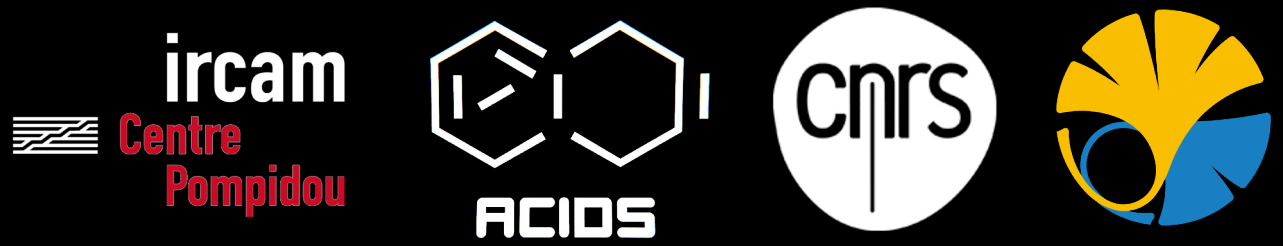


Deep Learning

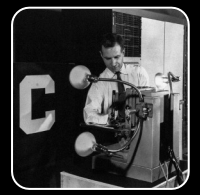
Creative Machine Learning - Course 04

Pr. Philippe Esling
esling@ircam.fr

Brief history of AI

1943 - Neuron

First model by McCulloch & Pitts (purely theoretical)



1957 - Perceptron

Actual **learning machine** built by Frank Rosenblatt

Learns character recognition analogically

1986 - Backpropagation

First to learn neural networks efficiently (*G. Hinton*)

1989 - Convolutional NN

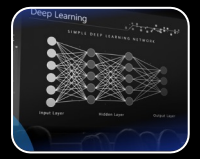
Mimicking the vision system in cats (*Y. LeCun*)

This lesson

2012 - Deep learning

Layerwise training to have deeper architectures

Swoop all state-of-art in classification competitions



Lesson #6

2015 - Generative model

First wave of interest in generating data

Led to current model craze (VAEs, GANs, Diffusion)

2012 onwards Deep learning era

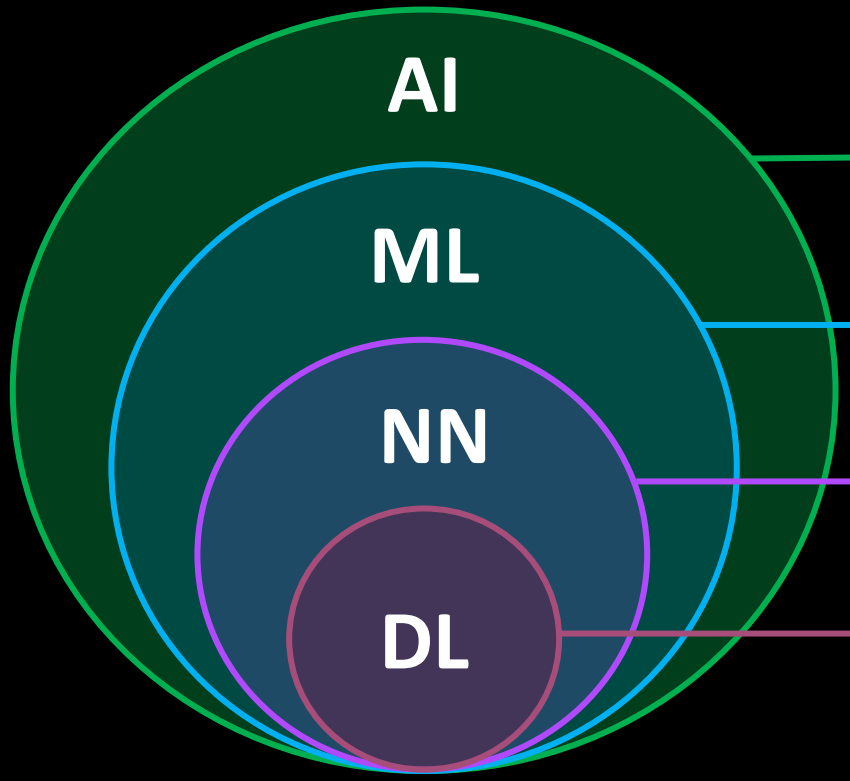
Going Machine Learning

Recalling previous discussion around machine ***intelligence***

Still wide controversies about AI (*weak vs. strong, feasible vs. fake*)

Avoiding any debate, we only discuss here *machine learning*.

- Can we automatically learn parameters of an approximation
- The most active field recently, and also very wide topics inside it



Artificial Intelligence (AI)
Any technique allowing machines to solve human tasks

Machine Learning (ML)
Learning inference models from examples

Neural Networks (NN)
Brain-inspired ML models

Deep Learning (DL)
Building (deep) hierarchies of NN representations

Deep learning

Motivation

- Supervised training is already difficult (optimization)
- Shallow models (Neural nets, *SVM*, *Boosting*)
  - Unlikely candidates for learning high-level abstractions
  - Solves quite “simple” non-linear models
  - Hard to combine this simpler models

Problems

Learning becomes complex with many layers

- Vanishing gradient problem
- Instability in the training
- Multiple local minima
- Overfitting issues

Aim to learn **deeper** architectures

We need to address those learning issues

Deep architectures ?

Insufficient depth can hurt

Experiences (past courses) tell us that few-layers models

- can only solve simple problems (even for non-linear ones)
- seem to be fastly bounded in accuracy (overfit)
- could require a very large number of dimensions

Known depths

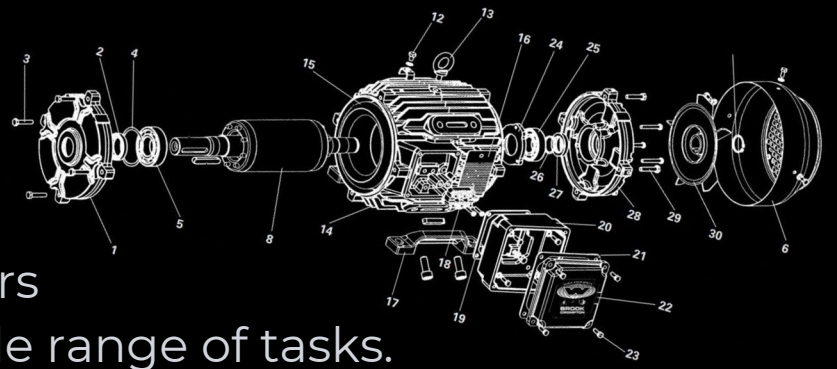
- Linear/logistic regression = **1 layer**
- SVM / Boosting = **2 layers**
- Neural Network = **3-4 layers**
  Could be of any depth but **gradient diffusion** problem
- Brain = around **10 layers**

Observations and goals

- The brain is known to have a deep architecture
- Cognitive processes seem to require multi-layer processing
- How to learn functions that represent high-level abstractions

Depth of existing approaches

Principle of compositionality



- Capture complex patterns with fewer parameters
- Better generalization and performance on a wide range of tasks.

When a function can be compactly represented by a deep architecture, it might need a very large architecture to be represented by an insufficiently deep one.

How to train this pragmatically

- Deep architectures should learn **hierarchical** representations.
- Ability to **combine simple concepts** to form more complex ones.
- Deep architectures aims at capturing **compositionality** in data.

Hierarchical representation learning

Lower layers learn local features
Higher layers capture abstract concepts.

Deep learning intuition

Composition of abstractions

Local features

Edges, corners, textures

Basic building blocks of representations.

Intermediate objects

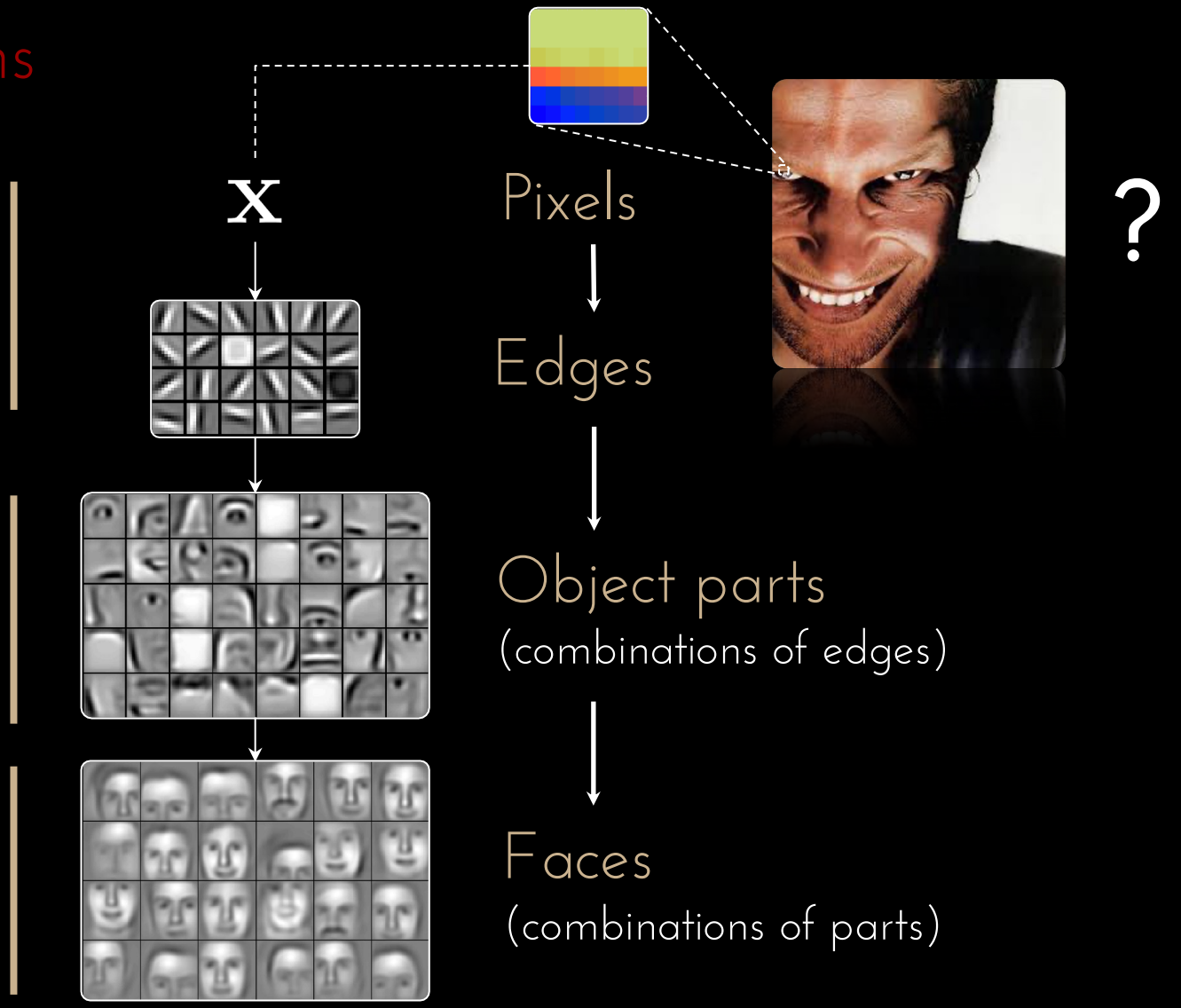
Shapes, objects, and patterns

Combinations of local features

High-Level abstractions

Scene understanding

Allow to make predictions and decisions



Representation learning

Principles of abstractions

Each level *is an abstraction*

Abstractions at one level:

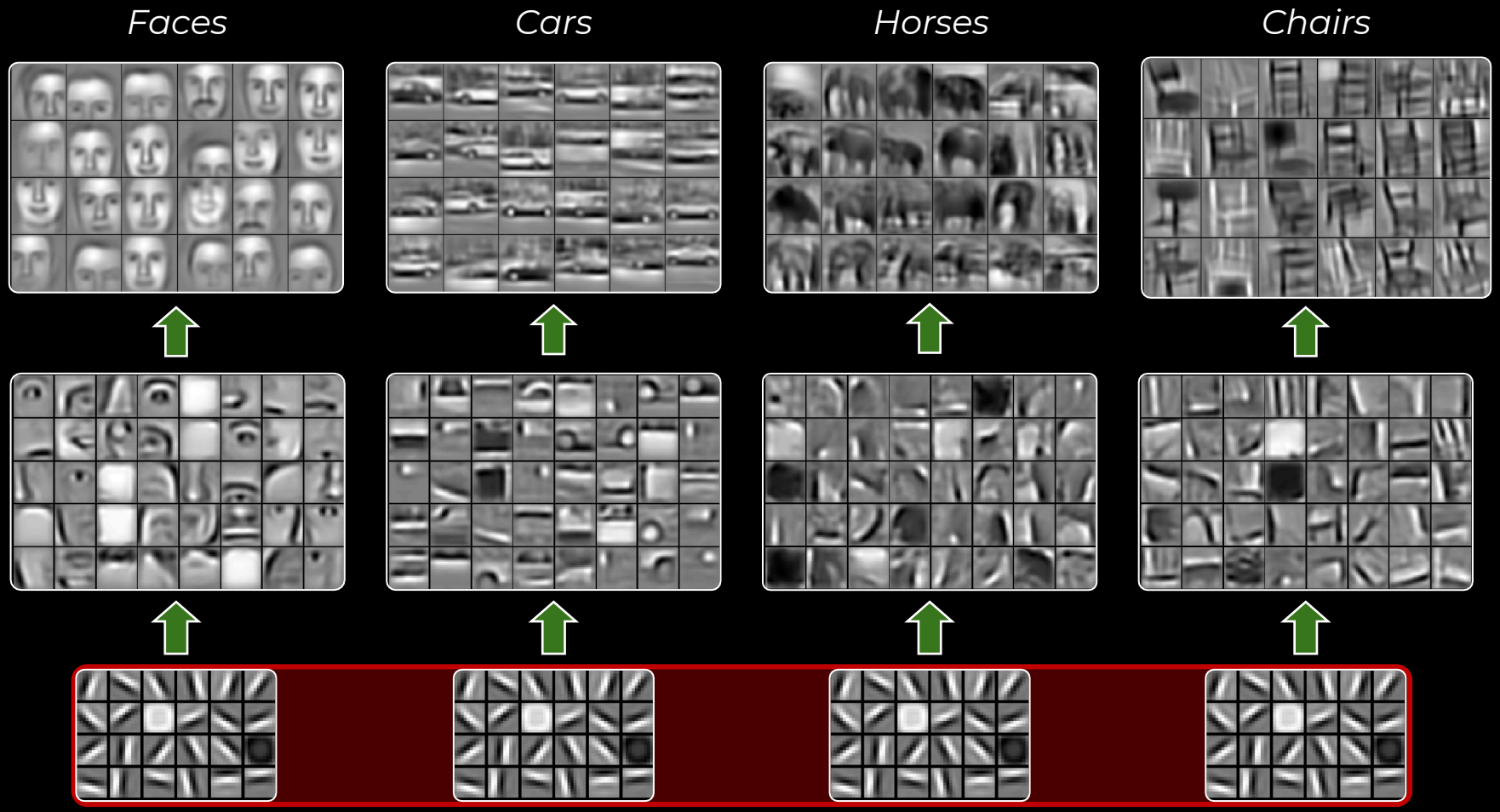
- various correlation of lower-level abstractions
- discriminating important variations

How to construct abstractions ?

- Ability of deconstructing an object
- Then reconstruct it from fewer parts
- Amounts to **learning its structure**
- Knowing higher-levels allows to re-generate

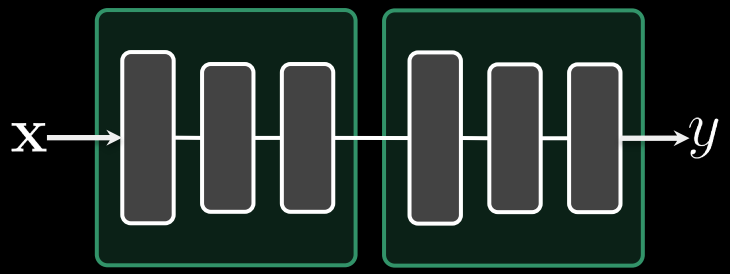
Deep learning intuitions

Another advantage of compositionality

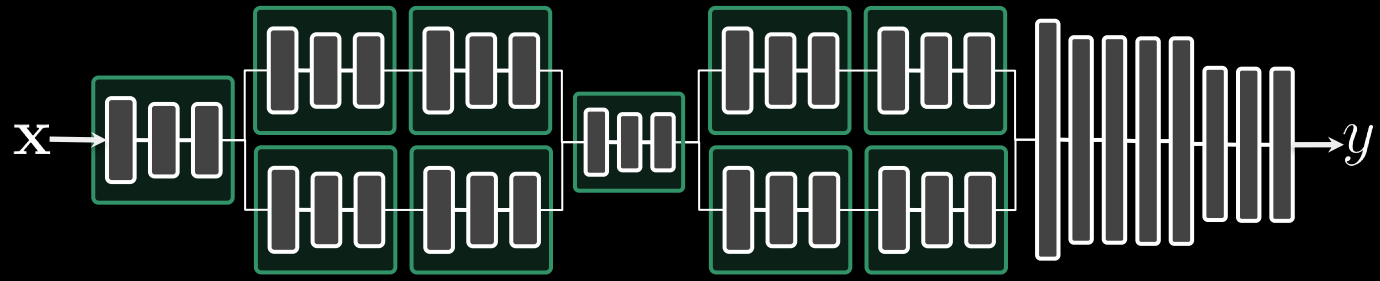


Deep architectures ?

Goal | Compose a large (deep) number of non-linear layers



Typical (x)NN



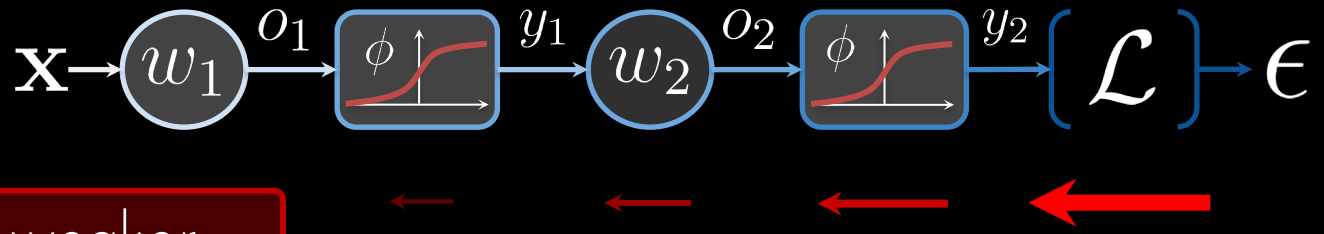
***Deep** architecture*

List of problems on our way

- Overfitting issues
- Instability in the training
- Multiple local minima

Can be addressed with adequate **regularization**

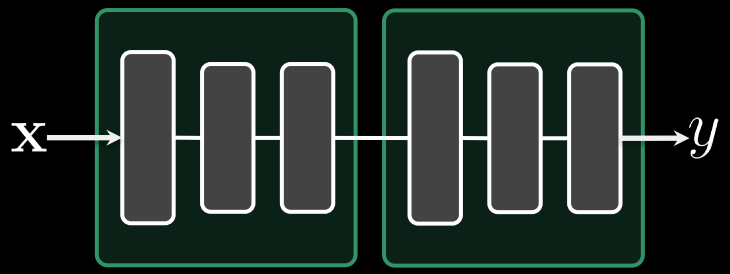
#1 - Vanishing gradient problem



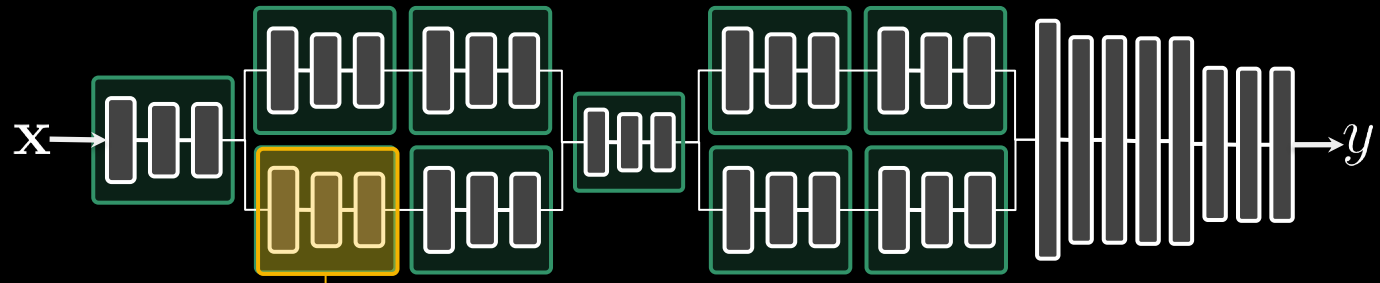
Gradient becomes increasingly weaker

Deep architectures ?

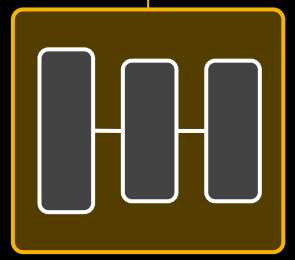
Goal | Compose a large (deep) number of non-linear layers



Typical (x)NN



Deep architecture



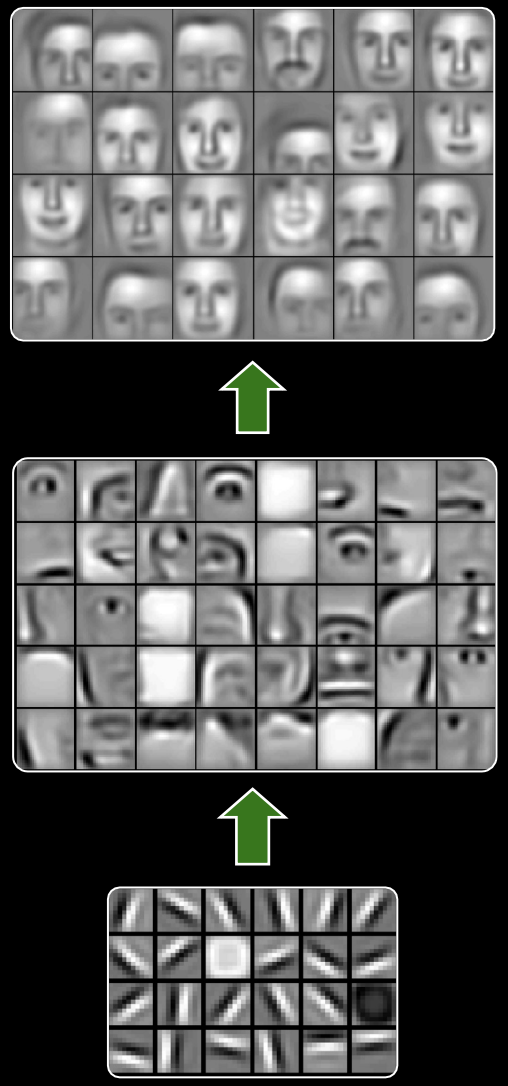
Looking at a single layer in our architecture

- Each of the layer performs a specific transform
- Are there relations between layers of different networks
- Could we learn these layers independently?

Unsupervised learning could do "local-learning"

Each module tries its best to model what it sees

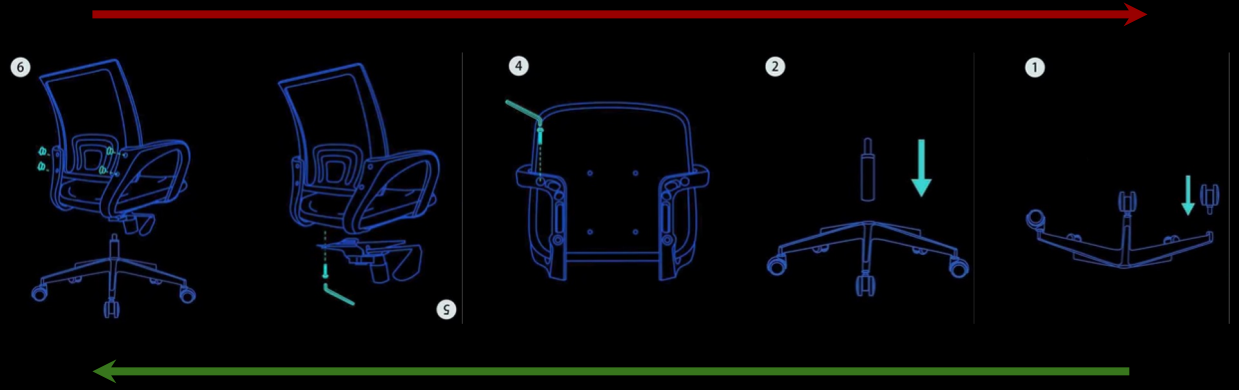
Divide and conquer (layerwise learning)



Our overarching goal

How to learn each transform separately

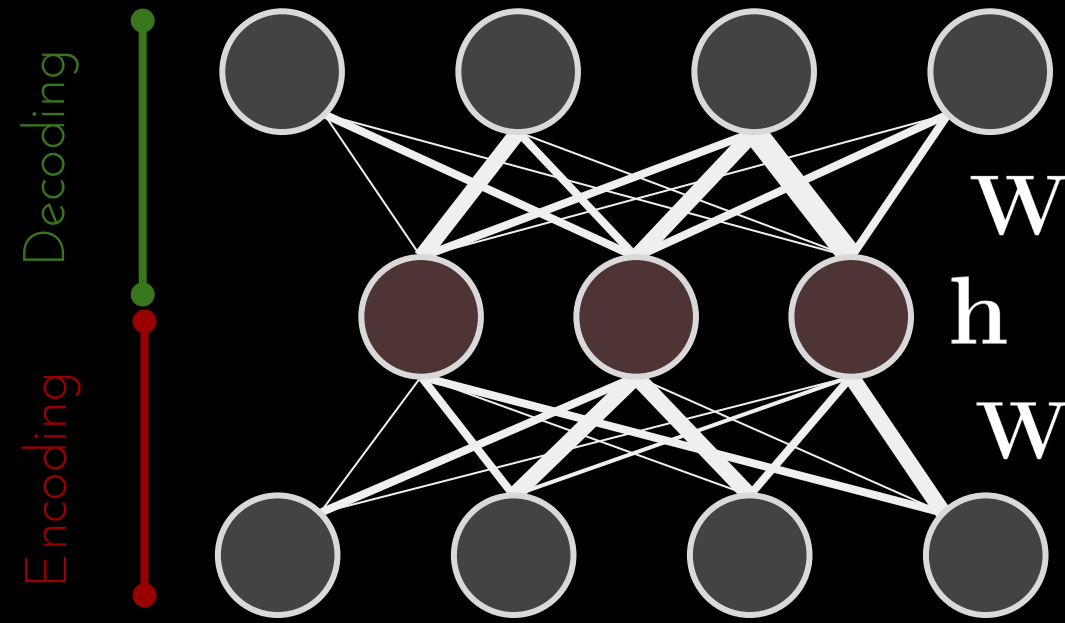
- Pass the transformed data to next layer.
- Learning our network one layer at a time
- Starting from the bottom layer
- Guarantees that each layer improves ?
  - Need to know higher layers to learn lower layers?



Solution: Re-representing the data

Auto-encoders

$$d : \mathbb{R}^{d_h} \rightarrow \mathbb{R}^{d_x} \qquad \mathbf{y} = d(\mathbf{h_x}) = \sigma(\mathbf{W}_d\mathbf{h_x} + \mathbf{b}_d)$$



Unsupervised learning

$$\min \left[\mathcal{L}(\mathbf{x}, \mathbf{y})\right]$$

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = \| \mathbf{x} - \mathbf{y} \|^2$$

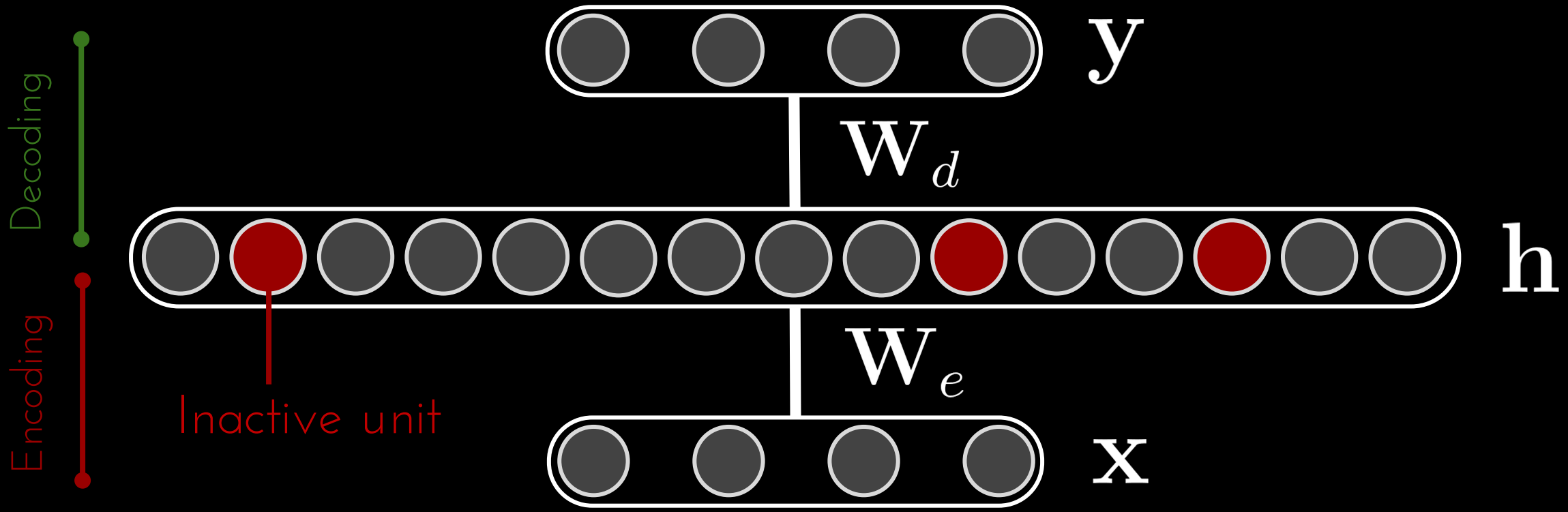
$$e : \mathbb{R}^{d_x} \rightarrow \mathbb{R}^{d_h} \qquad \mathbf{h_x} = e(\mathbf{x}) = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e)$$

Problem ?

$$\mathcal{L}_{AE}(\theta) = \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, d(e(\mathbf{x}))) + \beta \sum_i \mathcal{D}_{KL}\left[\rho, \hat{\rho}_i\right]$$

Sparsity constraint

Sparse auto-encoders



Measuring sparsity

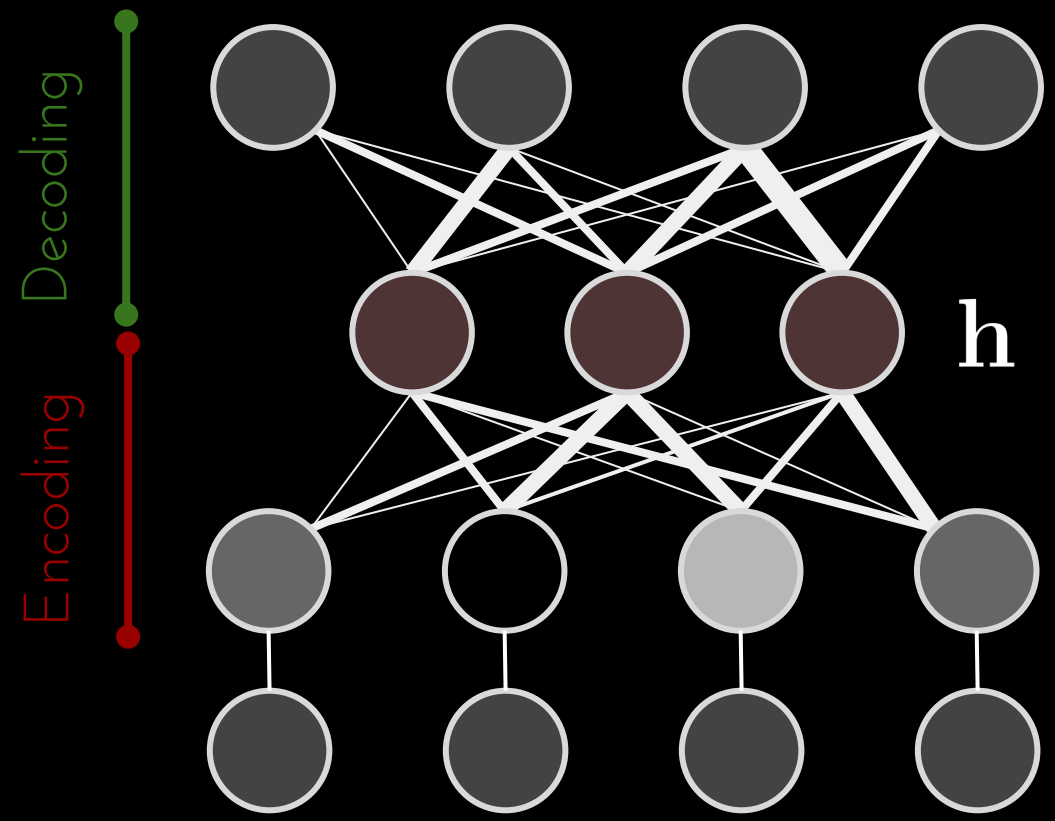
Kullback-Leibler divergence (later in course)

$$\mathcal{D}_{KL}\left[\rho, \hat{\rho}_i\right]$$

Incentive for the activation patterns $\hat{\rho}_i$

To follow a desired target pattern ρ

Denoising auto-encoders



Keep the same overall structure

Add noise to the input

$$\tilde{\mathbf{x}} = \mathbf{x} + \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$\min \; [\mathcal{L}(\mathbf{x}, \mathbf{y})]$$

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = \| \mathbf{x} - \mathbf{y} \|^2$$

Still try to reconstruct clean input

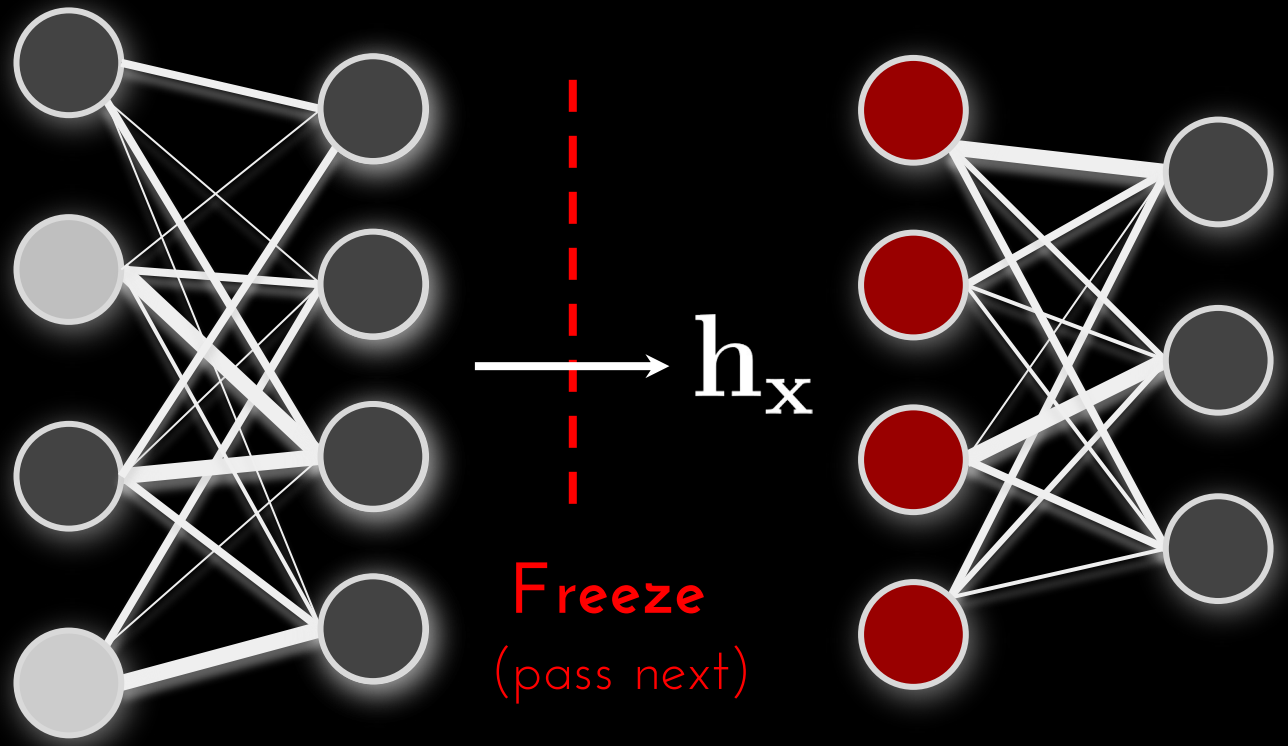
Implicit ***denoising*** mechanism

$$\mathcal{L}_{DAE}(\theta) = \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, d(e(\tilde{\mathbf{x}})))$$

Inherent identity learning prevention

Stacked auto-encoders

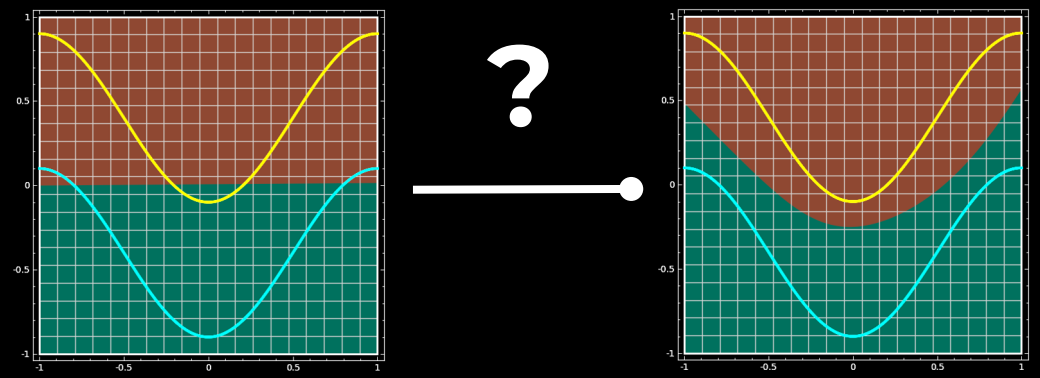
Layer-wise learning by stacking representations

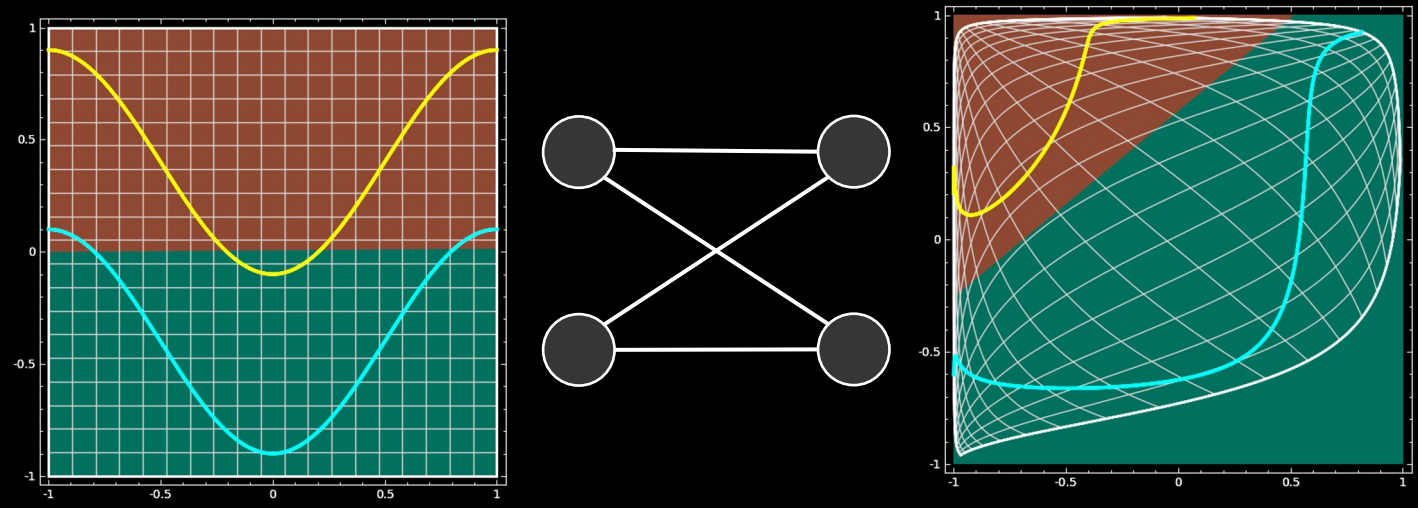


$$\mathbf{h_x} = e(\mathbf{x}) = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e)$$

Networks as space transform

Extending our geometrical insight to **non-linearities** ?





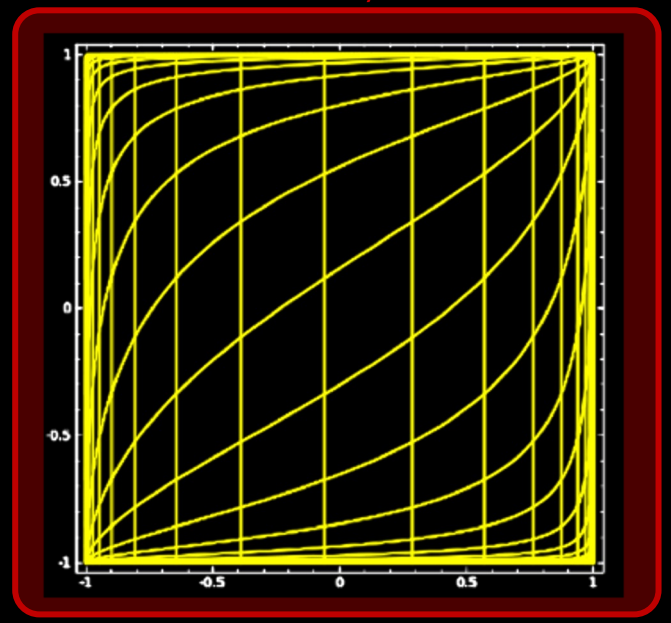
$$\mathbf{o}^l \in \mathbb{R}^2 \quad \mathbf{o}^{l-1} \in \mathbb{R}^2$$

Interpreting our previous equation

$$\mathbf{o}^l = \phi\left(\mathbf{W}^l * \mathbf{o}^{l-1} + \mathbf{b}^l_m\right)$$

Affine transform

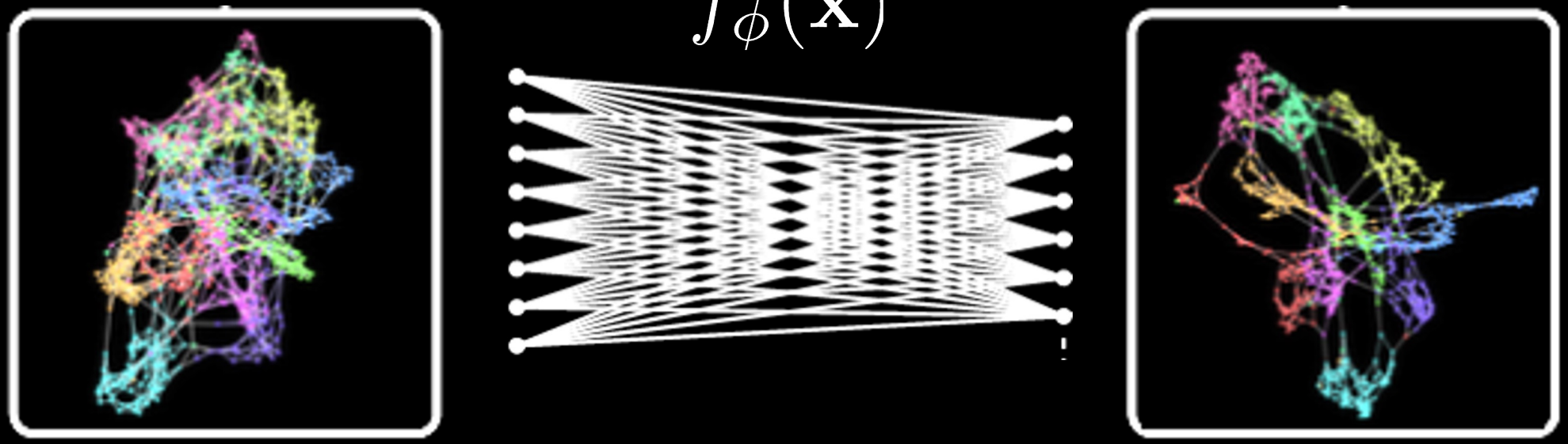
Non-linearity



Networks as space transform

This idea extends even to very complex spaces

$$f_\phi(\mathbf{x})$$



Our goal is to find these mathematical unfolding
In order to provide **new ways of representing the data**

Networks as **representation learning**

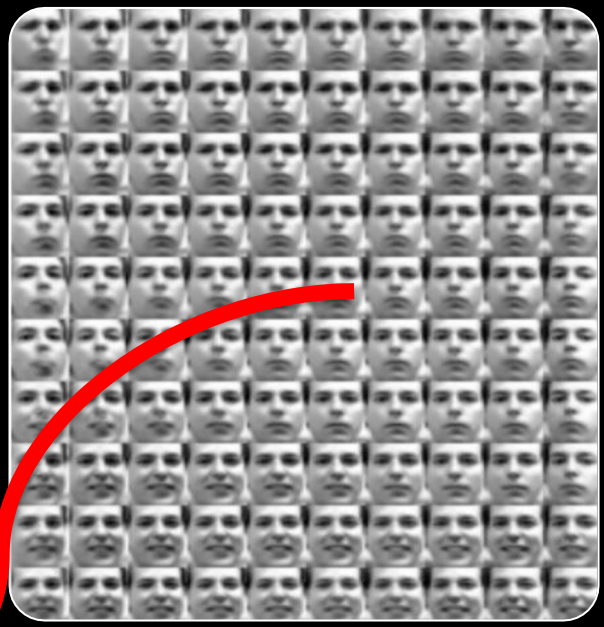
Representation learning

This concept forms the basis of **latent representation**

$f_\phi(\mathbf{x})$

$p_\phi(\mathbf{z}|\mathbf{x})$

emotion

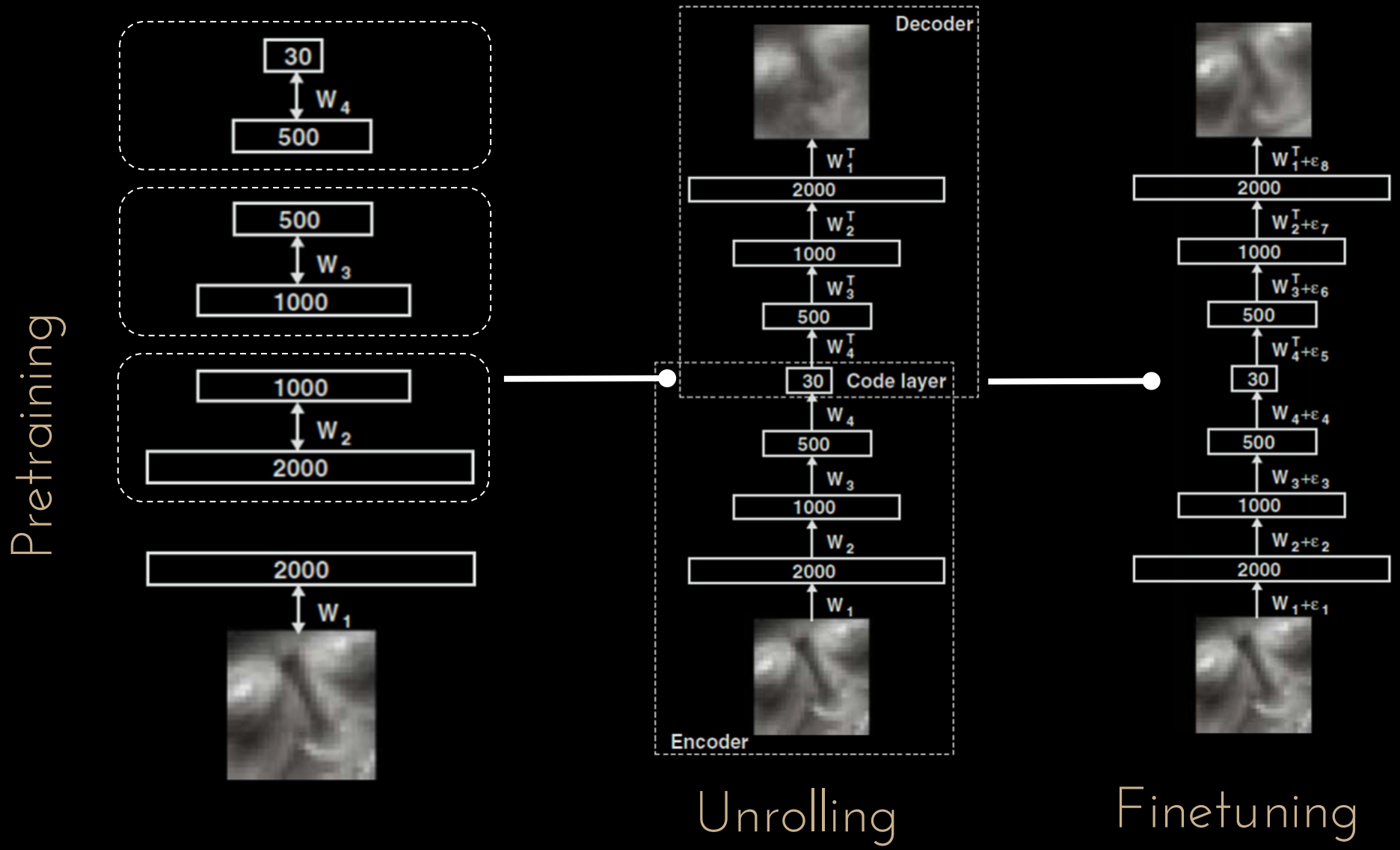


f^{-1}

orientation

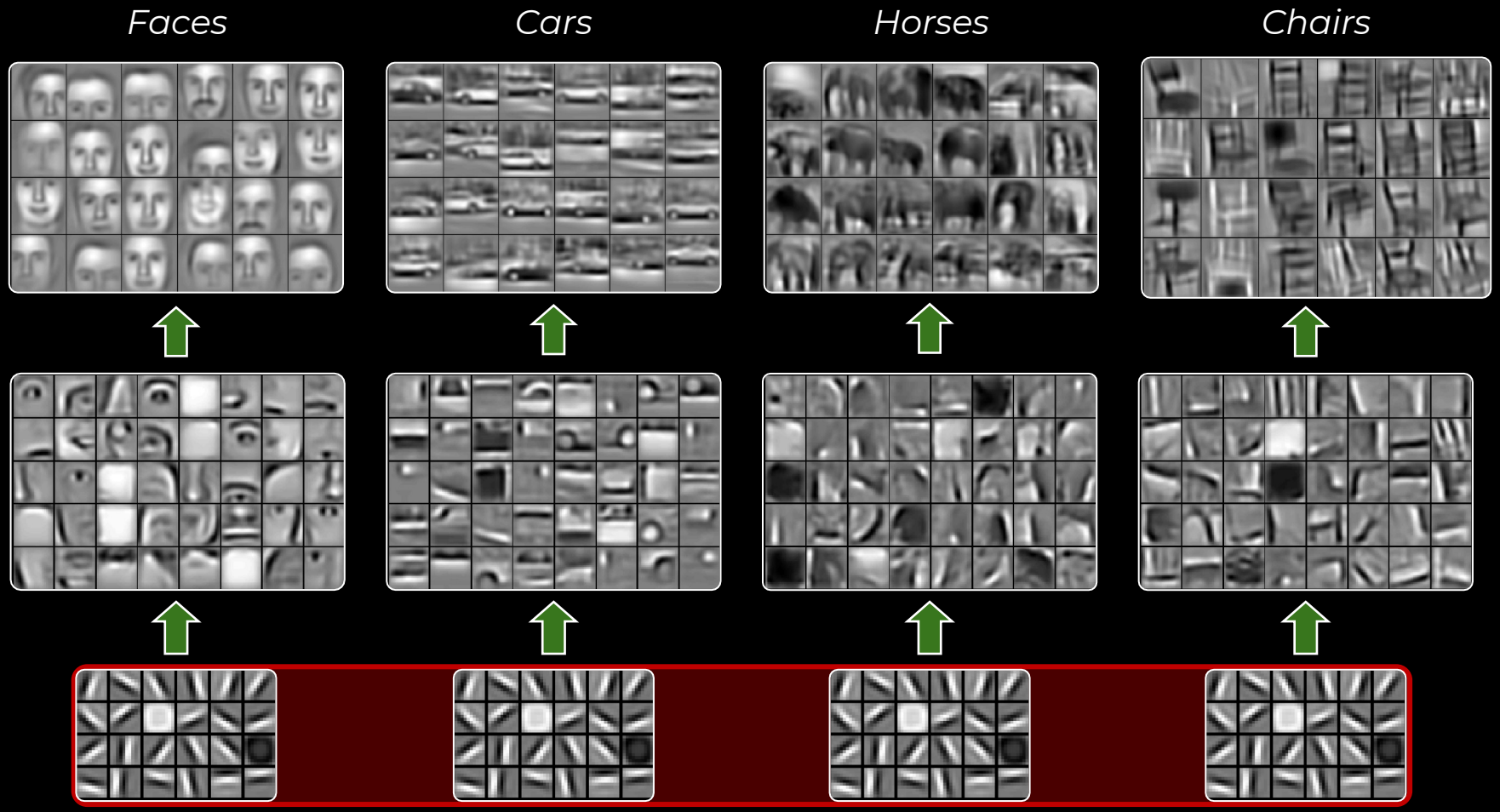
$q_\theta(\mathbf{z}|\mathbf{x})$

Pre-train and fine-tune (auto-encoders)



Deep learning intuitions

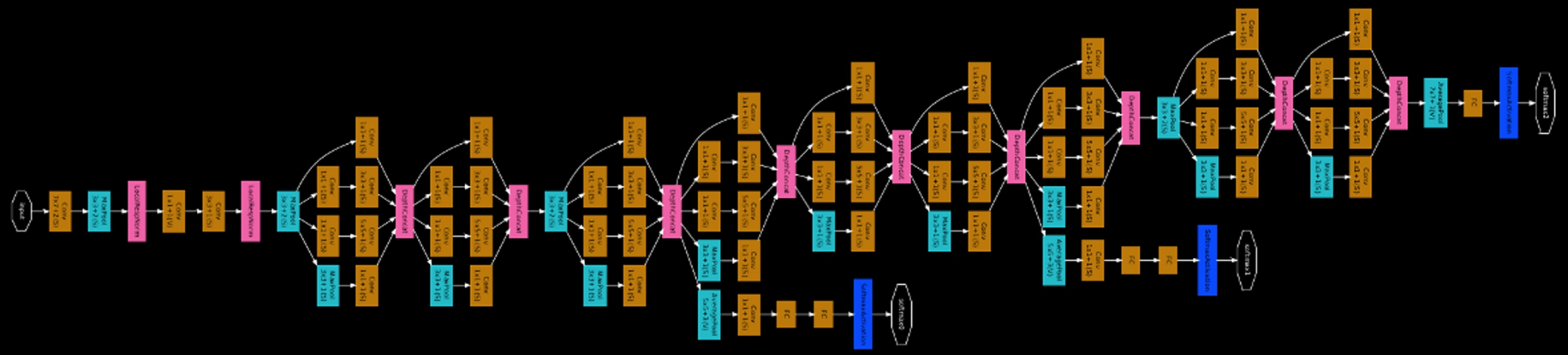
Another advantage of compositionality



Modern version

That was true 5 years ago

Now we are just back with



Major deep improvements

Targets

Solve the vanishing gradient problem
Better regularize the training

"ReLU" (Rectified Linear Unit)

$$g(\mathbf{x}) = \max(0, \mathbf{x})$$

- Gradient doesn't vanish

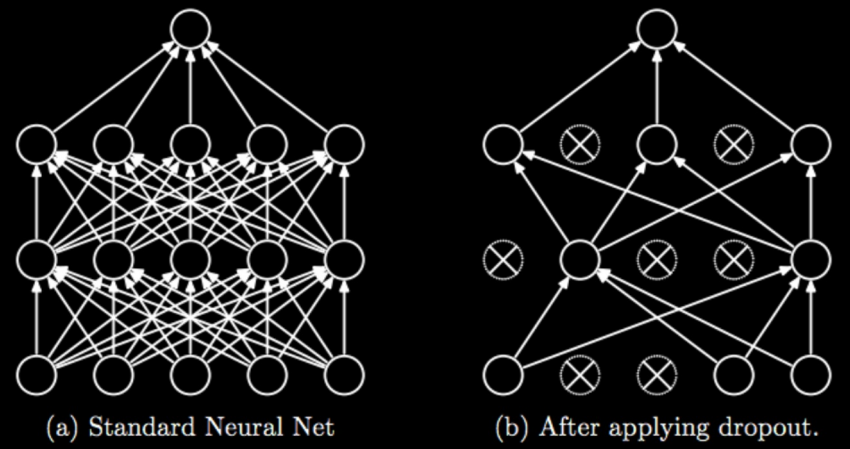
Major **theoretical** changes

"Dropout" regularization

- Avoids shared processing in units
- Reduces overfitting

Batch Normalization (next slide)

- Ensure Gaussian at each layer
- Speeds up training



(a) Standard Neural Net (b) After applying dropout.

Major **technical** changes

Fast GPU implementations.
Largely bigger data

Convolutional Neural Networks (CNN)

Extending neural networks with convolutions

- Inspired by behavior of the visual cortex in cats (V1)
- CNN **replace the affine transforms (dense layers) by convolutions**
- Allows processing data with a grid-like or array topology

Key concepts

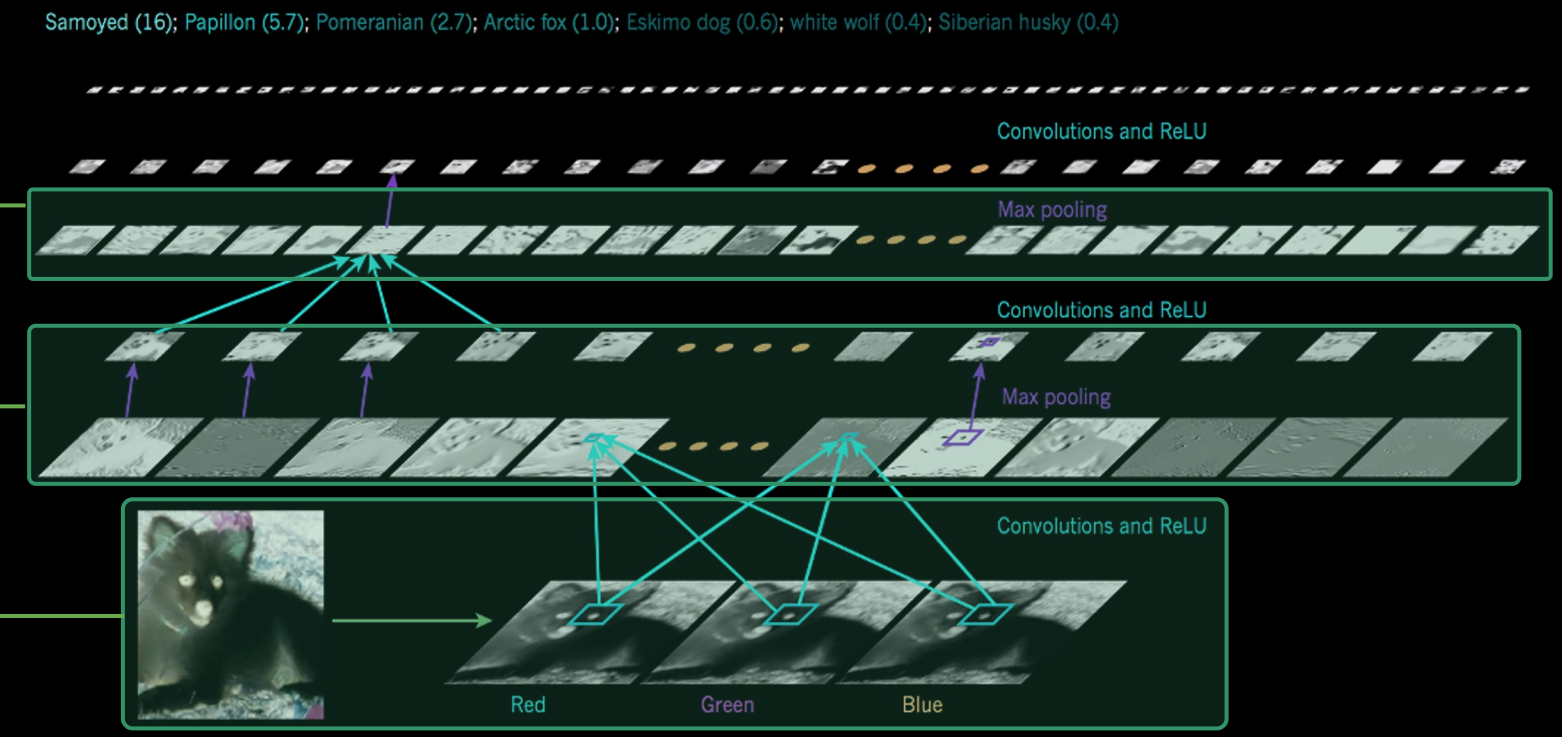
Kernels are **convolved across all channels and summed**

Convolutional layer produces **feature maps** (for each kernel)

$$\mathbf{F} \in \mathbb{R}^{K \times N \times M}$$

Input has several **channels**

$$\mathbf{X} \in \mathbb{R}^{C \times H \times W}$$

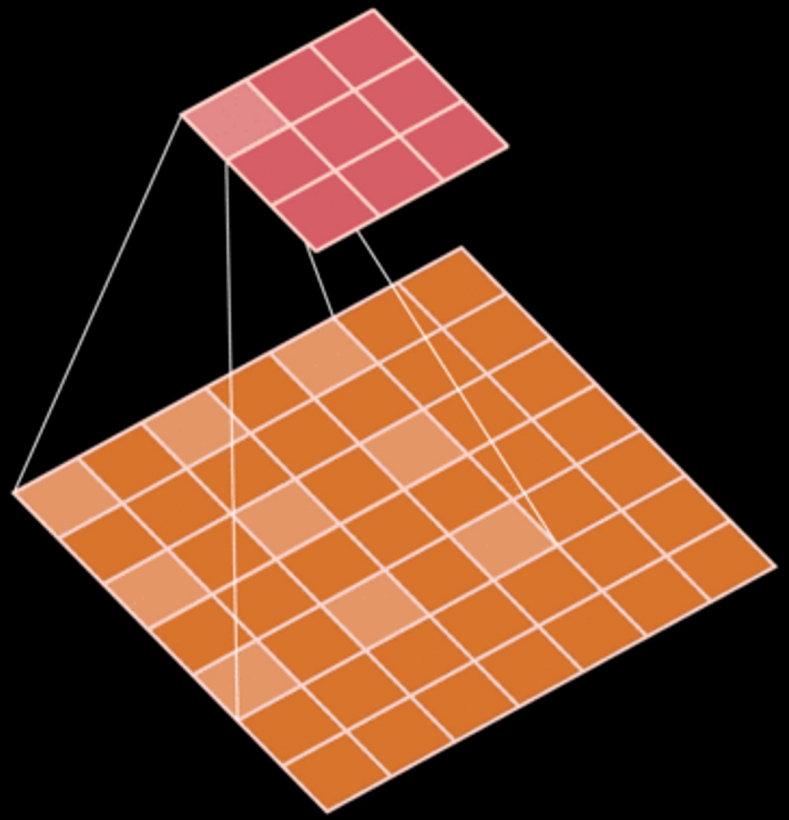


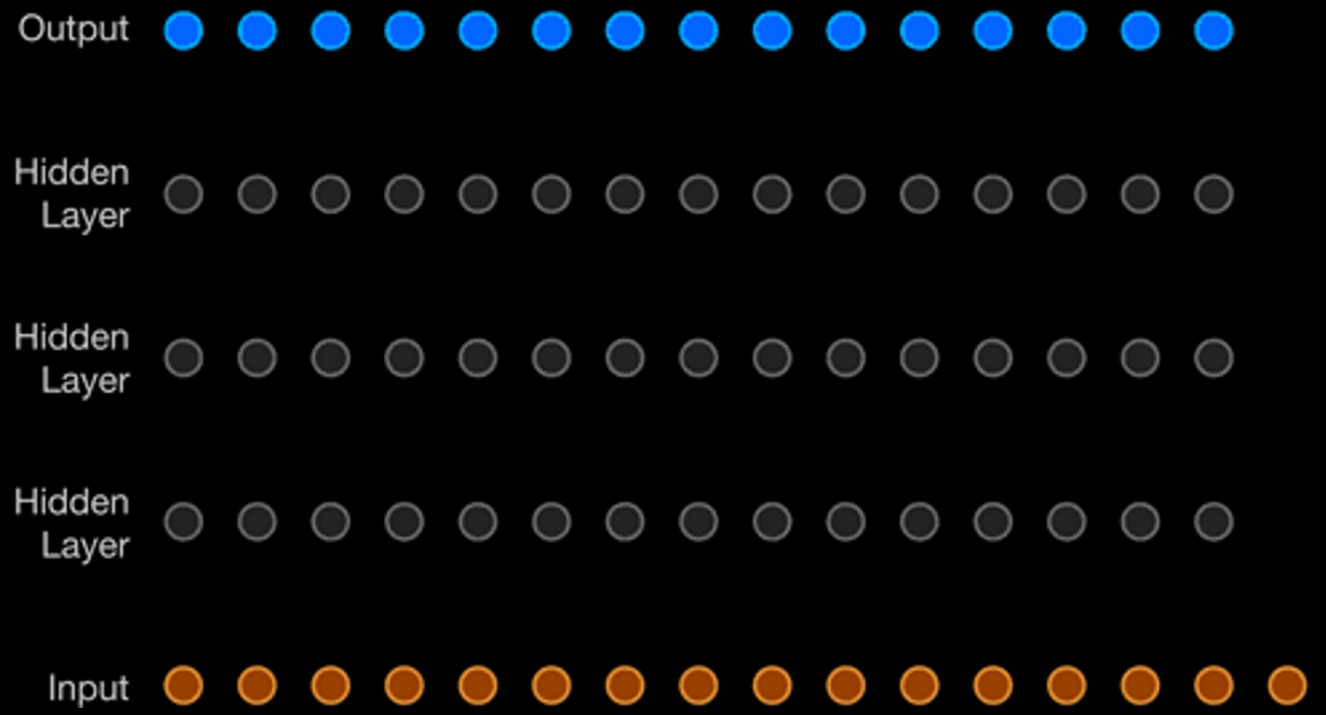
Advances in convolutions

Dilation system particularly efficient

Increased receptive field

dilation





Batch normalization

Regularization

Improve training by **normalizing the inputs of each layer**.
Mitigates vanishing and exploding gradient and allow higher learning rates.
Very successful regularization technique (**deep learning era**)

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\mathrm{Var}[\mathbf{x}] + \epsilon}}$$

$\mathbf{x}$	input to a layer
$\tilde{\mathbf{x}}$	normalized input
$\mathbb{E}[\mathbf{x}]$	mean of the batch
$\mathrm{Var}[\mathbf{x}]$	variance of the batch
ϵ	small constant

Compute the mean and variance of the inputs over a mini-batch
Normalize the inputs to **zero mean and unit variance**.

$$\tilde{\mathbf{x}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

Learnable parameters

Introduce learnable parameters to then scale and shift the normalized inputs

$$\hat{\mathbf{x}} = \gamma\tilde{\mathbf{x}} + \beta$$

Interpretation

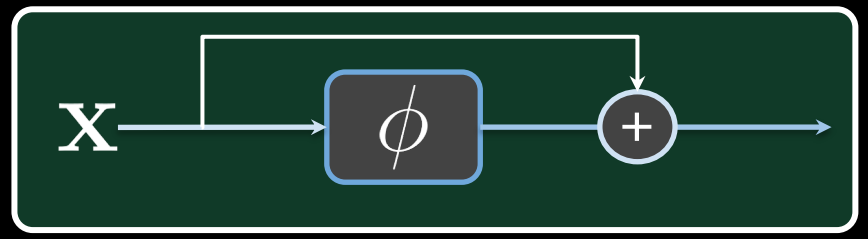
Reduces internal covariate shift (change in the input distribution).
Reduces dependence on the gradient magnitude (larger learning rates).
Batch Normalization regularizes the model and reduces overfitting.

Can be applied to any type of layer

Residual Networks

Motivation: Solving vanishing and exploding gradient in deep networks.
Key idea: Add a skip connection that allows the gradient to bypass layers.

Residual blocks $\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{x}$



Identity shortcut — No additional parameters, only direct connection
$\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{x}$

Projection shortcut — Linear projection to match potentially changing dimensions
$\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{W}_s\mathbf{x}$

Stacking residual blocks — Residual blocks are stacked to create deep ResNets.

Example: *ResNet-50, ResNet-101*, and *ResNet-152*, where the numbers indicate the total number of layers.

Benefits
- Improved gradient flow through the network.
- Mitigates the vanishing/exploding gradient problem.
- Enables training of very deep neural networks.

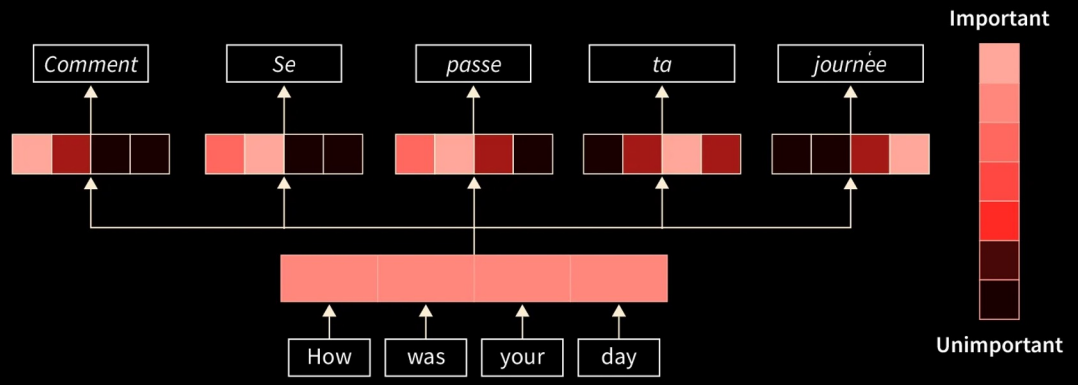
Same idea obviously applies to convolutional networks

Attention layers

Powerful technique to capture dependencies between different elements in a sequence.
Particularly useful in *machine translation*, *question answering*, and *image captioning*.

Key idea

Weigh the importance of each input element in the context of a given output.
Assign a score to each input element, then compute a weighted sum.



Attention function

Takes as input query q and key-value pairs (K, V)

1. Compute attention scores $a_i = f(q, K_i)$
2. Normalize using softmax $\alpha_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$
3. Compute output as weighted sum $o = \sum_i \alpha_i V_i$

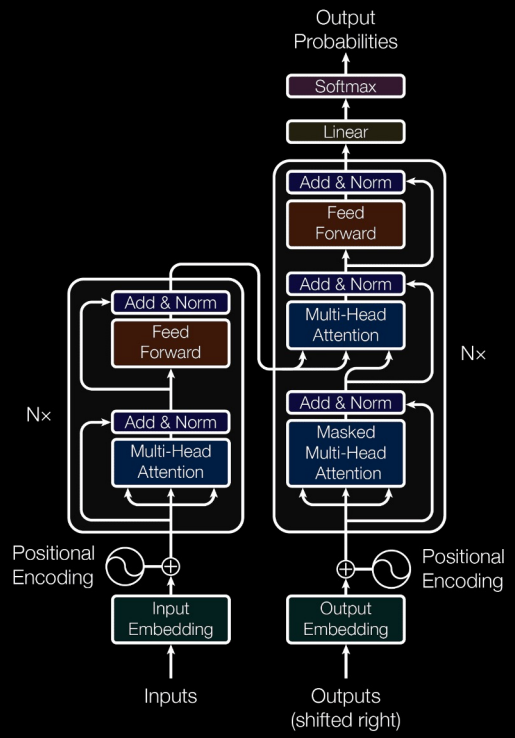
Scoring functions

Dot product $f(q, K_i) = q \cdot K_i$

Scaled dot product $f(q, K_i) = \frac{q \cdot K_i}{\sqrt{d_k}}$

Transformers (*ChatGPT*)

Transformers are a class of models composed of attention layers
Popular as they capture long-range dependencies in sequence data.



Transformer architecture

Encoder and decoder with identical layers.
Each layer has two main components

Multi-head self-attention
Weigh the importance of each input in the context

Position-wise feed-forward
Two linear layers with ReLU activation

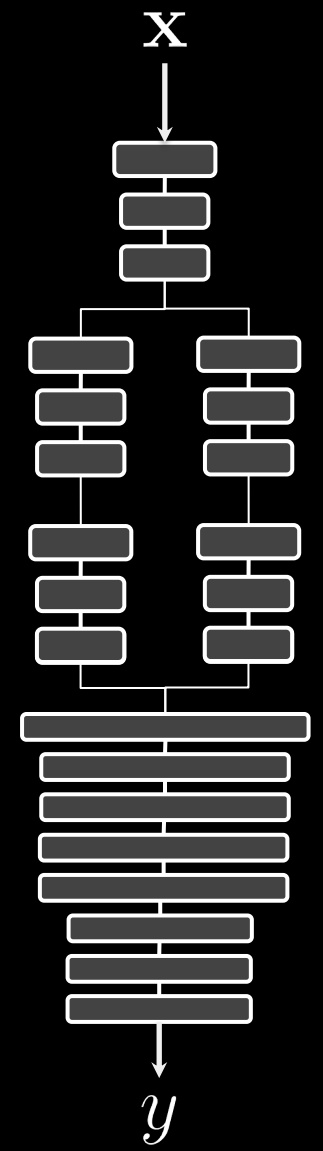
Positional encoding

Provide information about position of each element in the input
Positional encoding is added to the input embeddings.

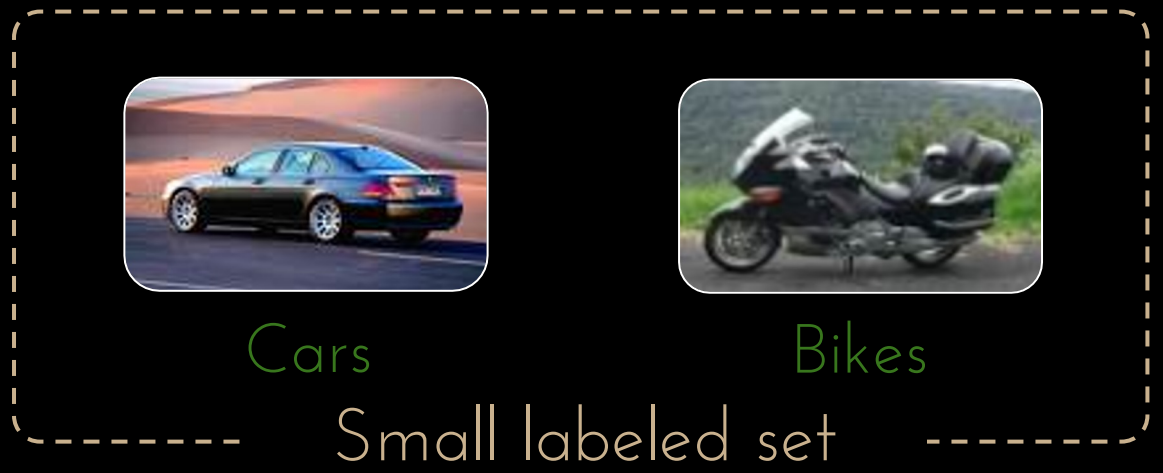
Supervised learning

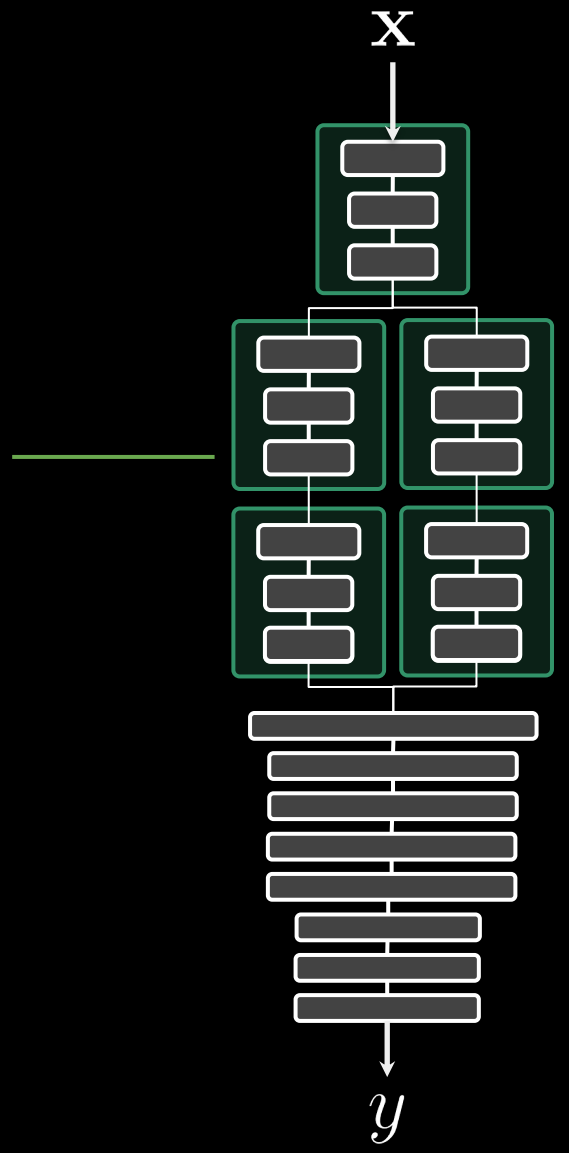
Testing time

?



Semi-supervised learning

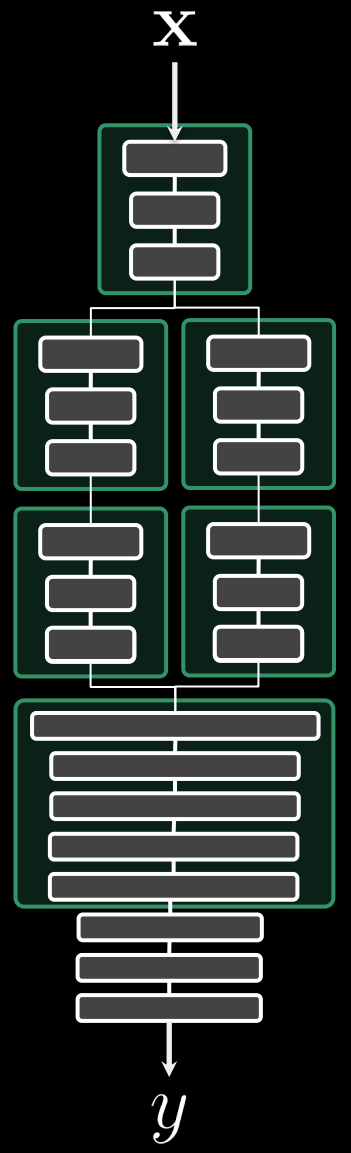




Transfer learning

Random unlabeled

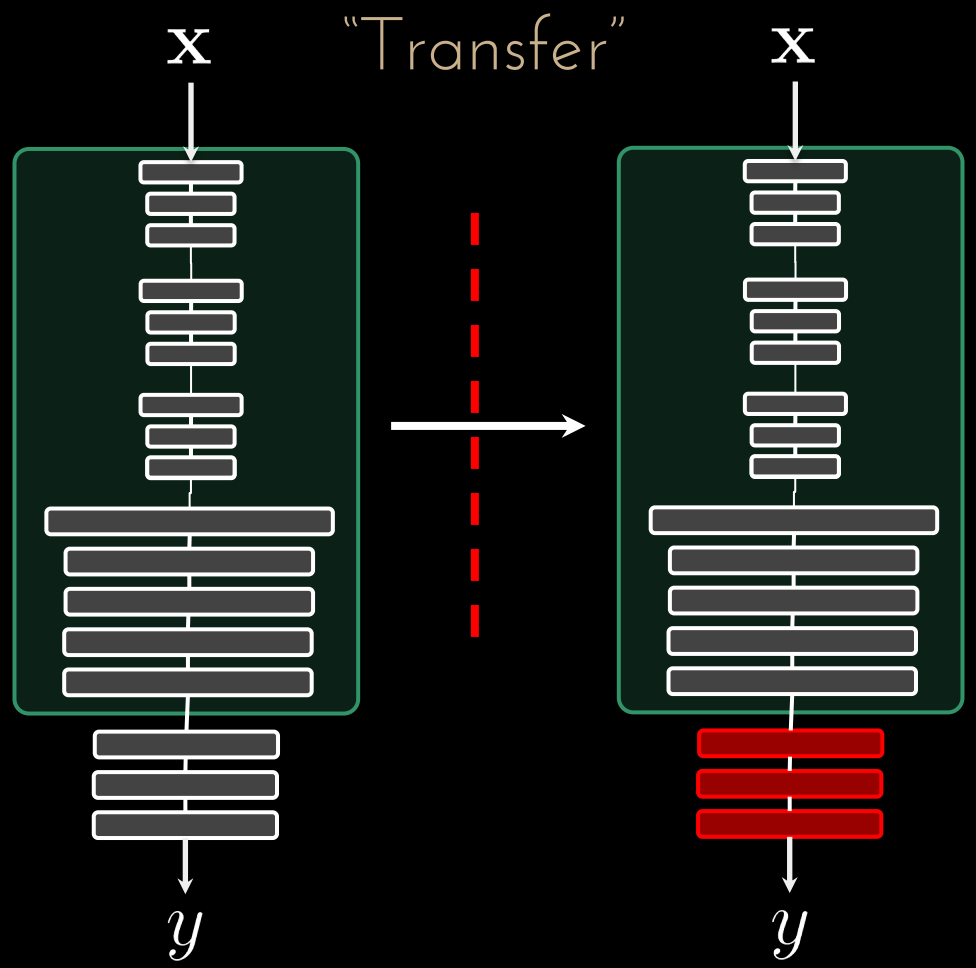
Small labeled set



We can learn with any data from the same domain

Transfer learning

Pretrain on large set



Train last layers on task

Deep learning frameworks

- Modern tools for deep learning
- Make it easy to implement neural networks
- Often used components are ready-to-use
  - Linear, convolution, recurrent, attention

Many frameworks available:

Torch (2002), Theano (2011), Caffe (2014), TensorFlow (2015), PyTorch (2016)

TensorFlow

TensorFlow

- ✓ Mature and extensive ecosystem
- ✓ Strong industry support (Google)
- ✓ Good set of tools for visualization (TensorBoard)
- ✓ Very good for **deployment**

- x Often counterintuitive architecture
- x Longer learning curve
- x Usually verbose

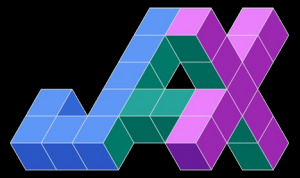
PyTorch

Pytorch

- ✓ Dynamic computation graphs
- ✓ Very large implementation flexibility
- ✓ Strong GPU support and extensibility
- ✓ TorchScript for easy deployment
- ✓ Excellent **research community**

- x Some lacking options (toolboxes)
- x Less capabilities for deployment (platforms)

JAX



JAX

- ✓ Functional programming
- ✓ Excellent automatic differentiation (Hessian)
- ✓ GPU and TPU support
- ✓ XLA Compiler
- ✓ Excellent **mathematical closeness**

- x Less mature ecosystem
- x Sometimes complex to understand and debug

Outline of the course

1. Basic concepts
    a. Tensors, parameters, modules
    b. Basic mathematics
2. Write a model
    a. Using the simple nn layers
    b. Defining your own modules
    c. Mixing it up
3. Data and optimization
    a. Writing your own dataloader
    b. Optimizing the networks
4. Convolutional model

Basic concepts

torch.Tensor
similar to numpy.array

nn.Parameter
contain *Parameters* and define functions on input *Variables*

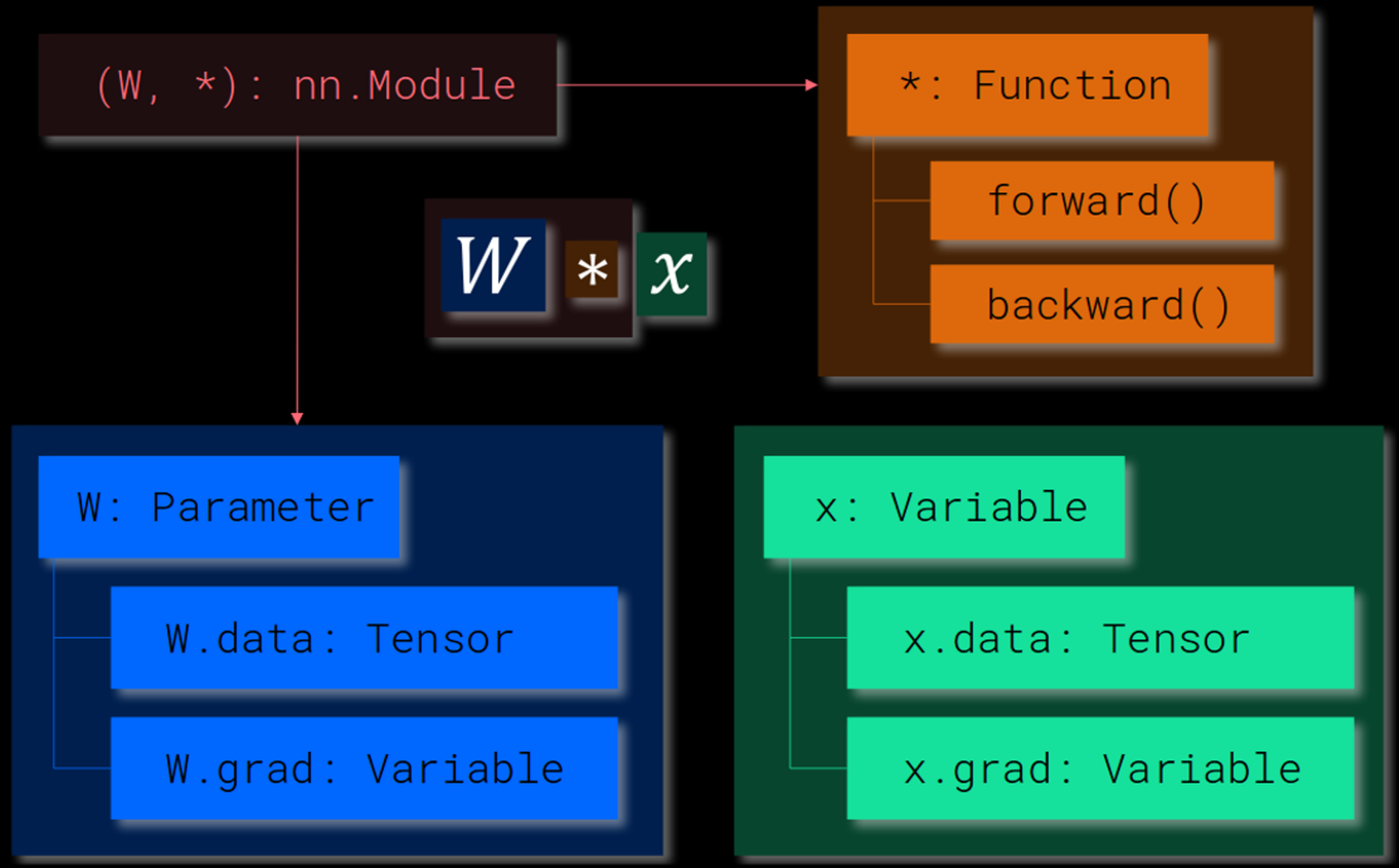
nn.Module
contain *Parameters* and define functions on input *Variables*

autograd.Variable
wraps a *Tensor* and enables auto differentiation

autograd.Function
operate on *Variables*, implements forward and backward

Now directly merged (since Torch 0.5) with Tensors

Basic concepts



Basic concepts

Basic operations in Torch centered around **Tensors**

```
y = torch.rand(5, 3)
print(x + y)
```

```
print(torch.add(x, y))
```

```
# In-place add
y.add_(x)
print(y)
```

```
import torch
x = torch.rand(10, 5)
y = torch.rand(10, 5)
z = x * y
```

```
import torch
x = torch.ones(5, 5)
b = x.numpy()
```

```
import torch
import numpy as np
a = np.ones(5)
x = torch.from_numpy(a)
```

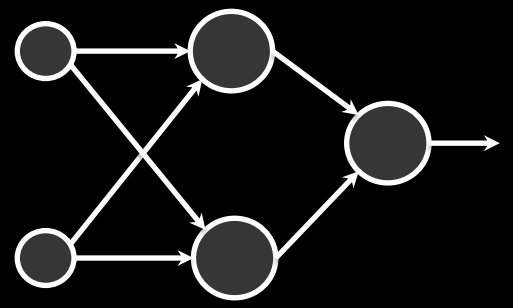
```
import torch
x= torch.rand(100,100)
x = x.cuda()
```

One single operation to go to GPU computing ☺️

Creating a network (lazy way)

PyTorch allows to write models in one line

```python
import torch.nn as nn
model = nn.Sequential
(
    nn.Linear(input_dim, hidden_dim),
    nn.ReLU(True),
    nn.Linear(hidden_dim, output_dim),
    nn.Softmax()
)
x = torch.rand(16, input_dim)
model(x)
```



Low amount of control but easily implemented

Module definition

More flexible and controllable definition

Object-oriented through nn.Module

```
1.  class TwoLayerNet(torch.nn.Module):
2.    def __init__(self, D_in, H, D_out):
3.      super(TwoLayerNet, self).__init__()
4.      self.linear1 = nn.Linear(D_in, H)
5.      self.linear2 = nn.Linear(H, D_out)
6.      self.relu = nn.ReLU(inplace=True)

7.    def forward(self, x):
8.      h_relu = self.linear1(x)
9.      h_relu = self.relu(h_relu)
10.     y_pred = self.linear2(h_relu)
11.     ypred = nn.functional.softmax(y_pred)
12. return y_pred
```

Operations can be layers

Or directly functions to apply
(when no parameters involved)

Residual network example

```python
class ResBlock(nn.Module):
  def __init__(self, dim, dim_res=32):
     super().__init__()
     self.block = nn.Sequential(
    nn.Conv2d(dim, dim_res, 3, 1, 1),
    nn.ReLU(True),
    nn.Conv2d(dim_res, dim, 1),
    nn.ReLU(True)
        )
                    Mixing modules and direct operations !
 def forward(self, x):
   return x + self.block(x)
```

Modules are themselves usable in more complex models !

```python
nn.Sequential(
    ResBlock(64, 32),
    ResBlock(64, 32),
)
```

Optimizing the network

```
# Have some data
N, D_in, H, D_out = 64, 1000, 100, 10
# Create a network
model = TwoLayerNet(D_in, H, D_out)
# Select an adequate loss
loss_fn = torch.nn.MSELoss(size_average=False)
# Create an optimizer (here stochastic gradient descent)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x); loss = loss_fn(y, y_pred)
    # Zero gradients (cumulative).
    optimizer.zero_grad()
    # Backward pass:
    loss.backward()
    # Apply your optimization (gradient descent)
    optimizer.step()
```

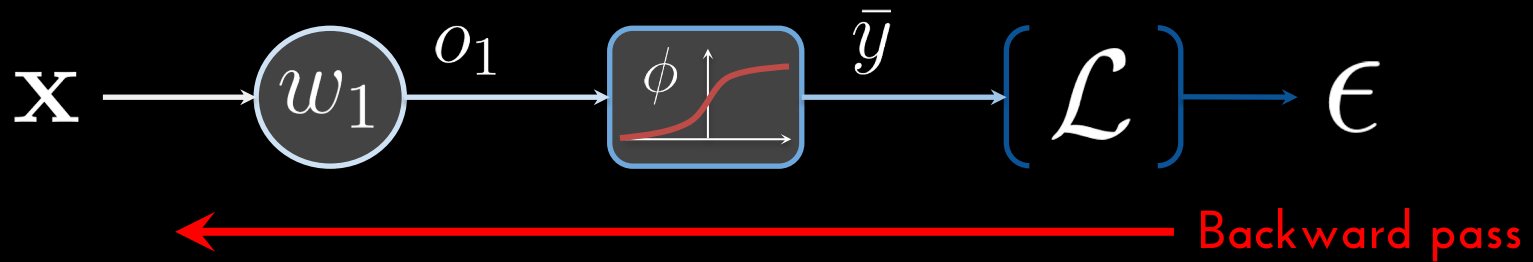
Details on losses

Loss functions

A loss function takes a (output, target) pair of inputs,
Computes the prediction error made by the network
A simple loss: nn.MSELoss (mean-squared)

```
1. output = net(input)
2. target = torch.randn(10) a dummy target
3. criterion = nn.MSELoss()
4. loss = criterion(output, target)
5. print(loss)
```

Flashback time



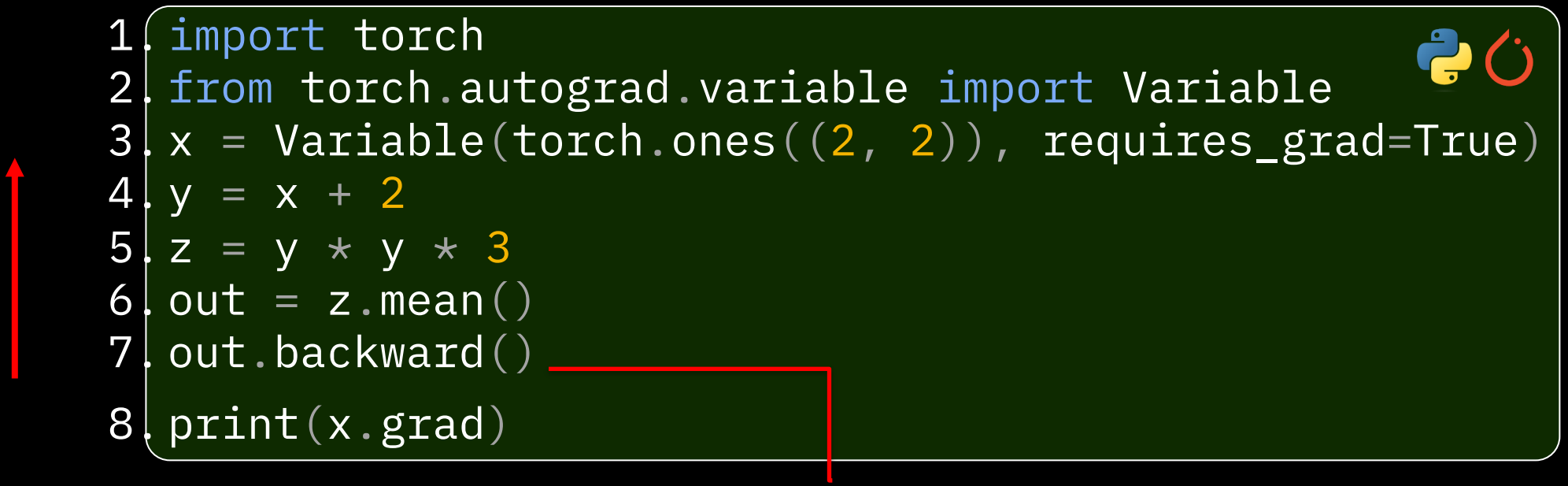
Network definition

THE REAL DEAL

Gradients

- Variables wrap a tensor and saves a gradient history
- Used for automatic differentiation

NOTE: Since Pytorch 0.5 all Tensors are Variables



```python
import torch
from torch.autograd.variable import Variable
x = Variable(torch.ones((2, 2)), requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
out.backward()
print(x.grad)
```

This operation goes through the whole differentiation graph

Attention module (transformers)

Implementing [Yang et. al 2016]
Attention layer

Affine transform

$$1 \quad \mathbf{u}_{it} = \tanh\left(\mathbf{W}_w \mathbf{h}_{it} + \mathbf{b}_w\right)$$

Softmax

$$2 \quad \alpha_{it} = \frac{\exp\left(\mathbf{u}_{it}^T \mathbf{u}_w\right)}{\sum_t \exp\left(\mathbf{u}_{it}^T \mathbf{u}_w\right)}$$

$$3 \quad \mathbf{s}_i = \sum_t \alpha_{it} \mathbf{h}_{it}$$

Trainable matrix

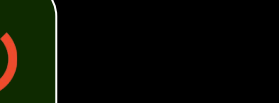
```
# Chord-level attention layer
class ChordLevelAttention(nn.Module):
    def __init__(self, n_hidden):
        super(ChordLevelAttention, self).__init__()
        self.mlp = nn.Linear(n_hidden, n_hidden)
        self.u_w = nn.Parameter(torch.rand(n_hidden))

    def forward(self, X):
        # get the hidden representation of the sequence
1       u_it = F.tanh(self.mlp(X))
        # get attention weights for each timestep
2       alpha = F.softmax(torch.matmul(u_it, self.u_w), dim=1)
        # get the weighted sum of the sequence
3       out = torch.sum(torch.matmul(alpha, X), dim=1)

        return out, alpha
```

And Pytorch handles all derivations and gradients !

Handling data

Using torchvision extremely easy to load known datasets.

Example on MNIST (that we will use)



```python
import torchvision
import torchvision.transforms as transforms
# Use transform to Tensors of normalized range [-1, 1].
transform = transforms.Compose(
[transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=4, shuffle=True, num_workers=2)
for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
```

Handling data

```python
# Simplest Pytorch Audio dataset object ever
class AudioDataset(Dataset):
    def __init__(self, datadir):
        self.data_files = sorted(glob.glob(datadir + '/*'))

    def __getitem__(self, idx):
        try :
            audio_file, sr = librosa.core.load(self.data_files[idx])
            loaded = librosa.feature.melspectrogram(audio_file, sr)
            loaded = torch.from_numpy(np.log1p(np.flipud(loaded)))
            return loaded, 0
        except :
            return torch.Tensor(1), 0

    def __len__(self):
        return len(self.data_files)
```

Things to remember

Deep learning work checklist

- Evaluate generalization on a separate test set
  - Beware of bad splits ! (cf. blue door / GTZAN)
  - *Is your neural network a horse ?* (Clever Hans)
- Check your input data thoroughly (normalization !)
- Address the characteristics of your data
- Select your loss function accordingly
- Always wonder about regularization
  - Dropout, BatchNorm, Augmentations
- Start by solving simple toy problems
  - Good test = *can you overfit on a small set ?*

Bonus creative application

Neural style transfer

Perform some funky neural transfer with Pytorch

https://pytorch.org/tutorials/advanced/neural_style_tutorial.html