# DEEP ARCHITECTURES

**Joan Serrà**
Telefónica Research
@serrjoa

# DEEP ARCHITECTURES


**Joan Serrà**
Telefónica Research
@serrjoa



https://docs.google.com/presentation/d/1MFhet5q-SIPqc_54CXWiBvIT9OdSi6P8kpkm6IxuyEM/edit#slide=id.g522eca1928_0_11

**Slides from..**

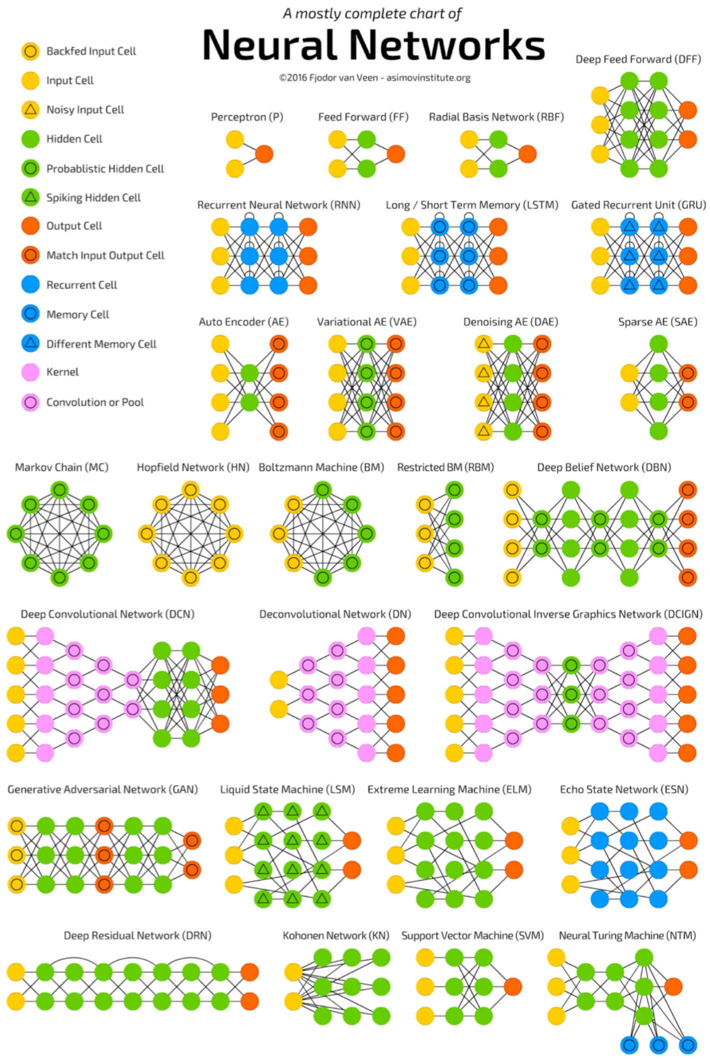

Santiago Pascual de la Puente
santi.pascual@upc.edu

PhD Candidate
Universitat Politècnica de Catalunya
Technical University of Catalonia

# Outline

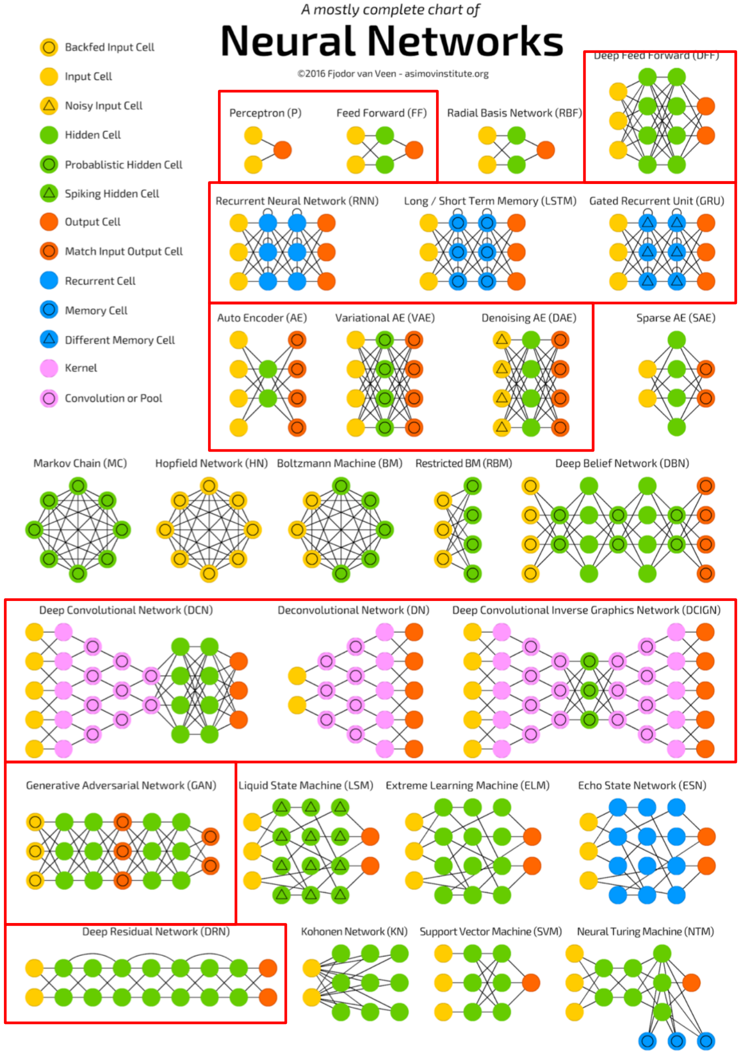
- What is in here?
- Basic Architectures
  - Fully Connected Layers
  - Recurrent Layers
  - Convolutional Layers
- Advanced Architectures
  - Hybrid CNN/RNN = QRNN
  - Auto-Encoders
  - Deep Classifiers/Deep Regressors
  - Residual Connections/Skip Connections and U-Net
  - GANs
- Conclusions

# The “Full” Story




F. Van Veen, [“The Neural Network Zoo”](#) (2016)

# The "Full" Story




F. Van Veen, "The Neural Network Zoo" (2016)

# What is in here?

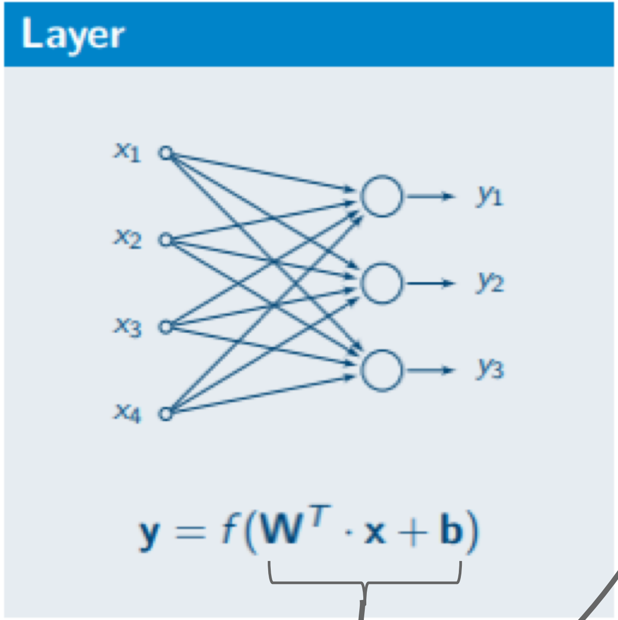
# What is in here?

- Let’s discuss about the options and nits of each topology.
- Let’s unveil the applicability of different complex structures to different tasks.
- **Let’s try to map our theory to bits of code.**

Follow-up code: https://colab.research.google.com/drive/1bDwAPU_jsmMynS3EDdq0KSEQZ3JCYoah

# Basic Architectures

# Fully Connected




CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

**Parameters:**

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: `True`

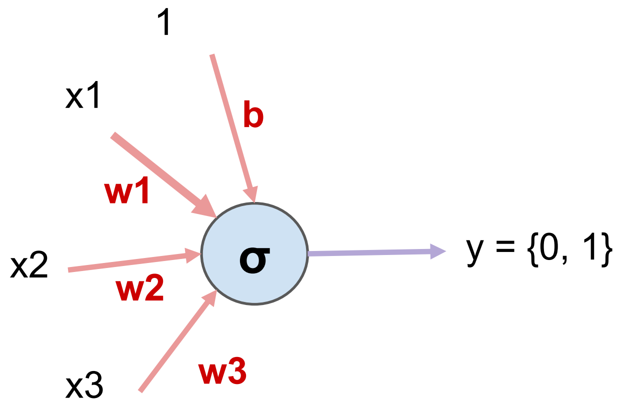
Shape:

- Input: $(N, *, \text{in\_features})$ where $*$ means any number of additional dimensions
- Output: $(N, *, \text{out\_features})$ where all but the last dimension are the same shape as the input.

**Variables:**

- **weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape $(\text{out\_features})$. If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

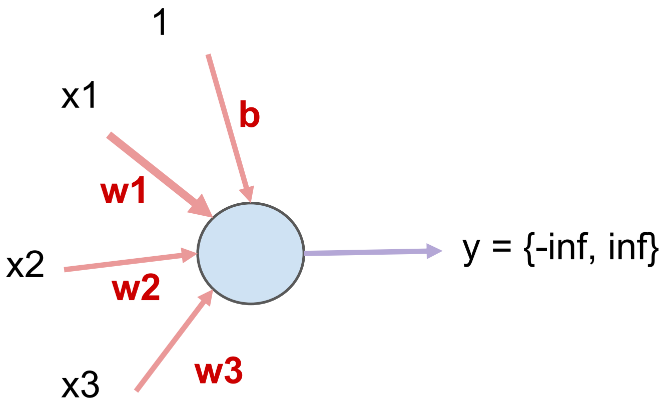
# Fully Connected: A perceptron




Fully connected layer with one unit. A sigmoid activation makes it a **logistic regression** (binary linear classifier):

```
lor = nn.Sequential(
      nn.Linear(NUM_INPUTS, 1),
      nn.Sigmoid()
)
```

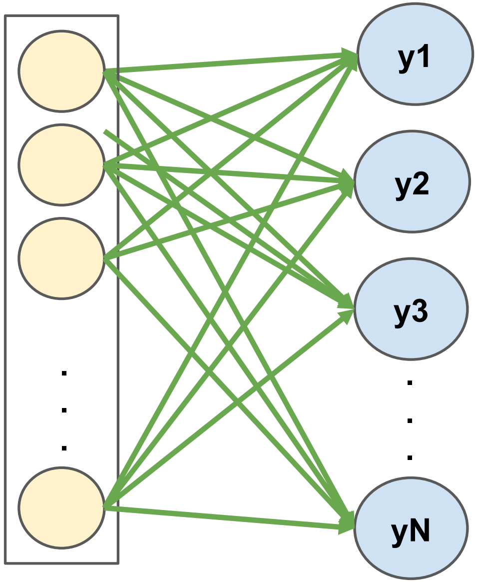
# Fully Connected: A perceptron




Fully connected layer with one unit. No activation makes it a **linear regression**:

```
lir = nn.Sequential(
    nn.Linear(NUM_INPUTS, 1)
)
```

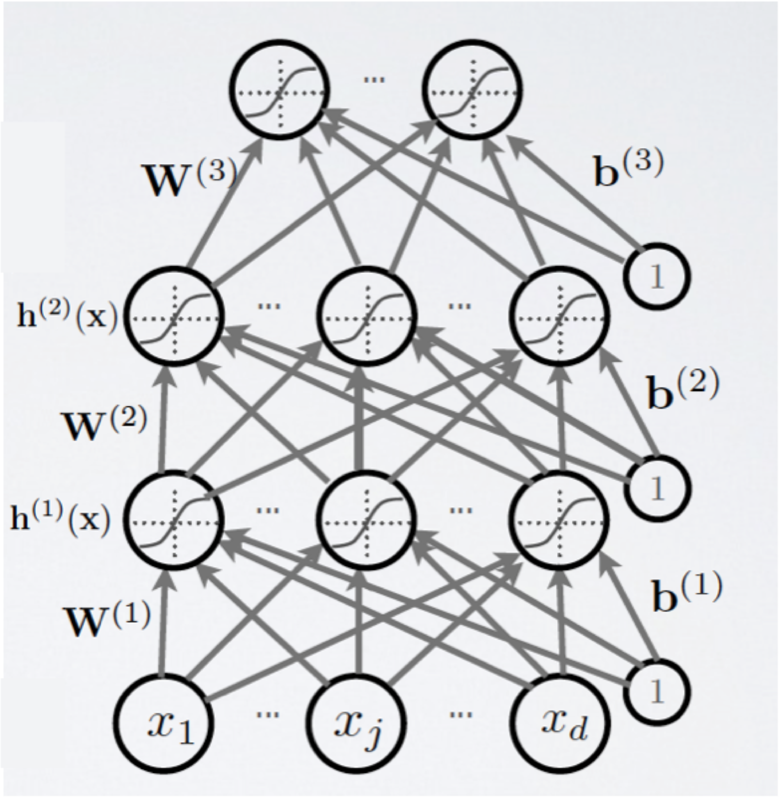
# Fully Connected: Multiclass classifier




Fully connected layer with many units. Softmax activation makes it a “softmax classifier”.

```
smx = nn.Sequential(
    nn.Linear(NUM_INPUTS, NUM_OUTPUTS),
    nn.LogSoftmax(dim=1)
)
```

# Fully Connected: MultiLayer Perceptron



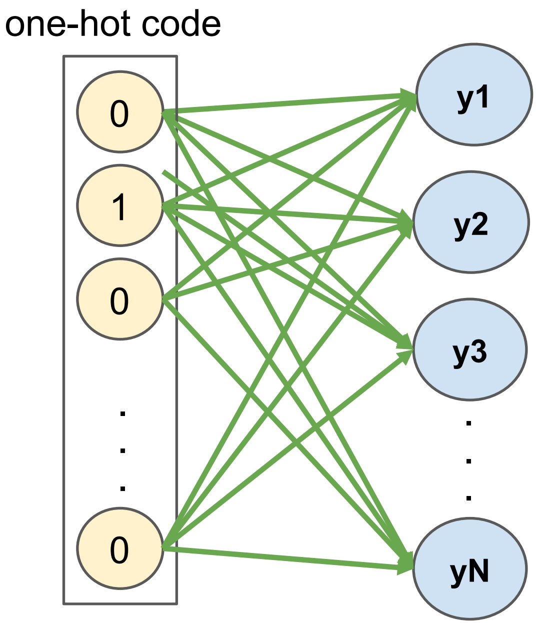

*Slide Credit: Hugo Laroche NN course*

**This is also a deep neural network of course.**

Many fully connected layers with many units.

```
NUM_INPUTS=100
HIDDEN_SIZE=1024
NUM_OUTPUTS=20

mlp = nn.Sequential(
    nn.Linear(NUM_INPUTS, HIDDEN_SIZE),
    nn.Tanh(),
    nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE),
    nn.Tanh(),
    nn.Linear(HIDDEN_SIZE, NUM_OUTPUTS),
    nn.LogSoftmax(dim=1)
)
```

# Fully Connected: When input is discrete...




We usually take one-hot codes as discrete tokens. Can we use a Linear layer to process it?

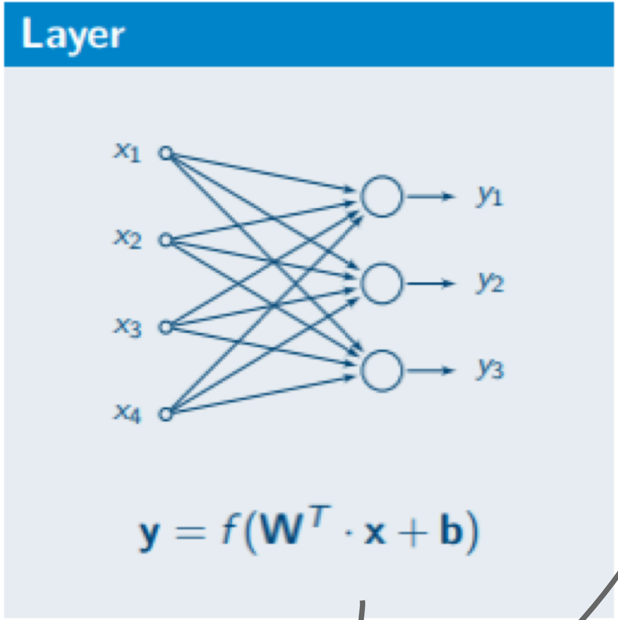
```
VOCAB_SIZE = 10000
HIDDEN_SIZE=100
# mapping a Vocabulary of size 10.000 to HIDDEN_SIZE projections
emb_1 = nn.Linear(VOCAB_SIZE, HIDDEN_SIZE)
```

```
# forward example [10, 10000] tensor
code = [1] + [0] * 9999
# copy 10 times the same code [1 0 0 0 ... 0]
x = torch.FloatTensor([code] * 10)
print('Input x tensor size: ', x.size())
y = emb_1(x)
print('Output y embedding size: ', y.size())
```

```
Input x tensor size:  torch.Size([10, 10000])
Output y embedding size:  torch.Size([10, 100])
```

# Embedding Layer

Each x as an Integer 0 or 1




CLASS `torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None)` [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

**Parameters:**

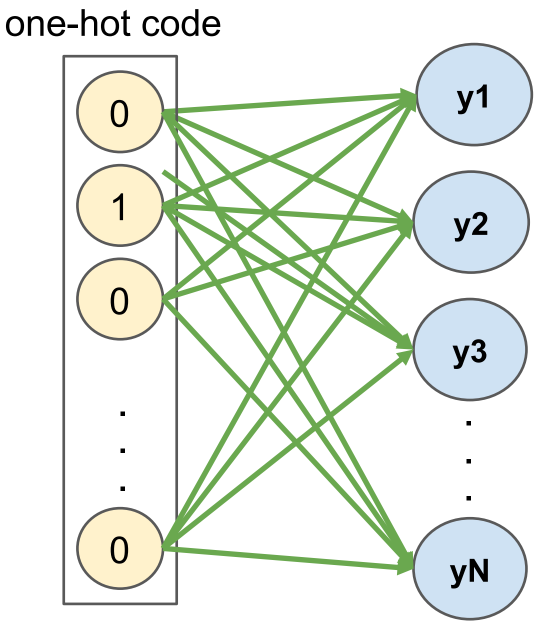
- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int, optional*) – If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- **max_norm** (*float, optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

**Variables:** **weight** (*Tensor*) – the learnable weights of the module of shape (num_embeddings, embedding_dim) initialized from $\mathcal{N}(0, 1)$

Shape:

- Input: LongTensor of arbitrary shape containing the indices to extract
- Output: (*, *embedding_dim*), where * is the input shape

# Fully Connected: Embedding Layer



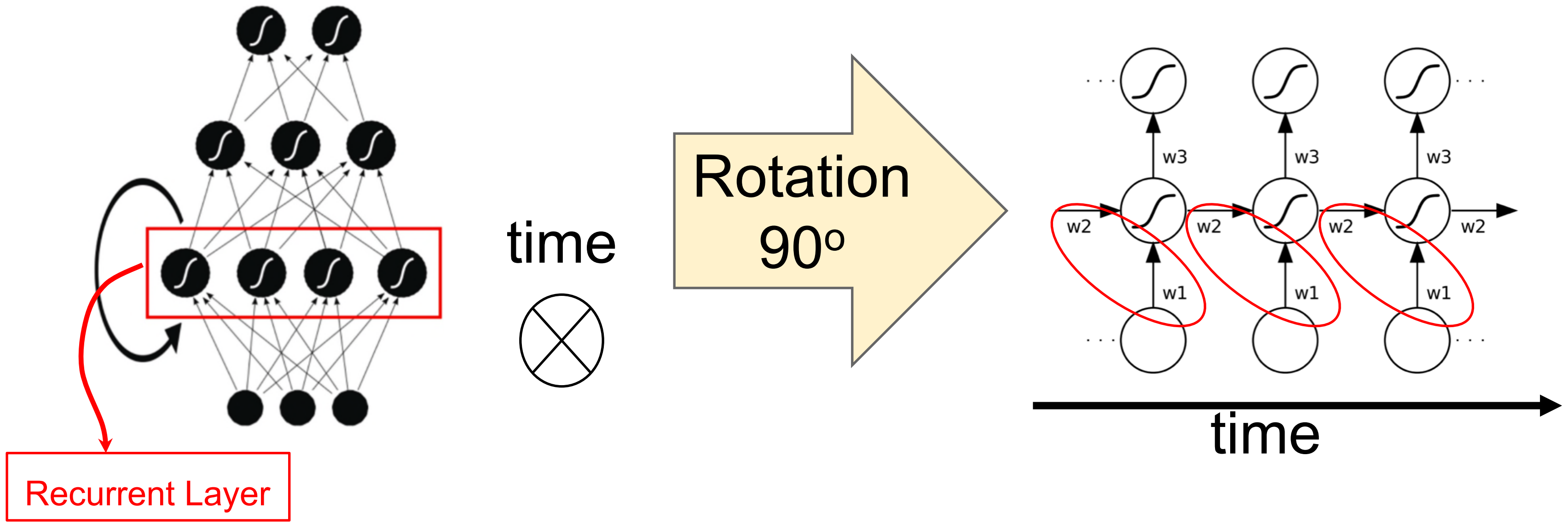

Embedding layer makes an efficient lookup operation, not a full matrix multiplication (**just select one-hot index column from weight matrix!**)

```
VOCAB_SIZE = 10000
HIDDEN_SIZE=100
# mapping a Vocabulary of size 10.000 to HIDDEN_SIZE projections
emb_2 = nn.Embedding(VOCAB_SIZE, HIDDEN_SIZE)
```

```
# Just make a long tensor with zero-index
x = torch.zeros(10, 1).long()
print('Input x tensor size: ', x.size())
y = emb_2(x)
print('Output y embedding size: ', y.size())
```

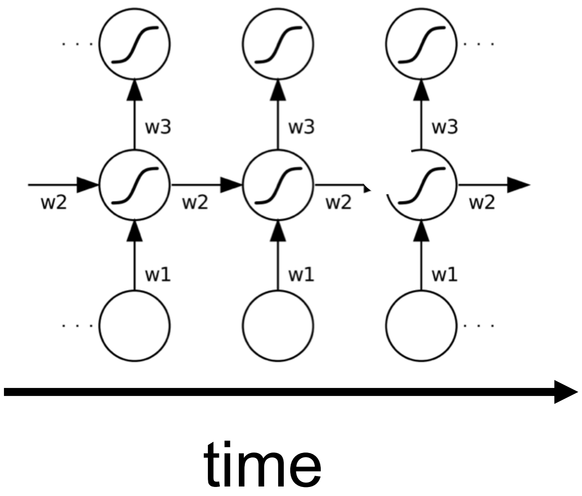
```
Input x tensor size:  torch.Size([10, 1])
Output y embedding size:  torch.Size([10, 1, 100])
```

# Recurrent Layer



# Recurrent Layer

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b)$$




CLASS `torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with $tanh$ or $ReLU$ non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(w_{ih} x_t + b_{ih} + w_{hh} h_{(t-1)} + b_{hh})$$

where $h_t$ is the hidden state at time *t*, $x_t$ is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is *'relu'*, then *ReLU* is used instead of *tanh*.

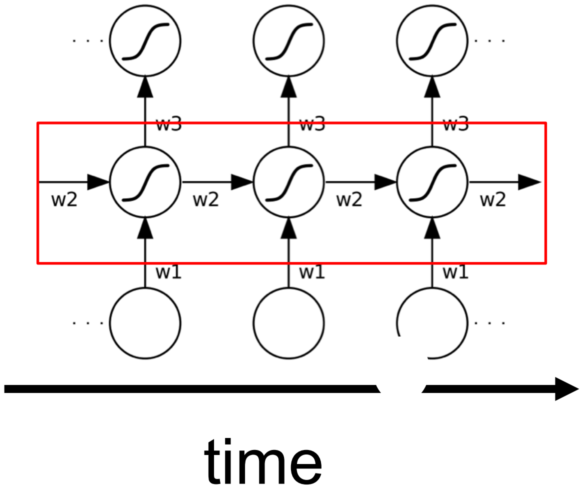
**Parameters:**

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Inputs: input, h_0

- **input** of shape (*seq_len, batch, input_size*): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0** of shape (*num_layers * num_directions, batch, hidden_size*): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

# Recurrent Layer



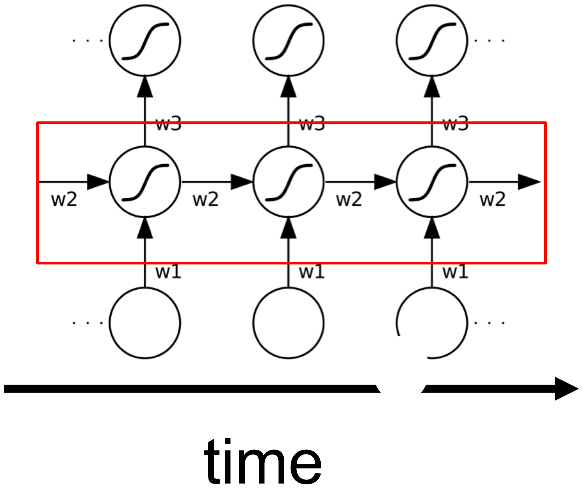

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b)$$

```
NUM_INPUTS = 100
HIDDEN_SIZE = 512
NUM_LAYERS = 1
# define a recurrent layer
rnn = nn.RNN(NUM_INPUTS, HIDDEN_SIZE, num_layers=NUM_LAYERS)
```

```
SEQ_LEN = 100
x = torch.randn(SEQ_LEN, 1, NUM_INPUTS)
print('Input tensor size [seq_len, bsize, hidden_size]: ', x.size())
ht, state = rnn(x, None)
print('Output tensor h[t] size [seq_len, bsize, hidden_size]: ', ht.size())
```

```
Input tensor size [seq_len, bsize, hidden_size]:  torch.Size([100, 1, 100])
Output tensor h[t] size [seq_len, bsize, hidden_size]:  torch.Size([100, 1, 512])
```

# Recurrent Layer



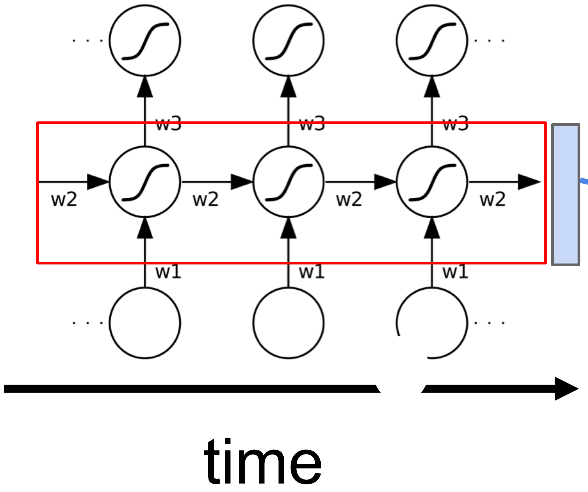

**BEWARE with axis definition in the 3D Tensor!**

```
NUM_INPUTS = 100
HIDDEN_SIZE = 512
NUM_LAYERS = 1
# define a recurrent layer
rnn = nn.RNN(NUM_INPUTS, HIDDEN_SIZE, num_layers=NUM_LAYERS)
```

```
SEQ_LEN = 100
x = torch.randn(1, SEQ_LEN, NUM_INPUTS)
print('Input tensor size [bsize, seq_len, hidden_size]: ', x.size())
ht, state = rnn(x, None)
print('Output tensor h[t] size [bsize, seq_len, hidden_size]: ', ht.size())
```

```
Input tensor size [bsize, seq_len, hidden_size]:  torch.Size([1, 100, 100])
Output tensor h[t] size [bsize, seq_len, hidden_size]:  torch.Size([1, 100, 512])
```

# Recurrent Layer




```
# let's check ht and state sizes
print('ht size: ', ht.size())
print('state size: ', state.size())
```

```
ht size:   torch.Size([1, 100, 512])
state size:   torch.Size([1, 1, 512])
```

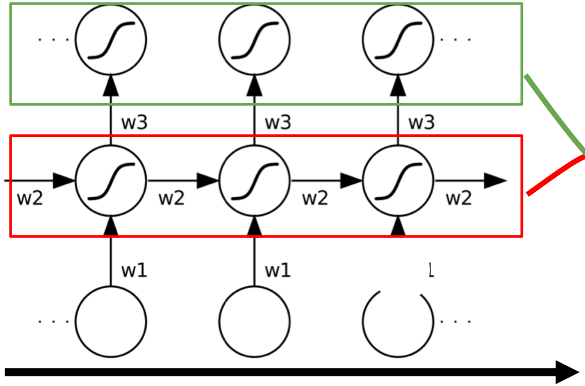
**Last time-step state**

```
SEQ_LEN = 100
x = torch.randn(1, SEQ_LEN, NUM_INPUTS)
print('Input tensor size [bsize, seq_len, hidden_size]: ', x.size())
ht, state = rnn(x, None)
print('Output tensor h[t] size [bsize, seq_len, hidden_size]: ', ht.size())
```

```
Input tensor size [bsize, seq_len, hidden_size]:  torch.Size([1, 100, 100])
Output tensor h[t] size [bsize, seq_len, hidden_size]:  torch.Size([1, 100, 512])
```

# Recurrent Layer

## Connecting an RNN with a FC layer



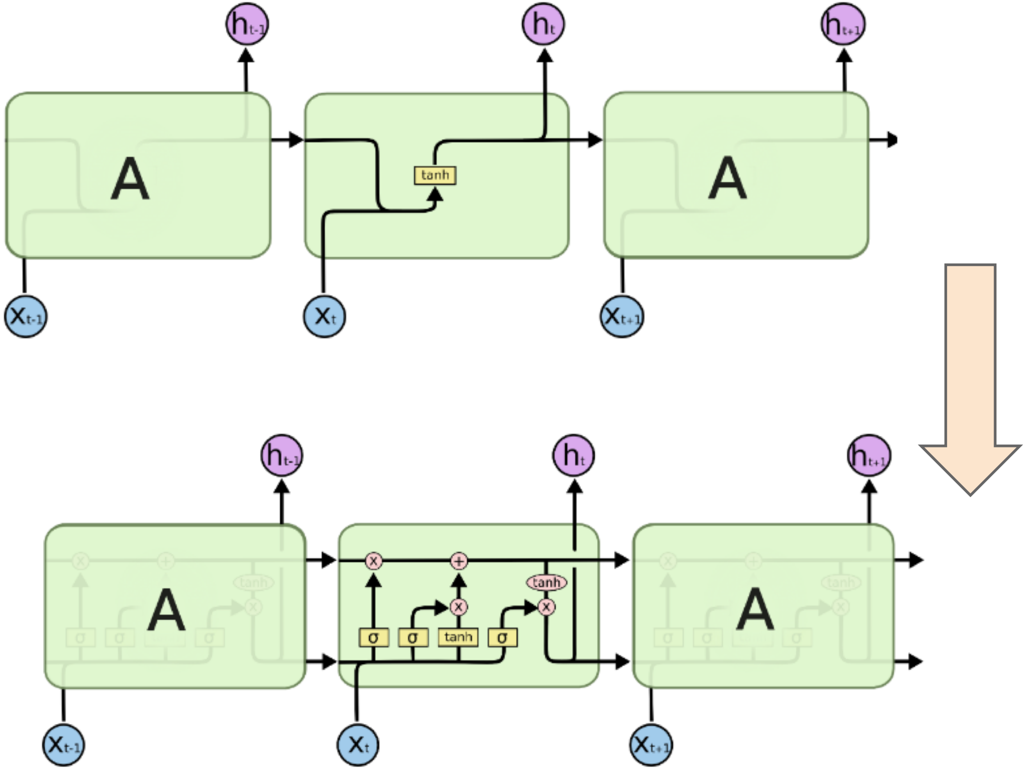

time

**BEWARE: with batch_first=False this straightforward connection would NOT work**

```
NUM_INPUTS = 100
NUM_OUTPUTS = 10
HIDDEN_SIZE = 512
SEQ_LEN = 100
NUM_LAYERS = 1
# define a recurrent layer, swapping batch and time axis and connect
# an FC layer as an output layer to build a full network
rnn = nn.RNN(NUM_INPUTS, HIDDEN_SIZE, num_layers=NUM_LAYERS,
             batch_first=True)
fc = nn.Sequential(
    nn.Linear(HIDDEN_SIZE, NUM_OUTPUTS),
    nn.LogSoftmax(dim=2)
)

x = torch.randn(1, SEQ_LEN, NUM_INPUTS)
print('Input tensor size x: ', x.size())
ht, state = rnn(x, None)
print('Hidden tensor size ht: ', ht.size())
y = fc(ht)
print('Output tensor y size: ', y.size())
```

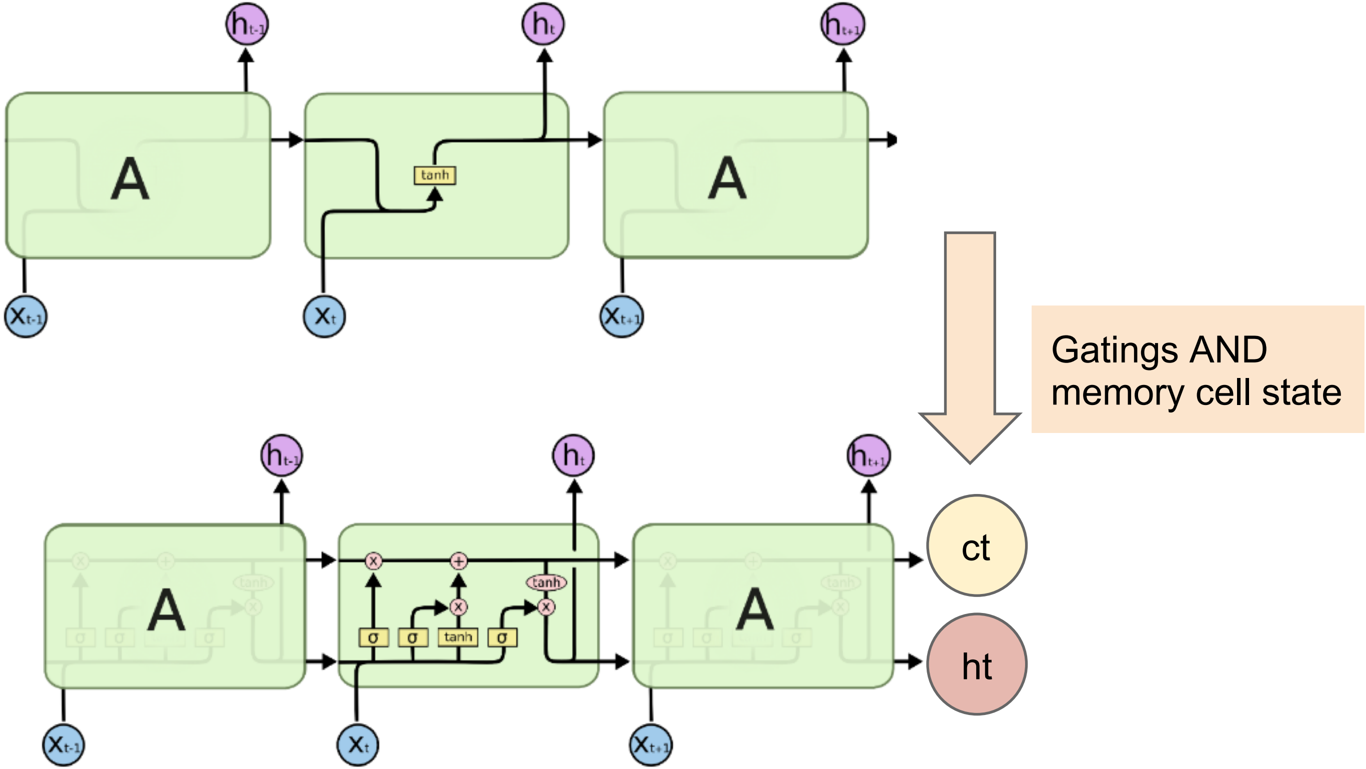
```
Input tensor size x:  torch.Size([1, 100, 100])
Hidden tensor size ht:  torch.Size([1, 100, 512])
Output tensor y size:  torch.Size([1, 100, 10])
```

# From RNN to LSTM

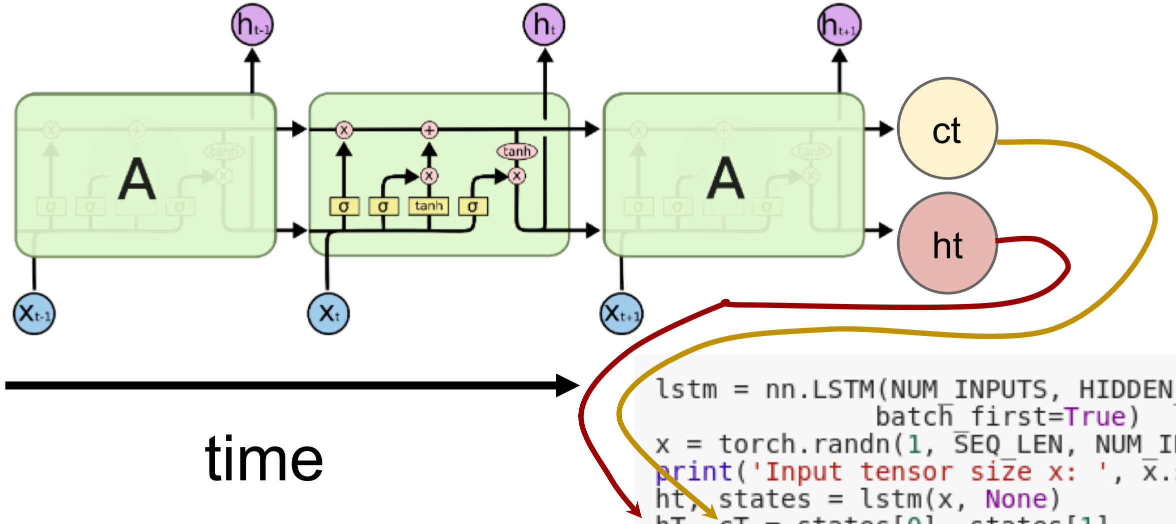



Gatings AND
memory cell state

# From RNN to LSTM



# LSTM Layer

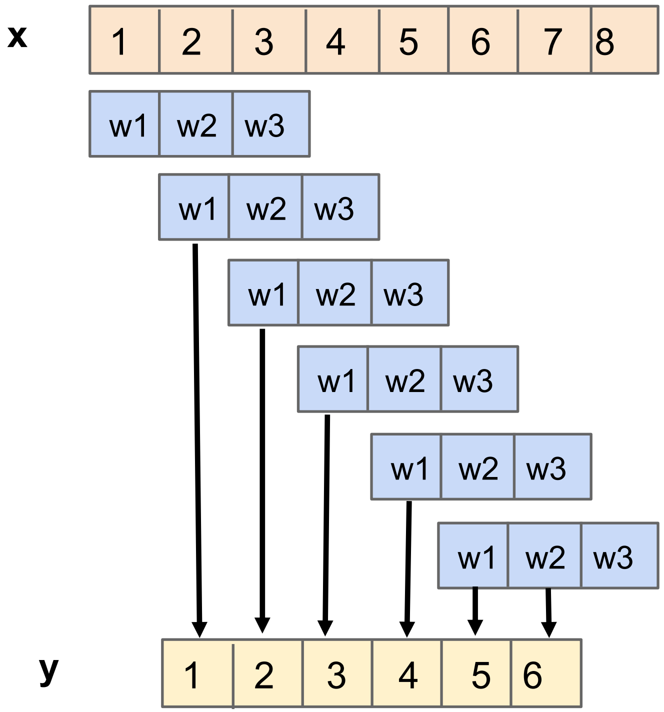



**BEWARE: LSTM has two states! The internal cumulative one, and the bounded output one.**

```
lstm = nn.LSTM(NUM_INPUTS, HIDDEN_SIZE, num_layers=NUM_LAYERS,
               batch_first=True)
x = torch.randn(1, SEQ_LEN, NUM_INPUTS)
print('Input tensor size x: ', x.size())
ht, states = lstm(x, None)
hT, cT = states[0], states[1]
print('Output tensor ht size: ', ht.size())
print('Last state h[T]: ', hT.size())
print('Cell state c[T]: ', cT.size())
```

```
Input tensor size x:  torch.Size([1, 100, 100])
Output tensor ht size:  torch.Size([1, 100, 512])
Last state h[T]:  torch.Size([1, 1, 512])
Cell state c[T]:  torch.Size([1, 1, 512])
```

# Convolutional Layer




CLASS `torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)` [SOURCE]

Applies a 1D convolution over an input signal composed of several input planes.

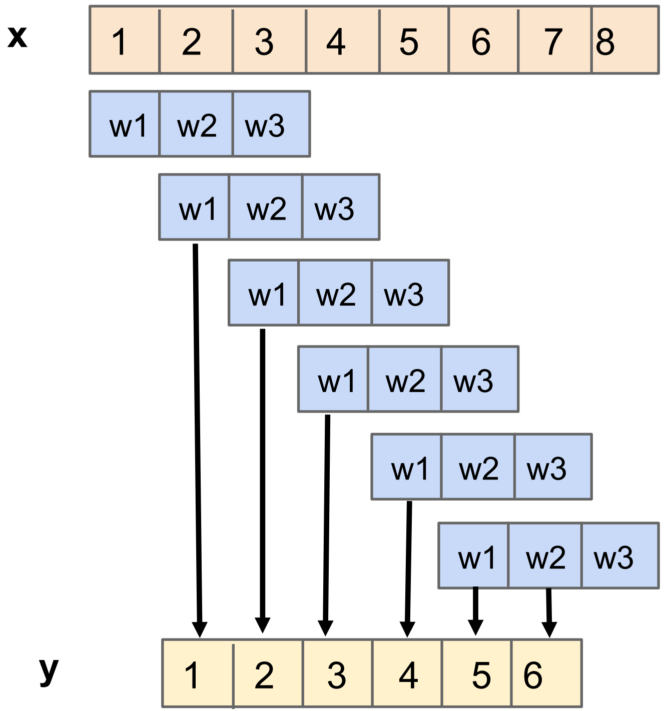
In the simplest case, the output value of the layer with input size $(N, C_{\text{in}}, L)$ and output $(N, C_{\text{out}}, L_{\text{out}})$ can be precisely described as:

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

where $\star$ is the valid cross-correlation operator, $N$ is a batch size, $C$ denotes a number of channels, $L$ is a length of signal sequence.

- `stride` controls the stride for the cross-correlation, a single number or a one-element tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
    - At groups=1, all inputs are convolved to all outputs.
    - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
    - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size $\left\lfloor \frac{C_{\text{out}}}{C_{\text{in}}} \right\rfloor$

# Convolutional Layer



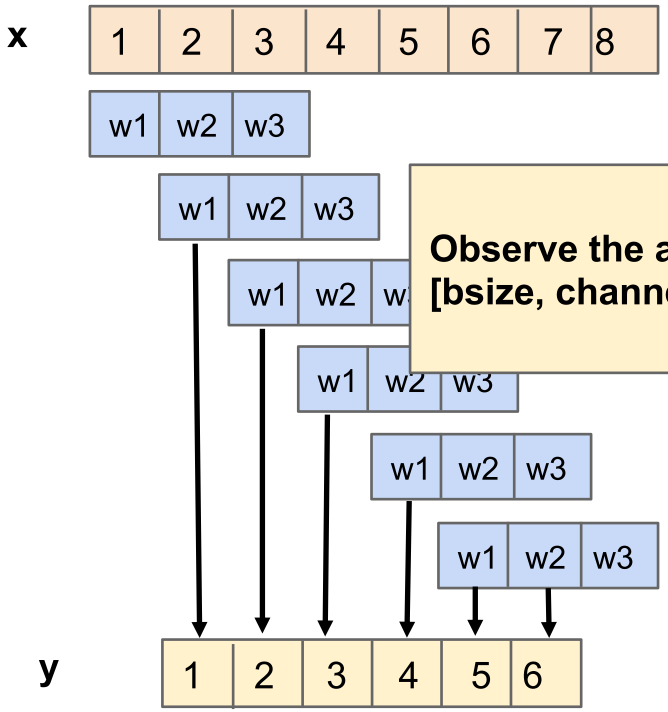

```
NUM_CHANNELS_IN = 1
HIDDEN_SIZE = 1024
KERNEL_WIDTH = 3
# Build a one-dimensional convolutional neural layer
conv1d = nn.Conv1d(NUM_CHANNELS_IN, HIDDEN_SIZE, KERNEL_WIDTH)
```

```
SEQ_LEN = 8
x = torch.randn(1, NUM_CHANNELS_IN, SEQ_LEN)
print('Input tensor size x: ', x.size())
y = conv1d(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor size x:  torch.Size([1, 1, 8])
Output tensor y size:  torch.Size([1, 1024, 6])
```

We obtain 1024 **y** sequences

# Convolutional Layer




```
NUM_CHANNELS_IN = 1
HIDDEN_SIZE = 1024
KERNEL_WIDTH = 3
# Build a one-dimensional convolutional neural layer
IDDEN_SIZE, KERNEL_WIDTH)
```

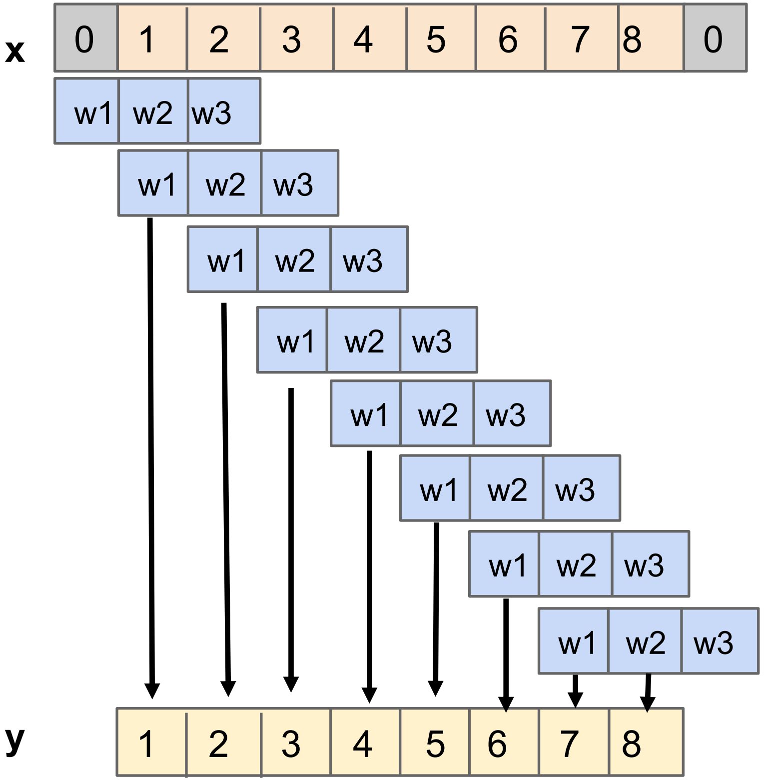
**Observe the axis shape of the convolutional layer: [bsize, channels, seq_len], different than RNN!**

```
EQ_LEN)
()))
y = conv1d(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor size x:  torch.Size([1, 1, 8])
Output tensor y size:  torch.Size([1, 1024, 6])
```

We obtain 1024 **y** sequences

# Convolutional Layer (padding)

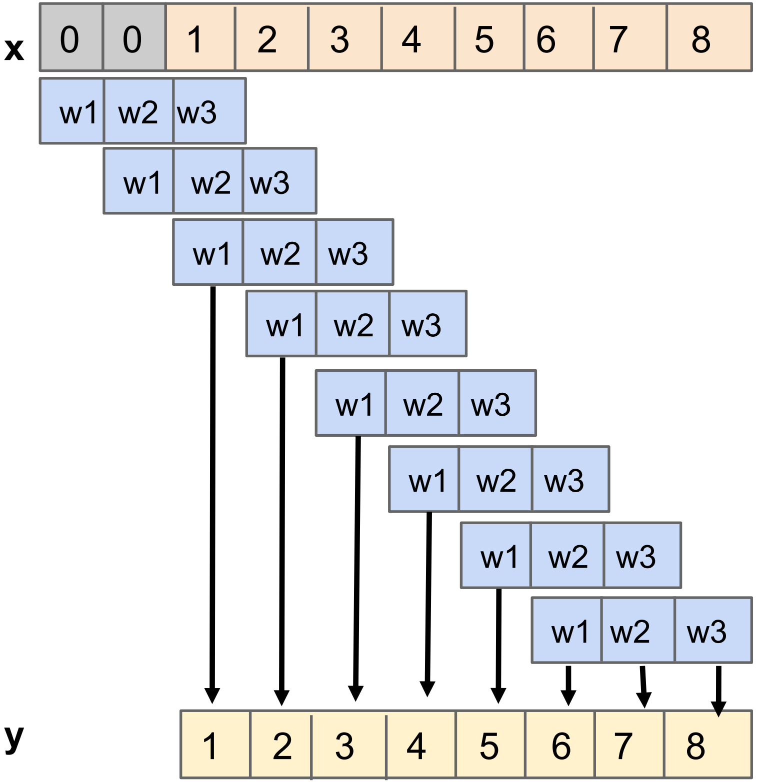



```
NUM_CHANNELS_IN = 1
HIDDEN_SIZE = 1024
KERNEL_WIDTH = 3
PADDING = KERNEL_WIDTH // 2 # = 1
# Build a one-dimensional convolutional neural layer
conv1d = nn.Conv1d(NUM_CHANNELS_IN, HIDDEN_SIZE, KERNEL_WIDTH,
                   padding=PADDING)

SEQ_LEN = 8
x = torch.randn(1, NUM_CHANNELS_IN, SEQ_LEN)
print('Input tensor size x: ', x.size())
y = conv1d(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor size x:  torch.Size([1, 1, 8])
Output tensor y size:  torch.Size([1, 1024, 8])
```

We obtain 1024 **y** sequences

# Causal Convolutional Layer

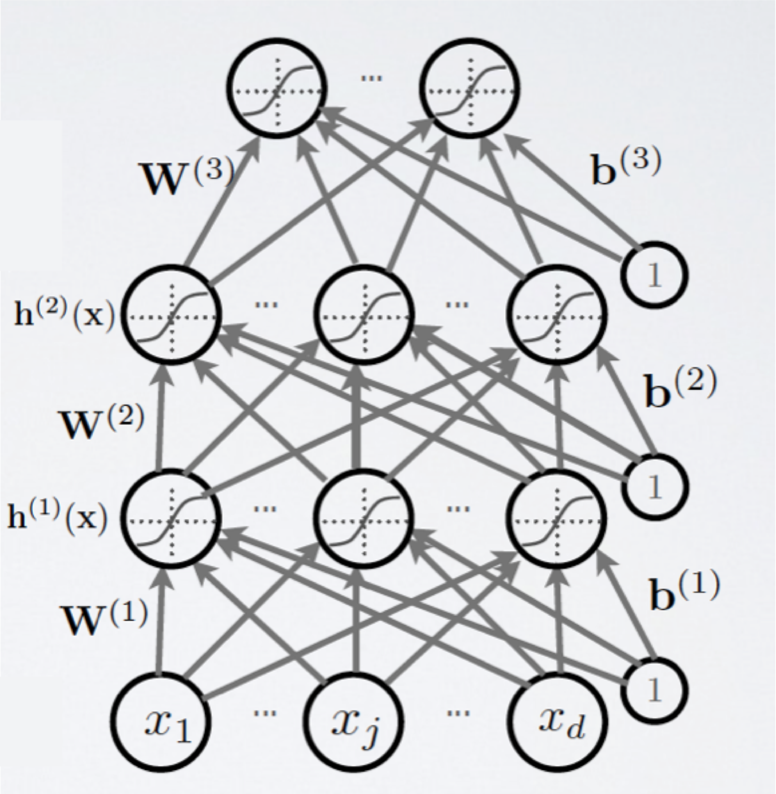



```
NUM_CHANNELS_IN = 1
HIDDEN_SIZE = 1024
KERNEL_WIDTH = 3
# Build a one-dimensional convolutional neural layer
conv1d = nn.Conv1d(NUM_CHANNELS_IN, HIDDEN_SIZE, KERNEL_WIDTH)

SEQ_LEN = 8
PADDING = KERNEL_WIDTH - 1 # = 2
x = torch.randn(1, NUM_CHANNELS_IN, SEQ_LEN)
print('Input tensor x size: ', x.size())
xpad = F.pad(x, (PADDING, 0))
print('Input tensor after padding xpad size: ', xpad.size())
y = conv1d(xpad)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 1, 8])
Input tensor after padding xpad size:  torch.Size([1, 1, 10])
Output tensor y size:  torch.Size([1, 1024, 8])
```

We obtain 1024 **y** sequences

# Multi-Layer Perceptron

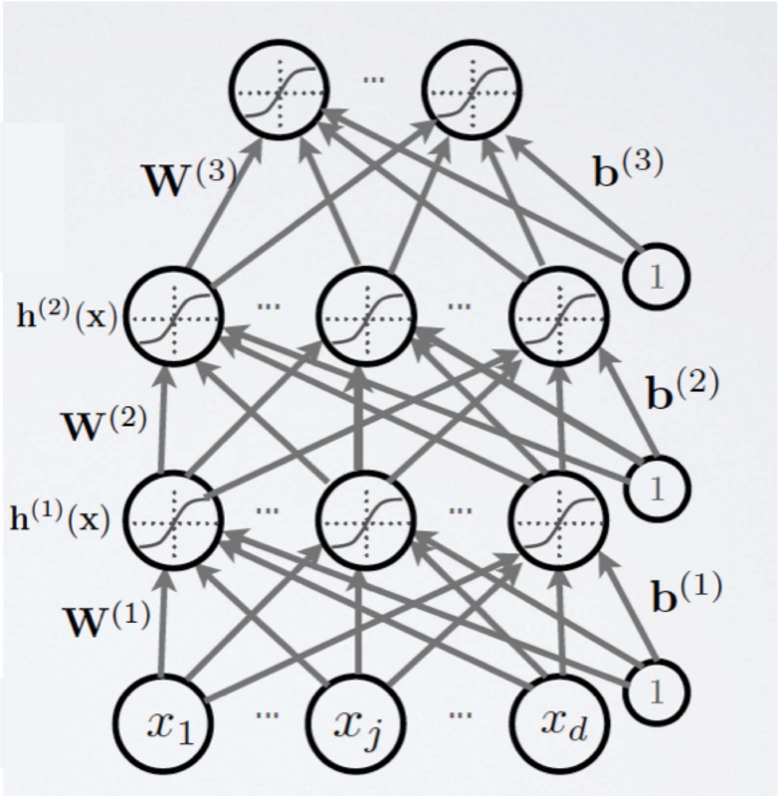



Slide Credit: Hugo Laroche NN course

```
NUM_INPUTS = 100
HIDDEN_SIZE = 1024
NUM_OUTPUTS= 20
# MLP as a CNN
mlp = nn.Sequential(
    nn.Conv1d(NUM_INPUTS, HIDDEN_SIZE, 1),
    nn.Tanh(),
    nn.Conv1d(HIDDEN_SIZE, HIDDEN_SIZE, 1),
    nn.Tanh(),
    nn.Conv1d(HIDDEN_SIZE, NUM_OUTPUTS, 1),
    nn.LogSoftmax(dim=1)
)

x = torch.randn(1, 100, 1)
print('Input tensor x size: ', x.size())
y = mlp(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 100, 1])
Output tensor y size:  torch.Size([1, 20, 1])
```

# Multi-Layer Perceptron




*Slide Credit: Hugo Laroche NN course*

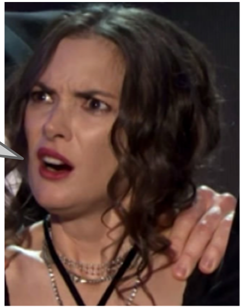
```
NUM_INPUTS = 100
HIDDEN_SIZE = 1024
NUM_OUTPUTS= 20
# MLP as a CNN
mlp = nn.Sequential(
        nn.Conv1d(NUM_INPUTS, HIDDEN_SIZE, 1),
        nn.Tanh(),
        nn.Conv1d(HIDDEN_SIZE, HIDDEN_SIZE, 1),
        nn.Tanh(),
        nn.Conv1d(HIDDEN_SIZE, NUM_OUTPUTS, 1),
        nn.LogSoftmax(dim=1)
)

x = torch.randn(1, 100, 1)
print('Input tensor x size: ', x.size())
y = mlp(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 100, 1])
Output tensor y size:  torch.Size([1, 20, 1])
```

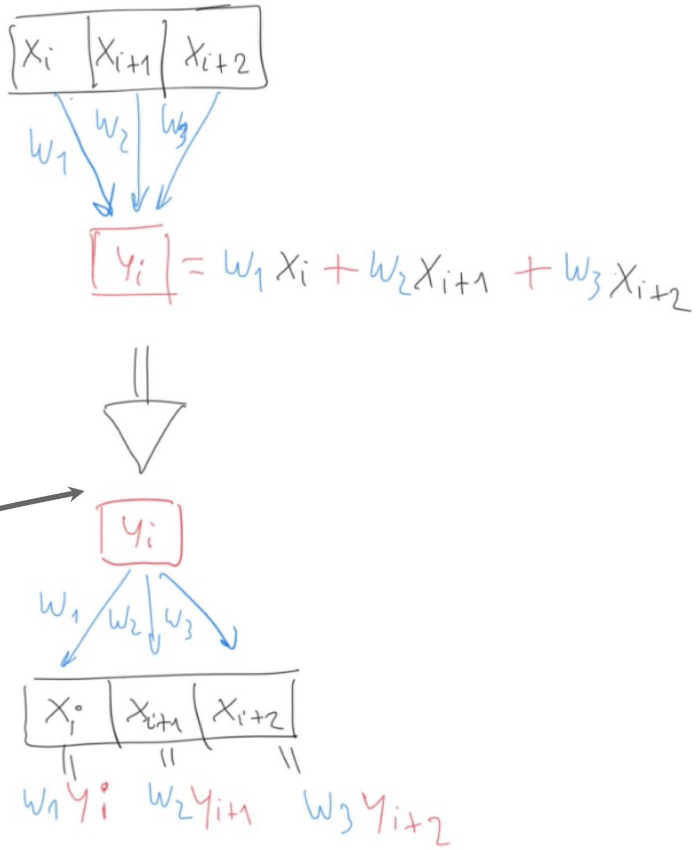
kernel of width 1 is equivalent to a fully connected layer sweeping in time!

Time axis MUST be included

AN MLP? Whut??



# Transposed Convolutional Layer

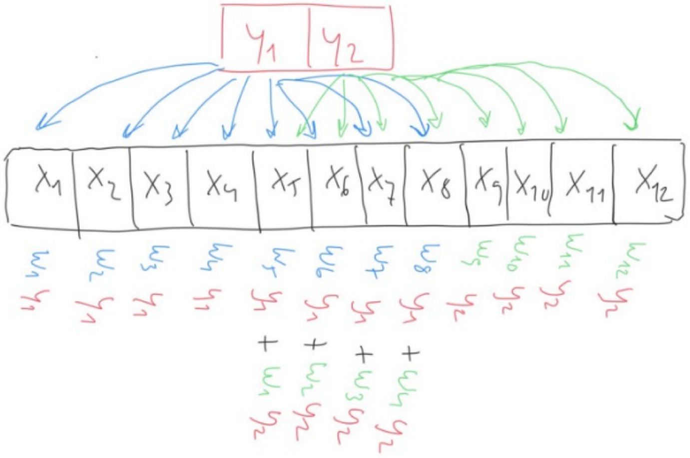



Revert the convolution operation, sending weighted pieces of input to make many outputs.

[Medium post on “How PyTorch Transposed Convs1D Work”]

# Transposed Convolutional Layer

**It works as a learnable upsampler!**




Example with x6 upscaling factor (stride of 4)

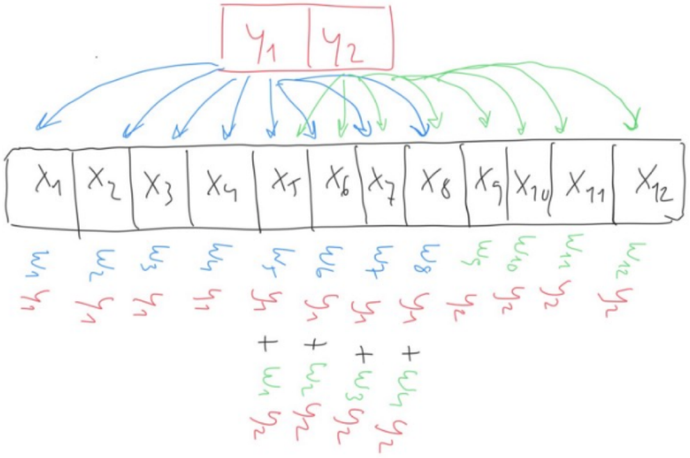
CLASS `torch.nn.ConvTranspose1d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True, dilation=1)` [SOURCE]

Applies a 1D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv1d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `kernel_size - 1 - padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor$).

# Transposed Convolutional Layer



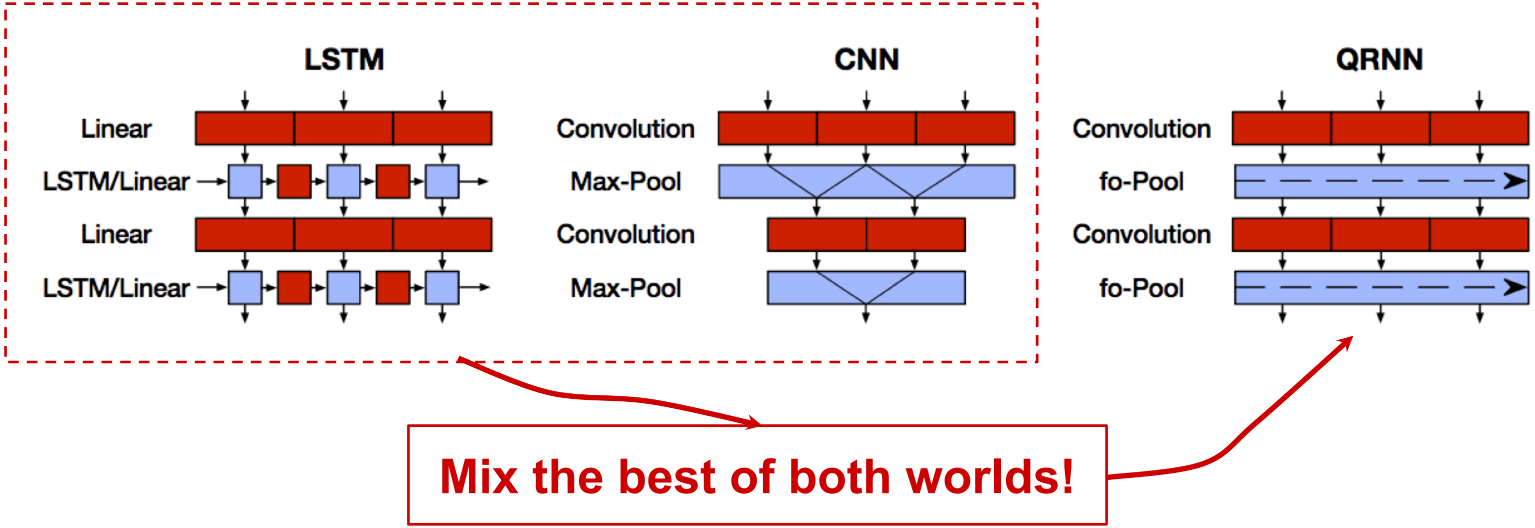
```python
NUM_CHANNELS_IN = 1
HIDDEN_SIZE = 1
KERNEL_WIDTH = 8
STRIDE = 4

deconv = nn.ConvTranspose1d(NUM_CHANNELS_IN, HIDDEN_SIZE, KERNEL_WIDTH,
                            stride=STRIDE)

SEQ_LEN = 2
y = torch.randn(1, NUM_CHANNELS_IN, SEQ_LEN)
print('Input tensor y size: ', y.size())
x = deconv(y)
print('Output (interpolated) tensor x size: ', x.size())
```

```
Input tensor y size:  torch.Size([1, 1, 2])
Output (interpolated) tensor x size:  torch.Size([1, 1, 12])
```

# Advanced Architectures

# Quasi Recurrent Neural Network (QRNN)

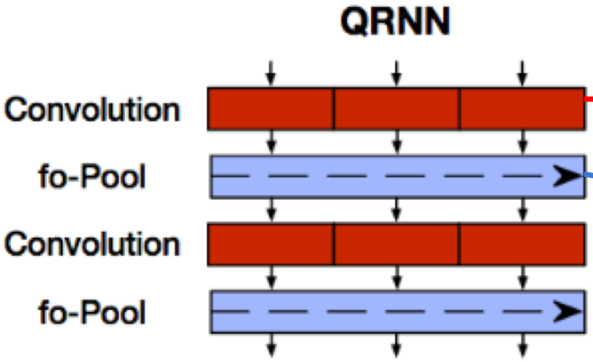
Advantage of CNN: We can compute all convolutions in parallel
Advantage of LSTM: It imposes the sense of order (appropriate for sequences)
QRNNs are up to x16 times faster than LSTMs!!




[Quasi-Recurrent Neural Networks (Bradbury et al. 2016)]

# Quasi Recurrent Neural Network (QRNN)

(1) Use a causal CNN to first forward all inputs sequentially; (2) and then accumulate long-term memory to impose ordered processing with simple pooling.




$$\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{z}_t,$$

```python
class fQRNNLayer(nn.Module):

  def __init__(self, num_inputs, num_outputs,
               kwidth=2):
    super().__init__()
    self.num_inputs = num_inputs
    self.num_outputs = num_outputs
    self.kwidth = kwidth
    # double feature maps for zt and ft predictions with same conv layer
    self.conv = nn.Conv1d(num_inputs, num_outputs * 2, kwidth)

  def forward(self, x, state=None):
    # x is [bsz, seq_len, num_inputs]
    # state is [bsz, num_outputs] dimensional
    # ---------- FEED FORWARD PART
    # inference convolutional part
    # transpose x axis first to work with CNN layer
    x = x.transpose(1, 2)
    pad = self.kwidth - 1
    xp = F.pad(x, (pad, 0))
    conv_h = self.conv(xp)
    # split convolutional layer feature maps into zt (new state
    # candidate) and forget activation ft
    zt, ft = torch.chunk(conv_h, 2, dim=1)
    # Convert forget gate into actual forget
    ft = torch.sigmoid(ft)
    # Convert zt into actual non-linear response
    zt = torch.tanh(zt)
    # ---------- SEQUENTIAL PART
    # iterate through time now to make pooling
    seqlen = ft.size(2)
    if state is None:
      # create the zero state
      ht_1 = torch.zeros(ft.size(0), self.num_outputs, 1)
    else:
      # add the dim=2 to match 3D tensor shape
      ht_1 = state.unsqueeze(2)
    zts = torch.chunk(zt, zt.size(2), dim=2)
    fts = torch.chunk(ft, ft.size(2), dim=2)
    hts = []
    for t in range(seqlen):
      ht = ht_1 * fts[t] + (1 - fts[t]) * zts[t]
      # transpose time, channels dims again to match RNN-like shape
      hts.append(ht.transpose(1, 2))
      # re-assign h[t-1] now
      ht_1 = ht
    # convert hts list into a 3D tensor [bsz, seq_len, num_outputs]
    hts = torch.cat(hts, dim=1)
    return hts, ht_1.squeeze(2)
```

$$\mathbf{z}_t = \tanh(\mathbf{W}_z^1 \mathbf{x}_{t-1} + \mathbf{W}_z^2 \mathbf{x}_t)$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_f^1 \mathbf{x}_{t-1} + \mathbf{W}_f^2 \mathbf{x}_t)$$

# Quasi Recurrent Neural Network (QRNN)

(1) Use a causal CNN to first forward all inputs sequentially; (2) and then accumulate long-term memory to impos

```
class fQRNNLayer(nn.Module):

  def __init__(self, num_inputs, num_outputs,
               kwidth=2):
    super().__init__()
    self.num_inputs = num_inputs
    self.num_outputs = num_outputs
    self.kwidth = kwidth
    # double feature maps for zt and ft predictions with same conv layer
    self.conv = nn.Conv1d(num_inputs, num_outputs * 2, kwidth)
```

**Good for advanced sequential processing: comparative performance to that of LSTMs at x16 less computational cost (with good CUDA implementation). Currently used by Google and Baidu for state of the art text/speech synthesis.**

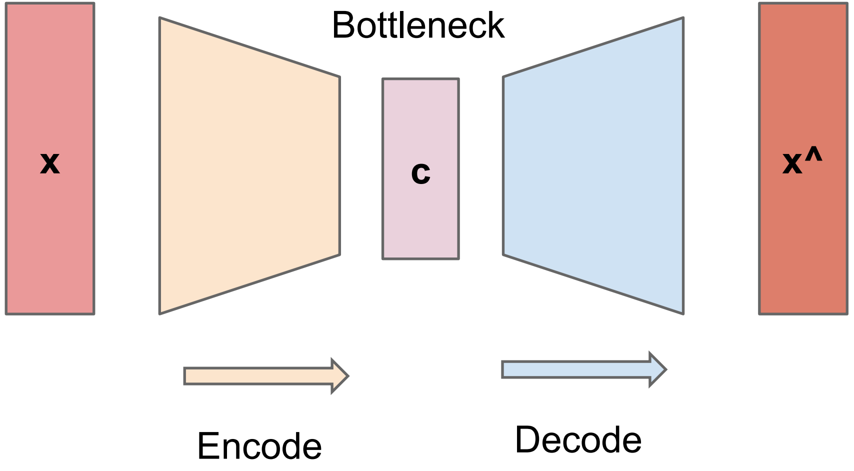
$\ldots\mathrm{h}(\mathbf{W}_z^1\mathbf{x}_{t-1} + \mathbf{W}_z^2\mathbf{x}_t)$

$\mathbf{W}_f^1\mathbf{x}_{t-1} + \mathbf{W}_f^2\mathbf{x}_t)$

```
      # add the dim=2 to match 3D tensor shape
      ht_1 = state.unsqueeze(2)
    zts = torch.chunk(zt, zt.size(2), dim=2)
    fts = torch.chunk(ft, ft.size(2), dim=2)
    hts = []
    for t in range(seqlen):
      ht = ht_1 * fts[t] + (1 - fts[t]) * zts[t]
      # transpose time, channels dims again to match RNN-like shape
      hts.append(ht.transpose(1, 2))
      # re-assign h[t-1] now
      ht_1 = ht
    # convert hts list into a 3D tensor [bsz, seq_len, num_outputs]
    hts = torch.cat(hts, dim=1)
    return hts, ht_1.squeeze(2)
```

$$\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{z}_t,$$

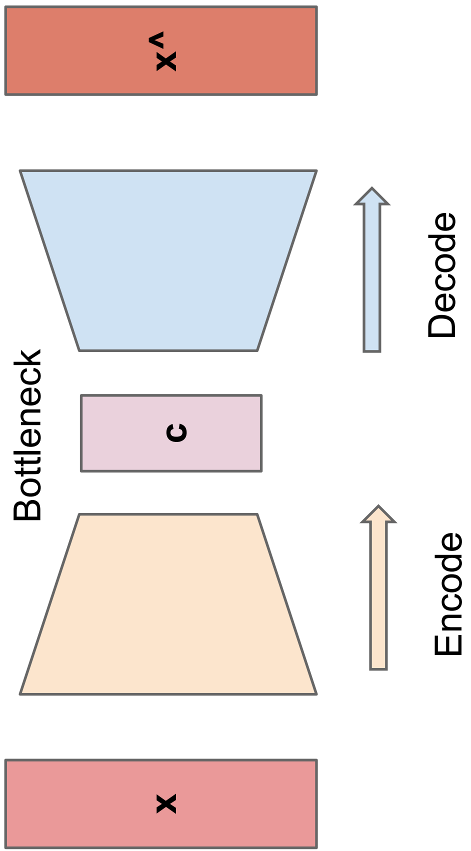
# Auto-Encoder Neural Network




Autoencoders:

- Predict at the output the same input data.
- Do not need labels

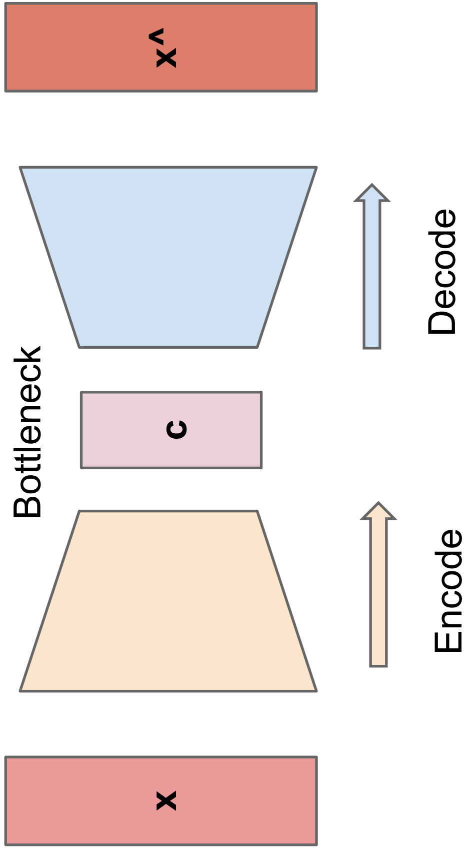
# Auto-Encoder Neural Network

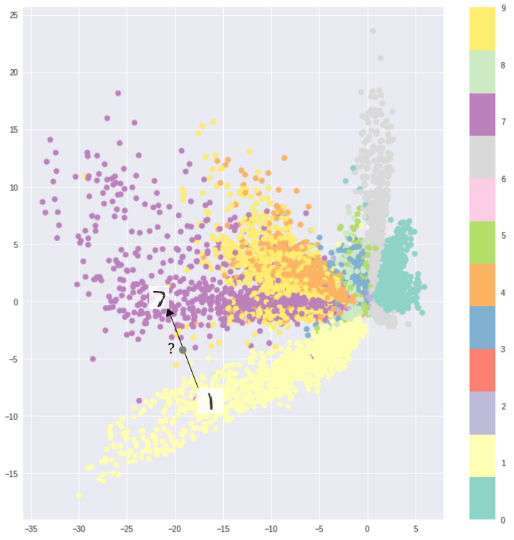


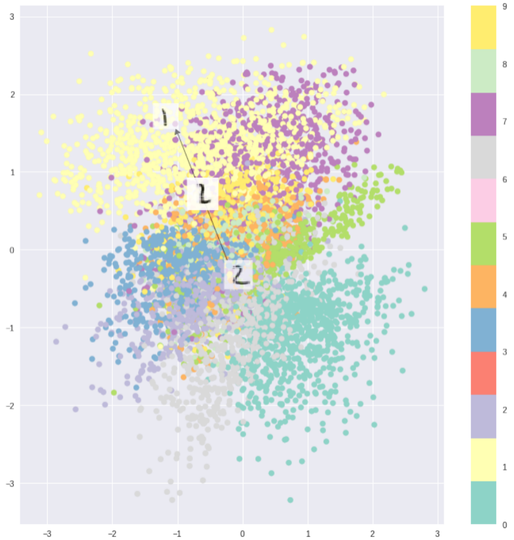

```python
class AE(nn.Module):
    def __init__(self, num_inputs=784):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(num_inputs, 400),
            nn.ReLU(inplace=True),
            nn.Linear(400, 400),
            nn.ReLU(inplace=True),
            nn.Linear(400, 20)
        )
        self.decoder = nn.Sequential(
            nn.Linear(20, 400),
            nn.ReLU(inplace=True),
            nn.Linear(400, 400),
            nn.ReLU(inplace=True),
            nn.Linear(400, num_inputs)
        )

    def forward(self, x):
        return self.decoder(self.encoder(x))


ae = AE(784)
x = torch.randn(10, 784)
print('Input tensor x size: ', x.size())
y = ae(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([10, 784])
Output tensor y size:  torch.Size([10, 784])
```

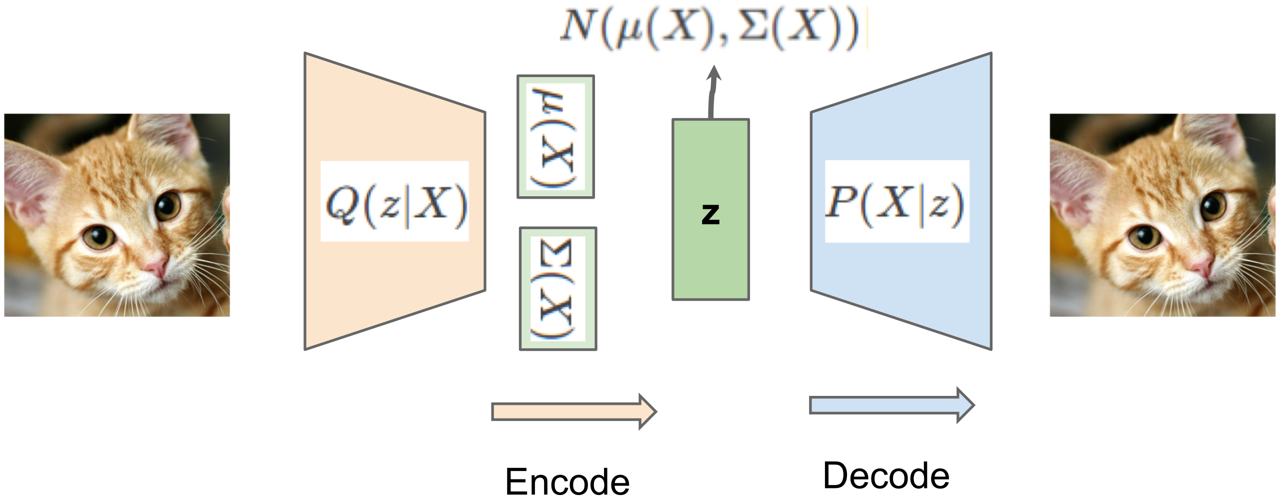
# Variational Auto-Encoder



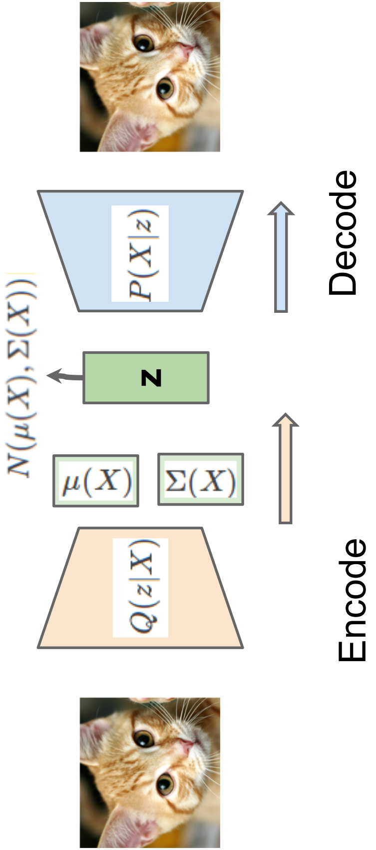





# Variational Auto-Encoder



# Variational Auto-Encoder




```python
# from https://github.com/pytorch/examples/blob/master/vae/main.py
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

vae = VAE()
x = torch.randn(10, 784)
print('Input tensor x size: ', x.size())
y, mu, logvar = vae(x)
print('Input tensor y size: ', y.size())
print('Mean tensor mu size: ', mu.size())
print('Covariance tensor logvar size: ', logvar.size())
```

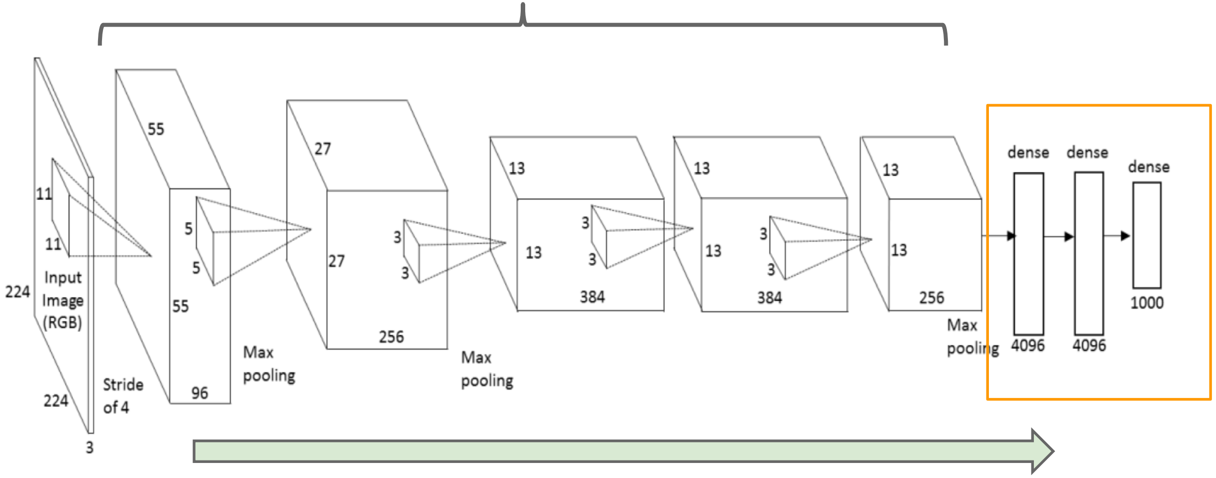
reparam. trick

```
Input tensor x size:  torch.Size([10, 784])
Input tensor y size:  torch.Size([10, 784])
Mean tensor mu size:  torch.Size([10, 20])
Covariance tensor logvar size:  torch.Size([10, 20])
```

# Deep Classifiers/Regressors

Front-end specific to signal type:

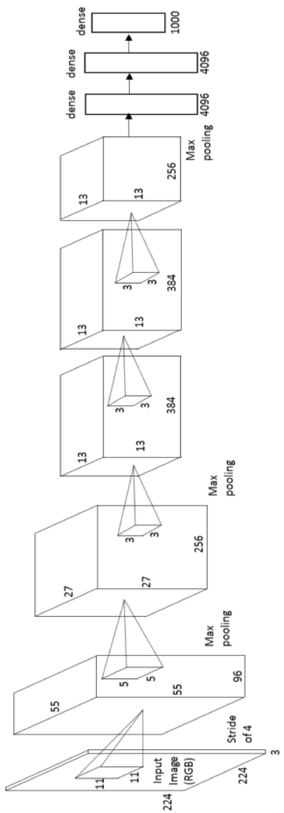
- Images: Conv2D
- Video: Conv2D + RNN or Conv3D
- Text: Conv1D or RNN or both
- Audio: Conv1d or Conv2d or RNN or combinations




MLP decisor with classification or regression output.

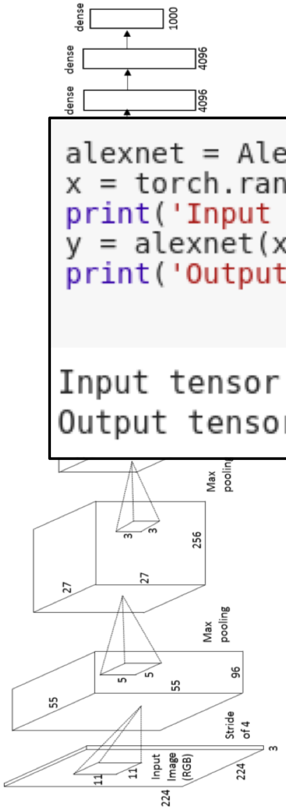
Pooling: MaxPool, AvgPool, Strided Convolutions...

# Deep Classifiers/Regressors




```python
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```

# Deep Classifiers/Regressors




```python
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
```

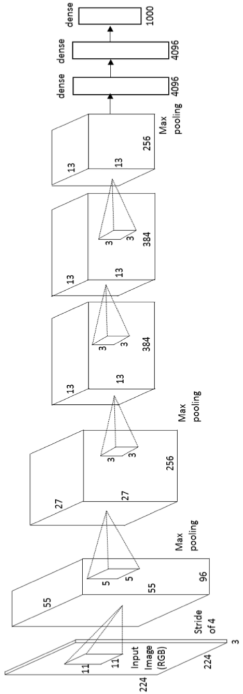
```python
alexnet = AlexNet()
x = torch.randn(1, 3, 224, 224)
print('Input tensor x size: ', x.size())
y = alexnet(x)
print('Output tensor y size: ', y.size())
```

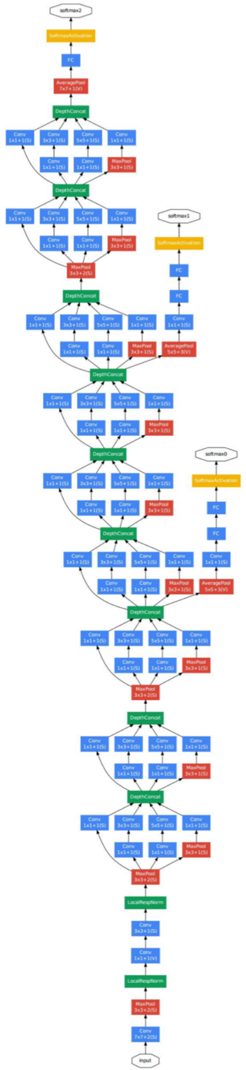
```
Input tensor x size:  torch.Size([1, 3, 224, 224])
Output tensor y size:  torch.Size([1, 1000])
```

```
ride=2),
=5, padding=2),

ride=2),
e=3, padding=1),

e=3, padding=1),

e=3, padding=1),

ride=2),

d((6, 6))
```

```python
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```

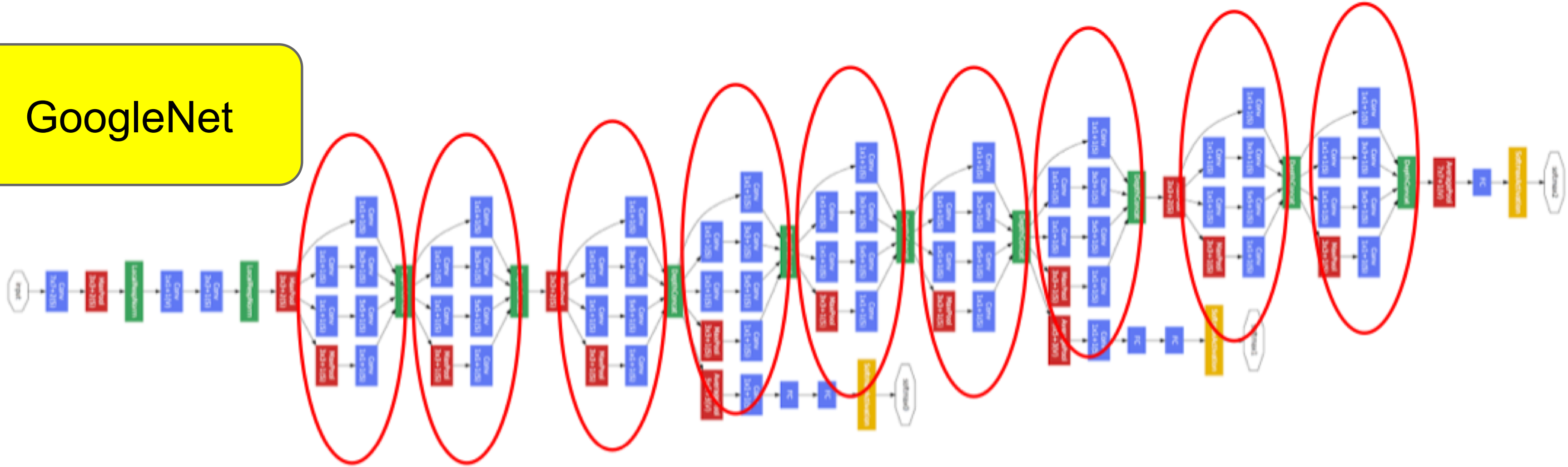
# Inception module



# Inception module / Network in Network

GoogleNet



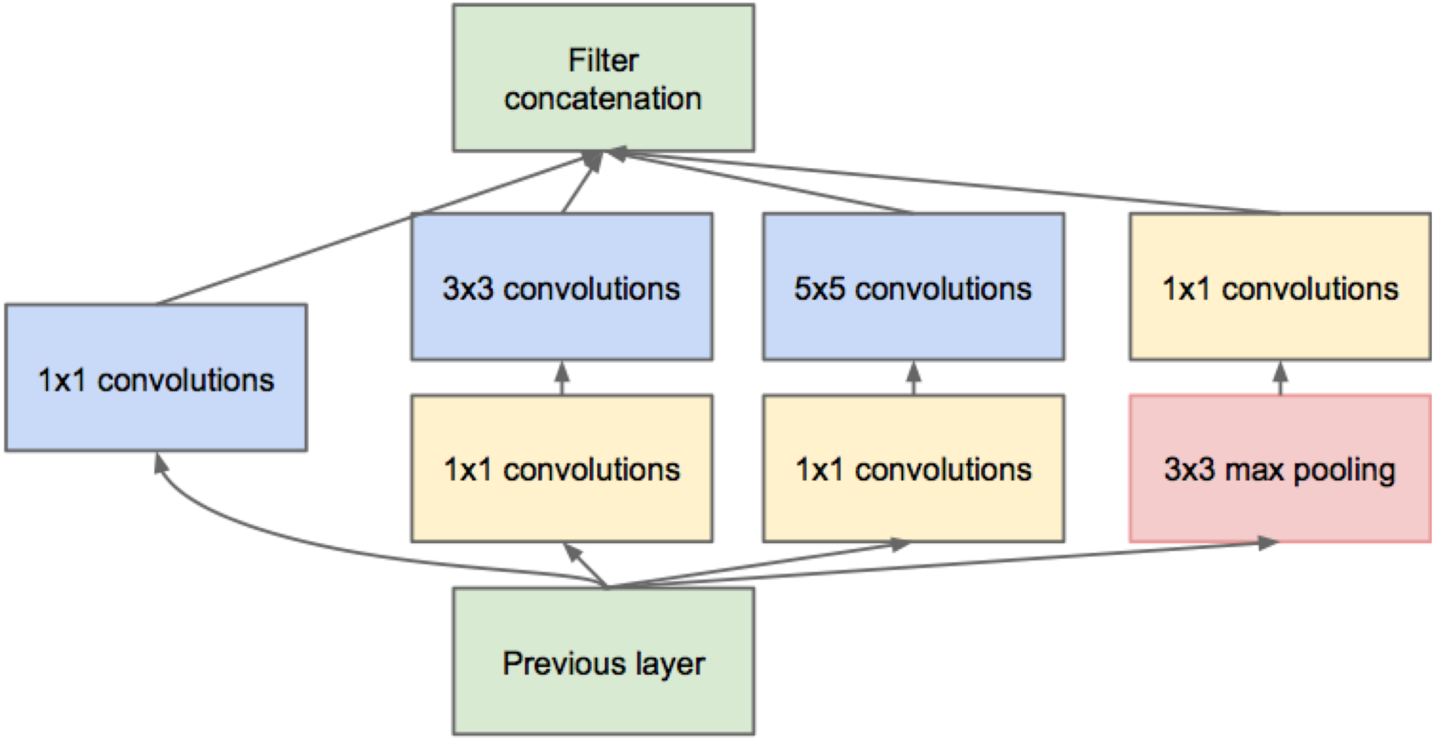

9 **Inception** modules

Network in a network in a network...

**Convolution**
**Pooling**
**Softmax**
**Other**

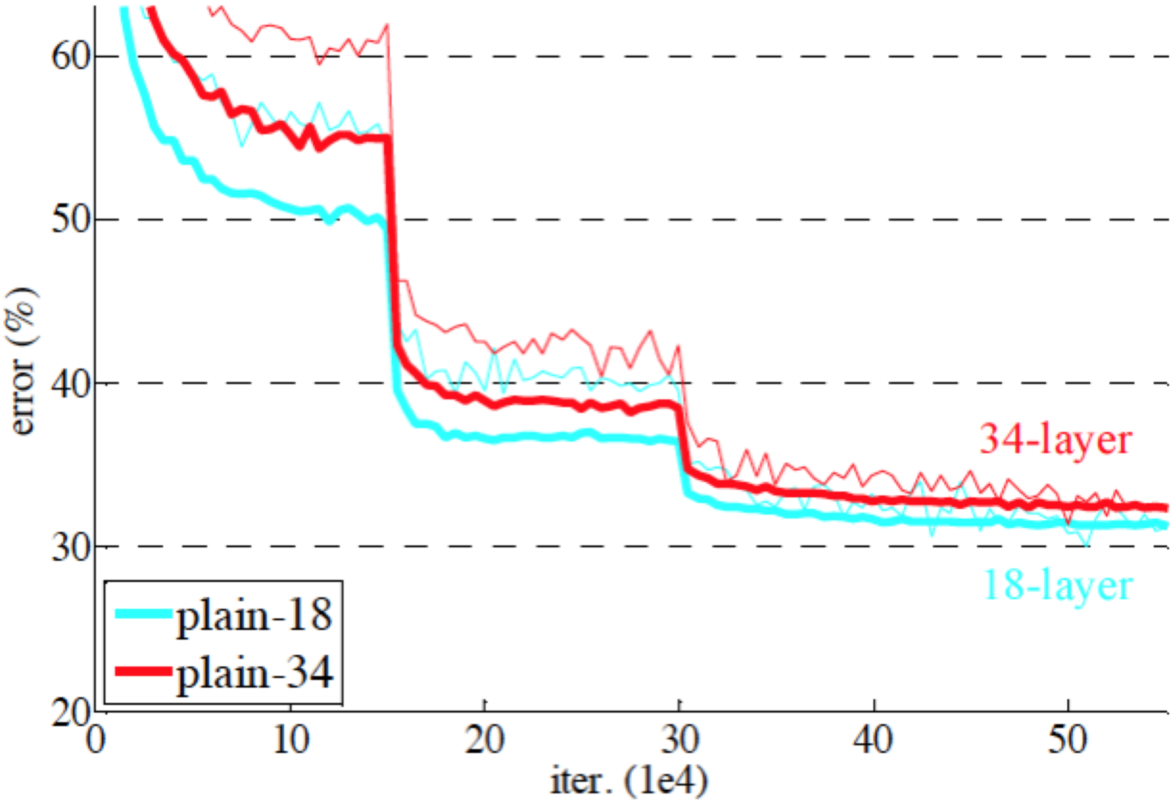
Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. ["Going deeper with convolutions."]() CVPR 2015

# Inception module / Network in Network




Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." *ICLR 2014.*

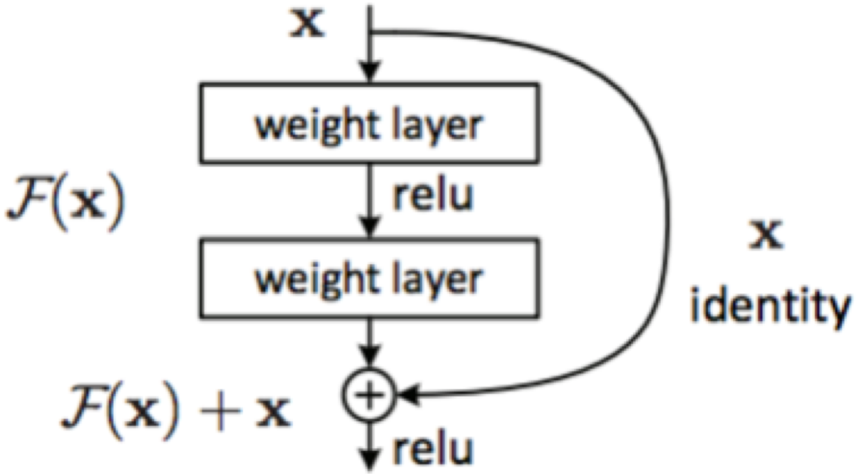
# Deep Residual Networks




He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. ["Deep residual learning for image recognition."]() CVPR 2016
[[slides]]()

# Deep Residual Networks

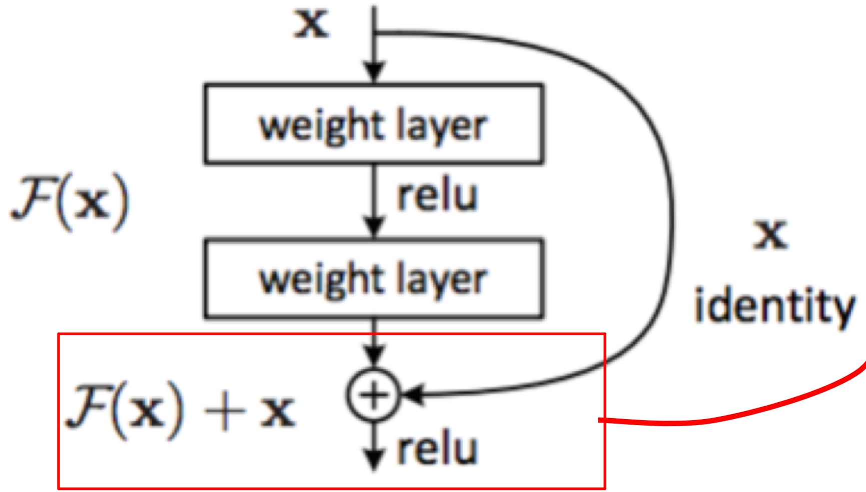
Residual learning: reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions




He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." CVPR 2016 [slides]

# Deep Residual Networks

Residual learning: reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions

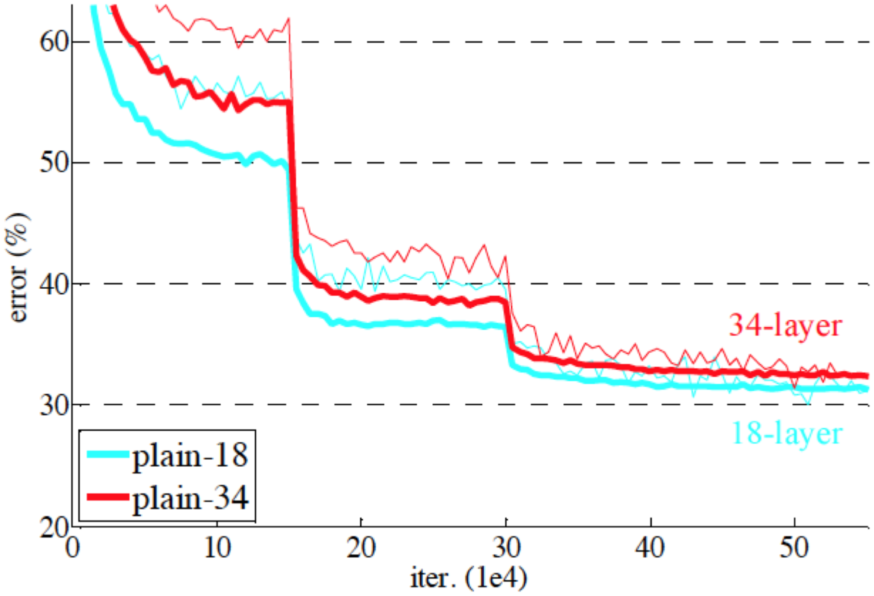



```python
class ResLayer(nn.Module):

  def __init__(self, num_inputs):
    super().__init__()
    self.num_inputs = num_inputs
    num_outputs = num_inputs
    self.num_outputs = num_outputs
    self.conv1 = nn.Sequential(
        nn.Conv2d(num_inputs, num_outputs, 3, padding=1),
        nn.BatchNorm2d(num_outputs),
        nn.ReLU(inplace=True)
    )
    self.conv2 = nn.Sequential(
        nn.Conv2d(num_outputs, num_outputs, 3, padding=1),
        nn.BatchNorm2d(num_outputs),
        nn.ReLU(inplace=True)
    )
    self.out_relu = nn.ReLU(inplace=True)

  def forward(self, x):
    # non-linear processing trunk
    conv1_h = self.conv1(x)
    conv2_h = self.conv2(conv1_h)
    # output is result of res connection + non-linear processing
    y = self.out_relu(x + conv2_h)
    return y

x = torch.randn(1, 64, 100, 100)
print('Input tensor x size: ', x.size())
reslayer = ResLayer(64)
y = reslayer(x)
print('Output tensor y size: ', y.size())
```

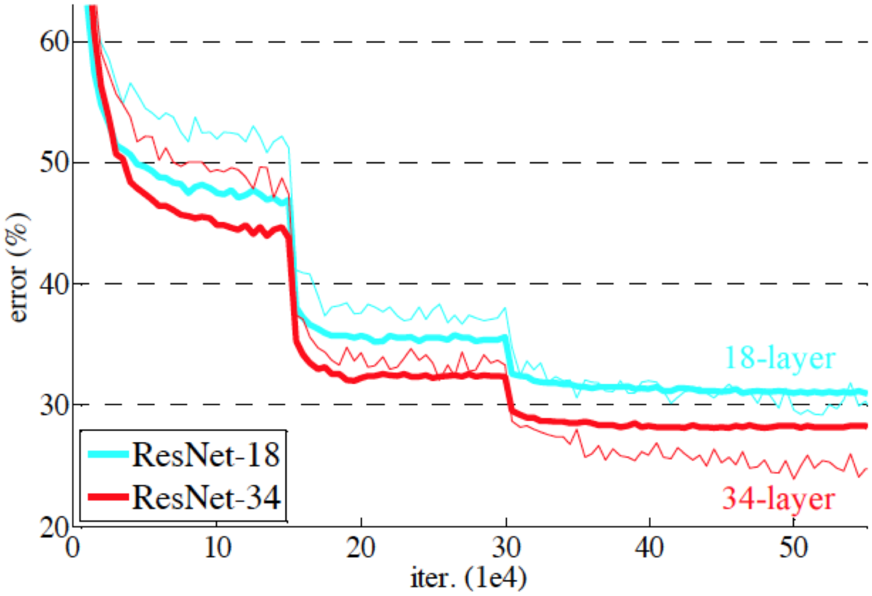
```
Input tensor x size:  torch.Size([1, 64, 100, 100])
Output tensor y size:  torch.Size([1, 64, 100, 100])
```

# Deep Residual Networks

Non-Residual connections

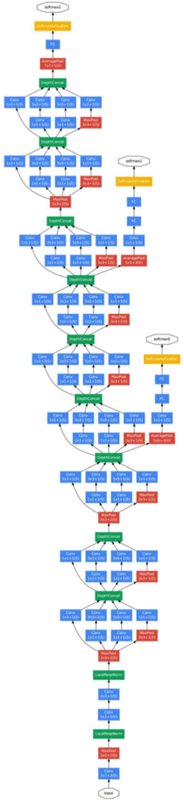


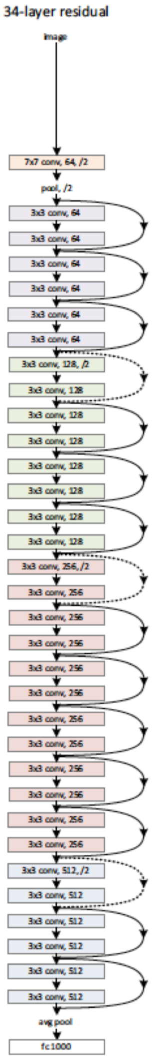

Residual connections

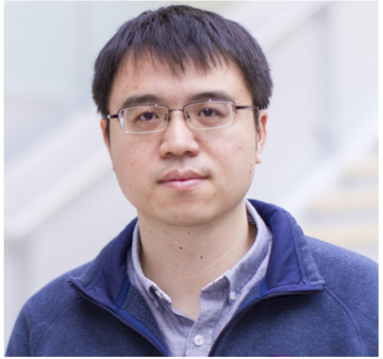



He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. ["Deep residual learning for image recognition."]() CVPR 2016
[slides]()

# Deep Residual Networks

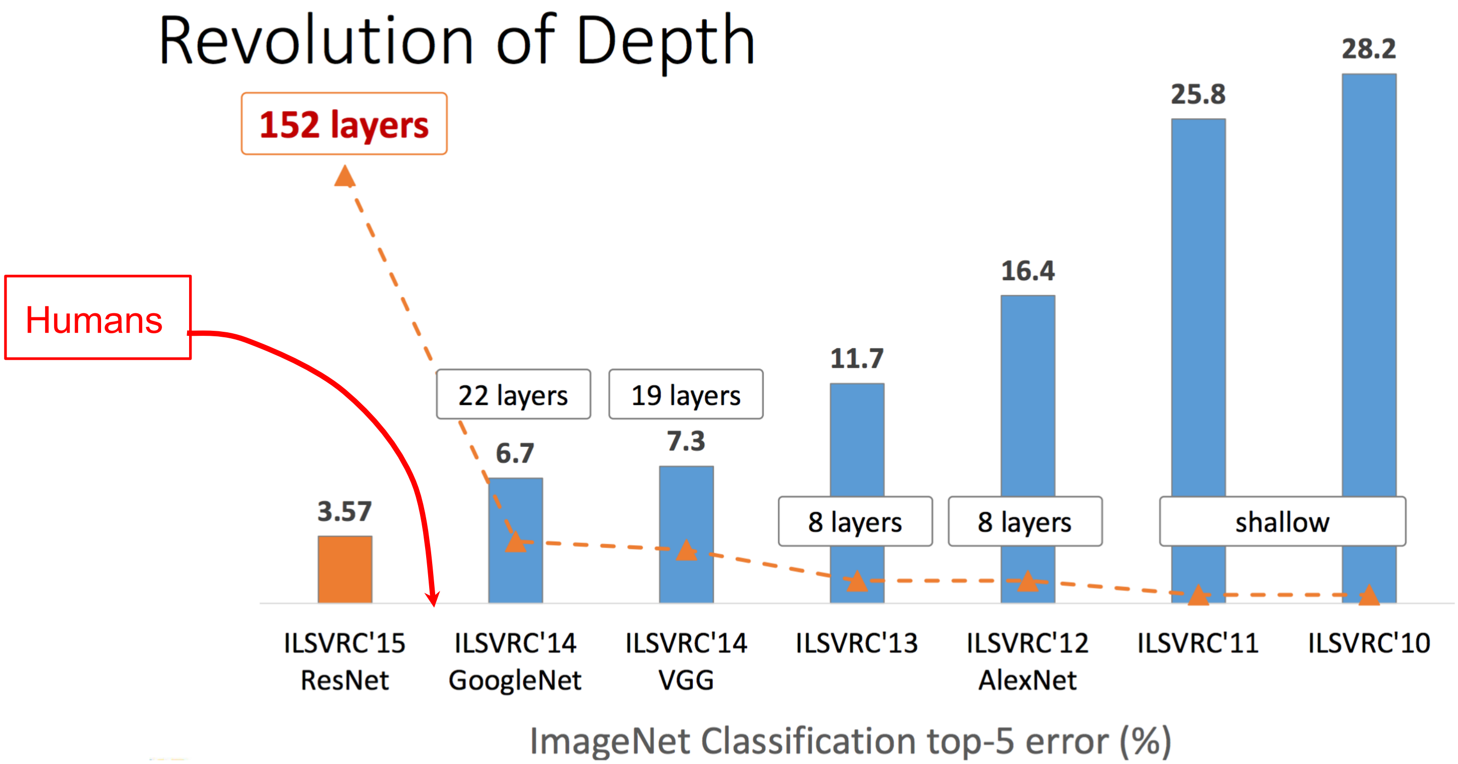


Microsoft Research

3.6% top 5 error…
with 152 layers !!

# Deep Residual Networks




He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. ["Deep residual learning for image recognition."]() CVPR 2016 [slides]()

# Skip connections

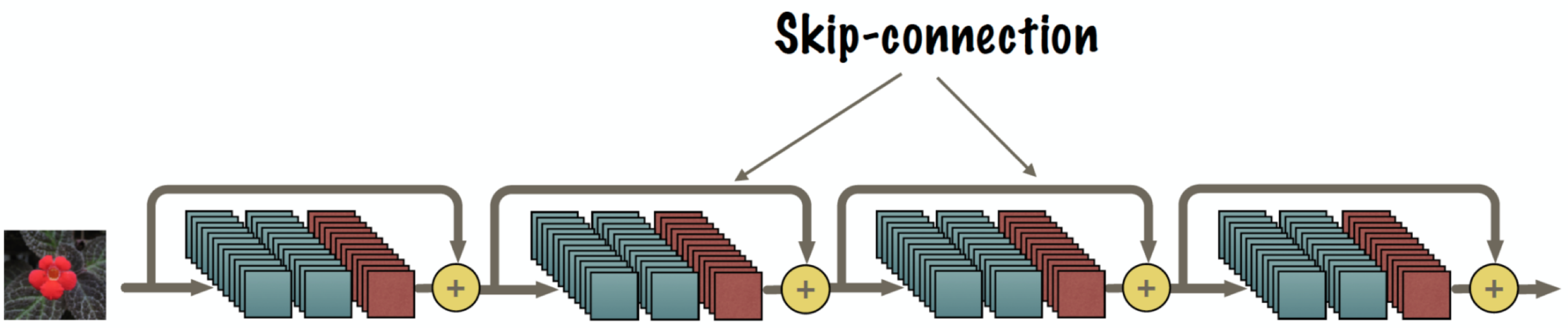
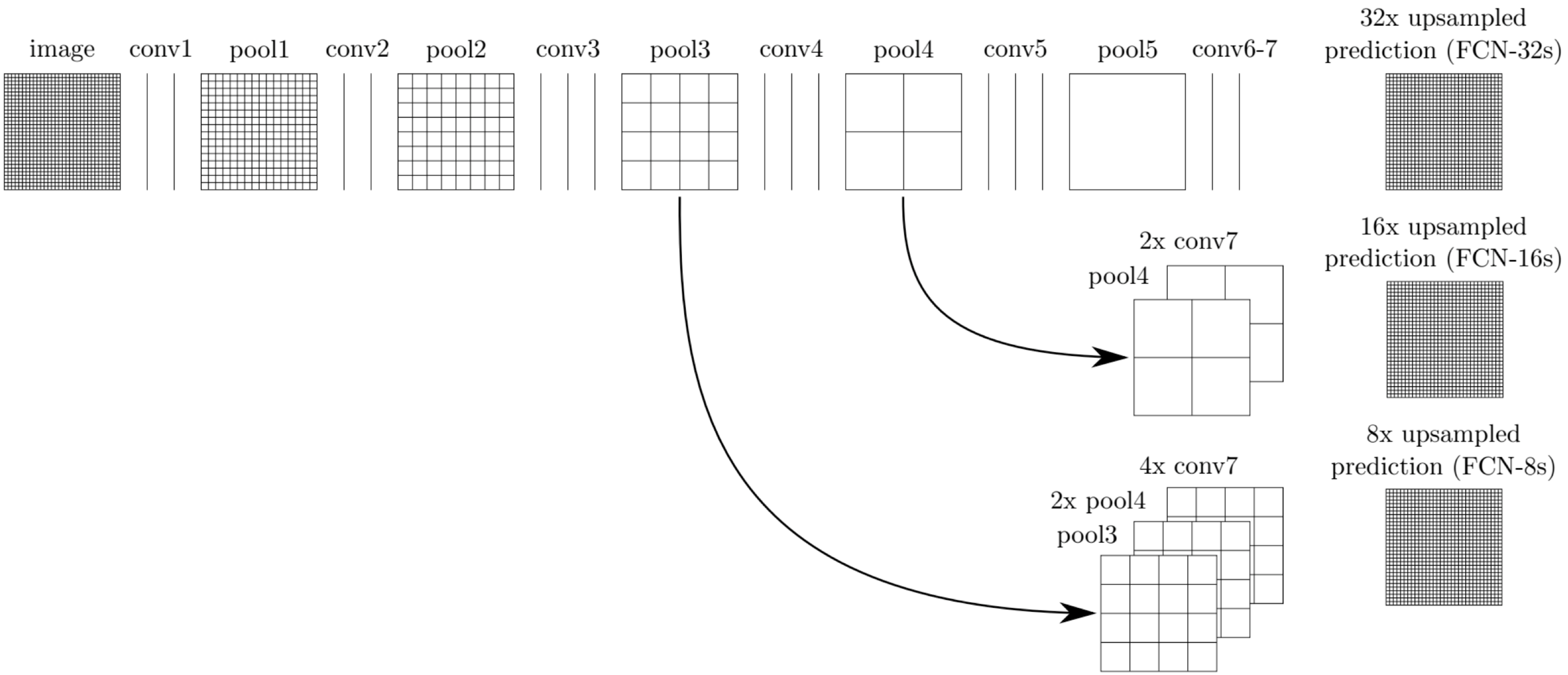


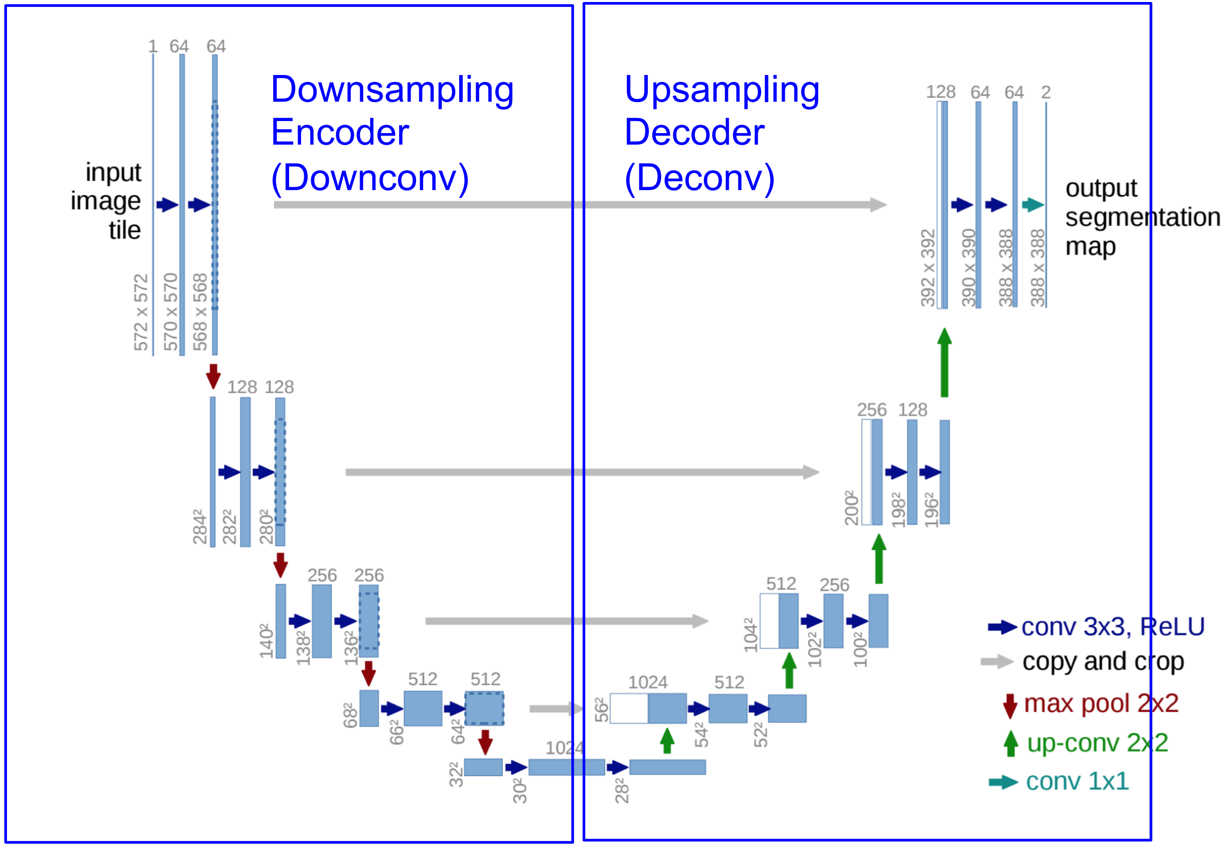

Figure: Kilian Weinberger

# Skip connections



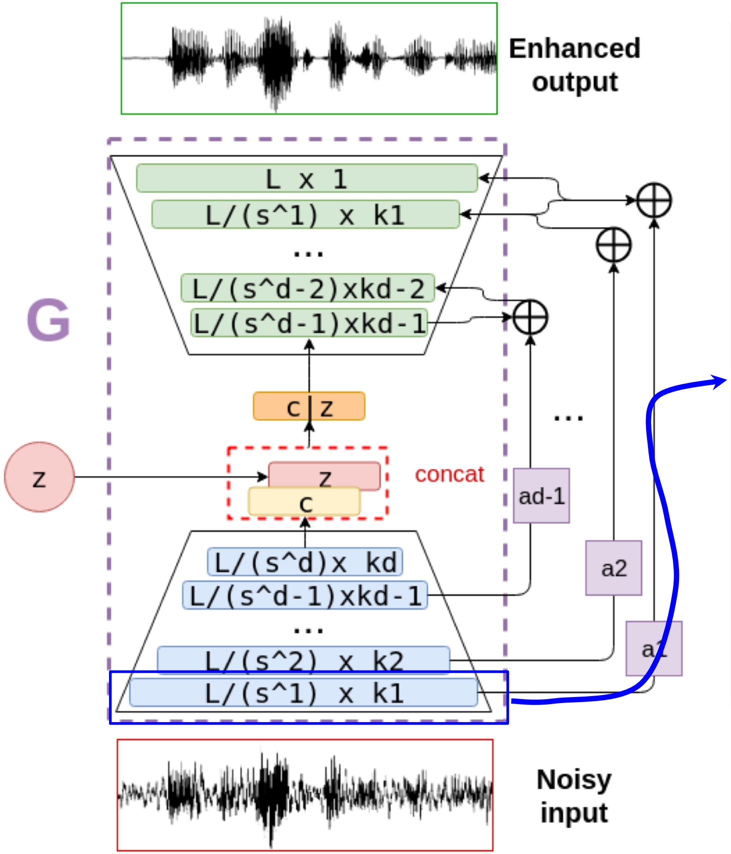

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." CVPR 2015 & PAMI 2016.

# Skip connections: U-Net




Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 234-241. Springer International Publishing, 2015

# Skip connections: SEGAN

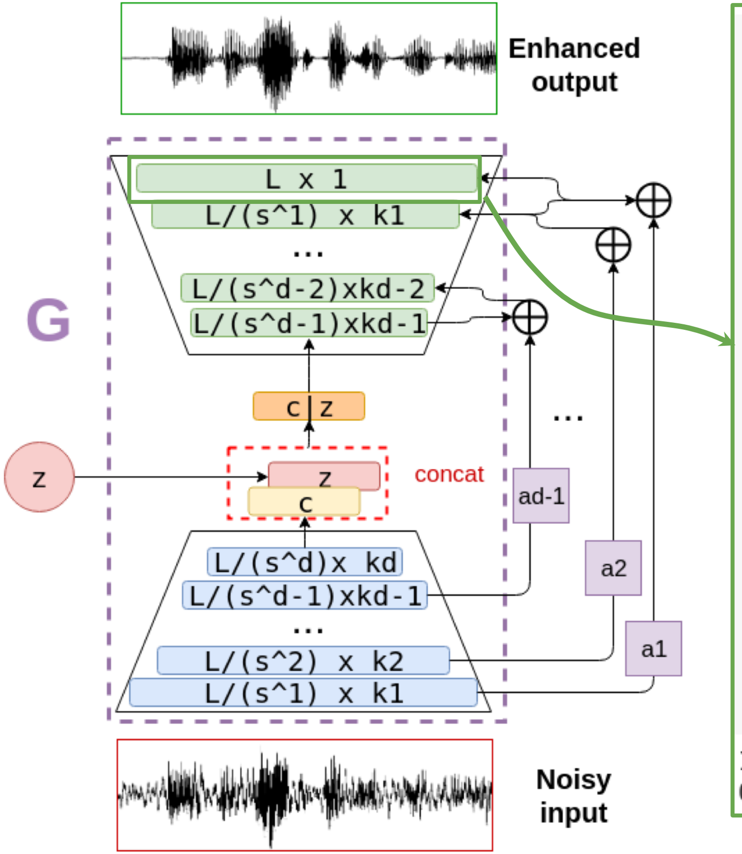



```python
class DownConv1dBlock(nn.Module):

  def __init__(self, ninp, fmap, kwidth, stride):
    super().__init__()
    assert stride > 1, stride
    self.kwidth = kwidth
    self.conv = nn.Conv1d(ninp, fmap, kwidth, stride=stride)
    self.act = nn.ReLU(inplace=True)

  def forward(self, x):
    # calculate padding with stride > 1
    pad_left = self.kwidth // 2 - 1
    pad_right = self.kwidth // 2
    xp = F.pad(x, (pad_left, pad_right))
    y = self.act(self.conv(xp))
    return y

block = DownConv1dBlock(1, 1, 31, 4)
x = torch.randn(1, 1, 4000)
print('Input tensor x size: ', x.size())
y = block(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 1, 4000])
Output tensor y size:  torch.Size([1, 1, 1000])
```

[SEGAN in PyTorch GitHub]

# Skip connections: SEGAN

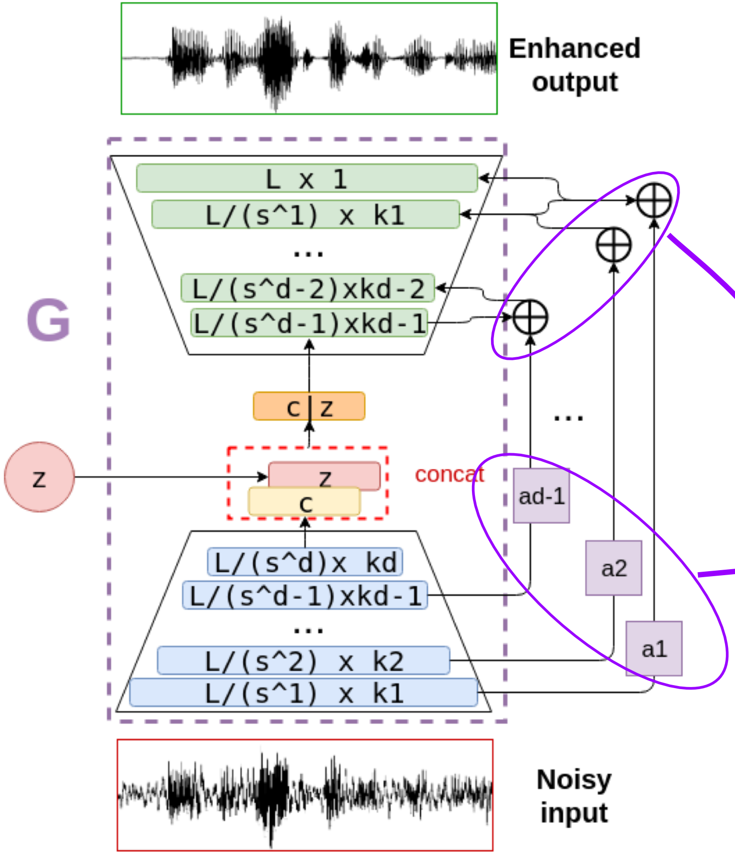



```python
class UpConv1dBlock(nn.Module):

  def __init__(self, ninp, fmap, kwidth, stride, act=True):
    super().__init__()
    assert stride > 1, stride
    self.kwidth = kwidth
    pad = max(0, (stride - kwidth) // -2)
    self.deconv = nn.ConvTranspose1d(ninp, fmap, kwidth,
                                     stride=stride,
                                     padding=pad)
    if act:
      self.act = nn.ReLU(inplace=True)

  def forward(self, x):
    h = self.deconv(x)
    if self.kwidth % 2 != 0:
      # drop last item for shape compatibility with TensorFlow deconvs
      h = h[:, :, :-1]
    if hasattr(self, 'act'):
      y = self.act(h)
    else:
      y = h
    return y

block = UpConv1dBlock(1, 1, 31, 4)
x = torch.randn(1, 1, 1000)
print('Input tensor x size: ', x.size())
y = block(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 1, 1000])
Output tensor y size:  torch.Size([1, 1, 4000])
```

SEGAN in PyTorch GitHub

# Skip connections: SEGAN



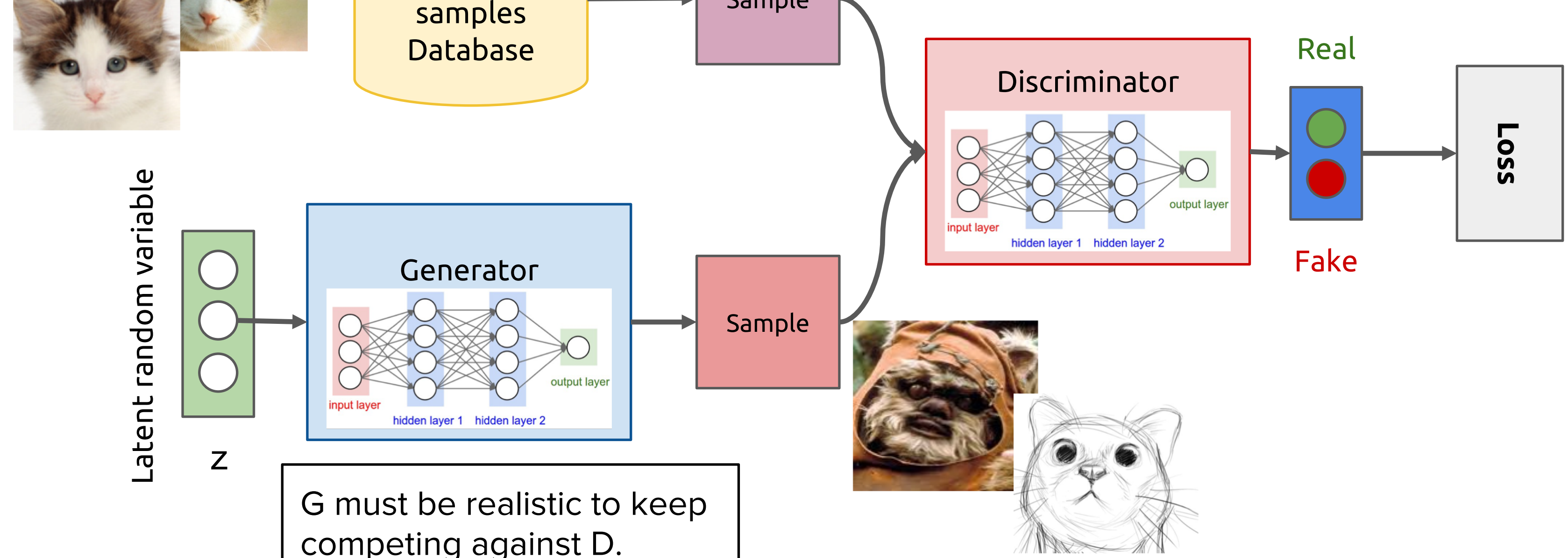

```python
class Conv1dGenerator(nn.Module):

  def __init__(self, enc_fmaps=[64, 128, 256, 512], kwidth=31,
               pooling=4):
    super().__init__()
    self.enc = nn.ModuleList()
    ninp = 1
    for enc_fmap in enc_fmaps:
      self.enc.append(DownConv1dBlock(ninp, enc_fmap, kwidth, pooling))
      ninp = enc_fmap

    self.dec = nn.ModuleList()
    # revert encoder feature maps
    dec_fmaps = enc_fmaps[::-1][1:] + [1]
    act = True
    for di, dec_fmap in enumerate(dec_fmaps, start=1):
      if di >= len(dec_fmaps):
        # last decoder layer has no activation
        act = False
      self.dec.append(UpConv1dBlock(ninp, dec_fmap, kwidth, pooling, act=act))
      ninp = dec_fmap

  def forward(self, x):
    skips = []
    h = x
    for ei, enc_layer in enumerate(self.enc, start=1):
      h = enc_layer(h)
      if ei < len(self.enc):
        skips.append(h)
    # now decode

    for di, dec_layer in enumerate(self.dec, start=1):
      if di > 1:
        # sum skip connection
        skip_h = skips.pop(-1)
        h = h + skip_h
      h = dec_layer(h)
    y = h
    return y

G = Conv1dGenerator()
x = torch.randn(1, 1, 8192)
print('Input tensor x size: ', x.size())
y = G(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 1, 8192])
Output tensor y size:  torch.Size([1, 1, 8192])
```

[SEGAN in PyTorch GitHub]

# Generative + Adversarial

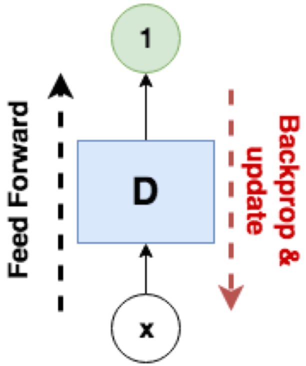
D is a binary classifier, whose objectives are: database images are Real, whereas generated ones are Fake .




G must be realistic to keep competing against D.

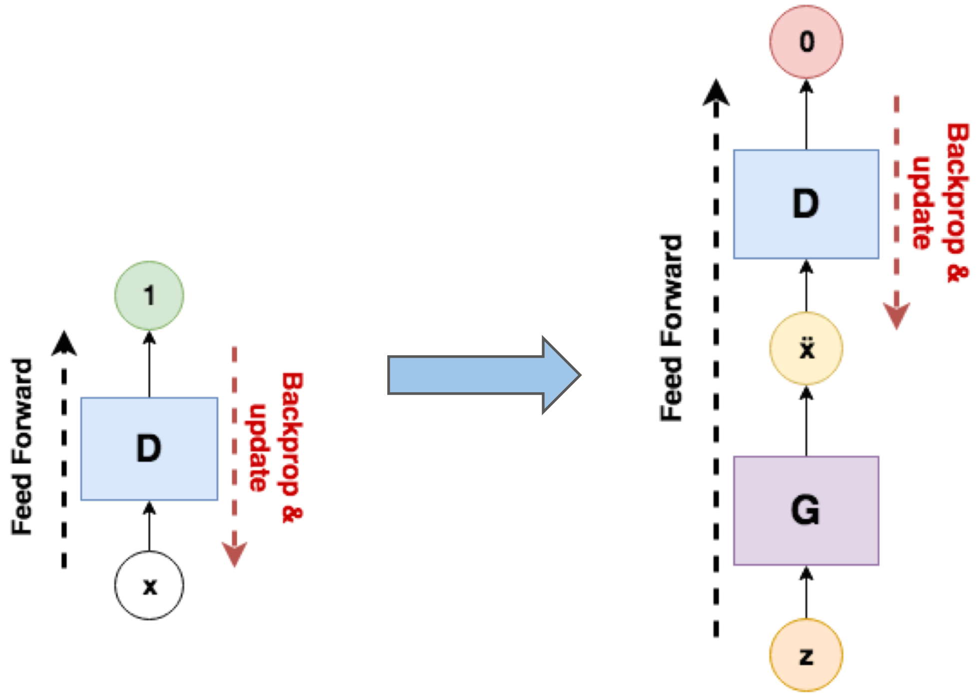
# Adversarial Training (batch update) (1)

- Pick a sample ***x*** from training set
- Show ***x*** to **D** and update weights to output 1 (real)



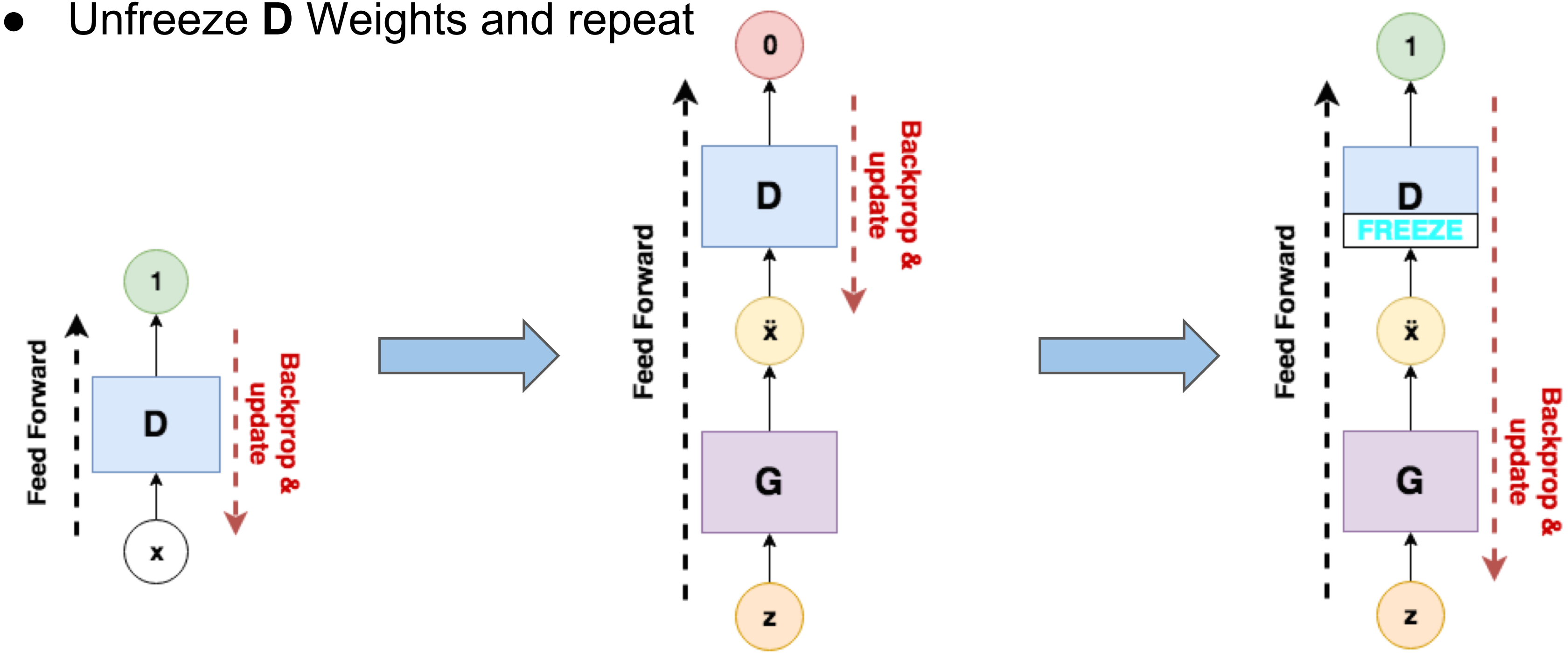
# Adversarial Training (batch update) (2)

- **G** maps sample ***z*** to ***ẍ***
- show ***ẍ*** and update weights to output 0 (fake)



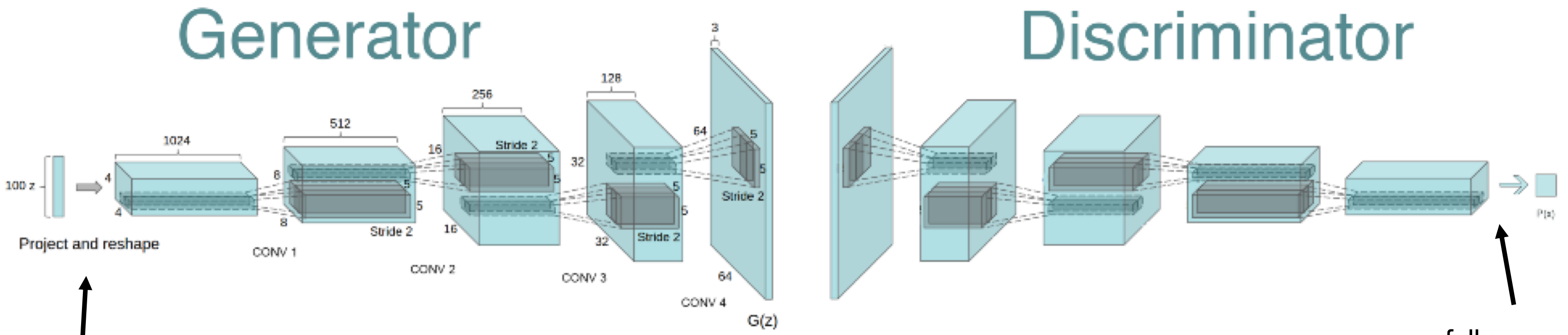
# Adversarial Training (batch update) (3)

- Freeze **D** weights
- Update **G** weights to make **D** output 1 (just **G** weights!)
- Unfreeze **D** Weights and repeat



# Generating images/frames

**Deep Conv. GAN (DCGAN)** effectively generated 64x64 RGB images in a single shot. It is also the base of all image generation GAN architectures.




fully
connected

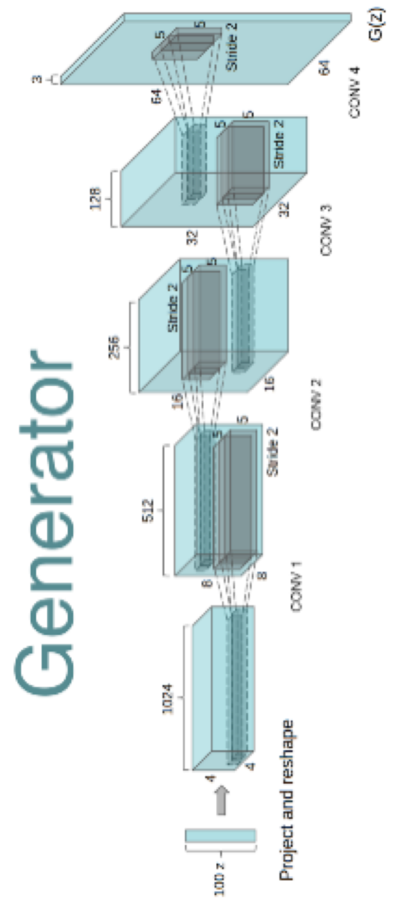
Strided
transposed
conv layers

Conv layers,
no max-pool

fully
connected

([Radford et al. 2015](https://arxiv.org/abs/1511.06434))

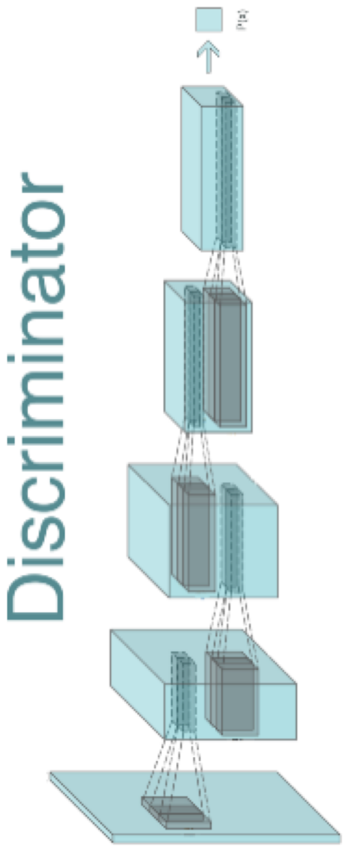
# DCGAN




```python
# from https://github.com/pytorch/examples/blob/master/dcgan/main.py
class Generator(nn.Module):
    def __init__(self, nc=3):
        super().__init__()
        nz = 100
        ngf = 64
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2,     ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(    ngf,      nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
      return self.main(input)

z = torch.randn(1, 100, 1, 1)
print('Input tensor z size: ', z.size())
G = Generator()
x = G(z)
print('Output tensor x size: ', x.size())
```

```
Input tensor z size:  torch.Size([1, 100, 1, 1])
Output tensor x size:  torch.Size([1, 3, 64, 64])
```

# DCGAN




```python
class Discriminator(nn.Module):
    def __init__(self, nc=3):
        super(Discriminator, self).__init__()
        ndf = 64
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)


x = torch.randn(1, 3, 64, 64)
print('Input tensor x size: ', x.size())
D = Discriminator()
y = D(x)
print('Output tensor y size: ', y.size())
```

```
Input tensor x size:  torch.Size([1, 3, 64, 64])
Output tensor y size:  torch.Size([1, 1, 1, 1])
```

# Conclusions

# Conclusions

- We have reviewed basic architectures (most fundamental layers), their relations against each other, and their implementations.
    - We made some implementations that change some basic block properties/efficiency (i.e. Embedding, causal convolution, etc.).
- We have reviewed some advanced architectures built on top of the basic ones.
    - Hybrid combinations of basic blocks with QRNN
    - Auto-Encoder structures to do unsupervised learning and generative modeling
    - Deep convolutional classifiers, their evolutions in ImageNet challenge and residual connections.
    - Skip connections and their usage in deep architectures and U-Net structures.
    - DCGAN has been revisited, breaking down its generator and discriminator structure.
- All these shown models serve as templates for many typical applications (at least as starting point)
- Combining the mentioned structures often boosts results, but often with a data hungry trade-off as complexity grows.