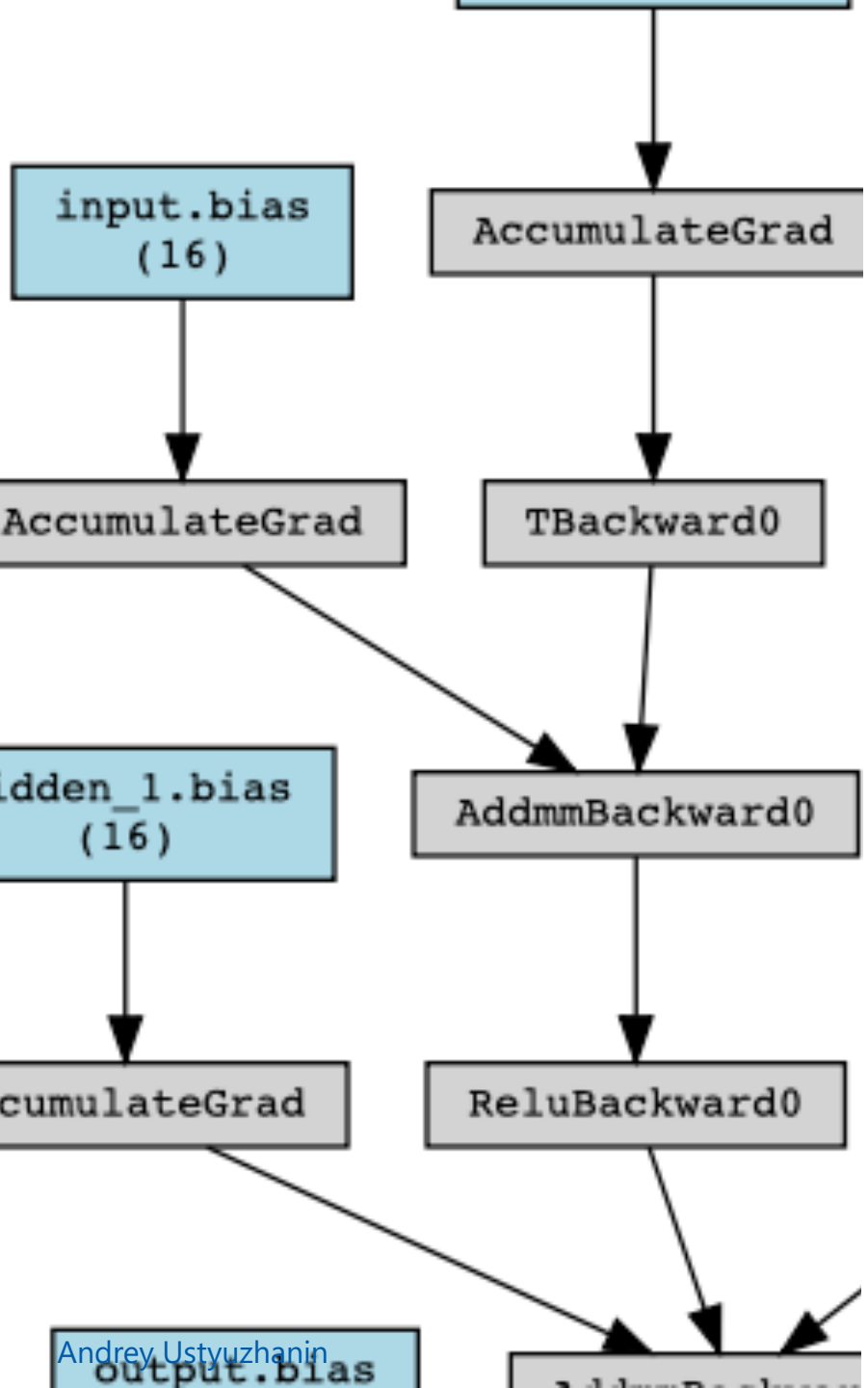
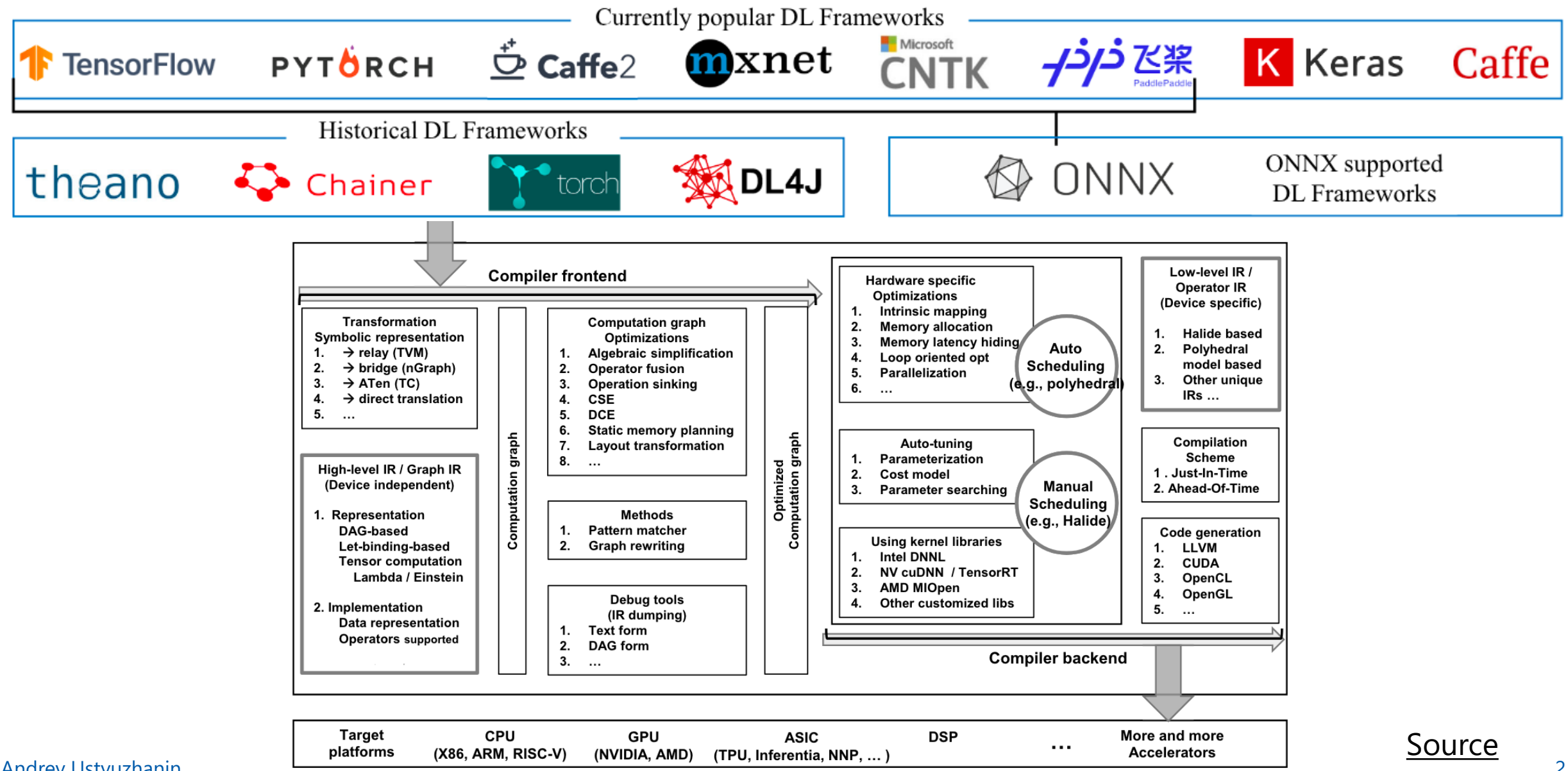


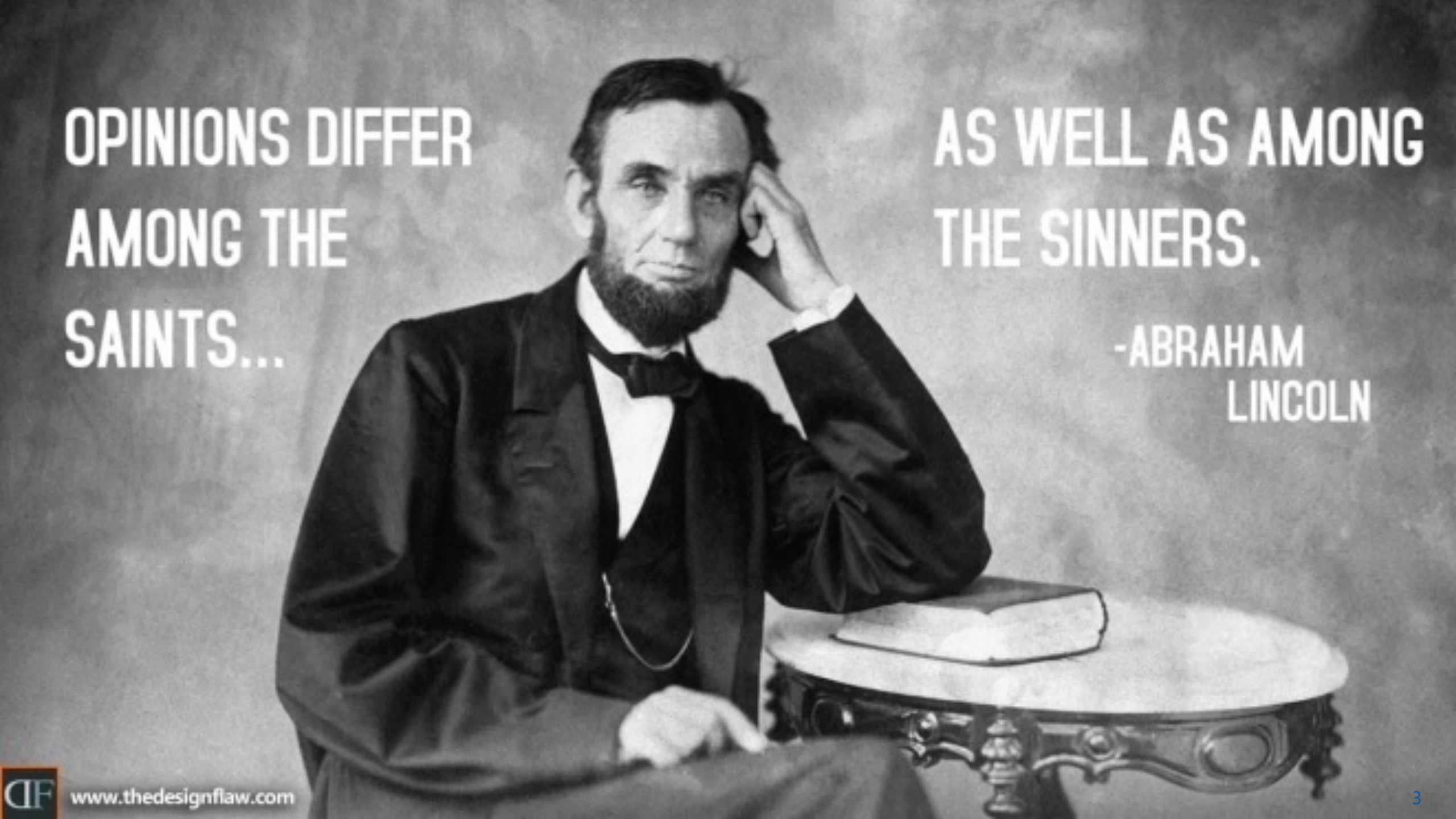

# ReCap: PyTorch, CNNs, RNNs

Andrey Ustyuzhanin

# Deep Learning Frameworks




Source

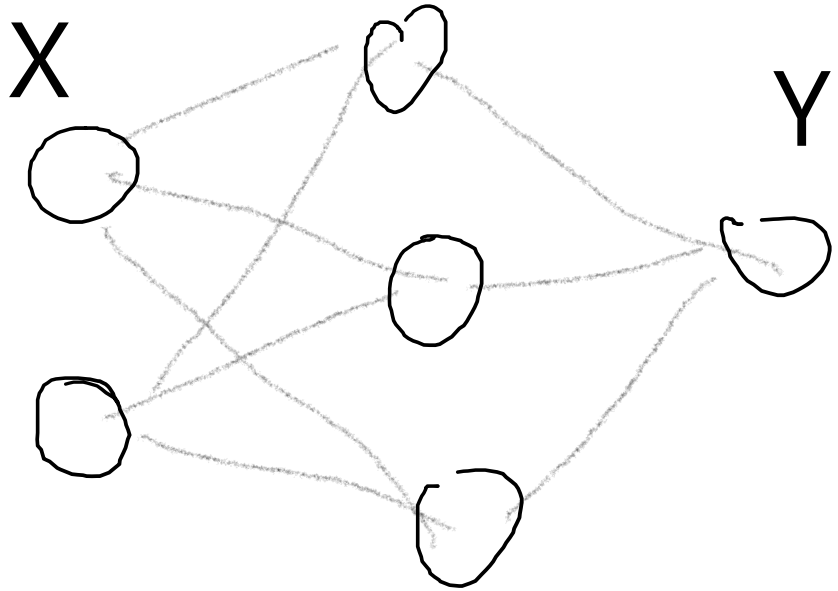

OPINIONS DIFFER AMONG THE SAINTS...
AS WELL AS AMONG THE SINNERS.
-ABRAHAM LINCOLN
www.thedesignflaw.com

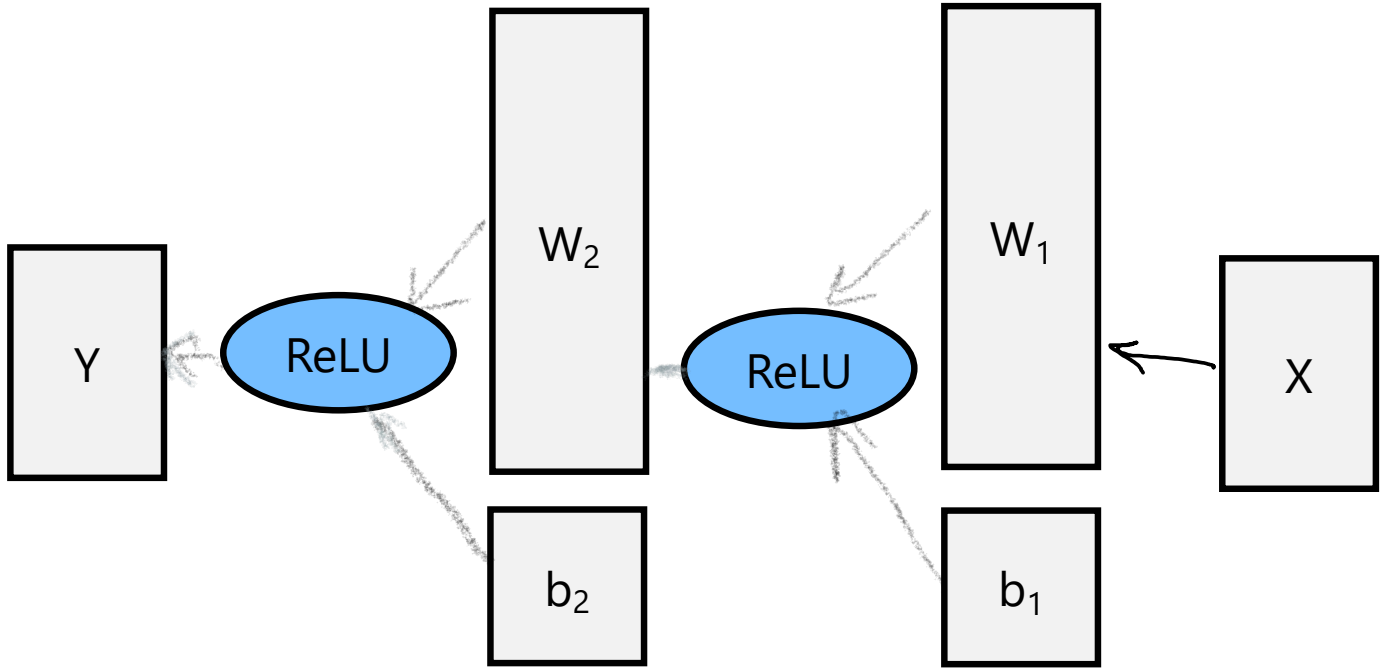
# PyTorch highlights

- Simple, transparent development/ debugging
- Rich Ecosystem:
  - Plenty of pretrained models
  - NLP, Vision, …
  - Interpretation
  - Hyper-optimization
- Production Ready (C++, ONNX, Services)
- Distributed Training, declarative data parallelism
- Cloud Deployment support
- Choice of many industry leaders and researchers

facebook Artificial Intelligence

# Neural network representation




Functions

Tensors

$$Y = \mathbf{relu}(W_2 \times \mathbf{relu}(W_1 X + b_1) + b_2)$$

# Building blocks, tensors

```
torch.randn(*size)                  # tensor with independent N(0,1) entries
torch.[ones|zeros](*size)           # tensor with all 1's [or 0's]
torch.Tensor(L)                     # create tensor from [nested] list or ndarray L
x.clone()                           # clone of x
with torch.no_grad():               # code wrap that stops autograd from tracking tensor history
requires_grad=True                  # arg, when set to True, tracks computation
                                    # history for future derivative calculations
```

```
x.size()                            # return tuple-like object of dimensions
torch.cat(tensor_seq, dim=0)        # concatenates tensors along dim
x.view(a,b,...)                     # reshapes x into size (a,b,...)
x.view(-1,a)                        # reshapes x into size (b,a) for some b
x.transpose(a,b)                    # swaps dimensions a and b
x.permute(*dims)                    # permutes dimensions
x.unsqueeze(dim)                    # tensor with added axis
x.unsqueeze(dim=2)                  # (a,b,c) tensor -> (a,b,1,c) tensor
```

# Tensor creation and placement

```
>>> torch.zeros([2, 4], dtype=torch.int32)
tensor([[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]], dtype=torch.int32)
>>> cuda0 = torch.device('cuda:0')
>>> torch.ones([2, 4], dtype=torch.float64, device=cuda0)
tensor([[ 1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000]], dtype=torch.float64,
device='cuda:0')
```

- Keep in mind occurrence of tensors on devices: CPU, GPU, TPU
- Operations can be performed only if its arguments are inhabiting the same device

# GPU, TPU support

```
torch.cuda.is_available                                  # check for cuda
x.cuda()                                                 # move x's data from
                                                         # CPU to GPU and return new object

x.cpu()                                                  # move x's data from GPU to CPU
                                                         # and return new object

if not args.disable_cuda and torch.cuda.is_available(): # device agnostic code
    args.device = torch.device('cuda')                   # and modularity
else:                                                    #
    args.device = torch.device('cpu')                    #

net.to(device)                                           # recursively convert their
                                                         # parameters and buffers to
                                                         # device specific tensors

mytensor.to(device)                                      # copy your tensors to a device
                                                         # (gpu, cpu)
```

- https://pytorch.org/docs/stable/cuda.html
- http://pytorch.org/xla/release/1.5/index.html

# Building blocks, graph

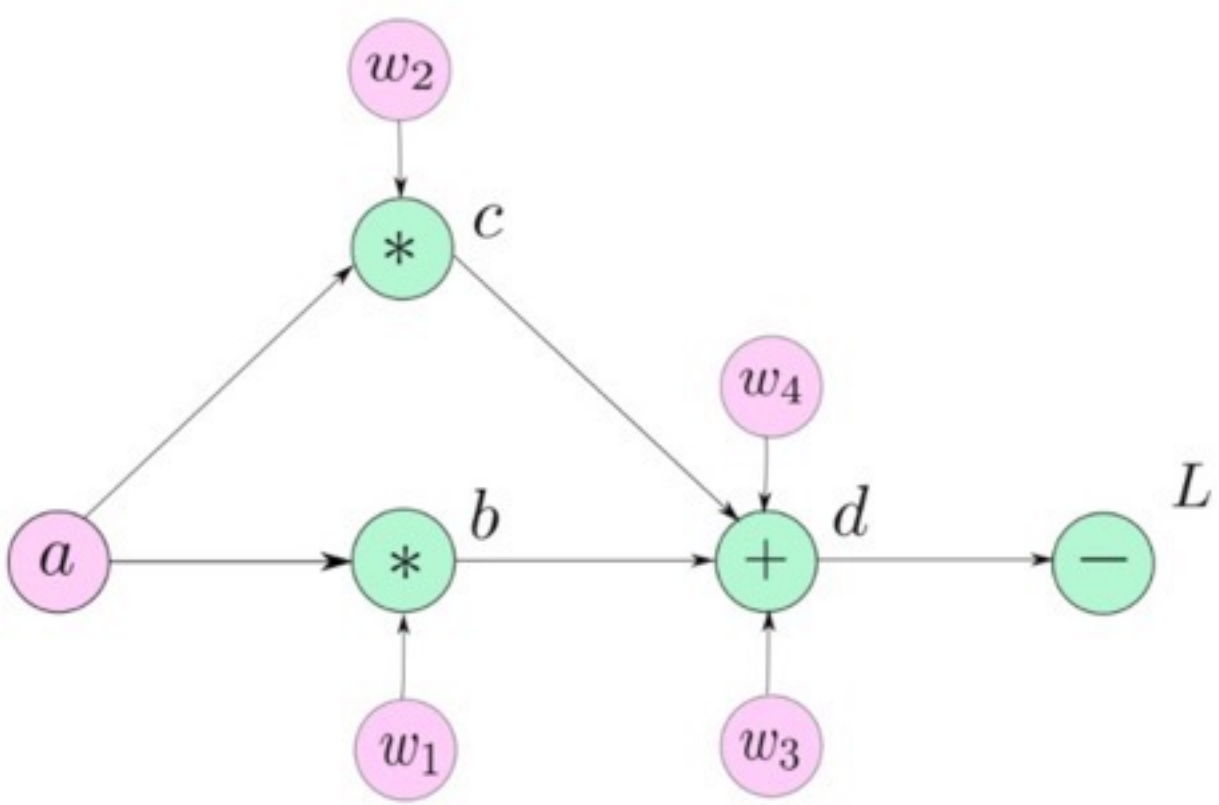
Toy example:

$$b = w_1 * a$$

$$c = w_2 * a$$

$$d = w_3 * b + w_4 * c$$

$$L = 10 - d$$




source

# Math operations

```
A.mm(B)       # matrix multiplication
A.mv(x)       # matrix-vector multiplication
x.t()         # matrix transpose
```

- https://pytorch.org/docs/stable/torch.html?highlight=mm#math-operations

# Computing backpropagation

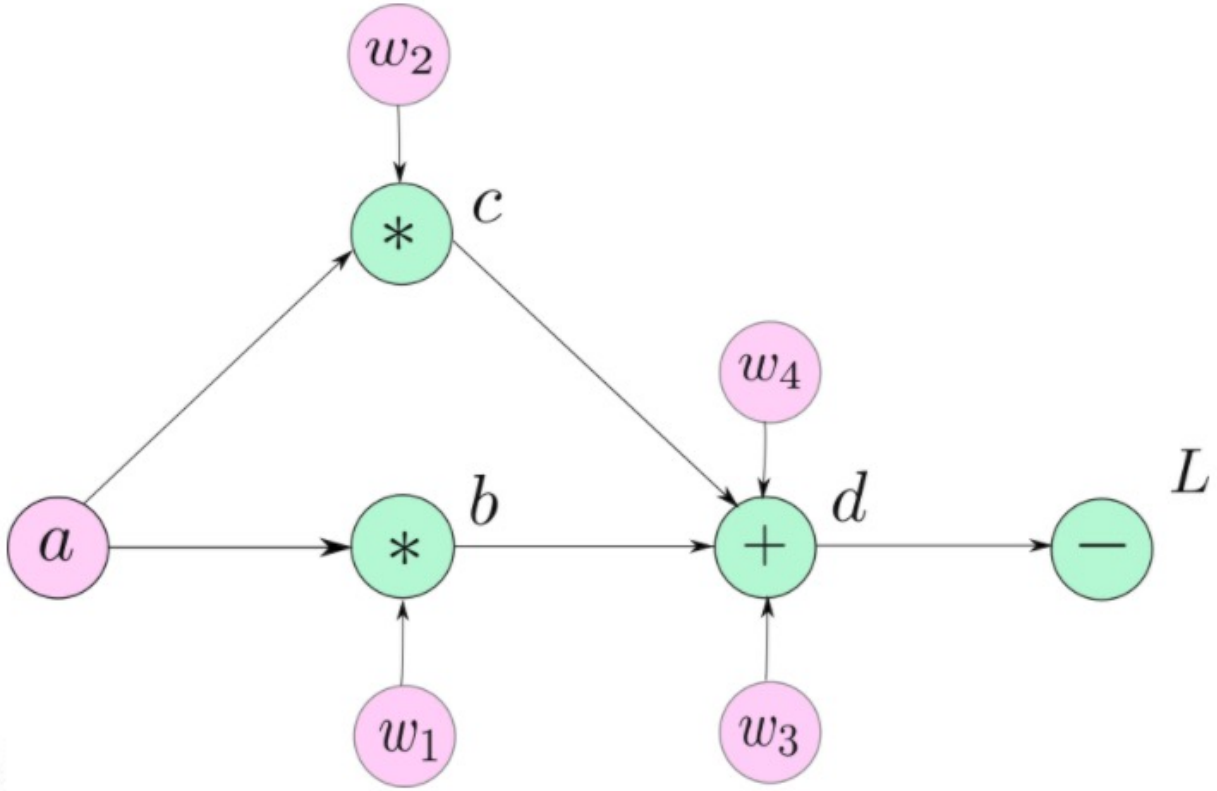
$$b = w_1 * a$$

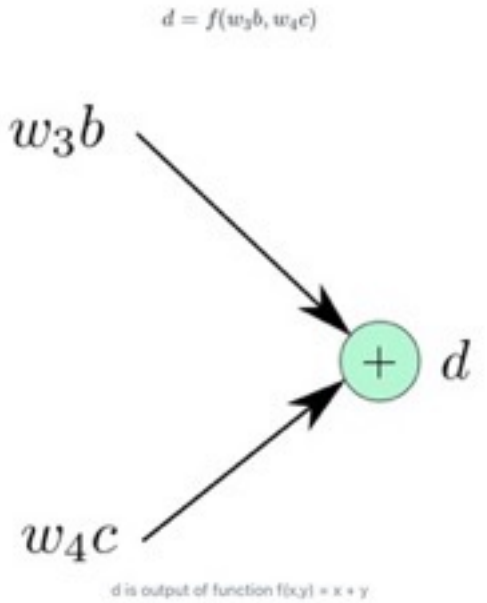
$$c = w_2 * a$$

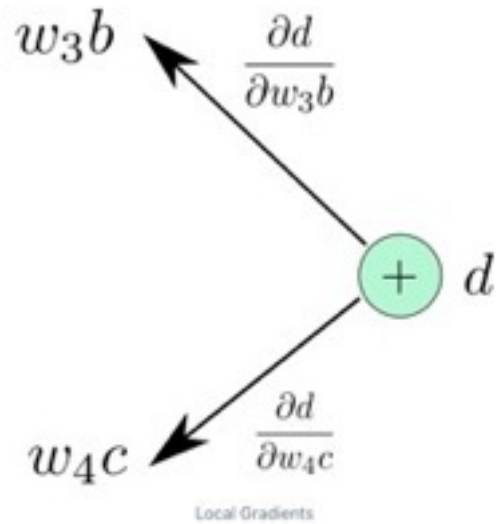
$$d = w_3 * b + w_4 * c$$

$$L = 10 - d$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}$$



# Computing gradient automatically

```
>> t1 = torch.randn((3,3), requires_grad = True)

>> t2 = torch.FloatTensor(3,3) # No way to specify requi
>> t2.requires_grad = True
```

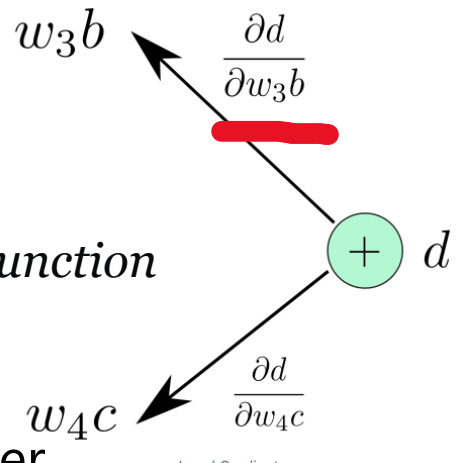
Each **Tensor** has an attribute **grad_fn**, which refers to the mathematical operator that created it.

If **Tensor** is a leaf node (initialized by the user), then the **grad_fn** is **None**.

```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1*a
c = w2*a

d = w3*b + w4*c

L = 10 - d

print("The grad fn for a is", a.grad_fn)
print("The grad fn for d is", d.grad_fn)
```

```
The grad fn for a is None
The grad fn for d is <AddBackward0 object at 0x1033afe48>
```

# Functions




- All math steps represented by classes inherited from *torch.autograd.Function*
  - *forward*, computes node output and buffers it
  - *backward*, stores incoming gradient in **grad** and passes further

```
def backward (incoming_gradients):
        self.Tensor.grad = incoming_gradients

        for inp in self.inputs:
                if inp.grad_fn is not None:
                        new_incoming_gradients = //
                          incoming_gradient * local_grad(self.Tensor, inp)

                        inp.grad_fn.backward(new_incoming_gradients)
                else:
                        pass
```

# Gradient descent

- Compute gradient for every tensor involved

```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1*a
c = w2*a

d = w3*b + w4*c

# Replace L = (10 - d) by
L = (10 -d).sum()

L.backward()
```

- Make gradient descent step in the opposite direction:

```
learning_rate = 0.5
w1 = w1 - learning_rate * w1.grad
```

# Dynamic graph

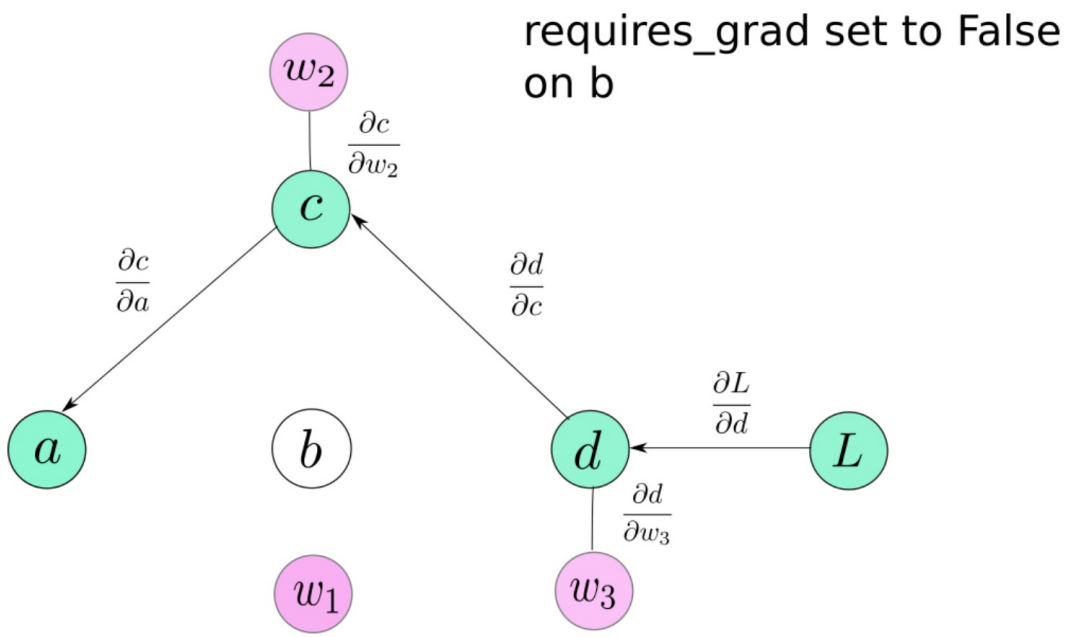
- Calling *forward* creates
  - graph with the intermediate node output values,
  - buffers for the non-leaf nodes,
  - buffers for intermediate gradient values.
- Calling *backward*
  - computes gradients and
  - frees the buffers and destroys the graph.
- Next time, calling *forward*
  - leaf node buffers from the previous run will be shared,
  - **non-leaf nodes buffers will be recreated**.

# Gradient cleaning

- Due to the flexibility of the network architecture, it is not obvious when does iteration of a gradient descent stops, so *backward's* gradients are accumulated each time a variable (Tensor) occurs in the graph;
- It is usually desired for RNN cases;
- If you do not need to accumulate those, you must **clean previous gradient values** at the end of each iteration:
  - Either by x.data.zero_() for every model tensor x;
  - Or by optimizers's *zero_grad*() method, which is more preferable.

# Freezing weights

- **Requires_grad** attribute of the *Tensor* class. By default, it's **False**. It comes handy when you must freeze some layers and stop them from updating parameters while training.
- Thus, no gradient would be propagated to them, or to those layers which depend upon these layers for gradient flow **requires_grad**.
- When set to **True**, **requires_grad** is contagious: even if one operand of an operation has **requires_grad** set to **True**, so will the result.



# Pre-trained models' enhancement

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by
default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

# Inference

- When we are computing gradients, we need to cache input values, and intermediate features as they maybe required to compute the gradient later. The gradient of **b=w1∗a** w.r.t it's inputs **w1** and **a** is **a** and **w1,** respectively.
- We need to store these values for gradient computation during the backward pass. This affects the memory footprint of the network.
- While, we are performing inference, we don't compute gradients

```
with torch.no_grad:
        inference code goes here
```

- Even better and recent optimized context: `with torch.inference_mode` (link)

# Neural Network class: torch.nn.Module

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

# Helpful functions

| Type | Classes/functions examples |
|---|---|
| Loss functions | BCELoss, CrossEntropyLoss, L1Loss, MSELoss, NLLLoss, SoftMarginLoss, MultiLabelSoftMarginLoss, CosineEmbeddingLoss, KLDivLoss, … |
| Activation functions | ReLU, ELU, SELU, PReLU, LeakyReLU, Threshold, HardTanh, Sigmoid, Tanh,LogSigmoid, Softplus, Softshrink, Softsign, TanhShrink, Softmin, Softmax, Softmax2d .. |
| Optimizers | opt = optim.X(model.parameters(), ...)<br><br>where X is SGD, Adadelta, Adagrad, Adam, SparseAdam, Adamax, ASGD, LBFGS, RMSProp or Rprop |
| Visualization | from torchviz import make_dot<br><br>make_dot(y.mean(), params=dict(model.named_parameters())) |

# ONNX

Open Neural Network eXchange (ONNX) is an open standard format for representing machine learning models. The torch.onnx module can export PyTorch models to ONNX. The model can then be consumed by any of the many runtimes that support ONNX.

- Export:

```
torch.onnx.export(model, dummy_input, "alexnet.onnx",
verbose=True, input_names=input_names,
output_names=output_names)
```

- Run:

```
import onnx

# Load the ONNX model
model = onnx.load("alexnet.onnx")

# Check that the model is well formed
onnx.checker.check_model(model)

# Print a human readable representation of the graph
print(onnx.helper.printable_graph(model.graph))
```

# ONNX Runtimes












Optimum














# Data Utils

## Datasets

```
Dataset                     # abstract class representing dataset
TensorDataset               # labelled dataset in the form of tensors
Concat Dataset              # concatenation of Datasets
```

- https://pytorch.org/docs/stable/data.html?highlight=dataset#torch.utils.data.Dataset

## Dataloaders and DataSamplers

```
DataLoader(dataset, batch_size=1, ...)      # loads data batches agnostic
                                            # of structure of individual data points

sampler.Sampler(dataset,...)                # abstract class dealing with
                                            # ways to sample from dataset

sampler.XSampler where ...                  # Sequential, Random, Subset,
                                            # WeightedRandom or Distributed
```

- https://pytorch.org/docs/stable/data.html?highlight=dataloader#torch.utils.data.DataLoader

# PyTorch 2.0 (since March 2023)

- PyTorch 2.0 has a new API called torch.compile that wraps models and returns compiled ones.
- It is 100% backward compatible and features technologies like TorchInductor with Nvidia and AMD GPUs, Accelerated Transformers, Metal Performance Shaders backend, and more.
- Metal Performance Shaders (MPS) backend provides GPU accelerated PyTorch training on Mac platforms with added support for Top 60 most used ops, bringing coverage to over 300 operators.
- Amazon AWS also added optimizations for PyTorch CPU inference on their Graviton3-based C7g instances.
- Lastly, there are new features and technologies like TensorParallel, DTensor, 2D parallel, TorchDynamo, AOTAutograd, PrimTorch, and TorchInductor.

https://bit.ly/3KgcsYE

# PyTorch Lightning

- The lightweight PyTorch wrapper for high-performance AI research.

**Maximal flexibility**

```
def training_step(self, batch, batch_nb):
  x, y = batch
  z = self.encoder(x)
  x_hat = self.decoder(z)
  mse = F.mse_loss(x_hat, x)
  gan_regularizer = self.discriminator(x_hat)
  loss = mse + gan_regularizer
  return loss
```

**No boilerplate**
**Maximal flexibility**

```
if gpu:
  x = x.cuda(0)
z = encoder(x)
x_hat = decoder(z)
.backward()
```

```
Trainer(
  devices=32,
  accelerator='gpu|tpu|hpu'
)
```

**Self contained models**

**Modular**


https://github.com/PyTorchLightning/pytorch-lightning

PYTORCH

```
# models
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))

encoder.cuda(0)
decoder.cuda(0)

# download on rank 0 only
if global_rank == 0:
    mnist_train = MNIST(os.getcwd(), train=True, download=True)

# split dataset
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize(0.5, 0.5)])
mnist_train = MNIST(os.getcwd(), train=True, download=True, transform=transform)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])

# The dataloaders handle shuffling, batching, etc...
mnist_train = DataLoader(mnist_train, batch_size=64)
mnist_val = DataLoader(mnist_val, batch_size=64)

 # optimizer
 params = [encoder.parameters(), decoder.parameters()]
 optimizer = torch.optim.Adam(params, lr=1e-3)

# TRAIN LOOP
model.train()
num_epochs = 1
for epoch in range(num_epochs):
  For train_batch in mnist_train:
    x, y = train_batch
    x = x.cuda(0)
    x = x.view(x.size(0), -1)
    z = encoder(x)
    x_hat = decoder(z)
    loss = F.mse_loss(x_hat, x)
    print('train loss: ', loss.item())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

# EVAL LOOP
model.eval()
with torch.no_grad():
  val_loss = []
  for val_batch in mnist_val:
    x, y = val_batch
    x = x.cuda(0)
    x = x.view(x.size(0), -1)
    z = encoder(x)
    x_hat = decoder(z)
    loss = F.mse_loss(x_hat, x)
    val_loss.append(loss)
  val_loss = torch.mean(torch.tensor(val_loss))
  model.train()
```

PYTORCH LIGHTNING

## Turn PyTorch into Lightning

Lightning is just plain PyTorch.



https://github.com/PyTorchLightning/pytorch-lightning

# Ecosystem

- PyTorch lightning
- PyTorch geometric – graph neural nets
- Hydra – parametrization and experiments
- Horovod – distributed training
- Skorch – scikit-learn compatible neural network library
- Captum – model interpretation
- Ray – accelerating ML workloads
- And many others, see https://pytorch.org/ecosystem/

# PyTorch wrap-up

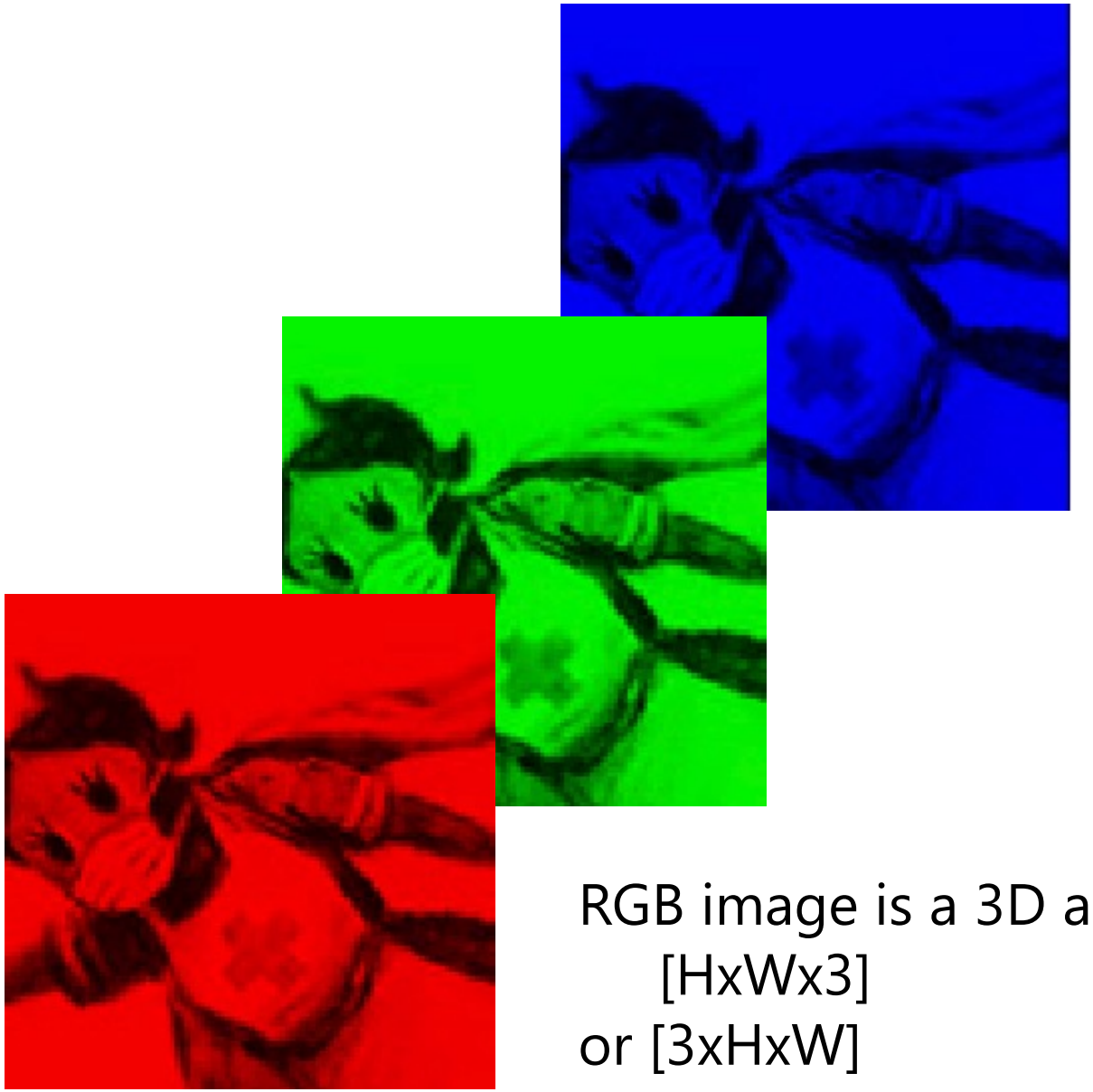
- PyTorch is a solid, flexible, production-ready foundation for real-life deep-learning applications
- Building blocks:
  - Tensors
  - Functions
- Dynamic graph automatic differentiation
  - CPU, GPU, TPU
- Rich ecosystem

# Convolutional Neural Networks

Andrey Ustyuzhanin

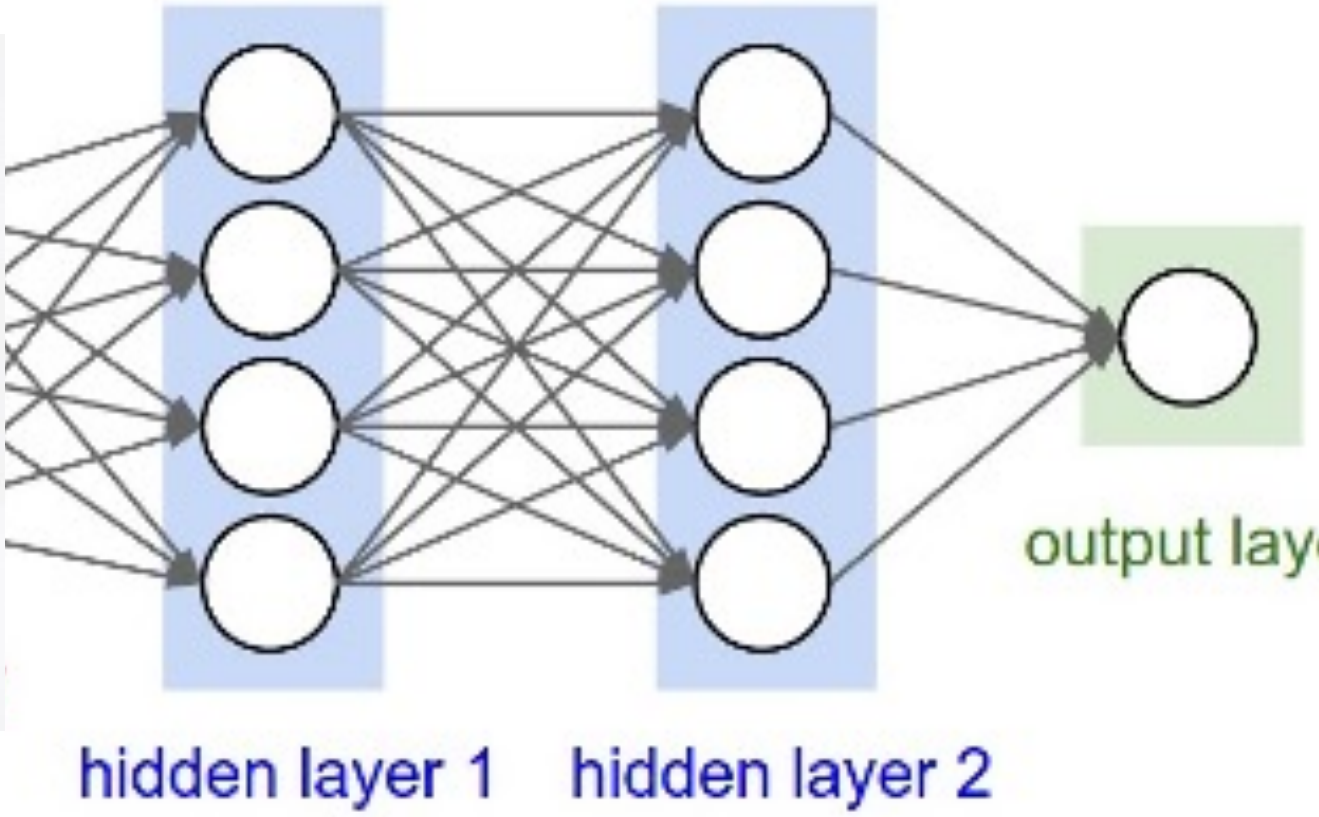
# Image Recognition

Banksy



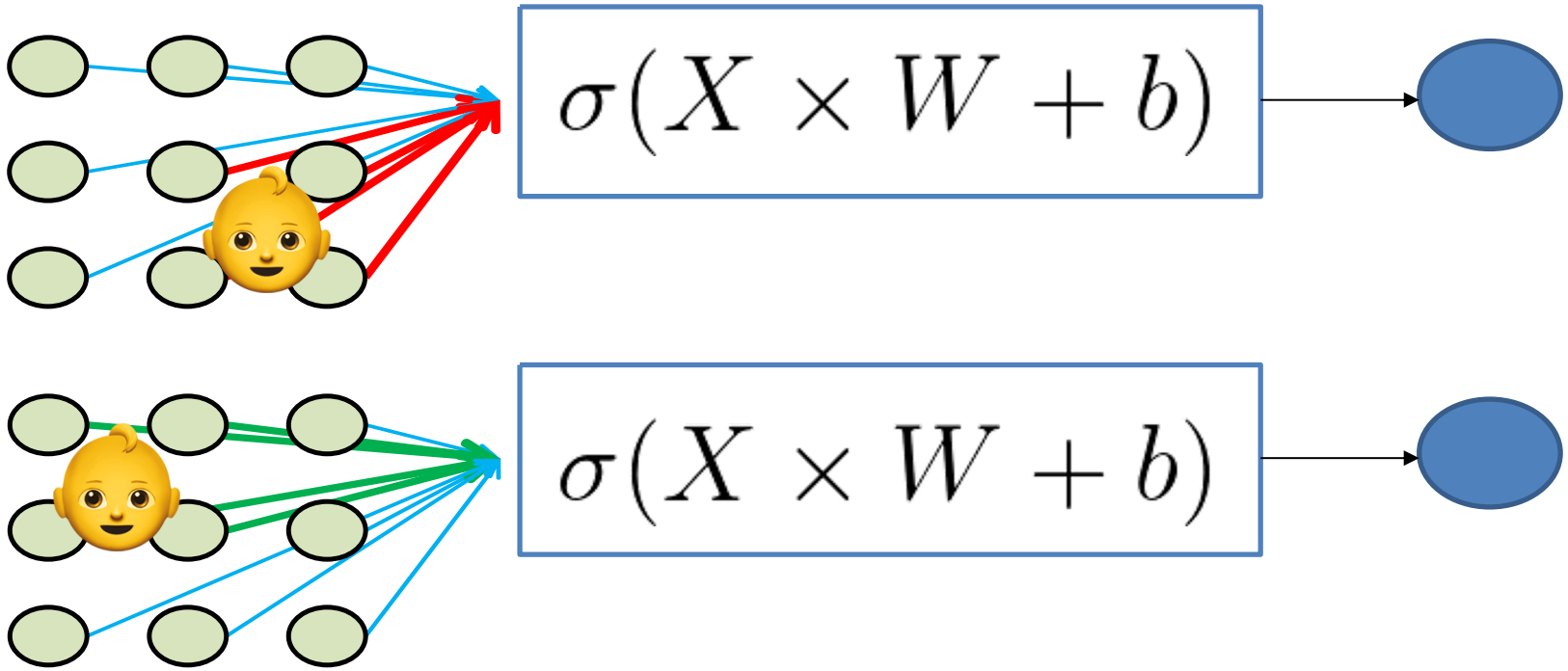
RGB image is a 3D array
[HxWx3]
or [3xHxW]

# General idea




P(”Kid”)

# Problem with images



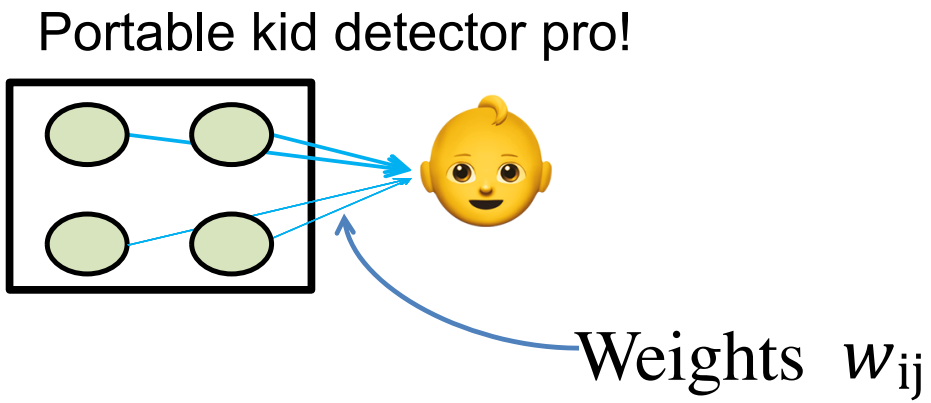

**You network will have to learn those two cases separately!**

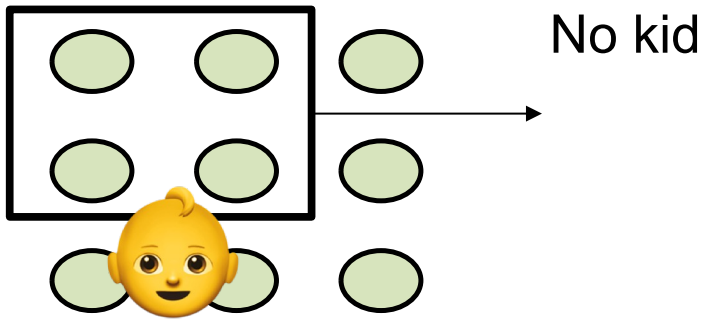
Worst case: one neuron per position.

Slide from Andrey Zimovnov's “Deep learning for computer vision”

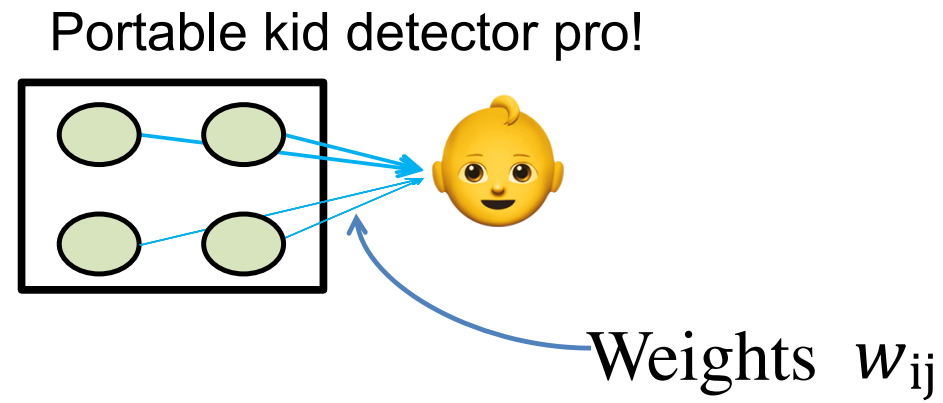
Main idea: let’s encode ”kid” by weight tensor that we can shift across the image.

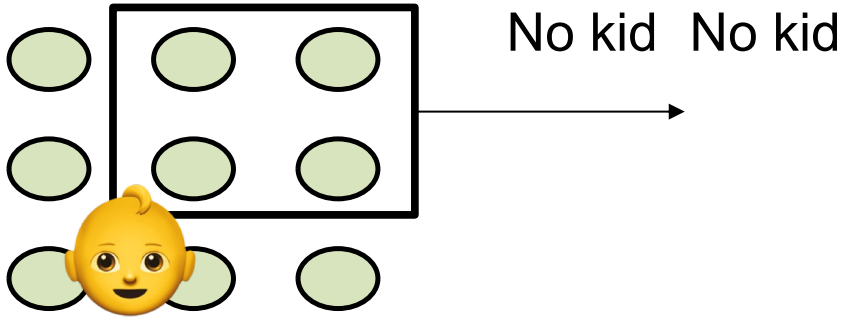
# Same features for each spot



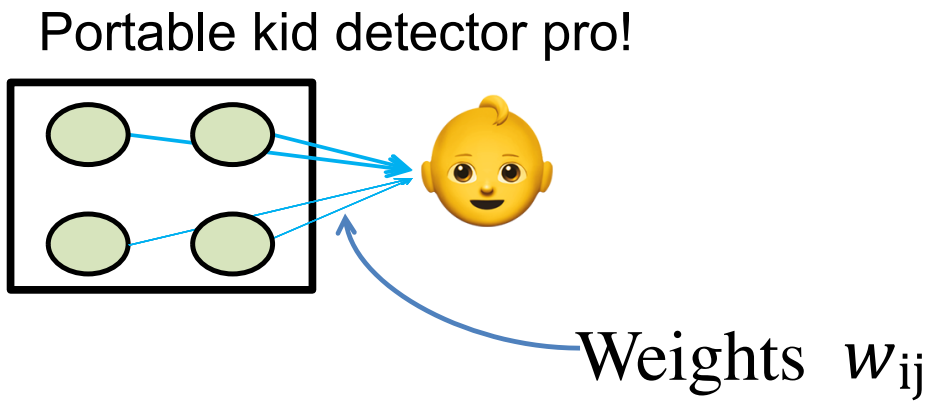



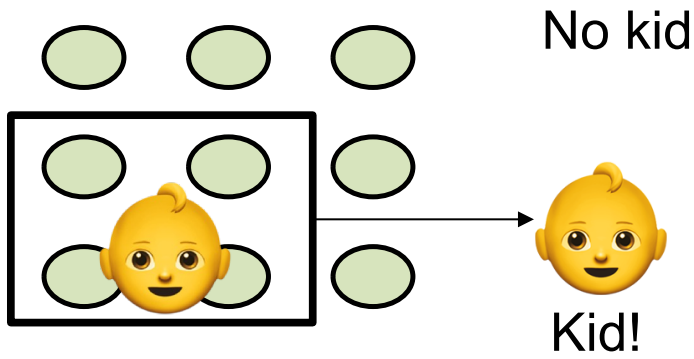
# Same features for each spot



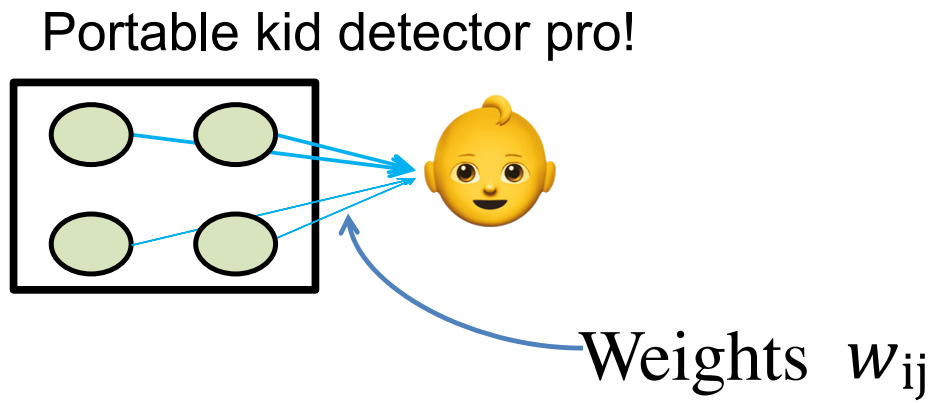



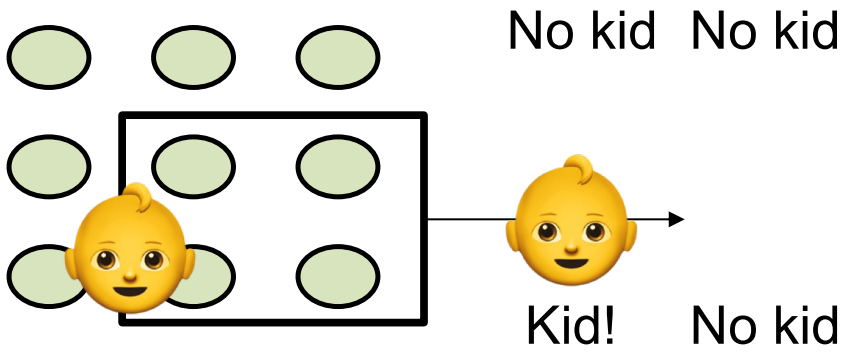
# Same features for each spot



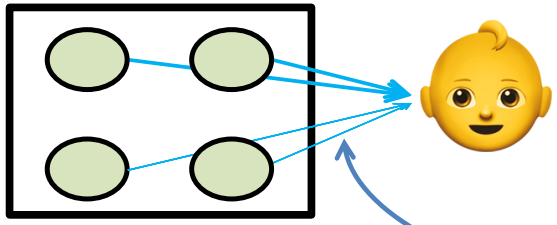
# Same features for each spot



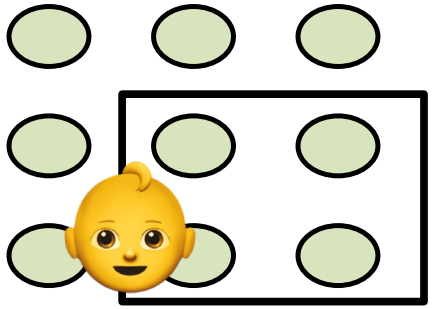



# Same features for each spot

Portable kid detector pro!



Weights $w_{ij}$



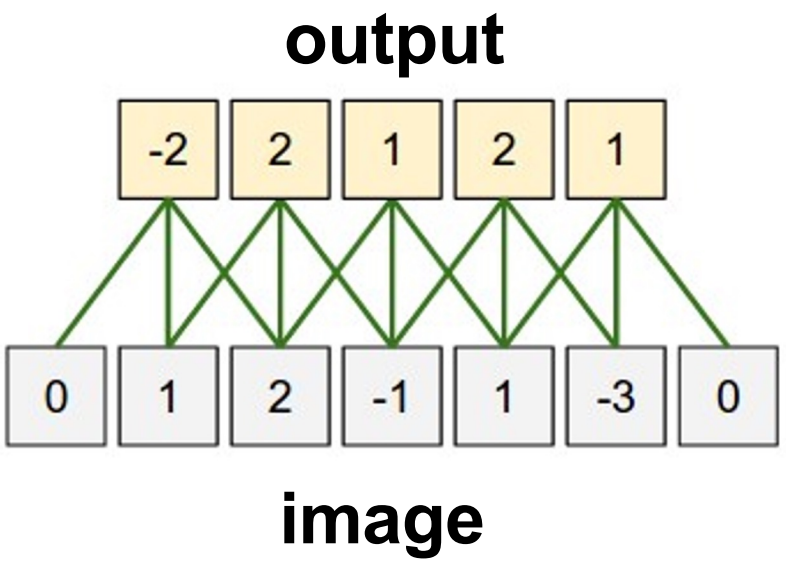
No kid No kid

Kid! No kid

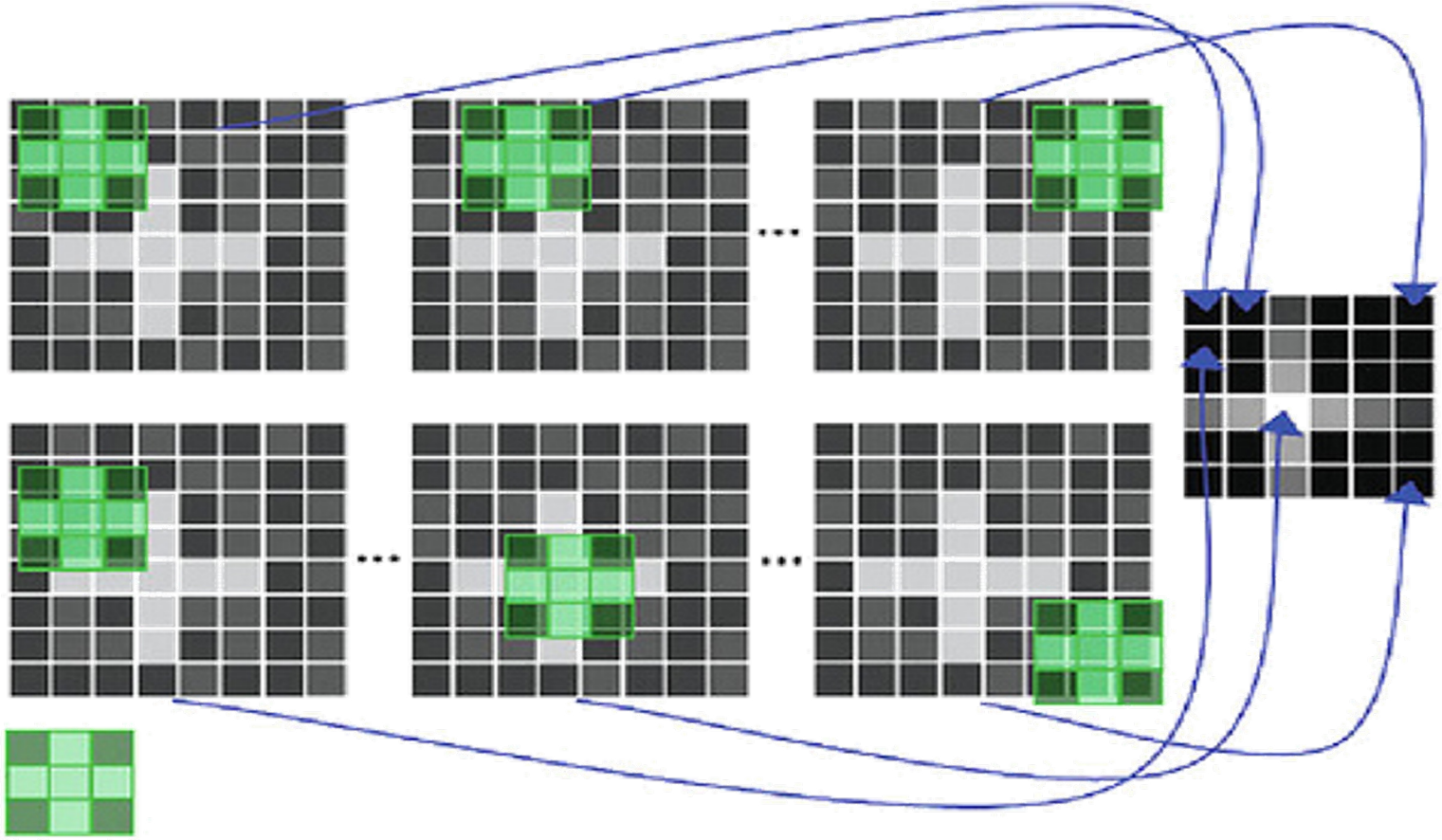
→ There's a kid on the pic!

# Convolution

- Apply same weights to all patches
- 1D example:

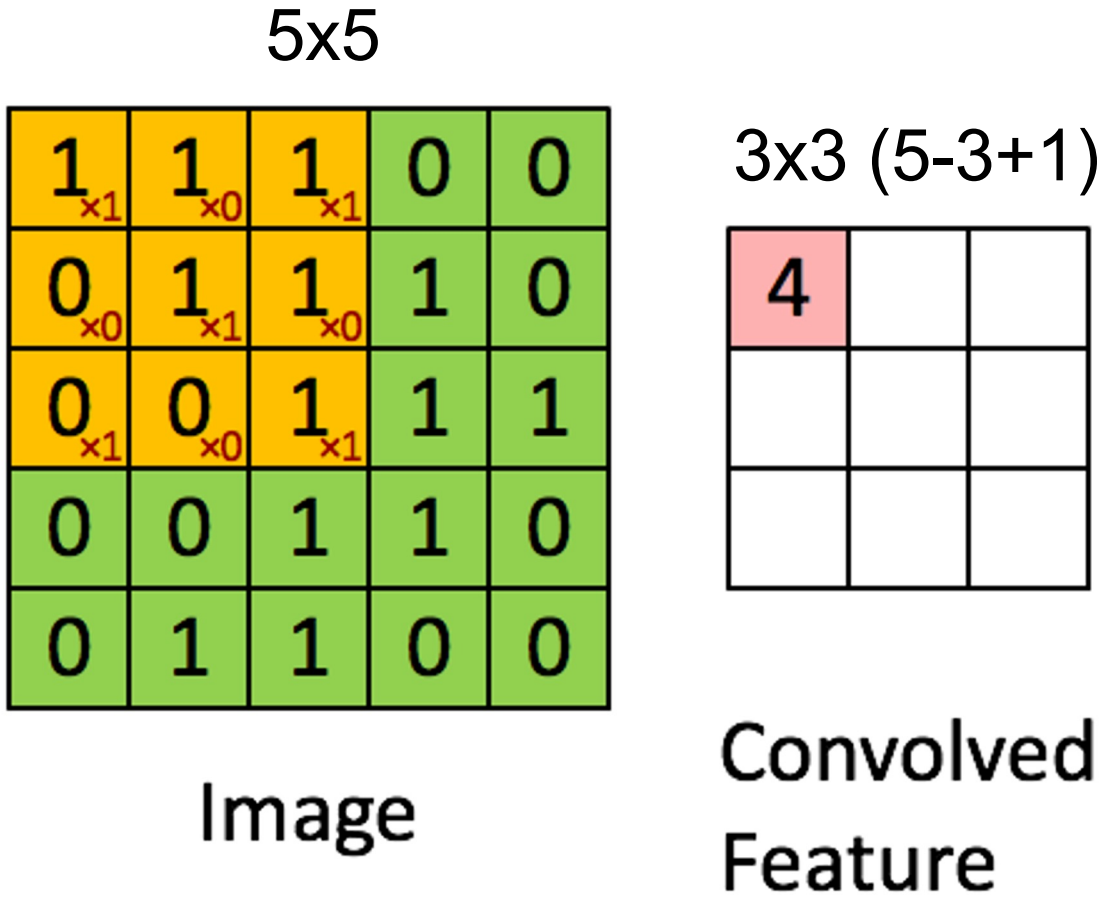



# Convolution, 2D



apply one “filter” to all patches

# Convolution




Intuition: how kid-like is this square?

# Example of convolution

Input image

Convolution Kernel

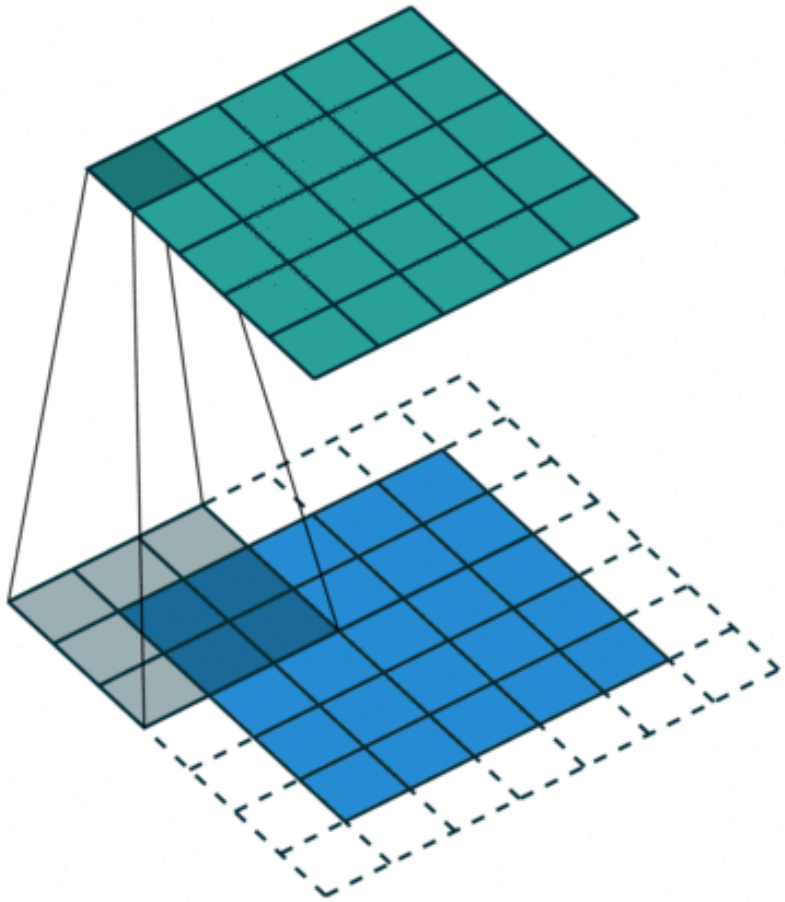
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Intuition: how **edge-like** is this square?

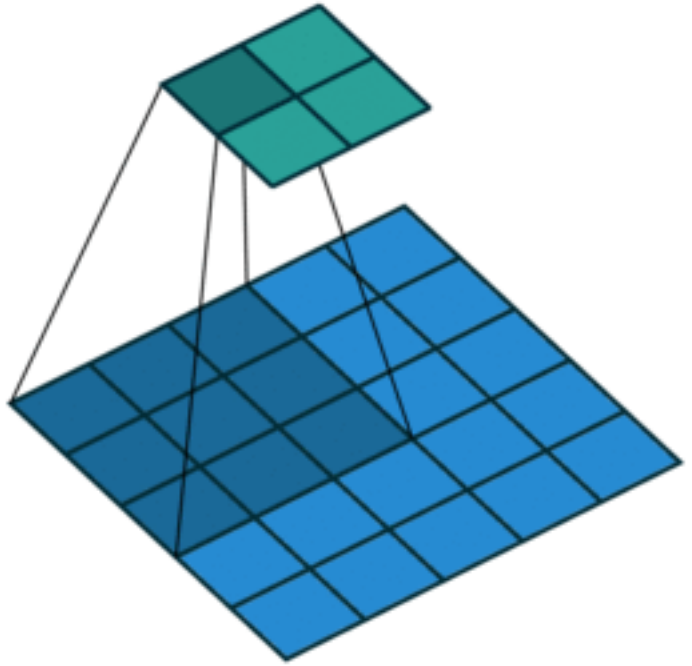
# Convolution tricks: padding

- pad the edges with extra, “fake” pixels (usually of value 0, hence the oft-used term “zero padding”). This way, the kernel when sliding can allow the original edge pixels to be at its center, while extending into the fake pixels beyond the edge, producing an **output the same size as the input;**
- **padding_mode (string, optional)** – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

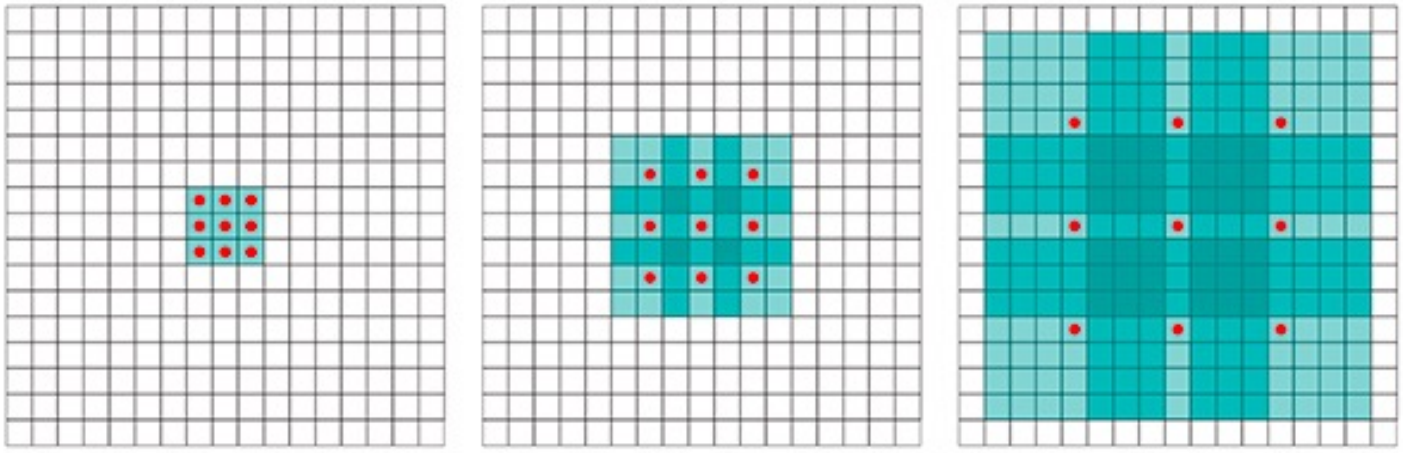


# Convolution tricks: striding and dilation

- The idea of the **stride** is to skip some of the slide locations of the kernel (see figure on the right)
- **Dilated convolution** is a basic convolution only applied to the input volume with defined gaps (see below). You may use dilated convolution when:



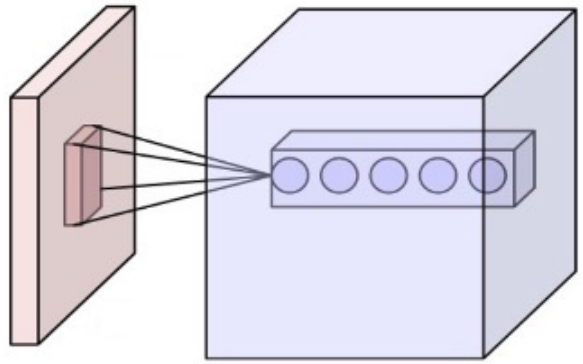
- You are working with higher resolution images, but fine-grained details are still important
- You are constructing a network with fewer parameters



# Convolution Demo

- <Demo>

# Convolution Layer

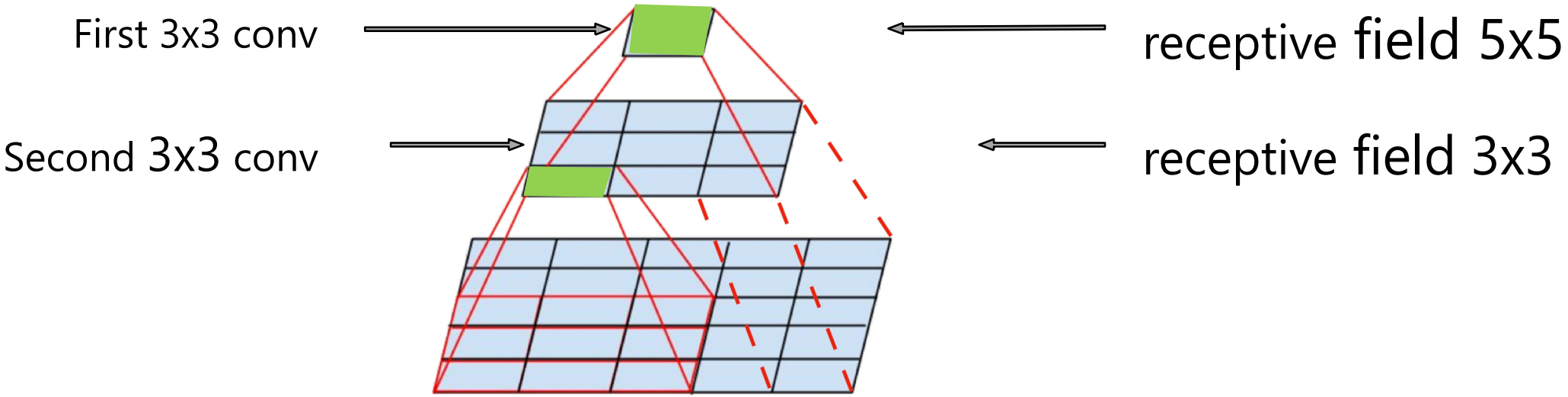


- Intuition: learns weights by gradient descent
- May learn arbitrary number of kernels, one per output channel
- the output value of the layer with input size (N, $C_{in}$, H, W) and output (N, $C_{out}$, $H_{out}$, $W_{out}$) is

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

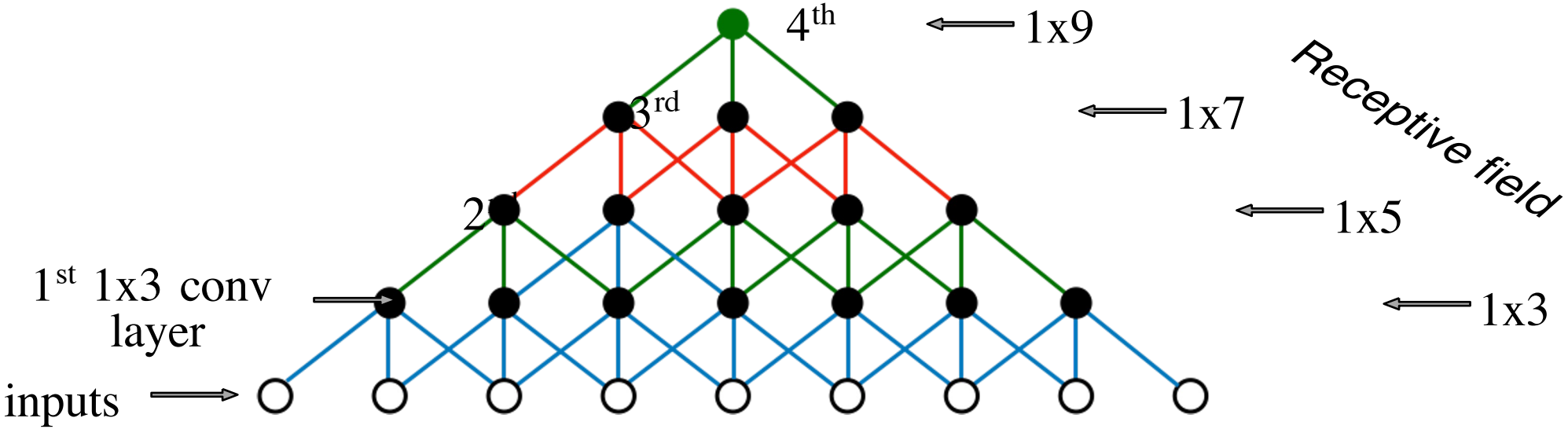
  N - batch size, C - number of channels.
- Number of learnable parameters: $C_{out}$ * ($C_{in}$ * size(kernel) + 1)
- https://pytorch.org/docs/master/generated/torch.nn.Conv2d.html

# Receptive field



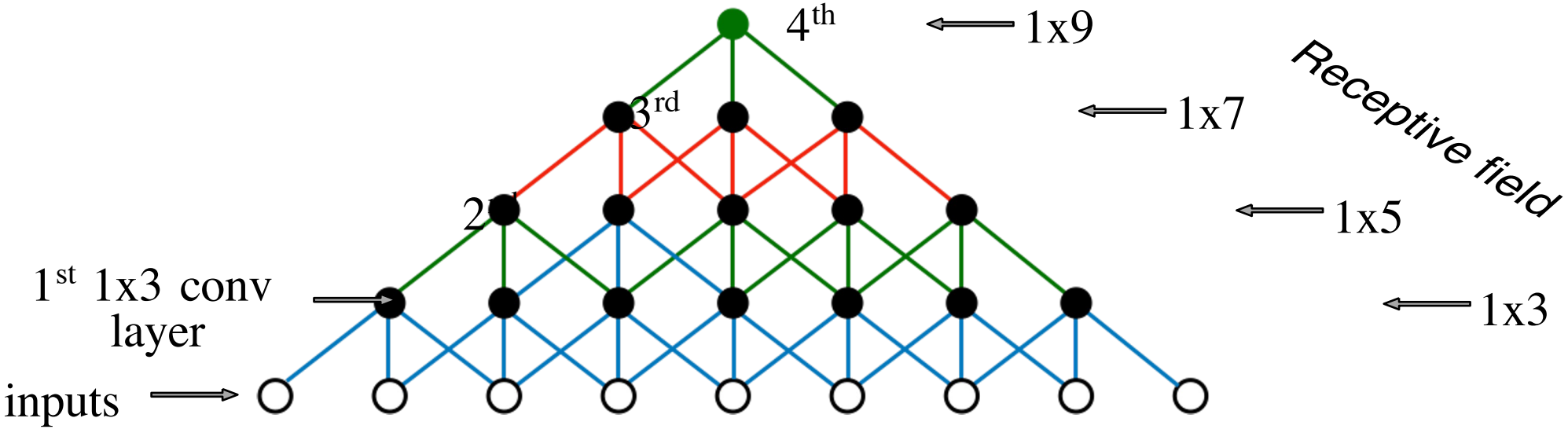

We can recognize larger objects by stacking several small convolutions!

# Receptive field




**Q:** how many 3x3 convolutions we should use to recognize a 100x100px kid

# Receptive field



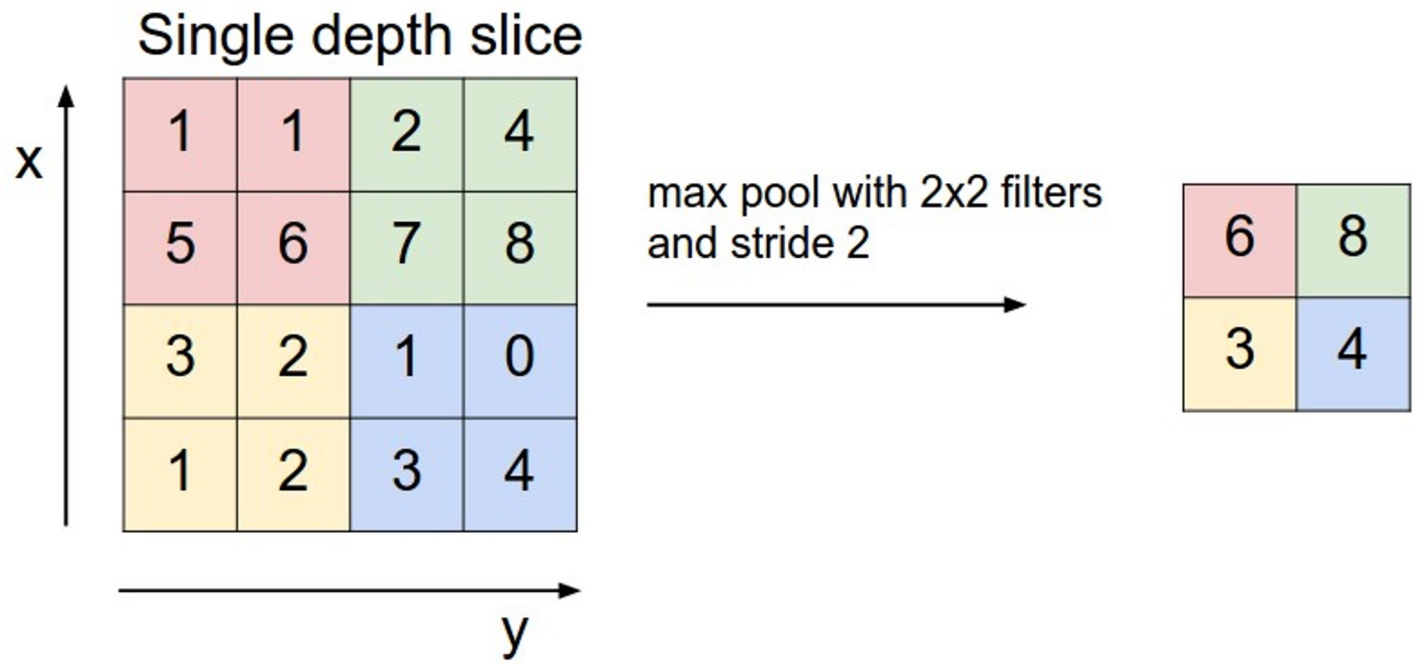

**Q:** how many 3x3 convolutions we should use to recognize a 100x100px kid

**A:** around 50... we need to increase receptive field faster!

# Kernel size choice heuristic

- Odd number per dimension (there should be the center of the kernel)
- If input image is larger than 128x128?
  - Use 5x5 or 7x7
  - and then quickly reduce spatial dimensions — then start working with *3×3* kernels
- Otherwise consider sticking with *1×1* and *3×3* filters.

# Pooling




Intuition: What is the highest kid-ness over this area?

# Pooling

Motivation:

. Reduce layer size by a factor

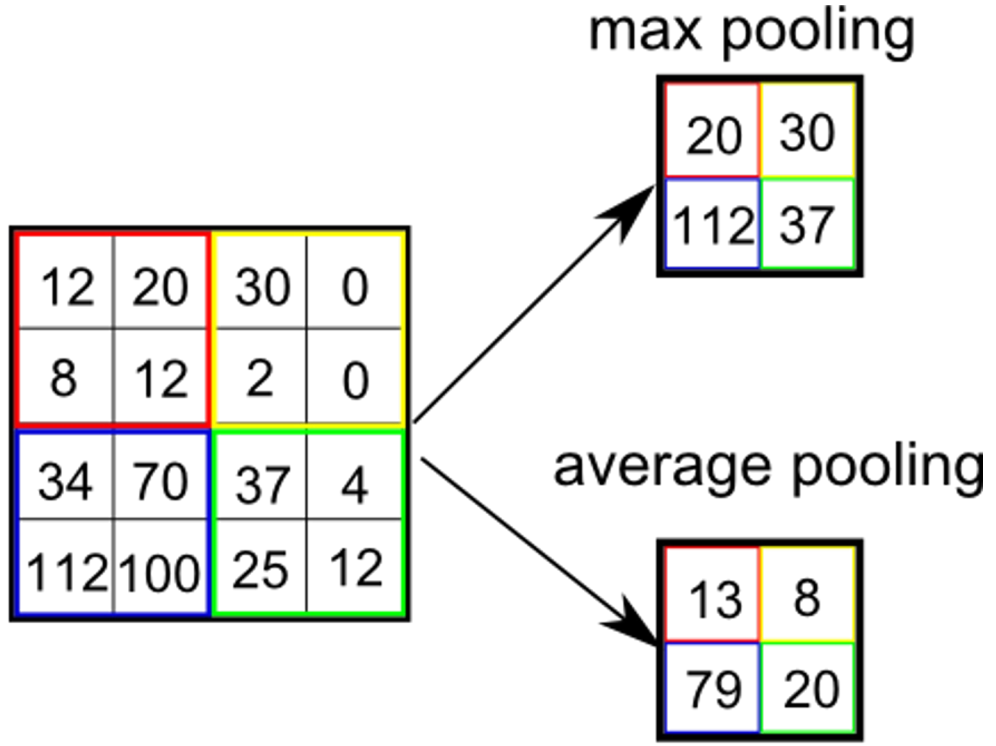
. Make NN less sensitive to small image shifts

Popular types:

. Max

. Mean(average)

**No params to train!**



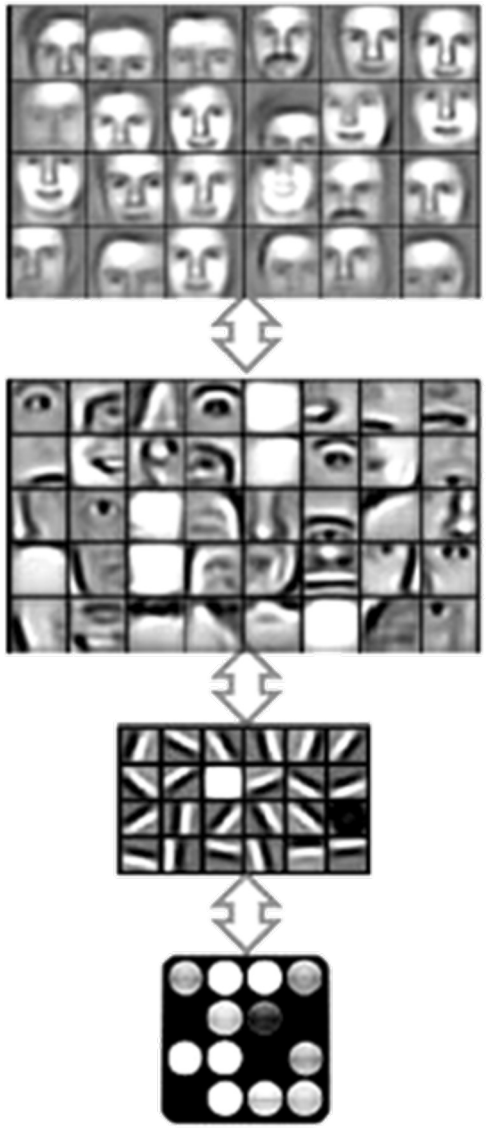
# Pooling Demo

- <Demo>

# Self-check

- How large is the receptive field for the following module:
  - Conv1d(3x1, stride=1)
  - MaxPool1d(2x1, stride=2)
- Answer:
- How many parameters should such network learn:
  - Input (100x100x3)
  - Conv2d(3x3, 2 out channels, bias=True)
  - MaxPool2d(3x3, stride=3, dilation=1)
- Answer:

# Convolutional features stacking



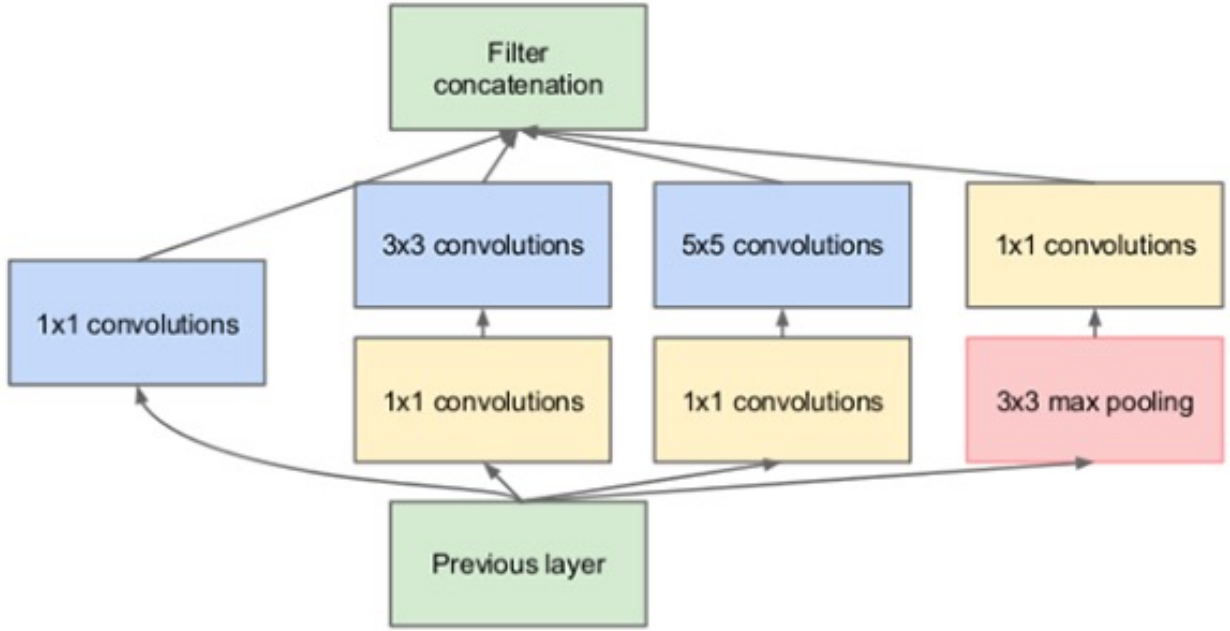
**Discrete Choices**

⋮

**Layer 2 Features**

**Layer 1 Features**

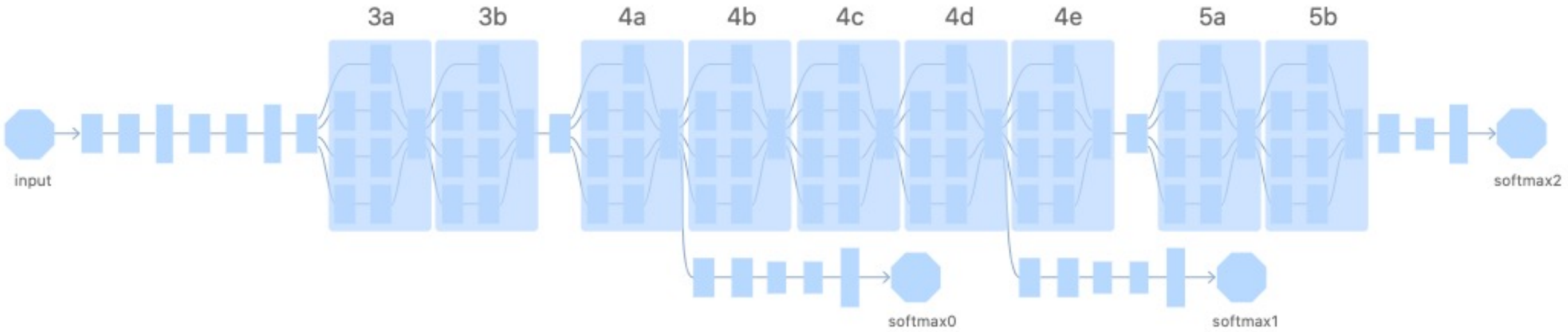
**Original Data**



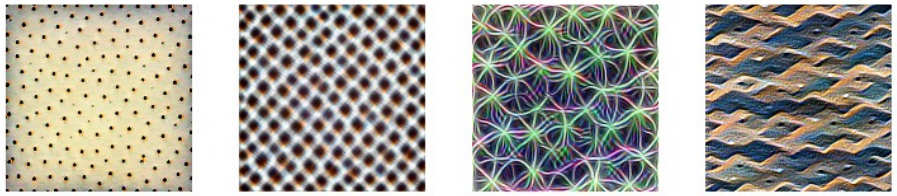

https://arxiv.org/pdf/1409.4842.pdf

# Computer vision example: GoogleNet

https://arxiv.org/pdf/1409.4842.pdf




**Layer 3a**



**Layer 4a**


**Layer 4e**


**Layer 5b**

https://distill.pub/2017/feature-visualization/appendix/

# Possible problems with large networks

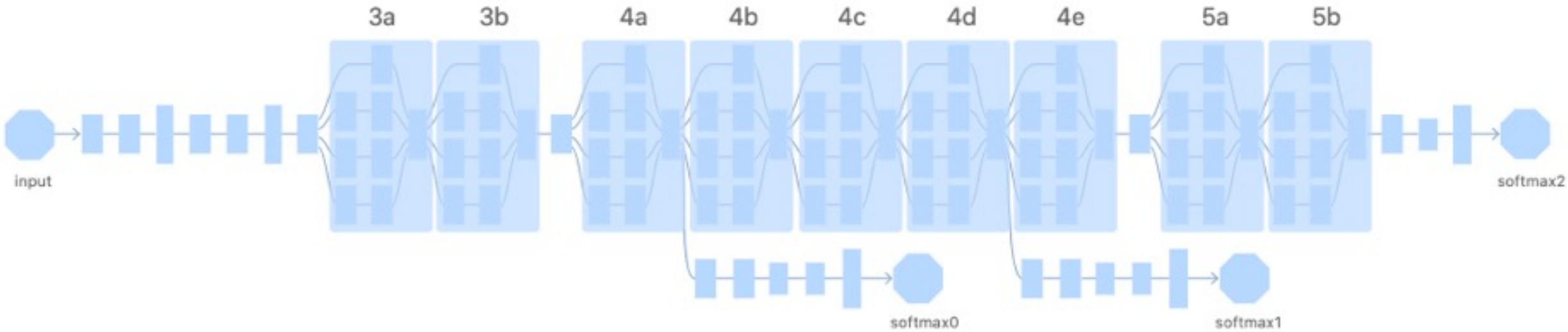
- MemoryError(0x...)
- Gradients can vanish
- Gradients can explode
- Activations can vanish
- Activations can explode

Possible solutions:

- Use pre-trained networks
- Different normalization techniques

  - BatchNorm, LayerNorm, InstanceNorm, LocalResponseNorm

# Combining networks




```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by
default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

# Other image-related tasks and libraries

General

- Image tagging, captioning, retrieval, morphing, encoding, upscaling
- Video processing, interpolation
- 3D point-clouds

HEP examples

- Jet tagging
- Particle tracking
- Calorimeter image analysis
- Cherenkov detector image analysis

- Torchvision,
- Kornia, Computer Vision Library for PyTorch
- MMF, A modular framework for vision & language multimodal research from Facebook AI Research (FAIR),
- PyTorch3D, https://pytorch3d.org/

# Moar info

- Dimensionality reduction using 1x1 convolutions, https://machinelearningmastery.com/introduction-to-1x1-convolutions-to-reduce-the-complexity-of-convolutional-neural-networks/
- Sparse convolutions, https://github.com/facebookresearch/SparseConvNet
- Gauge Equivariant Convolutional Networks, https://arxiv.org/pdf/1902.04615.pdf
- Dilated convolutions, http://www.erogol.com/dilated-convolution/
- Receptive field guide https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

# Convolutions wrap-up

- Images are the first-class citizens in the deep Learning world;
- Images are high-dimensional objects that demand powerful techniques for efficient processing:
  - Convolution;
  - Maxpooling;

  Which effectively makes neural network capable of dealing with certain kind of symmetries: translational, scale equivariant. For dealing with rotational and other kind of symmetries see Steerable CNNs (arXiv:1612.08498v1).
- Architecture selection:
  - pre-trained models;
  - Neural architecture search (see module on optimization);
- Further tasks: segmentation, object detection, saliency map.

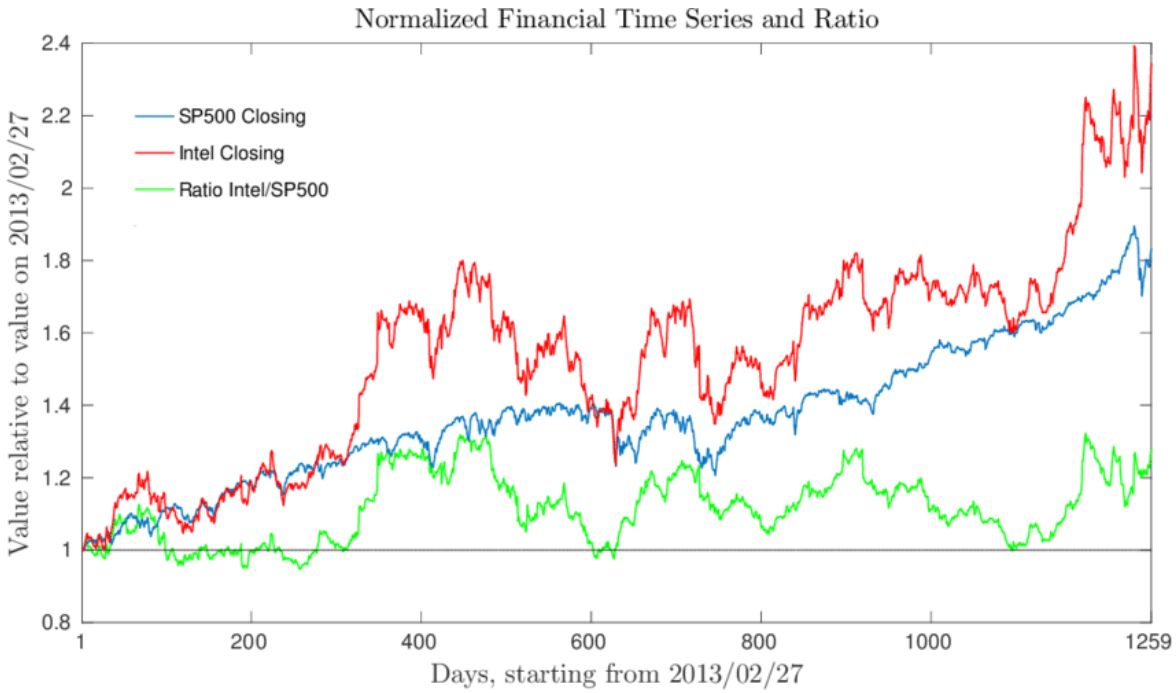
# Sequence modeling

Andrey Ustyuzhanin

# Sequential data

Represent:

- Time series:
  - Financial data, stock market
  - Healthcare: pulse rate, sugar level
- Text and speech
- Spatiotemporal data:
  - Self-driving and object tracking
  - Plate tectonic activity
- Physics:
  - Accelerator's, detector's data flows, anomaly detection
  - Tracking
- etc.



# 1. Sequence classification

- Problem statement:
  - $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ - sequence of objects $x_i \in V$
  - $y \in \{1 \ldots C\}$ - labels (C is a number of classes)
  - $\{(\mathbf{x}^1, y_1), (\mathbf{x}^2, y_2), \ldots, (\mathbf{x}^m, y_m)\}$ - training data of size m

**Aim**: predict y given

Examples:

- Medicine: **x** - pulse rate, y - health (healthy, unhealthy)
- Opinion mining: **x** - sentence, y - sentiment (positive, negative)
- **Physics**: ghost tracks filtering. **x** - sequence of hits for some track, y - class (ghost or not)

# Sequence classification. Ghost tracks filtering

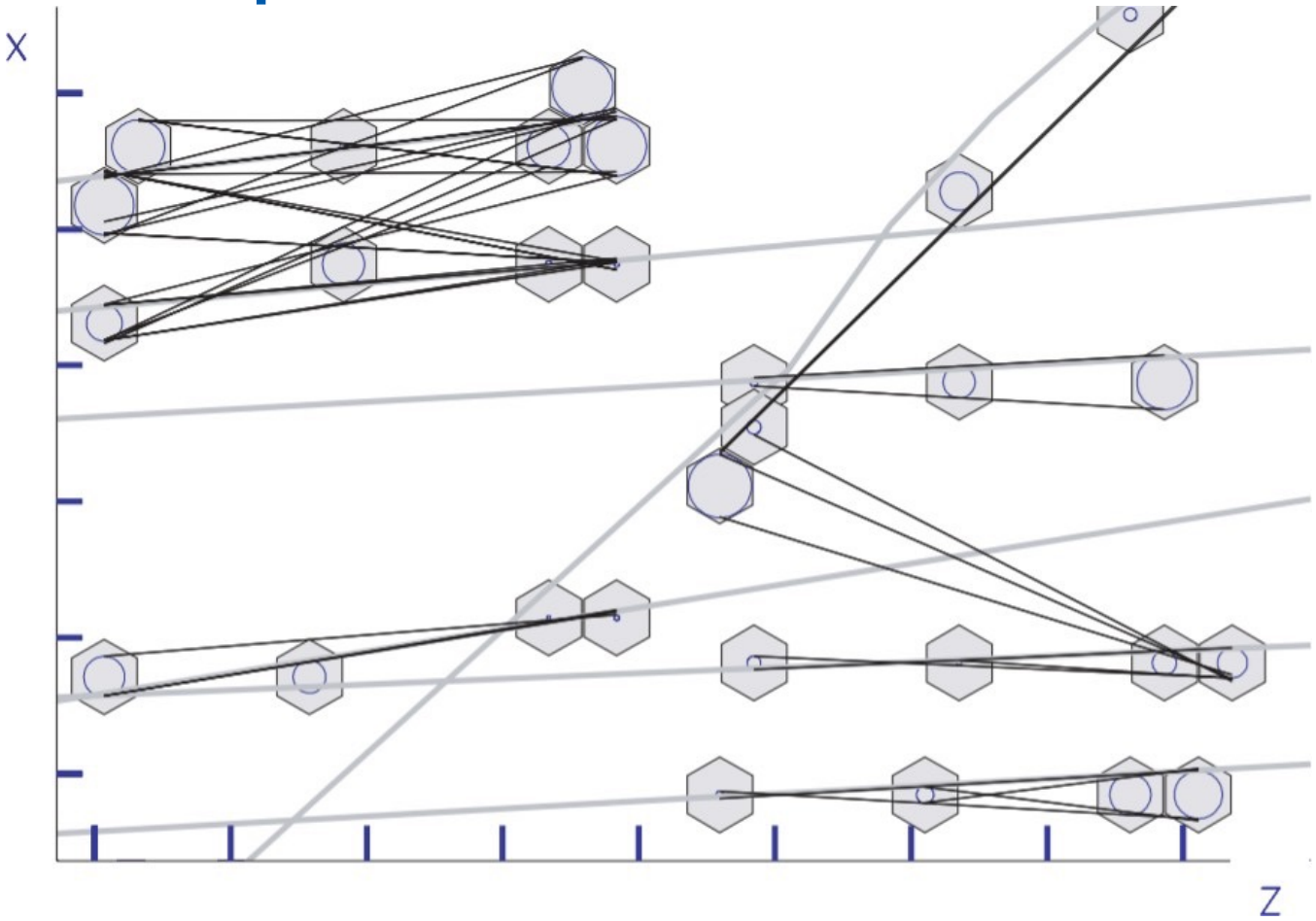



Figure: link

**Problem**: which set of hits form a true track, and which are just a random track-like set of hits (ghost)?

# 2. Sequence labelling

Problem statement:

- $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ - sequence of objects $x_i \in V$
- $y = \{y_1, y_2, \ldots, y_n\}$ - **sequence** of labels, $y_i \in \{1 \ldots C\}$ (C is a number of classes)
- $\{(x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m)\}$ - training data of size m
- **Aim**: predict y given x

Examples:

- Genome annotation: **x** - DNA, y - genes
- Hardware: **x** are system logs, y is system state at each time
- **Physics**: track reconstruction. **x** - set of hits, y - set of tracks (each hit is assigned to some track)

# Sequence labelling. Track reconstruction example

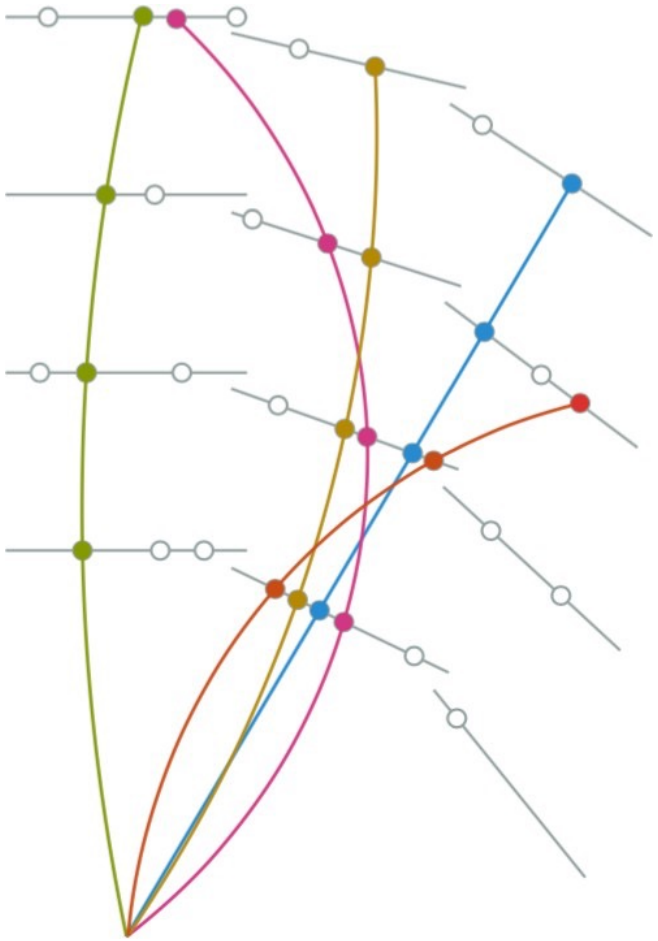


Figure:
http://old.inspirehep.net/record/1729722/plots

# 3. Sequence to sequence transformation

- Problem statement:
  - **x** = $\{x_1, x_2, \ldots, x_n\}$ - source sequence of objects $x_i \in V_{source}$
  - **y** = $\{y_1, y_2, \ldots, y_n\}$ - target sequence of objects $y_i \in V_{target}$
  - $\{(x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m)\}$ - training data of size m
  - **Aim**: fit transformation from x to y Examples:
  - Speech to text - **x** is speech, y is text
  - Machine translation - **x** are sentences in English, y are sentences in German
  - Logs compression - **x** are raw logs, y are compressed logs

# Sequence transformation. Example

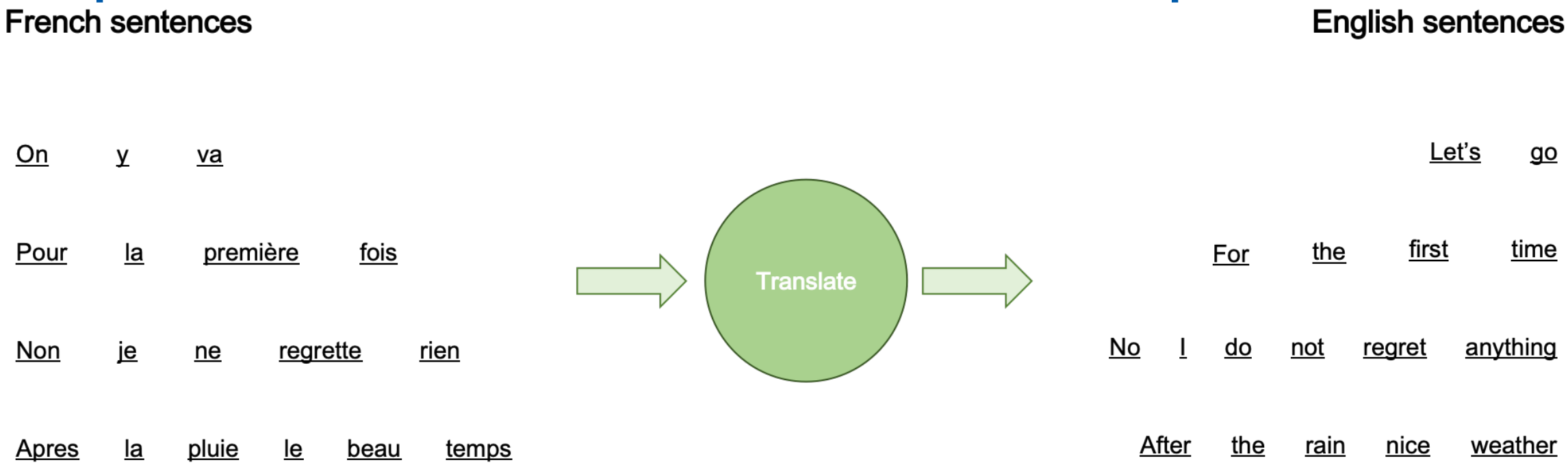



Figure: https://buomsoo-kim.github.io/attention/2020/01/12/Attention-mechanism-3.md/

# Recurrent neural networks

# Problem

**Question:** How to perform elementwise processing of sequential data effectively preserving its context?

**Answer:** Let's keep all the context in a separate term (hidden state).

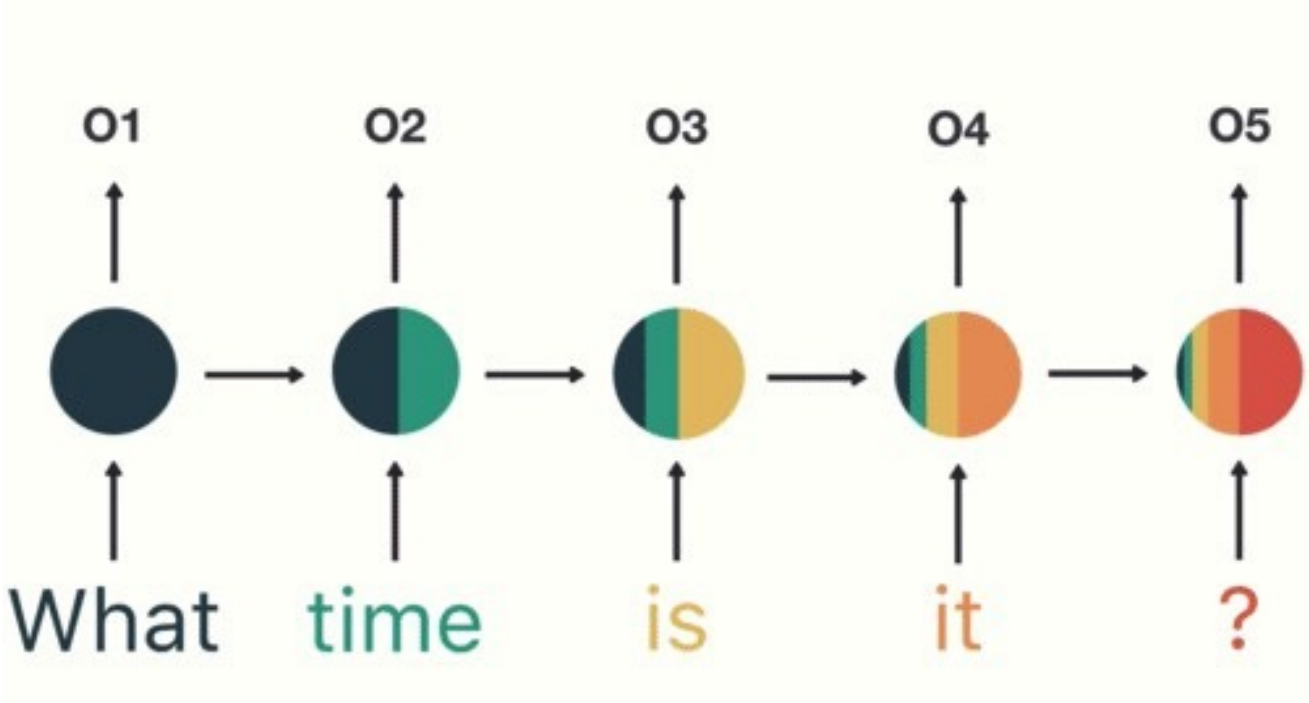
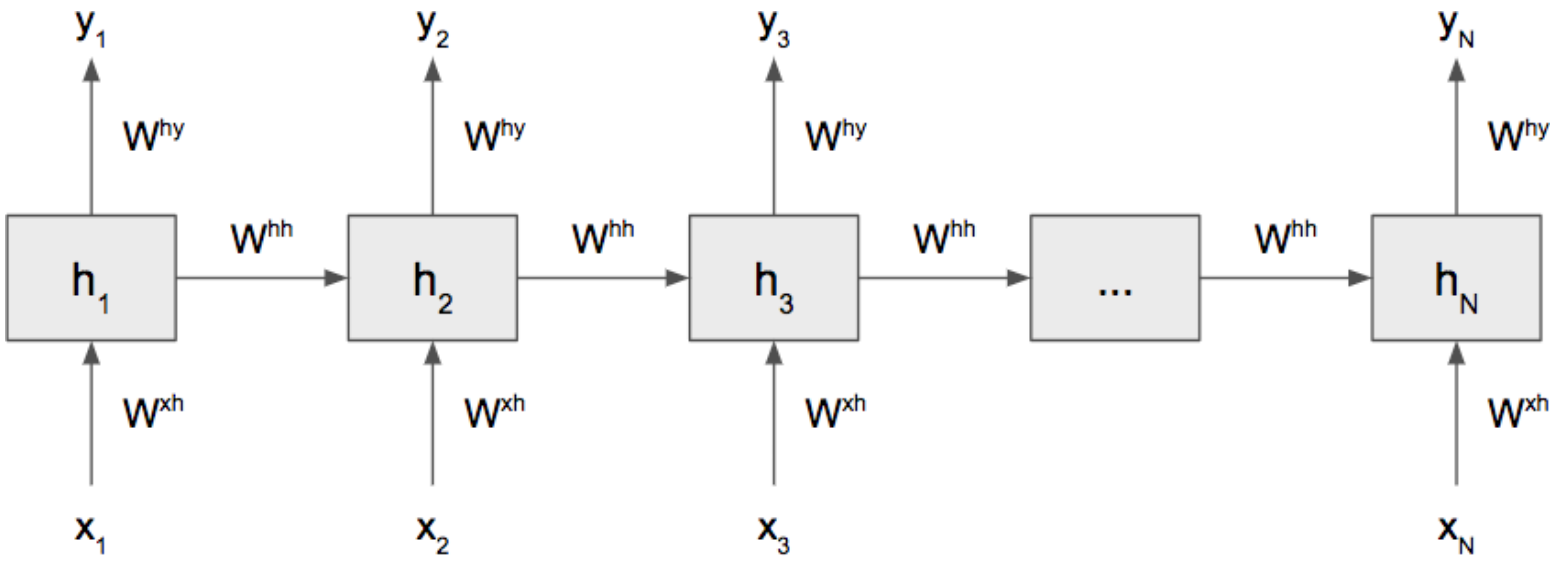



Figure: towardsdatascience.com

# Recurrent neural networks (RNN)

- Input: $(x_i, h_i)$ from sequence of input vectors $x = \{x_1, x_2, \ldots, x_n\}$ and hidden states (context) $h = \{0, h_2, \ldots, h_{n-1}\}$, $x_i \in R^{d_{in}}$, $h \in R^{d_{hid}}$
- Output: $(y_i, h_{i+1})$ from sequence of output vectors $y = \{y_1, y_2, \ldots, y_n\}$ and next hidden states (context) $h = \{h_2, h_3 \ldots, h_n\}$, $y_i \in R^{d_{out}}$, $h \in R^{d_{hid}}$



# Stacked RNN

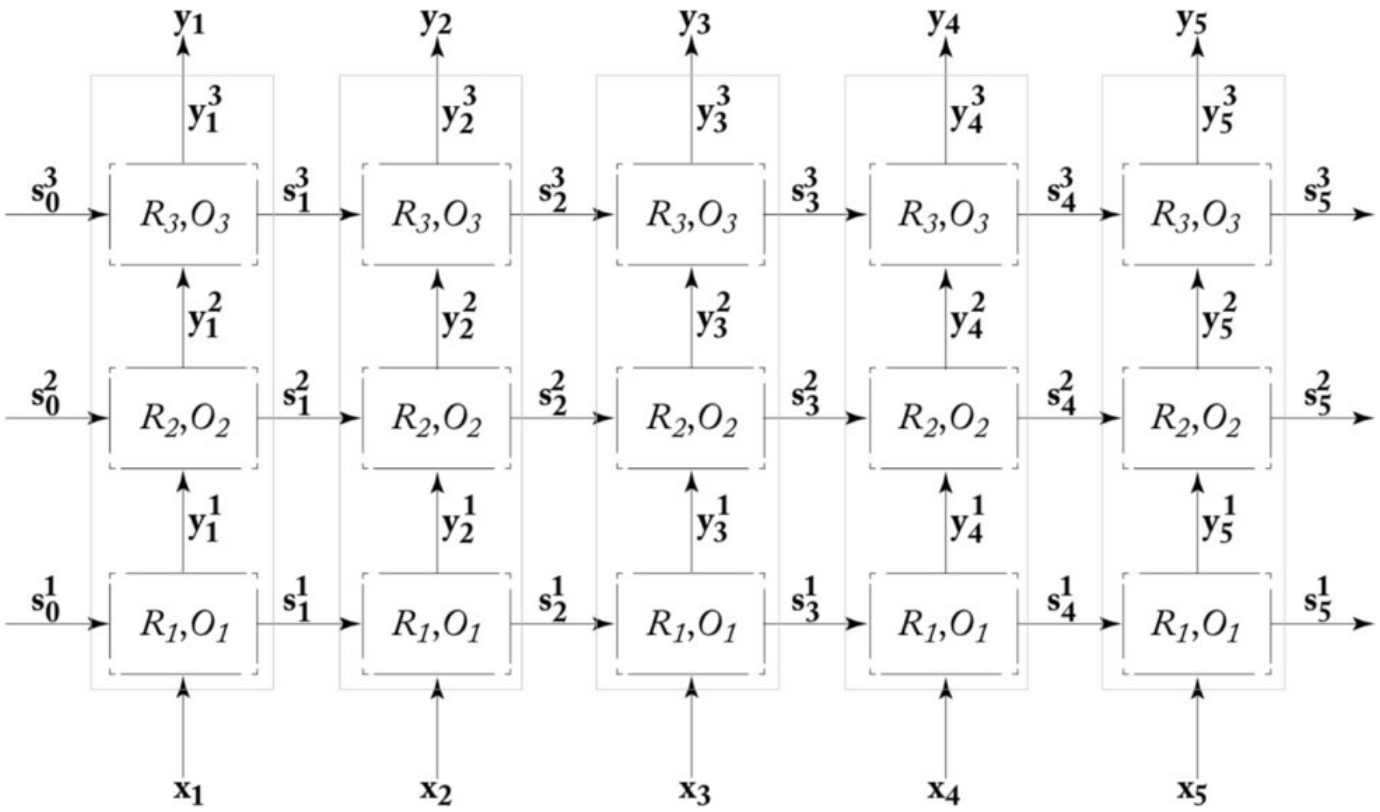
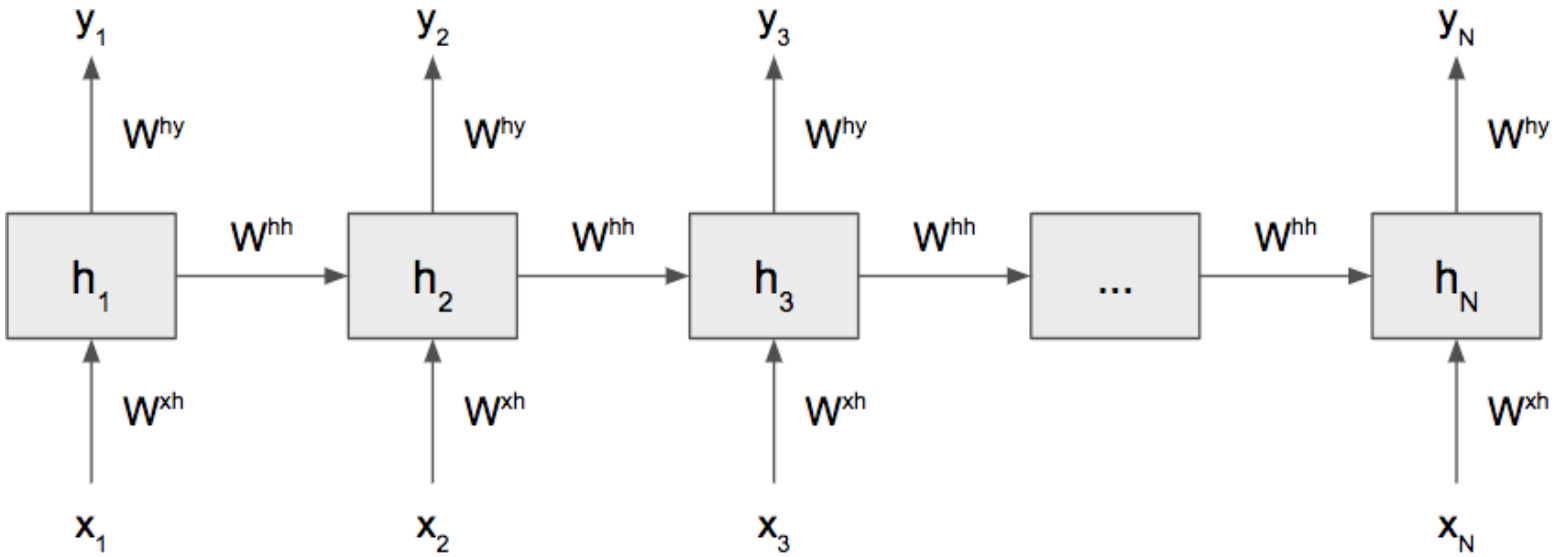



Figure: Goldberg, Yoav. Neural network methods for natural language processing

# RNN in sequence modeling

- **Sequence classification** - $x = \{x_1, x_2, \ldots, x_n\} \rightarrow y_n$, $y_n$ is final prediction, x is input sequence
- **Sequence labelling** - $x = \{x_1, x_2, \ldots, x_n\} \rightarrow y = \{y_1, y_2, \ldots, y_n\}$, x is input sequence and y is labeled sequence
- **Sequence transformation** - $x = \{x_1, x_2, \ldots, x_n\} \rightarrow y = \{y_1, y_2, \ldots, y_n\}$, x is input sequence and y is output sequence



# RNN vs. Fully-connected. Advantages and disadvantages

Advantages:

- Preserves sequence context
- Good performance and flexibility
- Effectively handles longer sequences (longer memory)
- Handles sequences with variable size
- Model complexity does not depend on sequence length

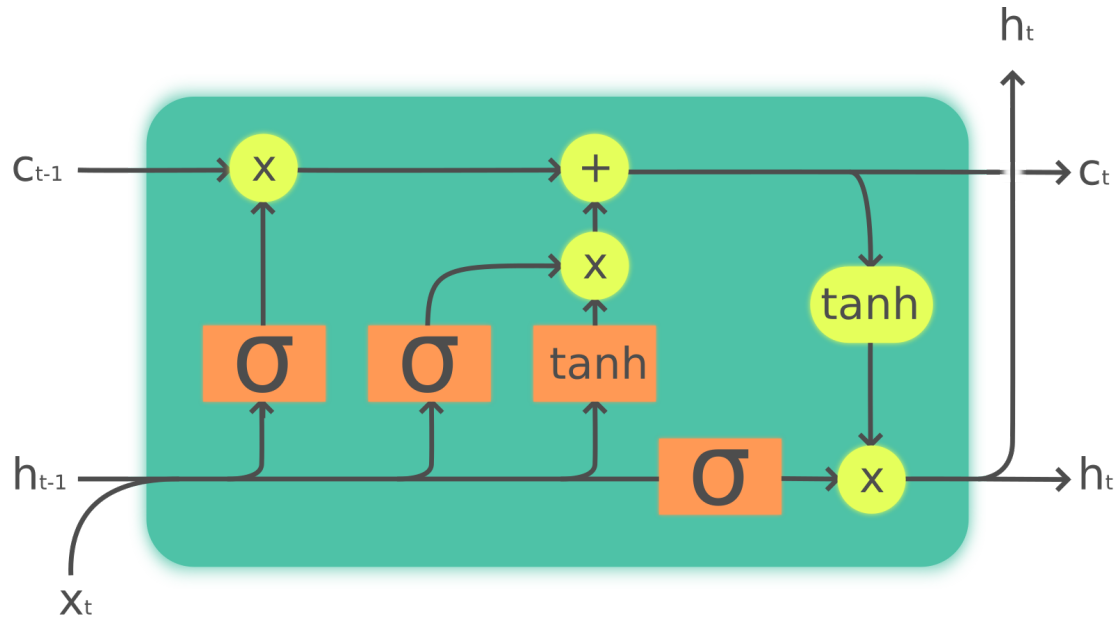
Disadvantages:

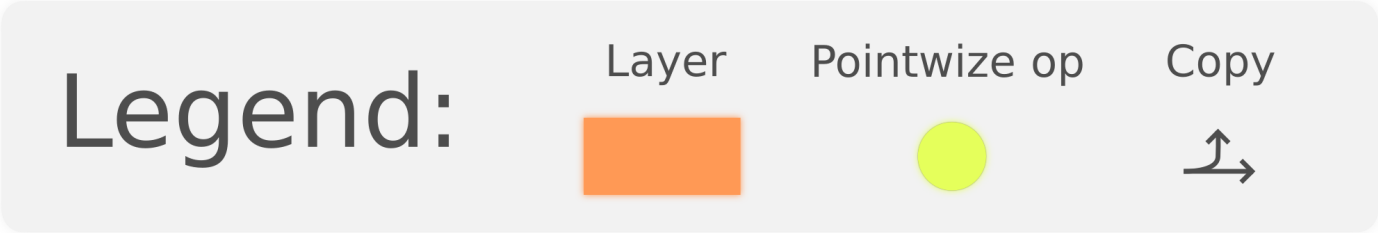
- **Moves in sequence in the only direction** (Some sequences must be processed in both directions)
- Still has short memory due to vanishing gradients problems
- Exploding gradients due to cycles in recurrent connections

# LSTM

# LSTM. Idea

**Idea**: regulate the flow of context by special gates (auxilary neural networks) and split memory (hidden state) in two separate terms: short-term $h$ and long-term $c$



# LSTM vs. RNN

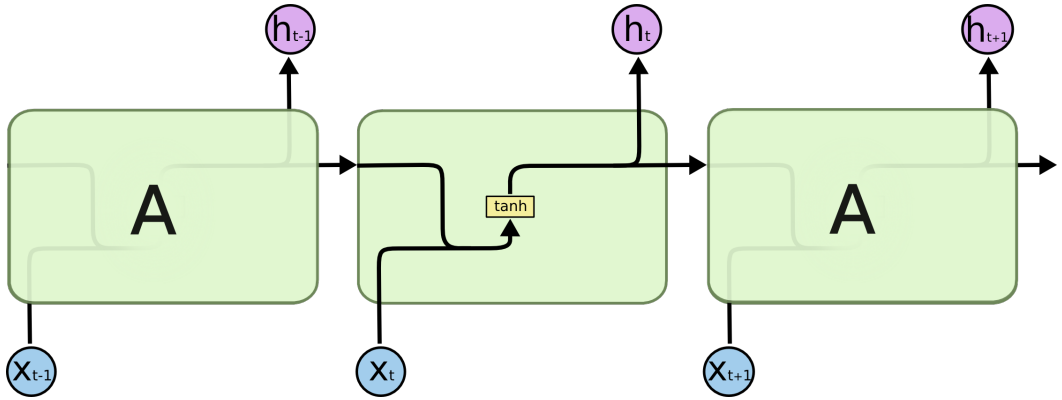



Figure 1: Basic RNN

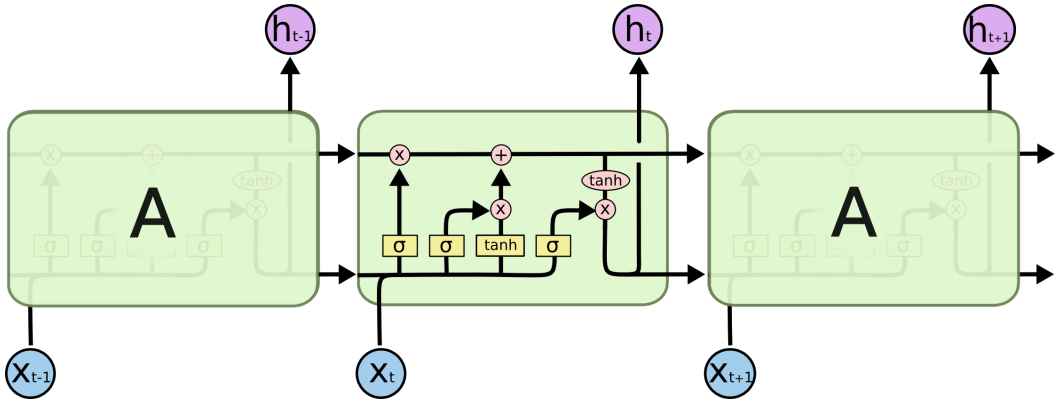
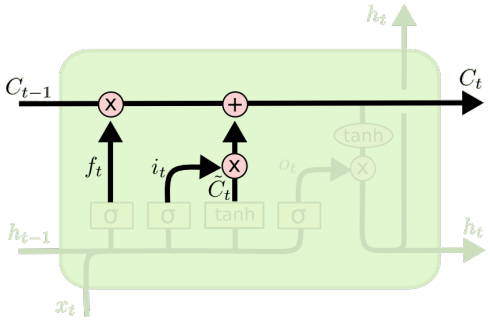



Figure 2: LSTM

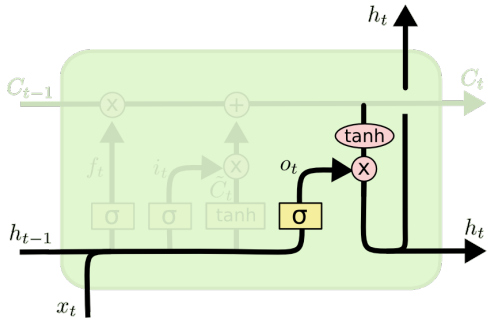
Figures: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# LSTM inside




$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 3: Long-memory update




$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 4: Short-memory and output update

Figures: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# LSTM vs. RNN. Advantages and disadvantages

Advantages:

- Preserves sequence context
- Good performance and flexibility

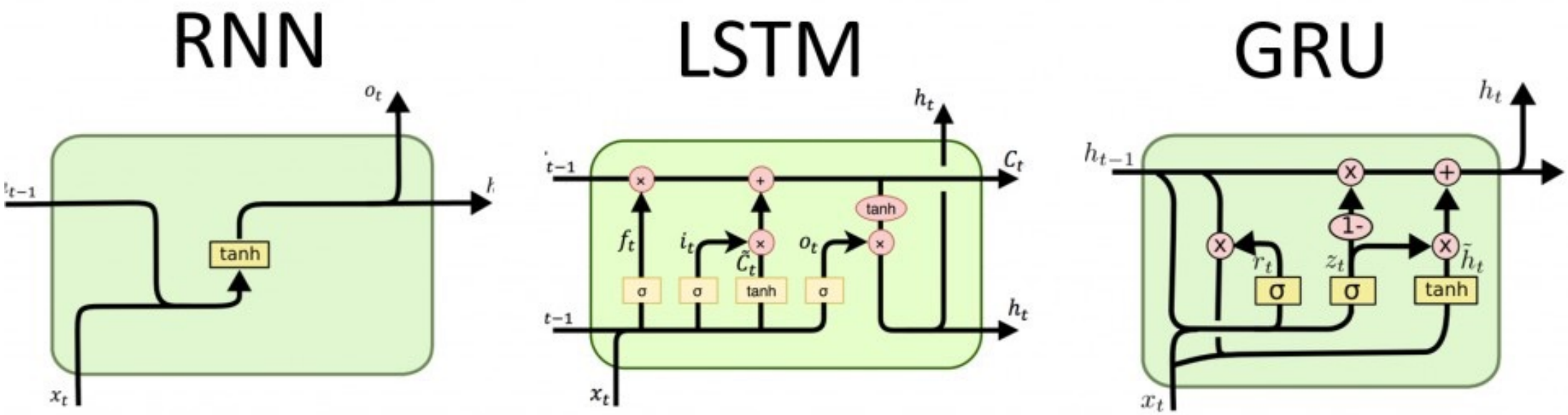
Disadvantages:

- ~~Moves in sequence in the only direction~~ (Some sequences must be processed in both directions)
- ~~Short memory due to vanishing gradients problems~~
- ~~Exploding gradients due to cycles in recurrent connections~~
- **Slower** and heavier than RNN

# GRU

# GRU

**Idea**: combine long-term and short-term updates in single channel



# GRU vs. LSTM. Advantages and disadvantages

Advantages:

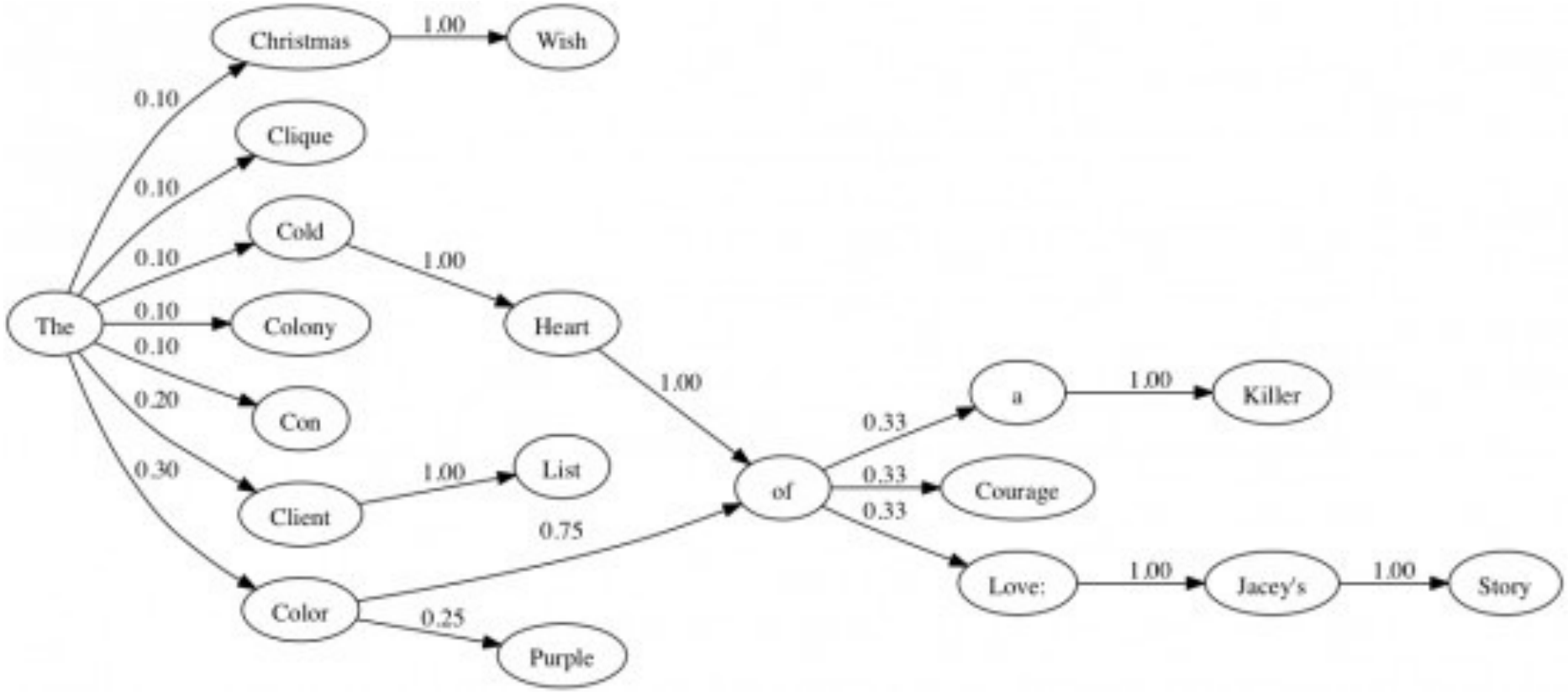
- All context stored in single hidden state
- Lower model complexity
- Faster

Disadvantages:

- Shorter memory

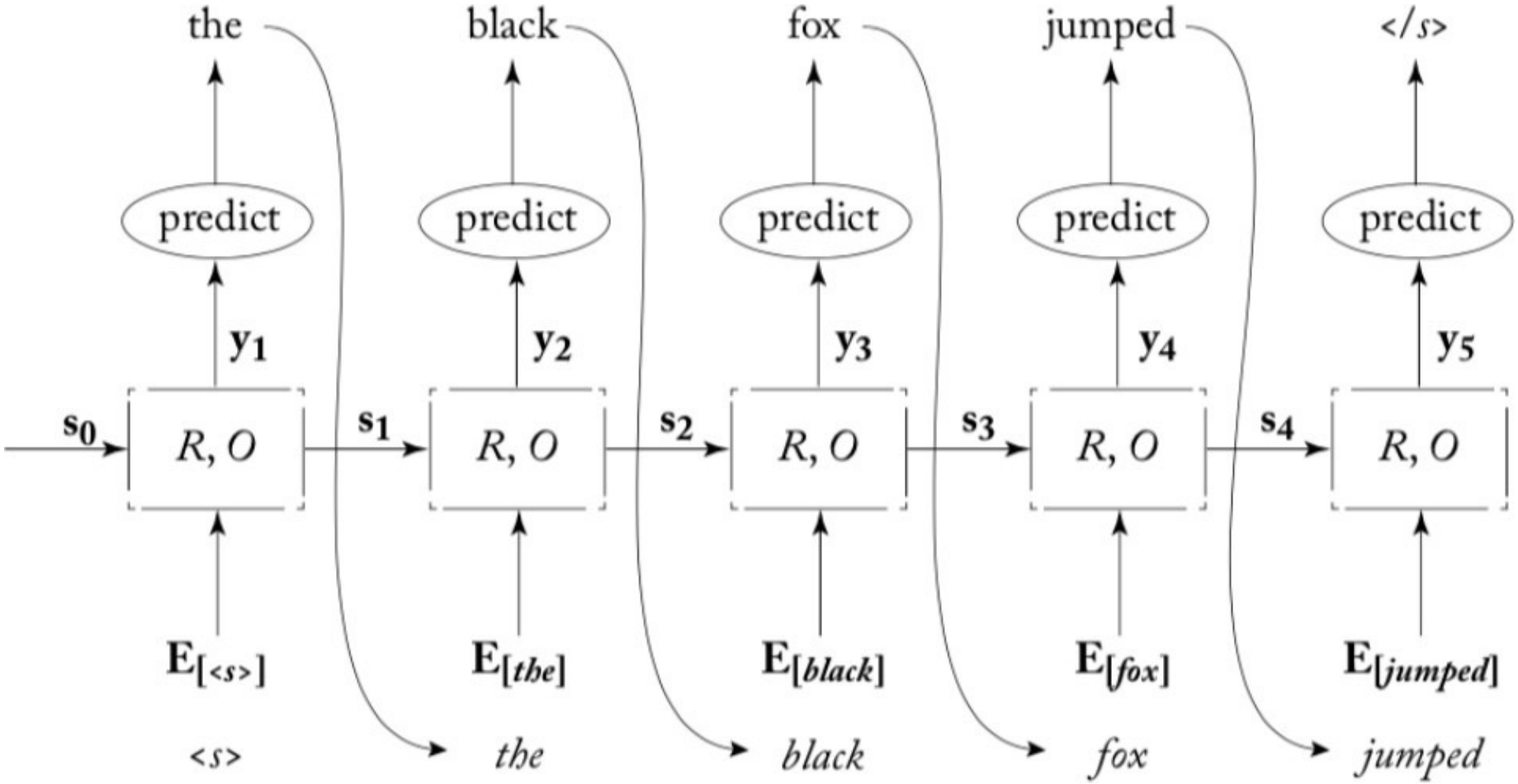
# Probabilistic sequence modeling

# Probabilistic sequence modeling




How to fit $p(x_i|x_{1:i})$ and generate sequence $x_i \sim p(x_i|x_{1:i})$?

# Probabilistic sequence modeling using RNN



# Probabilistic sequence modeling using RNN

- Examples of generated texts:

"The surprised in investors weren't going to raise money. I'm not the company with the time there are all interesting quickly, don't have to get off the same programmers..."

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

- Examples of generated MIDI music: https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5

# Pros and cons of RNNs

Advantages:

- RNNs are popular and successful for variable-length sequences
- The gating models such as LSTM are suited for long-range error propagation

Problems:

- No transfer learning options
- The sequentiality prohibits parallelization within instances
- Long-range dependencies still tricky, despite gating (long-term dependencies are actually not so long due to gradient vanishing)

# Conclusion

- Each data exhibits certain kind of symmetry, which need to be recognized by a learner (neural network, NN) in order to generalize successfully;
- NN architecture is our assumptions on data symmetry encoded as a data transformation pipeline
  - Images: translation, scale -> convolutions;
  - Sequences: context -> RNNs, LSTMs;
- PyTorch provides a flexible toolbox for writing such pipelines (almost) effortlessly;
- Rich PyTorch ecosystems spans widely.

# Thank you!

andrey.ustyuzhanin@constructor.org

anaderiRu

Andrey Ustyuzhanin

# More on PyTorch

- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/tutorials/beginner/ptcheat.html
- http://neuralnetworksanddeeplearning.com/chap2.html
- https://www.khanacademy.org/math/differential-calculus/dc-chain
- https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/

# Bi-RNN

# Bi-RNN

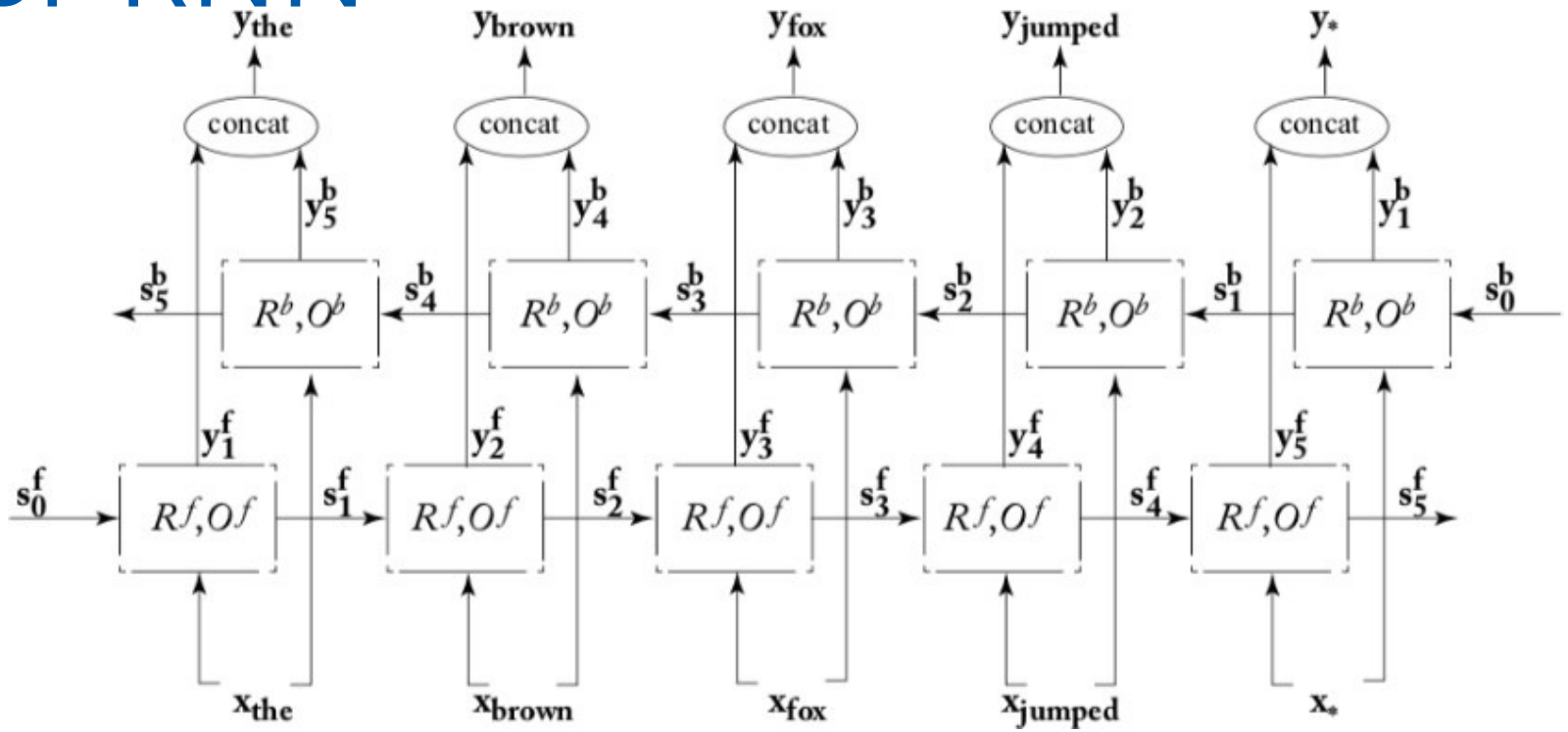



Figure: Goldberg, Yoav. Neural network methods for natural language processing

# Bi-RNN. Advantages and disadvantages

Advantages:

- Preserves sequence context
- Good performance and flexibility

Disadvantages:

- ~~Moves in sequence in the only direction~~ (Some sequences must be processed in both directions. For instance, words meaning depends both on previous and further context)
- **Short memory** due to vanishing gradients problems
- **Exploding gradients** due to cycles in recurrent connections