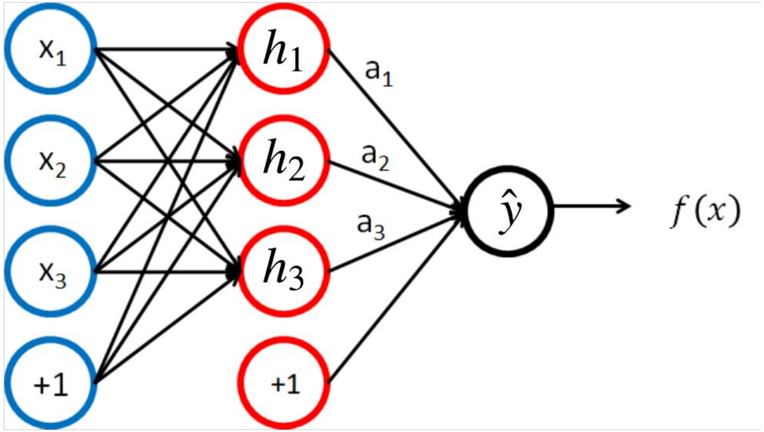
Deep Learning
CNN - RNN - Pytorch

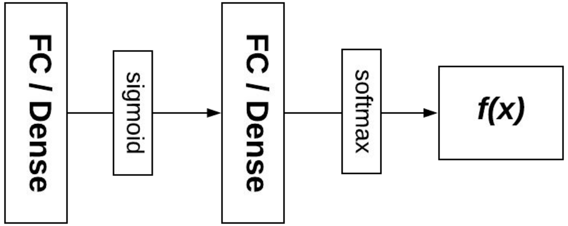
Christodoulos Benetatos
2019

MLP - Pytorch



$$\sigma\left(\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

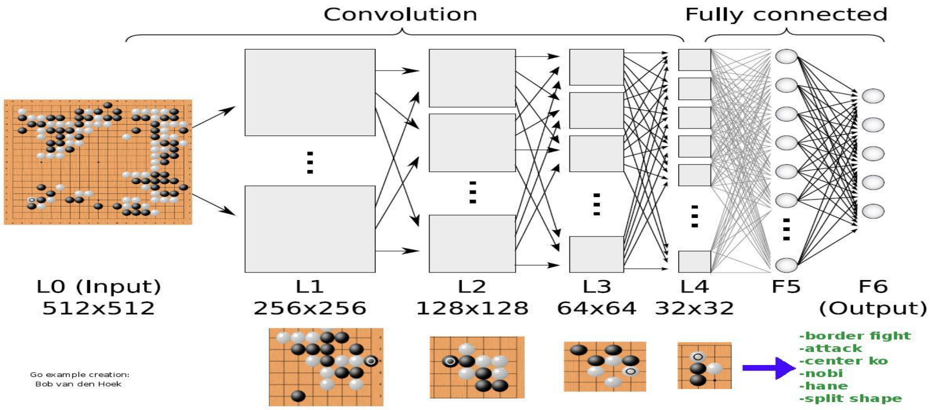
$$softmax\left(\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} + \begin{bmatrix} b_a \end{bmatrix}\right) = \begin{bmatrix} \hat{y} \end{bmatrix} = f(x)$$

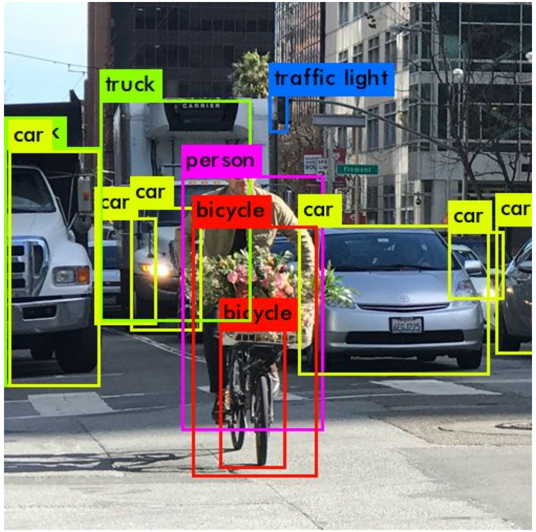



```
x = torch.rand(3,1)
fc1 = nn.Linear(in_features = 3, out_features = 3)
fc2 = nn.Linear(in_features = 3, out_features = 1)
sigma = torch.sigmoid
softmax = torch.softmax
h = sigma(fc1(x))
y = softmax(fc2(h), dim=0)
```

Convolutional Neural Nets

- 2012 : AlexNet achieves state-of-the-art results on ImageNet
- 2013 : DQN beats humans on 3 Atari games
- 2014 : GaussianFace surpasses humans on face detection
- 2015 : PReLU-Net surpasses humans on ImageNet
- 2016 : AlphaGo beats Lee Sedol on Go game
- 2016 : WaveNet synthesizes high-fidelity speech





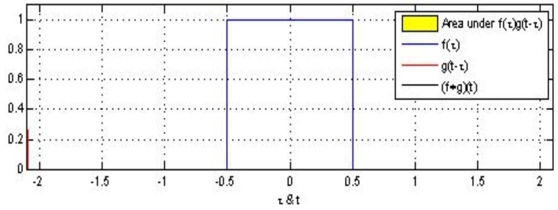
Convolution

1D convolution

Input Signal 1D

Window/Kernel/Filter 1D

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$



2D convolution

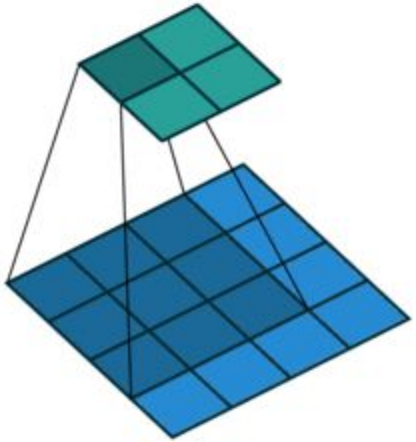
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

2D cross-correlation

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Input Signal 2D
(i.e Image)

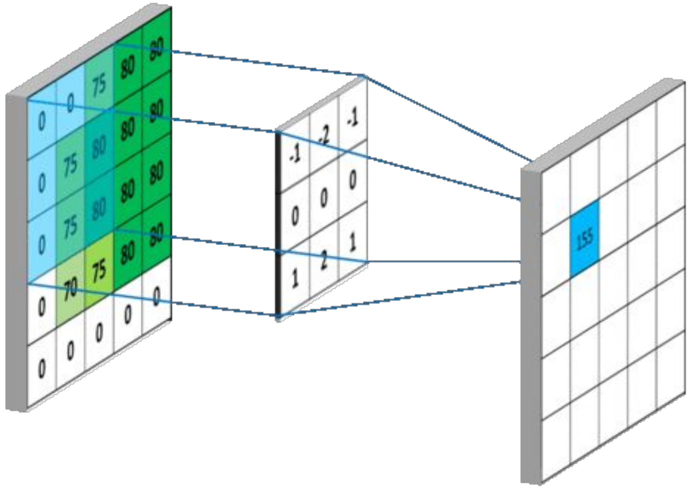
Filter/Kernel 2D



Why Convolution ?

1. Sparse Connections
    - Output units are calculated from a small neighborhood of input units

2. Parameter Sharing
    - Each member (weight) of the kernel is used at every position of the input



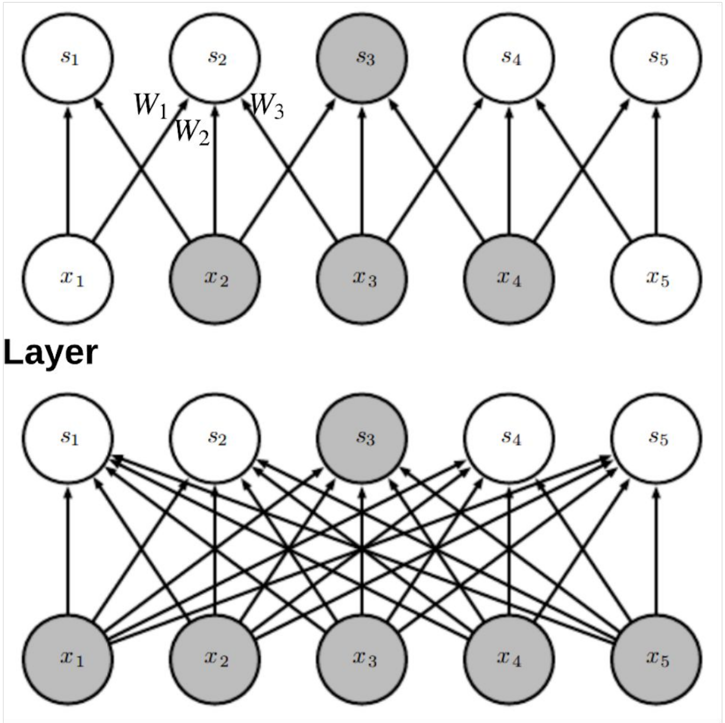
Why Convolution ?

Convolution Layer

$$\begin{bmatrix} W_2 & W_3 & 0 & 0 & 0 \\ W_1 & W_2 & W_3 & 0 & 0 \\ 0 & W_1 & W_2 & W_3 & 0 \\ 0 & 0 & W_1 & W_2 & W_3 \\ 0 & 0 & 0 & W_1 & W_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}$$

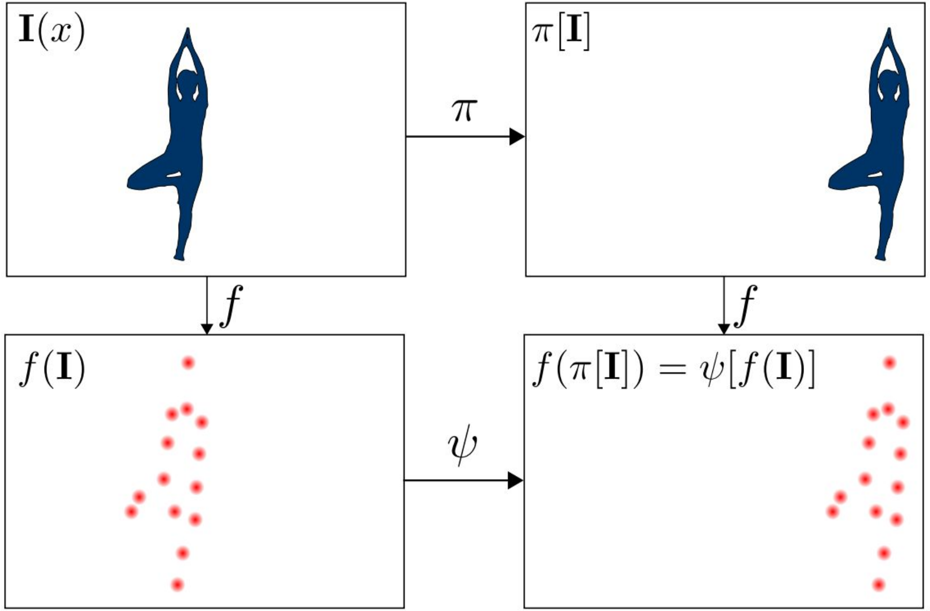
Fully Connected Layer

$$\begin{bmatrix} W_{00} & W_{01} & W_{02} & W_{03} & W_{04} \\ W_{10} & W_{11} & W_{12} & W_{13} & W_{14} \\ W_{20} & W_{21} & W_{22} & W_{23} & W_{24} \\ W_{30} & W_{31} & W_{32} & W_{33} & W_{34} \\ W_{40} & W_{41} & W_{42} & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}$$



Why Convolution ?

3. Equivariant to translation
   - Shifting input by x, results in a shift in the output feature map by x.
     f(g(x)) = g(f(x))

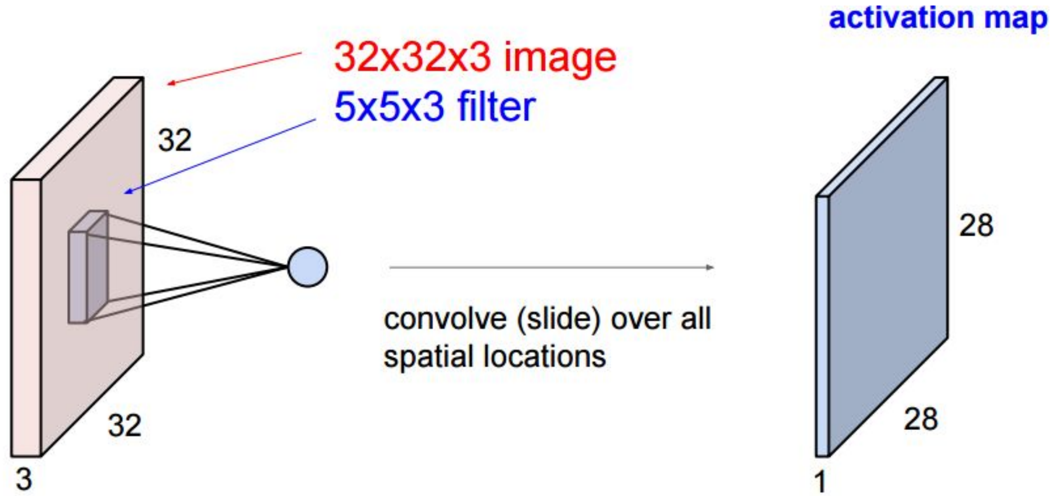


http://visual.cs.ucl.ac.uk/pubs/harmonicNets/pdfs/worrallEtAl2017.pdf

Convolution Layer

Hyperparameters : **channel_in**, **channel_out**, **kernel_size, zero_pad**
Input : (H_in, W_in, channel_in)
Output : (H_out, W_out, channel_out)

```
inp # (32, 32, 3)
conv = nn.Conv2d(in_channels = 3,
                 out_channels = 1,
                 kernel_size = 5,
                 padding=0)
out = conv(inp) # (28, 28, 1)
```

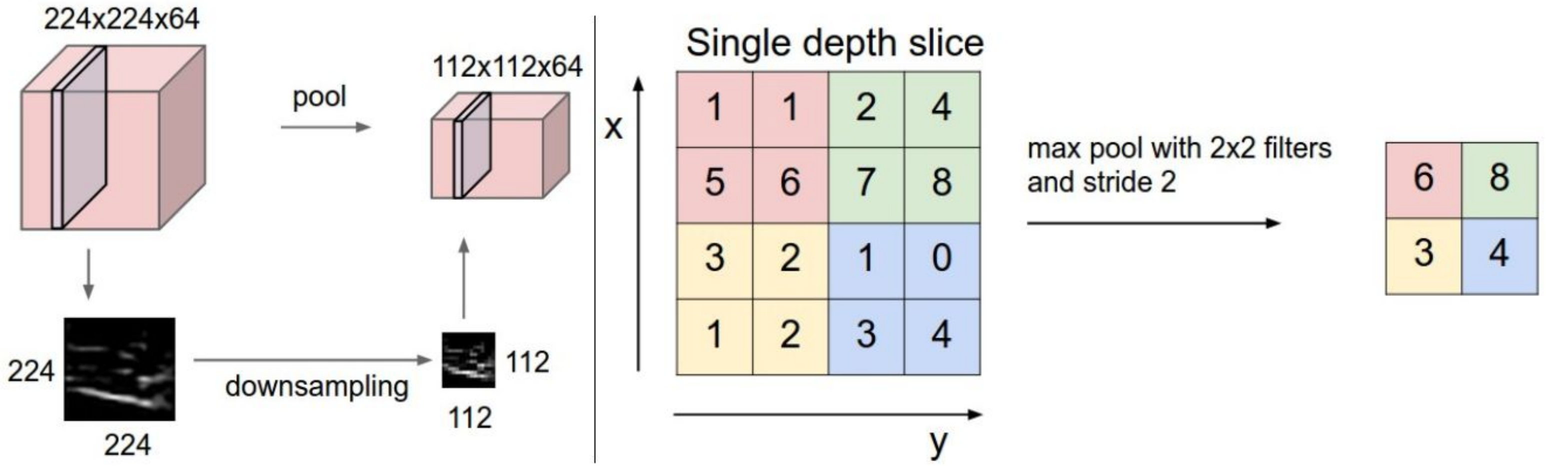


https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolutional_neural_networks.html

Pooling Layers

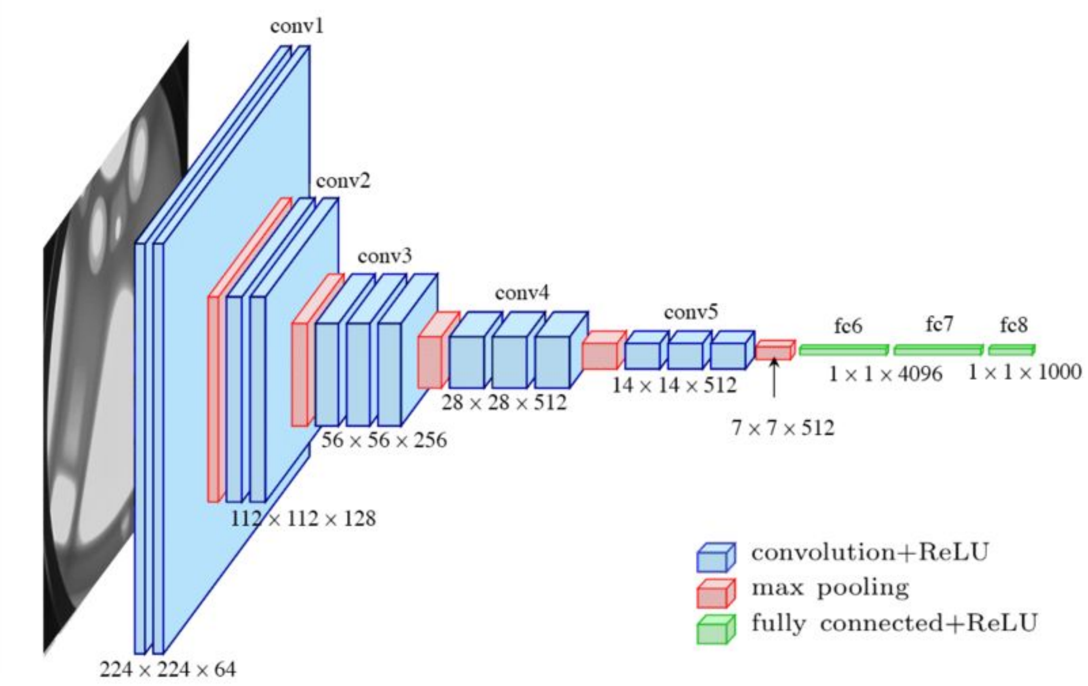
- Reduce spatial size
- Reduce parameters

```
inp # (224, 224, 64)
maxPool = nn.MaxPool2d(kernel = 2)
out = maxPool(inp) # (112, 112, 64)
```



CNN Architectures

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC



224 x 224 x 3

CNN for Audio

- Apply 1D convolution on audio samples (Wavenet)
- Audio → Spectrogram → Treat spectrogram as an image

Applications :

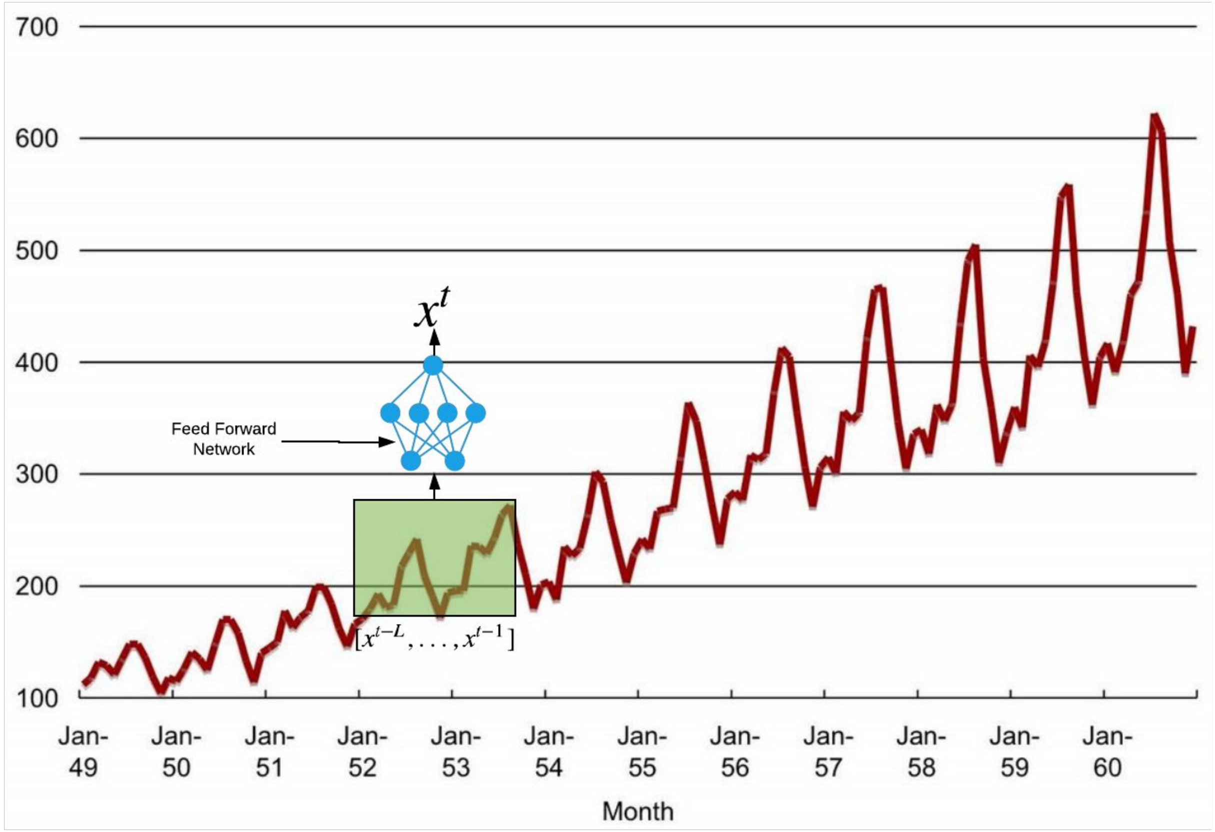
- Classification/Identification: sound, genre, instrument, speaker, etc.
- Source Separation: mask prediction
- Generation: predict the next audio sample

Disadvantages:

- In images, neighbor pixels belong to the same object, not the same for spectrograms.
- CNNs are applied in magnitude, and not phase
- CNNs do not exploit the temporal information

Feed forward NNs on Sequential Data

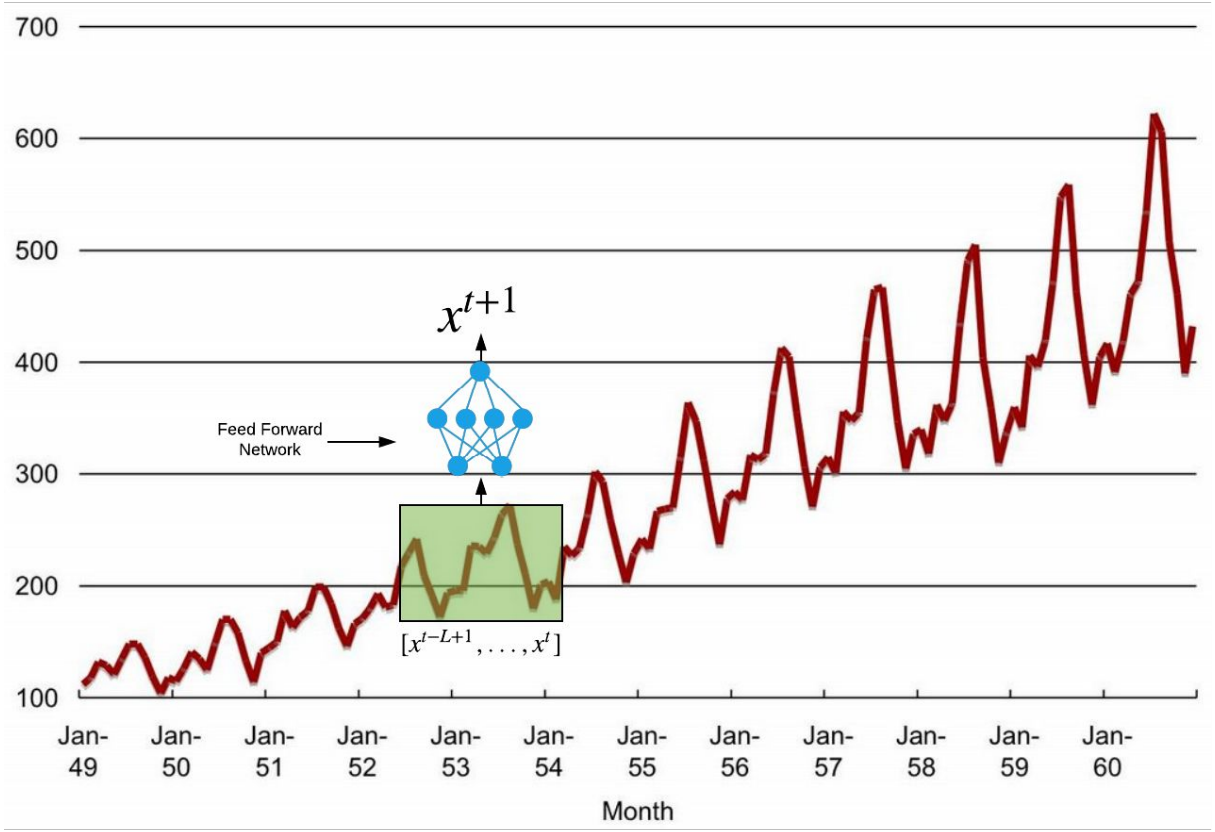
- Limited Memory
- Fixed window size L
- Increasing L → Parameters increase fast



Time Series Analysis: Forecasting and Control by Box and Jenkins (1976)

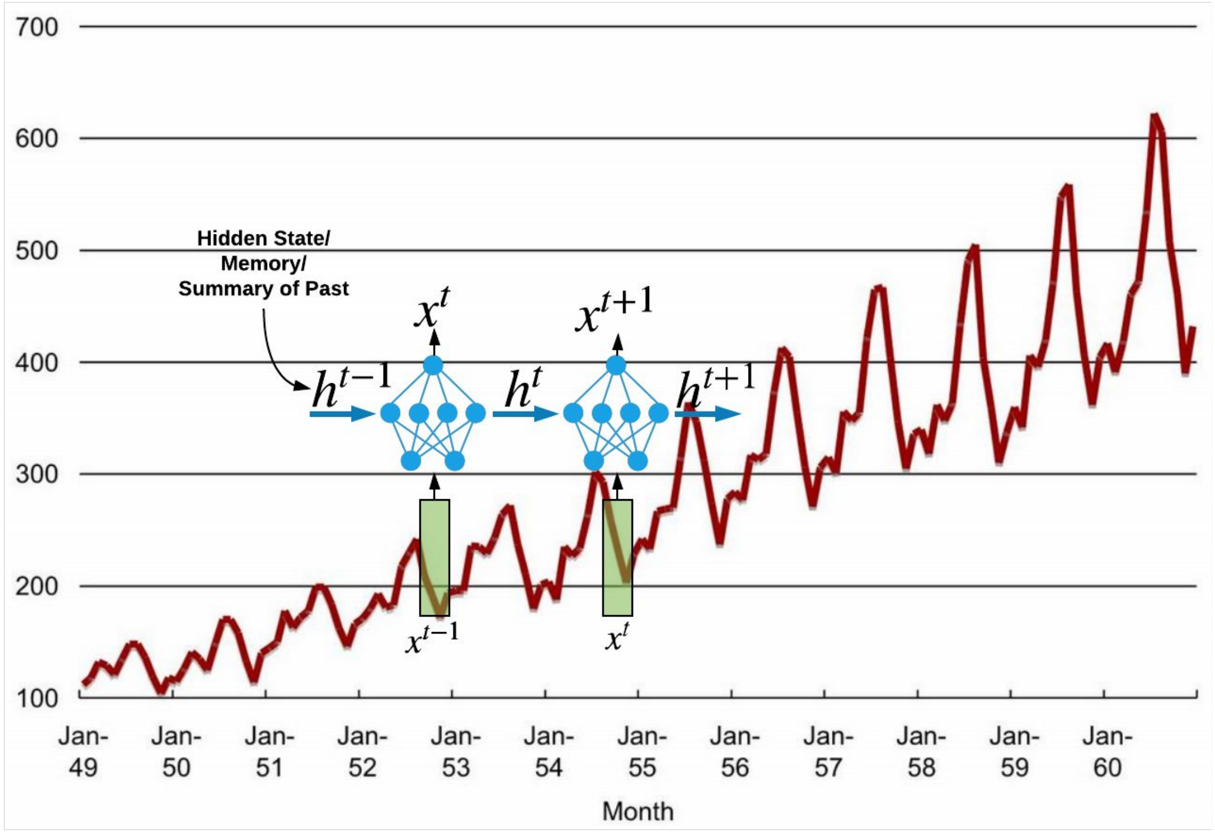
Feed forward NNs on Sequential Data

- Limited Memory
- Fixed window size L
- Increasing L → Parameters increase fast

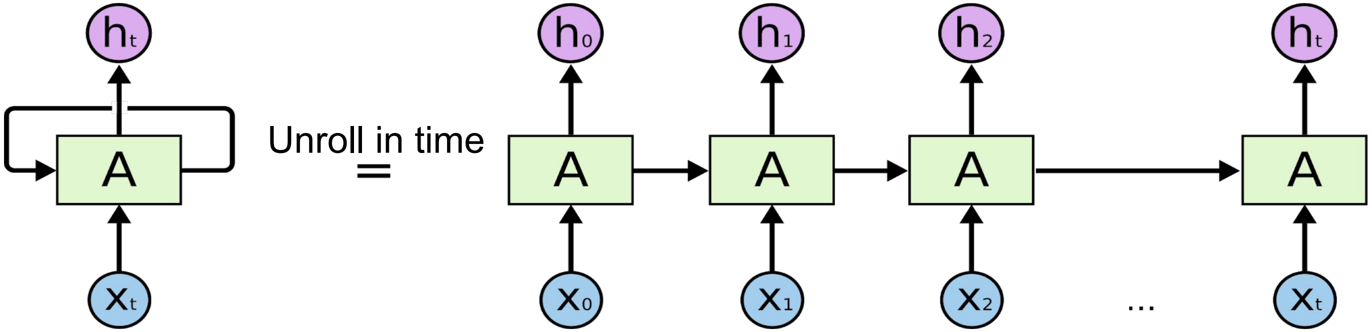


Recurrent NNs on Sequential Data

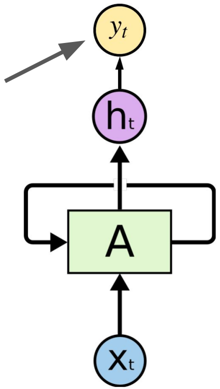
- Unlimited Memory (in theory)
- Variable input length
- Increasing L → Parameters remain the same



RNN



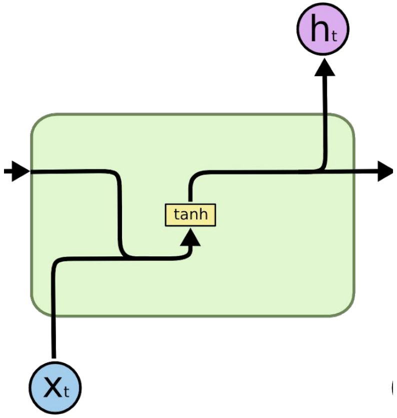
At some timesteps we may want to generate an output



$$h_t = f_W(h_{t-1}, x_t)$$

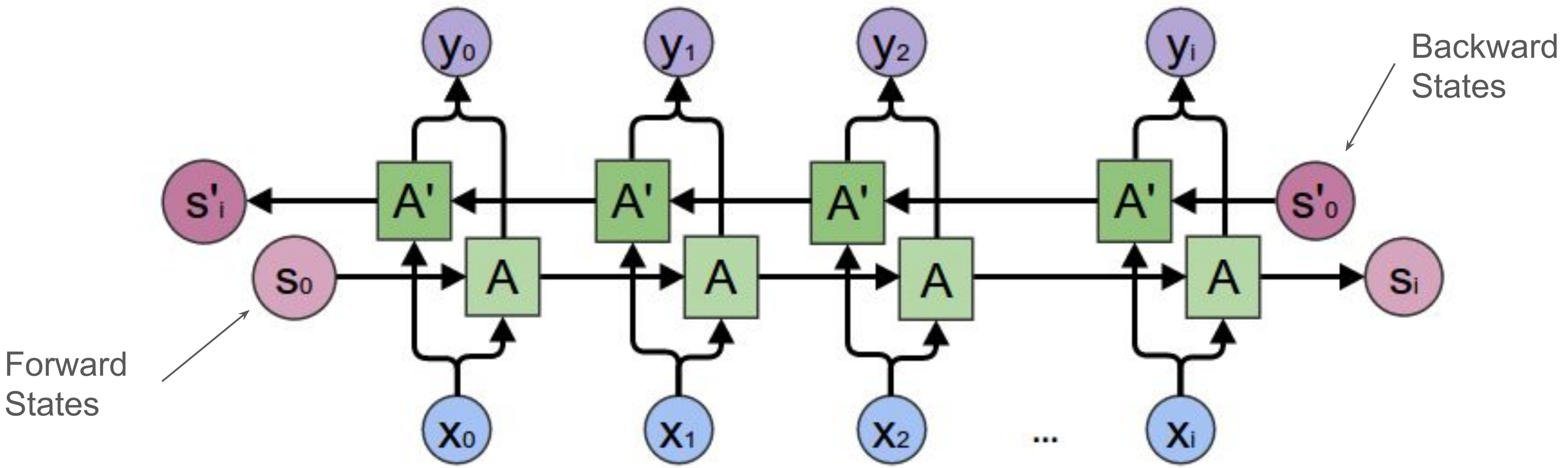
$$h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$



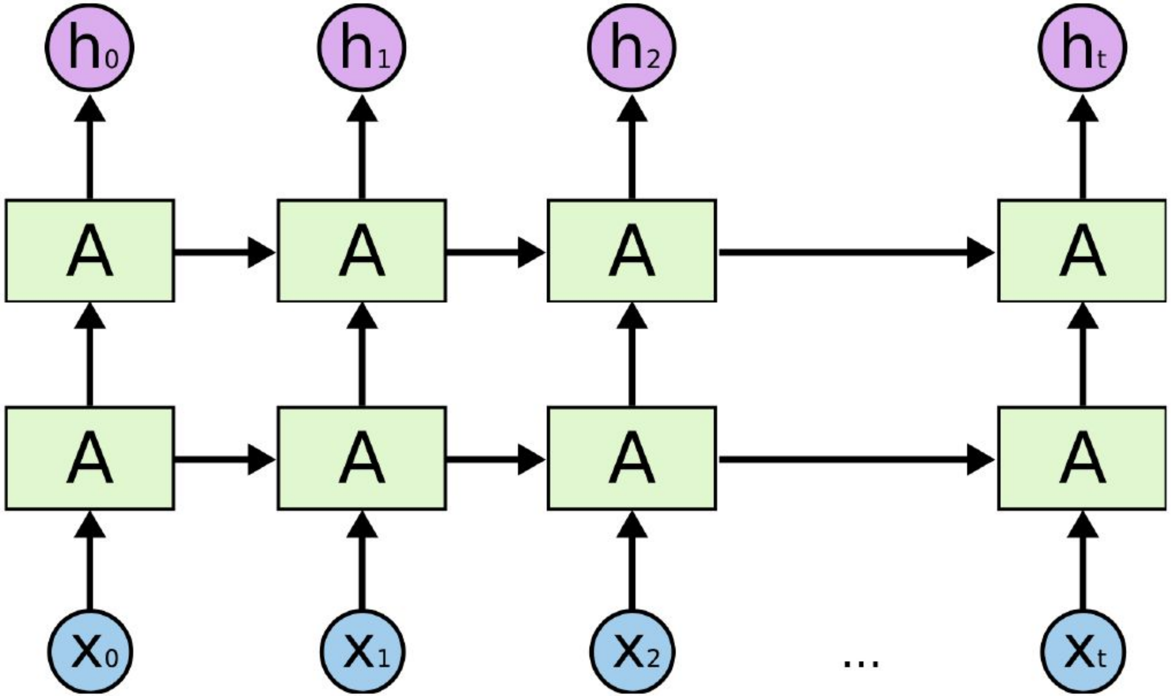
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Bidirectional RNNs



http://colah.github.io/posts/2015-09-NN-Types-FP/img/RNN-bidirectional.png

Stacked RNNs



BackPropagation Through Time

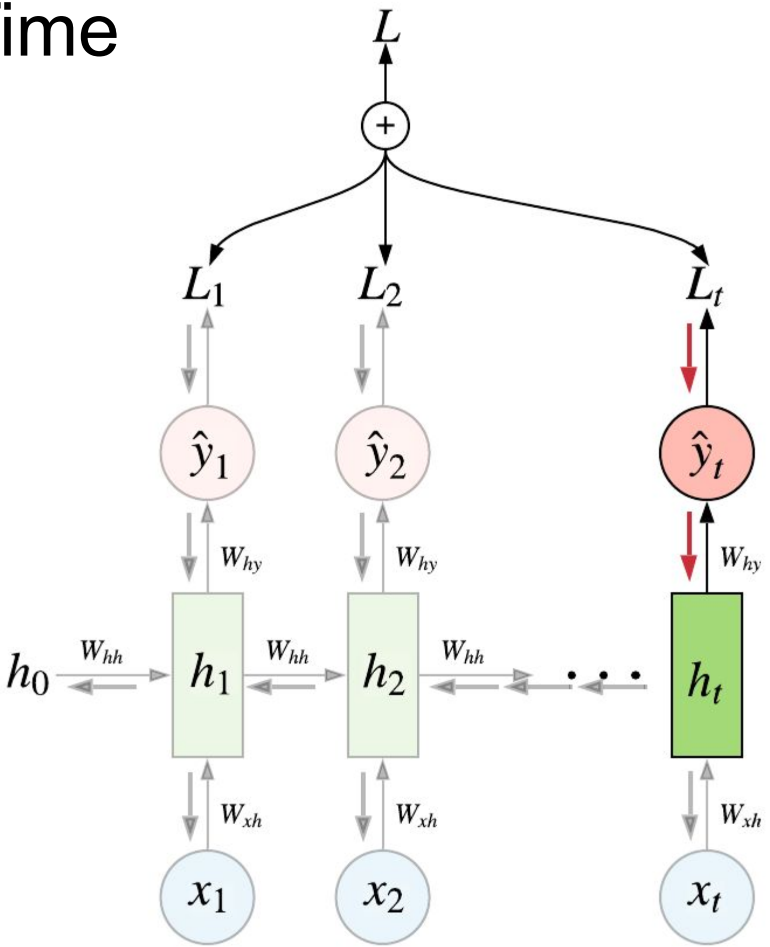
- Same as regular backpropagation → repeatedly apply chain rule
- For W_{hy}, we propagate on the vertical axis

$$\frac{\partial L}{\partial W_{hy}} = \sum_{i=0}^{t} \frac{\partial L_i}{\partial W_{hy}}$$

$$\frac{\partial L_t}{\partial W_{hy}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{W_{hy}}$$

$$\hat{y}_t = W_{hy} h_t$$

Easy to calc



BackPropagation Through Time

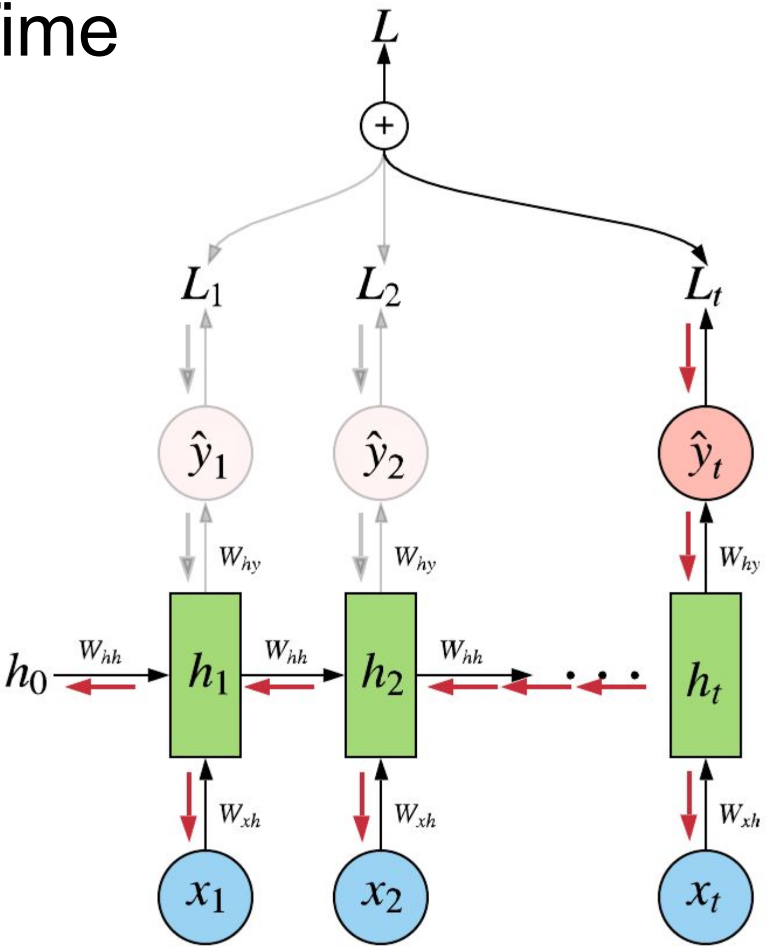
- Same as regular backpropagation → repeatedly apply chain rule
- For W_{hh} and W_{xh}, we propagate On the horizontal time axis

$$\frac{\partial L}{\partial W_{hh}} = \sum_{i=0}^{t} \frac{\partial L_i}{\partial W_{hh}}$$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

$$h_t = tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

It also depends on W_{hh}

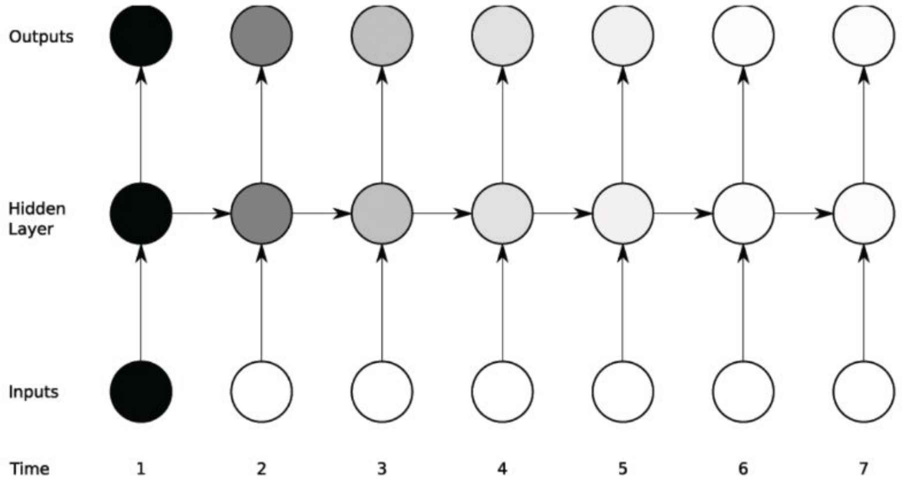


Vanishing - Exploding Gradients

- Computing gradient for the first timestep Includes multiple factors of W_{hh} matrix for each timestep.

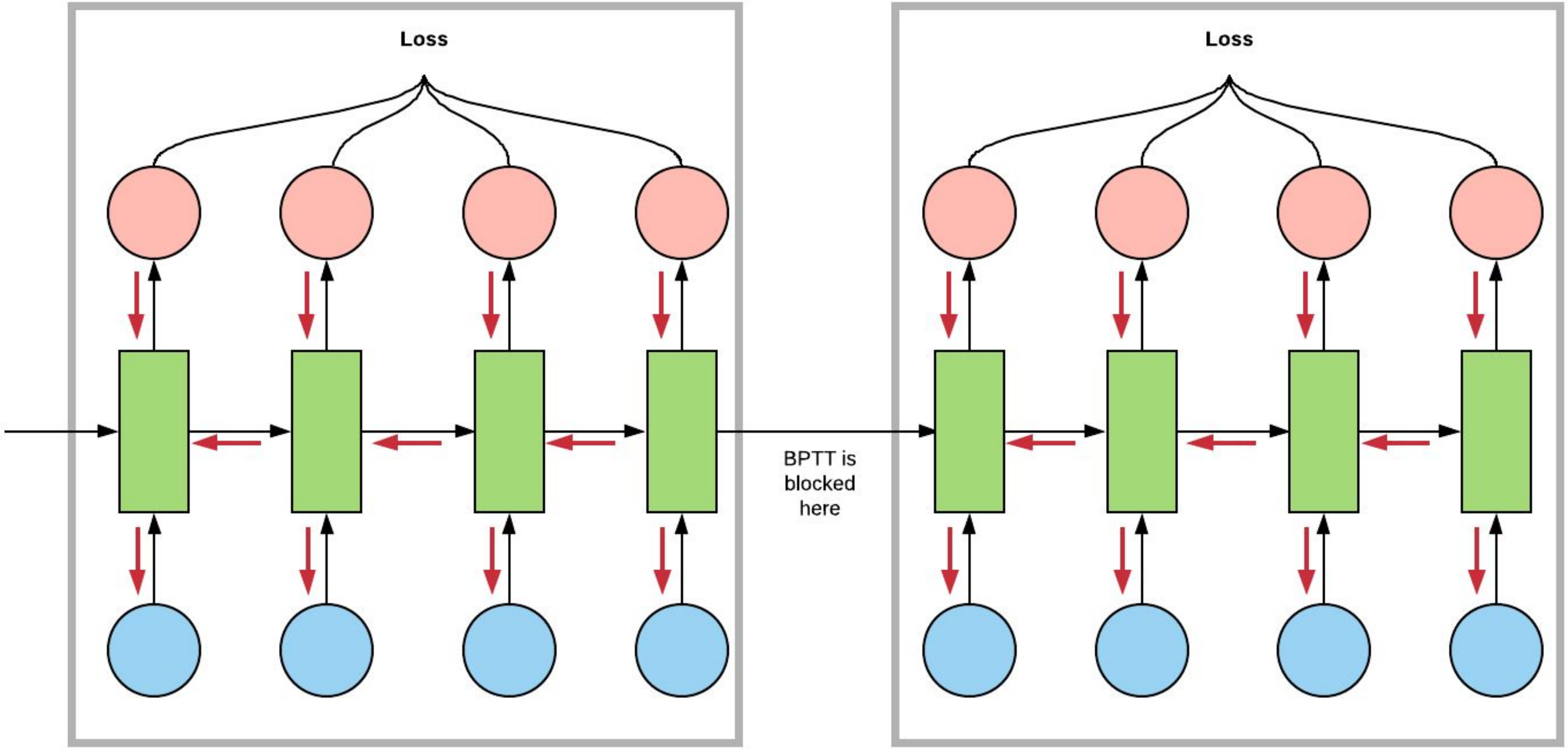
- norm(W_{hh})>1 → Exploding Gradient
  - Clip gradients

- norm(W_{hh})<1 → Vanishing Gradient
  - Truncated BPTT
  - Gated architectures

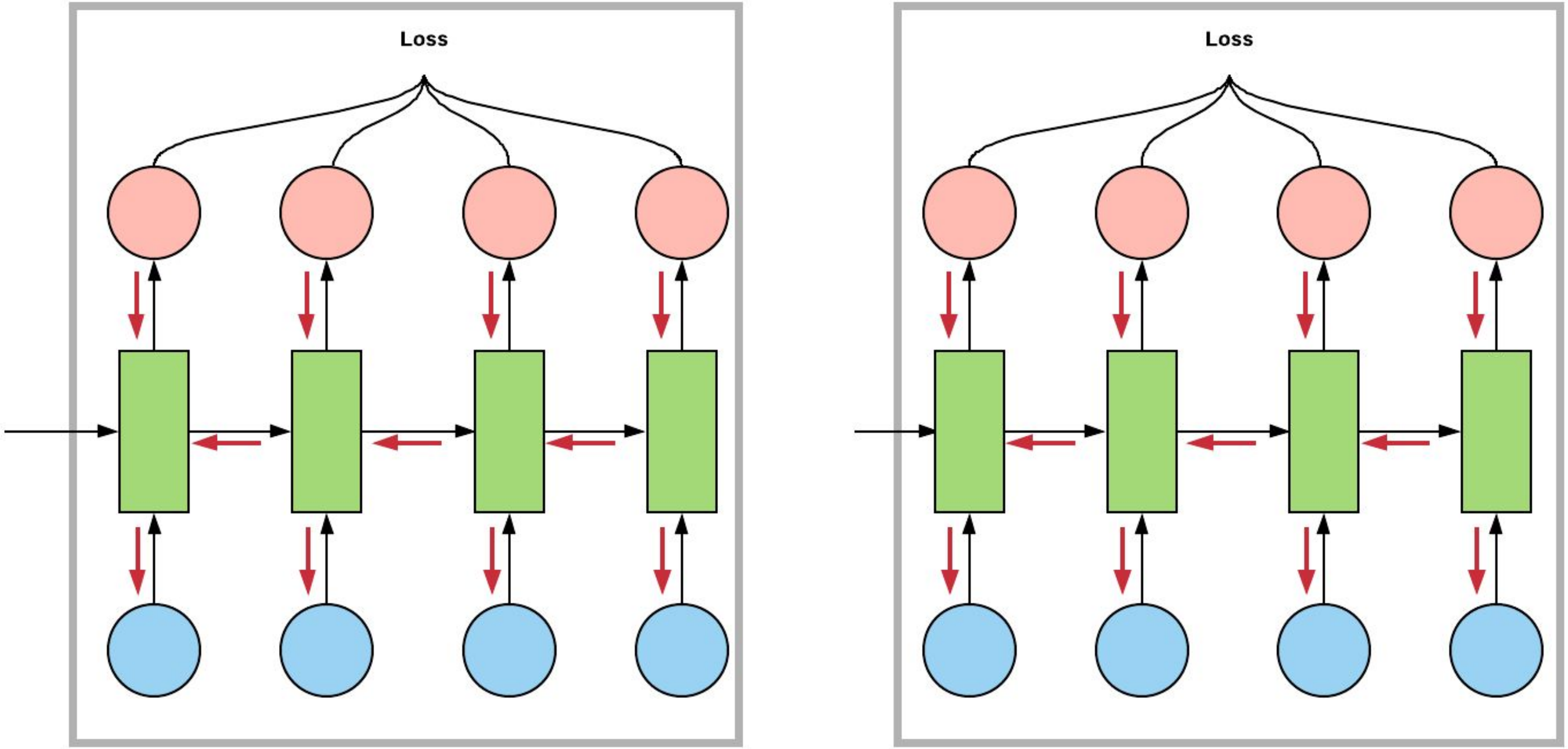


Darkness indicates the influence of input at time 1
Figure from [Graves, 2008]

Truncated BPTT (stateful)



Truncated BPTT (stateless)

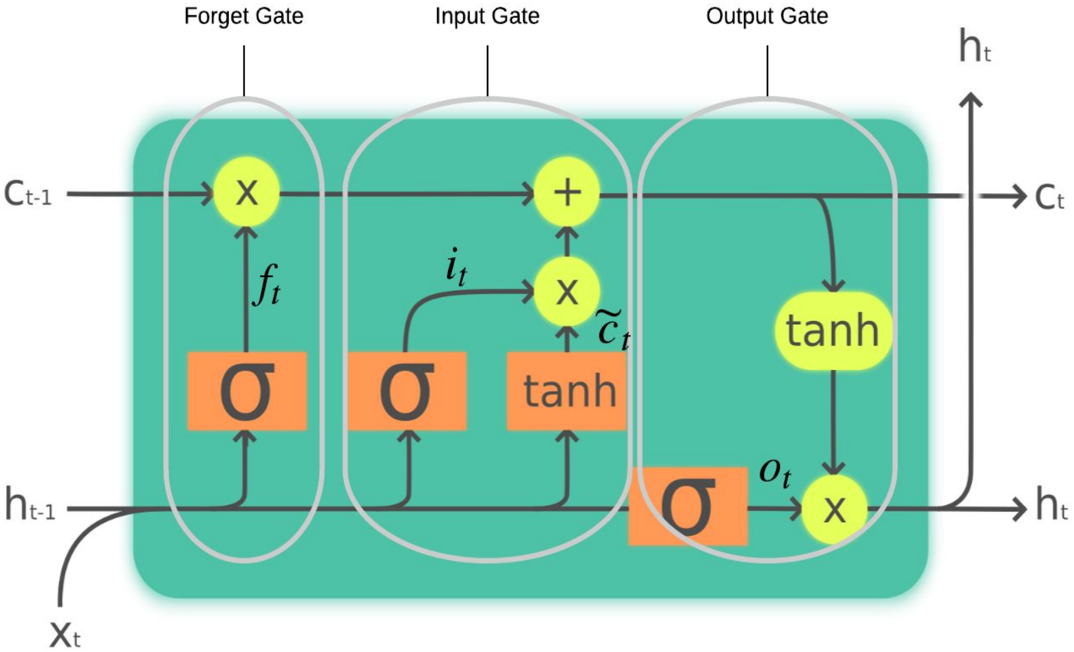


Gated Architectures - LSTM

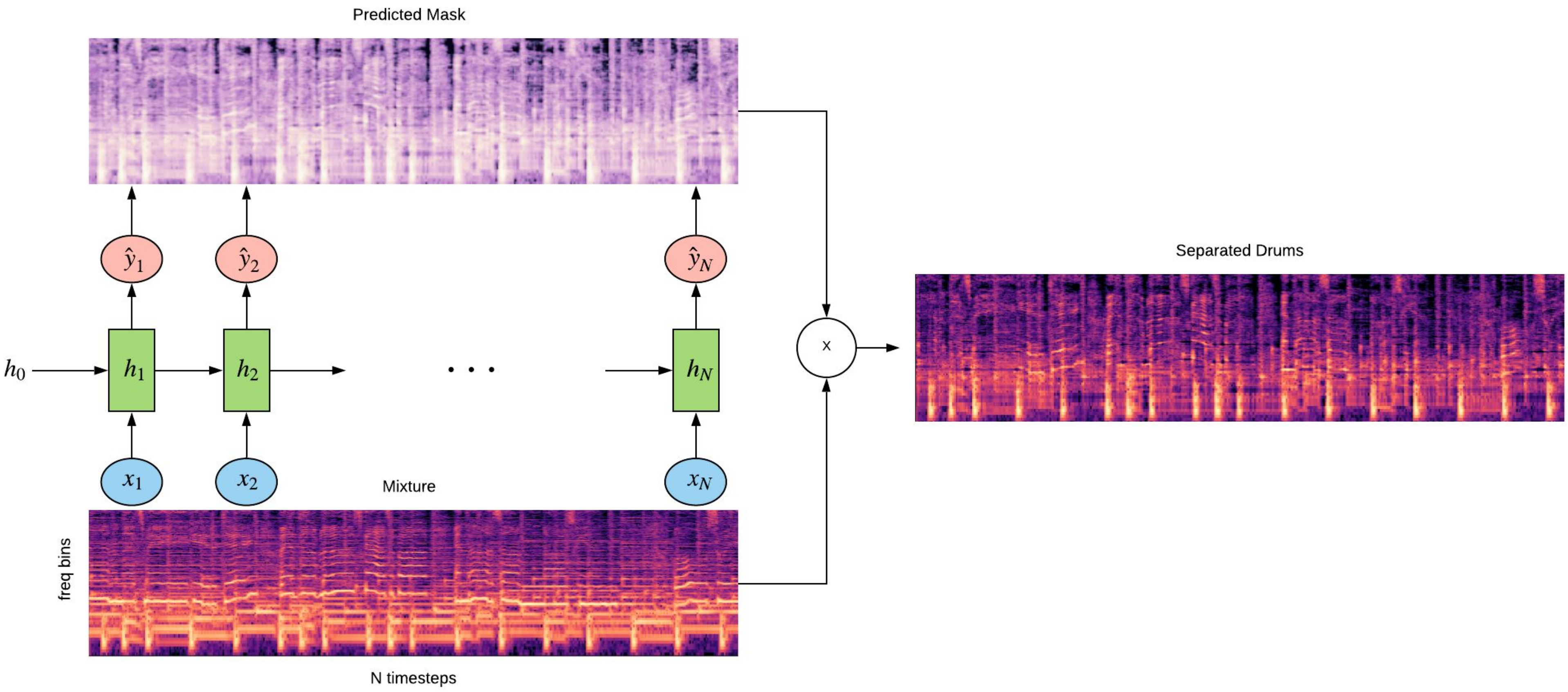
- Idea : Allow gradients to also flow unchanged.
- Cell state is the internal memory
- Three Gates perform delete/write/read operations on memory

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$
$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$
$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$\tilde{c}_t = tanh(w_c[h_{t-1}, x_t] + b_c)$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$
$$h_t = o_t * tanh(c^t)$$



Application : Source Separation



LSTM - Pytorch

CLASS `torch.nn.LSTMCell(input_size, hidden_size, bias=True)`

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`

Inputs: input, (h_0, c_0)

- **input** of shape (*batch, input_size*): tensor containing input features
- **h_0** of shape (*batch, hidden_size*): tensor containing the initial hidden state for each element in the batch.
- **c_0** of shape (*batch, hidden_size*): tensor containing the initial cell state for each element in the batch.

  If (*h_0, c_0*) is not provided, both **h_0** and **c_0** default to zero.

Outputs: (h_1, c_1)

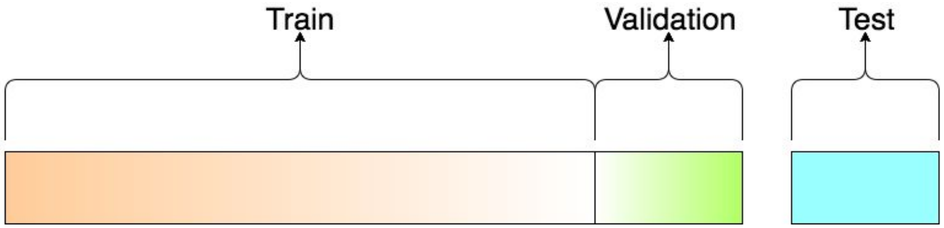
- **h_1** of shape (*batch, hidden_size*): tensor containing the next hidden state for each element in the batch
- **c_1** of shape (*batch, hidden_size*): tensor containing the next cell state for each element in the batch

```
rnn = nn.LSTMCell(10, 20)
# inp =seqLen,batch, inputSize
inp = torch.randn(6, 3, 10)
#hidden State
hx = torch.randn(3, 20)
# cell State
cx = torch.randn(3, 20)
state = (hx,cx)
output = []
for i in range(6):
    state = rnn(inp[i], state)
    hx = state(0)
    output.append(hx)
```

Model Validation

Split the dataset in three subsets

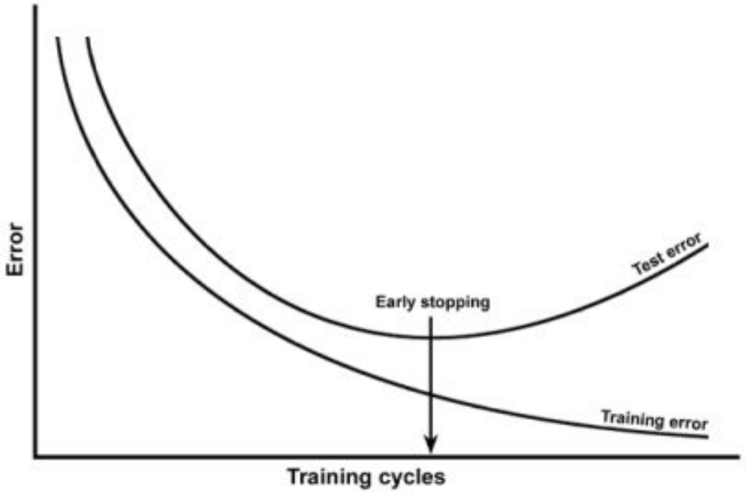
- **Training Set** : Data used for learning, namely to fit the parameters (weights) of the model
- **Validation Set :** Data used to tune the design parameters [i.e., architecture, not weights] of a model (hidden units, layers, batch size, etc.). Also used to prevent overfitting
- **Test Set :** Data used to evaluate the generalization of the model on unseen data



Overfitting - Regularization

- When model capacity is very large → memorizing input data instead of learning useful features
- “Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error” - *Deep Learning 2016*
- Goal of regularization is to prevent overfitting

- Some regularization methods:
  - Early stopping
  - Adding noise to train data
  - Penalize the norm of weights
  - Data Set Augmentation
  - Dropout

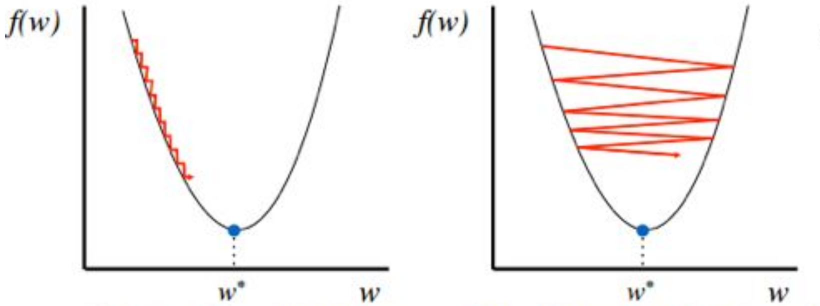


SGD Algorithm

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

- m=1 : Stochastic Gradient Descent (SGD)
- m<dataset : Minibatch SGD
- m=dataset : Gradient Descent

Learning Rate



Too small: converge very slowly

Too big: overshoot and even diverge

SGD Pytorch Code - Feedforward NN

```
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-4)
dataset = Data.DSD100Dataset(subset = 'Train')
dataloader = DataLoader(dataset, batch_size = 8, shuffle=True)
 # loop over the dataset multiple times
for epoch in range(100):
    for miniBatch in enumerate(dataloader, 0):
        # miniBatch consists of m=8 training examples
        inputs, targets = miniBatch
        # zero the parameter gradients
        model.zero_grad()
        # forward
        outputs = model(inputs)
        # calculate loss
        loss = lossFunction(outputs, targets)
        # Compute gradient estimate
        loss.backward()
        # Apply update
        optimizer.step()
```

SGD Pytorch Code - RNN

```
for epoch in range(100):
    for miniBatch in enumerate(dataloader, 0):
        inputs, targets = miniBatch
        # inputs.shape = batchSize x inputSize x seqLen (timesteps)
        model.zero_grad()
        # model.initState can be a process of your model class, that returns
        # a state in the appropriate shape.
        state = model.initState(batchSize)
        for timeStep in range(seqLen):
            # You can see the loop here, the output state of this step
            # will be the input state of the next step
            out, state = model(mixture[:,:,timeStep], state)

            partialLoss +=  lossFunction(targets[:,:,timeStep], out)
            if timeStep%length_TBPTT == length_TBPTT-1:
                # every length_TBPTT steps, calculate gradients (BPTT)

                # Detach state from the computation graph
                # so BPTT stops at the beggining of every truncated sequence
                # This is a statefull implementation
                state = (state[0].detach(), state[1].detach())
                # For stateless, comment the above line, and uncomment the following
                # state = model.initState(batchSize)
```