Computer Vision in Python

Day 2, Part 3/4

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this lecture

1. What are **videos** and how can they be represented as a **sequence of image frames**?
2. What are typical uses of **video processing models** and why are image models alone **insufficient** to process videos?
3. What are **sequential models** and **recurrent neural networks**?
4. What are **RNN** models? **LSTM** models? **GRU** models?
5. How to **combine RNNs and CNNs** to process images?
6. (If time allows, what are **transformers**?)
7. (And, if time allows, how to **combine transformers and CNNs** for video processing?)

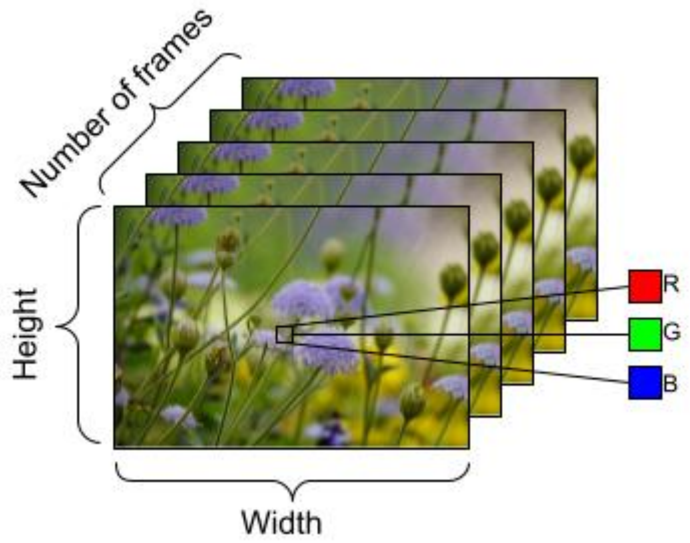
What is a video, mathematically?

Definition (video datatype):

The video datatype consists of an **ordered sequence of frames**

- Each frame I_t is an image, represented as a tensor with dimensions (C, H, W)
- A video clip V is then a sequence of image frames, represented as a tensor with dimensions (T, C, H, W)
- Temporal order matters!

$$V = \{ I_1, I_2, \ldots, I_T \}$$



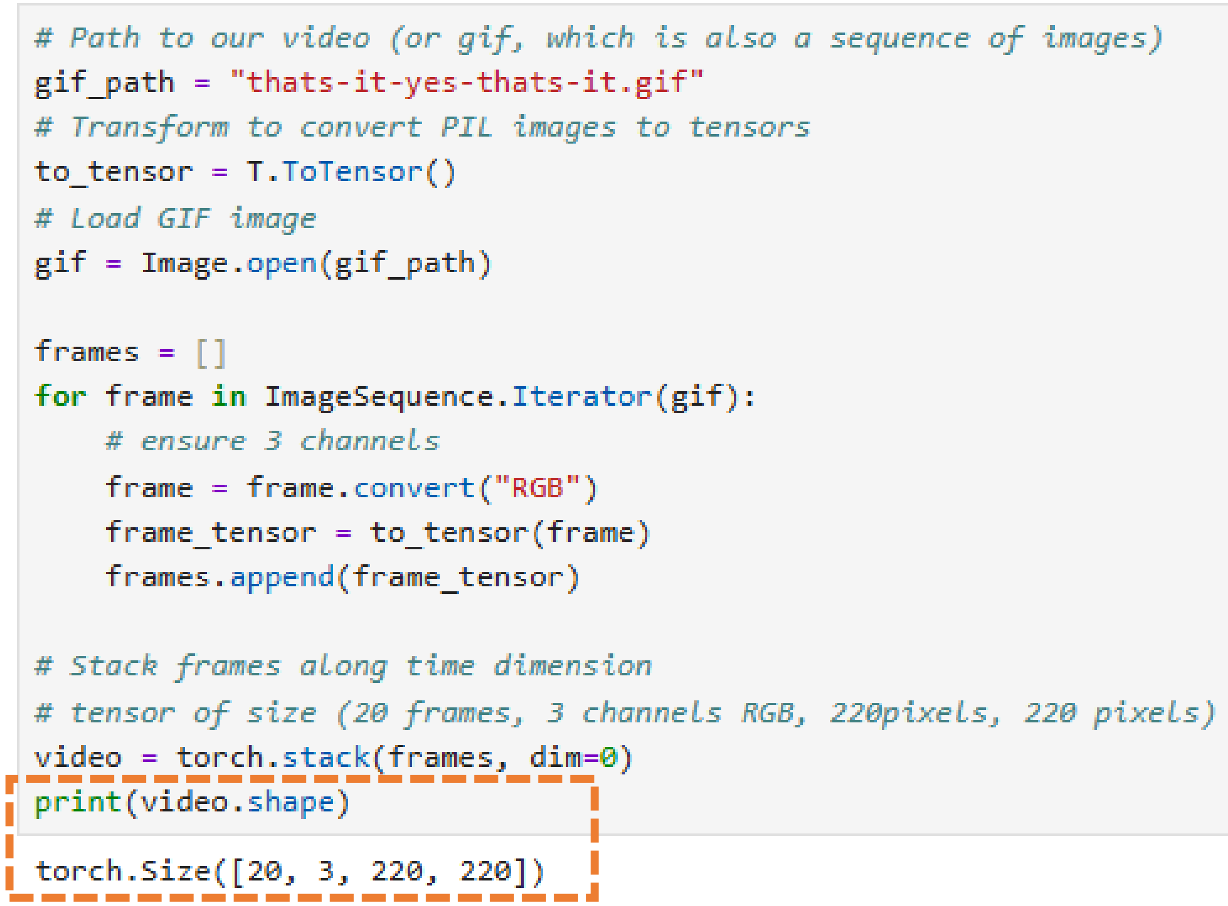
What is a video, mathematically?

Definition (video datatype):

The video datatype consists of an **ordered sequence of frames**

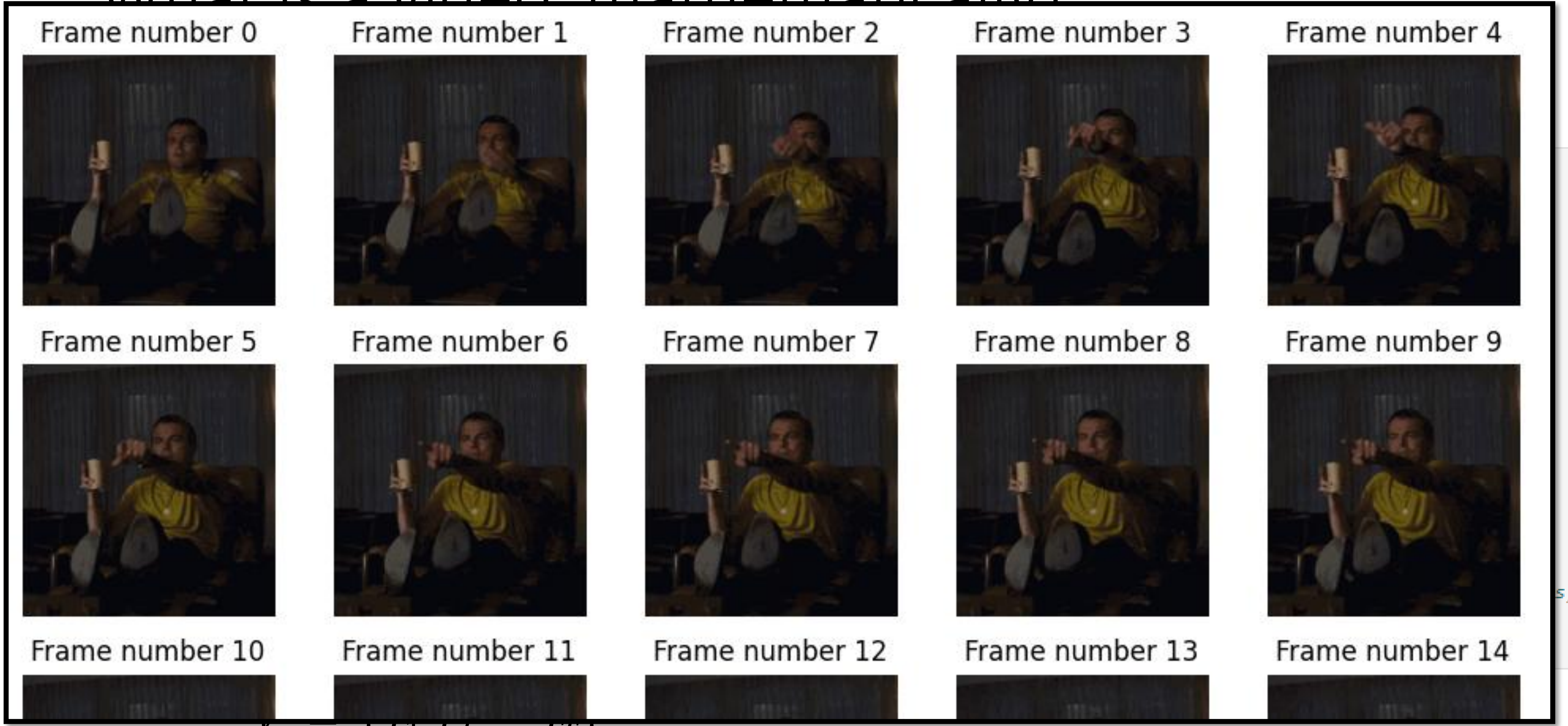
- Each frame I_t is an image, represented as a tensor with dimensions (C, H, W)
- A video clip V is then a sequence of image frames, represented as a tensor with dimensions **(T, C, H, W)**
- Temporal order matters!

$$V = \{ I_1, I_2, \ldots, I_T \}$$



```
# Path to our video (or gif, which is also a sequence of images)
gif_path = "thats-it-yes-thats-it.gif"
# Transform to convert PIL images to tensors
to_tensor = T.ToTensor()
# Load GIF image
gif = Image.open(gif_path)

frames = []
for frame in ImageSequence.Iterator(gif):
    # ensure 3 channels
    frame = frame.convert("RGB")
    frame_tensor = to_tensor(frame)
    frames.append(frame_tensor)

# Stack frames along time dimension
# tensor of size (20 frames, 3 channels RGB, 220pixels, 220 pixels)
video = torch.stack(frames, dim=0)
print(video.shape)

torch.Size([20, 3, 220, 220])
```

What is a video, mathematically?



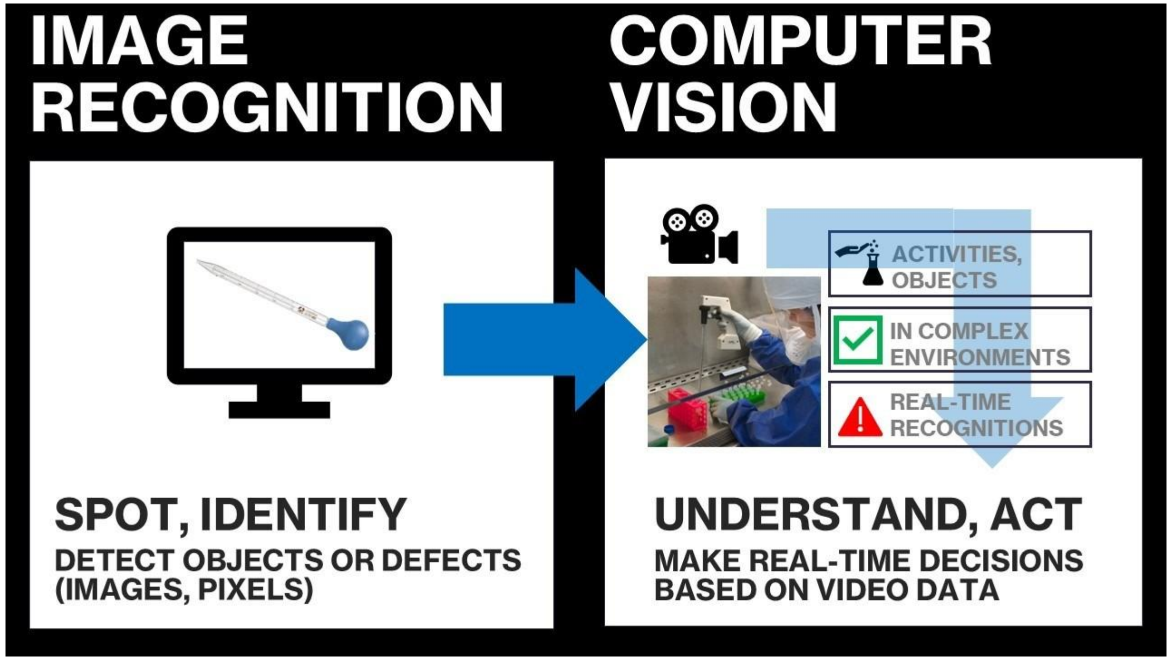
$V = \{I_1, I_2, \dots, I_T\}$

From images to videos

Key difference between image processing and video processing.

- Performing image recognition boiled down to **recognizing what is in the image.**
- Video recognition takes it one step further as it requires to **recognize what is happening over time.**

Far more complex task!



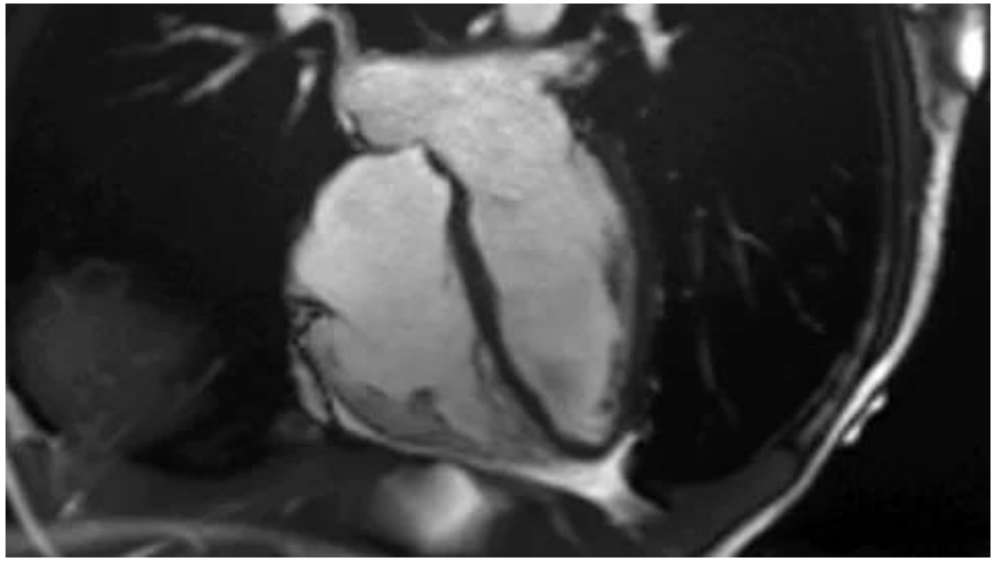
Applications

Self driving cars

- Is there a pedestrian in the picture? (image recognition)
- Is the pedestrian waiting or is he about to cross the street? (video processing)

Medical imaging

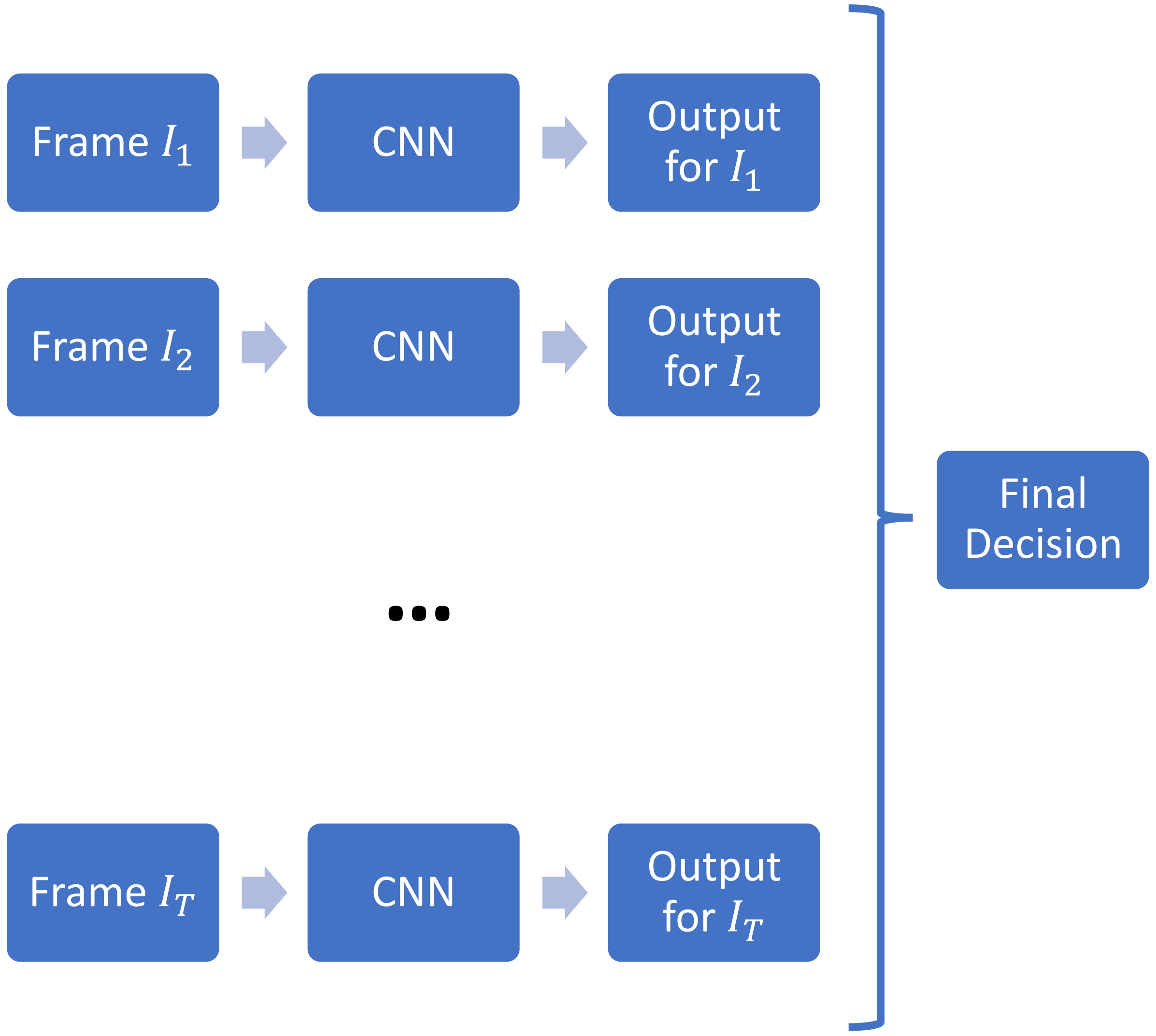
- Is there cancer? Where are the cancer cells in the image? (image recognition)
- Is the heart beating incorrectly? (video processing)



The simplest video baseline

A simple baseline for video understanding

- Treat a video as a set of frames
- Apply a CNN to each frame independently
- Aggregate frame-level information after each image has been processed
- Predict one label per video



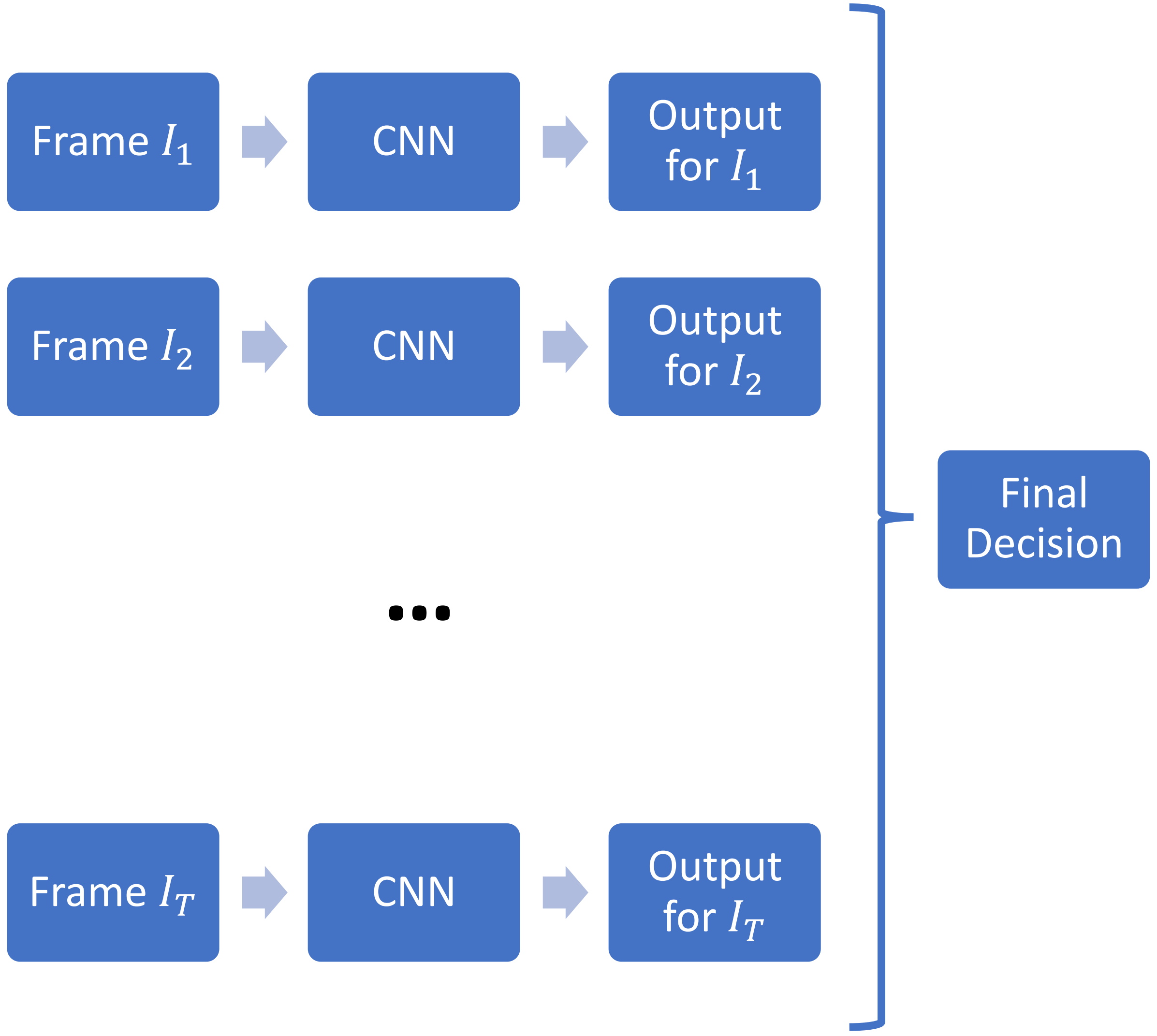
The simplest video baseline

Strengths

- Appearance-based reasoning
- Simple and efficient
- Strong baseline

Limitations

- Treats images independently
- No motion modeling
- No temporal order or sequence
- Cannot distinguish reversed actions or shuffled images

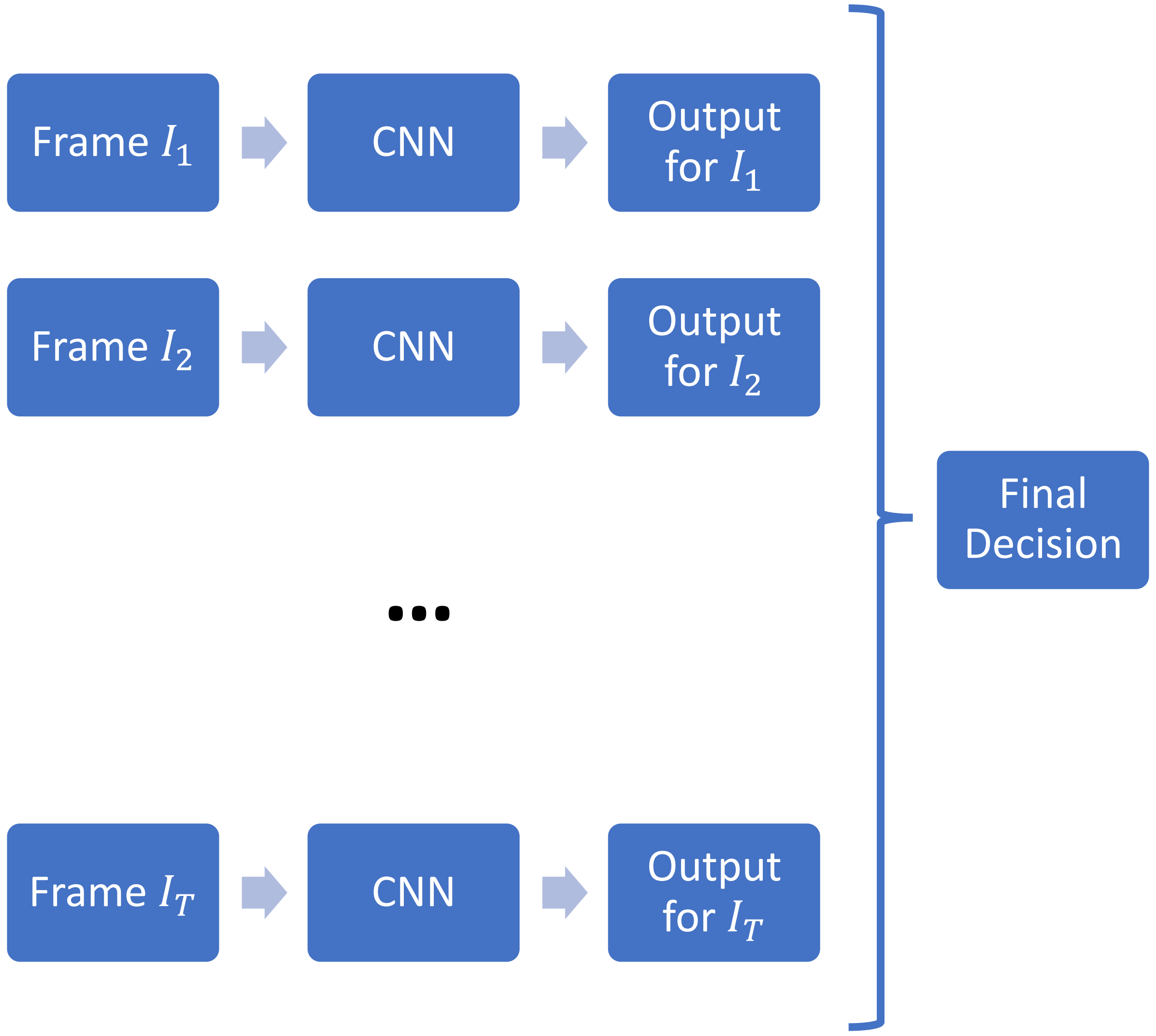


What’s missing?

Correctly processing videos requires to understand:

- Motion direction
- Temporal causality
- Long-term dependencies

In other words: “How do we model **order** to better understand what is happening in the **sequence of images**?”



Quick interlude: about time series

Definition (time series dataset):

A **time series dataset** is a **sequence of data points that are collected over time**, typically at regular intervals.

Time series data can be used to study how a particular variable changes over time, with many applications as stock prices, weather patterns, etc.

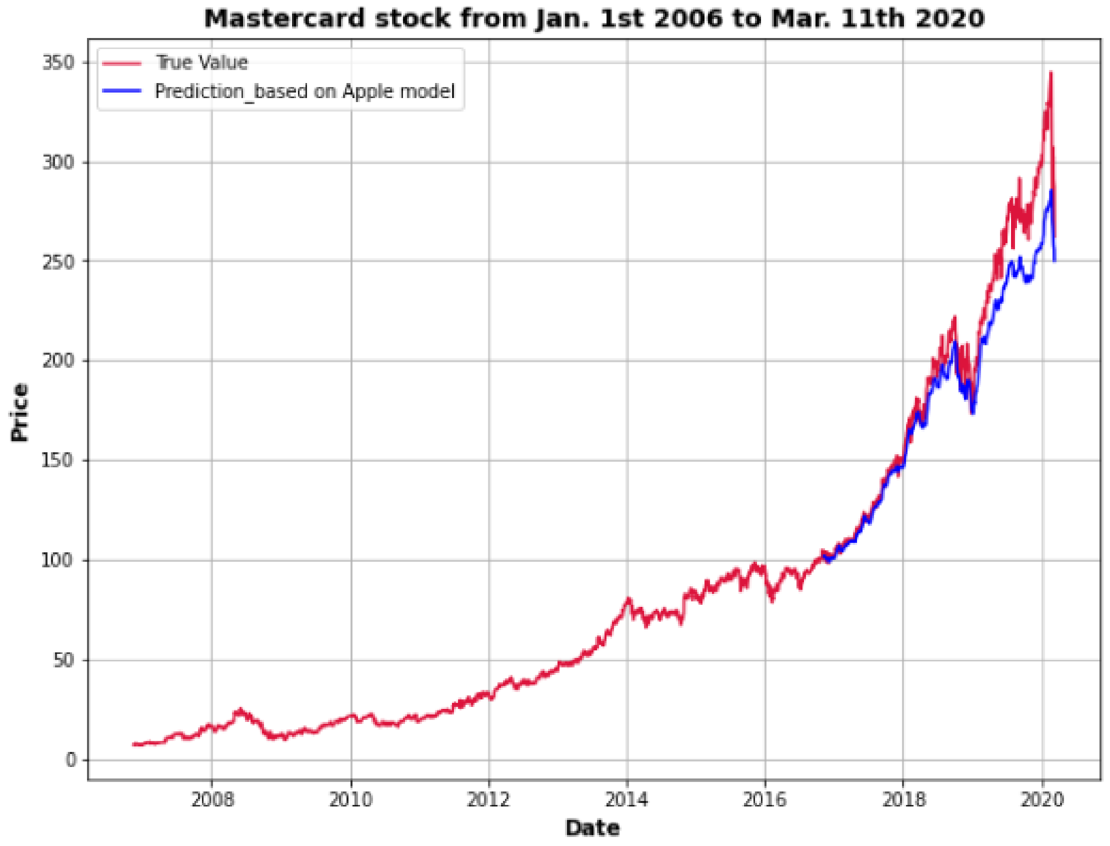


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Quick interlude: about time series

Definition (time series dataset):

A time series **dataset** is a **sequence of data points that are collected over time**, typically at regular intervals.

Time series data can be used to study how a particular variable changes over time, with many applications as stock prices, weather patterns, etc.

- Time series have **structure** (somewhat similar to videos).
- This means that shuffling the data points in a time series has a destructive effect on your dataset, just like shuffling pixels in an image or frames in a video!
- They are usually ordered chronologically, with each data point representing the value of a variable at a specific time.

Quick interlude: about time series

Time series are often represented as tensors of size (T, V).

- With T being the number of time steps (or data points in the time series).
- And values at a given time are assembled in a vector of size V, often set to just 1.

These time series tensors can be seen as the numerical “cousin” of video tensors.

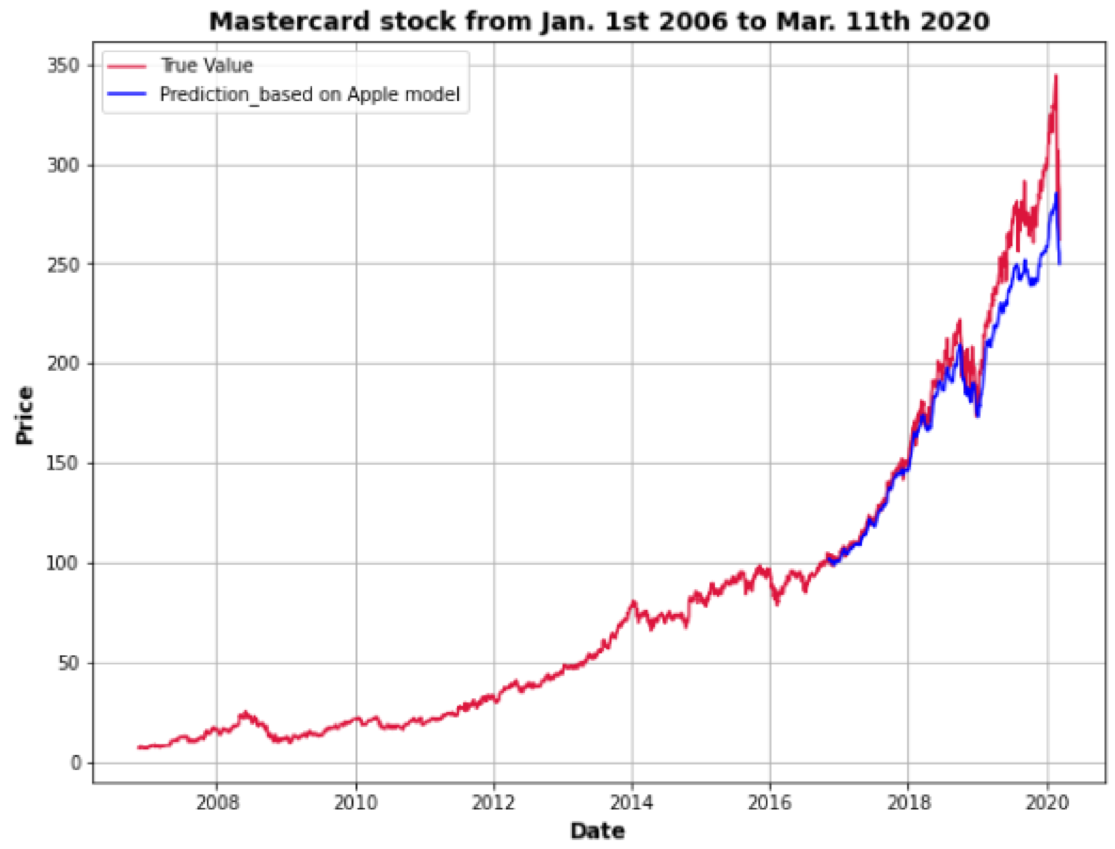
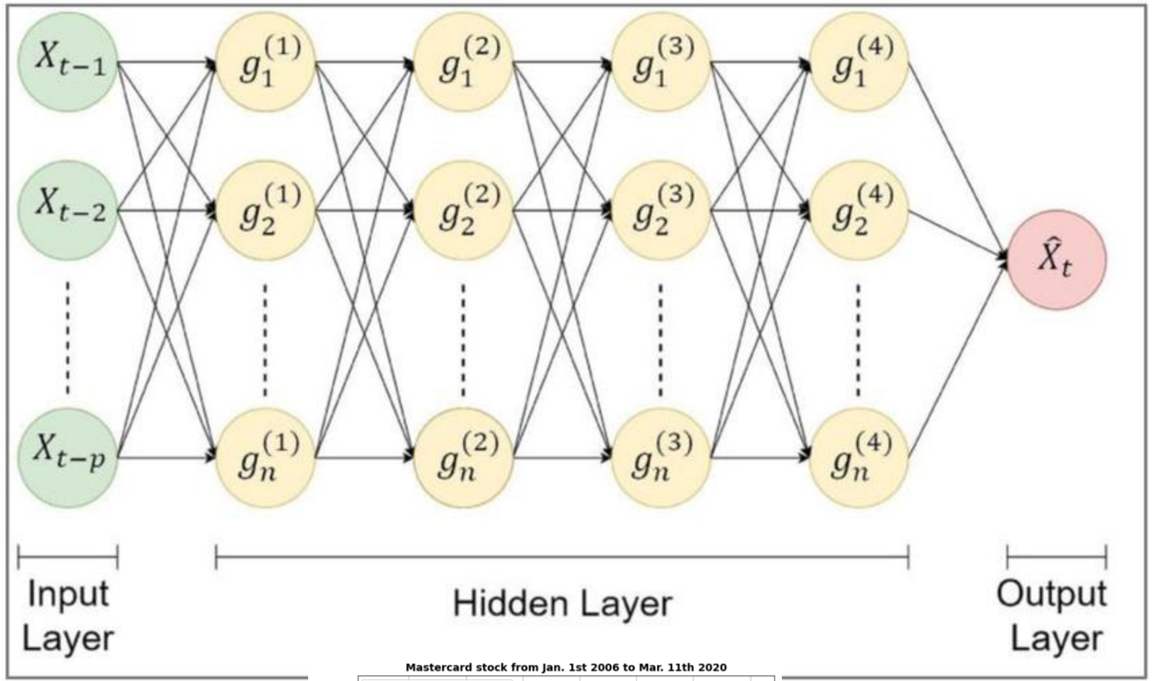


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Why standard NNs fail on time series

Why linear NN are insufficient

- Linear neural networks assume fixed-size input
- They have no notion of order
- No memory of previous values
- Shuffle timesteps → Same input to an MLP (same idea as shuffling pixels in an image or frames in a video, earlier!)



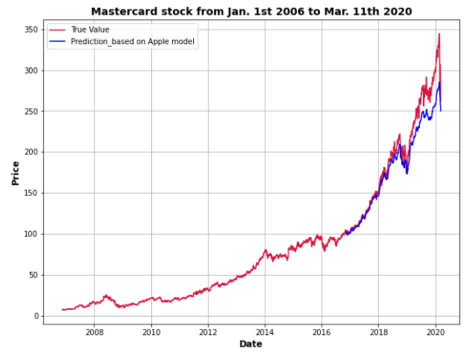
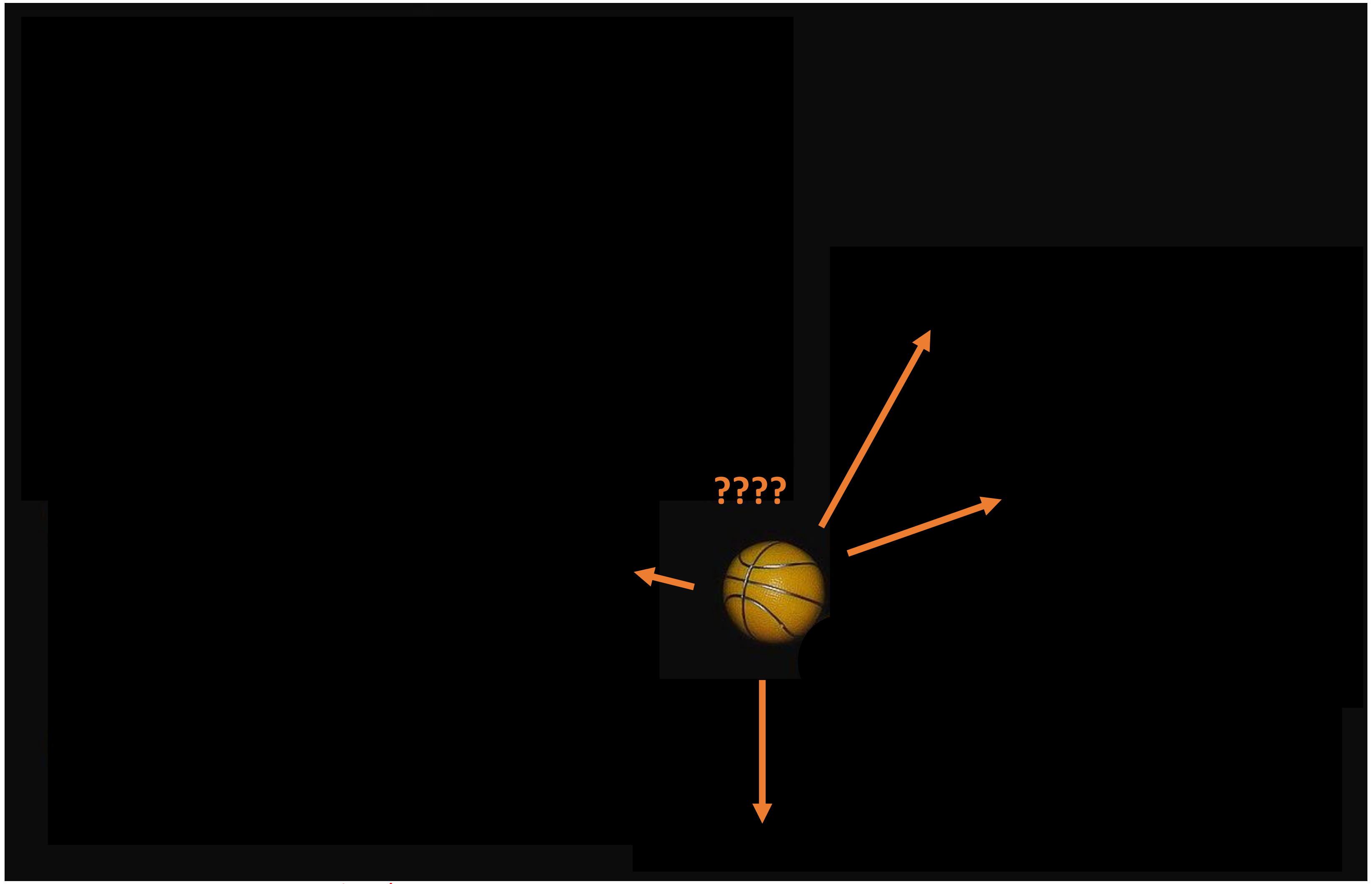


Figure 7. MAST stock price predictions using LSTM trained on AAPL

A visual example for the need of "memory"

Let us pretend that x_t is the position of the basketball at time t.

Looking at the picture on the right, can you guess what the next ball position x_{t+1} will be?

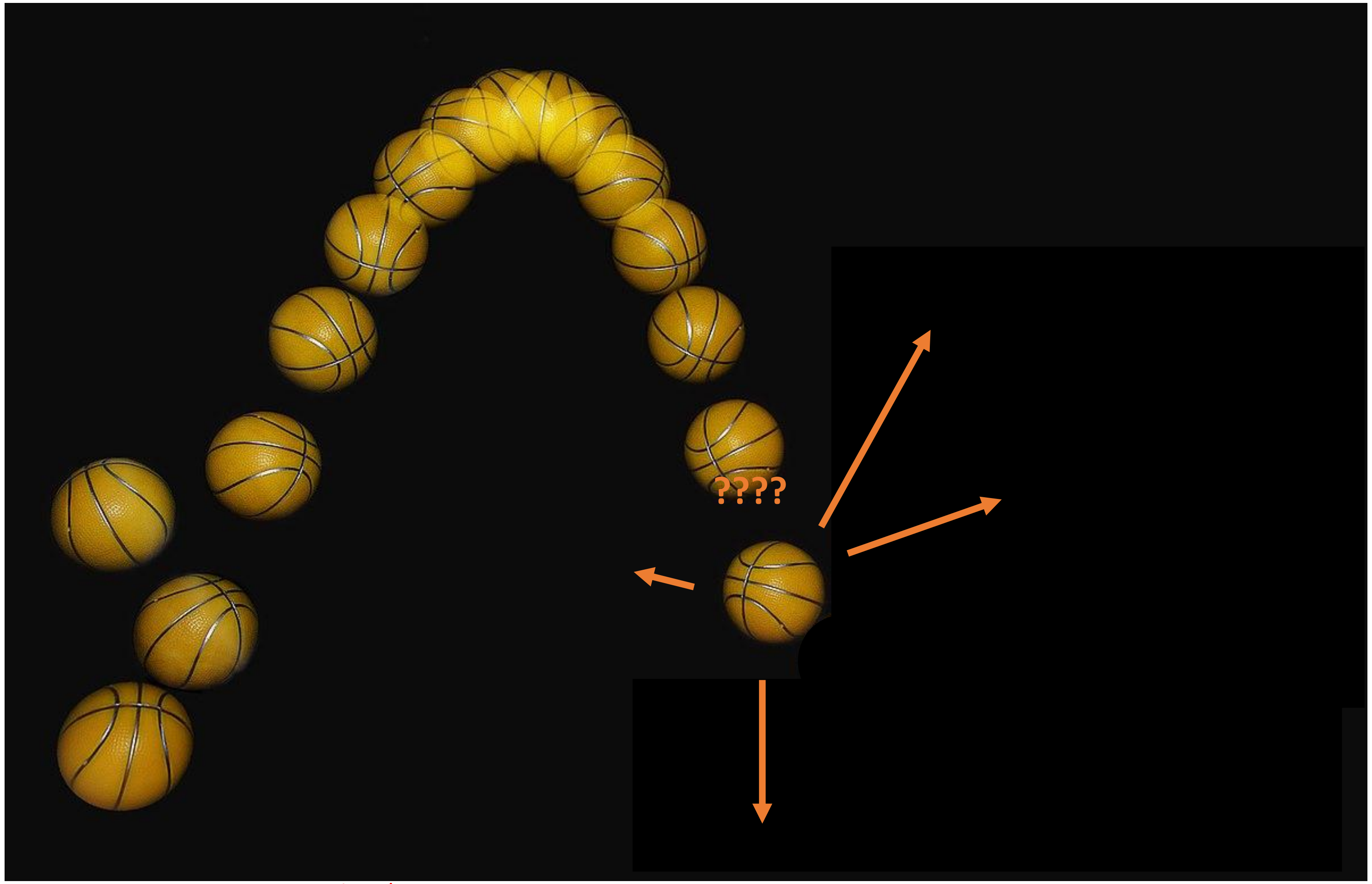


A visual example for the need of "memory"

Let us pretend that x_t is the position of the basketball at time t.

Looking at the picture on the right, can you guess what the next ball position x_{t+1} will be?

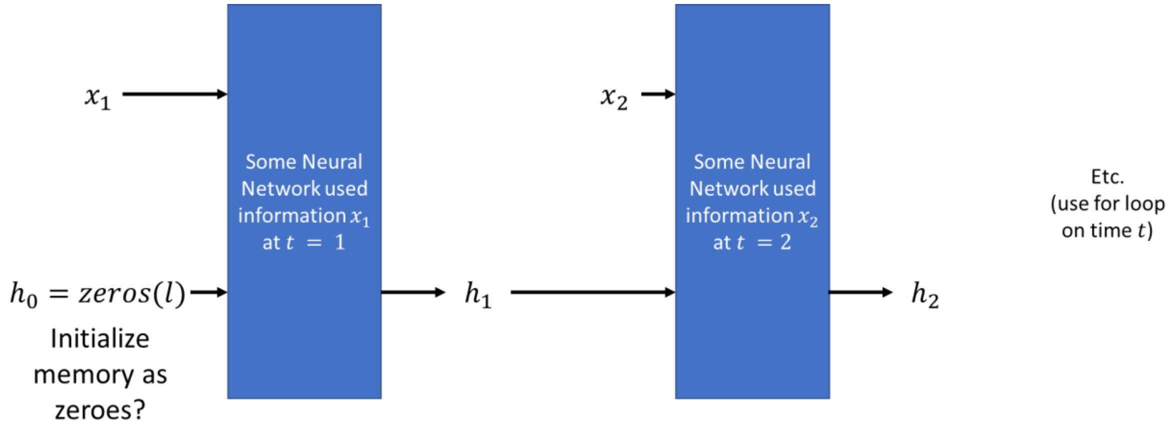
What if I give you some info about the past?



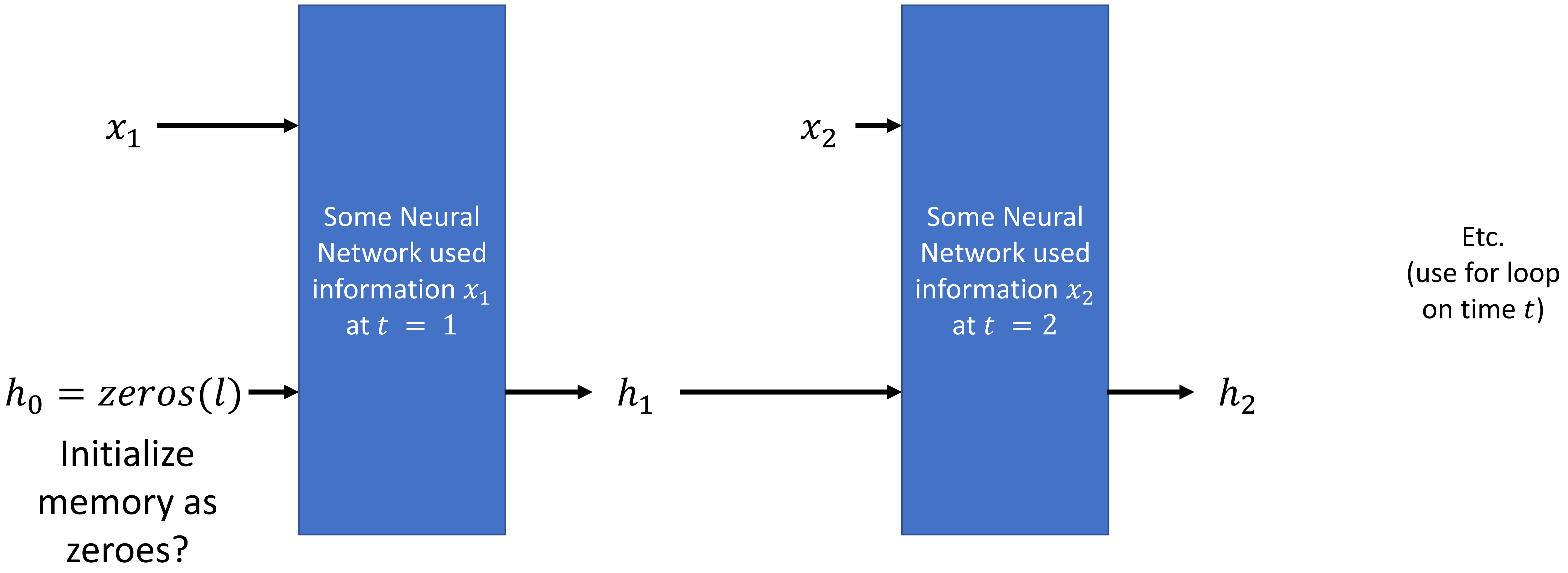
The need for memory and RNNs

Need to introduce the concept of "memory" to our models

- Process sequences one step at a time, as before, but…
- Maintain a hidden state h_t (memory) to remember relevant information from seen pictures
- Store memory in a hidden state vector h_t that summarizes the past
- Leave it to model to decide what and how to remember at each step


x_1
x_2
Some Neural Network used information x_1 at $t = 1$
Some Neural Network used information x_2 at $t = 2$
$h_0 = zeros(l)$
Initialize memory as zeroes?
h_1
h_2
Etc. (use for loop on time t)

The need for memory and RNNs


x_1
x_2
Some Neural Network used information x_1 at $t = 1$
Some Neural Network used information x_2 at $t = 2$
$h_0 = zeros(l)$
Initialize memory as zeroes?
h_1
h_2
Etc. (use for loop on time t)

Defining a Recurrent Neural Network (RNN)

Definition (Recurrent Neural Network):

A **Recurrent Neural Network (or RNN)** is a neural network that:

- Receives an input which consists of **the observation $\boldsymbol{x_t}$ at time $\boldsymbol{t}$**, and a **memory vector $\boldsymbol{h_t}$ computed as one of the Neural Network outputs at time $\boldsymbol{t-1}$**. Its initial value $\boldsymbol{h_0}$ often gets initialized as zeroes.
- Computes an **updated memory vector $\boldsymbol{h_{t+1}}$**, hopefully keeping a memory of what has happened in the previous operations.

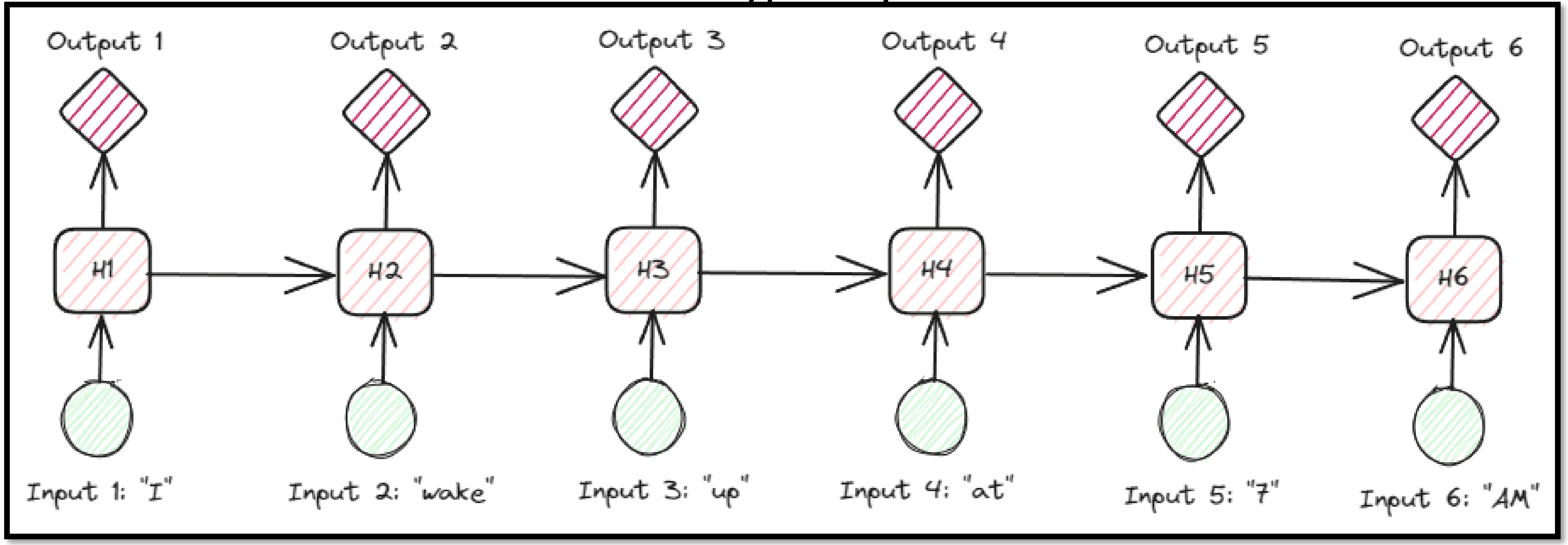
The **RNN** is then used on all datapoints in the time series, using a for loop repeating the forward pass operation on all data points. Once all data points have been seen the **final memory vector value** can be used as the **produced output** from the model.

Want to learn more about RNNs?

- Recurrent Neural Networks are typically used on numerical time series data (think weather, finance, etc.)
- They are also used a lot in Natural Language Processing:
  A sentence can be seen as a time series of words!
- Their intuition, and implementation logic is out-of-scope for this course.
- *(But if curious, bonus slides!)*

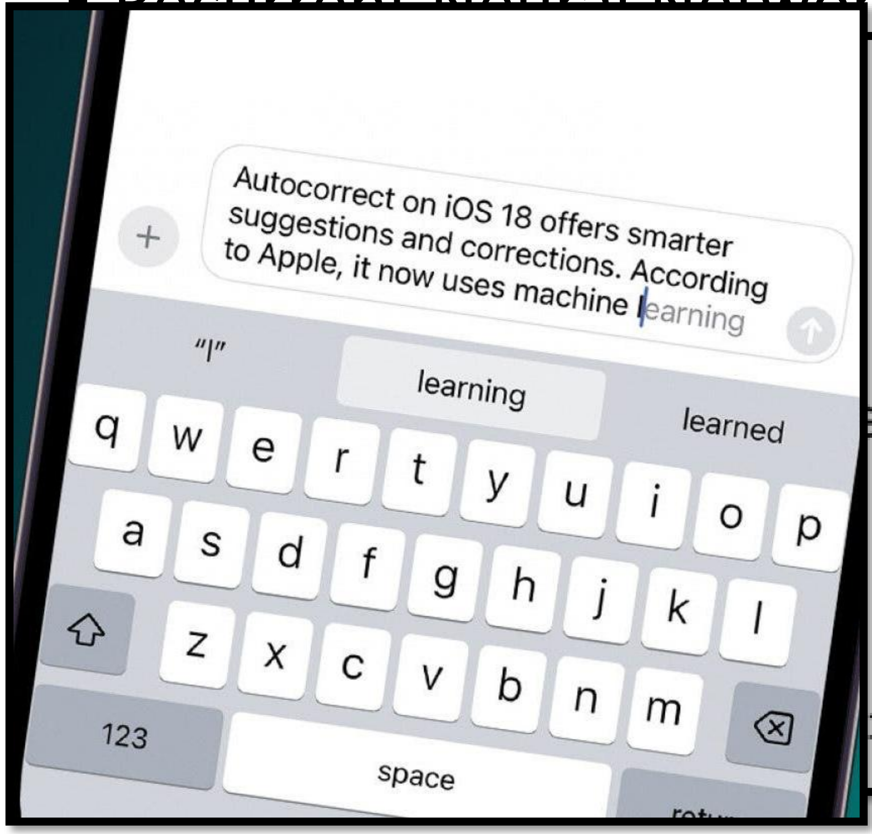
Want to learn more about RNNs?

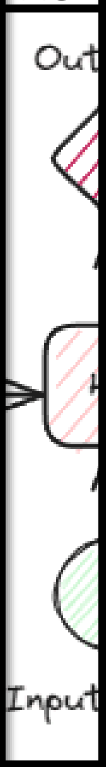
- Recurrent Neural Networks are typically used on numerical time

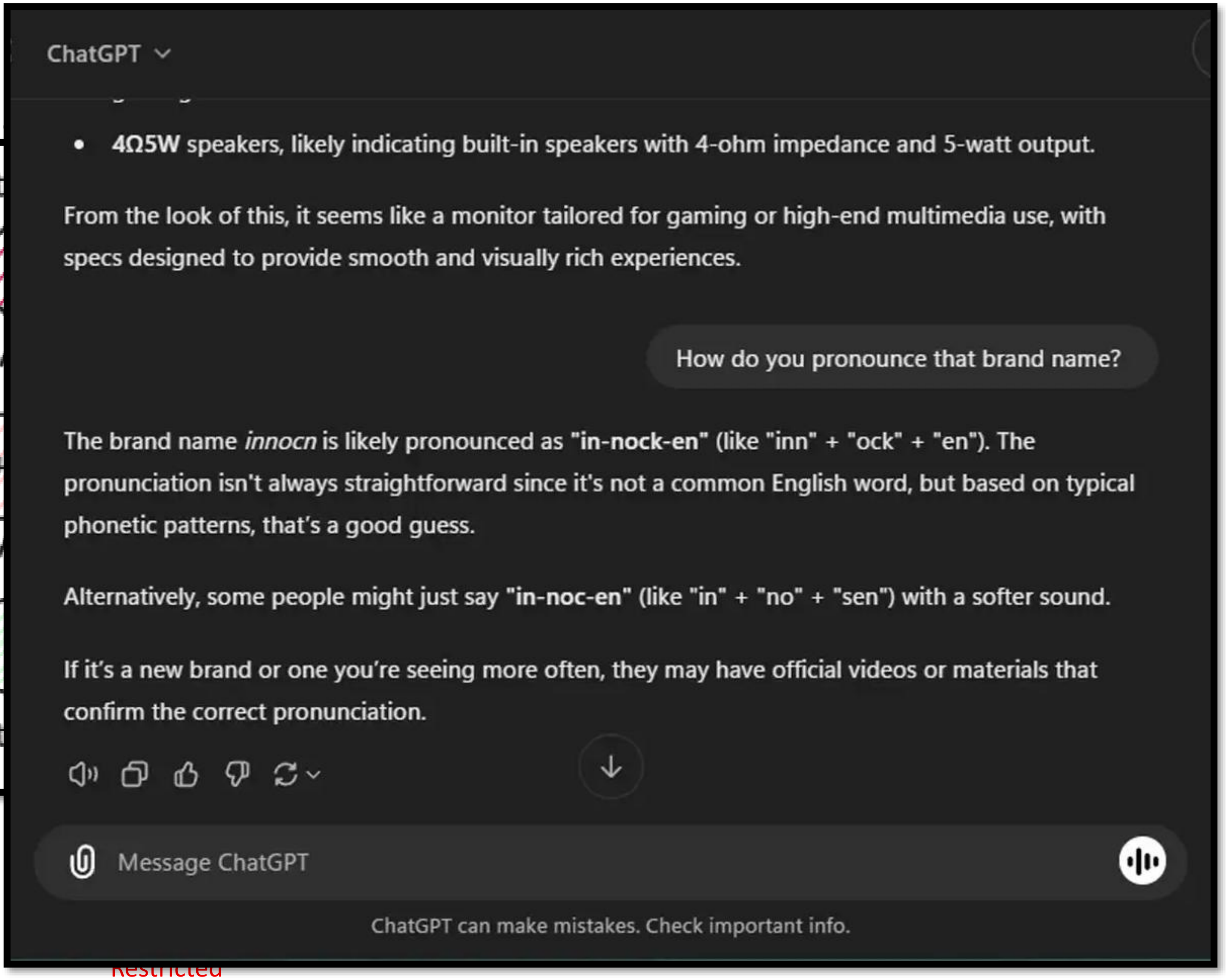


Want to learn more about RNNs?

- Recurrent Neural Networks


Autocorrect on iOS 18 offers smarter suggestions and corrections. According to Apple, it now uses machine learning





- **4Ω5W** speakers, likely indicating built-in speakers with 4-ohm impedance and 5-watt output.

From the look of this, it seems like a monitor tailored for gaming or high-end multimedia use, with specs designed to provide smooth and visually rich experiences.

How do you pronounce that brand name?

The brand name *innocn* is likely pronounced as **"in-nock-en"** (like "inn" + "ock" + "en"). The pronunciation isn't always straightforward since it's not a common English word, but based on typical phonetic patterns, that's a good guess.

Alternatively, some people might just say **"in-noc-en"** (like "in" + "no" + "sen") with a softer sound.

If it's a new brand or one you're seeing more often, they may have official videos or materials that confirm the correct pronunciation.

ChatGPT can make mistakes. Check important info.

More ideas behind memory-based models

But hold on a second, what is the human brain doing when it comes to memory anyway?

- Our brains will decide what is an important information to **remember** over time,
- But it might also choose to **forget** about information that is no longer relevant!
- (Basically, freeing space in the memory of your brain!)

Learn more about memory and the brain, here: https://lesley.edu/article/stages-of-memory

More ideas behind memory-based models

The brain also has **several types of memories**. For instance, it has:

- A **short-term** memory (What did you have for lunch today? How about lunch two months ago?)
- A **long-term** memory (What was your favourite toy as a kid? What was the first video game you ever played?)

The human brain seems capable to decide what makes it into each of these memories, in a **controlled** way.

Learn more about memory and the brain, here: https://lesley.edu/article/stages-of-memory

Introducing LSTM

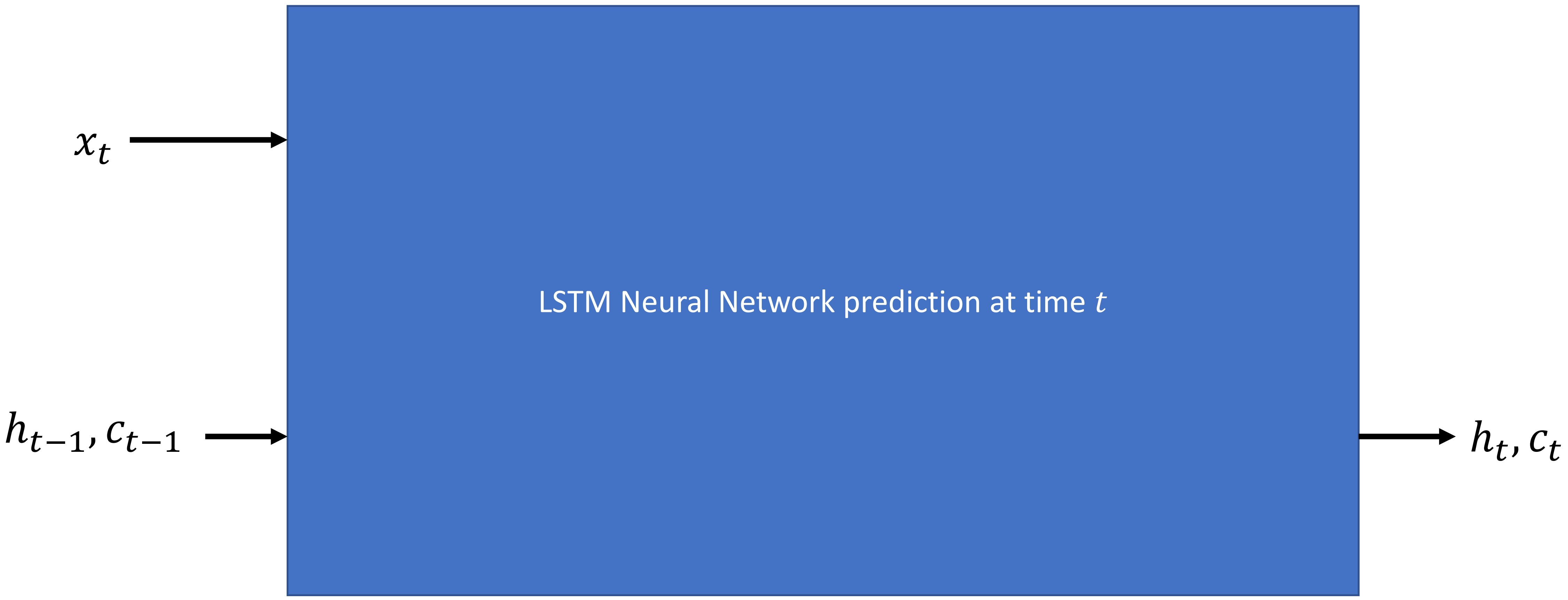
Definition (Long Short-Term Memory models):

LSTMs (Long Short-Term Memory models) are a type of recurrent neural network architecture introduced in [Hochreiter1997] and revised in [Gers2013].

LSTMs is an advanced type of RNNs, which **adds gating mechanisms to selectively update and discard information in the memory vectors**. It also keep track of memory using two memory vectors, instead of one, to track short-term and long-term memory information.

This allows the model to **selectively choose which information from the previous time step to retain and which to discard**, in order to give more importance to recently seen values, for instance.

The idea behind LSTM



Introducing LSTM

The **memory vectors c_t and h_t (often called cell state and hidden state) can be seen as conveyor belts that carries information, or memory, across the network.**

- **The gates can either allow or prevent information from flowing through the belt, <u>mimicking this idea or remembering or forgetting things in the brain</u>.**
- This feature makes LSTMs better suited for modelling complex time series data that exhibit long-term dependencies, as they can selectively choose which information to remember or forget at each time step, depending on the input and the context.
- **Therefore, mimicking the behavior of the human brain!**

Introducing LSTM

Below are the equations used in a LSTM model.

- **Forget Gate:** will be used to decide what information to throw away from the previous cell state, c_{t-1}.

$$f_t = \sigma(W_f\, x_t + U_f\, h_{t-1} + b_f)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Similarly,

- **Input Gate:** will be used to decide what new information to store in the current cell state, c_t.

$$i_t = \sigma(W_i\, x_t + U_i\, h_{t-1} + b_i)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Introducing LSTM

Below are the equations used in a LSTM model.

- **Output Gate:** will be used to decide what information to output from the current cell state, c_t.

$$o_t = \sigma(W_o\, x_t + U_o\, h_{t-1} + b_o)$$

The presence of a sigmoid, forces f_t to have a value in $[0, 1]$.
(Will be important later on).

Important to note:

- All three equations for the forget, input and output gates seem to be following the same logic.
- However, they have their own sets of parameters (weights and biases), which can be trained independently later on.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New cell state:** The new cell state c_t, is then computed by:

$$c_t = f_t\, c_{t-1} + i_t \tanh(W_c\, x_t + U_c\, h_{t-1} + b_c)$$

The new cell state will consist of

- **A proportion $f_t \in [0, 1]$ to keep from the previous cell state c_{t-1}** (= “remembering” part of the previous memory state in the new c_t),
- **A new memory state**, decided as a function of the previous hidden state h_{t-1} and the current observation x_t. A certain proportion $i_t \in [0, 1]$, is then added to the new memory vector c_t.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New cell state:** The new cell state c_t, is then computed by:

$$c_t = f_t \, c_{t-1} + i_t \tanh(W_c \, x_t + U_c \, h_{t-1} + b_c)$$

Question: why use **tanh()** here instead of sigmoid?

- We do not want the values in c_t to be positive real values only. But at the same time, we do not want these values to go crazy (normalize!).
- Using tanh() keeps values in [-1, 1] and the gates keep the values in c_t under control and somewhat normalized.

Introducing LSTM

Below are the equations used in the forward method of a LSTM model.

- **New hidden state:** The new hidden state h_t, is computed by:

$$h_t \ = \ o_t \, tanh(c_t)$$

The new cell state will therefore simply consist of a proportion $o_t \in [0,1]$ of the current cell state c_t.

Important question: Hold on a second, why do we need two parameters, c_t and h_t to keep track of the memory?!

Introducing LSTM

Important question: But wait, why do we need two memory vectors, c_t and h_t to keep track of the memory?!

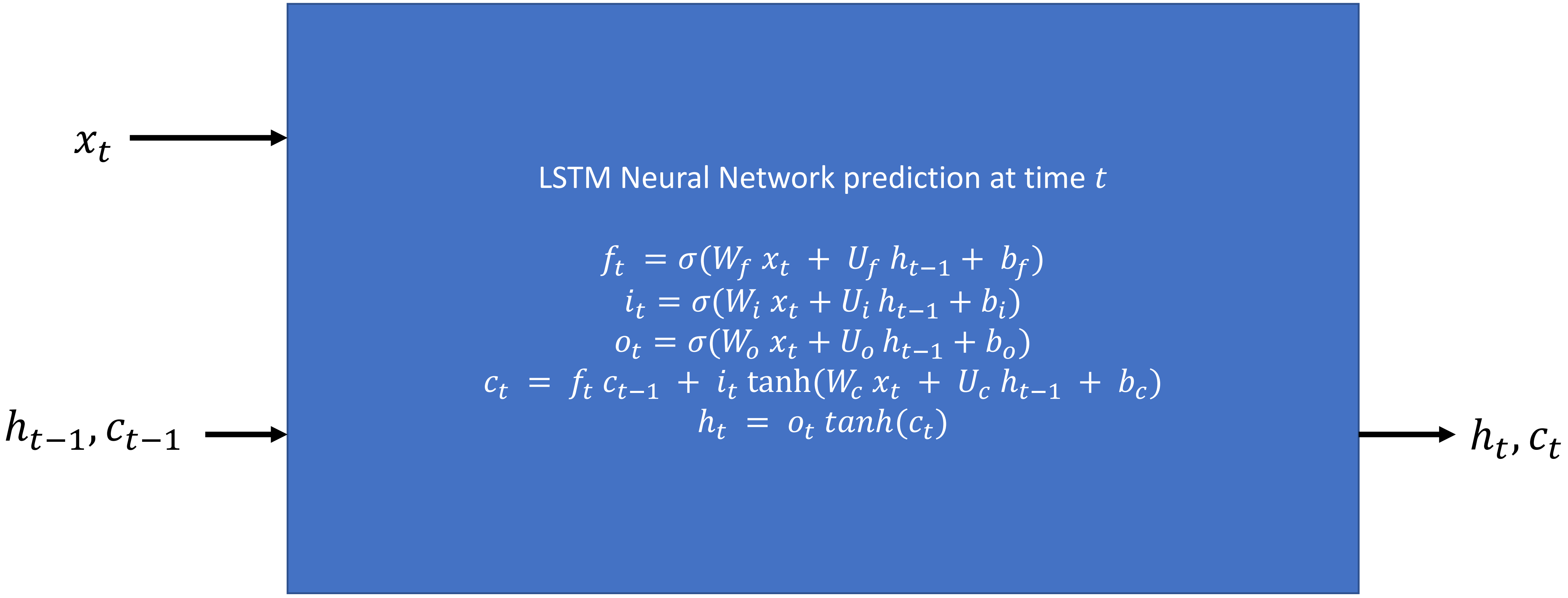
- They serve different purposes in the model…!
- The **cell state** $\boldsymbol{c_t}$ acts as a first type of memory for the LSTM.
- In fact, it is **responsible for retaining long-term information** over time. It is the "state" of the cell and is passed from one time step to the next, allowing the LSTM to **maintain a longer-term memory**.

Introducing LSTM

Important question: But wait, why do we need two memory vectors, c_t and h_t to keep track of the memory?!

- The **hidden state** $\boldsymbol{h_t}$, on the other hand, is used to **output the prediction** and **capture shorter-term dependencies in the data**.
- It serves as a second type of memory that allows the model to **capture patterns in the input data over shorter time periods**.
- Separating both helps the model avoid the vanishing gradient problem and, at the same time, allows it to better capture short- and long-term dependencies in the data.

To recap, our LSTM



LSTM Neural Network prediction at time t

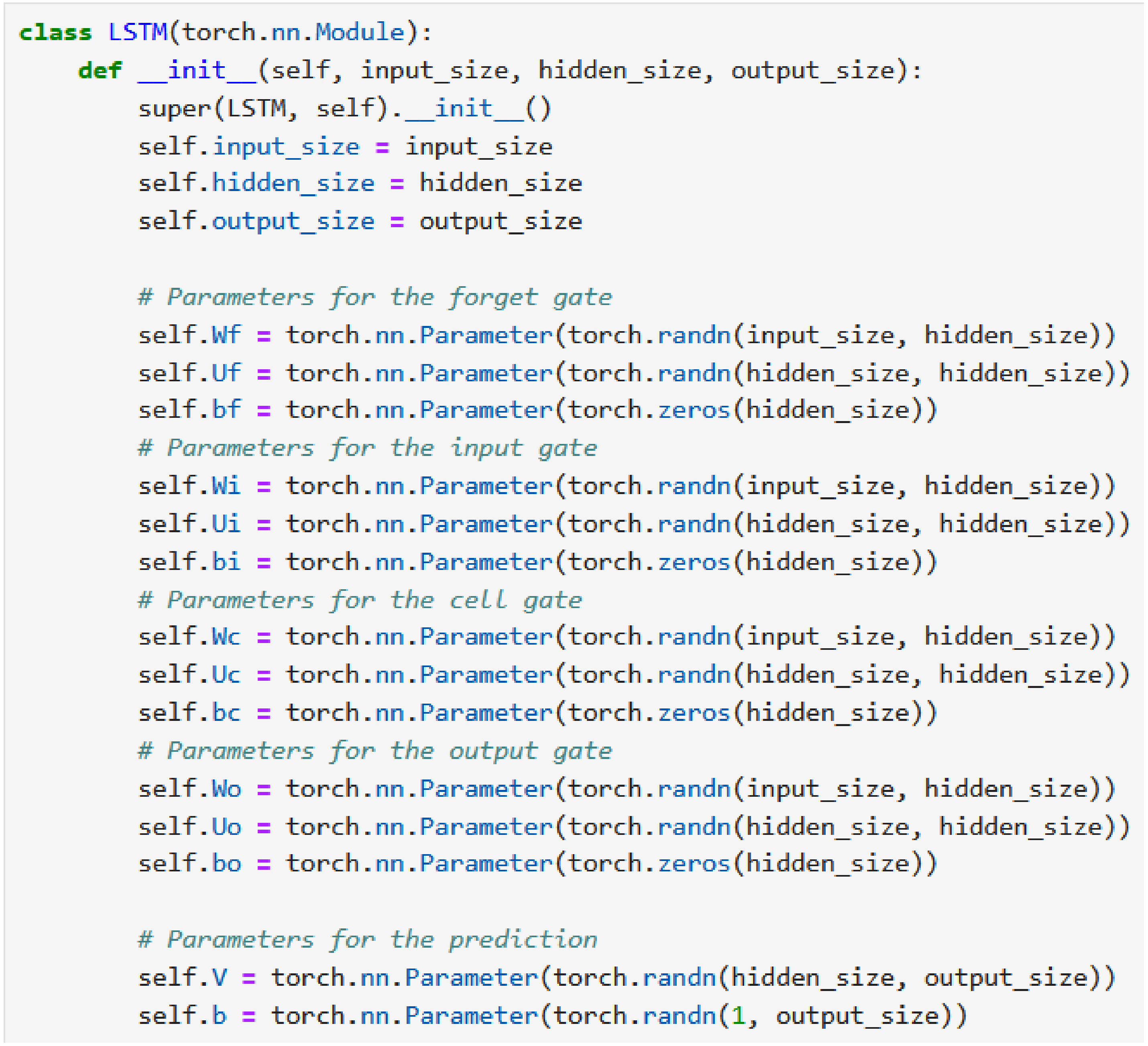
$$
\begin{aligned}
f_t &= \sigma(W_f\, x_t + U_f\, h_{t-1} + b_f) \\
i_t &= \sigma(W_i\, x_t + U_i\, h_{t-1} + b_i) \\
o_t &= \sigma(W_o\, x_t + U_o\, h_{t-1} + b_o) \\
c_t &= f_t\, c_{t-1} + i_t \tanh(W_c\, x_t + U_c\, h_{t-1} + b_c) \\
h_t &= o_t\, tanh(c_t)
\end{aligned}
$$

Building our own LSTM model

Let us start by defining all the weights and biases we need for each of the 6 operations we discussed.

We implement them in the init method of our class as before.

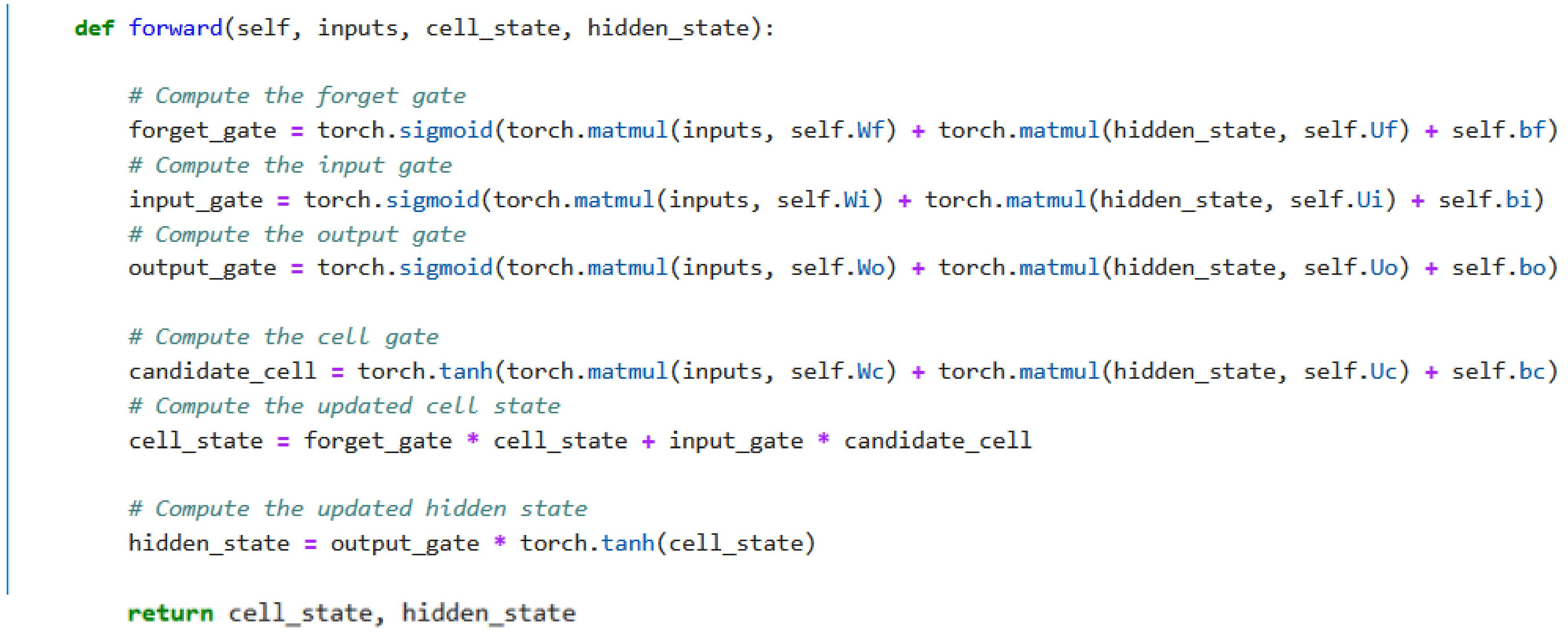
Pay attention to the sizes used for each parameter.



```python
class LSTM(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Parameters for the forget gate
        self.Wf = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uf = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bf = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the input gate
        self.Wi = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Ui = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bi = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the cell gate
        self.Wc = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uc = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bc = torch.nn.Parameter(torch.zeros(hidden_size))
        # Parameters for the output gate
        self.Wo = torch.nn.Parameter(torch.randn(input_size, hidden_size))
        self.Uo = torch.nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.bo = torch.nn.Parameter(torch.zeros(hidden_size))

        # Parameters for the prediction
        self.V = torch.nn.Parameter(torch.randn(hidden_size, output_size))
        self.b = torch.nn.Parameter(torch.randn(1, output_size))
```

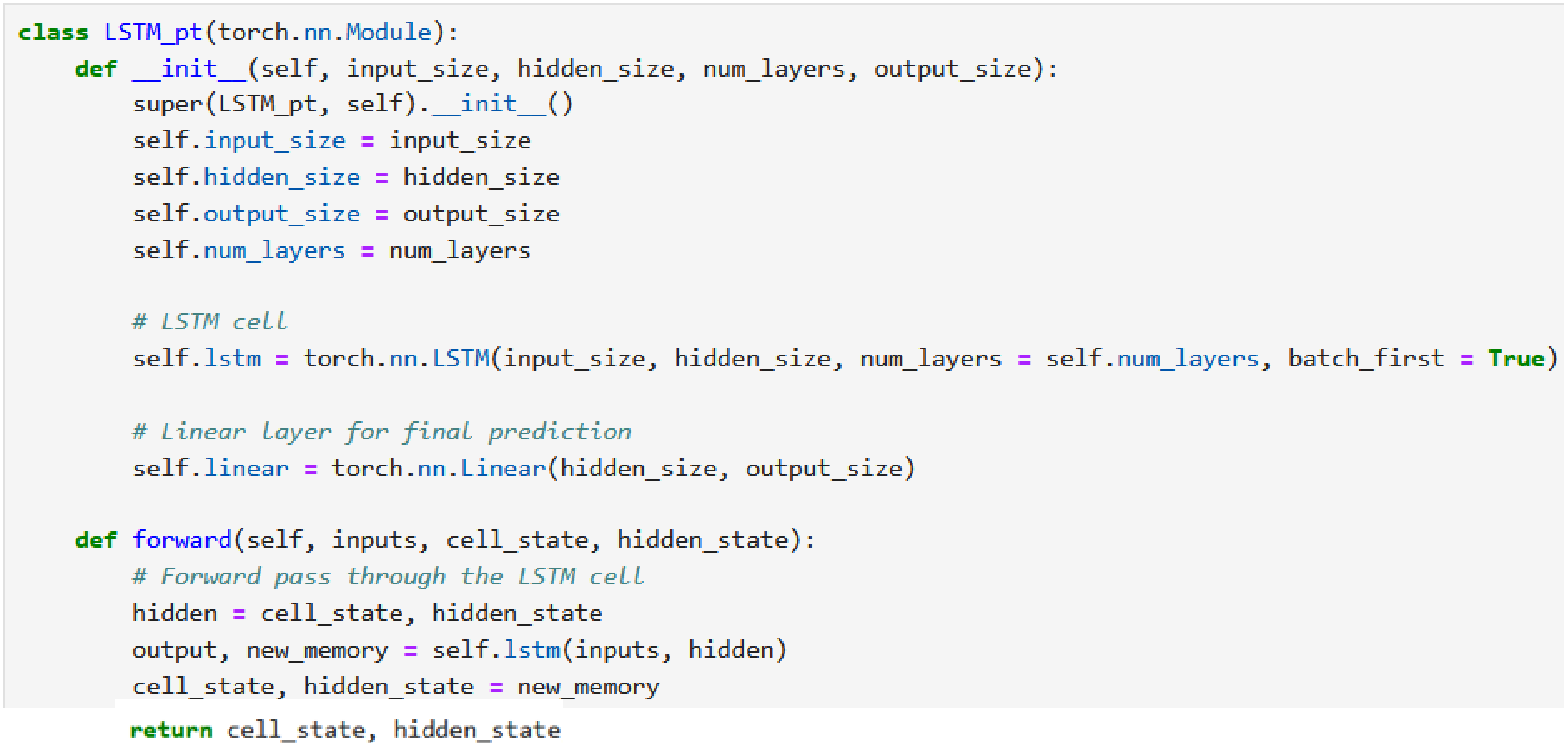
Building our own LSTM

Eventually, compute all 6 LSTM operations in the forward method.



```python
def forward(self, inputs, cell_state, hidden_state):

    # Compute the forget gate
    forget_gate = torch.sigmoid(torch.matmul(inputs, self.Wf) + torch.matmul(hidden_state, self.Uf) + self.bf)
    # Compute the input gate
    input_gate = torch.sigmoid(torch.matmul(inputs, self.Wi) + torch.matmul(hidden_state, self.Ui) + self.bi)
    # Compute the output gate
    output_gate = torch.sigmoid(torch.matmul(inputs, self.Wo) + torch.matmul(hidden_state, self.Uo) + self.bo)

    # Compute the cell gate
    candidate_cell = torch.tanh(torch.matmul(inputs, self.Wc) + torch.matmul(hidden_state, self.Uc) + self.bc)
    # Compute the updated cell state
    cell_state = forget_gate * cell_state + input_gate * candidate_cell

    # Compute the updated hidden state
    hidden_state = output_gate * torch.tanh(cell_state)

    return cell_state, hidden_state
```

The PyTorch LSTM (equivalent to ours, but can be repeated num_layers times in a row)

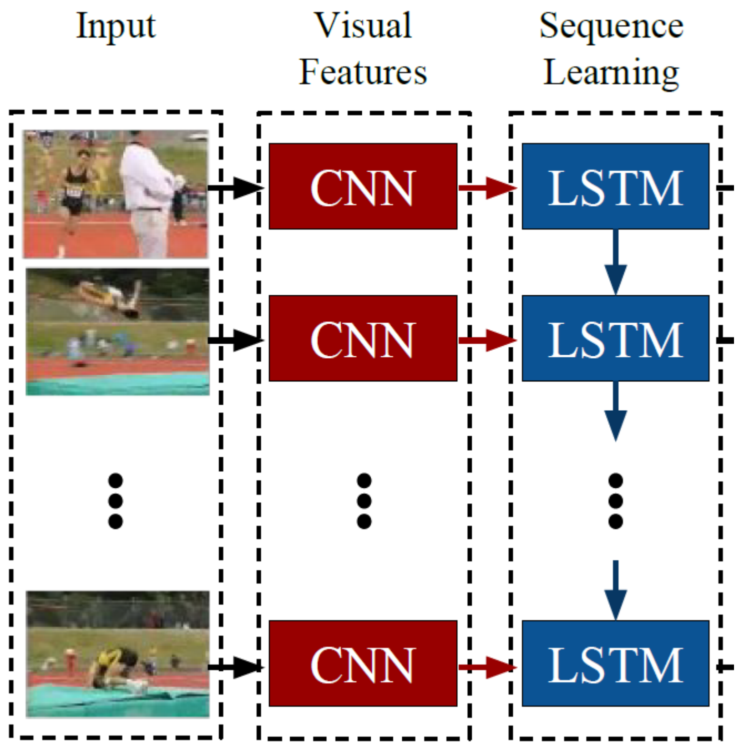


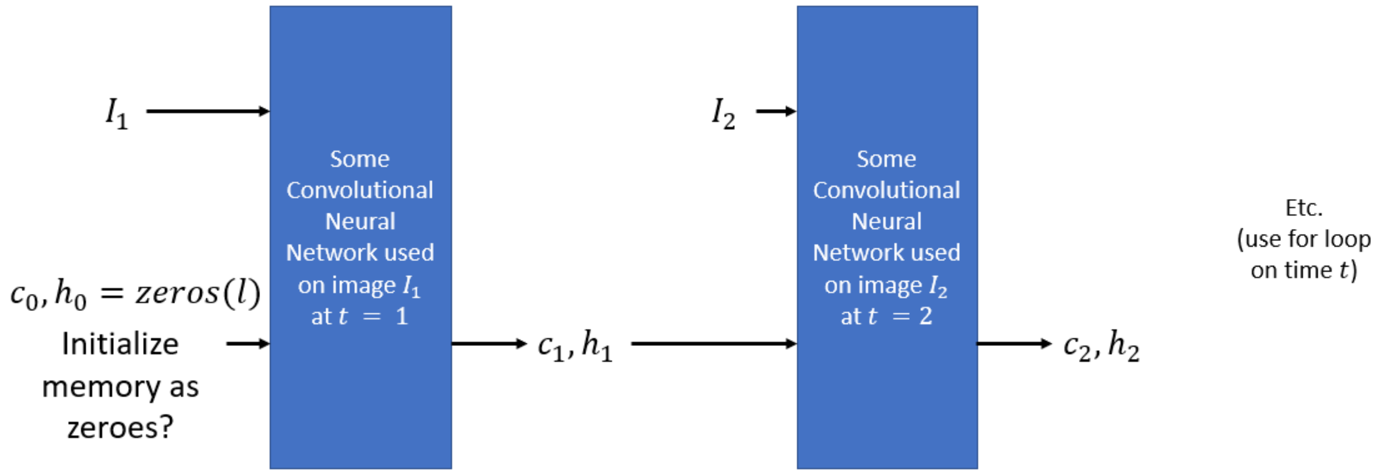
```python
class LSTM_pt(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTM_pt, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        # LSTM cell
        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers = self.num_layers, batch_first = True)

        # Linear layer for final prediction
        self.linear = torch.nn.Linear(hidden_size, output_size)

    def forward(self, inputs, cell_state, hidden_state):
        # Forward pass through the LSTM cell
        hidden = cell_state, hidden_state
        output, new_memory = self.lstm(inputs, hidden)
        cell_state, hidden_state = new_memory
        return cell_state, hidden_state
```

Back to our videos

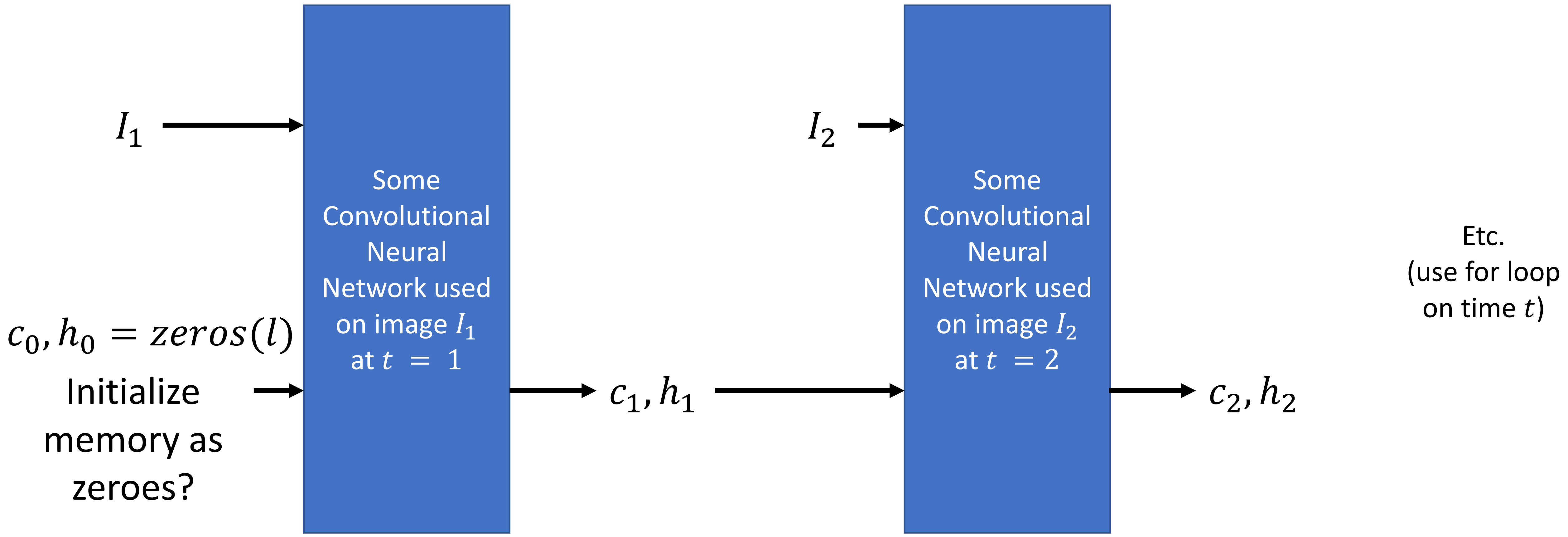
Our videos are defined as **sequences of images.**

- Images have to be processed with CNNs.
- Sequences require RNNs with memory vectors.
- **Idea:** combine both concepts and make a RNN of CNNs!




I_1
I_2
Some Convolutional Neural Network used on image I_1 at $t = 1$
Some Convolutional Neural Network used on image I_2 at $t = 2$
$c_0, h_0 = zeros(l)$
Initialize memory as zeroes?
c_1, h_1
c_2, h_2
Etc. (use for loop on time t)

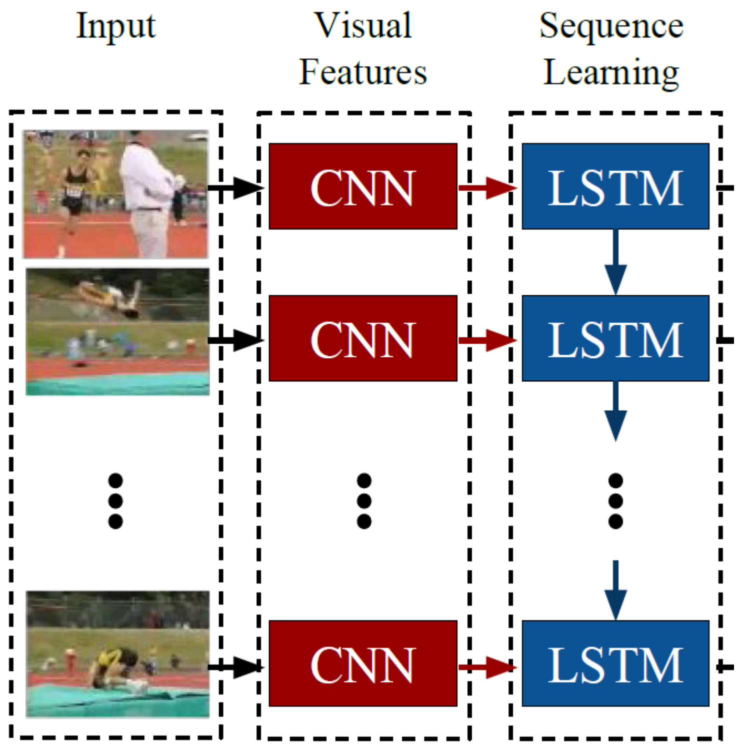
The RNN+CNN idea

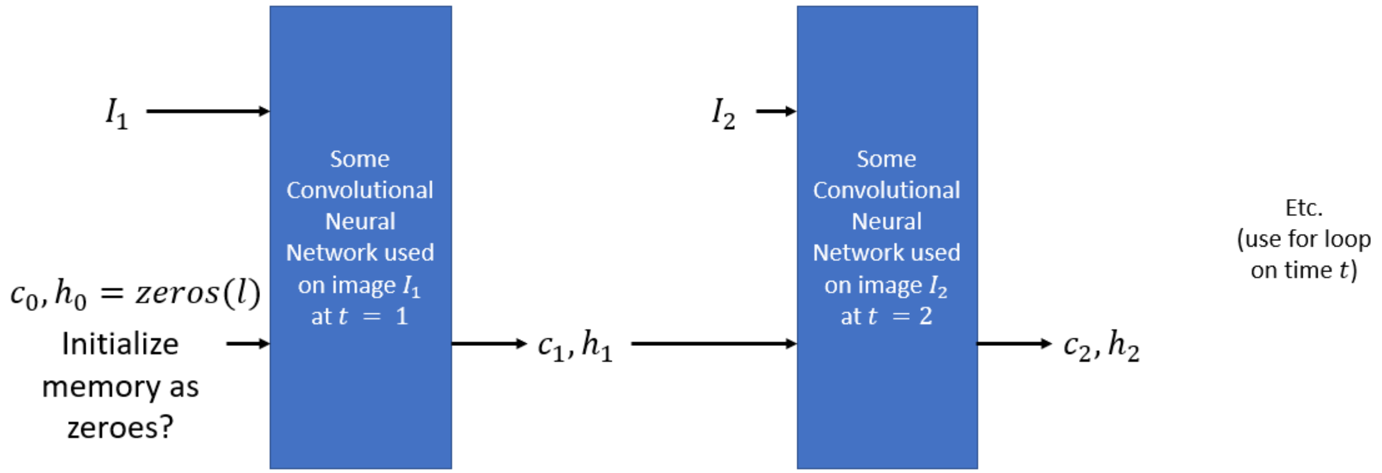

I_1
I_2
Some Convolutional Neural Network used on image I_1 at $t = 1$
Some Convolutional Neural Network used on image I_2 at $t = 2$
$c_0, h_0 = zeros(l)$
Initialize memory as zeroes?
c_1, h_1
c_2, h_2
Etc. (use for loop on time t)

Back to our videos

Our videos are defined as **sequences of images.**

- Images have to be processed with CNNs.
- Sequences require RNNs with memory vectors.
- **Idea:** combine both concepts and make a RNN of CNNs!
- **Eventually, use the final memory vector for predictions!**




I_1
I_2
Some Convolutional Neural Network used on image I_1 at $t = 1$
Some Convolutional Neural Network used on image I_2 at $t = 2$
$c_0, h_0 = zeros(l)$
Initialize memory as zeroes?
c_1, h_1
c_2, h_2
Etc. (use for loop on time t)

Guided implementation
Step 1 – Make a CNN to process each frame

Our CNN will simply consist of:

- 3 Convolutional layers,
- Flatten,
- And a single final linear layer,
- ReLU as activation functions, except for the final layer.

```python
class CNN(nn.Module):
    """

    Frame encoder: (C,H,W) -> (D,)
    A few Conv2d layers then flatten then a Linear.
    """
    def __init__(self, in_channels=3, feat_dim=128):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size = 5, stride = 2, padding = 2)  # -> (16, H/2, W/2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size = 3, stride = 2, padding = 1)          # -> (32, H/4, W/4)
        self.conv3 = nn.Conv2d(32, 64, kernel_size = 3, stride = 2, padding = 1)          # -> (64, H/8, W/8)
        self.fc = nn.Linear(50176, feat_dim)

    def forward(self, frame_batch):
        """

        frame_batch: (B, C, H, W)
        returns: (B, D)
        """
        x = F.relu(self.conv1(frame_batch))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        # Flatten
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        return x
```

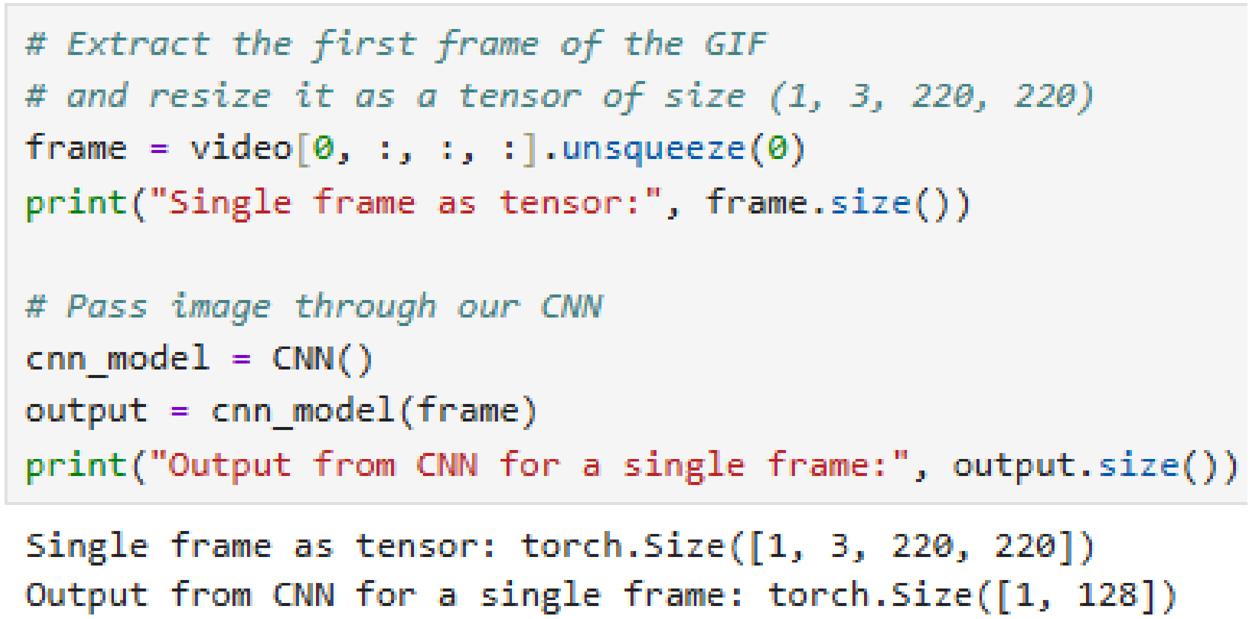
Guided implementation
Step 1 – Make a CNN to process each frame

Our CNN will simply consist of:

- 3 Convolutional layers,
- Flatten,
- And a single final linear layer,
- ReLU as activation functions, except for the final layer.

Can test it on our video!

- Each frame is processed by the CNN and is transformed into a vector of size 128!



```python
# Extract the first frame of the GIF
# and resize it as a tensor of size (1, 3, 220, 220)
frame = video[0, :, :, :].unsqueeze(0)
print("Single frame as tensor:", frame.size())

# Pass image through our CNN
cnn_model = CNN()
output = cnn_model(frame)
print("Output from CNN for a single frame:", output.size())
```

```
Single frame as tensor: torch.Size([1, 3, 220, 220])
Output from CNN for a single frame: torch.Size([1, 128])
```

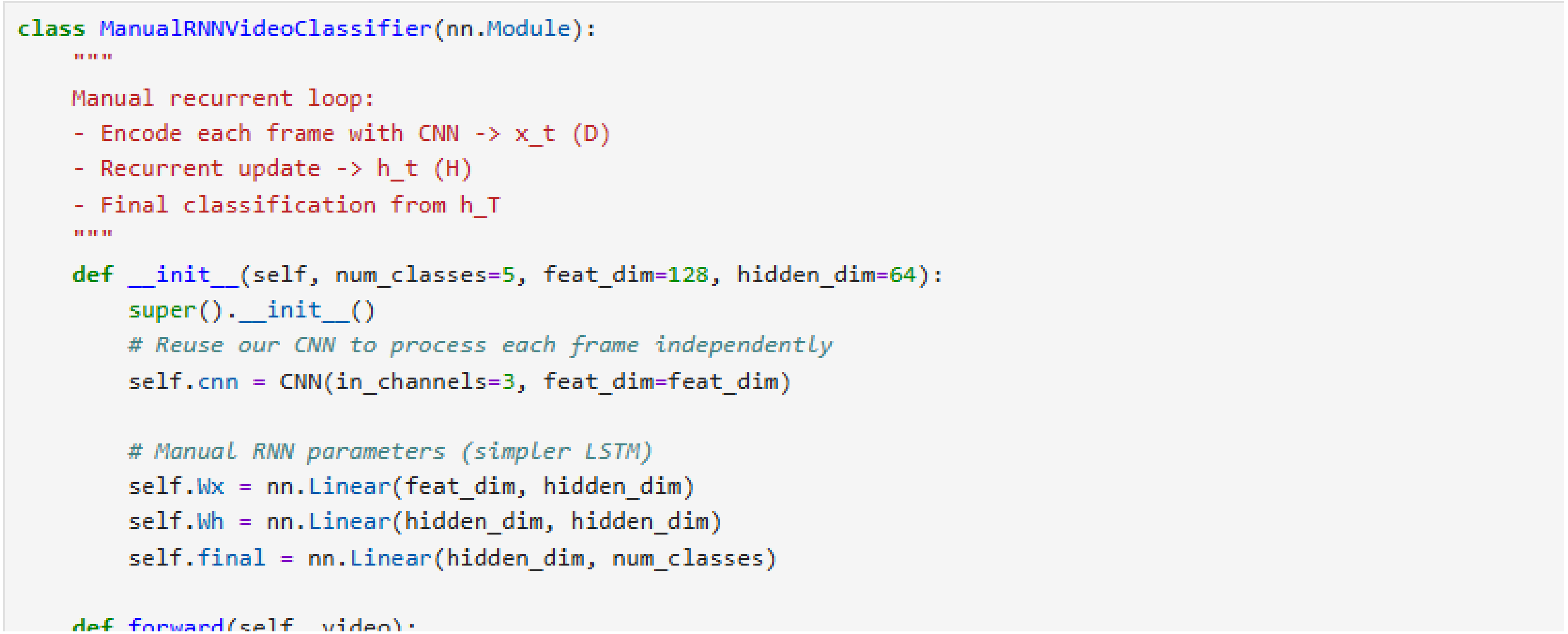
Guided implementation
Step 2 – Make a RNN to process all frames

We will start simple

- Only one memory vector h,
- Simple memory update, following the gating operation from LSTMs,
- Reuse the CNN on each frame of our video, and pass the output of CNNs as inputs for the memory update operation.
- Final memory vector through a linear to produce final output

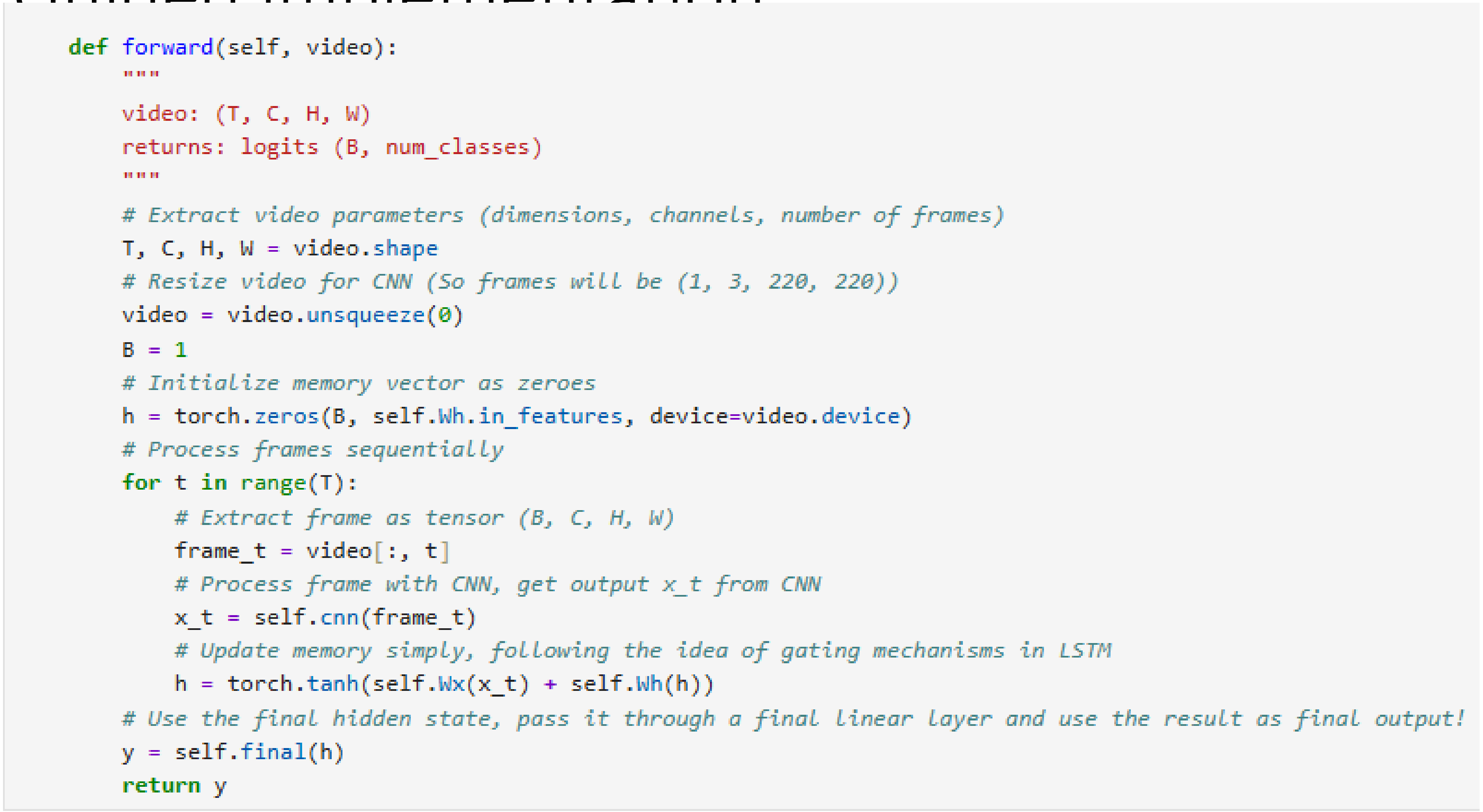
Guided implementation

Step 2 – Make a RNN to process all frames



```python
class ManualRNNVideoClassifier(nn.Module):
    """
    Manual recurrent loop:
    - Encode each frame with CNN -> x_t (D)
    - Recurrent update -> h_t (H)
    - Final classification from h_T
    """
    def __init__(self, num_classes=5, feat_dim=128, hidden_dim=64):
        super().__init__()
        # Reuse our CNN to process each frame independently
        self.cnn = CNN(in_channels=3, feat_dim=feat_dim)

        # Manual RNN parameters (simpler LSTM)
        self.Wx = nn.Linear(feat_dim, hidden_dim)
        self.Wh = nn.Linear(hidden_dim, hidden_dim)
        self.final = nn.Linear(hidden_dim, num_classes)

    def forward(self, video):
```

Guided implementation

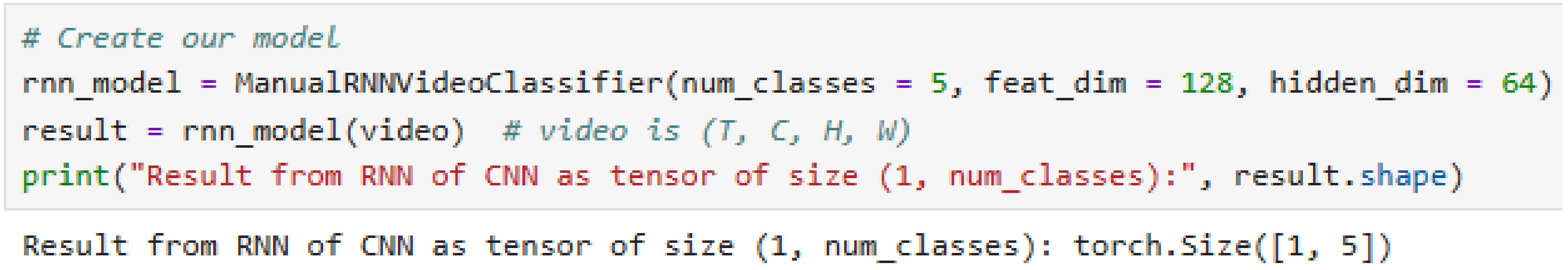


```python
def forward(self, video):
    """
    video: (T, C, H, W)
    returns: logits (B, num_classes)
    """
    # Extract video parameters (dimensions, channels, number of frames)
    T, C, H, W = video.shape
    # Resize video for CNN (So frames will be (1, 3, 220, 220))
    video = video.unsqueeze(0)
    B = 1
    # Initialize memory vector as zeroes
    h = torch.zeros(B, self.Wh.in_features, device=video.device)
    # Process frames sequentially
    for t in range(T):
        # Extract frame as tensor (B, C, H, W)
        frame_t = video[:, t]
        # Process frame with CNN, get output x_t from CNN
        x_t = self.cnn(frame_t)
        # Update memory simply, following the idea of gating mechanisms in LSTM
        h = torch.tanh(self.Wx(x_t) + self.Wh(h))
    # Use the final hidden state, pass it through a final linear layer and use the result as final output!
    y = self.final(h)
    return y
```

Guided implementation
Step 2 – Make a RNN to process all frames

We will start simple

- Only one memory vector h,
- Simple memory update, following the gating operation from LSTMs,
- Reuse the CNN on each frame of our video, and pass the output of CNNs as inputs for the memory update operation.
- Final memory vector through a linear to produce final output



```
# Create our model
rnn_model = ManualRNNVideoClassifier(num_classes = 5, feat_dim = 128, hidden_dim = 64)
result = rnn_model(video)  # video is (T, C, H, W)
print("Result from RNN of CNN as tensor of size (1, num_classes):", result.shape)
```

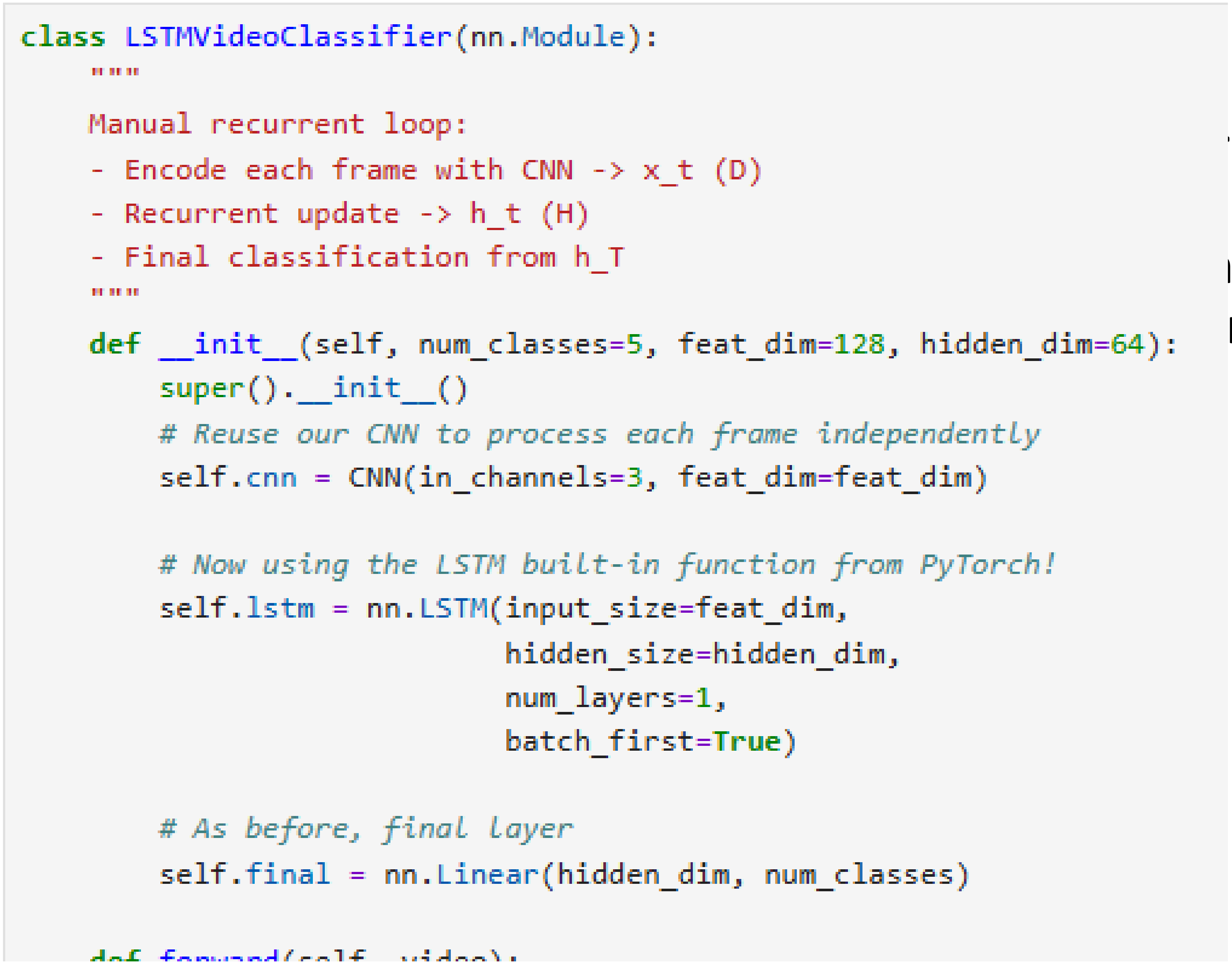
```
Result from RNN of CNN as tensor of size (1, num_classes): torch.Size([1, 5])
```

Guided implementation
Step 2bis – Make a LSTM to process all frames

Later, we can use the built-in LSTM function from PyTorch, to make a more advanced RNN using the LSTM ideas, and multiple memory vectors to keep track of long-term and short-term memory.

Guide
Step 2 ames

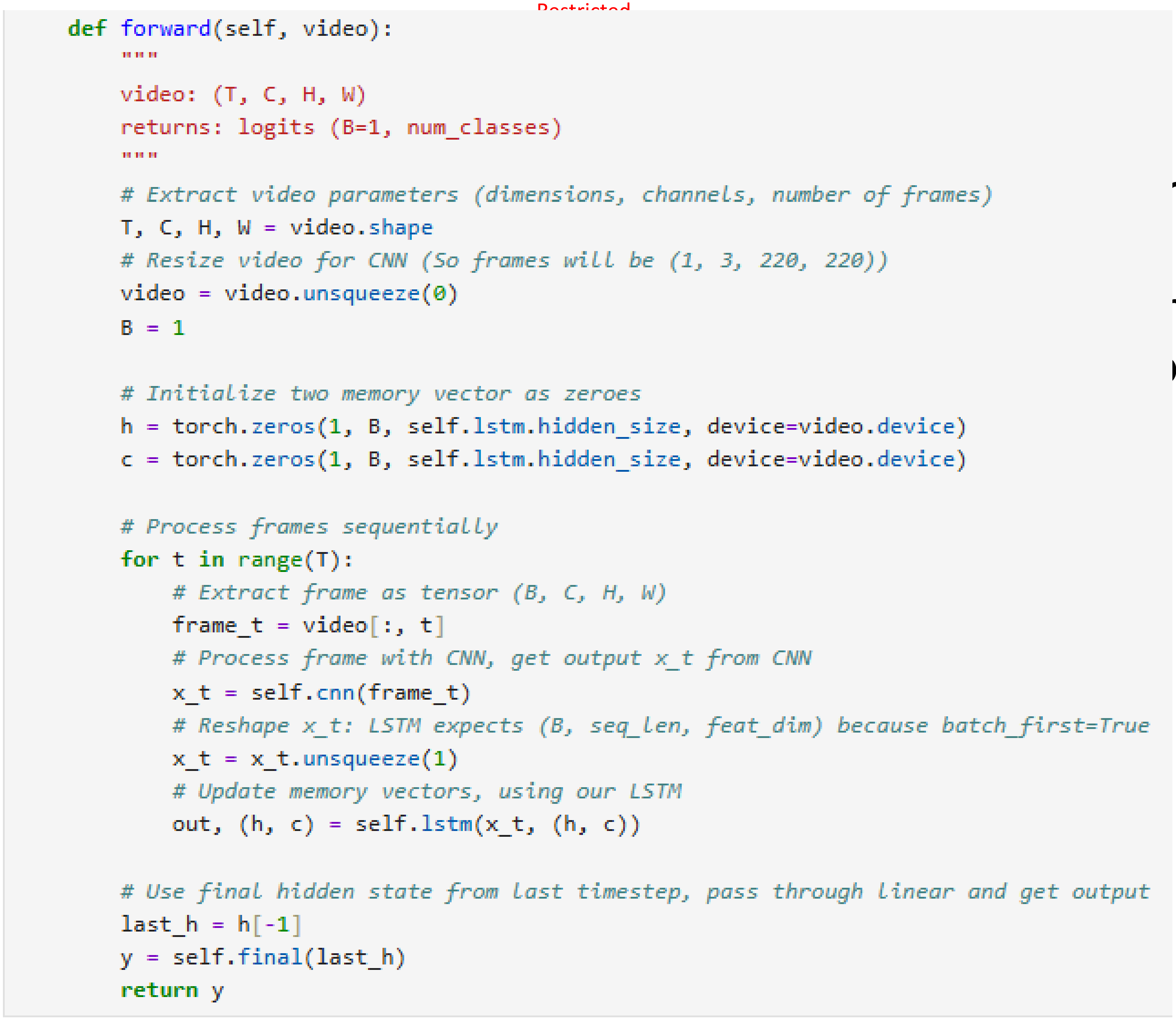
Later, we ake a
more ad ry
vectors t



```python
class LSTMVideoClassifier(nn.Module):
    """
    Manual recurrent loop:
    - Encode each frame with CNN -> x_t (D)
    - Recurrent update -> h_t (H)
    - Final classification from h_T
    """
    def __init__(self, num_classes=5, feat_dim=128, hidden_dim=64):
        super().__init__()
        # Reuse our CNN to process each frame independently
        self.cnn = CNN(in_channels=3, feat_dim=feat_dim)

        # Now using the LSTM built-in function from PyTorch!
        self.lstm = nn.LSTM(input_size=feat_dim,
                            hidden_size=hidden_dim,
                            num_layers=1,
                            batch_first=True)

        # As before, final layer
        self.final = nn.Linear(hidden_dim, num_classes)

    def forward(self, video):
```

Guide
Step 2 ... rames

Later, we ... make a more adv ... ory vectors to

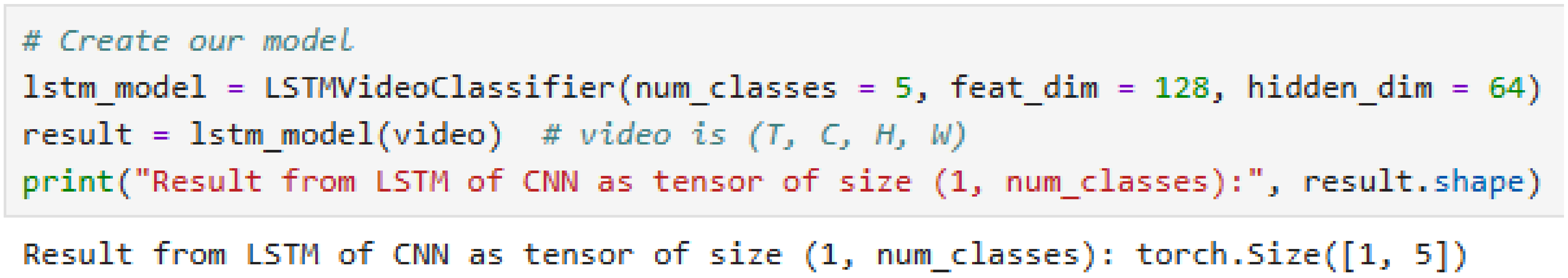


```
def forward(self, video):
    """
    video: (T, C, H, W)
    returns: logits (B=1, num_classes)
    """
    # Extract video parameters (dimensions, channels, number of frames)
    T, C, H, W = video.shape
    # Resize video for CNN (So frames will be (1, 3, 220, 220))
    video = video.unsqueeze(0)
    B = 1

    # Initialize two memory vector as zeroes
    h = torch.zeros(1, B, self.lstm.hidden_size, device=video.device)
    c = torch.zeros(1, B, self.lstm.hidden_size, device=video.device)

    # Process frames sequentially
    for t in range(T):
        # Extract frame as tensor (B, C, H, W)
        frame_t = video[:, t]
        # Process frame with CNN, get output x_t from CNN
        x_t = self.cnn(frame_t)
        # Reshape x_t: LSTM expects (B, seq_len, feat_dim) because batch_first=True
        x_t = x_t.unsqueeze(1)
        # Update memory vectors, using our LSTM
        out, (h, c) = self.lstm(x_t, (h, c))

    # Use final hidden state from last timestep, pass through linear and get output
    last_h = h[-1]
    y = self.final(last_h)
    return y
```

Guided implementation
Step 2bis – Make a LSTM to process all frames

Later, we can use the built-in LSTM function from PyTorch, to make a more advanced RNN using the LSTM ideas, and multiple memory vectors to keep track of long-term and short-term memory.

Works!



```python
# Create our model
lstm_model = LSTMVideoClassifier(num_classes = 5, feat_dim = 128, hidden_dim = 64)
result = lstm_model(video)  # video is (T, C, H, W)
print("Result from LSTM of CNN as tensor of size (1, num_classes):", result.shape)
```

```
Result from LSTM of CNN as tensor of size (1, num_classes): torch.Size([1, 5])
```

Guided implementation
Step 3 – Make a trainer function

Of course, later on, we would need to find a dataset containing multiple videos to train our model, for instance:

- 10 sec videos, resolution 220 by 220 of MRI of beating hearts
- Each video has a ground truth among N possible classes, e.g.:
    - 0: Normal, no abnormalities
    - 1: Cardiomyopathy
    - 2: Heart Failure
    - 3: Ischemia/Infarcts
    - Etc.

Write datasets, dataloaders and trainer functions as before with MNIST, adjusting tensor sizes so it takes sequences of frames and not images!

Let us call it a break for now

We will continue on the next lecture with another NLP-inspired Computer Vision Model called a Vision Transformer!