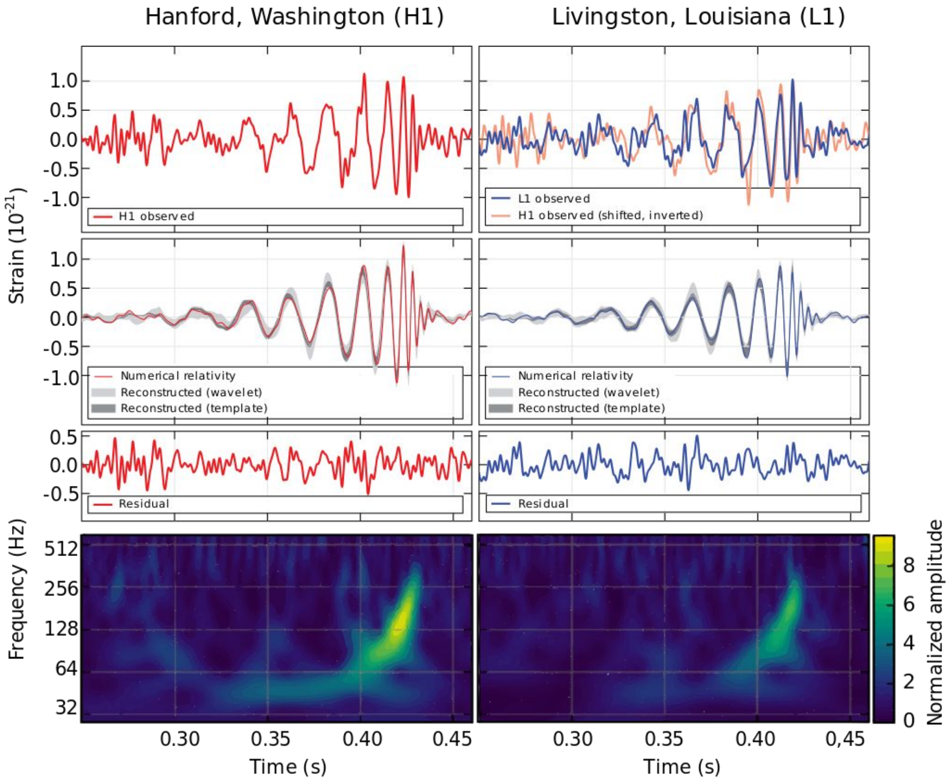
Sequences in Deep Learning

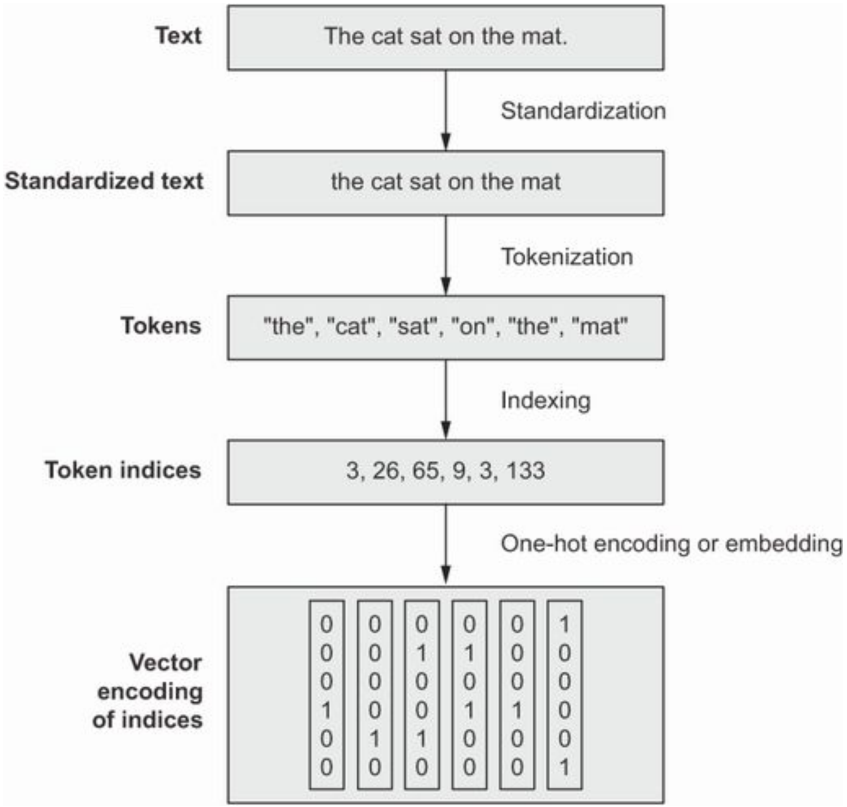
University of Victoria - PHYS-555

Stock Market

Gravitational Waves

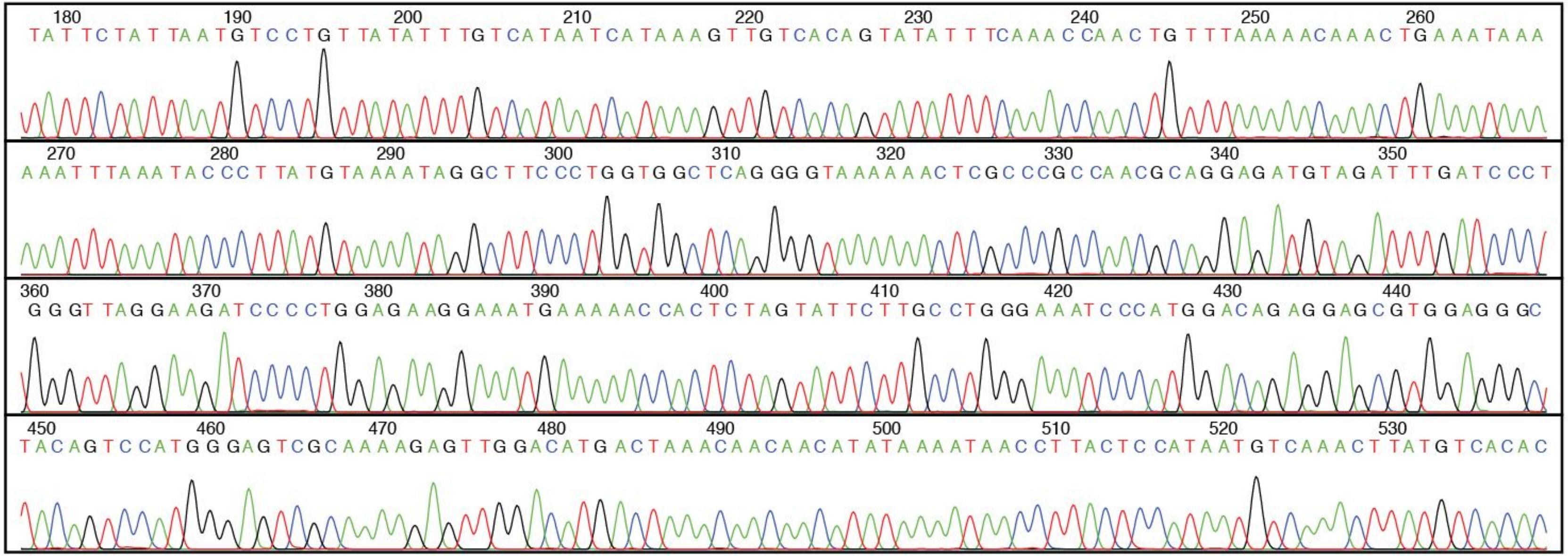


Text



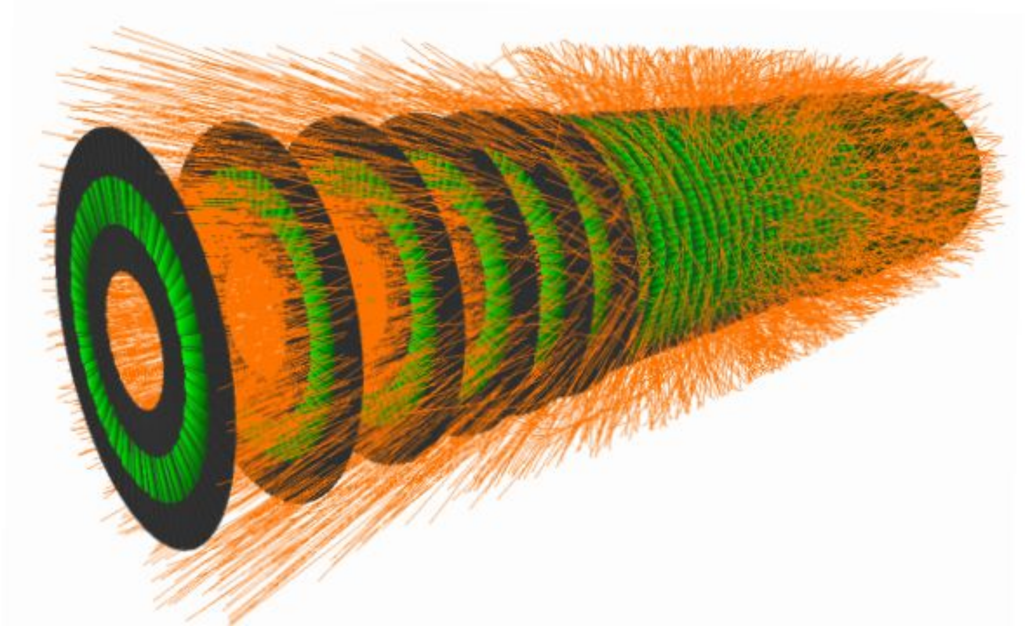
Text
The cat sat on the mat.
Standardization
Standardized text
the cat sat on the mat
Tokenization
Tokens
"the", "cat", "sat", "on", "the", "mat"
Indexing
Token indices
3, 26, 65, 9, 3, 133
One-hot encoding or embedding
Vector encoding of indices
0 0 0 1 0 0
0 0 0 0 1 0
0 1 0 0 1 0
0 1 0 1 0 0
0 0 0 1 0 0
1 0 0 0 0 1

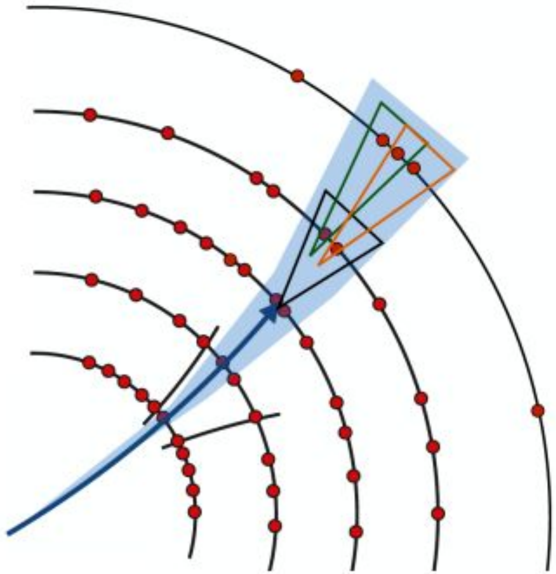
DNA Sequencing



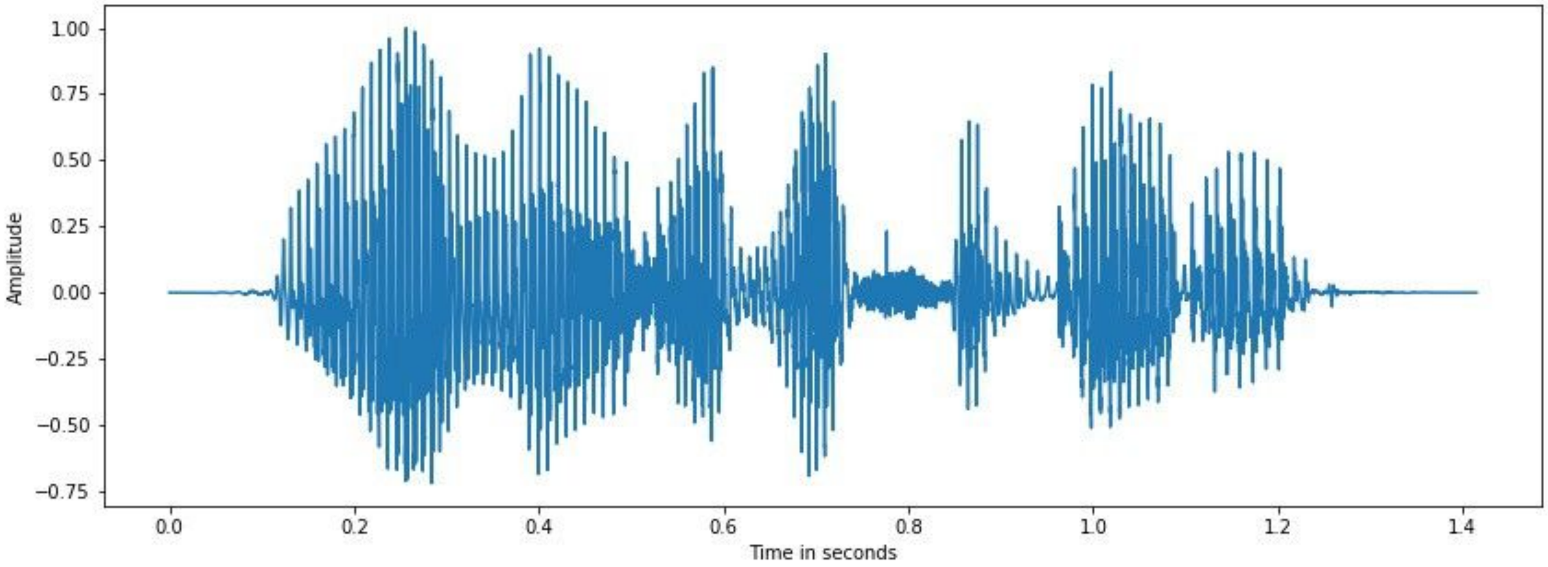
DNA sequence data from an automated sequencing machine

Particle Track Reconstruction



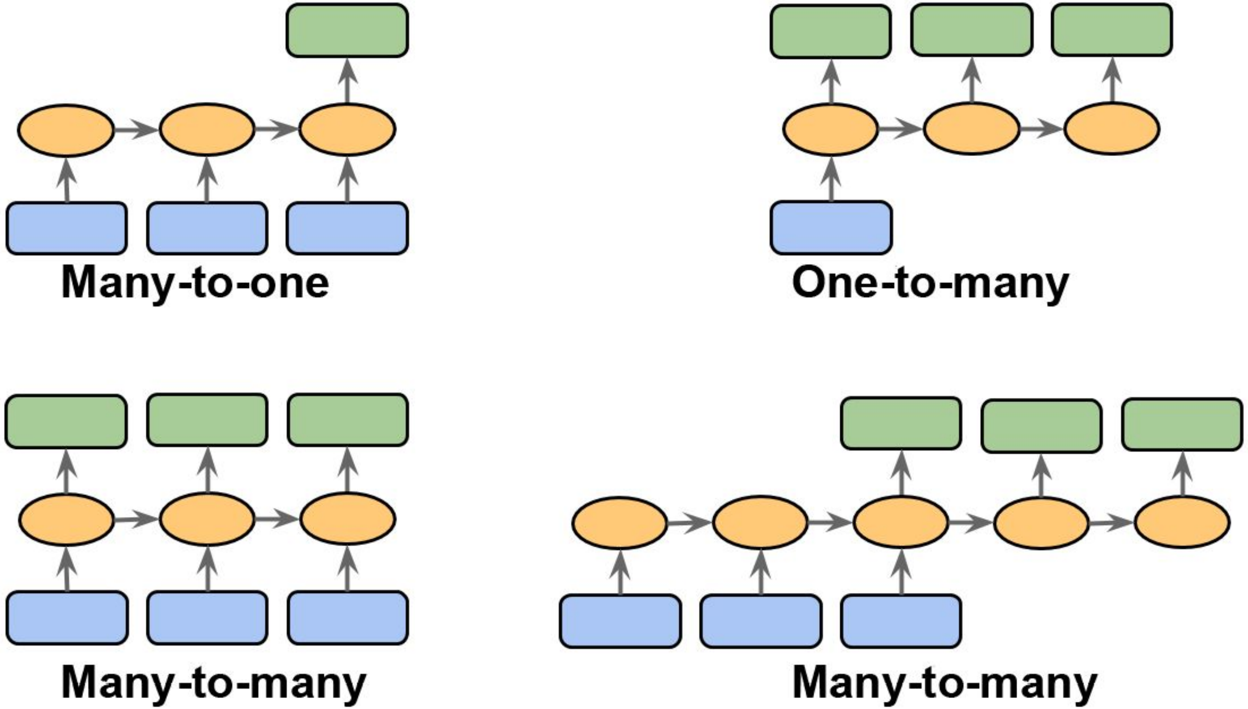


Speech Analysis

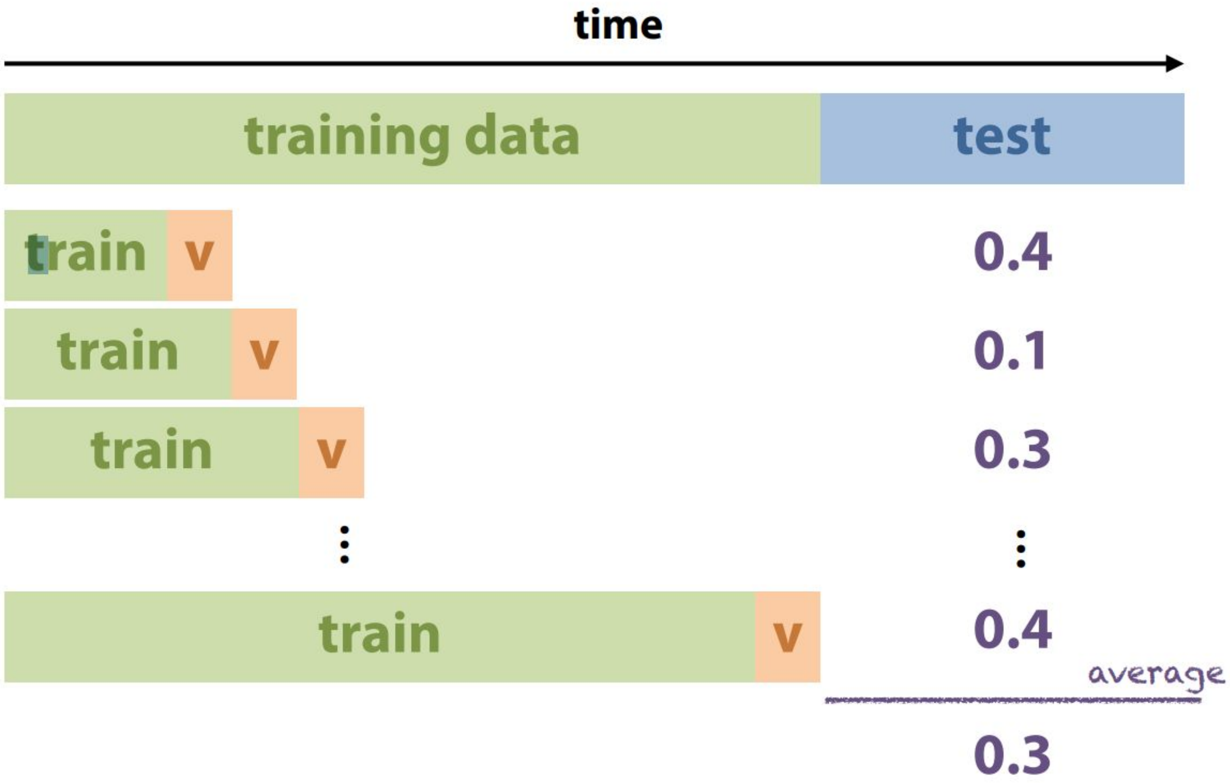


Try this sketching app

Type of Sequences



Learning from Sequences



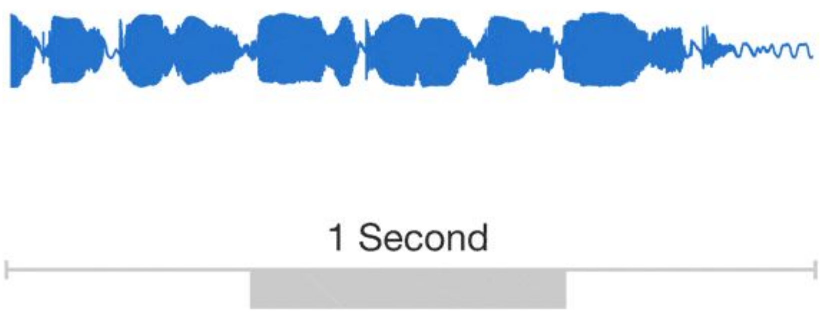
Why not using a CNN?

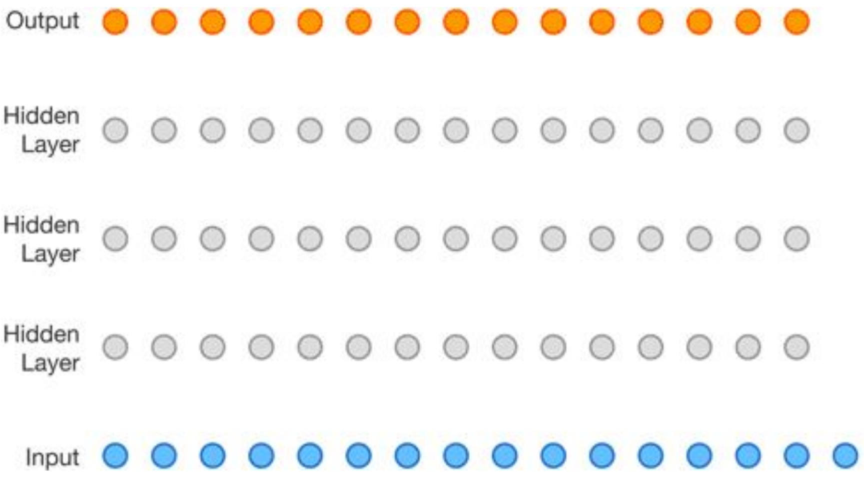
- Extensions of 1D CNN.

Example: time series of N steps, each step with c features.

CNNs can be extended easily to other domains having grid-like structure of various dimensions. For example, consider a time-series of n steps, each step having c features (e.g., c different readings from different sensors). We would need masked convolution (dilated)

Example: WaveNet



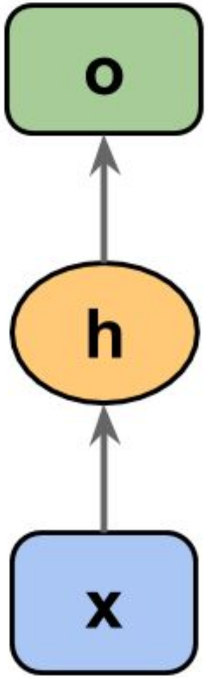


[WaveNet: A Generative Model for Raw Audio]

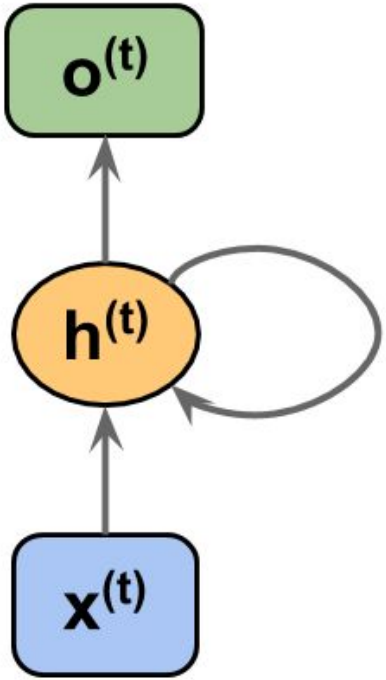
[DeepMind Blog Post]

Recurrent Neural Networks

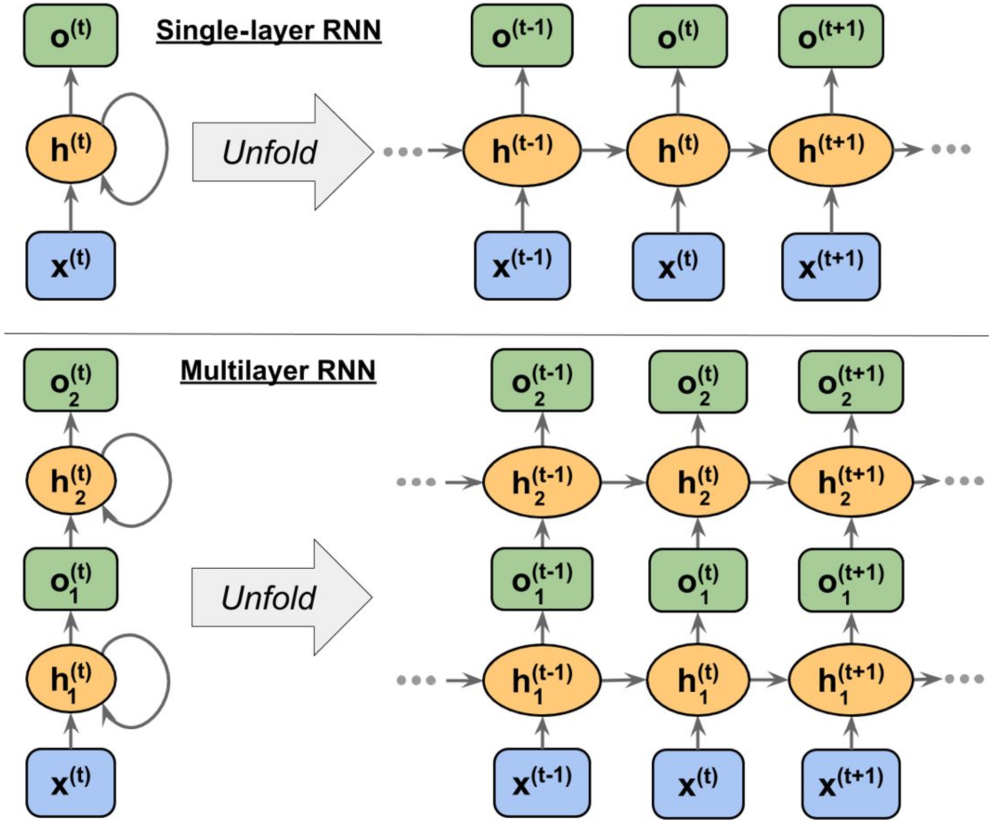
A standard feedforward network



Recurrent neural network



Unfolding the RNN



Elman Network (1990)

1. Start with hidden state:

$$h_0 = 0$$

2. Update with new state

$$h_t = \mathrm{ReLU}(\mathbf{w}_{xh} x_t + \mathbf{w}_{hh} h_{t-1} + b_h)$$

3. Final prediction

$$y_T = \mathbf{w}_{hy} h_T + b_y.$$

Elman Network (1990)

1. Start with hidden state:

$$h_0 = 0$$

2. Update with new state

$$h_t = \mathrm{ReLU}(\mathbf{w}_{xh} x_t + \mathbf{w}_{hh} h_{t-1} + b_h)$$

3. Final prediction

$$y_T = \mathbf{w}_{hy} h_T + b_y.$$

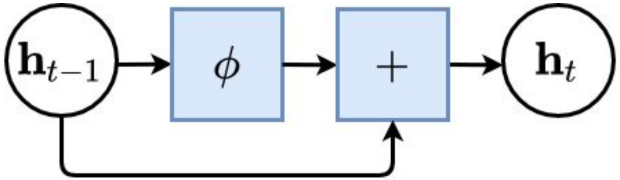
```
class ElmanNet(nn.Module):
    def __init__(self, size_input, size_hidden, size_output):
        super(ElmanNet, self).__init__()
        self.fc_x2h = nn.Linear(size_input, size_hidden)
        self.fc_h2h = nn.Linear(size_hidden, size_hidden, bias=False)
        self.fc_h2y = nn.Linear(size_recurrent, size_output)
    def forward(self, x):
        h = x.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(x.size(0)):
            h = torch.relu(self.fc_x2h(x[t,:]) + self.fc_h2h(h))
        return self.fc_h2y(h)
```

Elman Network in PyTorch

```
class ElmanNet(nn.Module):
    def __init__(self, size_input, size_hidden, size_output):
        super(ElmanNet, self).__init__()
        self.fc_x2h = nn.Linear(size_input, size_hidden)
        self.fc_h2h = nn.Linear(size_hidden, size_hidden, bias=False)
        self.fc_h2y = nn.Linear(size_recurrent, size_output)
    def forward(self, x):
        h = x.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(x.size(0)):
            h = torch.relu(self.fc_x2h(x[t,:]) + self.fc_h2h(h))
        return self.fc_h2y(h)
```

```
rnn = ElmanNet(size_input=10, size_hidden=50, size_output=2)
cross_entropy = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
for k in range(data.size()):
    x,label = data.get_batch()
    y = rnn(x)
    loss = cross_entropy(y, label)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Gating



- Gates decides which information to go through.
  They are composed out of a sigmoid **σ** neural net layer and a pointwise multiplication operation.

- The sigmoid outputs numbers between zero and one, zero means “let nothing through,” while a value of one means “let everything through!”

Gating Implementation

Update hidden state proposal: (same as Elman)

$$\overline{h}_t = \mathrm{ReLU}(\mathbf{w}_{xh}x_t + \mathbf{w}_{hh}h_{t-1} + b_h)$$

Forget gate:

$$z_t = \sigma(\mathbf{w}_{xz}x_t + \mathbf{w}_{hz}h_{t-1} + b_z)$$

Hidden State:

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

Gating Implementation

Update hidden state proposal: (same as Elman)

$$\overline{h}_t = \mathrm{ReLU}(\mathbf{w}_{xh}x_t + \mathbf{w}_{hh}h_{t-1} + b_h)$$

Forget gate:

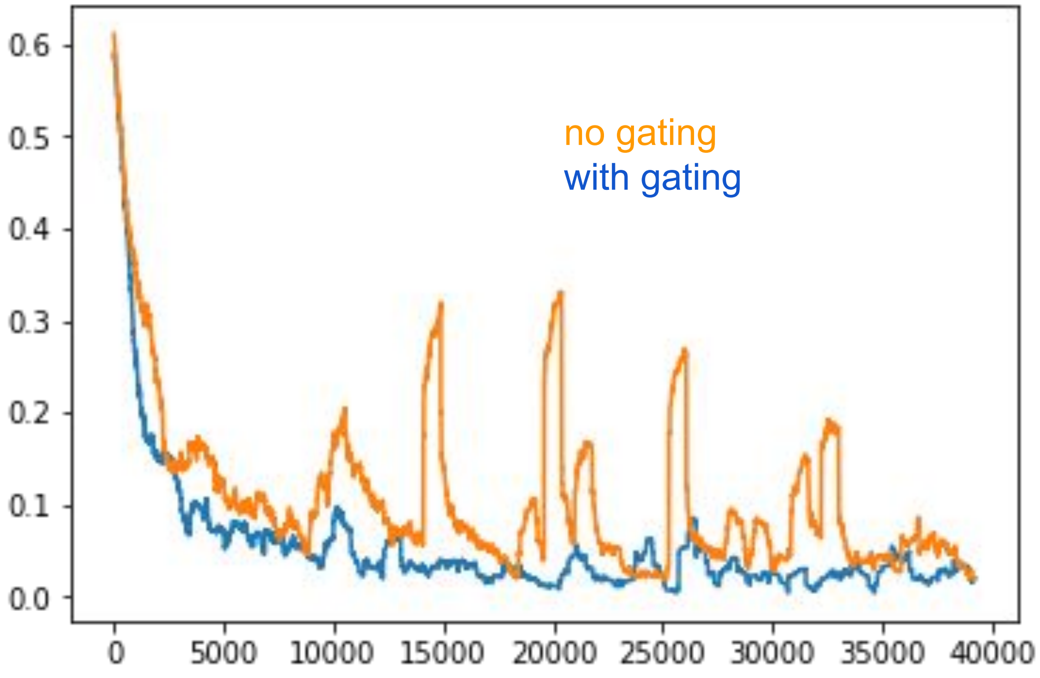
$$z_t = \sigma(\mathbf{w}_{xz}x_t + \mathbf{w}_{hz}h_{t-1} + b_z)$$

Hidden State:

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

```
class ElmanNetGating(nn.Module):
   def __init__(self,size_input, size_hidden, size_output):
       super(ElmanNetGating, self).__init__()
       self.fc_x2h = nn.Linear(size_input, size_hidden)
       self.fc_h2h = nn.Linear(size_hidden, size_hidden, bias=False)
       self.fc_x2z = nn.Linear(size_input, size_hidden)
       self.fc_h2z = nn.Linear(size_hidden, size_hidden, bias=False)
       self.fc_h2y = nn.Linear(size_hidden, size_output)
   def forward(self, x):
       h = x.new_zeros(1, self.fc_h2y.weight.size(1))
       for t in range(x.size(0)):
           z = torch.sigmoid(self.fc_x2z(x[t,:])+self.fc_h2z(h))
           hb = torch.relu(self.fc_x2h(x[t,:]) + self.fc_h2h(h))
           h = z * h + (1-z) * hb
       return self.fc_h2y(h)
```

Training with Gating



Loss curve

Gated Recurrent Units (GRU)

Update hidden state proposal:

$$\overline{h}_t = \tanh(\mathbf{w}_{xh}x_t + \mathbf{w}_{hh}(r_t \odot h_{t-1}) + b_h)$$

Forget gate:

$$z_t = \sigma(\mathbf{w}_{xz}x_t + \mathbf{w}_{hz}h_{t-1} + b_z)$$

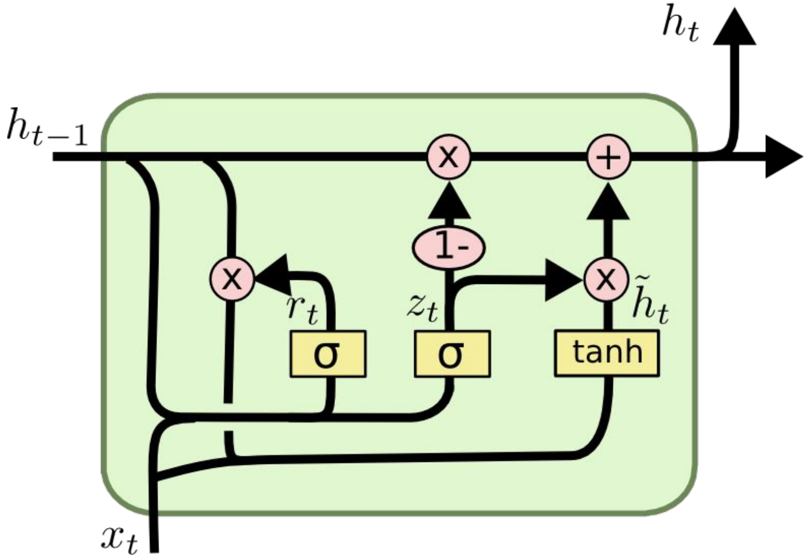
Reset gate:

$$r_t = \sigma(\mathbf{w}_{xr}x_t + \mathbf{w}_{hr}h_{t-1} + b_r)$$

Hidden State:

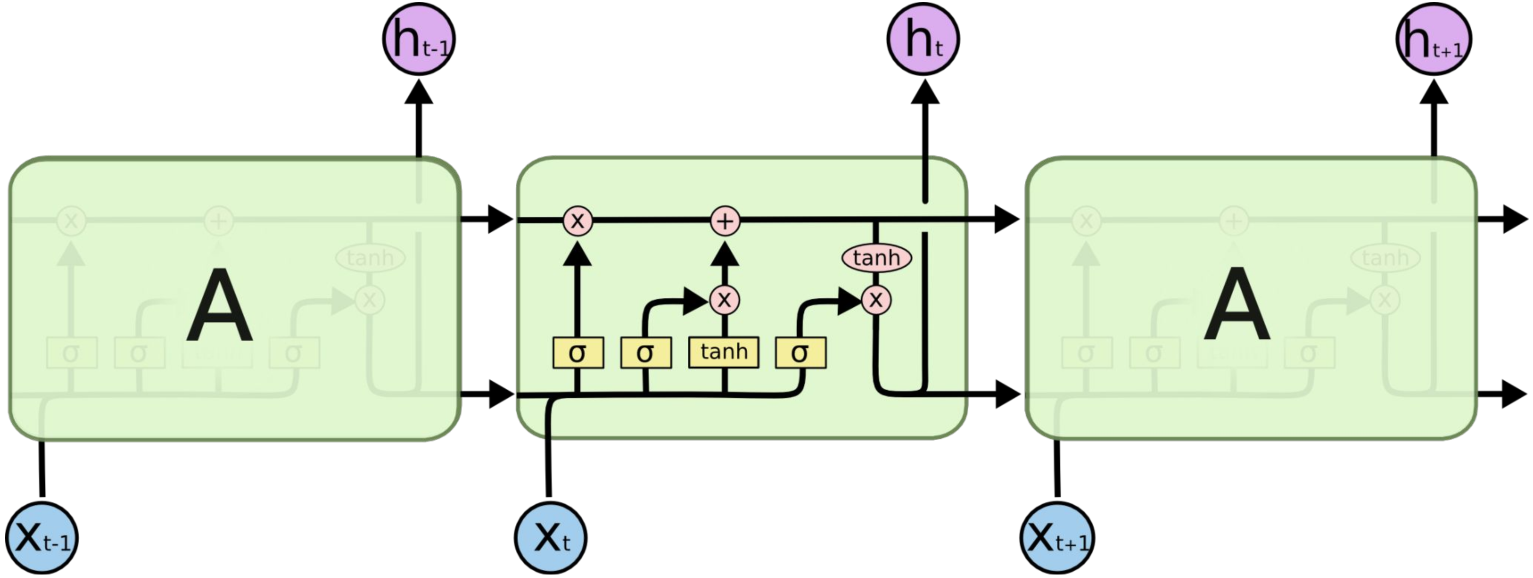
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t$$

GRU



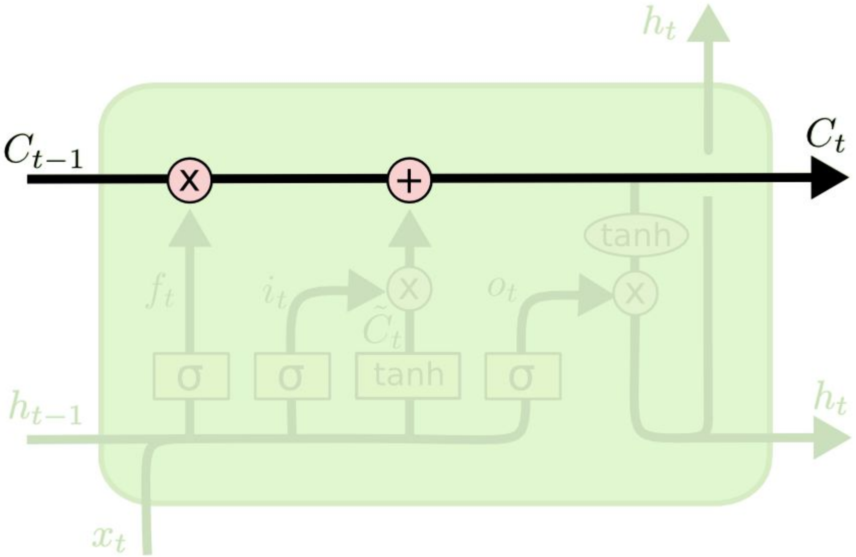
Cho et. al (2014)

Long Short-Term Memory (LSTM)



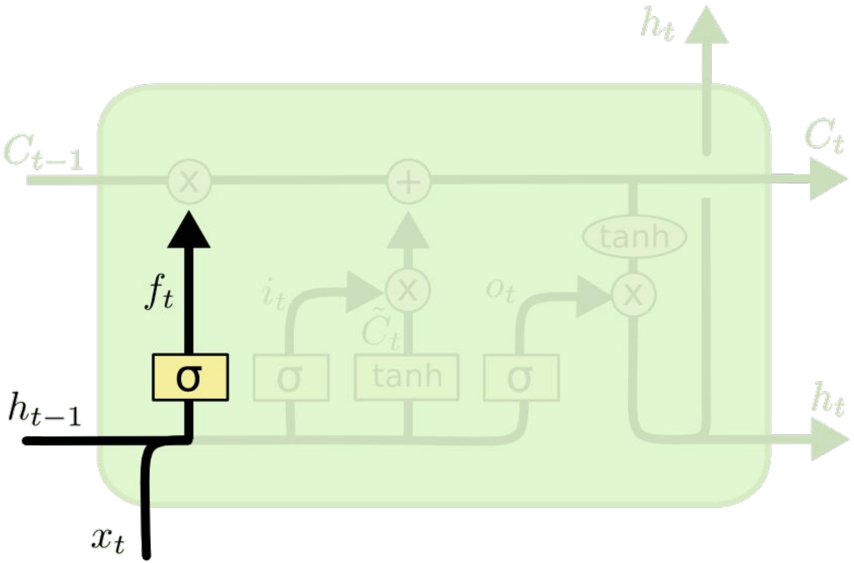
Hochreiter & Schmidhuber (1997)

LSTM



Cell State

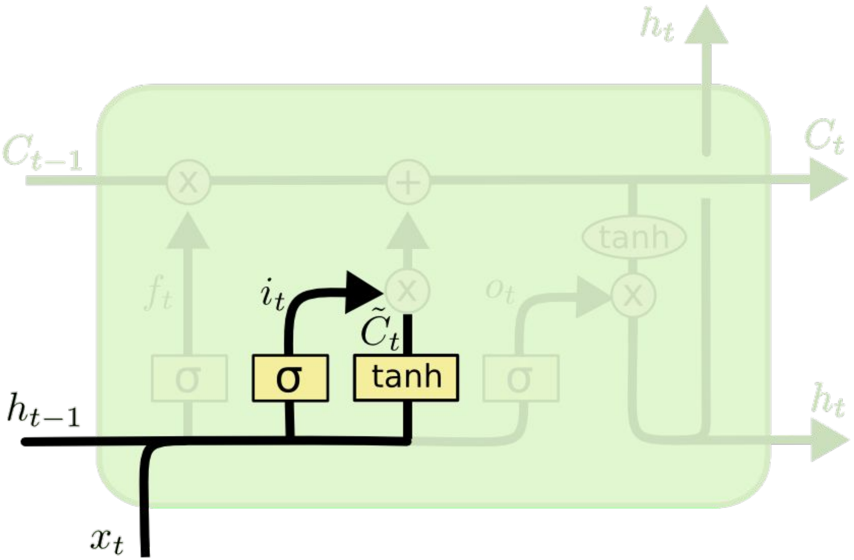
LSTM



Forget gate layer

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

LSTM

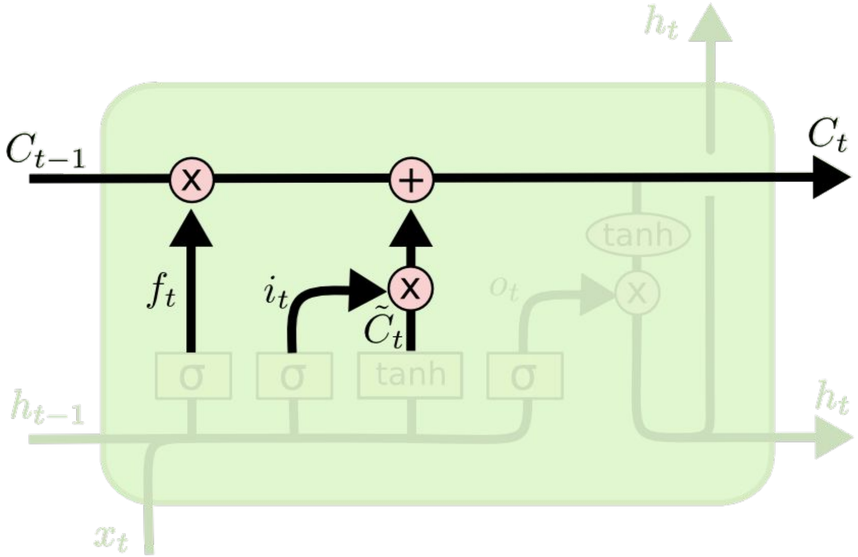


Input Gate Layer

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

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

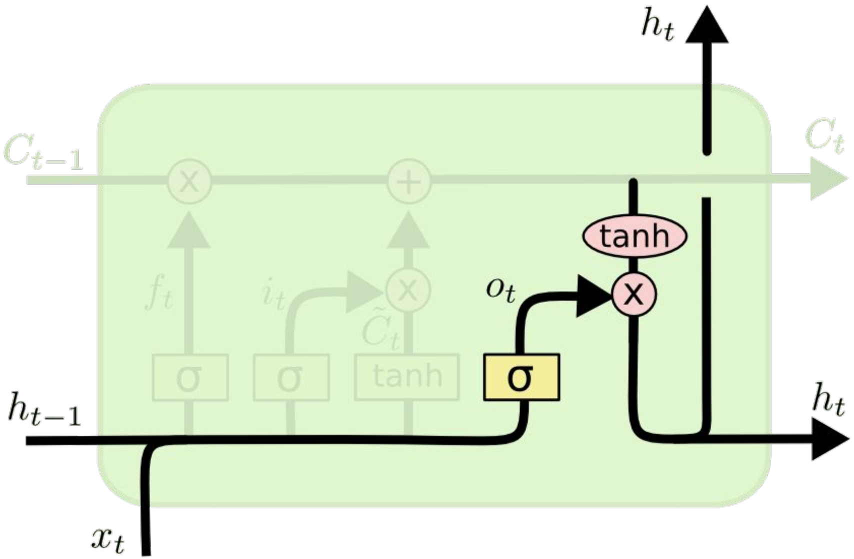
LSTM



Update Cell State

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

LSTM



Output gate

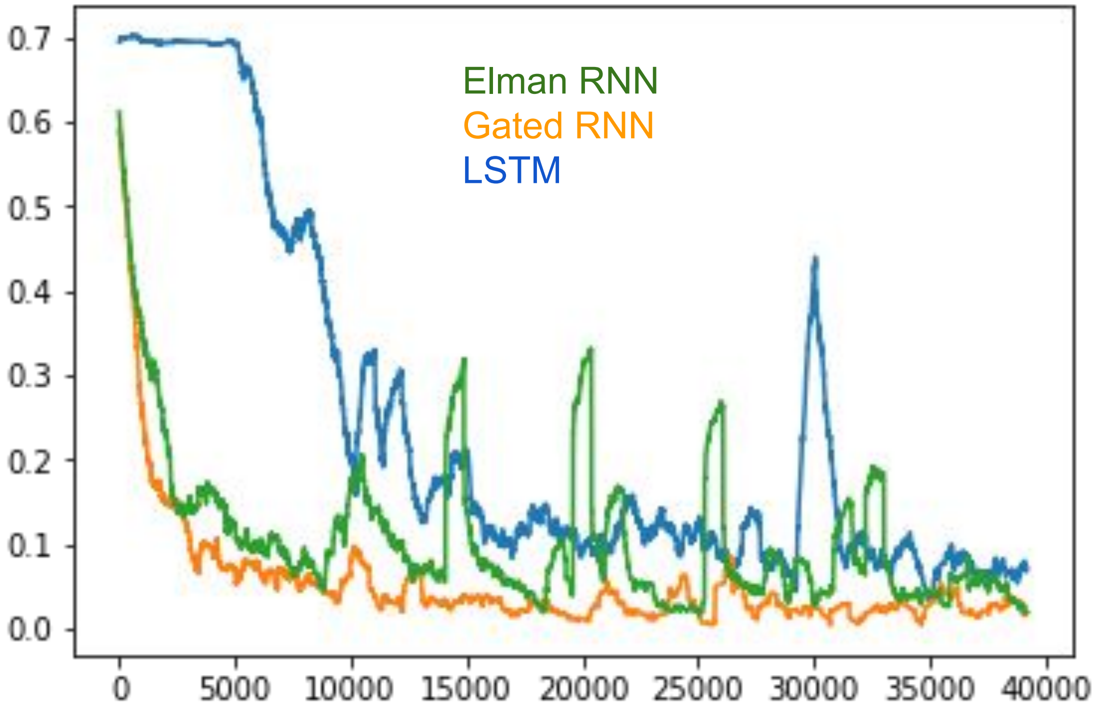
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

LSTM in PyTorch

```
class MyLSTM(nn.Module):
    def __init__(self, size_input, size_hidden, num_layers, size_output):
        super(MyLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size=size_input, hidden_size=size_hidden, num_layers=num_layers)
        self.fc_o2y = nn.Linear(size_hidden, size_output)
    def forward(self, x):
        x = x.unsqueeze(1) # expect a batch size (here is 1)
        output, _ = self.lstm(x)
        output = output.squeeze(1) # only last layer, shape (seq. len., bs, dim_recurrent) and drop the batch index
        output = output.narrow(0, output.size(0)-1,1) # keep only the last hidden variable
        return self.fc_o2y(F.relu(output))   # shape (1, dim_recurrent)
```

Training with LSTM



Loss curve

Resources

- Chris Olah: Understanding LSTM Networks
- Andrej Karpathy: The Unreasonable Effectiveness of Recurrent Neural Networks