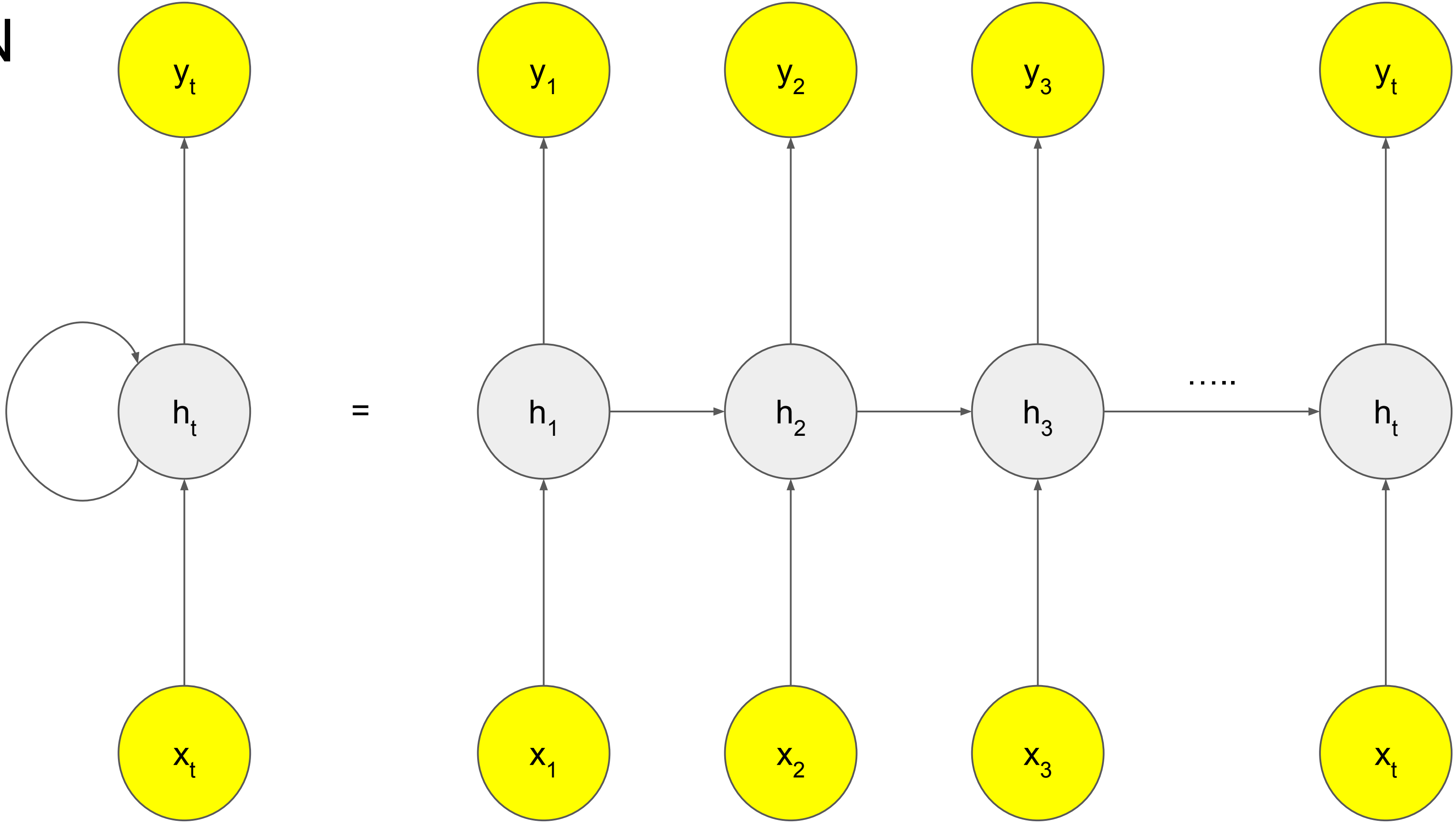
# Recurrent Neural Networks

Intro to Deep Learning

# RNN



# RNN

When the first input ($x_1$) comes in, the first memory ($h_1$) is created. When the second input ($x_2$) comes in, the existing memory ($h_1$) is referenced along with the new input to create a new memory ($h_2$). This process can be repeated for any length of inputs.

In short, the input is (x) from the data and memory (h), and the output becomes y along with the memory (h).

# RNN type

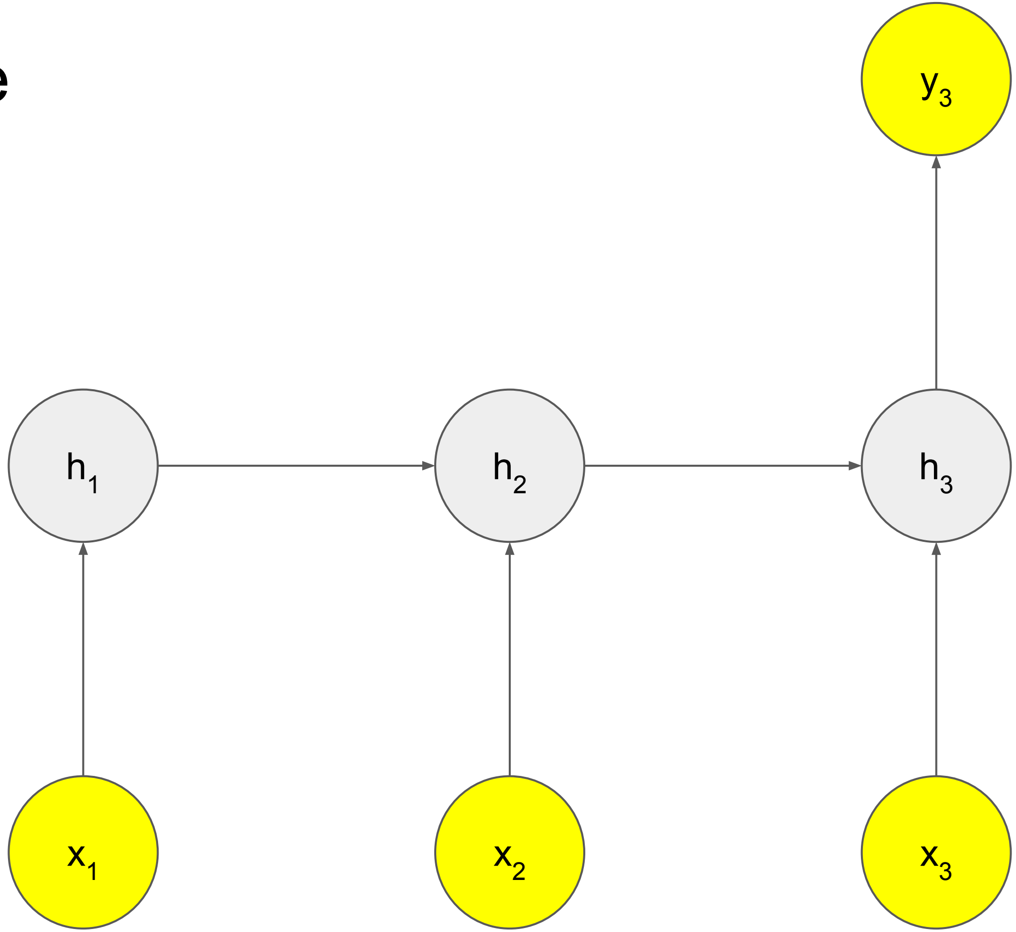
**One-to-One**: It is difficult to call it an RNN because there is no recurrence. It is a basic neural network structure where each input produces one output, without any recurrent connections.

**One-to-Many**: It is a structure where one input produces multiple outputs. A typical example is **image captioning**, where an image is given as input, and a sentence describing the image is generated as the output.

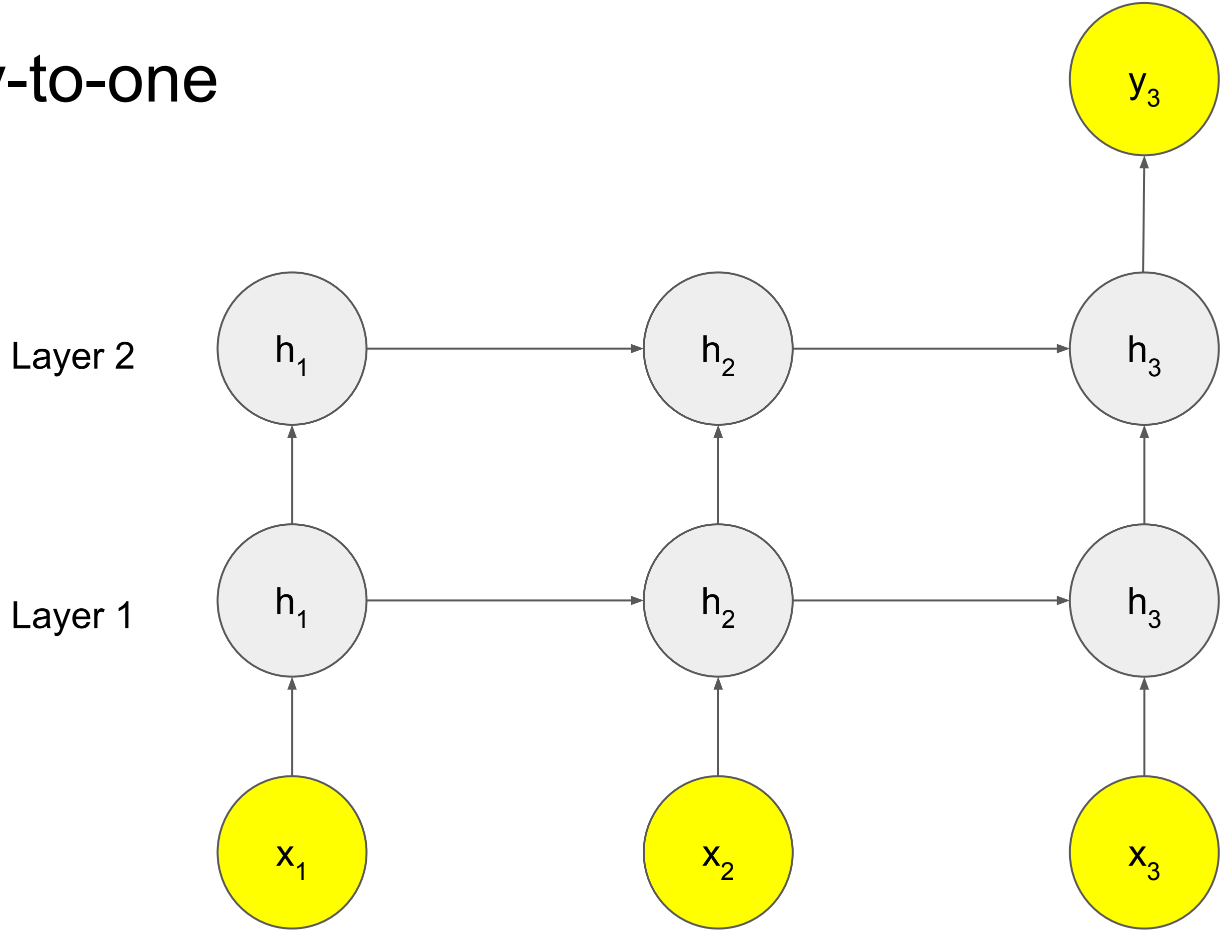
**Many-to-One**: It is a structure where multiple inputs produce one output. A **sentiment analyzer** is a representative example of this. It takes a sentence as input and outputs the sentiment (positive/negative) of that sentence.

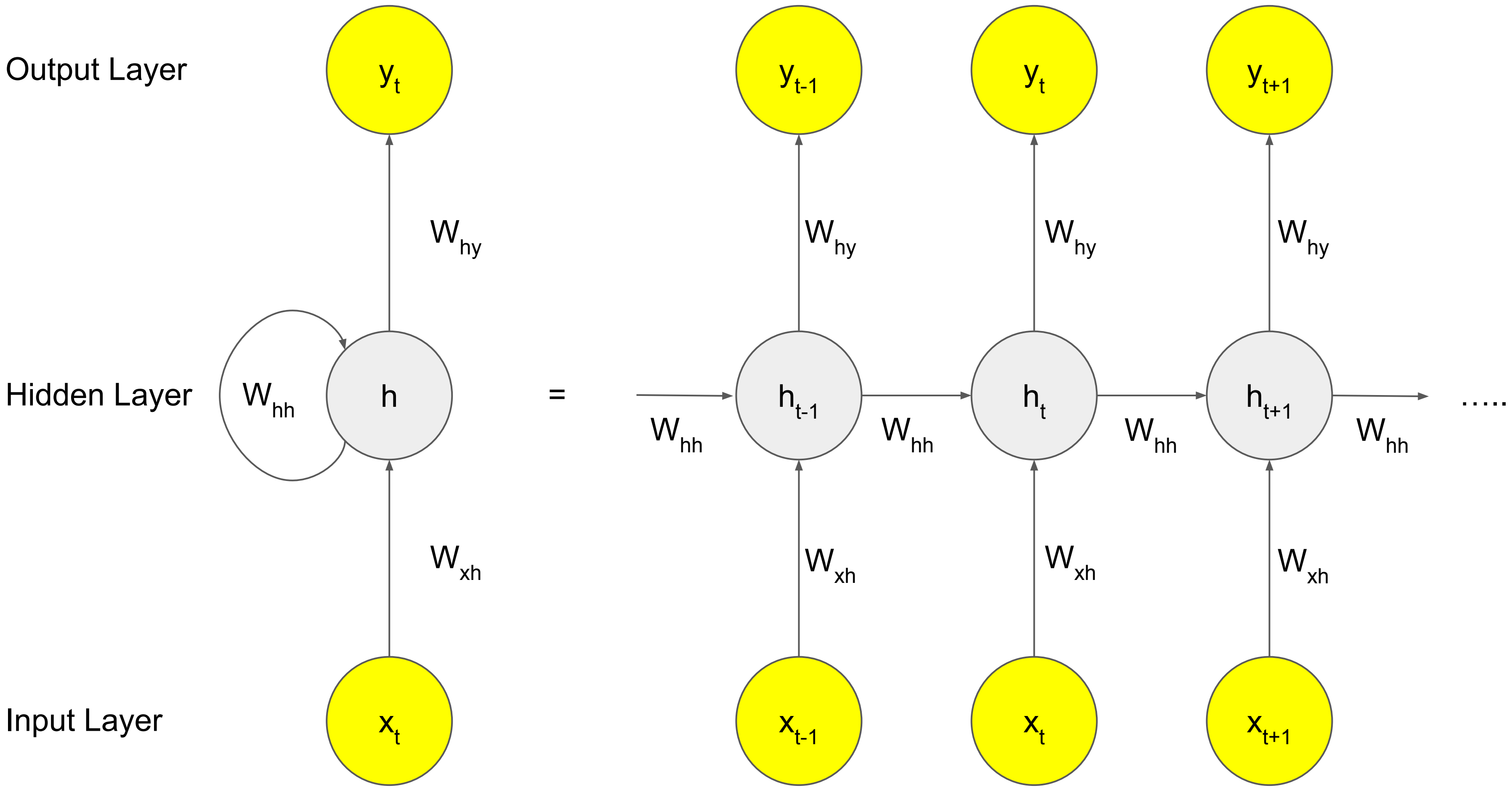
**Many-to-Many**: It is a structure where multiple inputs produce multiple outputs. The lengths of the input and output sequences can be different. **Machine translation** or speech recognition tasks can be examples of this structure.

# Many-to-one



# Many-to-one





Output Layer
Hidden Layer
Input Layer
$y_t$
$W_{hy}$
$W_{hh}$
h
$W_{xh}$
$x_t$
=
$y_{t-1}$
$y_t$
$y_{t+1}$
$W_{hy}$
$W_{hy}$
$W_{hy}$
$W_{hh}$
$h_{t-1}$
$W_{hh}$
$h_t$
$W_{hh}$
$h_{t+1}$
$W_{hh}$
.....
$W_{xh}$
$W_{xh}$
$W_{xh}$
$x_{t-1}$
$x_t$
$x_{t+1}$

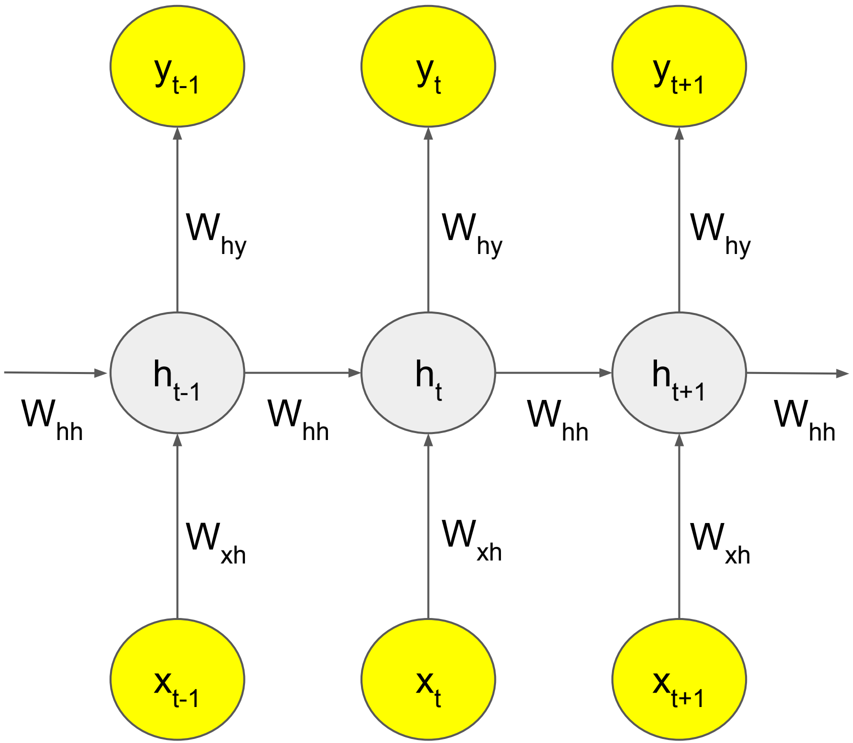
# Hidden Layer and Output Layer

- Hidden Layer

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

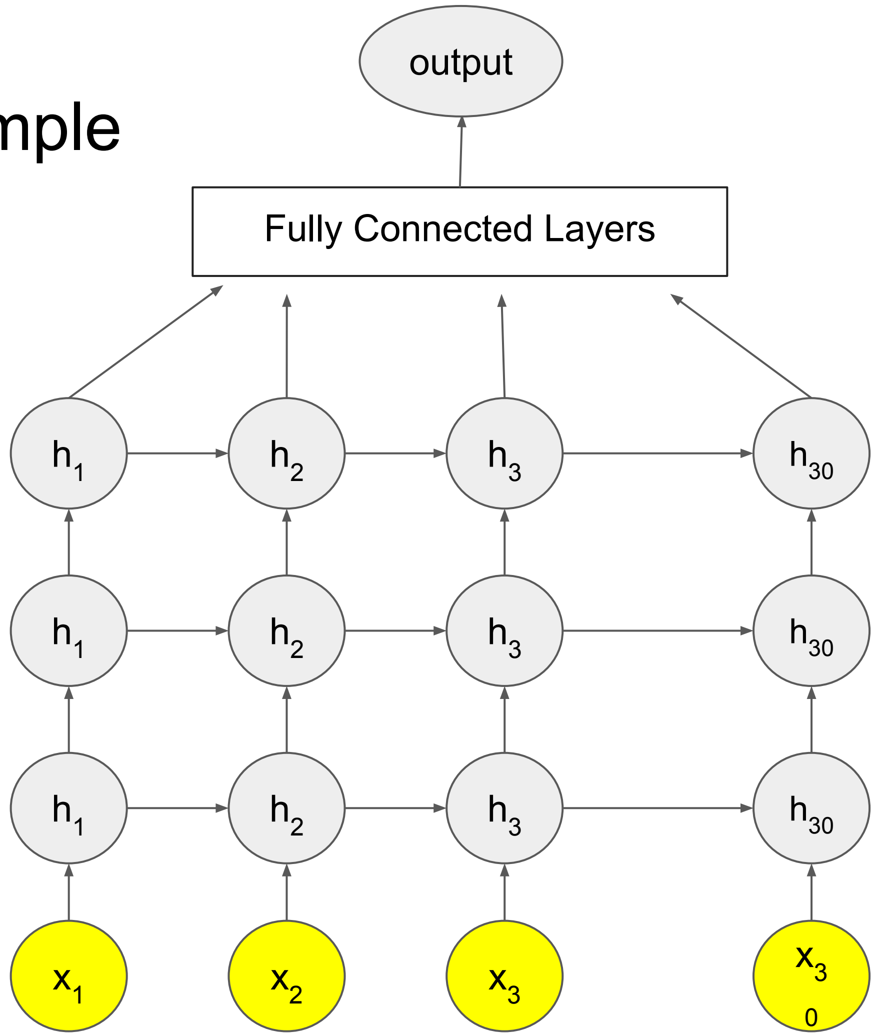
- Output Layer

$$y_t = softmax(W_{hy} \text{ x } h_1)$$



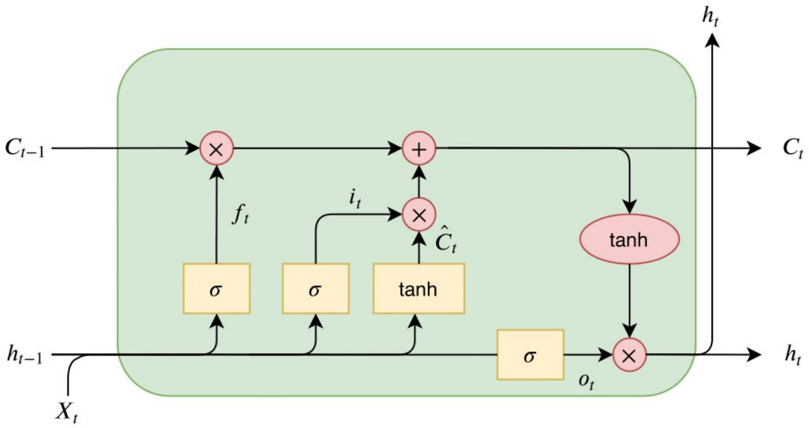
# Netflix stock example

5 hidden layers



# Long Short-Term Memory (LSTM)

LSTM has added new elements to the hidden layer: the forget gate, the input gate, and the output gate.

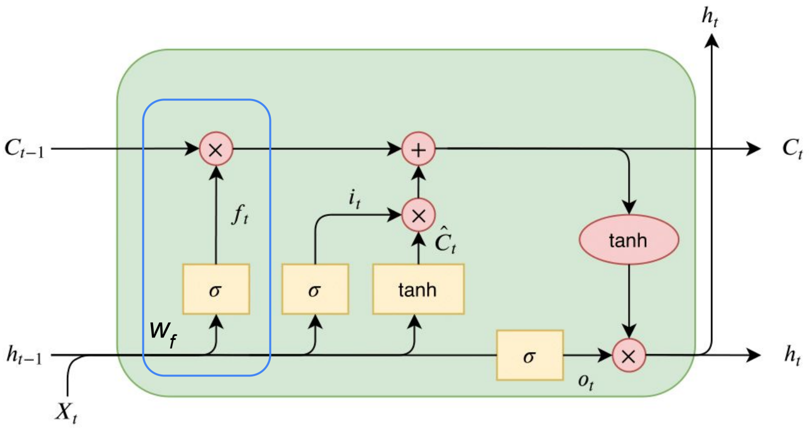


# Forget gate

The forget gate in LSTM determines how much of the past information to retain. It takes the past information, i.e., the memory, and the current input data, and after applying the sigmoid function to them, it multiplies the result with the past information. Therefore, if the output of the sigmoid is **0**, the past information is discarded, but if it's **1**, the past information is fully preserved.

$$f_t = sigmoid(w_f\,[h_{t-1}, x_t])$$

$$c_t = f_t \times c_{t-1}$$



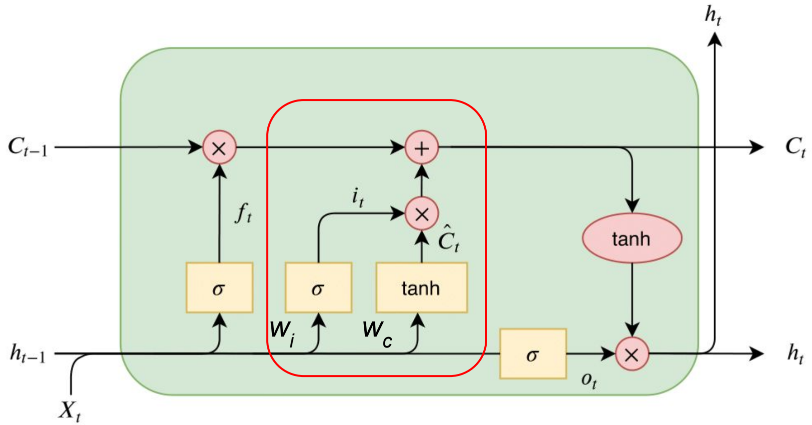
# Input gate

The input gate is responsible for preserving the current input information. It uses the sigmoid and tangent functions to determine how much of the current input information should be retained. In other words, it decides how much new information should be added to the memory.

$$i_t = sigmoid(w_i [h_{t-1}, x_t])$$

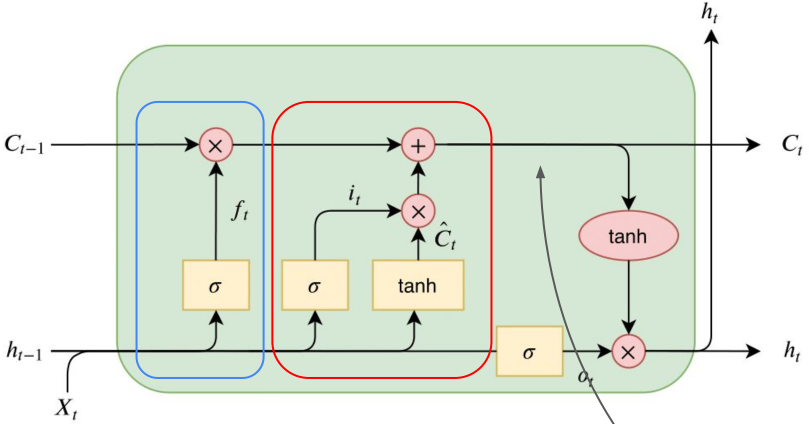
$$c'_t = tanh(w_c [h_{t-1}, x_t]$$

$$c_t = c_{t-1} + i_t \times c'_t$$



# Cell gate

Update the cell state




$$f_t = sigmoid(w_f\,[h_{t-1}, x_t])$$

$$c_t = f_t \text{ x } c_{t-1}$$

$$i_t = sigmoid(w_f\,[h_{t-1}, x_t])$$

$$c'_t = tanh(w_c\,[h_{t-1}, x_t]$$

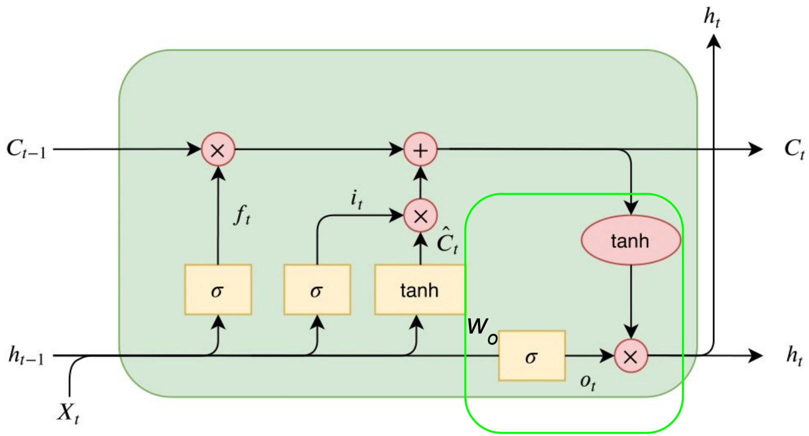
$$c_t = c_{t-1} + i_t \text{ x } c'_t$$

# Output gate

Output gate controls memory to output ($h_t$)

$$o_t = sigmoid(w_o\,[h_{t-1}, x_t])$$

$$h_t = o_t \times tanh(c_{t-1})$$



# LSTM in Pytorch

Docs > torch.nn > LSTM

## LSTM

CLASS torch.nn.LSTM(*self*, *input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *proj_size=0*, *device=None*, *dtype=None*) [SOURCE]

**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 LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **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*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

# Model

```
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_dim, output_dim) # Define the output layer

    def forward(self, x):
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
        out = self.linear(out[:, -1, :]) # Index hidden state of last time step
        return out
```

# Parameters and DataLoader

```
# Data parameters
sequence_length = 10
input_dim = 5
num_samples = 1000
num_classes = 2

# Random data generation
data = torch.randn(num_samples, sequence_length, input_dim)
labels = torch.randint(0, num_classes, (num_samples,))

# TensorDataset
dataset = TensorDataset(data, labels)

# DataLoader setting
batch_size = 64
train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

# Train

```
model = LSTMModel(input_dim, hidden_dim=50, output_dim=num_classes, num_layers=2)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

for epoch in range(20):
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f"Loss: {loss.item():.4f}")
```

# Results

```
Loss: 0.6949
Loss: 0.6960
Loss: 0.6902
Loss: 0.6843
Loss: 0.6831
Loss: 0.6872
Loss: 0.7516
Loss: 0.6750
Loss: 0.6405
Loss: 0.5504
Loss: 0.4740
Loss: 0.5475
Loss: 0.4156
Loss: 0.3247
Loss: 0.3120
Loss: 0.1653
Loss: 0.1970
Loss: 0.0627
Loss: 0.0142
Loss: 0.0080
```