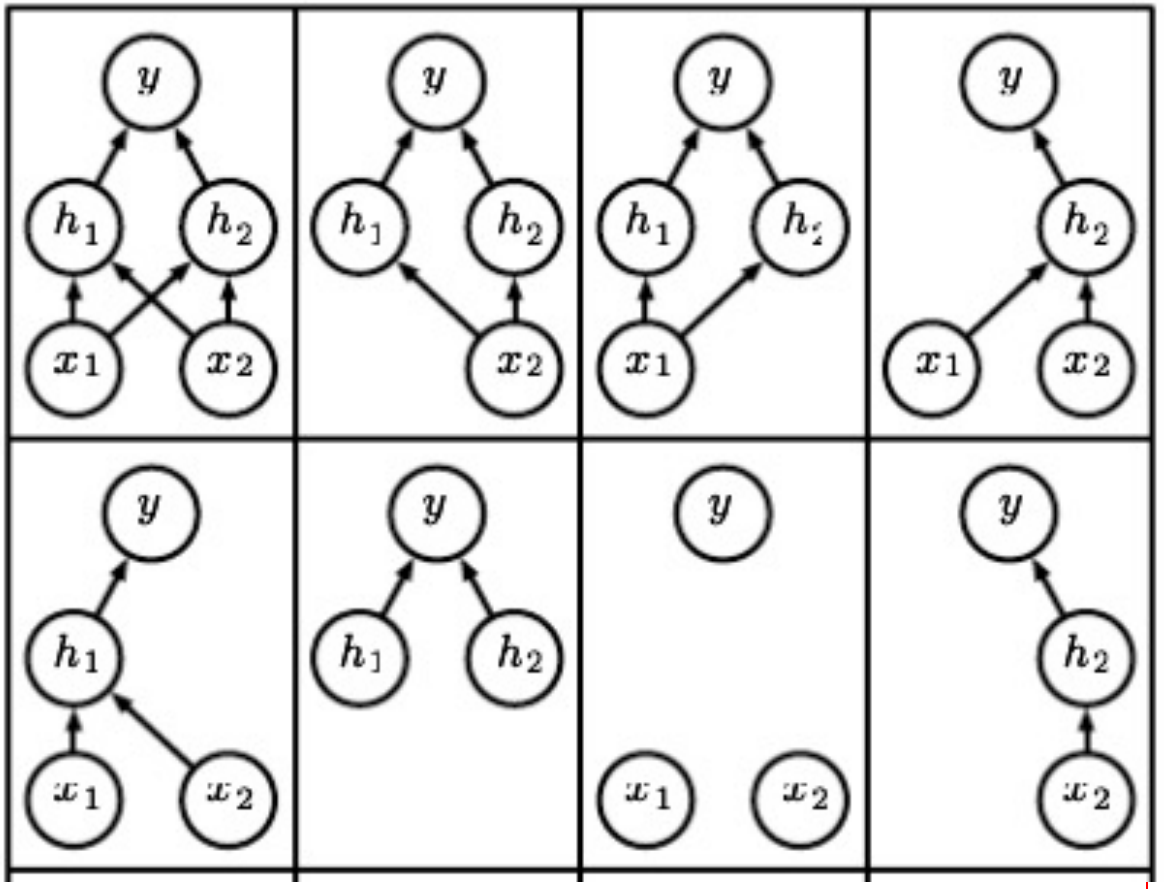


Regularization in deep nets

- How to reduce number of weights?
  - Weight sharing
- We only have 6 weights to train

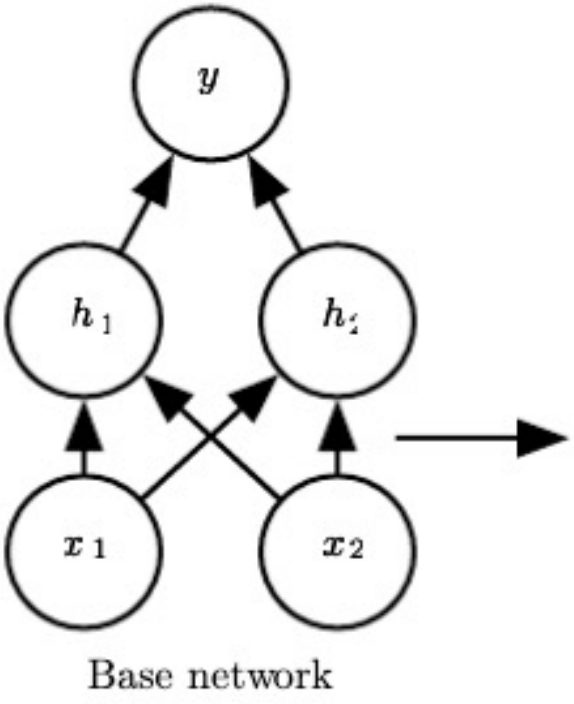


Base network

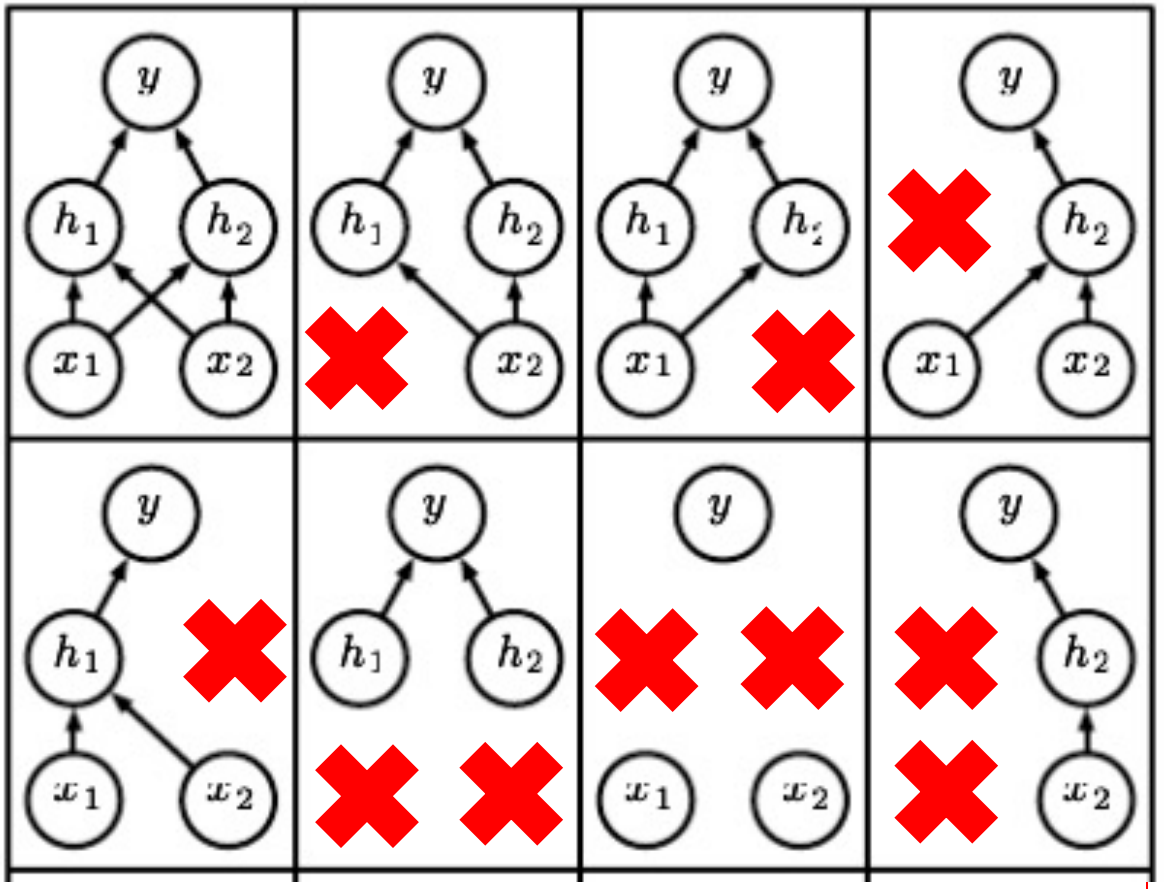


Dropout technique

- Dropout: how to train all these ensembles?
- Keep one base network with its weights, just randomly "drop" neurons
  - Multiply their output by 0

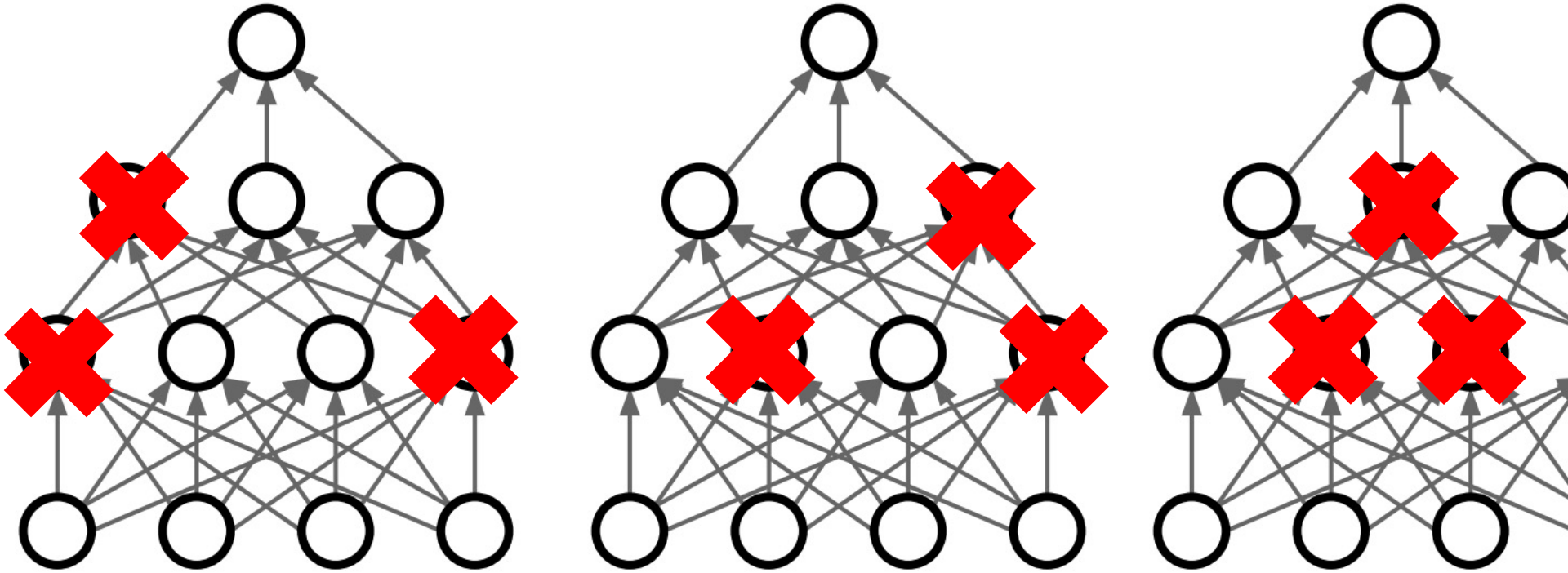


Base network



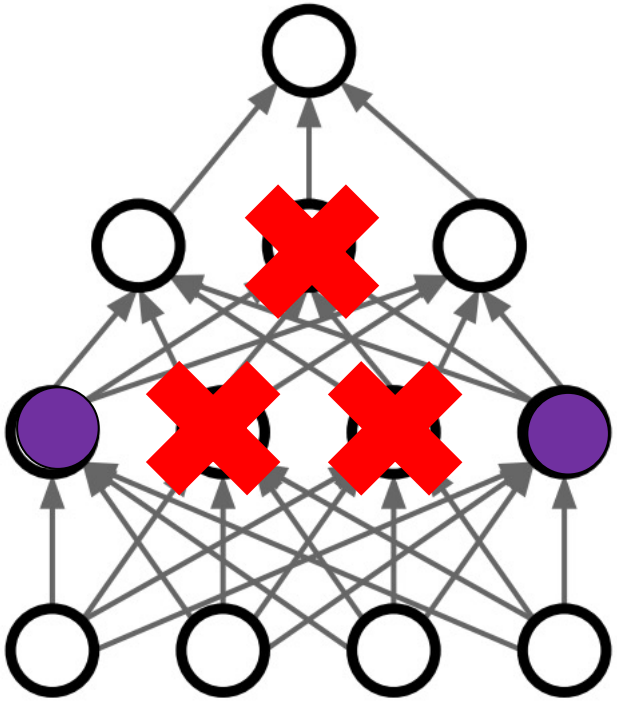
Dropout technique

- Weight sharing in an ensemble
  - Dropout – train multiple networks, each with some neurons missing
  - The edges/weights are kept the same across networks



Dropout technique

- Dropout – during testing:
  - Using the whole network "as is" is not optimal
  - During training e.g. in layer 3 we have input from two neurons
  - If during testing we use all four neurons, we'll have a higher input to layer 3 neurons
- Solution:
  - Reduce the weights after training
  - E.g. if you drop out 50% of neurons
    - Then reduce weights by 2



Summary

- Gradient descent can be encapsulated inside packages for automated differentiation
  - We just write what the model is, no need to worry about coding the details of the math needed to train it
- MSE loss and sigmoids may cause problems, use Cross Entropy loss and ReLU or similar
- Deep networks have many parameters layer after layer, can lead to problems with convergence to minimum, or to overtraining
  - Use specialized architectures: ConvNets, ResNets, Transformers
- Many techniques are being proposed to improve convergence and generalization
  - E.g. dropout, BatchNorm/LayerNorm