Natural Language Processing with PyTorch

Week 2 Deep Neural Networks in PyTorch

https://github.com/juneoh/fastcampus-pytorch-nlp

Review & Warranty

Contents

1. Convolutional Neural Networks
    - The convolutional layer
    - Batch normalization
    - Residual connections
    - Variations: dilated convolution, deconvolution, separable convolution

Contents

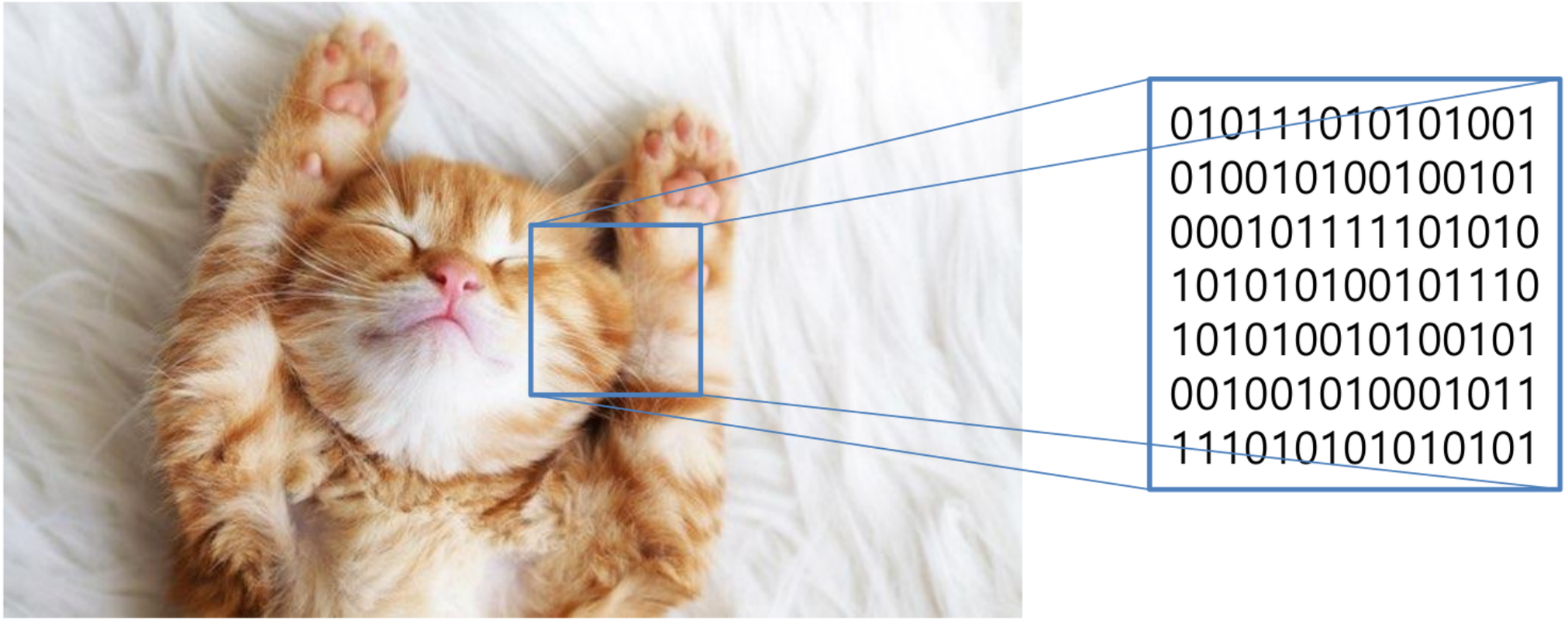
Contents

Convolutional Neural Networks

- The convolutional layer
- Batch normalization
- Residual connections
- Variations: dilated convolution, deconvolution, separable convolution

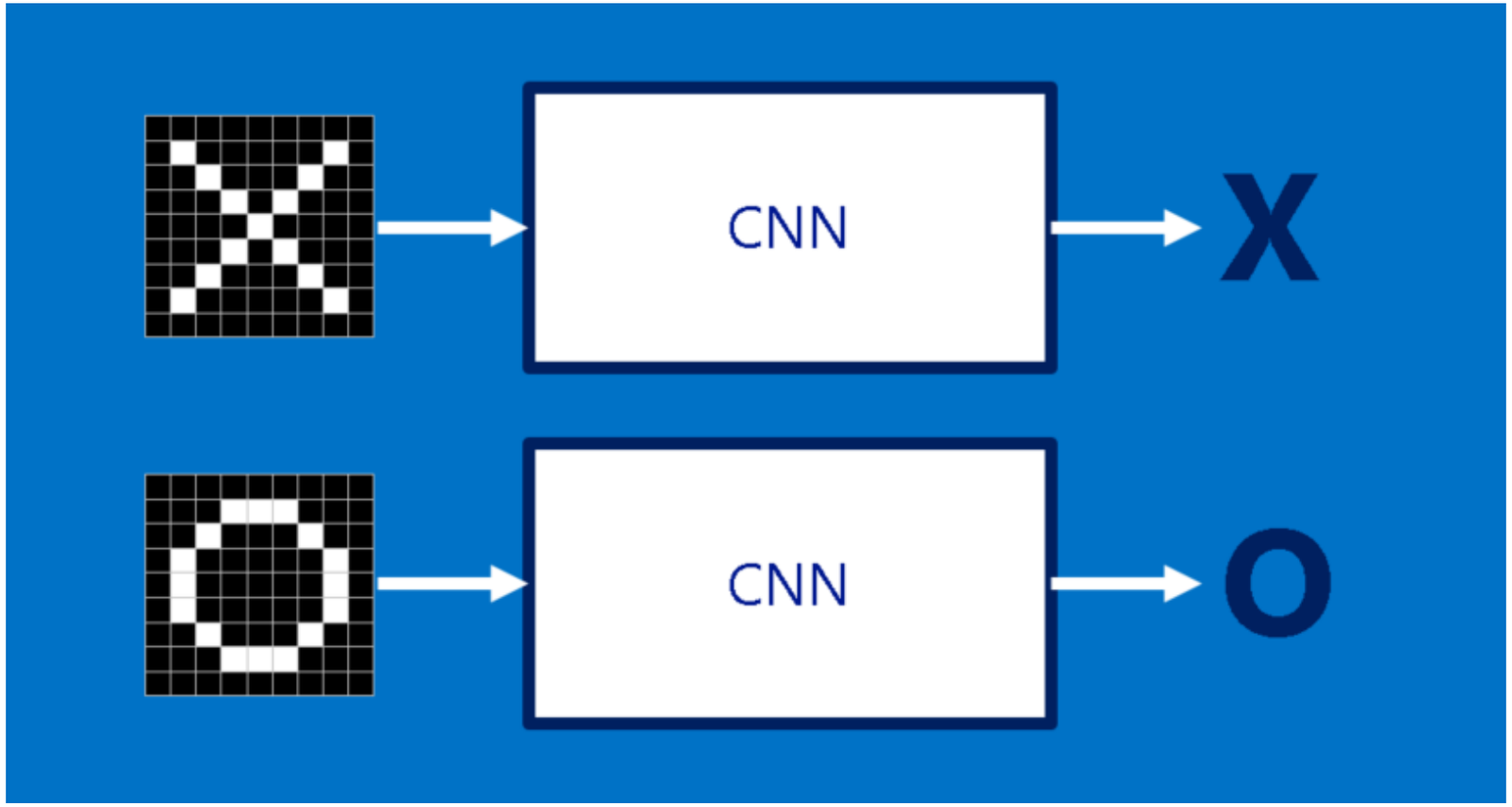
The convolutional layer

The convolutional layer



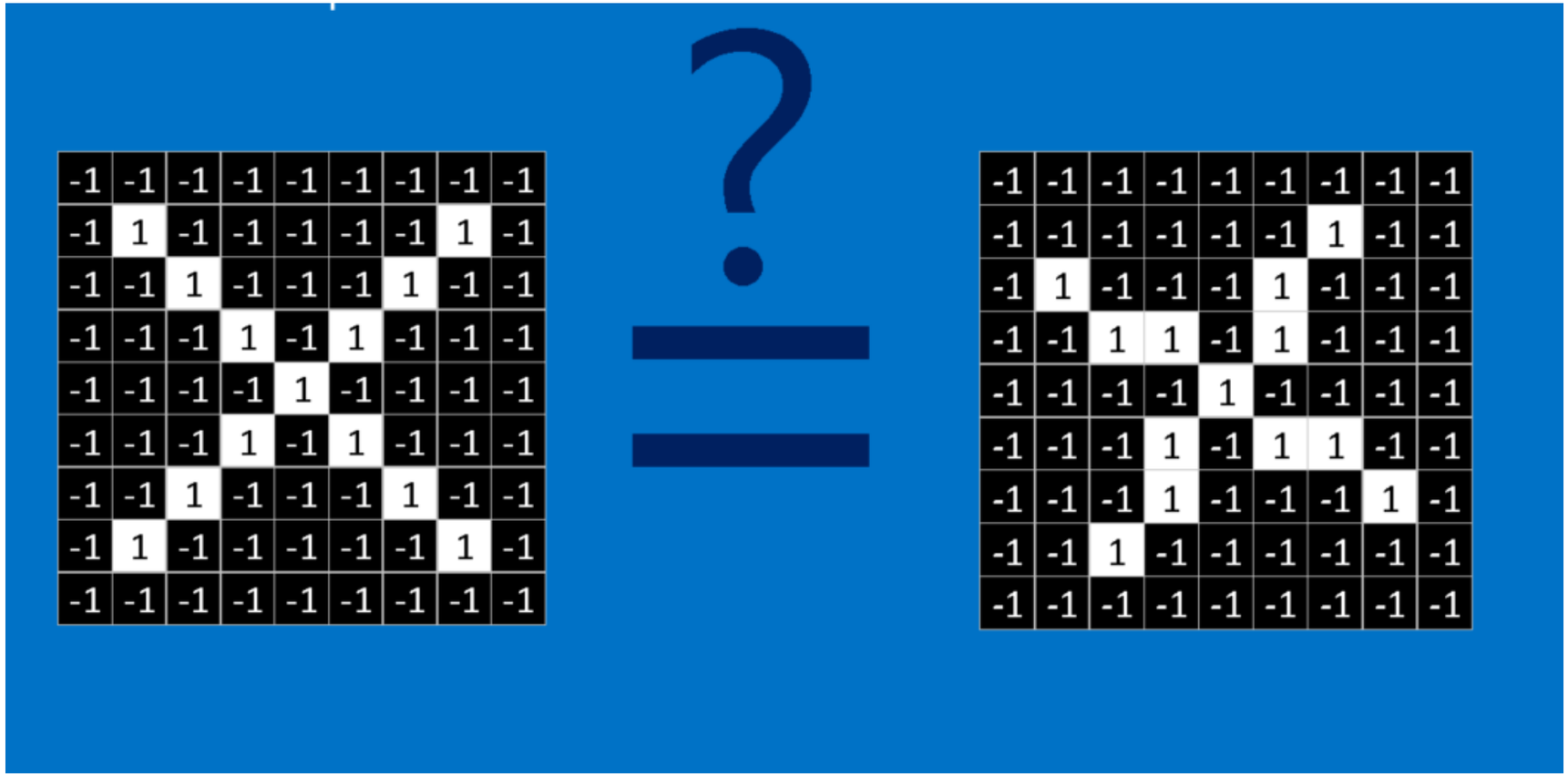
https://github.com/GunhoChoi/PyTorch-FastCampus/blob/master/03_CNN_Basics/CNN_basic.pdf

The convolutional layer



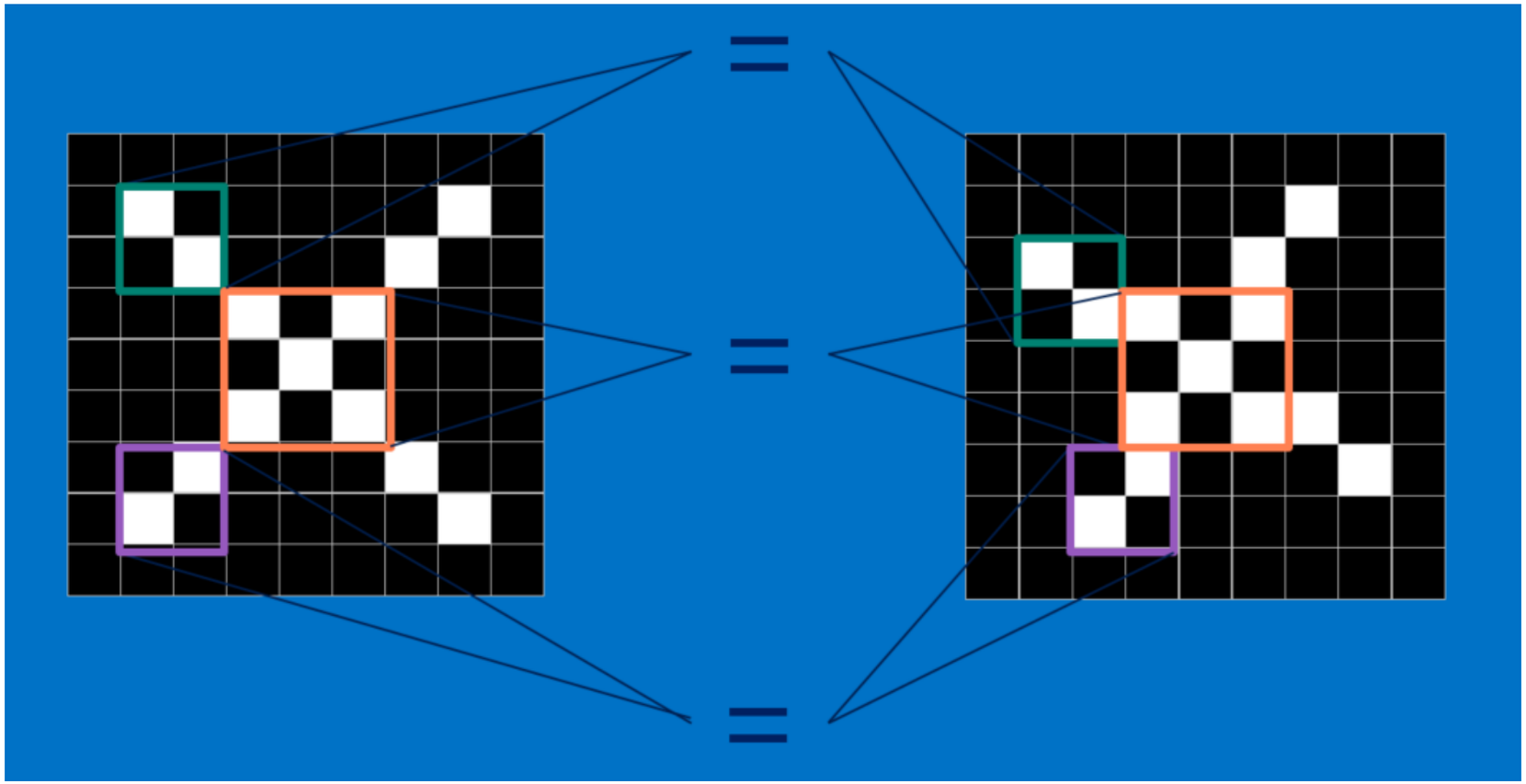
https://brohrer.github.io/how_convolutional_neural_networks_work.html

The convolutional layer



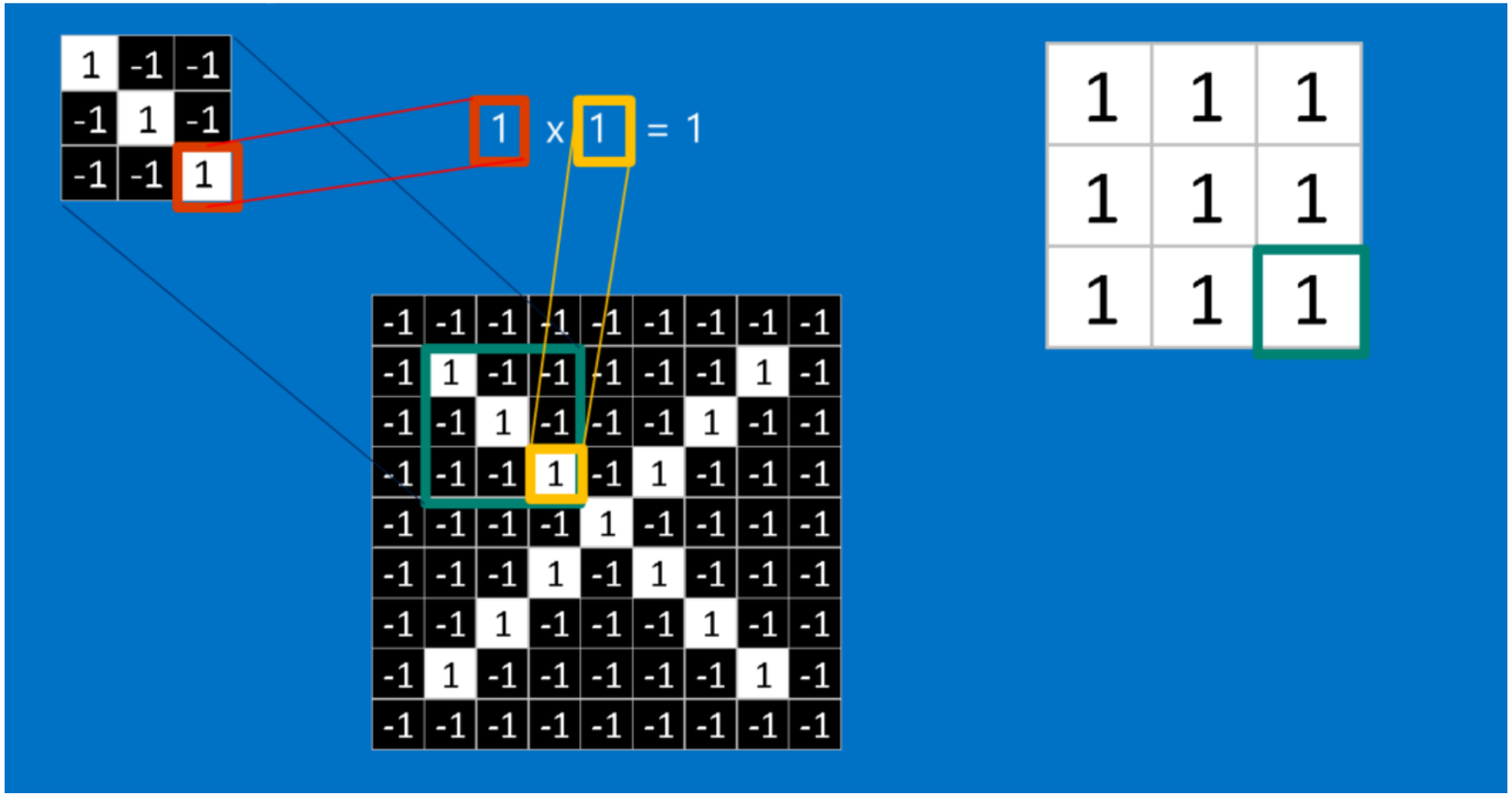
https://brohrer.github.io/how_convolutional_neural_networks_work.html

The convolutional layer



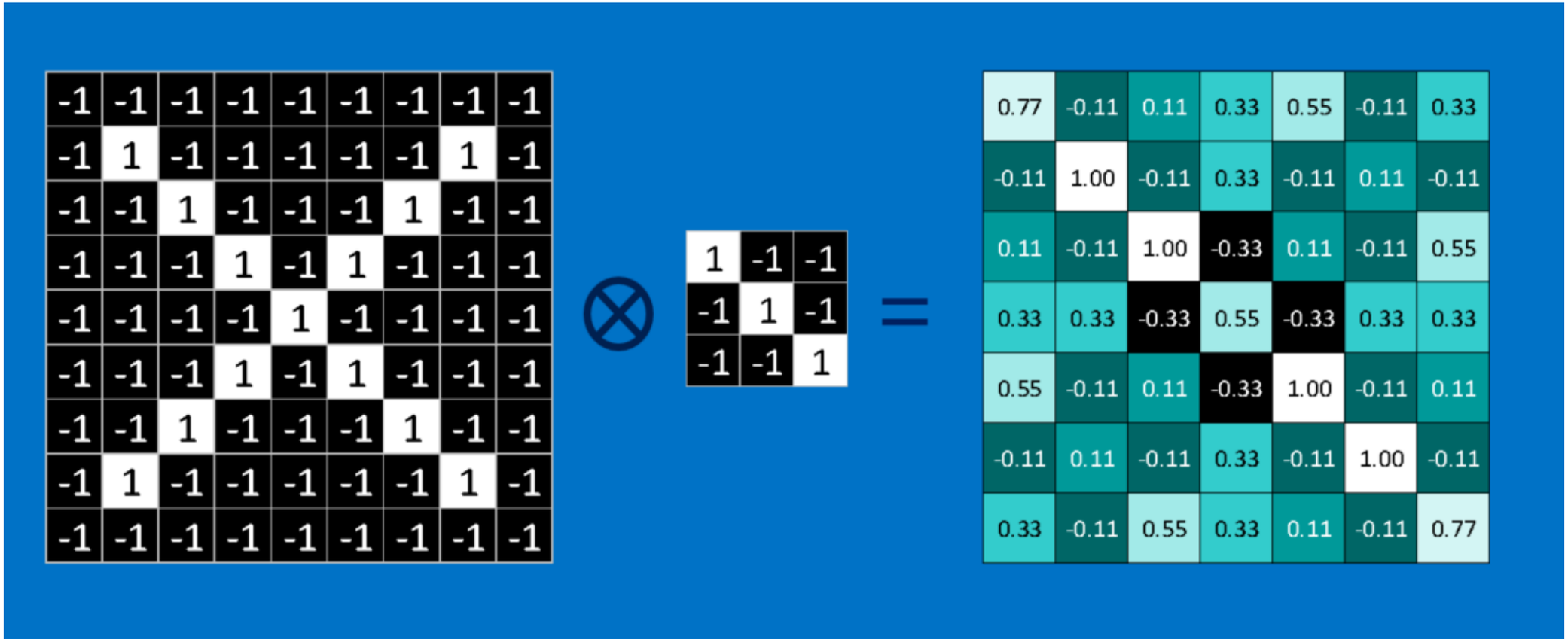
https://brohrer.github.io/how_convolutional_neural_networks_work.html

The convolutional layer

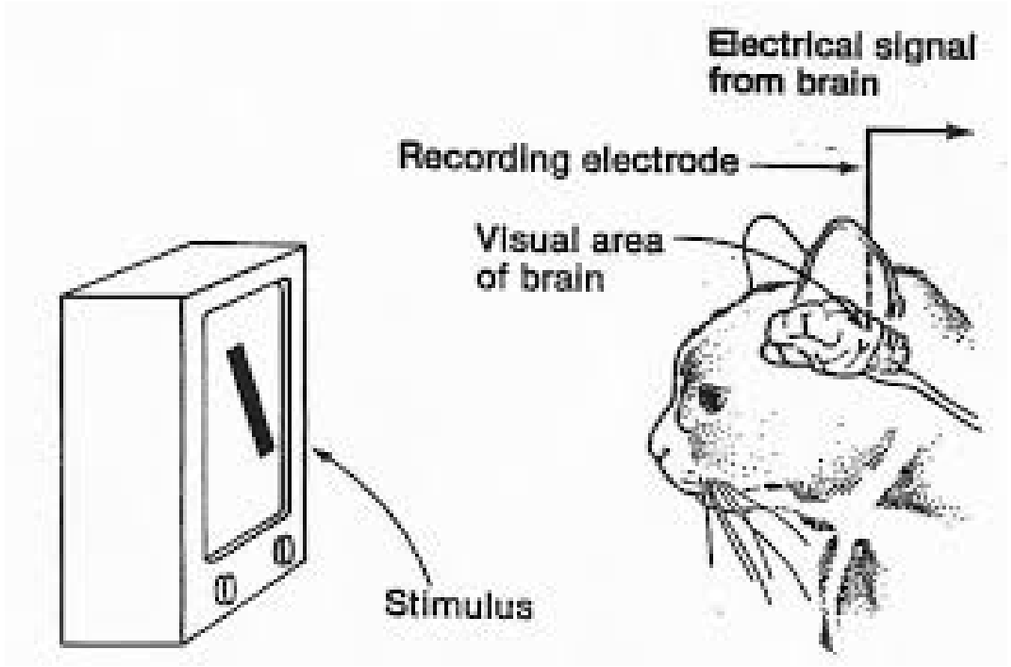


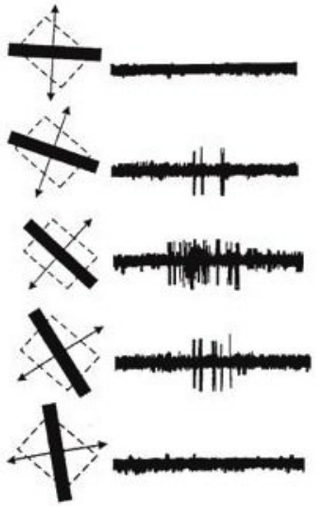
https://brohrer.github.io/how_convolutional_neural_networks_work.html

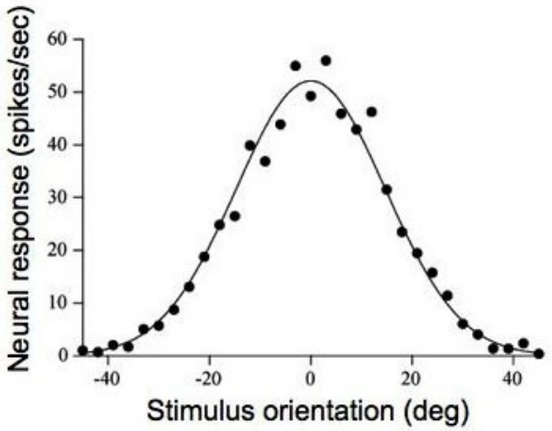
The convolutional layer



The convolutional layer



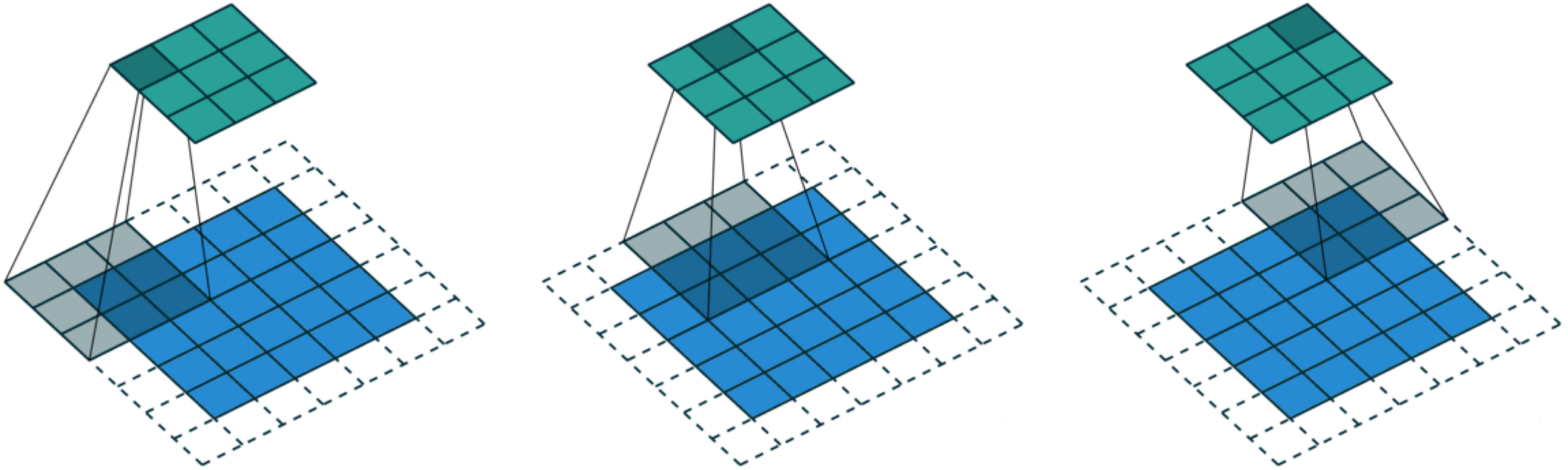




Hubel & Wiesel, 1968

Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex

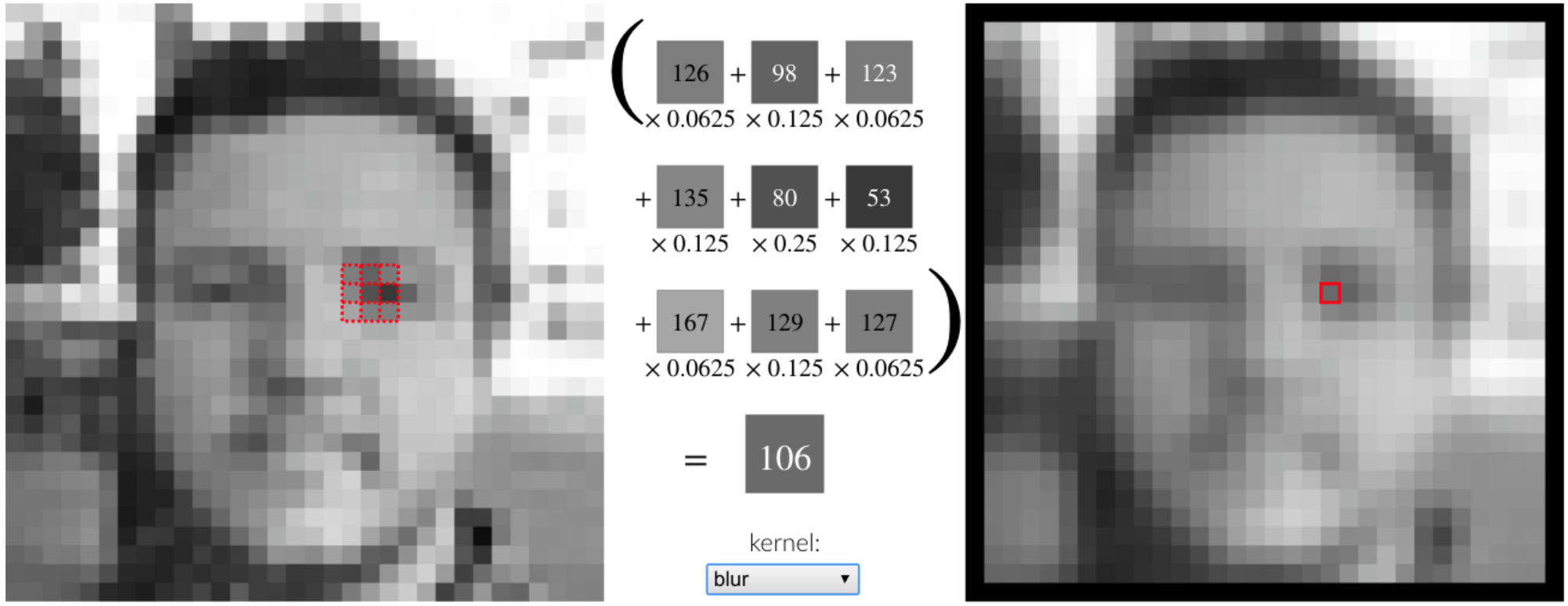
The convolutional layer



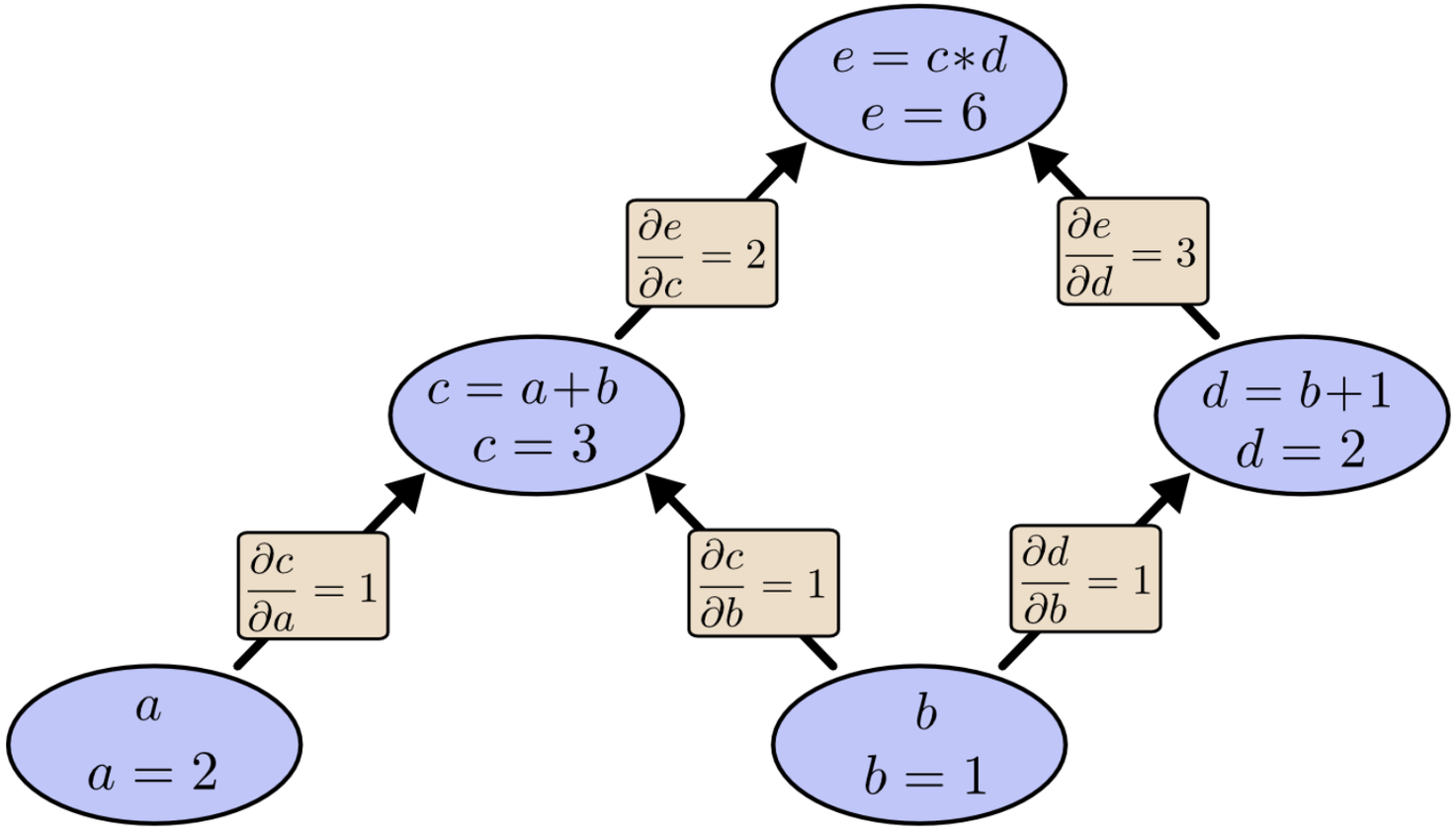
```
torch.nn.Conv2d(in_channels=1, out_channels=1, kernel=(3, 3),
stride=2, padding=1, dilation=1, bias=False)
```


https://github.com/vdumoulin/conv_arithmetic

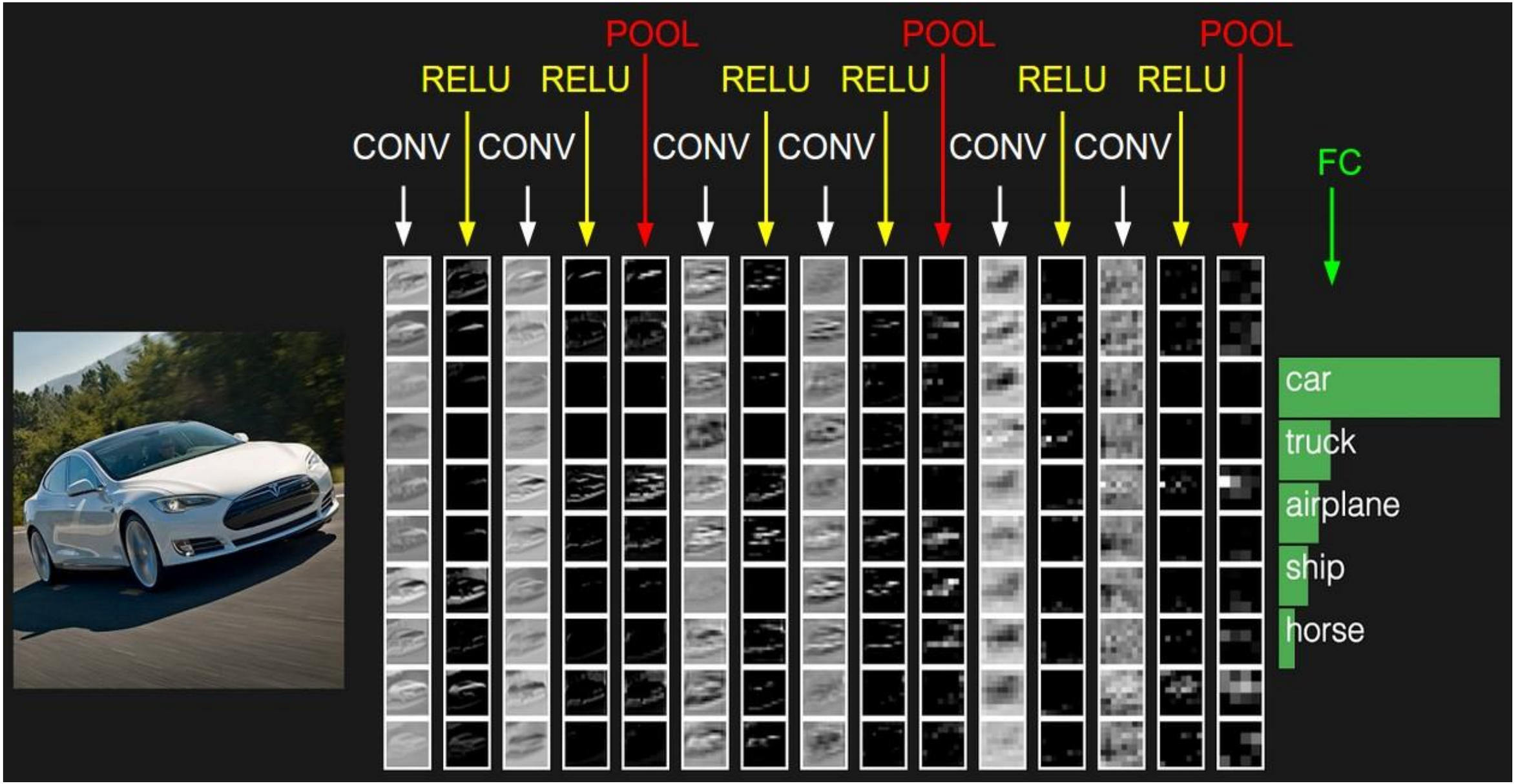
The convolutional layer



The convolutional layer



http://colah.github.io/posts/2015-08-Backprop/



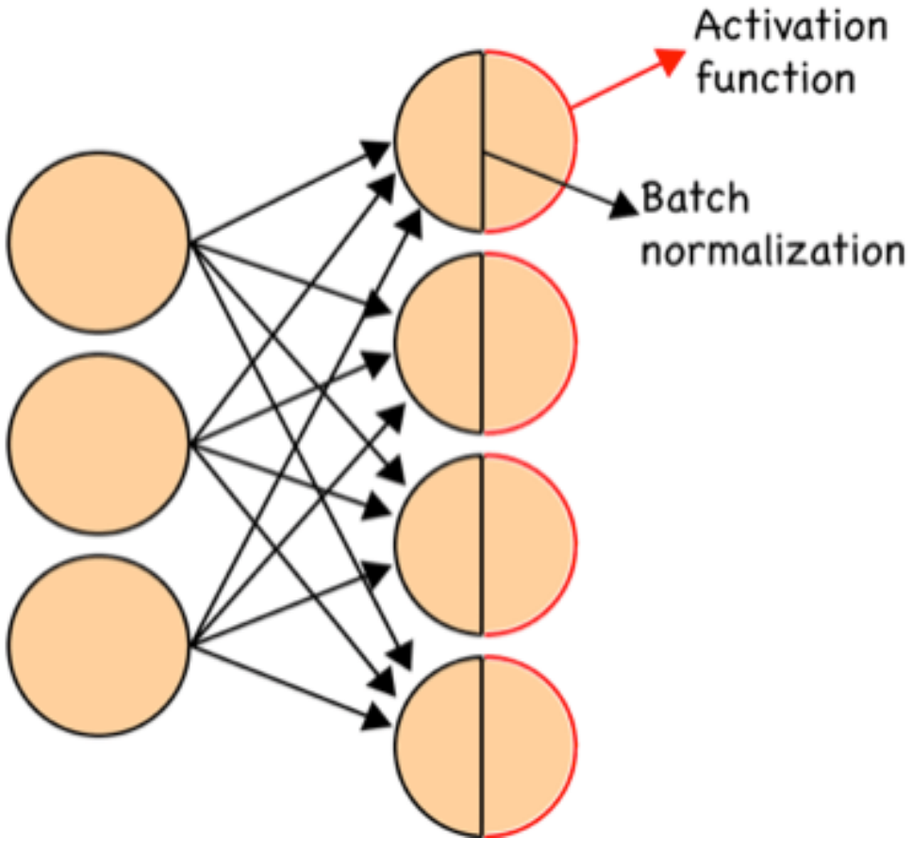
http://cs231n.github.io/convolutional-networks/

Batch normalization

> Training Deep Neural Networks is complicated by the fact that **the distribution of each layer's inputs changes during training**😰, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as internal covariate shift, and address the problem by **normalizing layer inputs**😌.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

Batch normalization



https://shuuki4.wordpress.com/2016/01/13/batch-normalization-%EC%84%A4%EB%AA%85-%EB%B0%8F-%EA%B5%AC%ED%98%84/

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

Batch normalization

In PyTorch: `torch.nn.BatchNorm2d`

```
class ConvWithBN(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super(self, ConvWithBN).__init__()

        self.conv = torch.nn.Conv2d(in_channels, out_channels)
        self.bn = torch.nn.BatchNorm2d(out_channels)

    def forward(self, input):
        output = self.conv(input)
        output = self.bn(output)

        return output
```

Residual connections

ResNet

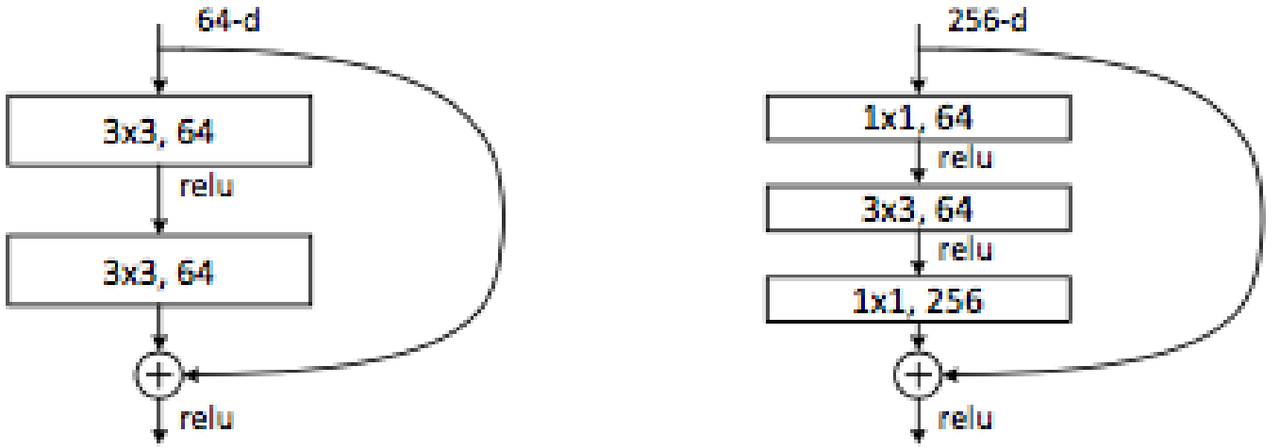
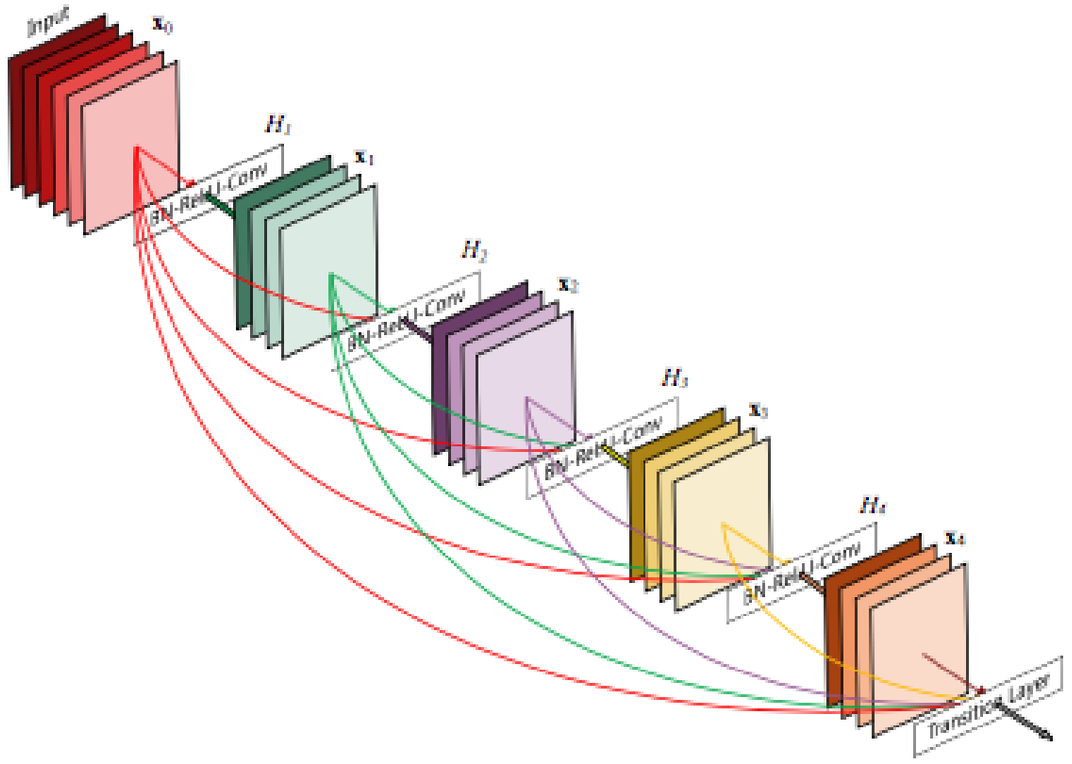


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

```
class ResidualConnection(torch.nn.Module):
    def __init__(self, channels):
        supert(self, ResidualConnection).__init__()

        self.conv1 = torch.nn.Conv2d(in_channels=channels,
                                     out_channels=channels)
        self.relu1 = torch.nn.ReLU()
        self.conv2 = torch.nn.Conv2d(in_channels=channels,
                                     out_channels=channels)
        self.relu2 = torch.nn.ReLU()

    def forward(self, input):
        output = self.conv1(input)
        output = self.relu1(output)
        output = self.conv2(output)
        output = self.relu2(output + input)

        return output
```

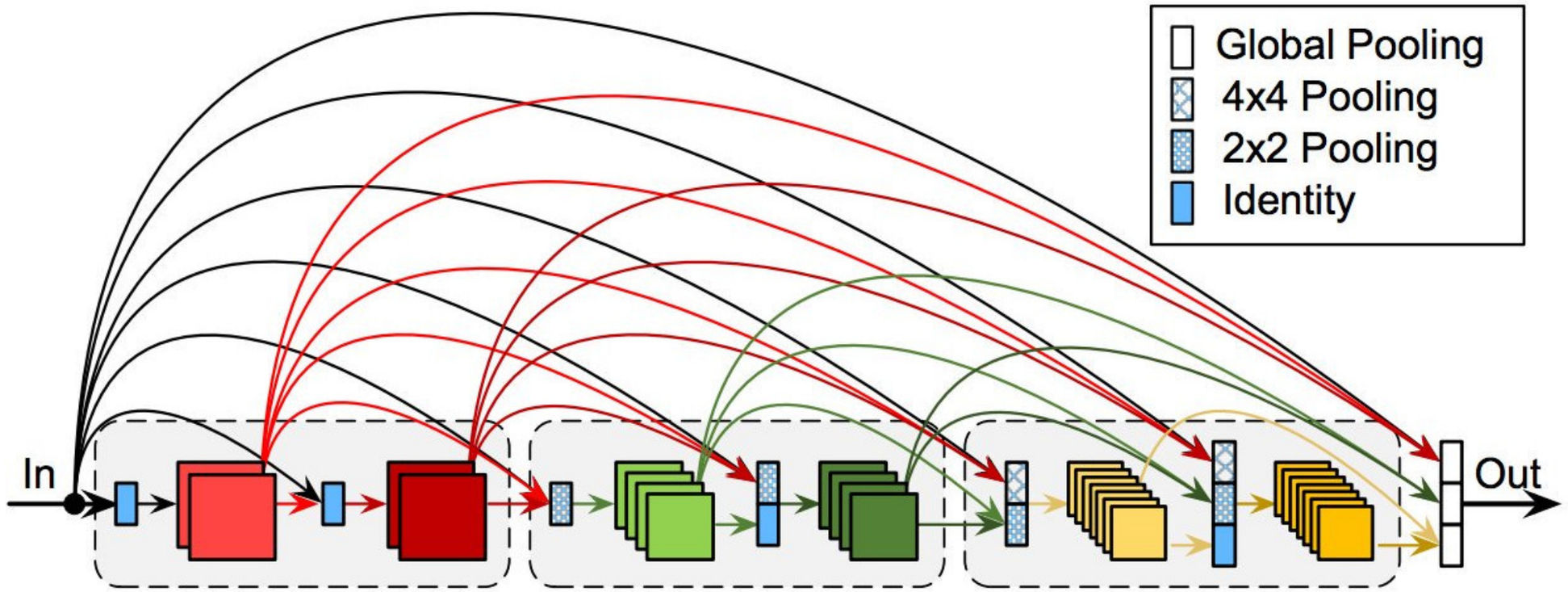
Residual connections

DenseNet



Residual connections

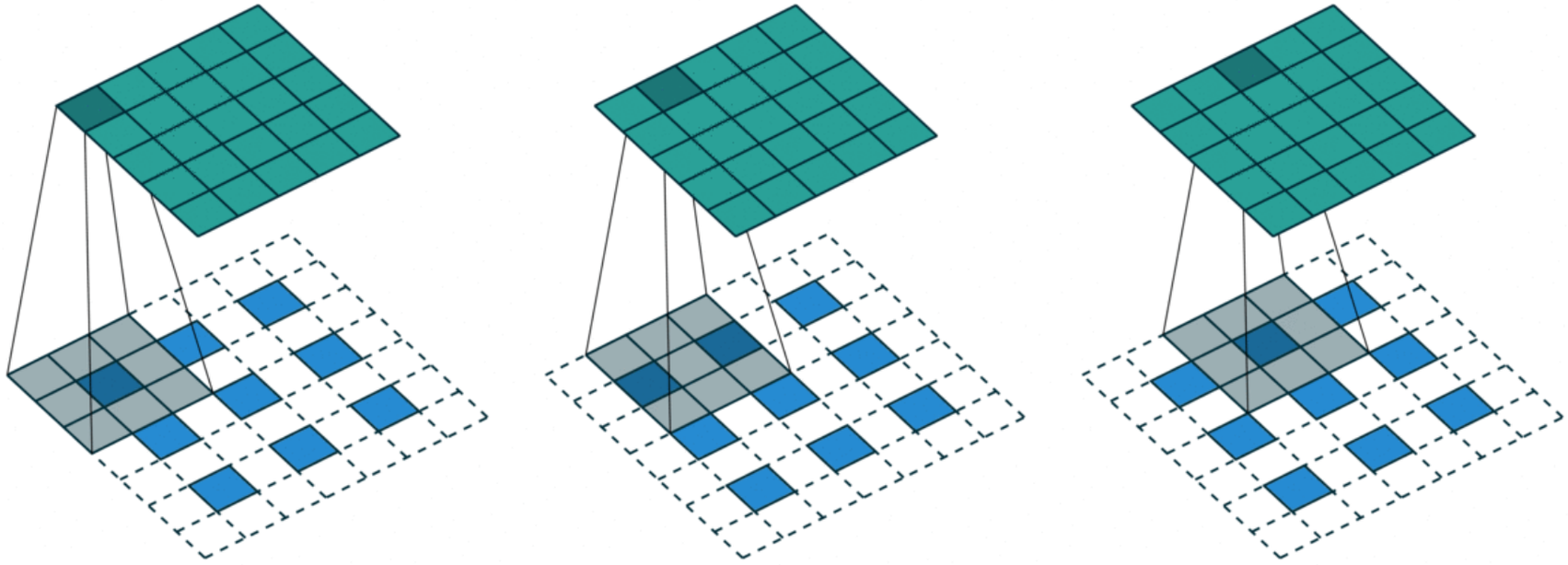
CondenseNet



Variations

- **Deconvolution**
  - Reverse of regular convolution: make small inputs larger
  - a.k.a. transposed convolution, fractionally strided convolution, upconvolution
- **Dilated convolution**
  - Same size, more context
  - a.k.a. atrous convolution
- **Separable convolution**
  - Less parameters, more effective learning
  - = depthwise convolution + pointwise convolution

Deconvolution

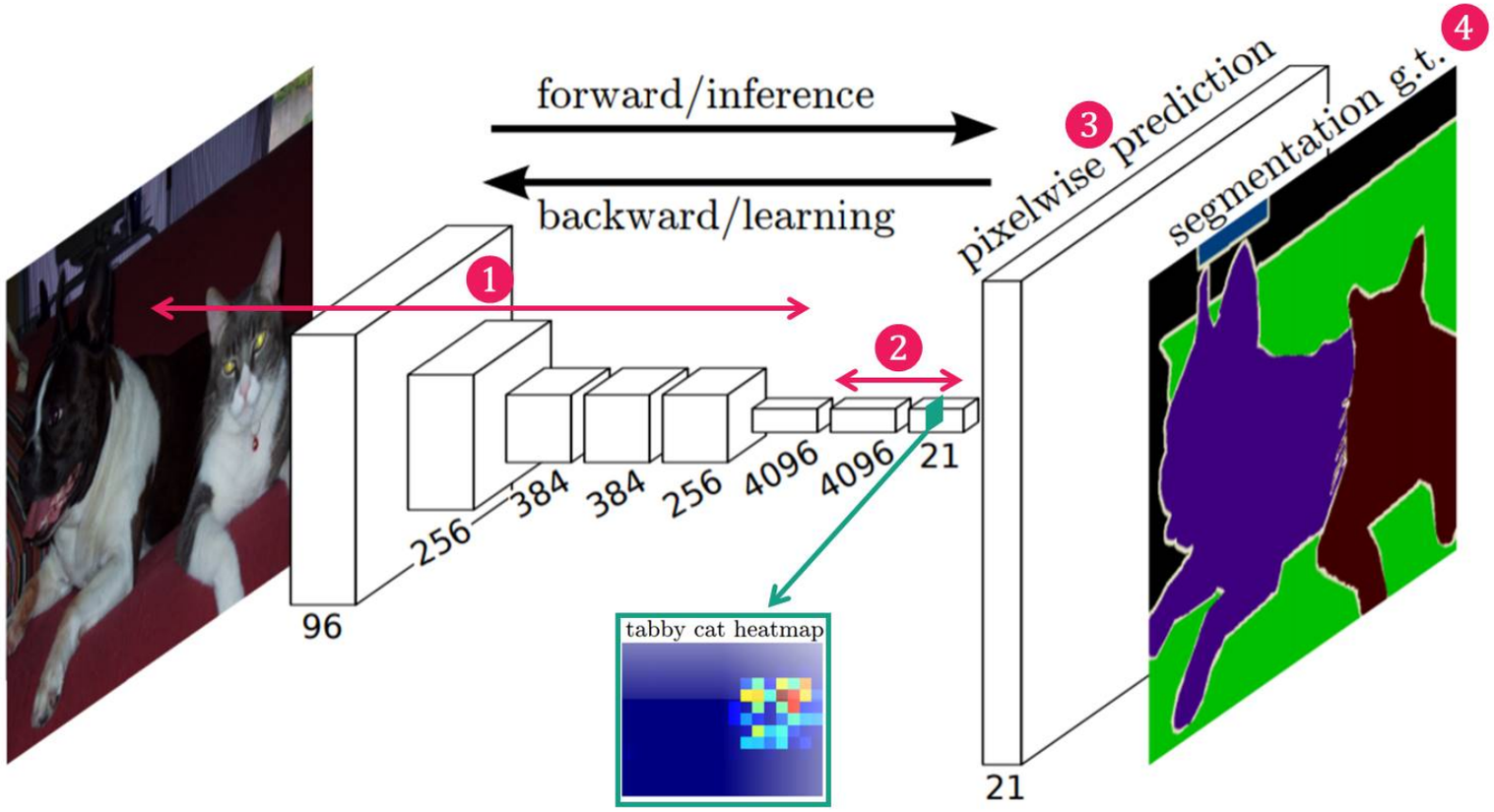


```
torch.nn.ConvTranspose2d(in_channels=1, out_channels=1, kernel=(3,
3), stride=2, padding=1, dilation=1, bias=False)
```

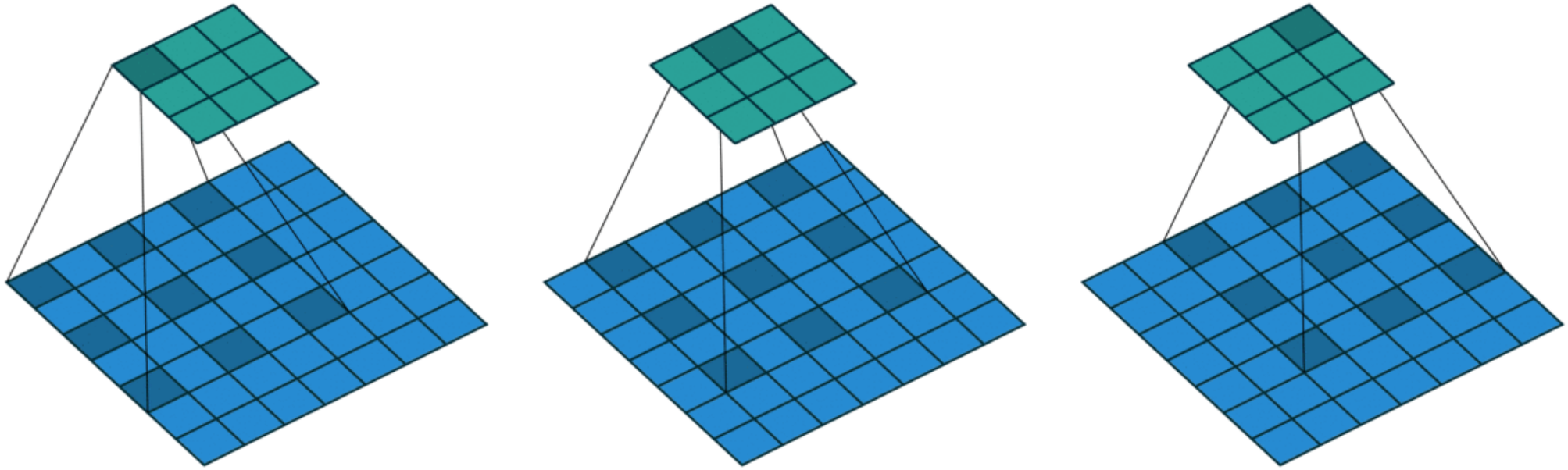

https://github.com/vdumoulin/conv_arithmetic

Deconvolution

Example: Fully Convolutional Networks



Dilated convolution

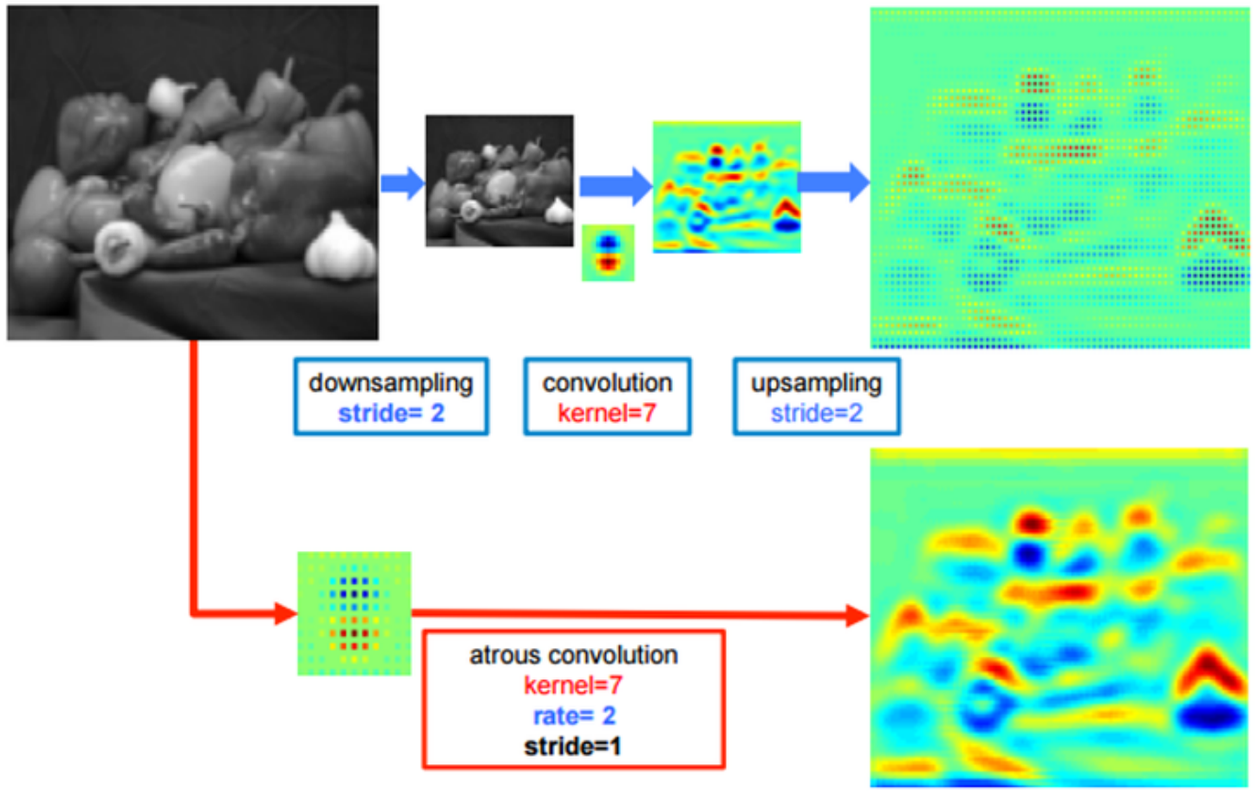


```
torch.nn.Conv2d(in_channels=1, out_channels=1, kernel=(3, 3),
stride=2, padding=1, dilation=2, bias=False)
```


https://github.com/vdumoulin/conv_arithmetic

Dilated convolution

Example: DeepLab v2



DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs (2016)

Separable convolution

- = Depthwise convolution + Pointwise convolution

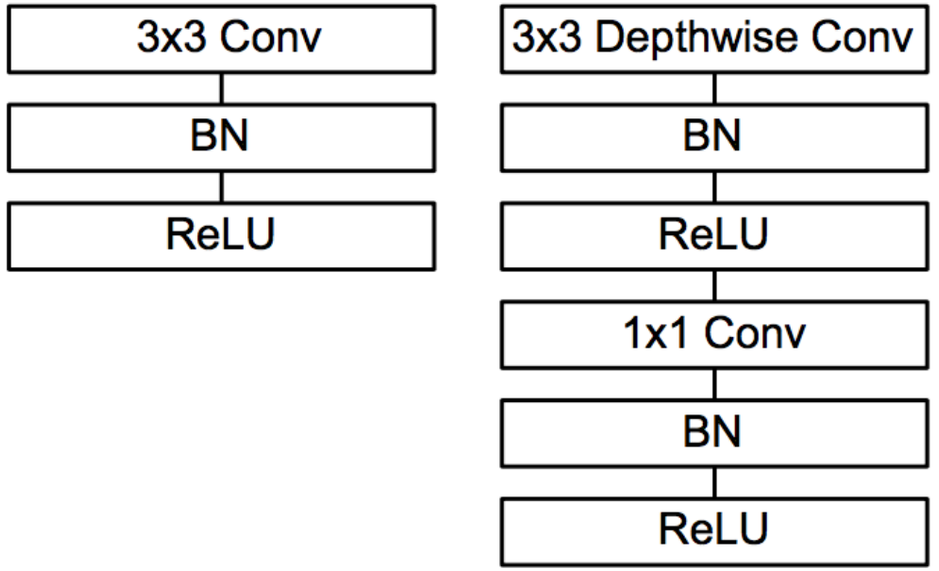
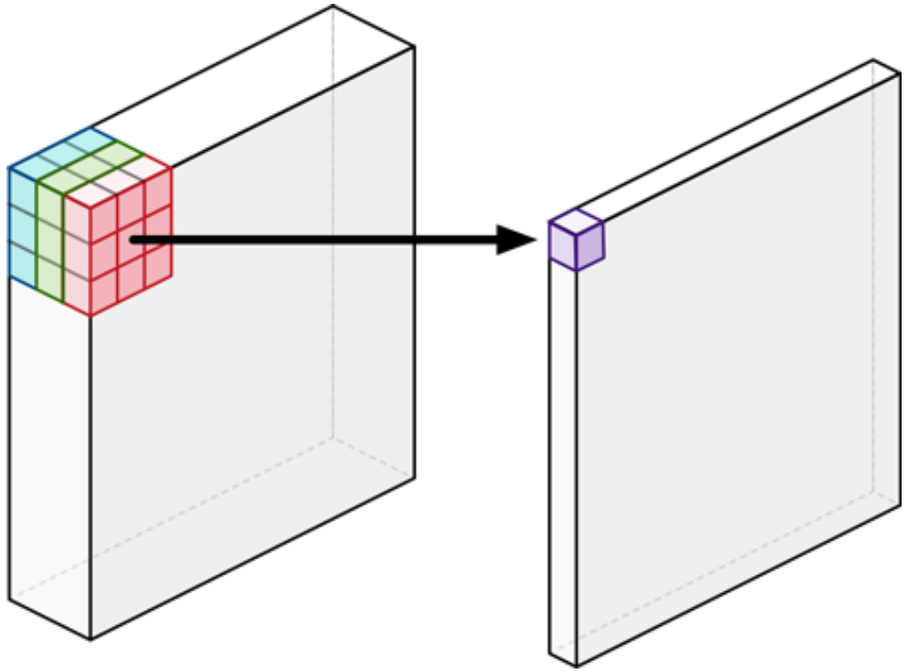


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Separable convolution

A regular convolution

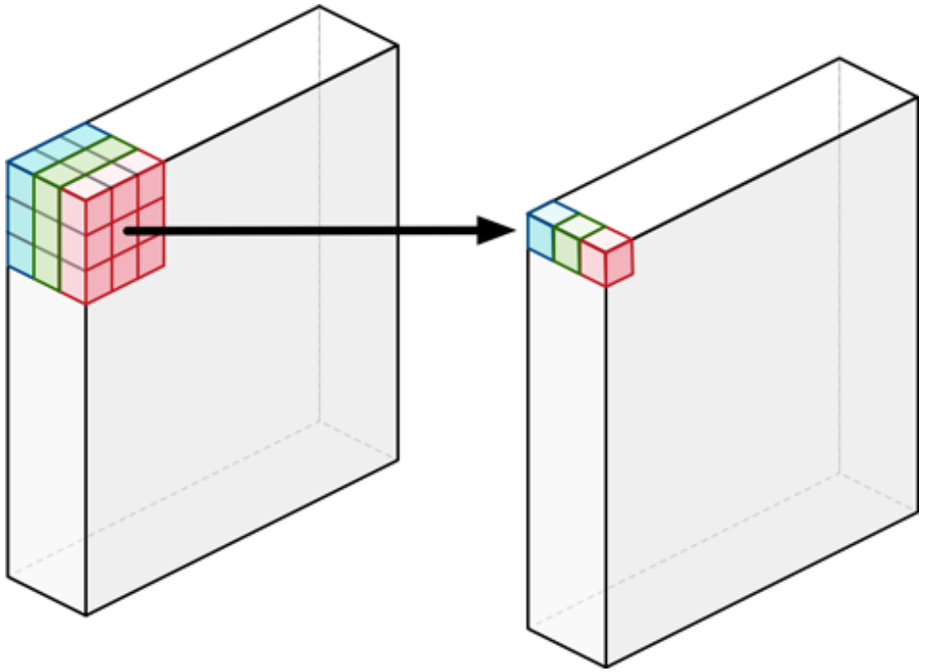


```
regular = torch.nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=3, padding=1)
```


http://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/

Separable convolution

Depthwise convolution

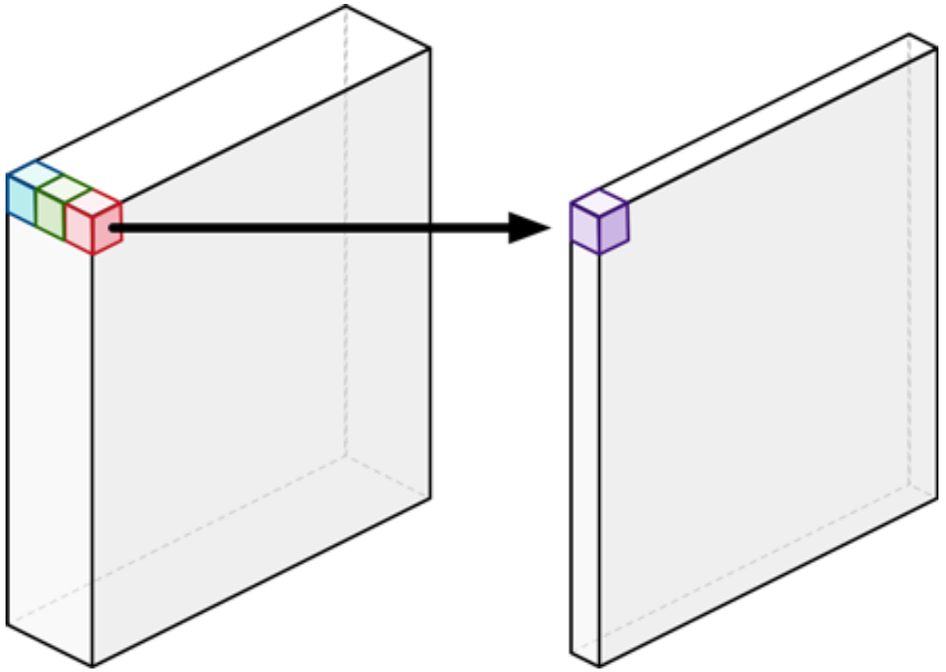


```
depthwise = torch.nn.Conv2d(in_channels=in_channels,
out_channels=in_channels, kernel_size=3, padding=1,
groups=in_channels)
```


http://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/

Separable convolution

Pointwise convolution



```
pointwise = torch.nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=1)
```


http://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/

```
class DepthwiseSeparableConv2d(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(self, DepthwiseSeparableConv2d).__init__()

        depthwise = torch.nn.Conv2d(in_channels=in_channels,
                                    out_channels=in_channels,
                                    kernel_size=3,
                                    padding=1,
                                    groups=in_channels)
        pointwise = torch.nn.Conv2d(in_channels=in_channels,
                                    out_channels=out_channels,
                                    kernel_size=1)

    def forward(self, input):
        output = self.depthwise(input)
        output = self.pointwise(output)

        return output
```

Recurrent Neural Networks

- The recurrent layer
- Gradient vanishing and exploding
- Gradient clipping
- Variations: Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU)

02

모든말에 의미를 부여하고

밥은중요
(의,식,주)

미래를 부탁
가장의 역할
믿음직해 함

연애

미래를약속,부모님의반대

사랑으로 결혼

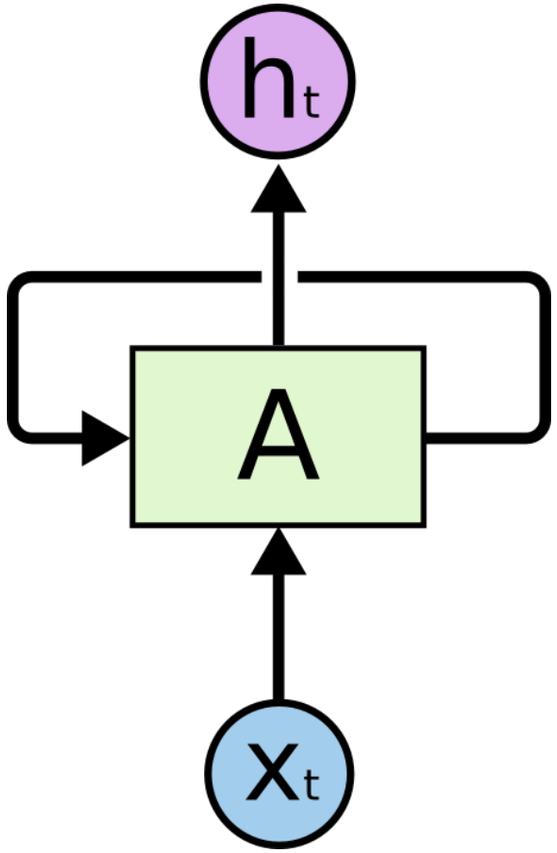
행복

밥을 달라

그 말의 의미를 고민한다

The recurrent layer

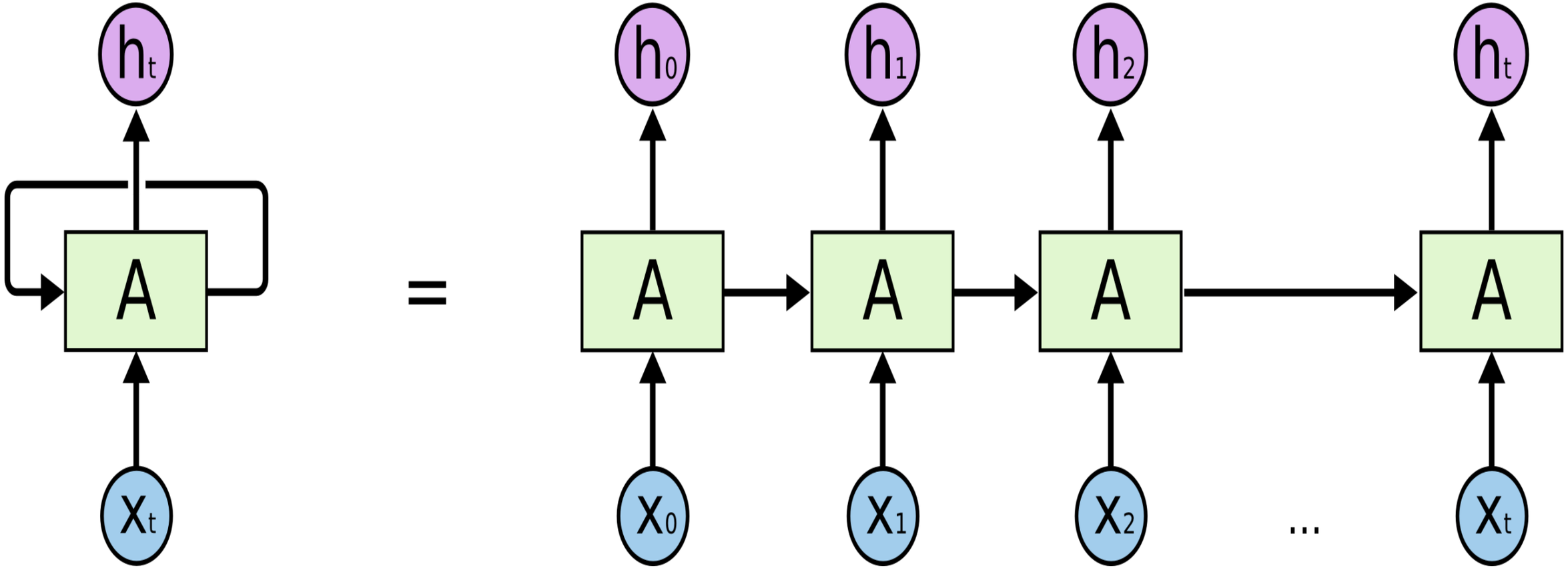
A single RNN cell:



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

The recurrent layer

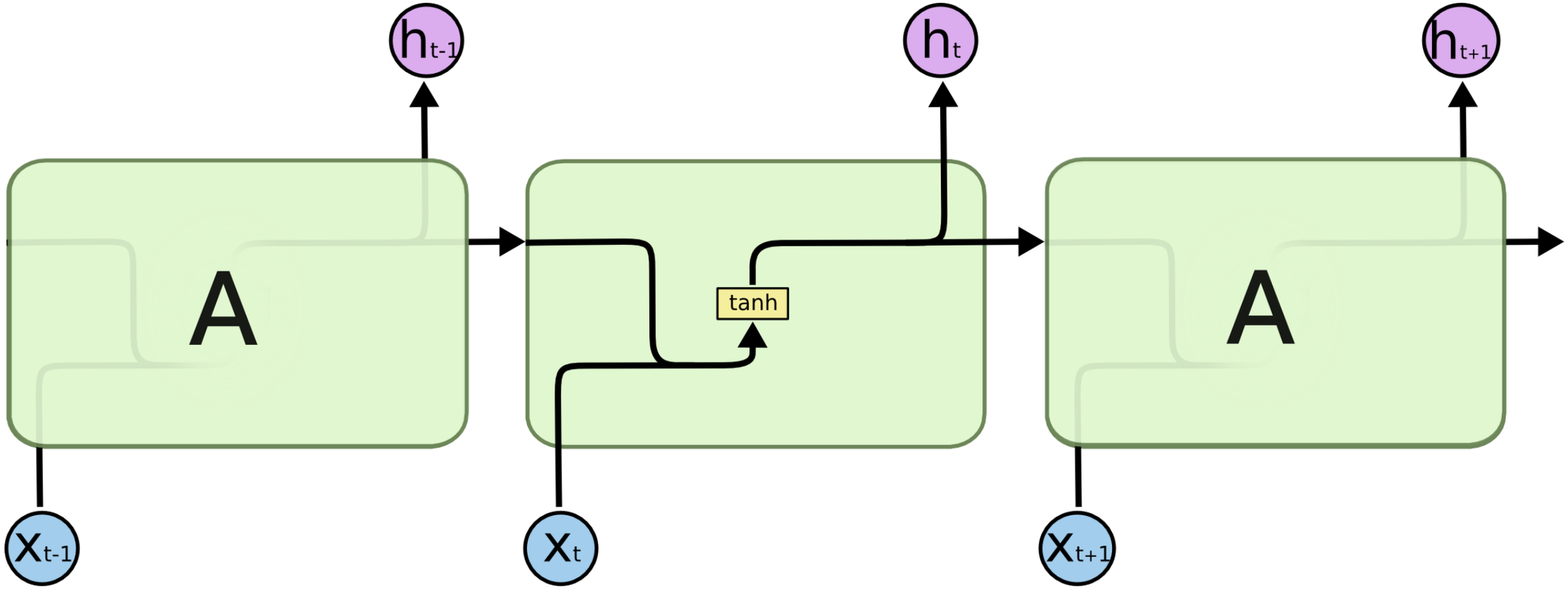
The RNN cell, unrolled:



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

The recurrent layer

A simple RNN: $h_t = tanh(w_x x_t + b_x + w_s s_{t-1} + b_t)$



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

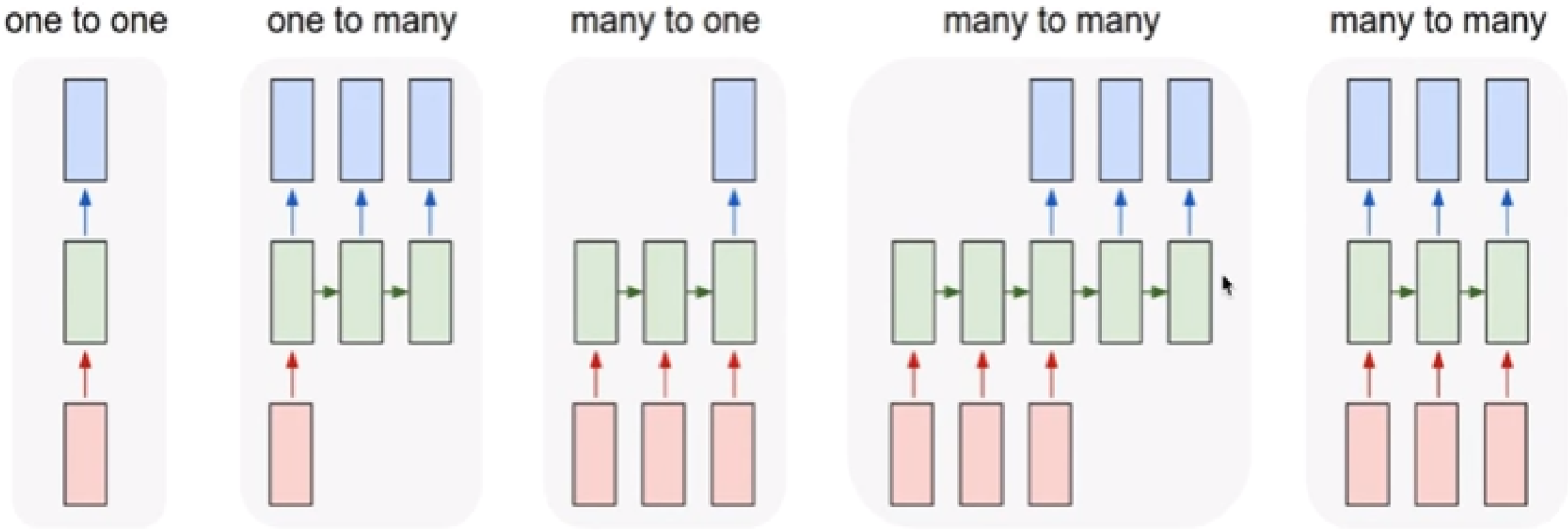
The recurrent layer

In PyTorch: `torch.nn.RNN`

```
>>> import torch
>>> import torch.nn as nn
>>> rnn = nn.RNN(input_size=10, hidden_size=20, num_layers=2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

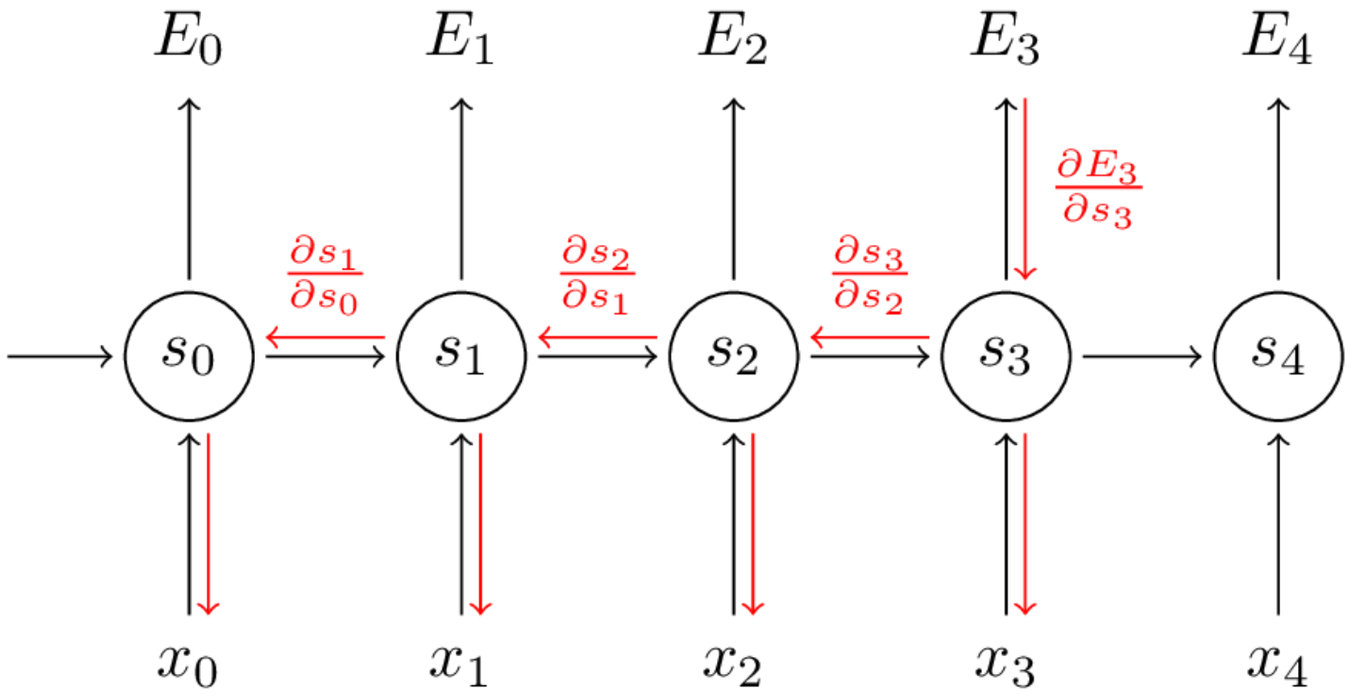
The recurrent layer

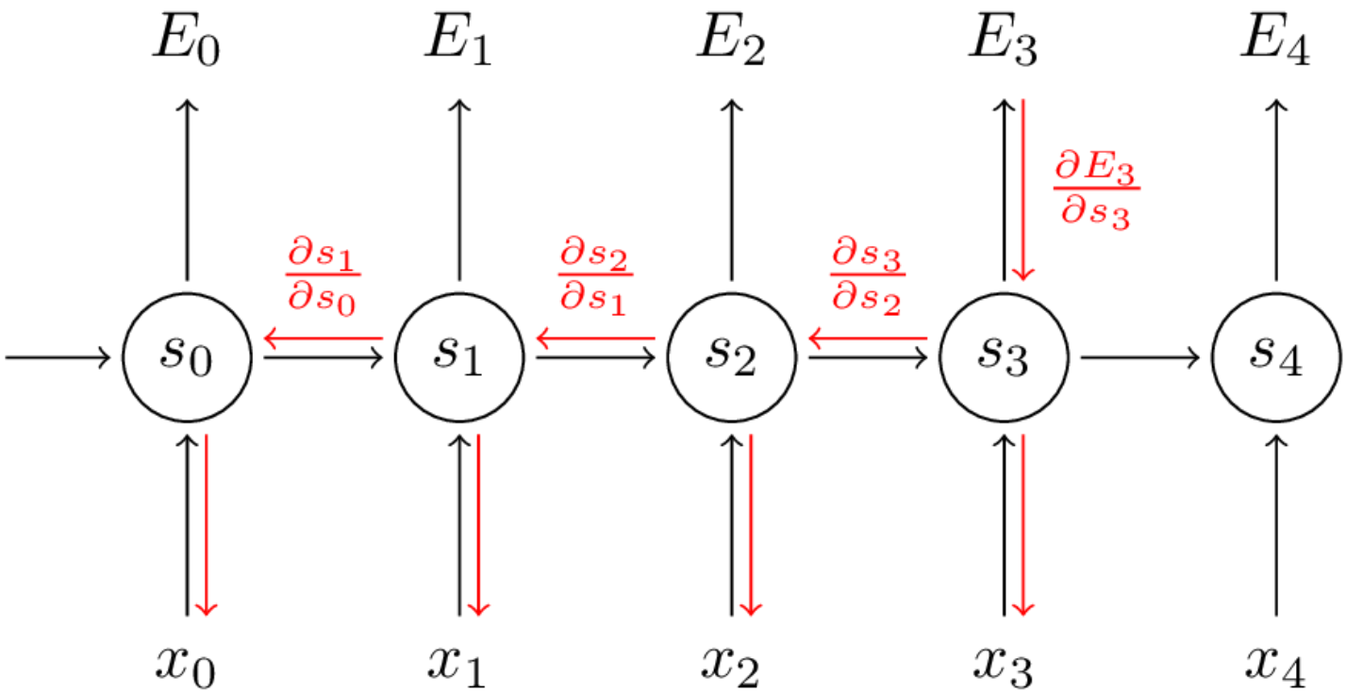
Possibilities:



The recurrent layer

Backpropagation Through Time





$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

The recurrent layer

Backpropagation Through Time

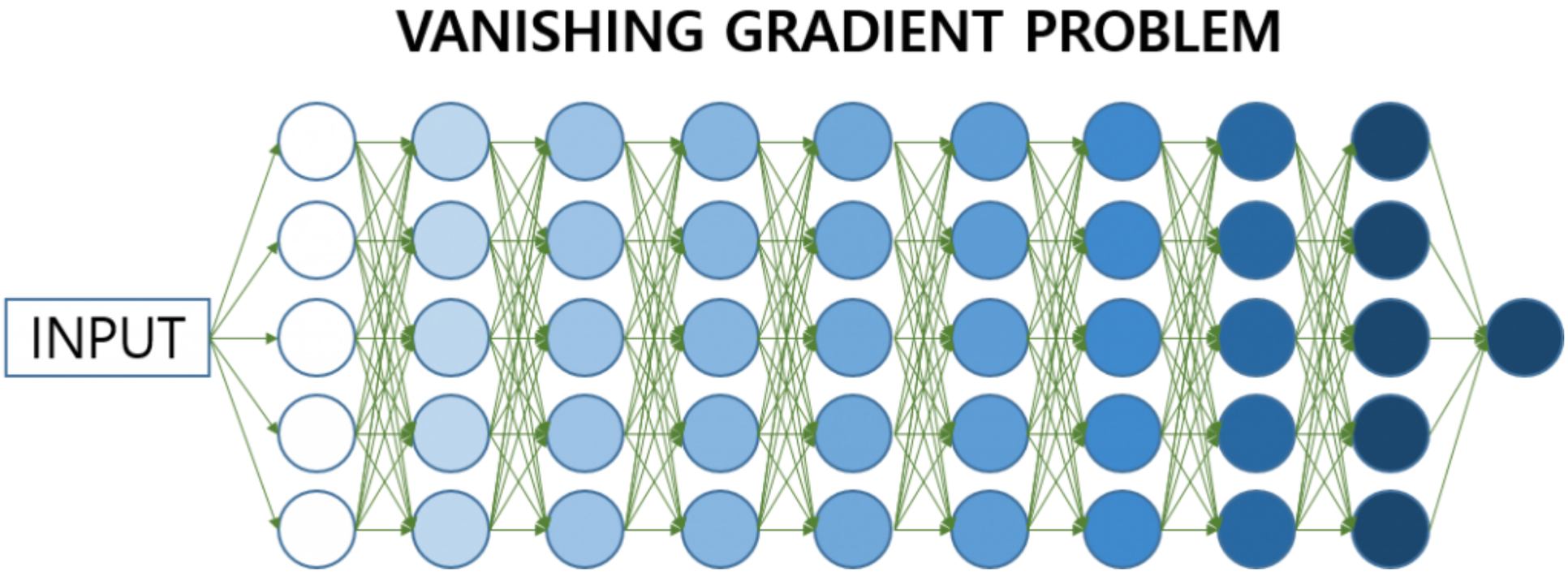
In PyTorch:

```
>>> output, hn = rnn(input, h0)
>>> loss = loss(output, target)
>>> loss.backward()
```

Gradient vanishing and exploding

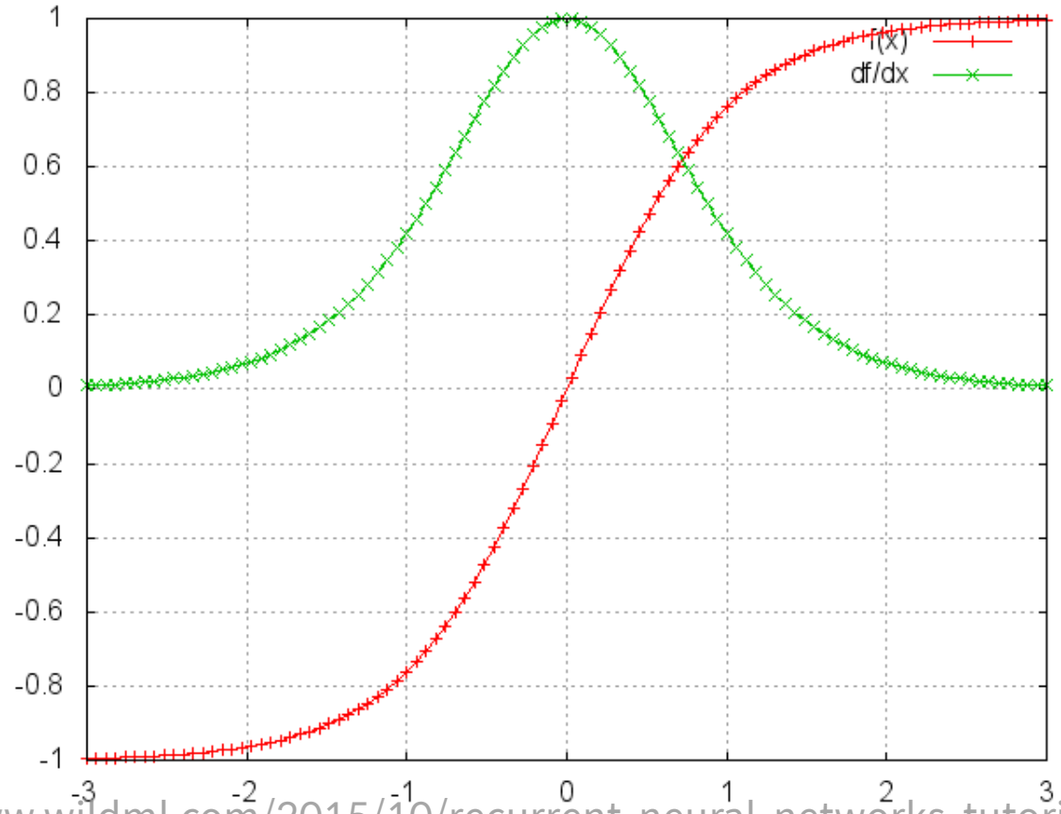
Gradient vanishing

😰 Error signals fade before reaching the beginning.



Gradient vanishing and exploding

Gradient vanishing

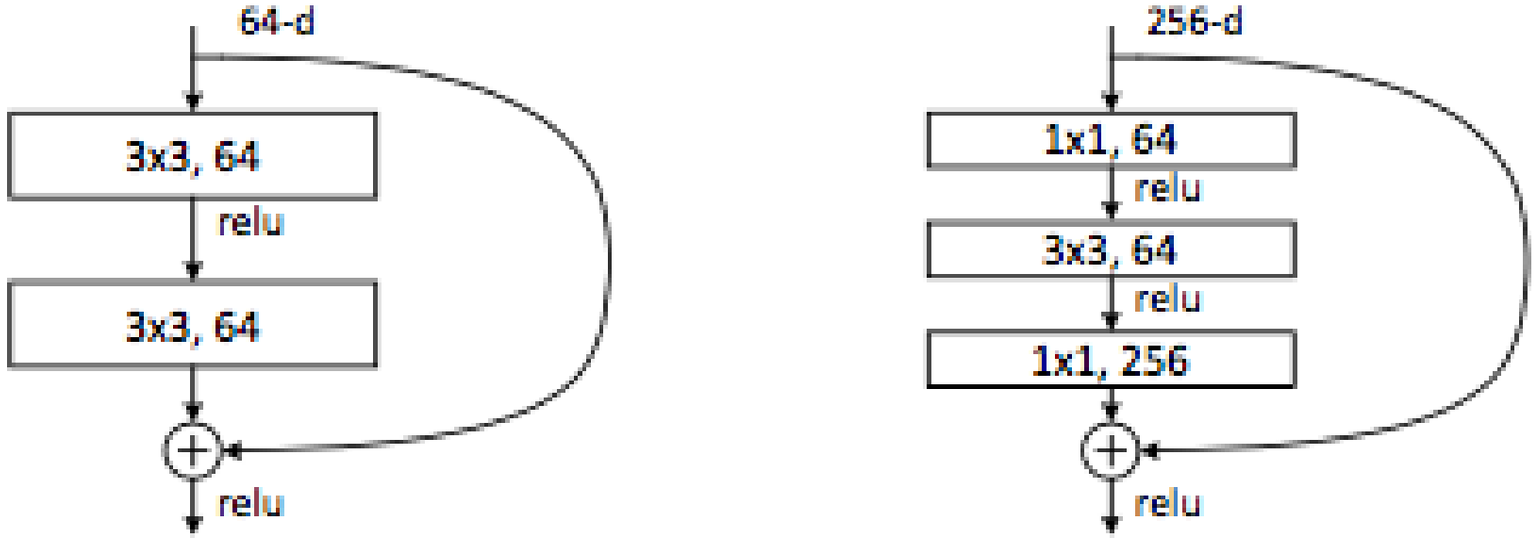


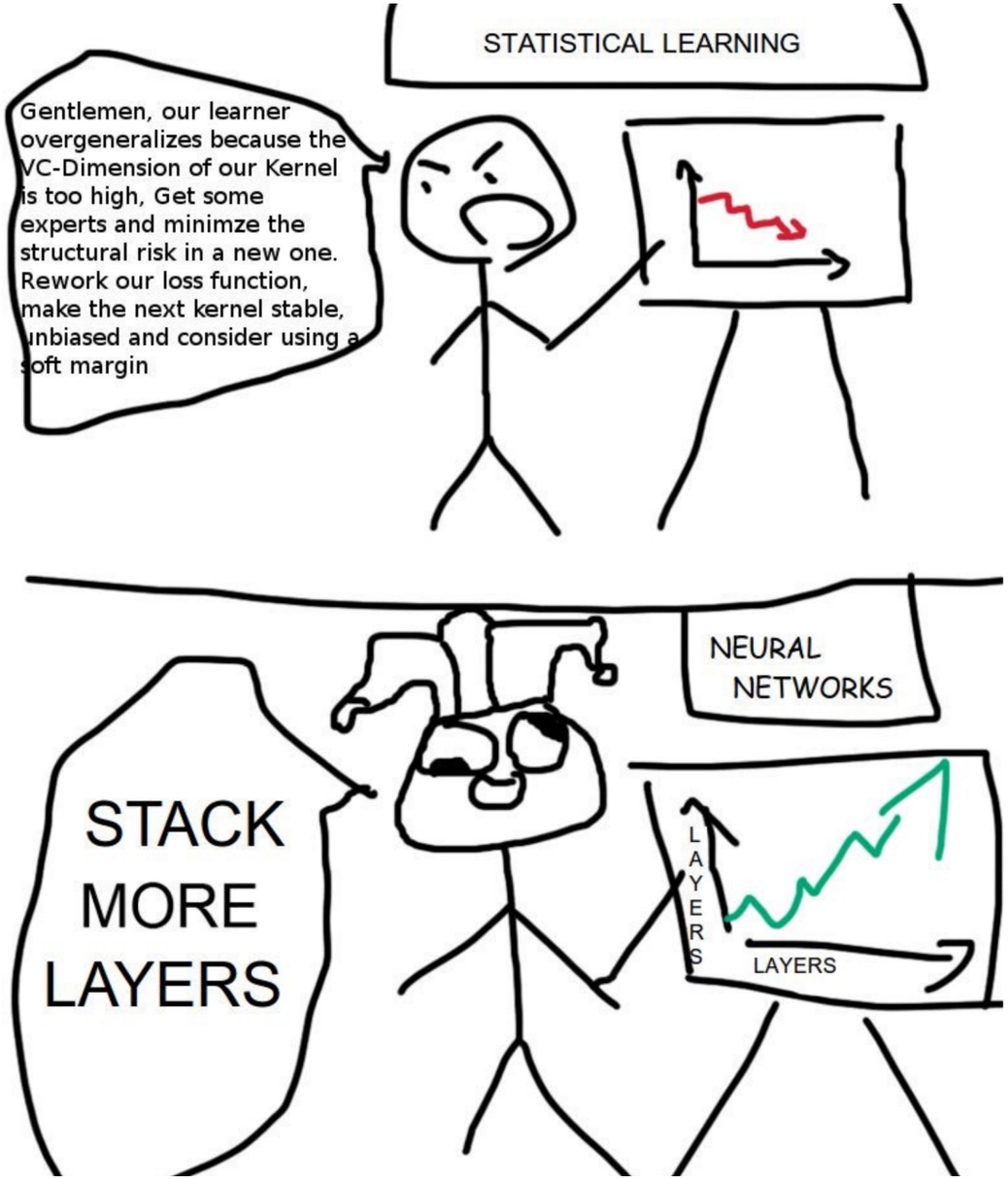
http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

Gradient vanishing and exploding

Gradient vanishing

😌 Let error signals skip layers! (e.g. ResNet, LSTM)

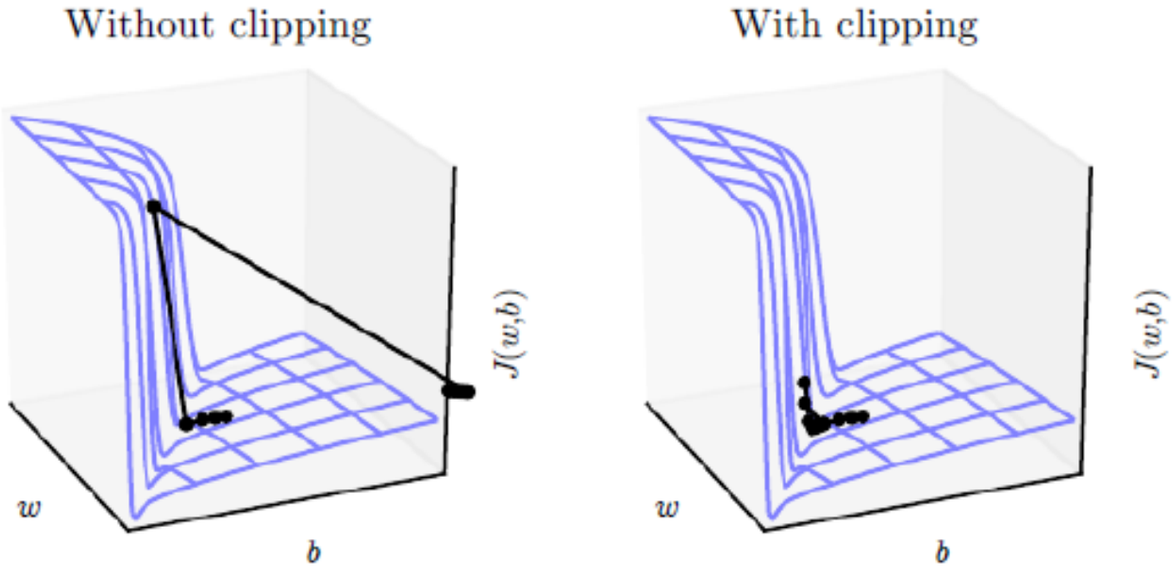



STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS

Gradient vanishing and exploding

Gradient exploding

😰 Error signals explode on "gradient cliffs".



https://www.quora.com/What-is-gradient-clipping-and-why-is-it-necessary

Gradient vanishing and exploding

Gradient exploding

😌 Set limits on gradients! (e.g. gradient clipping)

Algorithm 1 Pseudo-code for norm clipping

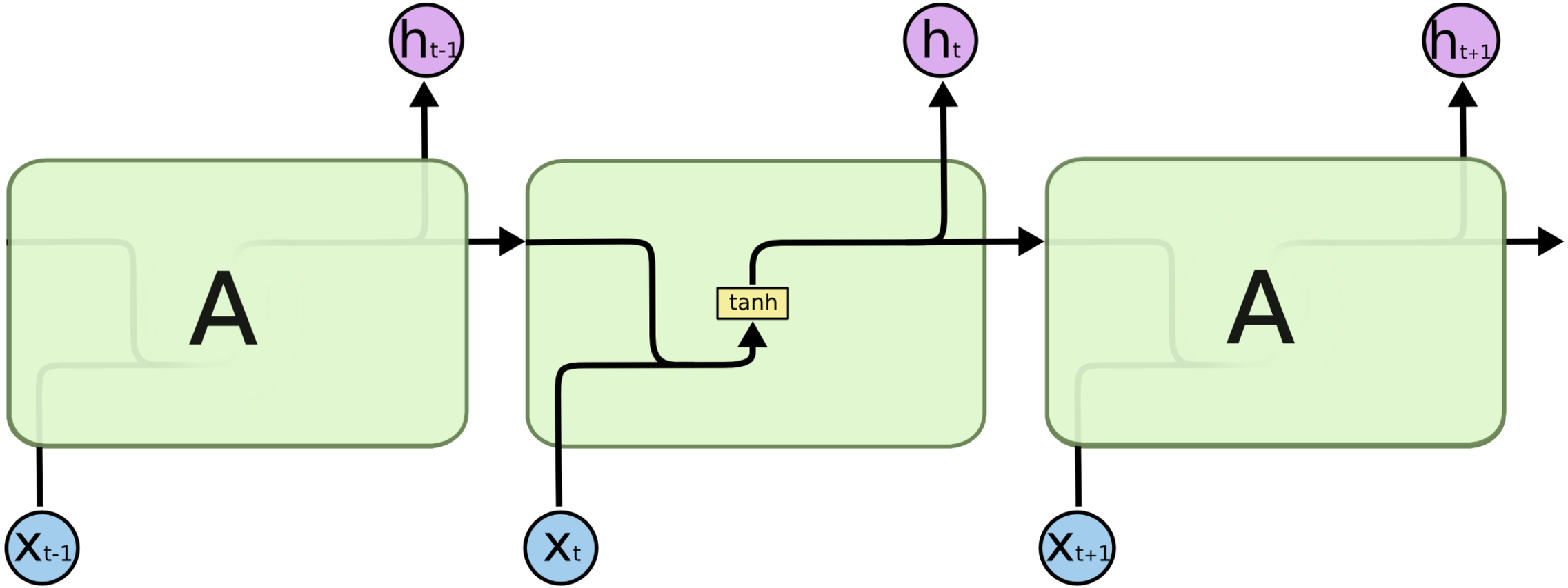
$$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$$
$$\textbf{if } \|\hat{\mathbf{g}}\| \geq threshold \textbf{ then}$$
$$\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$$
$$\textbf{end if}$$

Variations of RNN

- Long Short-Term Memory (LSTM)
  - Another recurrent path as a highway for gradients
- Gated Recurrent Unit (GRU)
  - Simplification of the LSTM

Long Short-Term Memory (LSTM)

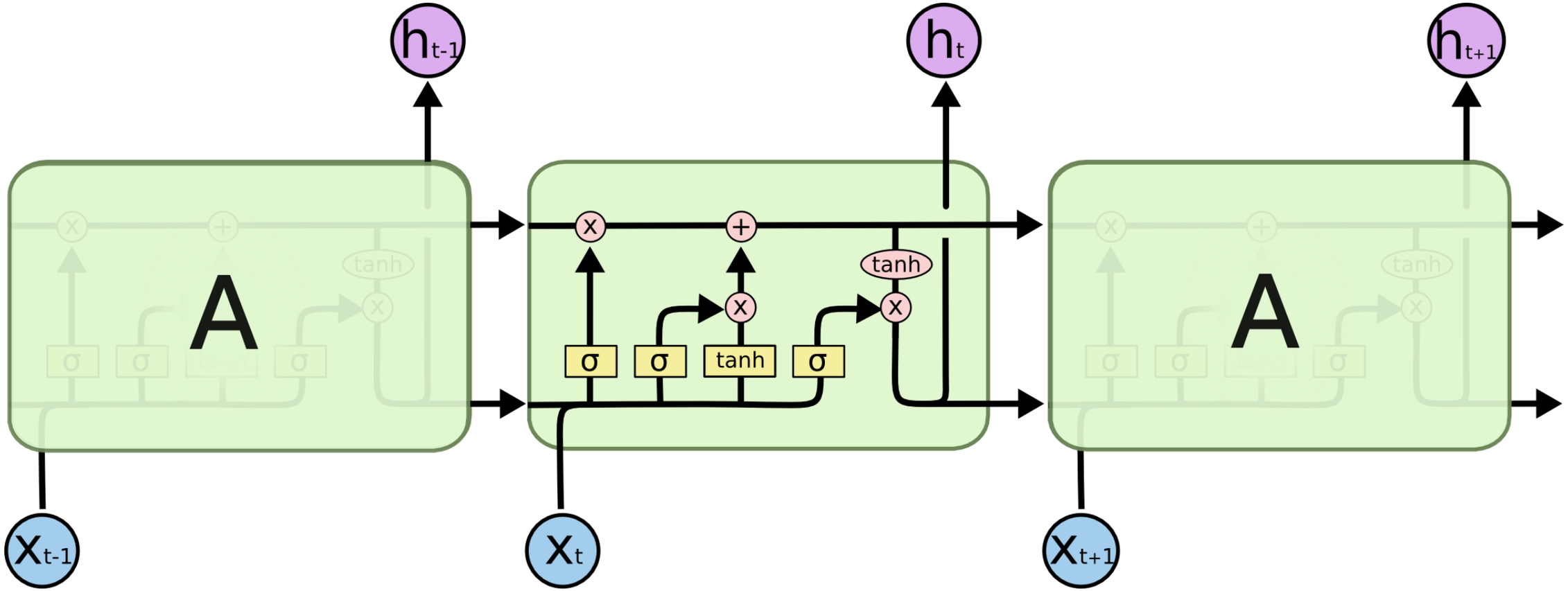
A simple RNN:



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

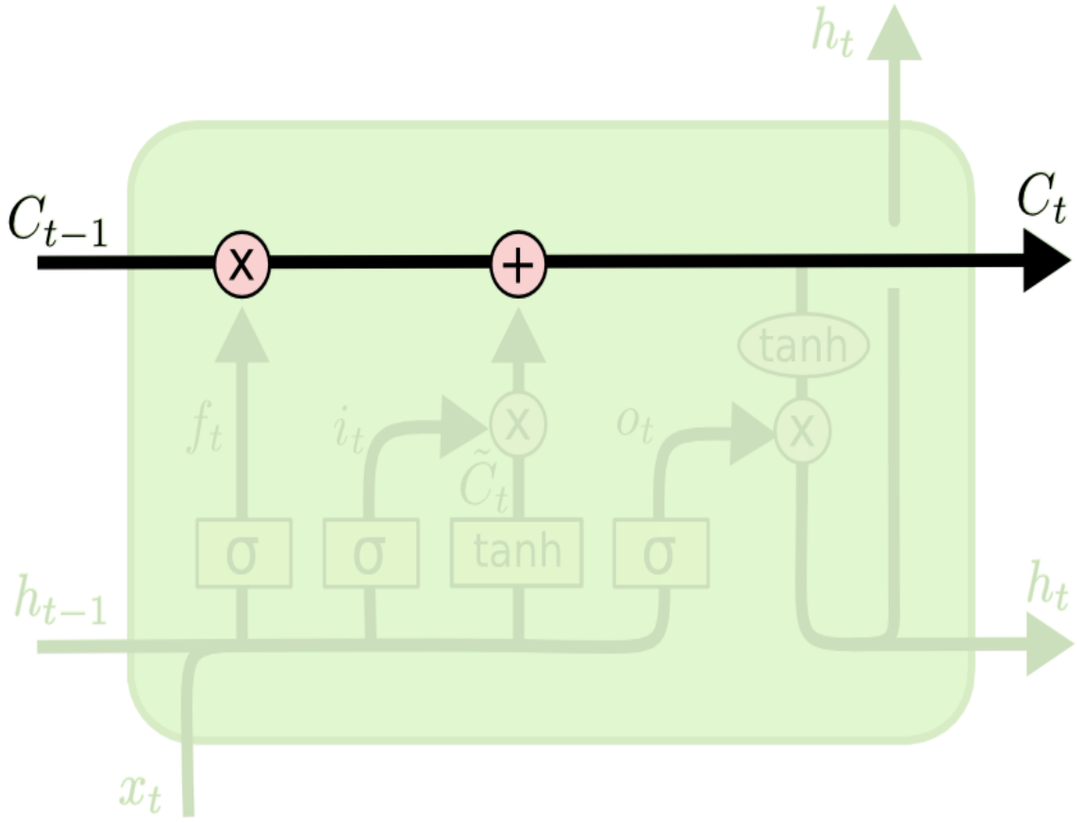
Long Short-Term Memory (LSTM)

LSTM:



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

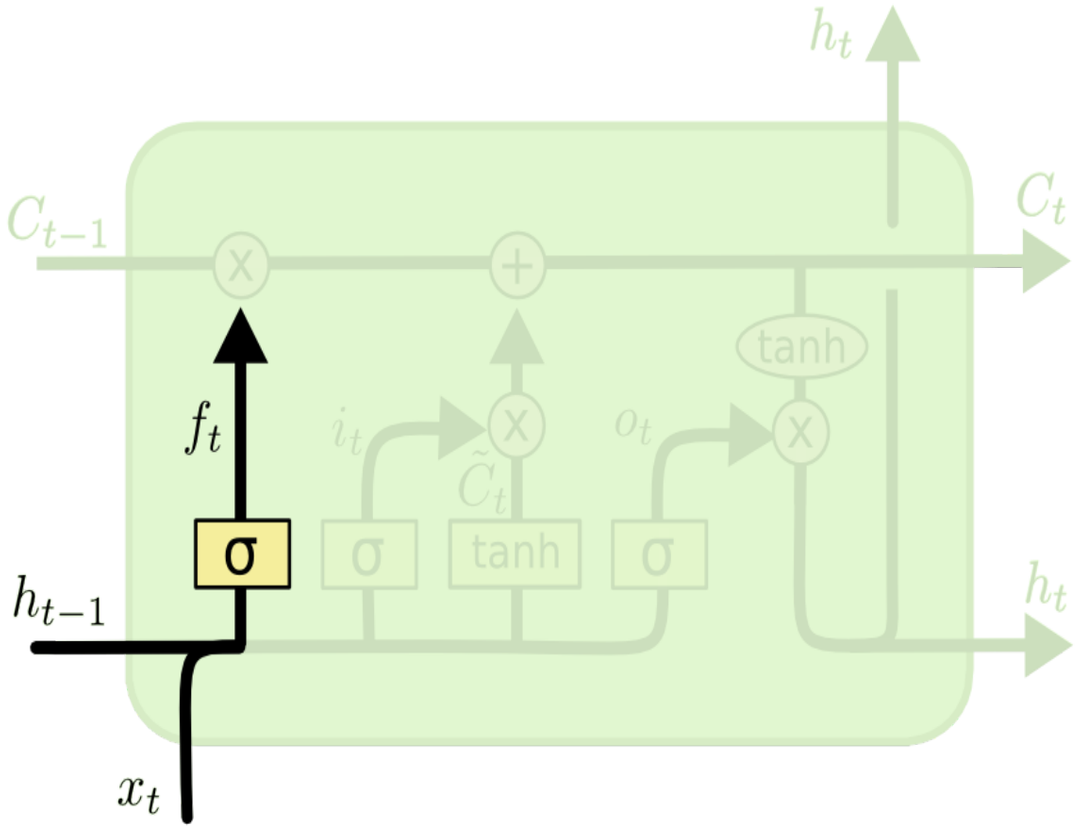
Long Short-Term Memory (LSTM)



Cell state

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

Long Short-Term Memory (LSTM)

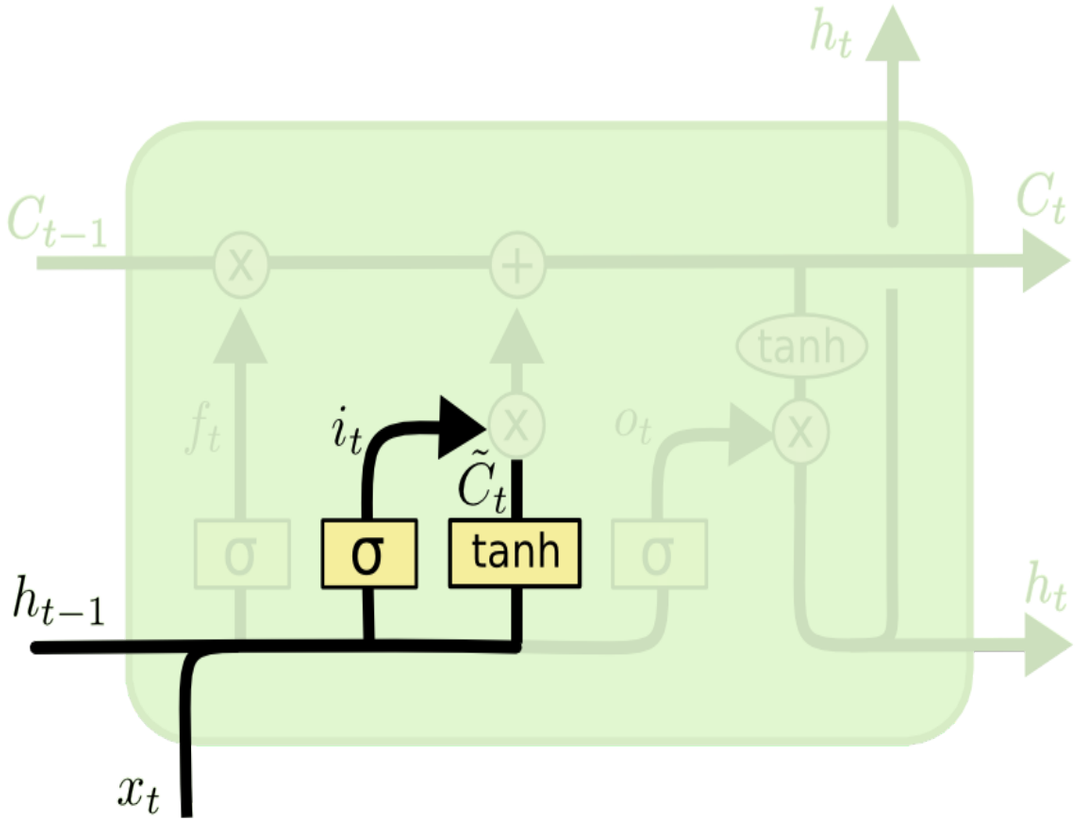


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Forget gate

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

Long Short-Term Memory (LSTM)



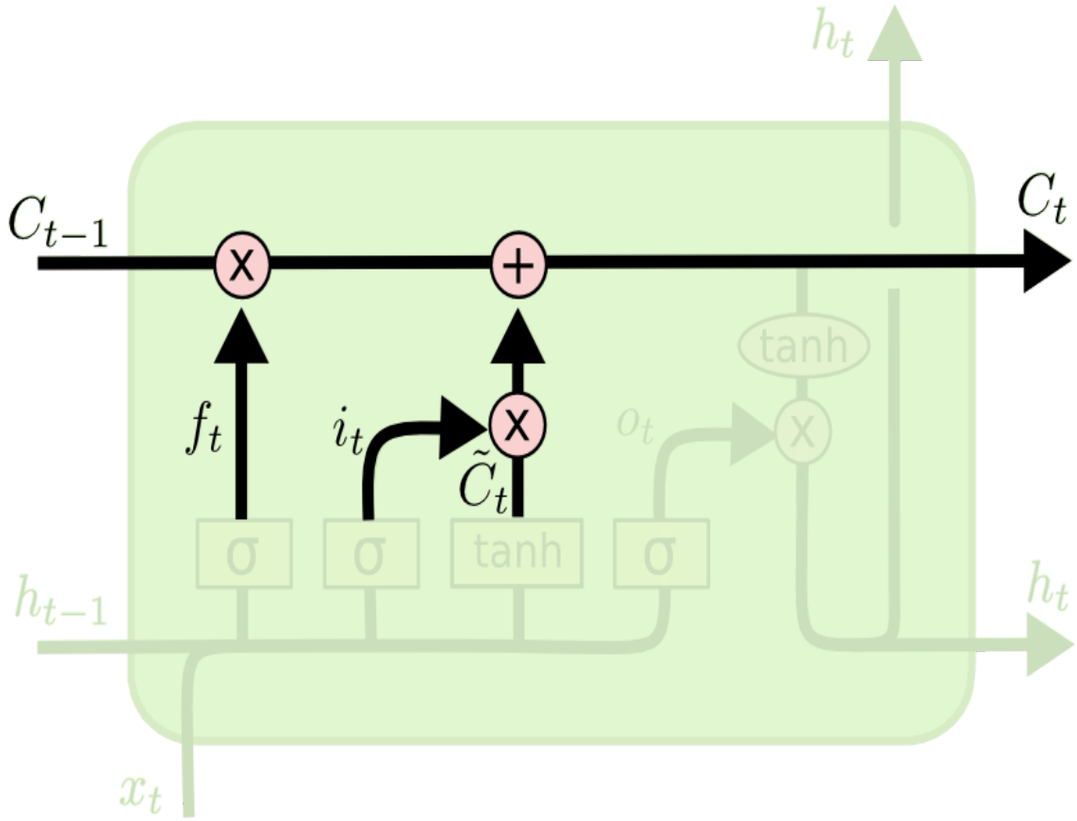
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Input gate

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

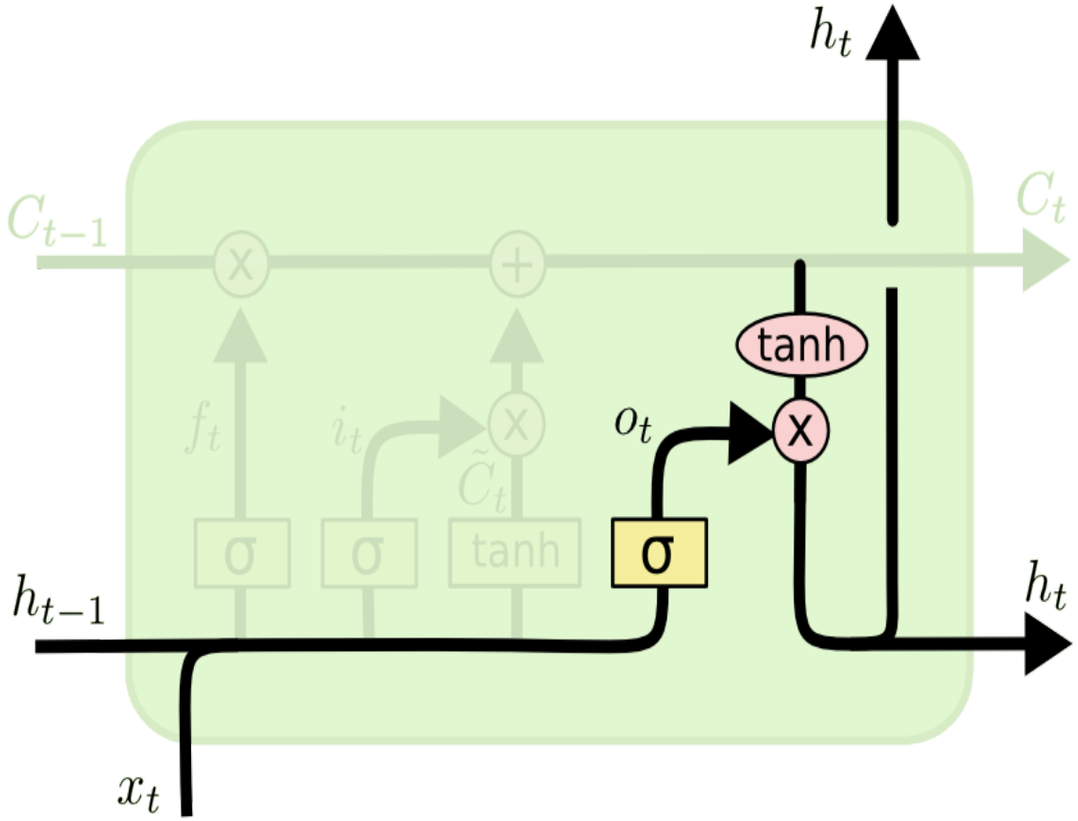
Long Short-Term Memory (LSTM)



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

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

Long Short-Term Memory (LSTM)



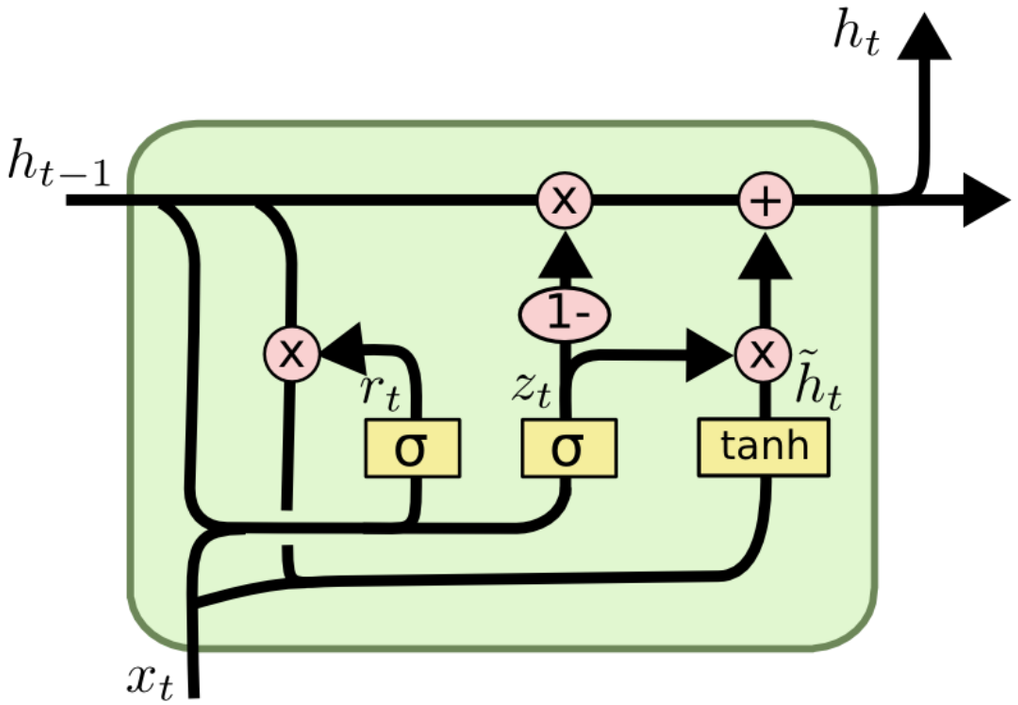
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

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

Variations of RNN

Gated Recurrent Unit (GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

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

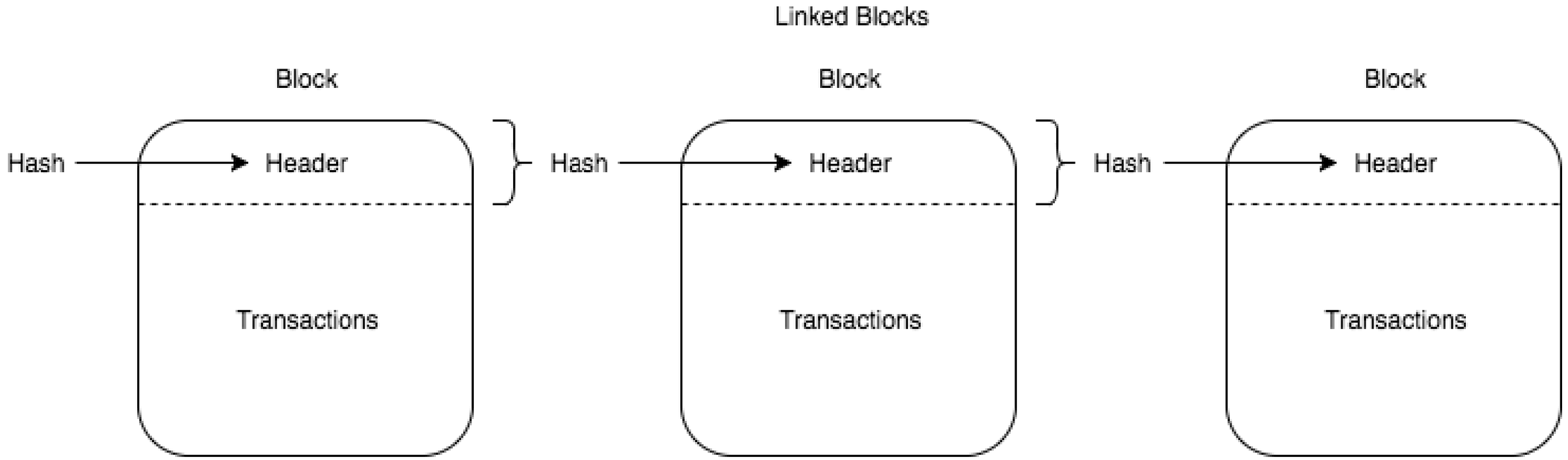
Cryptocurrency price prediction

- Cryptocurrency 101
- Obtaining and preprocessing the data
- Building our first CNN model
- Building our first RNN model
- Running live

Cryptocurrency 101

- Cryptocurrency ≠ blockchain
  - Cryptocurrency is **an application** of the blockchain
  - Blockchain is a **distributed, peer-to-peer database**
    - Chains blocks together by specially conditioned **hash** values, created by altering the **nonce** value
- Blockchain-based cryptocurrencies
  - Bitcoin, Ethereum, Bitcoin Cash, ...
  - Just like stocks, traded for real currencies at **market exchanges**
  - Just like stocks, each fulfilled orders are called **ticks**

Cryptocurrency 101



Obtaining and preprocessing the data

"There's the joke that 80 percent of data science is cleaning the data and 20 percent is complaining about cleaning the data."

- Anthony Goldbloom, Kaggle founder and CEO

Obtaining and preprocessing the data

`data.py`

1. Download tick data from coinmarketcap.com
2. Create a Pandas `DataFrame` from the data
3. Create target values `rise` (whether the price rises in the next tick)
4. Normalize price values
5. Define a custom dataset class for `torch.nn.utils.DataLoader`

Obtaining and preprocessing the data

Pandas

- Python Data Analysis Library
- Provides `DataFrame` objects for fast and efficient data manipulation
- https://pandas.pydata.org/

Obtaining and preprocessing the data

Pandas

Reading CSV as a `DataFrame`

```
>>> import pandas as pd
>>> df = pd.read_csv('tick.csv')
>>> df.head()
         time    price    amount
0  1535291170  6701.00  1.010000
1  1535291193  6700.55  0.001095
2  1535291212  6691.07  0.084010
3  1535291215  6695.81  0.440000
4  1535291267  6700.81  0.037278
```

Obtaining and preprocessing the data

Pandas

Selecting by column

```
>>> df.price.head()
0    6701.00
1    6700.55
2    6691.07
3    6695.81
4    6700.81
Name: price, dtype: float64
```

Obtaining and preprocessing the data

Pandas

Selecting by row

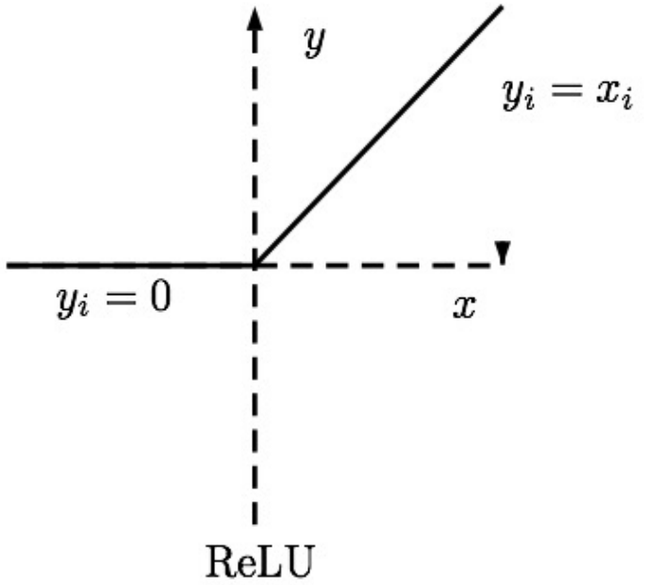
```
>>> df.ix[0]
time      1.535291e+09
price     6.701000e+03
amount    1.010000e+00
Name: 0, dtype: float64
```

Building our first CNN model

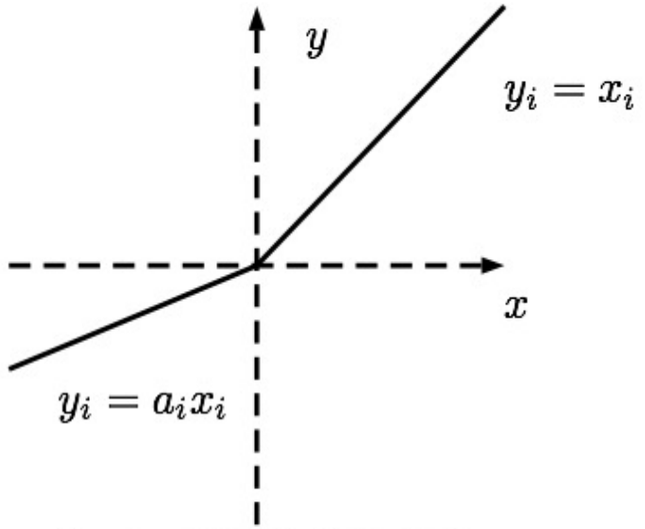
`cnn.py`

- `torch.nn.Conv1d` as the core logic
- `torch.nn.LeakyReLU` as layer activations
- `torch.nn.Linear` for the final output
- `torch.nn.BCEWithLogitLoss` as the loss function
- `torch.optim.Adam` as the optimizer
- `torch.optim.lr_scheduler,StepLR` as the learning rate strategy
- `torch.nn.Sigmoid` as the final activation

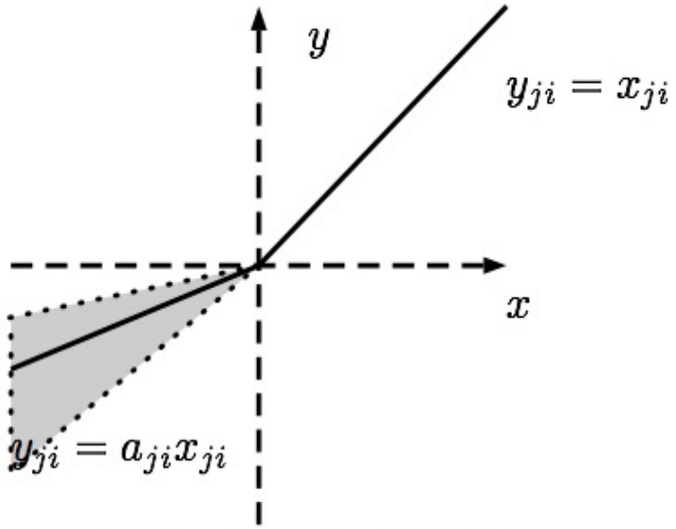
Building our first CNN model



ReLU



Leaky ReLU/PReLU



Randomized Leaky ReLU

https://www.datasciencecentral.com/m/blogpost?id=6448529%3ABlogPost%3A408853

Building our first CNN model

`rnn.py`

- `torch.nn.LSTM` as the core logic
- `torch.nn.BCEWithLogitLoss` as the loss function
- `torch.optim.Adam` as the optimizer
- `torch.optim.lr_scheduler,StepLR` as the learning rate strategy
- `torch.nn.Sigmoid` as the final activation

Train and test

`train.py`

- `--type` : The type of the model, either `rnn` or `cnn` .
- `--num_layers` : The number of layers for the model.
- `--hidden_size` : The size of the hidden state.
- `--sequence_length` : The number of previous ticks per each sample.

```
$ python run.py --type rnn ticks.csv
$ python run.py --type cnn ticks.csv
```

Thank you!