Agenda:

1. Vanilla/Simple RNN - code → Vanishing/exploding grads → BBC titles ....
2. LSTMs → 2015 - 
3. GRUs → how, when ....
4. ✓ Code for LSTMs → 100 words → 10-15 words

inshorts

Weights	W, U, V	W, U, V, W', U', V'

Code Implementation

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Dense, Embedding, Input, InputLayer, RNN, SimpleRNN, LSTM, GRU, TimeDistributed
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tensorflow as tf

import string

import nltk
nltk.download('stopwords')

from nltk.corpus import stopwords
stop_words = stopwords.words('english')

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Text cleaning

```
[ ] def data_cleaning(text):

        # Lower the words in the sentence
        cleaned = text.lower()

        # Replace the full stop with a full stop and space
        cleaned = cleaned.replace(".", ". ")

        # Remove the stop words
        tokens = [word for word in cleaned.split() if not word in stop_words]

        # Remove the punctuations
        tokens = [tok.translate(str.maketrans(' ', ' ', string.punctuation)) for tok in tokens]

        # Joining the tokens back to form the sentence
        cleaned = " ".join(tokens)

        # Remove any extra spaces
        cleaned = cleaned.strip()

        return cleaned
```

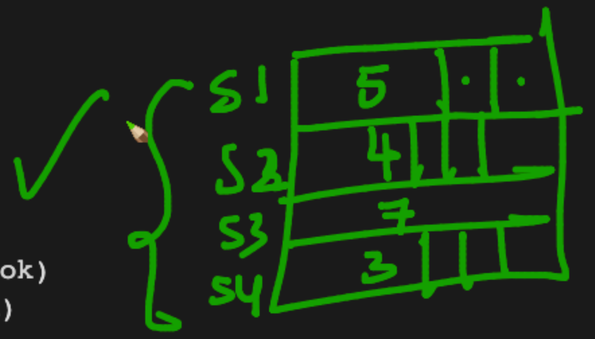
```
[ ] for index, data in tqdm(df.iterrows(), total=df.shape[0]):
        df.loc[index, 'title'] = data_cleaning(data['title'])
```

Text cleaning

```
[ ] def data_cleaning(text):

        # Lower the words in the sentence
        cleaned = text.lower()

        # Replace the full stop with a full stop and space
        cleaned = cleaned.replace(".", ". ")

        # Remove the stop words
        tokens = [word for word in cleaned.split() if not word in stop_words]

        # Remove the punctuations
        tokens = [tok.translate(str.maketrans(' ', ' ', string.punctuation)) for tok in tokens]

        # Joining the tokens back to form the sentence
        cleaned = " ".join(tokens)

        # Remove any extra spaces
        cleaned = cleaned.strip()

        return cleaned
```

```
[ ] for index, data in tqdm(df.iterrows(), total=df.shape[0]):
        df.loc[index, 'title'] = data_cleaning(data['title'])
```

```
def tokenize_and_pad(inp_text, max_len, tok):

    text_seq = tok.texts_to_sequences(inp_text)
    text_seq = pad_sequences(text_seq, maxlen=max_len, padding='post')

    return text_seq

text_tok = Tokenizer()
text_tok.fit_on_texts(train_X)
train_text_X = tokenize_and_pad(inp_text=train_X, max_len=max_sentence_len, tok=text_tok)
test_text_X = tokenize_and_pad(inp_text=test_X, max_len=max_sentence_len, tok=text_tok)
vocab_size = len(text_tok.word_index)+1

print("Overall text vocab size", vocab_size)
```



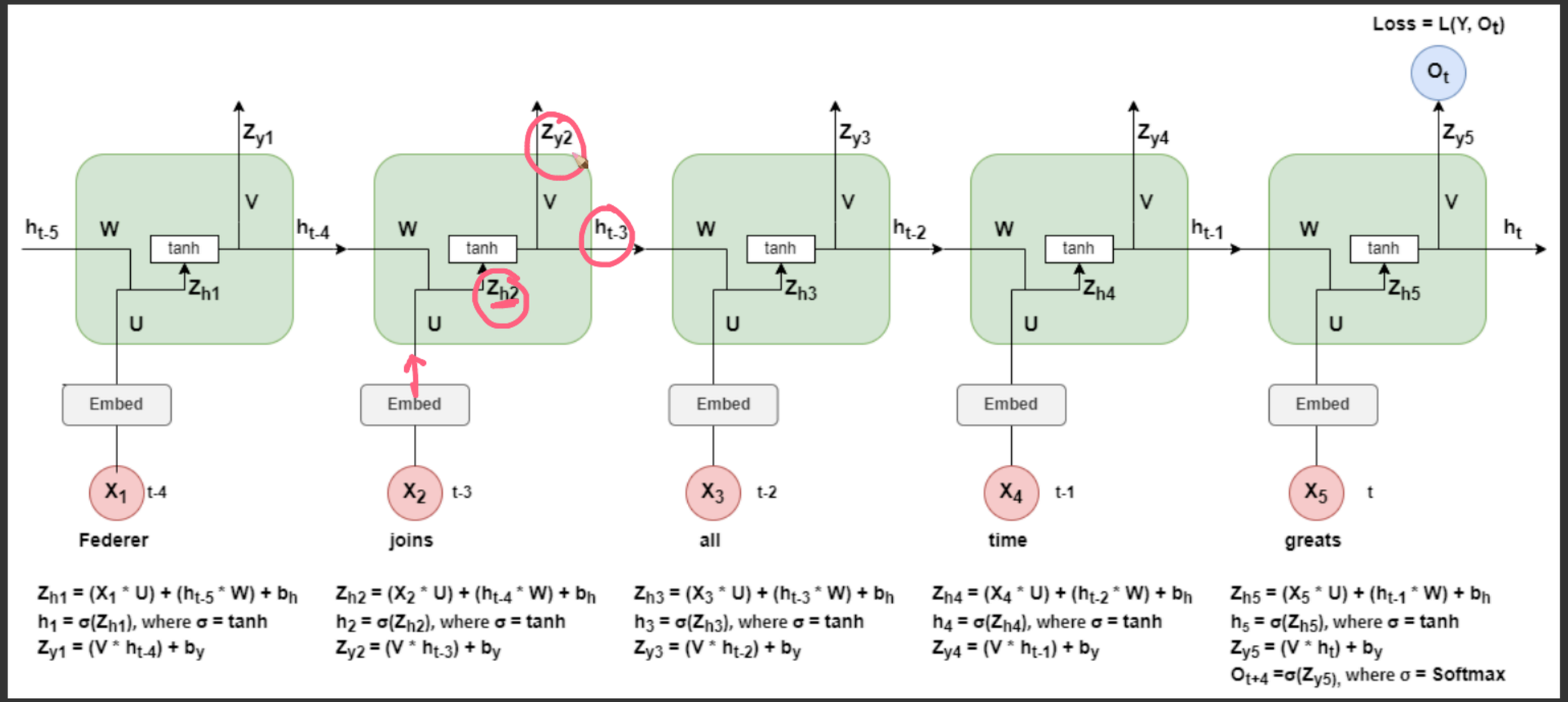

```
Overall text vocab size 3360
```

Choose the latent dimension and embedding dimension

Latent dimension: Dimension of the weight matrix U, V, W

Embedding dimension: Dimension of the word embeddings at the embedding layer

```
[ ] latent_dim=50
    embedding_dim=100
```



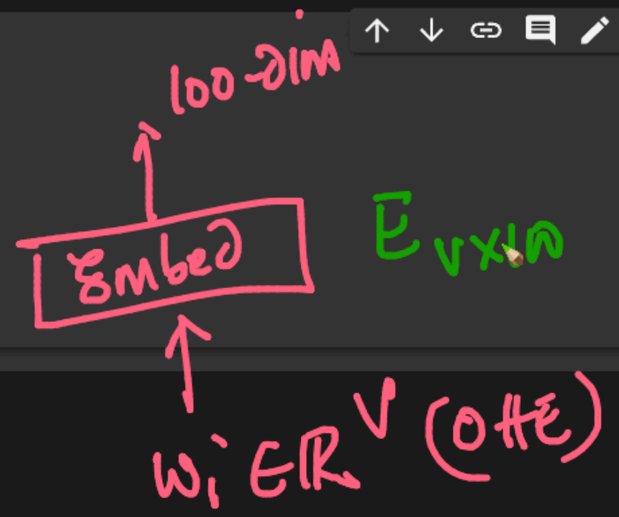
Steps in Forward Propagation

For each word in the input:

```
[ ]  embedding_dim=100
```

Define the RNN model architecture

- The embedding layer with 100 dimension
- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function




```
[ ]  seed=56
     tf.random.set_seed(seed)
     np.random.seed(seed)

     model = Sequential()
     model.add(Embedding(vocab_size, embedding_dim, trainable=True))
     model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
     model.add(Dense(total_classes, activation='softmax'))
     model.summary()
```

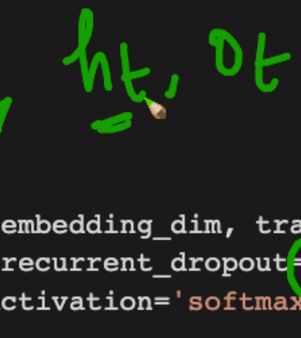
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255
```

```
[ ] embedding_dim=100
```

Define the RNN model architecture

- The embedding layer with 100 dimension
- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
[ ] seed=56
    tf.random.set_seed(seed)
    np.random.seed(seed)

    model = Sequential()
    model.add(Embedding(vocab_size, embedding_dim, trainable=True))
    model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
    model.add(Dense(total_classes, activation='softmax'))
    model.summary()
```




```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255

_________________________________________________________________
```

```
[ ]  embedding_dim=100
```

Define the RNN model architecture

- The embedding layer with 100 dimension
- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
[ ]  seed=56
     tf.random.set_seed(seed)
     np.random.seed(seed)

     model = Sequential()
     model.add(Embedding(vocab_size, embedding_dim, trainable=True))
     model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
     model.add(Dense(total_classes, activation='softmax'))
     model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255
```

```
[ ]  embedding_dim=100
```

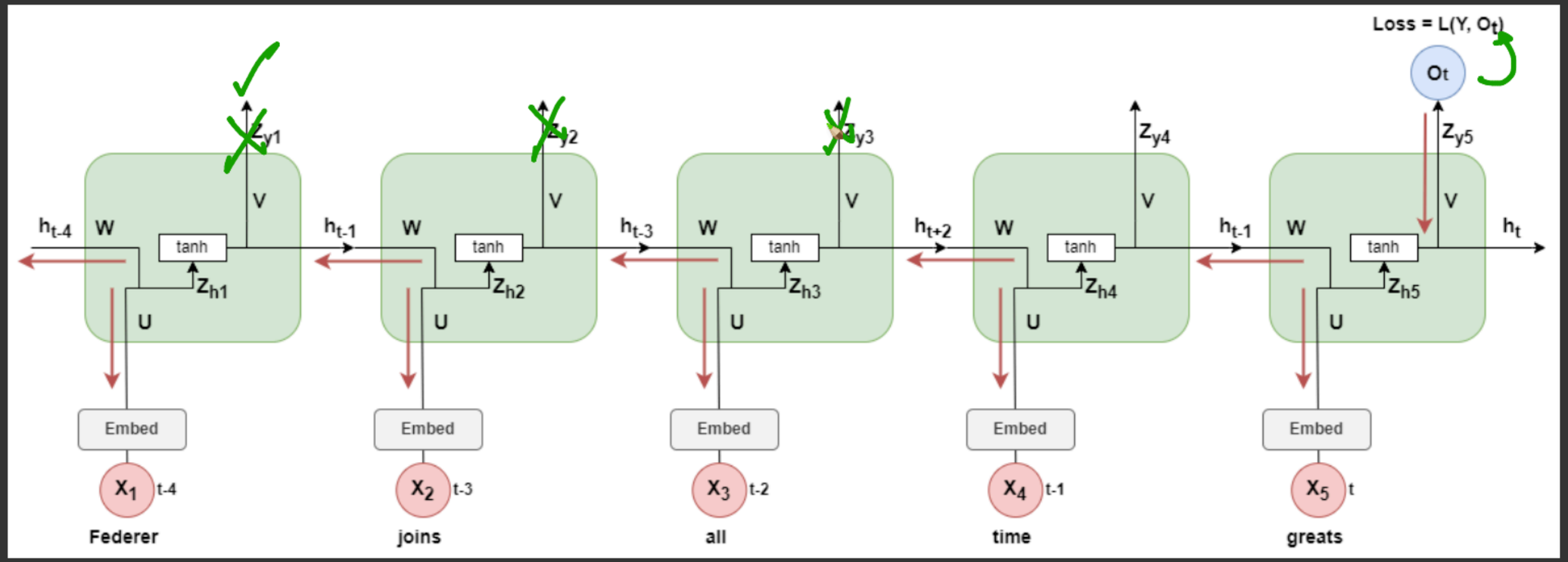
Define the RNN model architecture

- The embedding layer with 100 dimension
- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
[ ]  seed=56
     tf.random.set_seed(seed)
     np.random.seed(seed)

     model = Sequential()
     model.add(Embedding(vocab_size, embedding_dim, trainable=True))
     model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
     model.add(Dense(total_classes, activation='softmax'))
     model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255
```

Backward propagation



Usual steps involved in updating parameter weights:

- Calculate the gradients of the loss with respect to the parameters
- Multiply it with the Learning rate
- Update the new weights

- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
seed=56
tf.random.set_seed(seed)
np.random.seed(seed)

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, trainable=True))
model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
model.add(Dense(total_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255

=================================================================
Total params: 343,805
Trainable params: 343,805
Non-trainable params: 0
_________________________________________________________________
```

Model training

- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
seed=56
tf.random.set_seed(seed)
np.random.seed(seed)

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, trainable=True))
model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
model.add(Dense(total_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336100

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255

=================================================================
Total params: 343,805
Trainable params: 343,805
Non-trainable params: 0
_________________________________________________________________
```

Model training

Model training

- Optimizer: Adam
- Loss: Categorical cross-entrophy since it is a multiclass classification probem
- Early stopping: Used to stop training if validation accuracy does not improve while training to avoid overfitting

```
[ ] tf.random.set_seed(seed)
    np.random.seed(seed)

    model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['acc'])

    early_stopping = EarlyStopping(monitor='val_acc',
                                   mode='max',
                                   verbose=1,
                                   patience=5)

    model.fit(x=train_text_X, y=train_Y,
              validation_data=(test_text_X, test_Y),
              batch_size=64,
              epochs=10,
              callbacks=[early_stopping])
```

```
Epoch 1/10
28/28 [==============================] - 2s 22ms/step - loss: 1.5804 - acc: 0.2713 - val_loss: 1.5594 - val_acc: 0.2921
Epoch 2/10
28/28 [==============================
                                    ss: 1.3880 - val_acc: 0.4472
```

```
[ ] model.fit(x=train_text_X, y=train_Y,
              validation_data=(test_text_X, test_Y),
              batch_size=64,
              epochs=10,
              callbacks=[early_stopping])
```

```
Epoch 1/10
28/28 [==============================] - 2s 22ms/step - loss: 1.5804 - acc: 0.2713 - val_loss: 1.5594 - val_acc: 0.2921
Epoch 2/10
28/28 [==============================] - 0s 11ms/step - loss: 1.2862 - acc: 0.5888 - val_loss: 1.3880 - val_acc: 0.4472
Epoch 3/10
28/28 [==============================] - 0s 11ms/step - loss: 0.6456 - acc: 0.8281 - val_loss: 0.8305 - val_acc: 0.7101
Epoch 4/10
28/28 [==============================] - 0s 11ms/step - loss: 0.2494 - acc: 0.9421 - val_loss: 0.7788 - val_acc: 0.7596
Epoch 5/10
28/28 [==============================] - 0s 11ms/step - loss: 0.1162 - acc: 0.9803 - val_loss: 0.7634 - val_acc: 0.7933
Epoch 6/10
28/28 [==============================] - 0s 11ms/step - loss: 0.0661 - acc: 0.9893 - val_loss: 0.7810 - val_acc: 0.7843
Epoch 7/10
28/28 [==============================] - 0s 11ms/step - loss: 0.0370 - acc: 0.9966 - val_loss: 0.8274 - val_acc: 0.7798
Epoch 8/10
28/28 [==============================] - 0s 15ms/step - loss: 0.0285 - acc: 0.9972 - val_loss: 0.8525 - val_acc: 0.7910
Epoch 9/10
28/28 [==============================] - 0s 17ms/step - loss: 0.0209 - acc: 0.9989 - val_loss: 0.9074 - val_acc: 0.7820
Epoch 10/10
28/28 [==============================] - 1s 18ms/step - loss: 0.0173 - acc: 0.9983 - val_loss: 0.9646 - val_acc: 0.7798
Epoch 10: early stopping
<keras.callbacks.History at 0x7fdedfddfd10>
```

Save the trained model

- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
seed=56
tf.random.set_seed(seed)
np.random.seed(seed)

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, trainable=True))
model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
model.add(Dense(total_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255

=================================================================
Total params: 343,805
Trainable params: 343,805
Non-trainable params: 0
_________________________________________________________________
```

Model training

- A single Vanilla RNN unit with 50 dimensions
- A final output layer with 5 units(5 classes) with the softmax activation function

```
seed=56
tf.random.set_seed(seed)
np.random.seed(seed)

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, trainable=True))
model.add(SimpleRNN(latent_dim, recurrent_dropout=0.2, return_sequences=False, activation='tanh'))
model.add(Dense(total_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 100)         336000

 simple_rnn (SimpleRNN)      (None, 50)                7550

 dense (Dense)               (None, 5)                 255

=================================================================
Total params: 343,805
Trainable params: 343,805
Non-trainable params: 0
_________________________________________________________________
```

Model training

```
Epoch 6/10
28/28 [==============================] - 0s 11ms/step - loss: 0.0661 - acc: 0.9893 - val_loss: 0.7810 - val_acc: 0.7843
Epoch 7/10
28/28 [==============================] - 0s 11ms/step - loss: 0.0370 - acc: 0.9966 - val_loss: 0.8274 - val_acc: 0.7798
Epoch 8/10
28/28 [==============================] - 0s 15ms/step - loss: 0.0285 - acc: 0.9972 - val_loss: 0.8525 - val_acc: 0.7910
Epoch 9/10
28/28 [==============================] - 0s 17ms/step - loss: 0.0209 - acc: 0.9989 - val_loss: 0.9074 - val_acc: 0.7820
Epoch 10/10
28/28 [==============================] - 1s 18ms/step - loss: 0.0173 - acc: 0.9983 - val_loss: 0.9646 - val_acc: 0.7798
Epoch 10: early stopping
<keras.callbacks.History at 0x7fdedfddfd10>
```

Save the trained model

```
model.save("BCC_classifier.h5")
```

Load the saved model

```
model = tf.keras.models.load_model("BCC_classifier.h5")
```

Make predictions on the test dataset

```
[ ] model.save("BCC_classifier.h5")
```

Load the saved model

```
[ ] model = tf.keras.models.load_model("BCC_classifier.h5")
```

Make predictions on the test dataset

```
[ ] prediction = model.predict(test_text_X)
    prediction = prediction.argmax(axis=1)
    print(f"Accuracy: {accuracy_score(prediction, validation)}")
```

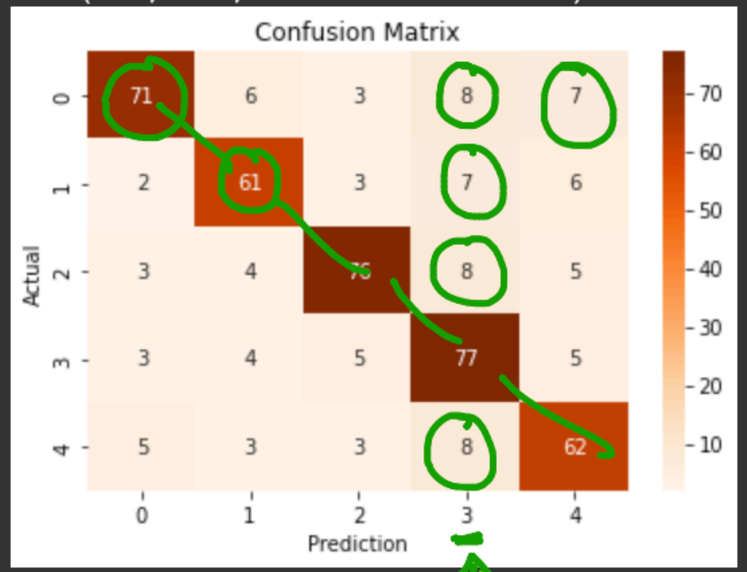
```
14/14 [==============================] - 0s 2ms/step
Accuracy: 0.7797752808988764
```

Confusion matrix of the prediction and actual

```
[ ] cm = confusion_matrix(validation, prediction)
```

```
[ ]  cm = confusion_matrix(validation, prediction)

     # print("")
     # plt.figure(figsize=(15,15))
     sns.heatmap(cm, annot=True, cmap='Oranges')
     plt.xlabel("Prediction")
     plt.ylabel("Actual")
     plt.title("Confusion Matrix")
```

```
Text(0.5, 1.0, 'Confusion Matrix')
```



Conclusions

text	summary

Let's look at a sample

what type of RNN ?

↓

many - many

```
ind = 1
print(f'Text: {pre.text[ind]}')
print()
print(f'Summary: {pre.summary[ind]}')
print()
print(f'Text length: {len(pre.text[ind].split())}')
print(f'Summary length: {len(pre.summary[ind].split())}')
```

```
Text: India recorded their lowest ODI total in New Zealand after getting all out for 92 runs in 30.5 overs in the fourth ODI at Hamilton o

Summary: India get all out for 92, their lowest ODI total in New Zealand

Text length: 56
Summary length: 13
```

What kind of problem is it?

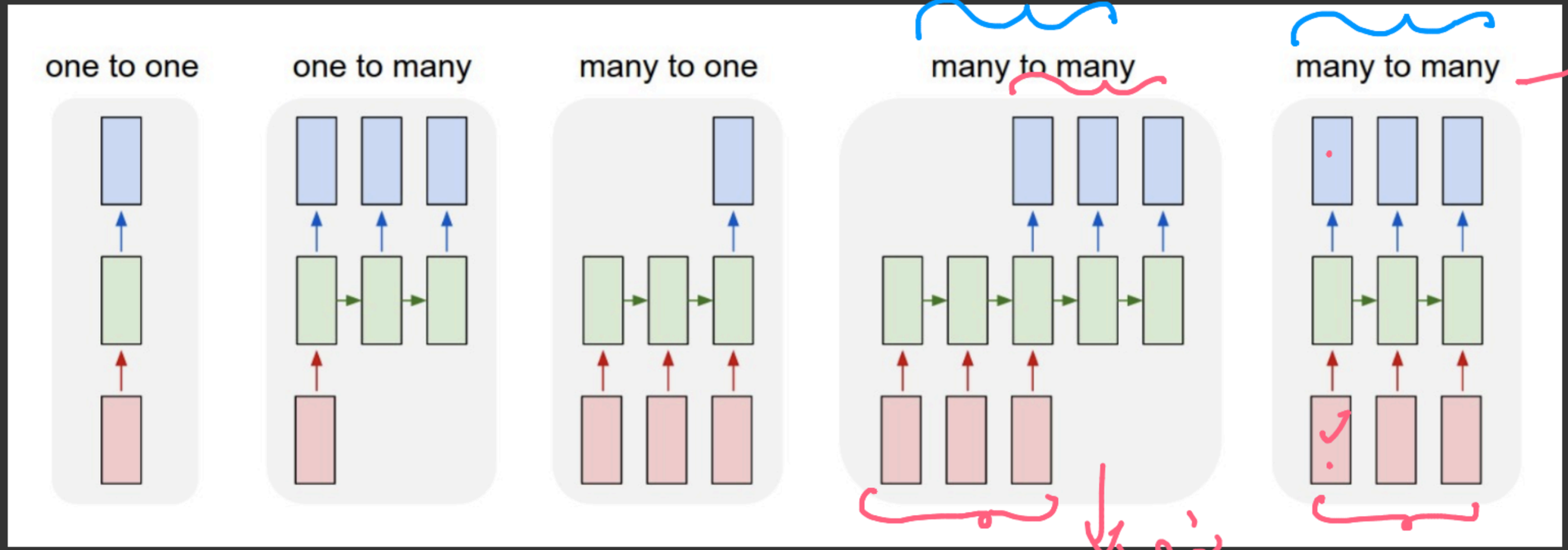
- **Many to Many:** From the problem statement we can understand that it has multiple input and output, so it's a many to many problem.

Which architecture works well for this?

- **Encoder Decoder:** Since the input and t architecture

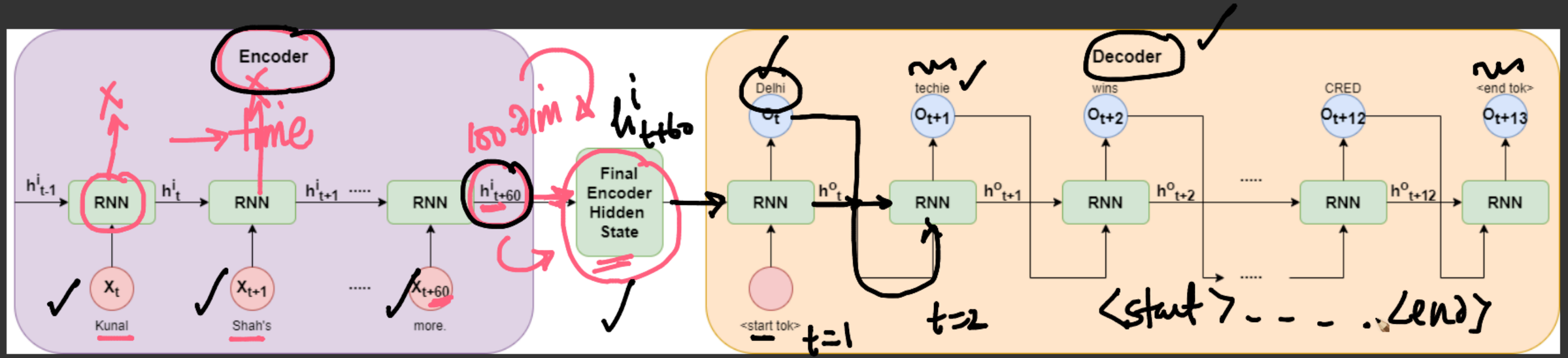
Types of input & output

There are couple of varaints of RNN based on the number of inputs and the outputs.



One to One: $T_x=1, T_y=1$

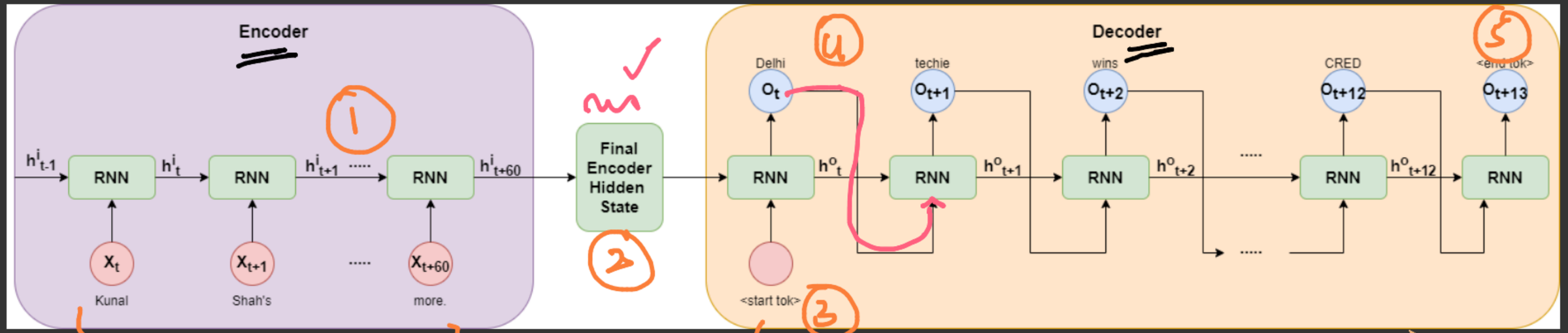
Example: Normal Feedforward Neural Network



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

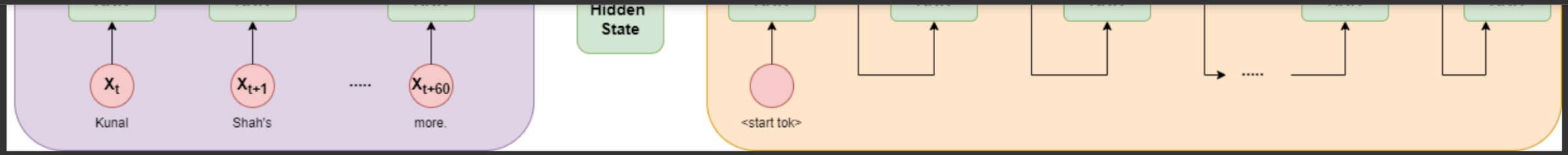
Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.

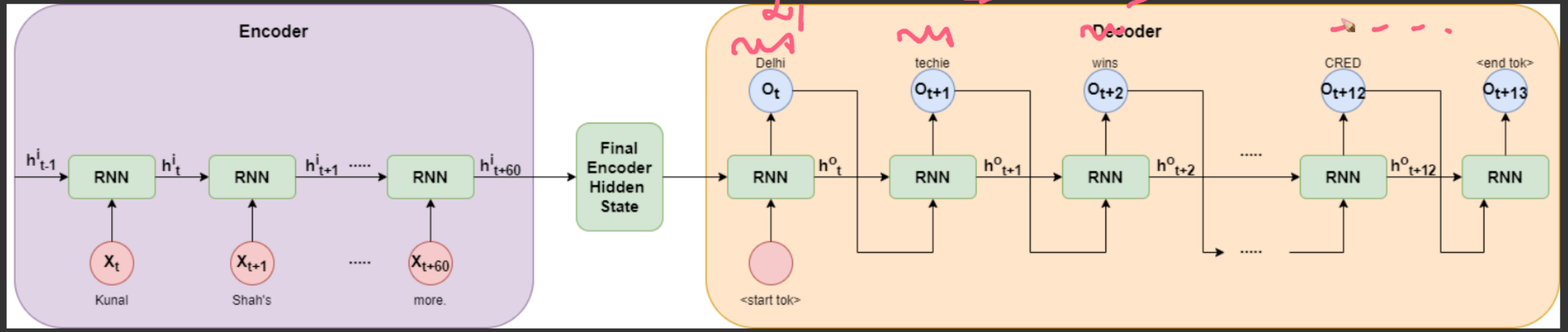
In the above example to complete the last word **Eiffel tower**, the location **Paris** mentioned at the 3rd words in the sentence had to be retained to be able to make the right prediction. This is called long term dependencies

What is the problem with it?

- **The bottle-neck at Final Encoder Hidden state**:

Which architecture works well for this?

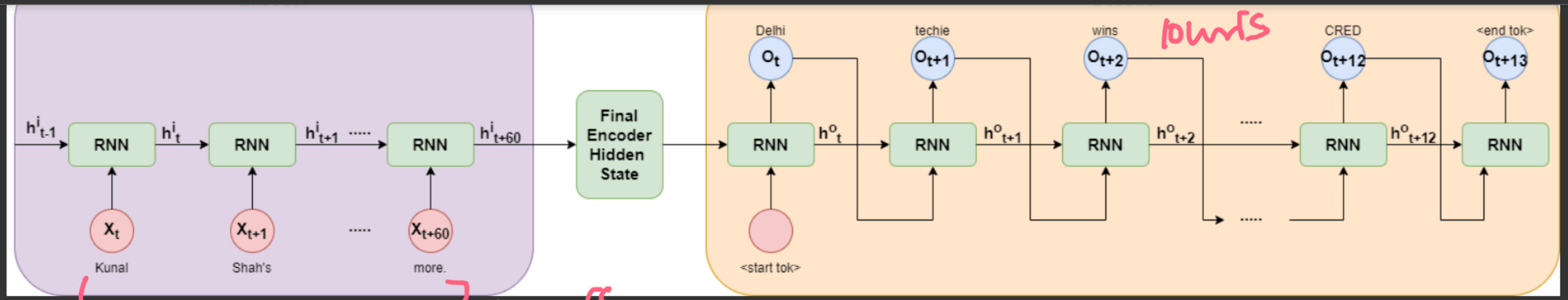
- **Encoder Decoder:** Since the input and the output lengths are not the same we will use the Encoder Decoder architecture



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited **Paris** last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel

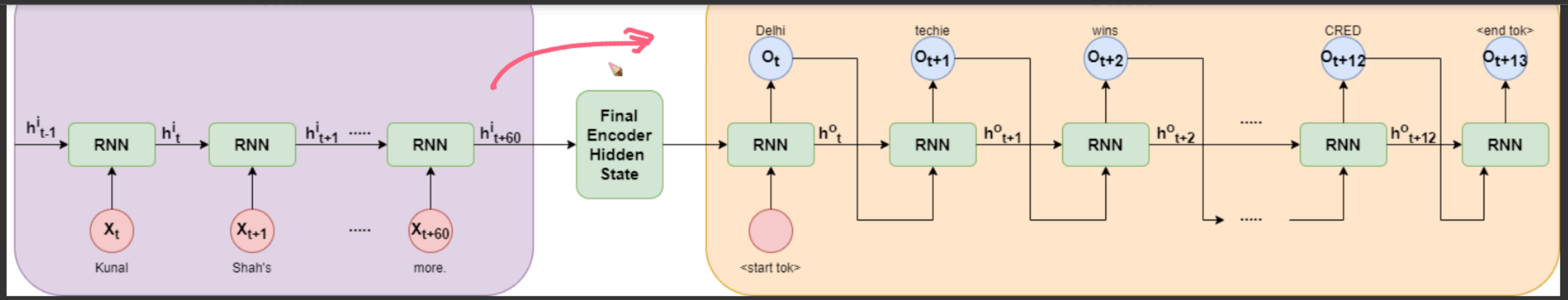


What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.

In the above example to complete the last word **Eiffel tower**, the location **Paris** mentioned at the 3rd words in the sentence had to be retained to be able to make the right prediction. This is called long term dependencies



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.

In the above example to complete the last word **Eiffel tower**, the location **Paris** mentioned at the 3rd words in the sentence had to be retained to be able to make the right prediction. This is called long term dependencies

Any Suggestions

Prob: Vanishing / exploding gradients in BPTT

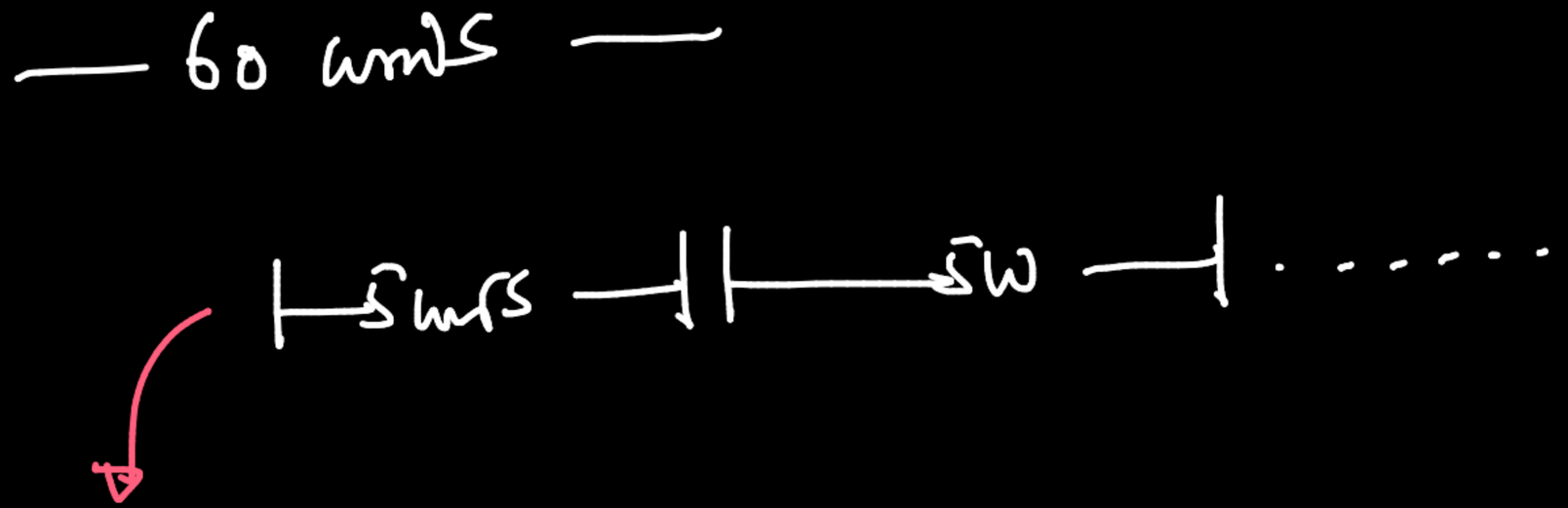
$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial C_4} \cdot \frac{\partial C_4}{\partial C_3} \cdot \frac{\partial C_3}{\partial C_2} \cdot \frac{\partial C_2}{\partial C_1} \cdot \frac{\partial C_1}{\partial W}$$

$\downarrow$

Not able to learn long term dependencies

Chrome File Edit View History Bookmarks Profiles Tab Window Help 9:49

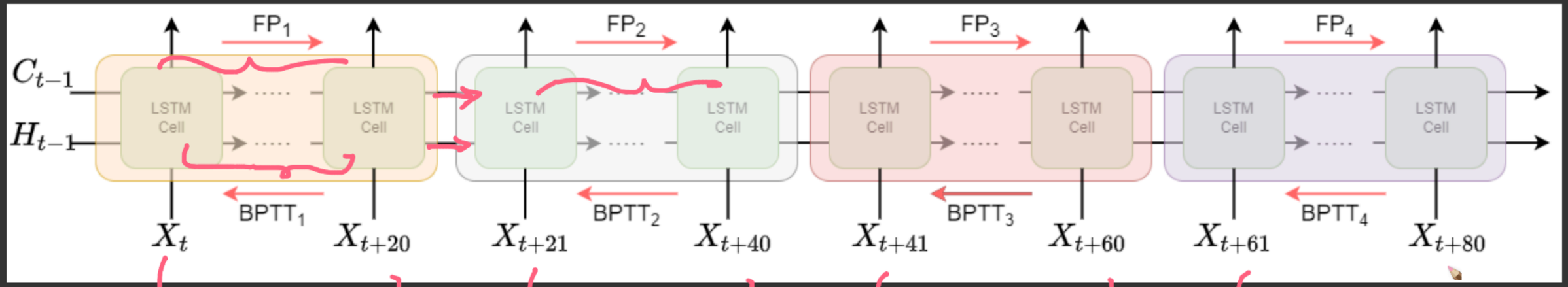
Truncated BPTT



- If there are 80 words and a window size of 20 is chosen then there are 4 chunks.
- The weights are updated for each of the 4 chunks or in another terms, the weights are updated for every 20 input words.
- Truncated BPPT is a less complex approach compared to normal BPPT.
- TBPPT is faster than BPPT because the weight update happens for each chunks and not just once from the last word to the first word.

Disadvantage

- Smaller chunks can result in dependencies not being learnt

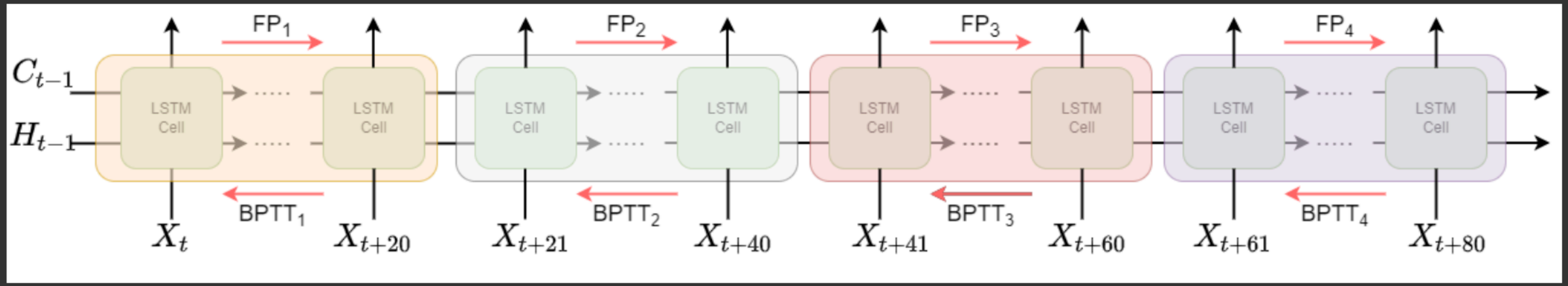


Gradient clipping

- If there are 80 words and a window size of 20 is chosen then there are 4 chunks.
- The weights are updated for each of the 4 chunks or in another terms, the weights are updated for every 20 input words.
- Truncated BPPT is a less complex approach compared to normal BPPT.
- TBPPT is faster than BPPT because the weight update happens for each chunks and not just once from the last word to the first word.

Disadvantage

- Smaller chunks can result in dependencies not being learnt



Gradient clipping

What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the weekend I even got a chance to see the Eiffel tower.

In the above example to complete the last word **Eiffel tower**, the location **Paris** mentioned at the 3rd words in the sentence had to be retained to be able to make the right prediction. This is called long term dependencies

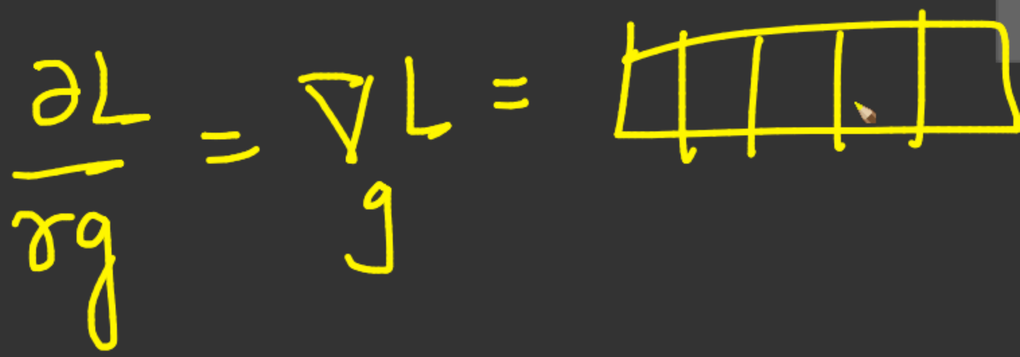
What is the problem with it?

- **The bottle-neck at Final Encoder Hidden state**:
  > As we can see from the architecture the entire inputs at the encoder has to be retained by the **final encoder hidden state**.
  > This also means that the context of the words at the begining of the sentence would be retained compared to the words at the begining of the sentence
  > Imagine if a input has sentences which could contain 100s or 1000s of words then retaining the context of the initial words is very

Can we overcome this ???

Well, we cannot completely overcome vanishing and exploding gradients problem but yes there are ways to mitigate them

- Truncated Backpropagation through time
- Gradient clipping
- Weight initialization
- Long Short-Term Memory Network(LSTM)



Truncated Backpopagation through time

Difference between BPTT and TBPTT

- In the normal RNN training their is a forward and backward pass for the entire input sequence to compute the loss gradients and update the weights
- In truncated Backpropagation the input sequence is split into chunks of specified windows size.
- The forwards and backpropagation through time happens only for the chunks.

Faster training

- If there are 80 words and a window size of 20 is chosen then there are 4 chunks.

Gradient clipping

Gradient clipping as the name suggest we clip the gradients if it goes beyond a threshold. It is most widely used for preventing exploding gradients but can help in vanishing gradients too. There are two methods of clipping.

Clip by value:

if $||g||$ >= upper threshold, then g = upper threshold

if $||g||$ <= lower threshold, then g = lower threshold

Clip by norm:

if $||g||$ >= upper threshold, then g = upper threshold * $\frac{g}{||g||}$

if $||g||$ <= lower threshold, then g = lower threshold * $\frac{g}{||g||}$

where

- g is the gradient with respect to the parameters
- $||g||$ is the norm of gradient g
- upper, lower thresholds are hyperparameters

Gradient clipping

Gradient clipping as the name suggest we clip the gradients if it goes beyond a threshold. It is most widely used for preventing exploding gradients but can help in vanishing gradients too. There are two methods of clipping.

Clip by value:

if $||g||$ >= upper threshold, then g = upper threshold

if $||g||$ <= lower threshold, then g = lower threshold

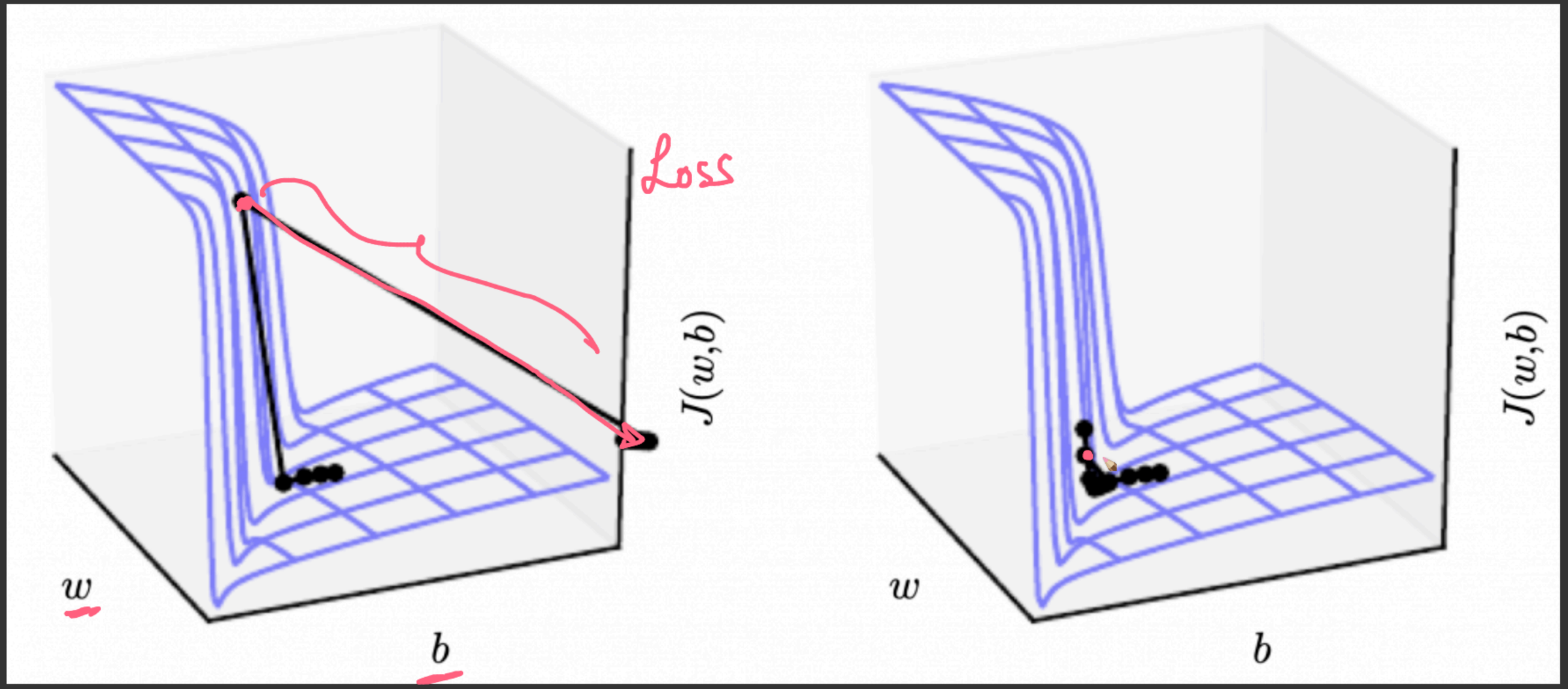
Clip by norm:

if $||g||$ >= upper threshold, then g = upper threshold * $\frac{g}{||g||}$

if $||g||$ <= lower threshold, then g = lower threshold * $\frac{g}{||g||}$

where

- g is the gradient with respect to the parameters
- $||g||$ is the norm of gradient g
- upper, lower thresholds are hyperparameters



- Initializing the weights also play a very important role due to which the vanishing and the exploding gradients arise
- Initializing the weights with Xavier intialization or He initialization forces the varaince of the weights to be smaller.
- This can help in mitigating the problem of Vanishing and Exploding gradients.

There are 4 ways of weight initialization

Normal Xavier initilization: Initialize the weights by drawing from a normal distribution with

$$Mean = 0, SD = \sqrt{\frac{2}{(fan_{in}+fan_{out})}}$$

Uniform Xavier initilization: Initialize the weights by drawing from a uniform distribution

$$between[-w, w], w = \sqrt{\frac{6}{(fan_{in}+fan_{out})}}$$

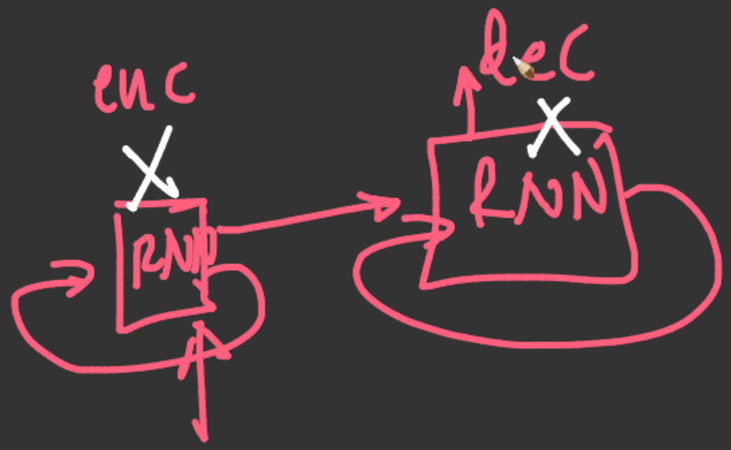
Normal He initilization: Initialize the weights by drawing from a normal distribution

$$Mean = 0, SD = \sqrt{\frac{2}{(fan_{in})}}$$

Uniform He initilization: Initialize the weights by drawing from a uniform distribution

$$between[-w, w], w = \sqrt{\frac{6}{(fan_{in})}}$$

Note: fan_{in}, fan_{out} mean the number of neurons coming in and going out respectively.

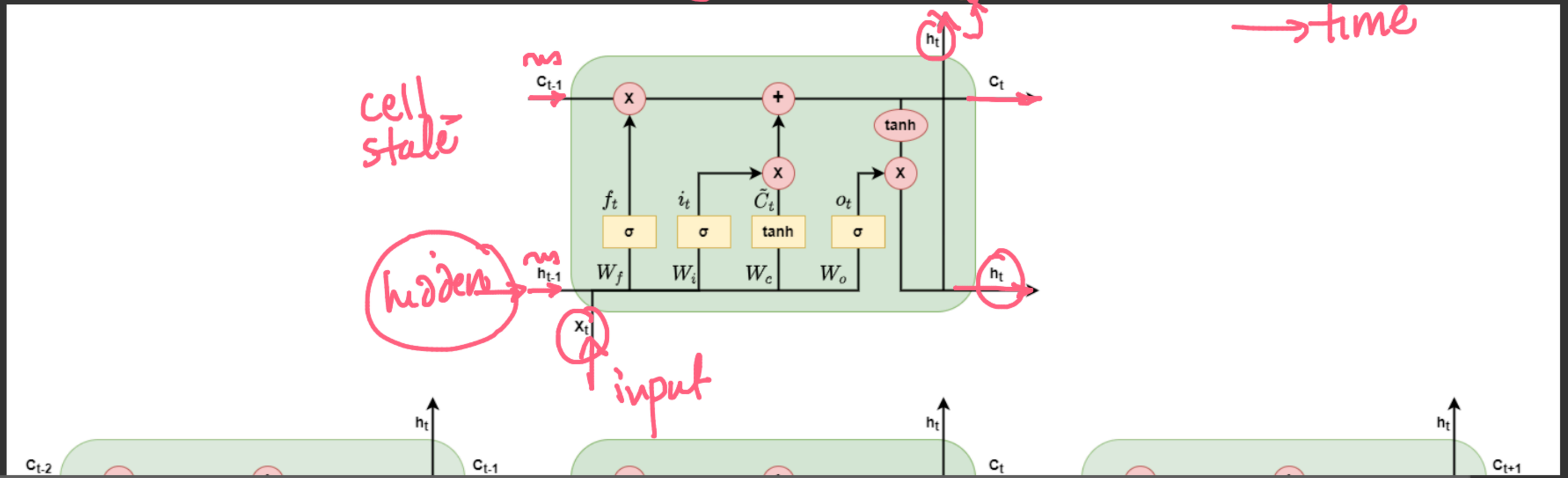


A closer look at LSTM

$between[-w, w], w = \sqrt{\frac{6}{(fan_{in})}}$

Note: fan_{in}, fan_{out} mean the number of neurons coming in and going out respectively.

A closer look at LSTM



How LSTM overcomes the short term memory problem?

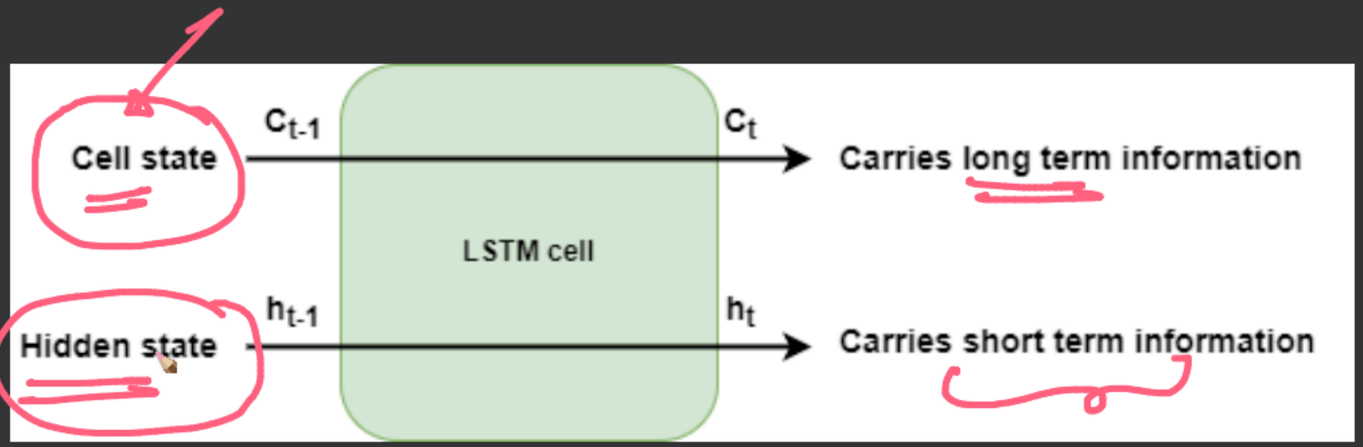
LSTMs are special type of RNN structures that contains 2 memory units compared to 1 in the Vanilla RNN.

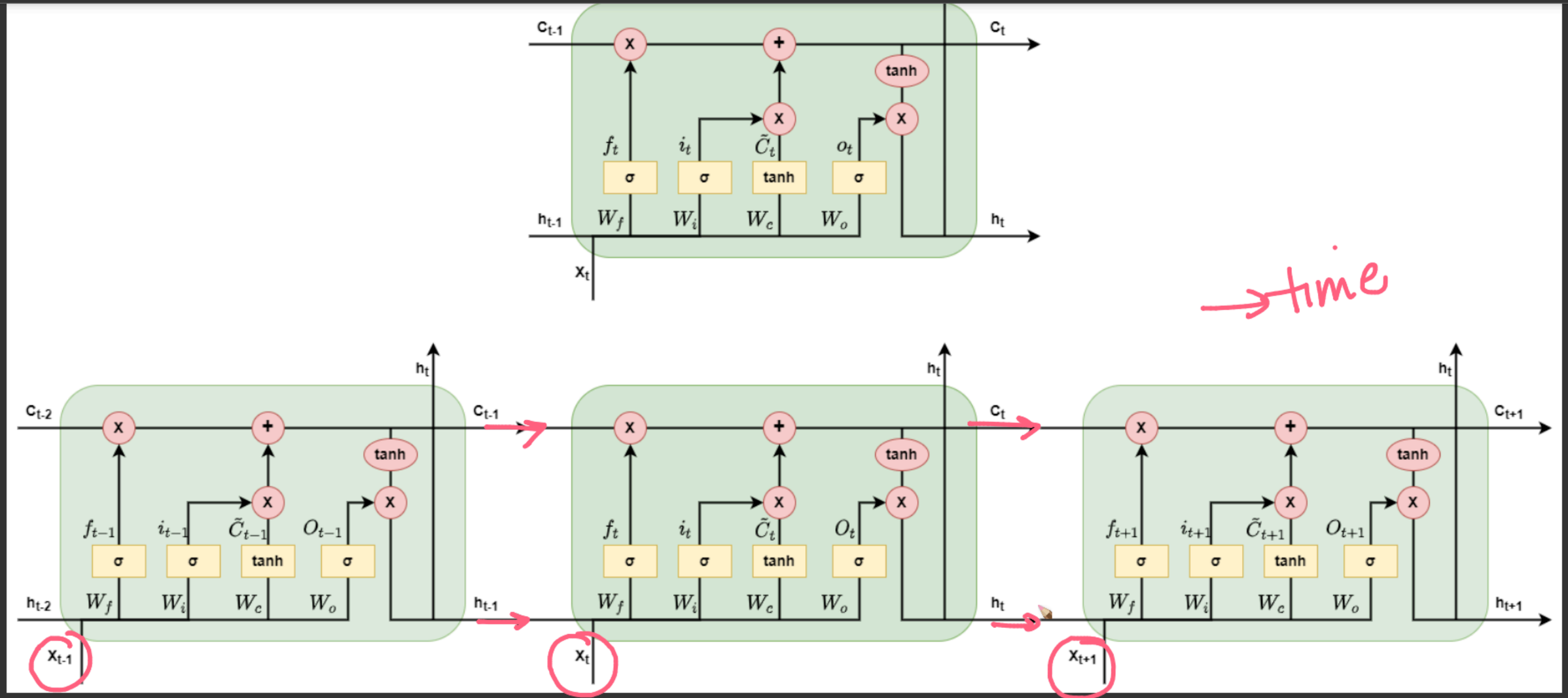
Cell state:

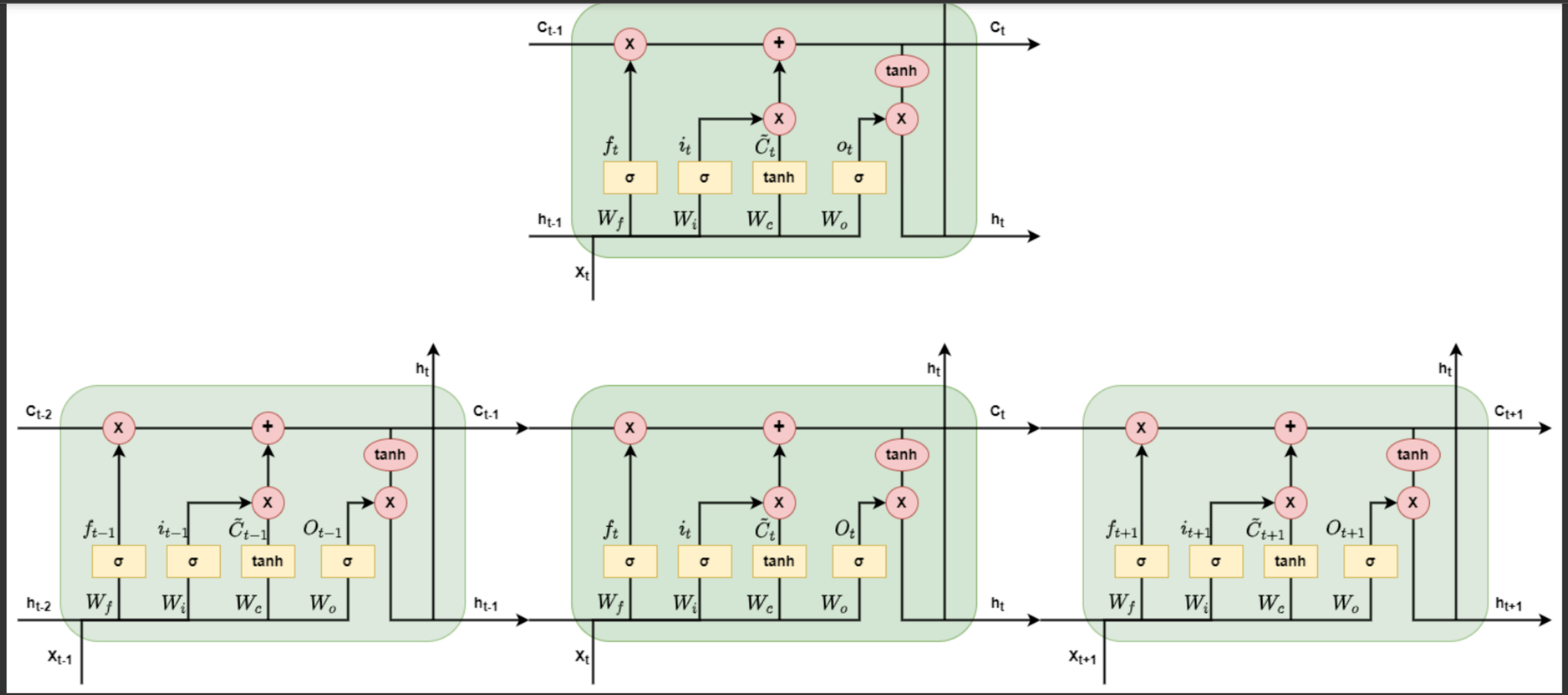
It has a new state called the Cell state $\mathbf{C_t}$ which retains long term information

Hidden state:

The hidden state $\mathbf{h_t}$ is similar to the hidden state in Vanilla RNN which carries the short term memory







Types of gates

- Forget gate
- Input gate
- Output gate

How Forget, Input and Output Gate works?

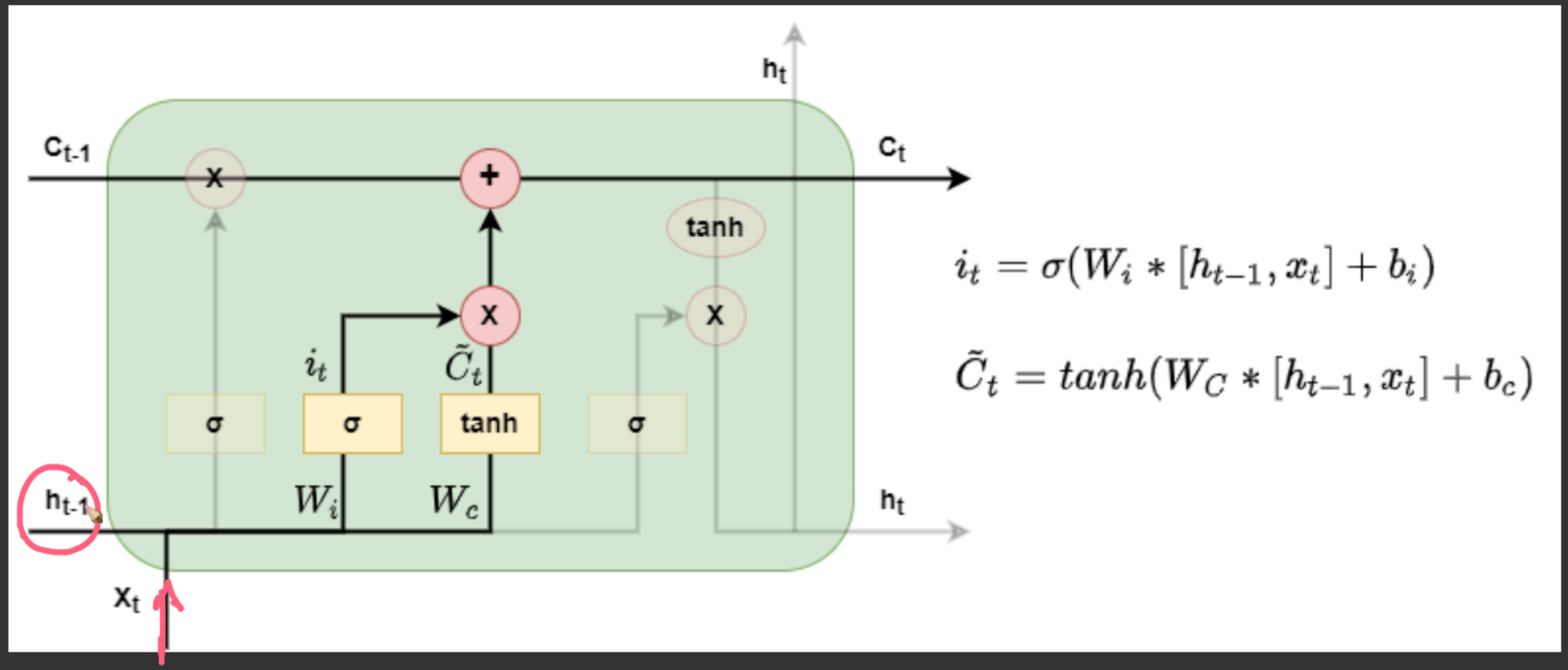
Let's understand it with an example sentence

Sentence: I like to travel and I visited Paris last year as a part of my business trip. It was winter season and too cold for someone from the sub-continent. I stayed at Hilton hotel and I was fortunate enough to get a room where from my window I could see the Eiffel tower. I took my family on vaction to Cairo, Egypt last month. They were quite a lot of market area for shopping to buy souvenirs and my kids enjoyed seeing the great pyramids for the first time.

Summary: I saw Eiffel tower when I was in Paris and during my vacation I visited the Pyramids in Cairo.

Input gate

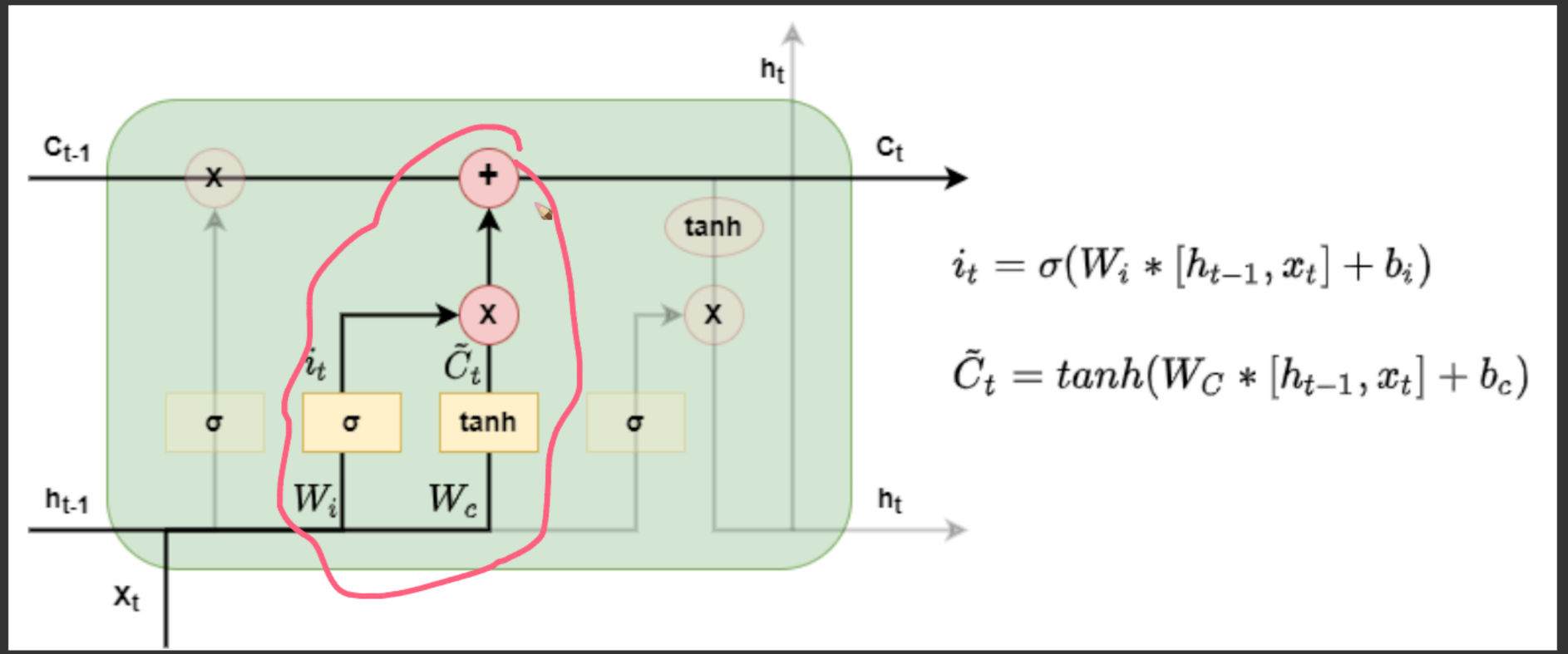
Input gate



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

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

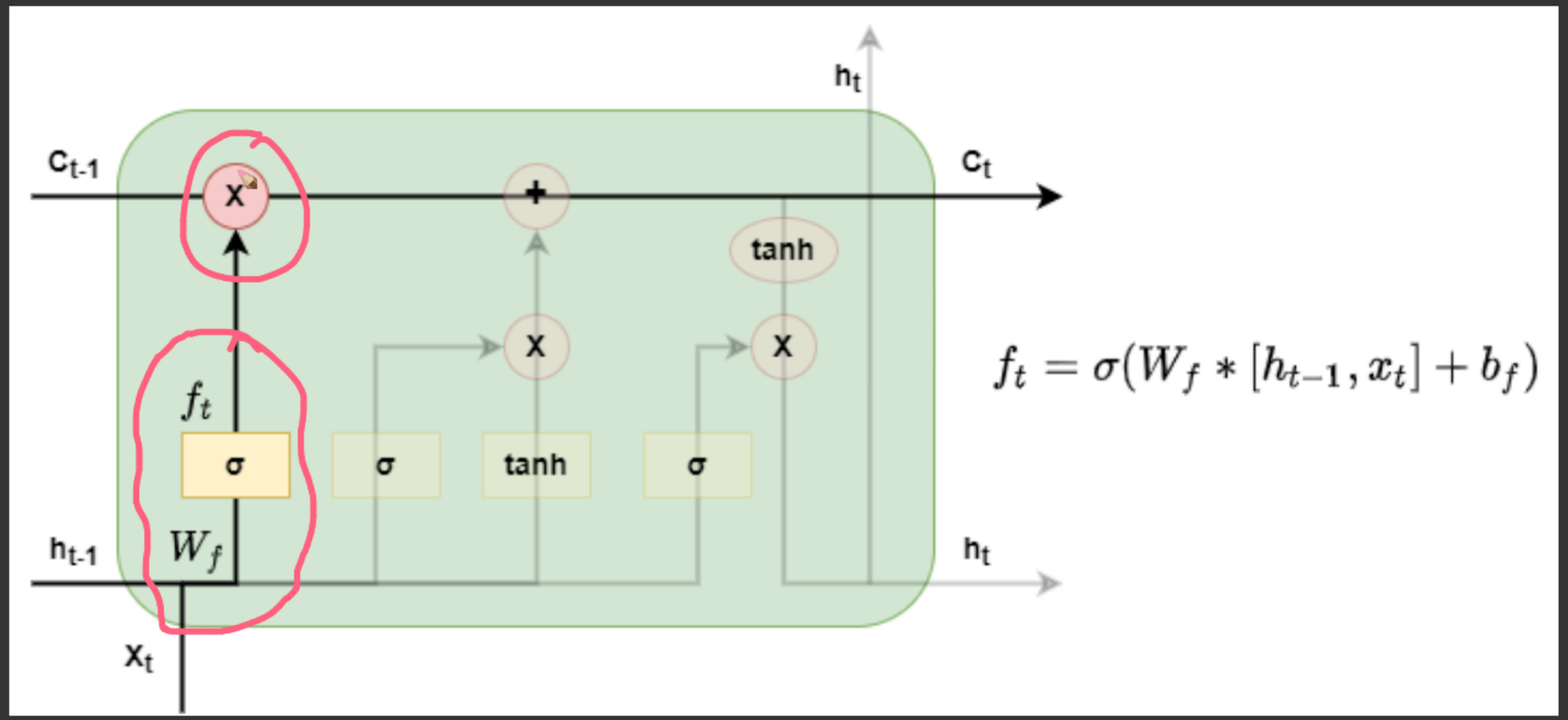
Input gate



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

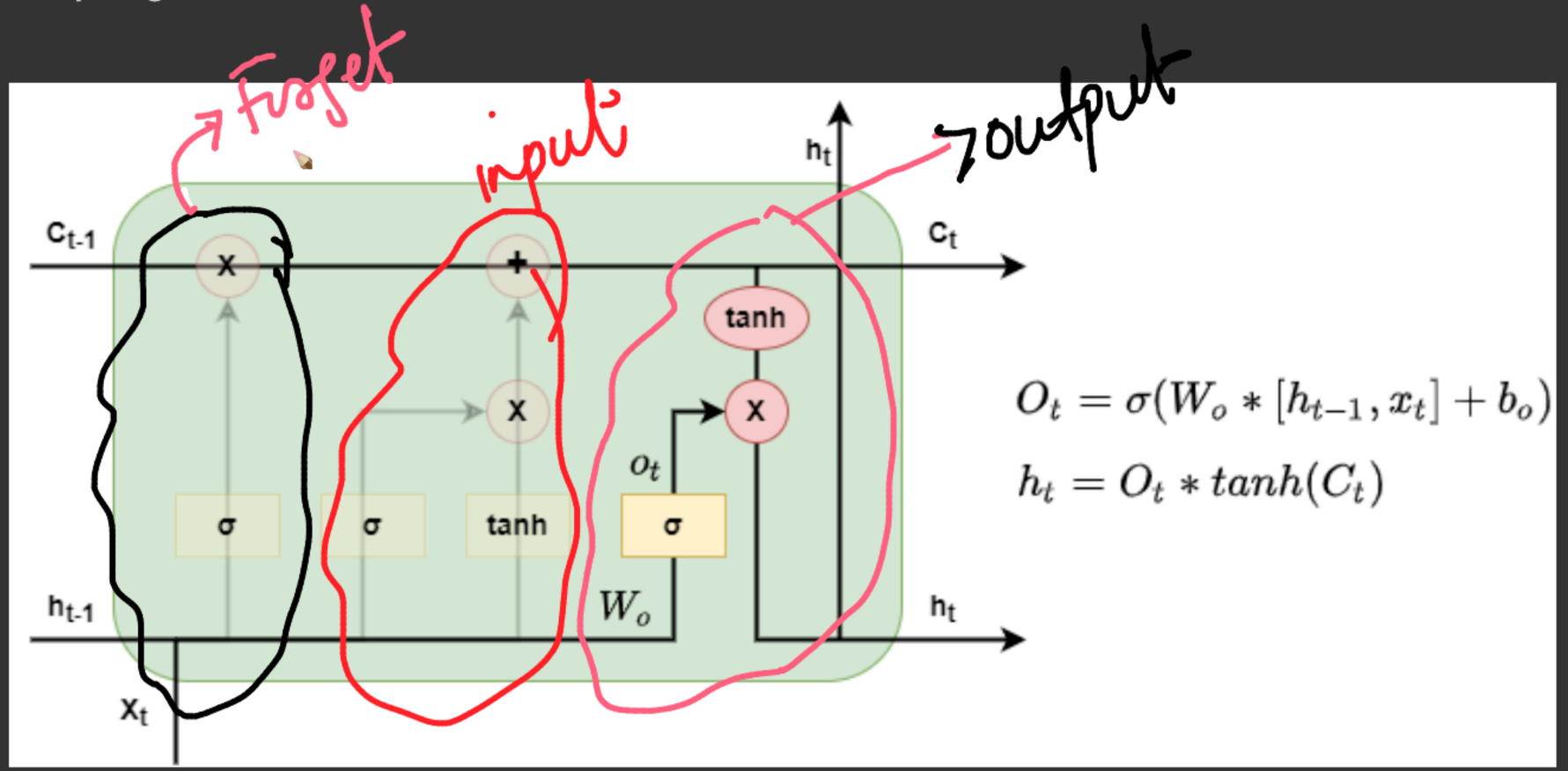
$$\tilde{C}_t = tanh(W_C * [h_{t-1}, x_t] + b_c)$$

Forget gate



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

Output gate



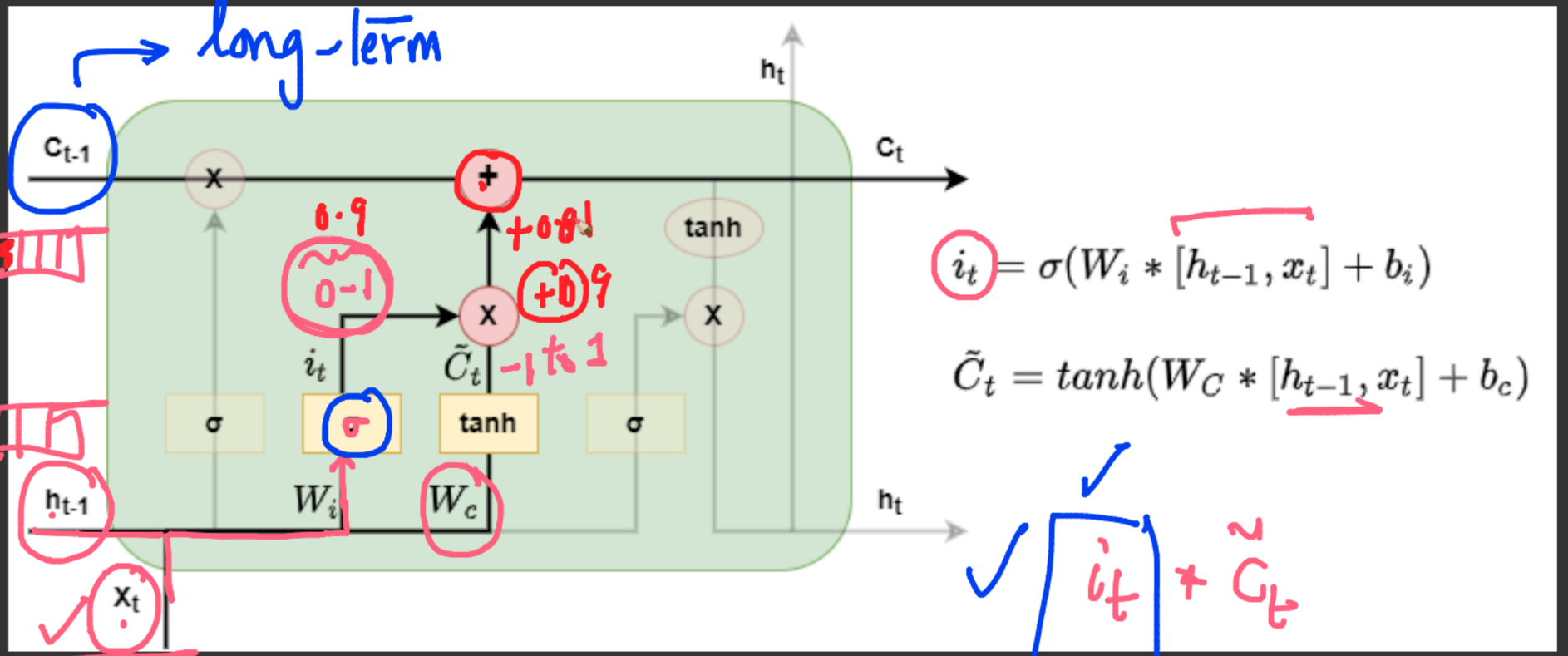
$$O_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = O_t * tanh(C_t)$$

- The output gate regulates the present hidden state h_t and it decides what relevant information has to be passed to the next hidden state

It takes h_{t-1} and X_t as input to the sigmoid function

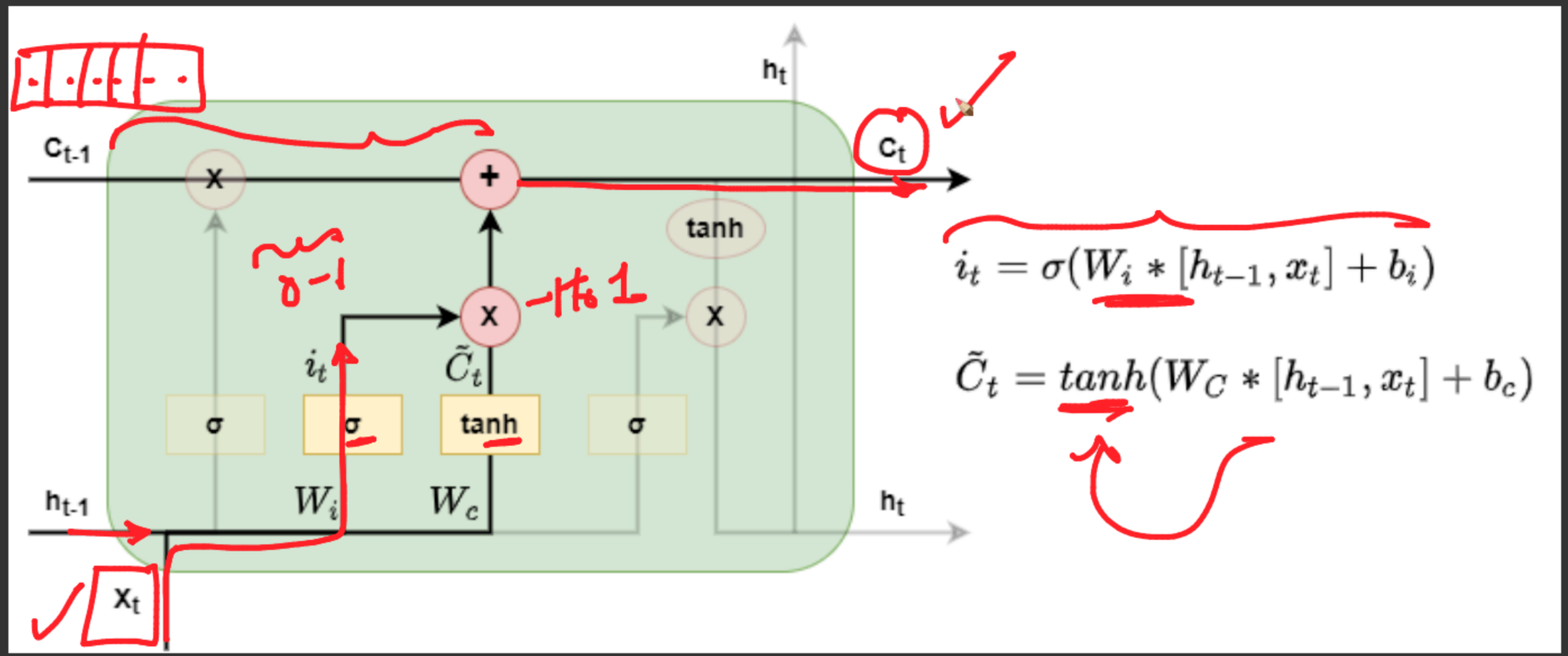
Input gate



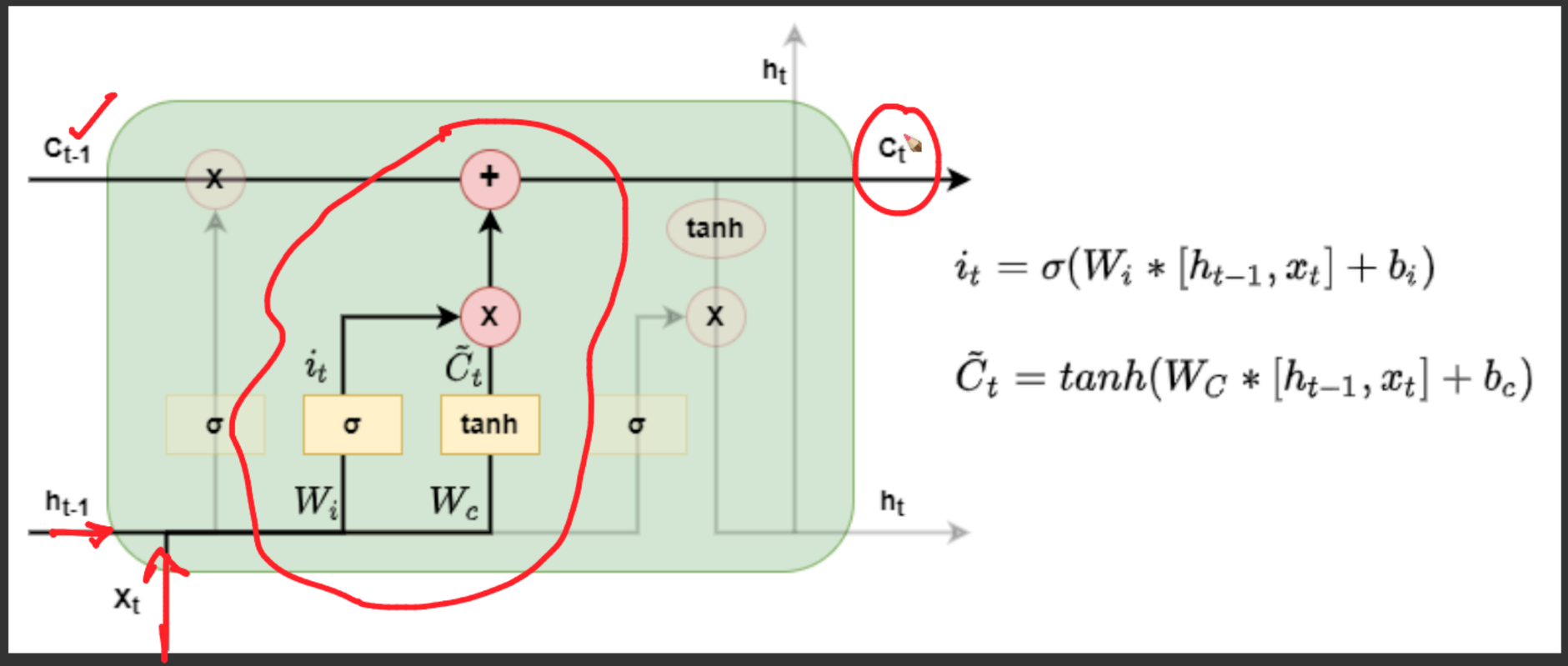
$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

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

Input gate

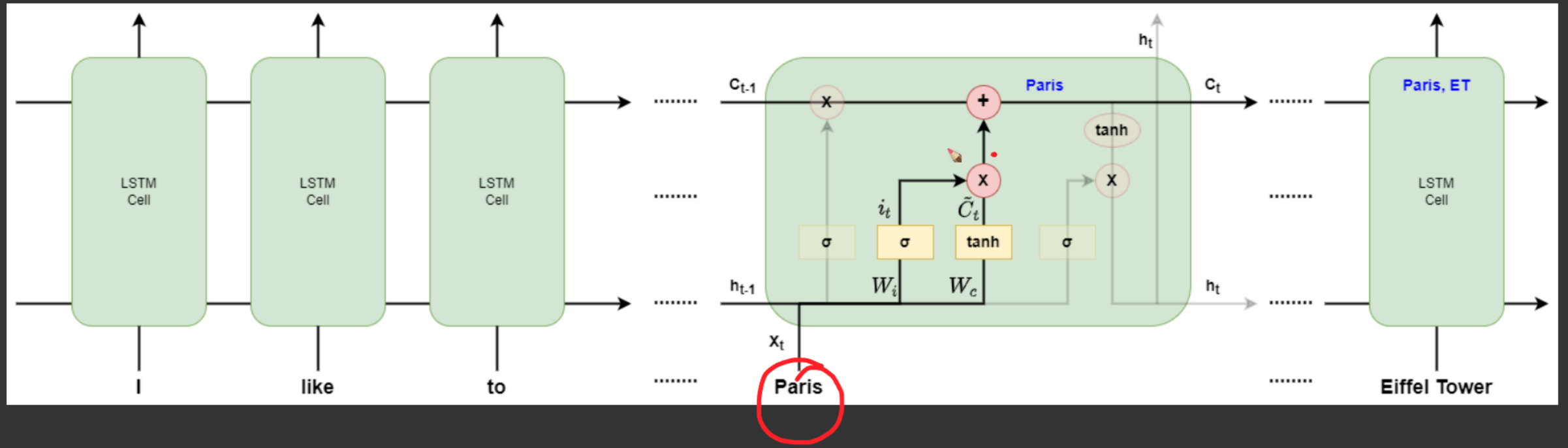


Input gate



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

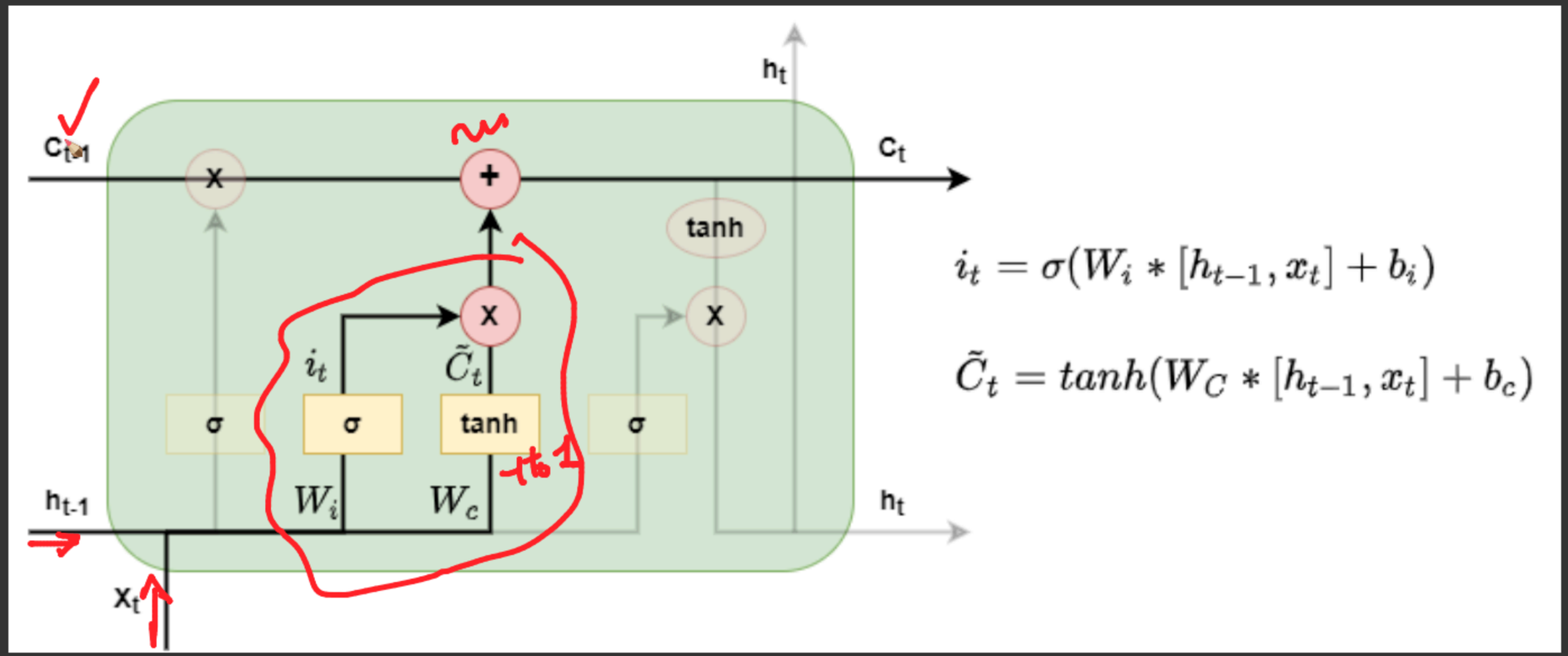
$$\tilde{C}_t = tanh(W_C * [h_{t-1}, x_t] + b_c)$$



At first the weights are randomly initialized and the values of C_{t-1} weight and h_{t-1} are all 0.

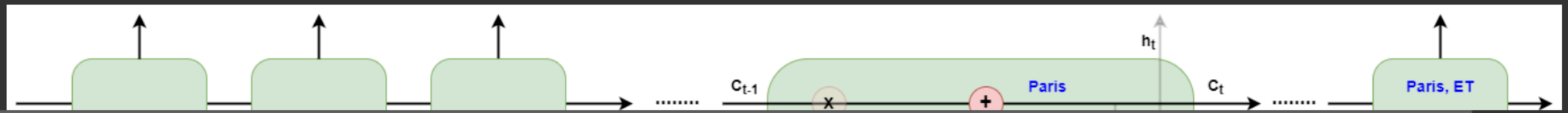
- As we know each word is passed at each time step.
- When the word **Paris** is passed to the LSTM cell,

input gate

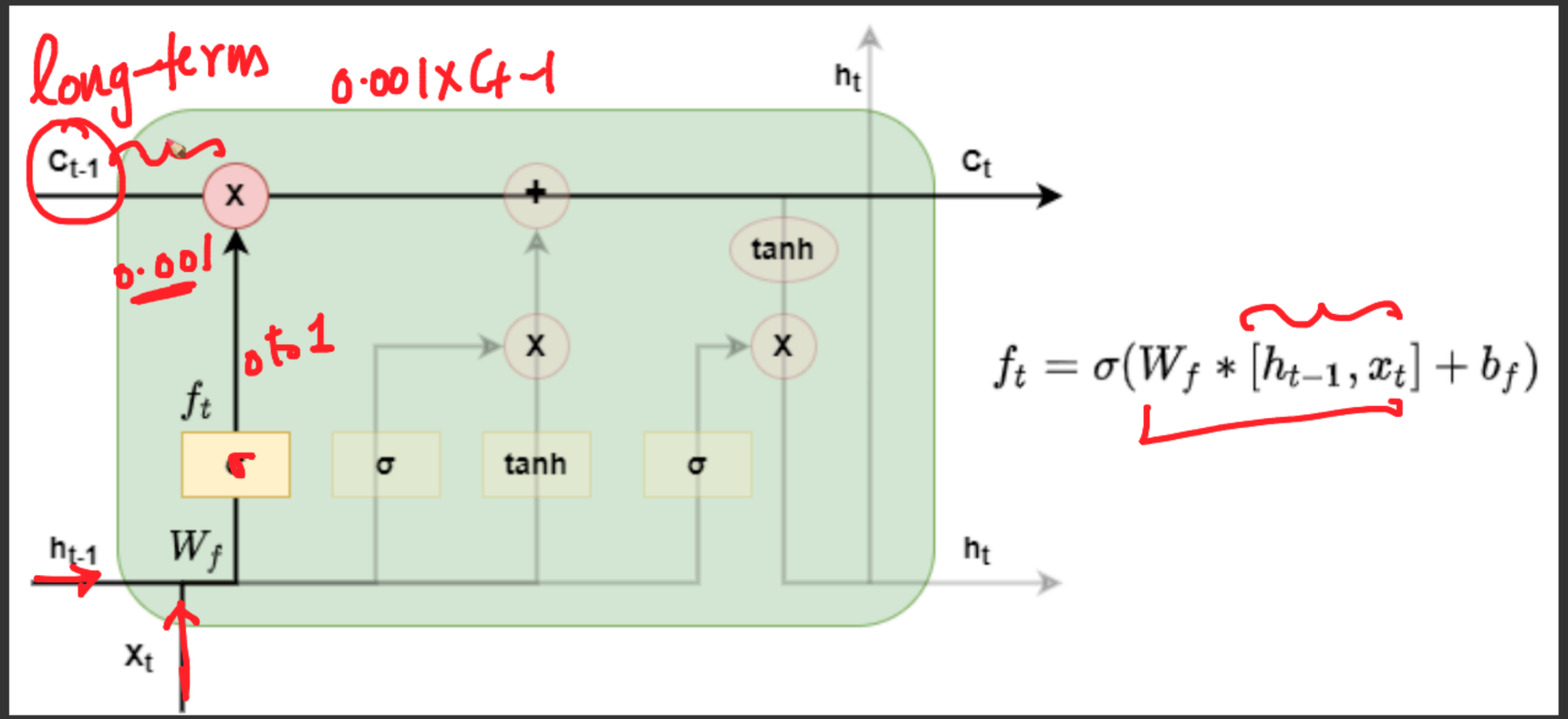


$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

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

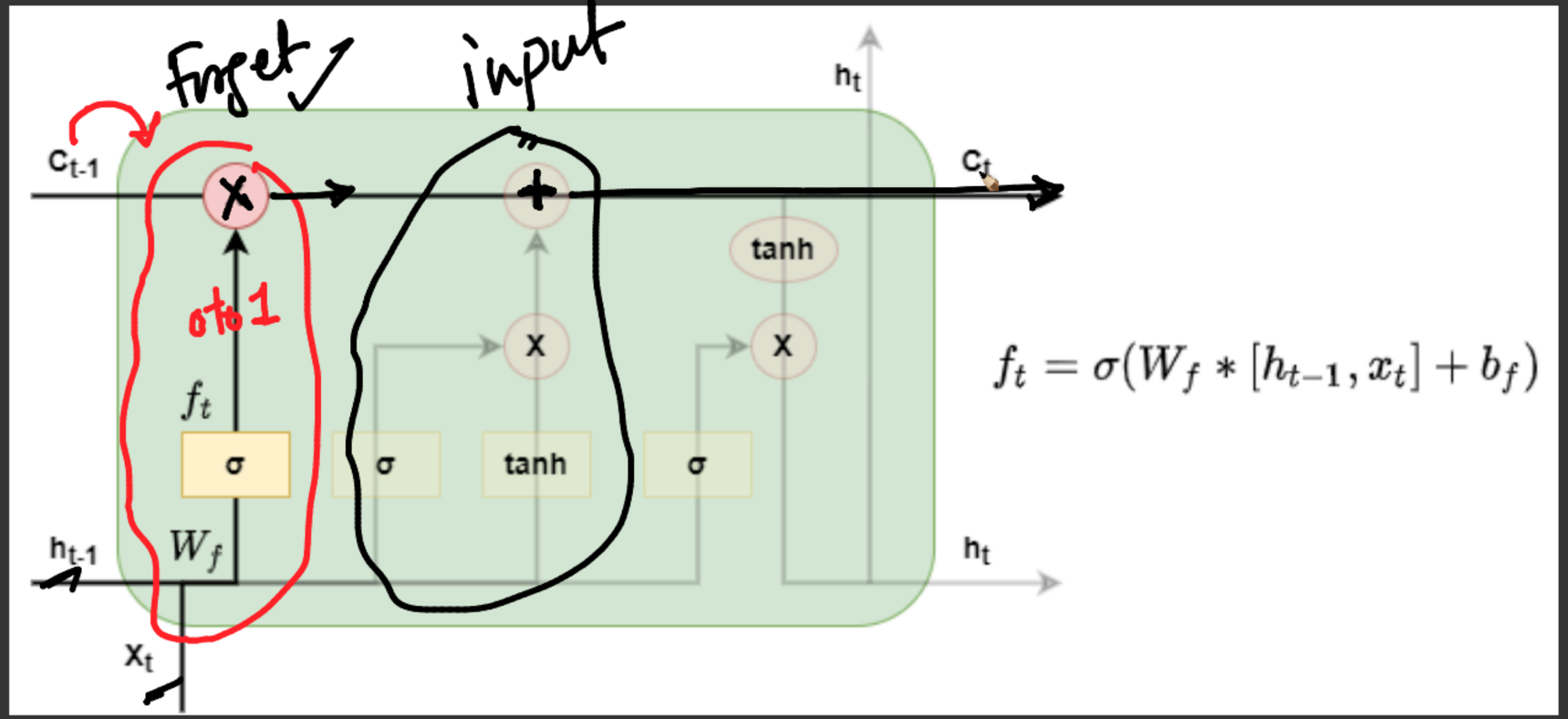


Forget gate

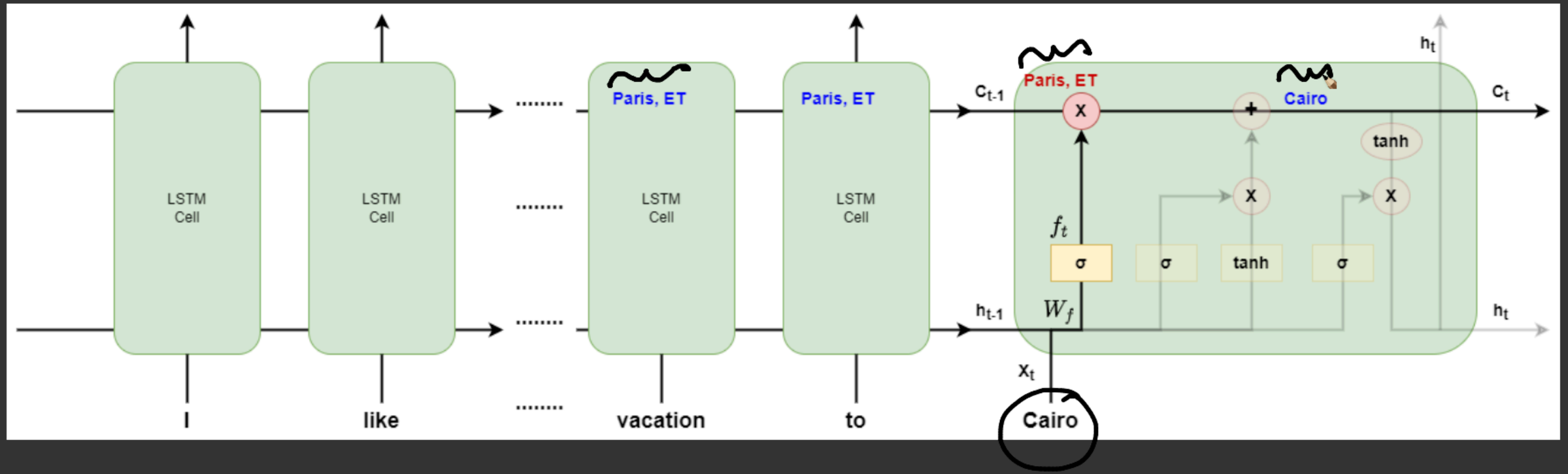


$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$

Forget gate

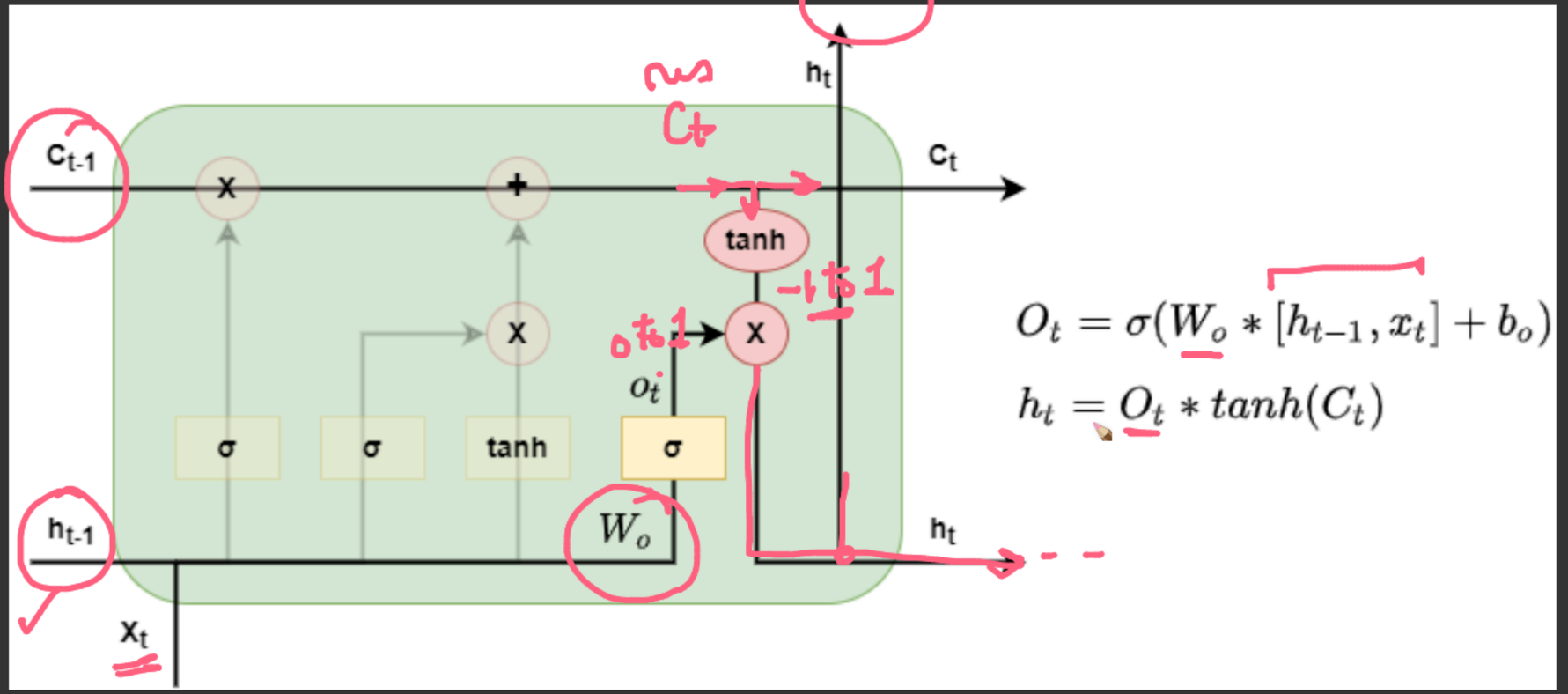


$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$



- When the word **Paris** was passed at a time step, it was updated in the cell state C_t from the input gate
- When the word **Cairo** is passed to the LSTM cell,
  - The long term memory has to forget about Paris

Output gate



$$O_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = O_t * tanh(C_t)$$

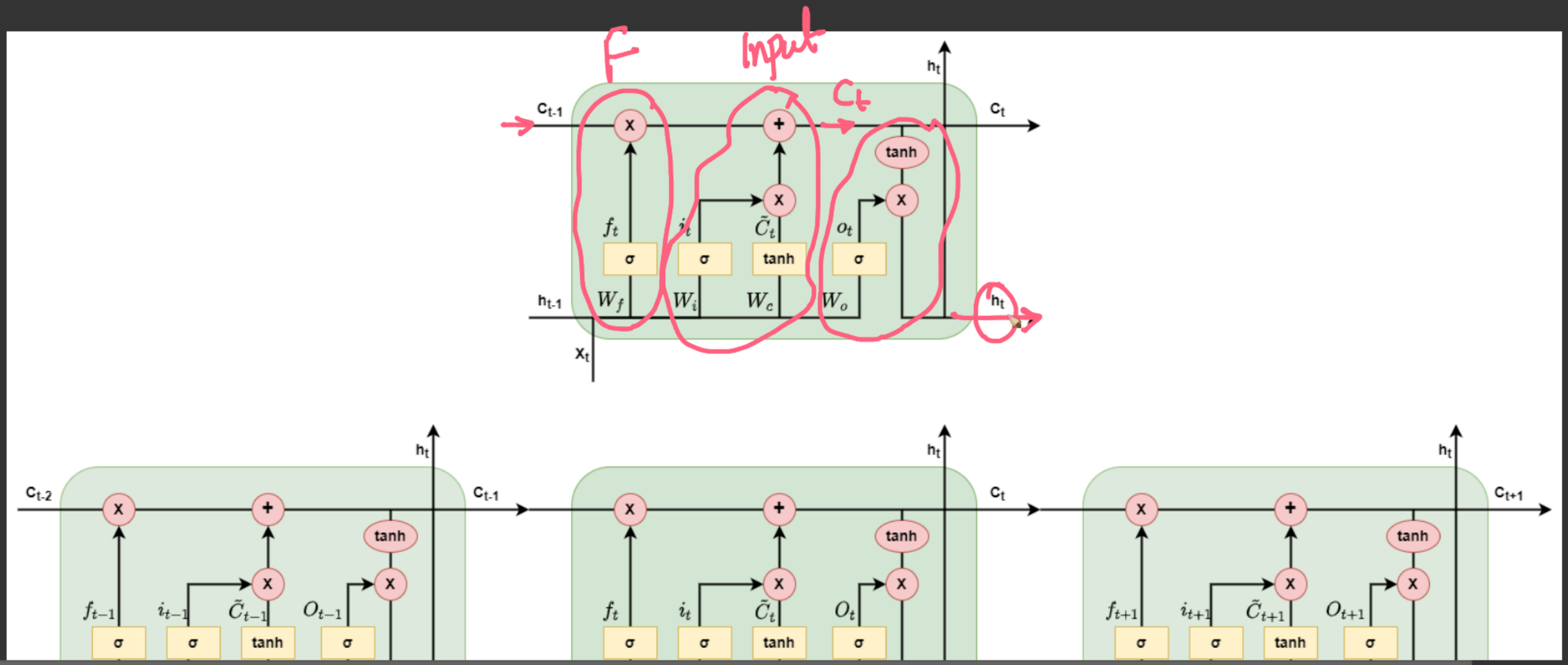
Chrome File Edit View History Bookmarks Profiles Tab Window Help 10:26

Vanilla-RNN_V1.ipynb - Colabo | LSTM_V1.ipynb - Colaboratory | TimeDistributed layer | ROUGE (metric) - Wikipedia

colab.research.google.com/drive/1jSDd8nTYGpPmDD4JsXF-lFeWJGOK1boX#scrollTo=242131e0

+ Code + Text Last saved at 16:21 Connect

A closer look at LSTM



57 / 57

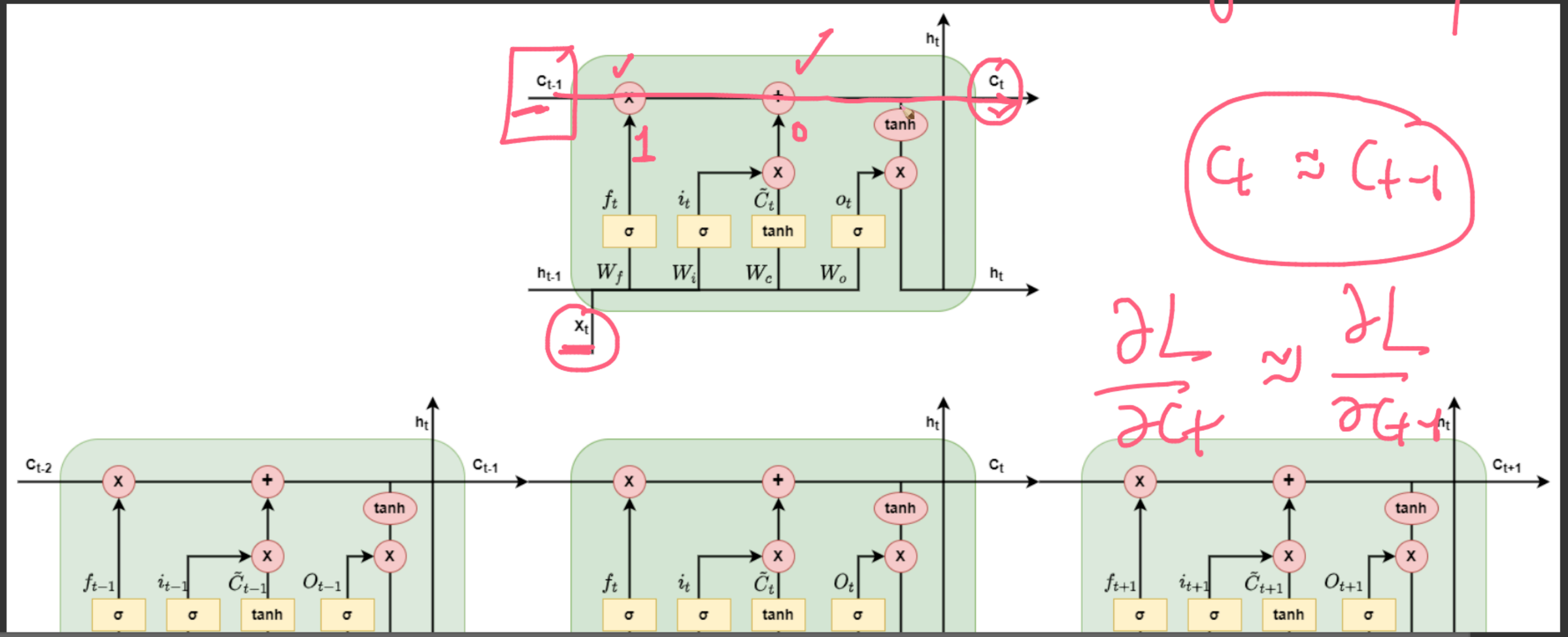
Chrome File Edit View History Bookmarks Profiles Tab Window Help 10:28

Vanilla-RNN_V1.ipynb - Colabo | LSTM_V1.ipynb - Colaboratory | TimeDistributed layer | ROUGE (metric) - Wikipedia

colab.research.google.com/drive/1jSDd8nTYGpPmDD4JsXF-IFeWJGOK1boX#scrollTo=242131e0

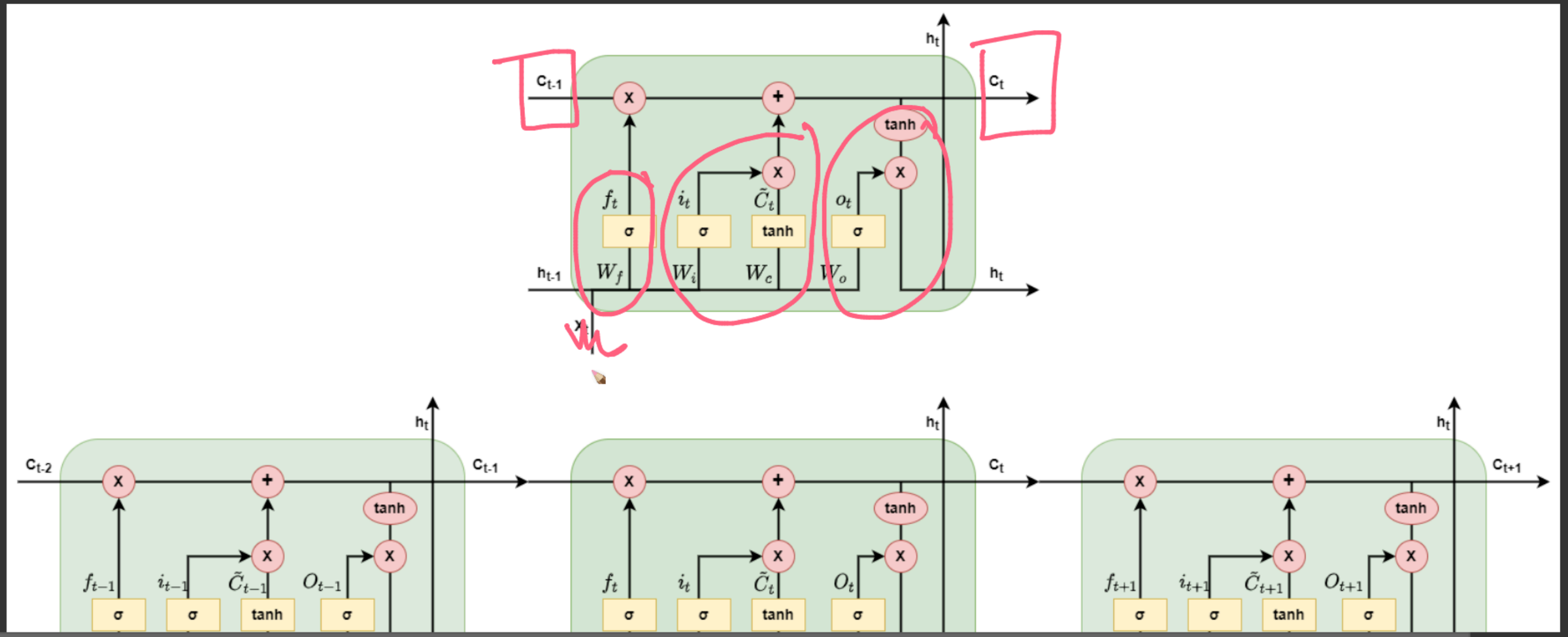
+ Code + Text Last saved at 16:21 Connect

A closer look at LSTM



58 / 58

A closer look at LSTM



How does LSTM mitigate the Vanishing/Exploding gradient problem?

- For a basic RNN, the term $\frac{\partial h_t}{\partial h_{t-1}}$ after a certain time starts to take values either > 1 or ~ 0 which is the root cause of exploding and vanishing gradient.
- In LSTM The cell state output: $C_t = (i_t * \tilde{C}_t) + (f_t * C_{t-1})$
- When we calculate the gradients for it

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

- The term $\frac{\partial C_t}{\partial C_{t-1}}$ does not have a fixed pattern and can take any positive value at any time step.
- Because of the sigmoid function at the gates who's value ranges from 0 to 1, the terms will not reach a very small value(converge to 0) or reach a very high value(Diverge).
- If the gradient begin to get small, then the weights of the gates are adjusted accordingly to get it closer to 1.
- All these happends during the training phase, were it learns when to let the gradient converge to 0 and when to preserve it.

What is being learnt?

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial W} +$$
$$\frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W} +$$
$$\frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial W} +$$
$$\frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial h_{t-3}}\frac{\partial h_{t-3}}{\partial W} +$$
$$\frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial h_{t-3}}\frac{\partial h_{t-3}}{\partial h_{t-4}}\frac{\partial h_{t-4}}{\partial W} +$$
$$\frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial h_{t-3}}\frac{\partial h_{t-3}}{\partial h_{t-4}}\frac{\partial h_{t-4}}{\partial h_{t-5}}\frac{\partial h_{t-5}}{\partial W}$$

Let's generalize this to a single formula

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial Z_t}\frac{\partial Z_t}{\partial h_t}\sum_{r=1}^{t}\frac{\partial h_t}{\partial h_r}\frac{\partial h_r}{\partial W}$$

Update the weight matrix W

$$W = W - \alpha\frac{\partial L_t}{\partial W}$$

How does LSTM mitigate the Vanishing/Exploding gradient problem?

- For a basic RNN, the term $\frac{\partial h_t}{\partial h_{t-1}}$ after a certain time starts to take values either > 1 or ~ 0 which is the root cause of exploding and vanishing gradient.
- In LSTM The cell state output: $C_t = (i_t * \tilde{C}_t) + (f_t * C_{t-1})$
- When we calculate the gradients for it

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

$C_t \simeq C_{t-1}$

- The term $\frac{\partial C_t}{\partial C_{t-1}}$ does not have a fixed pattern and can take any positive value at any time step.
- Because of the sigmoid function at the gates who's value ranges from 0 to 1, the terms will not reach a very small value(converge to 0) or reach a very high value(Diverge).
- If the gradient begin to get small, then the weights of the gates are adjusted accordingly to get it closer to 1.
- All these happends during the training phase, were it learns when to let the gradient converge to 0 and when to preserve it.

What is being learnt?

- If the gradient begin to get small, then the weights of the gates are adjusted accordingly to get it closer to 1.
- All these happends during the training phase, were it learns when to let the gradient converge to 0 and when to preserve it.

What is being learnt?

Similar to what we saw in the Vanilla RNN were there are 3 weight matrices, LSTM has 4 weight matrices that are being learning during the training process

- 1 going to the forget gate (W_f)
- 2 going to the input gate (W_i, W_c)
- 1 going to the output gate (W_o)

In simple terms everything that are maked in yellow rectangle box has a weight matrix.

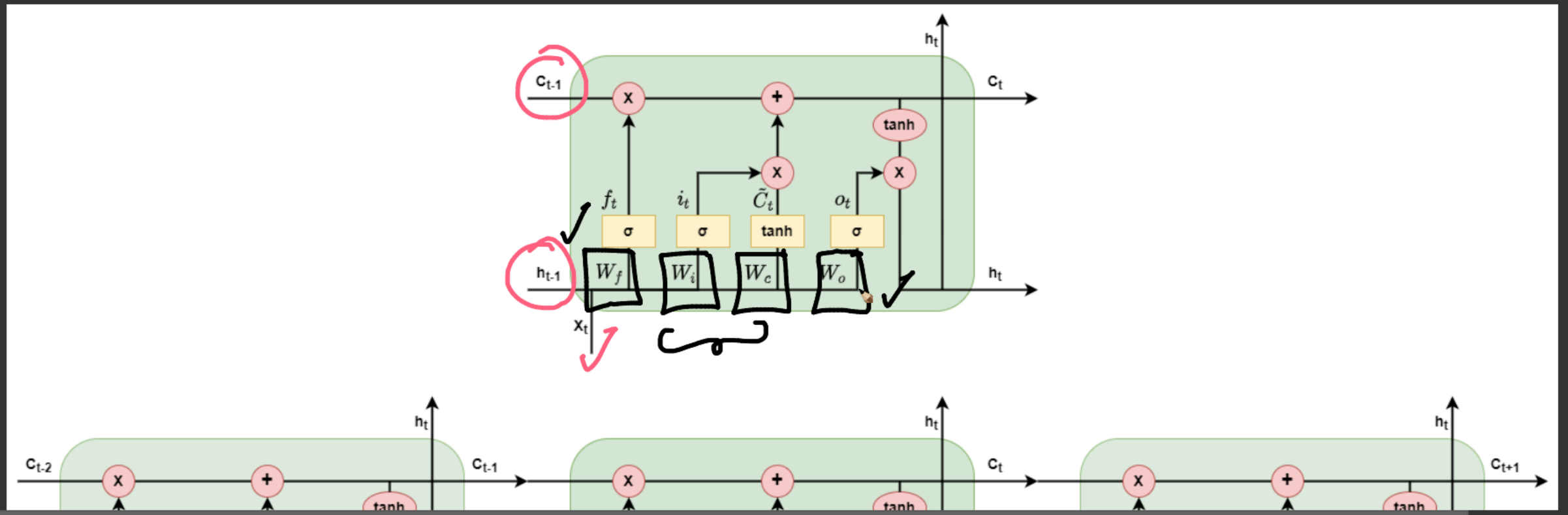
Variants of LSTM - GRU

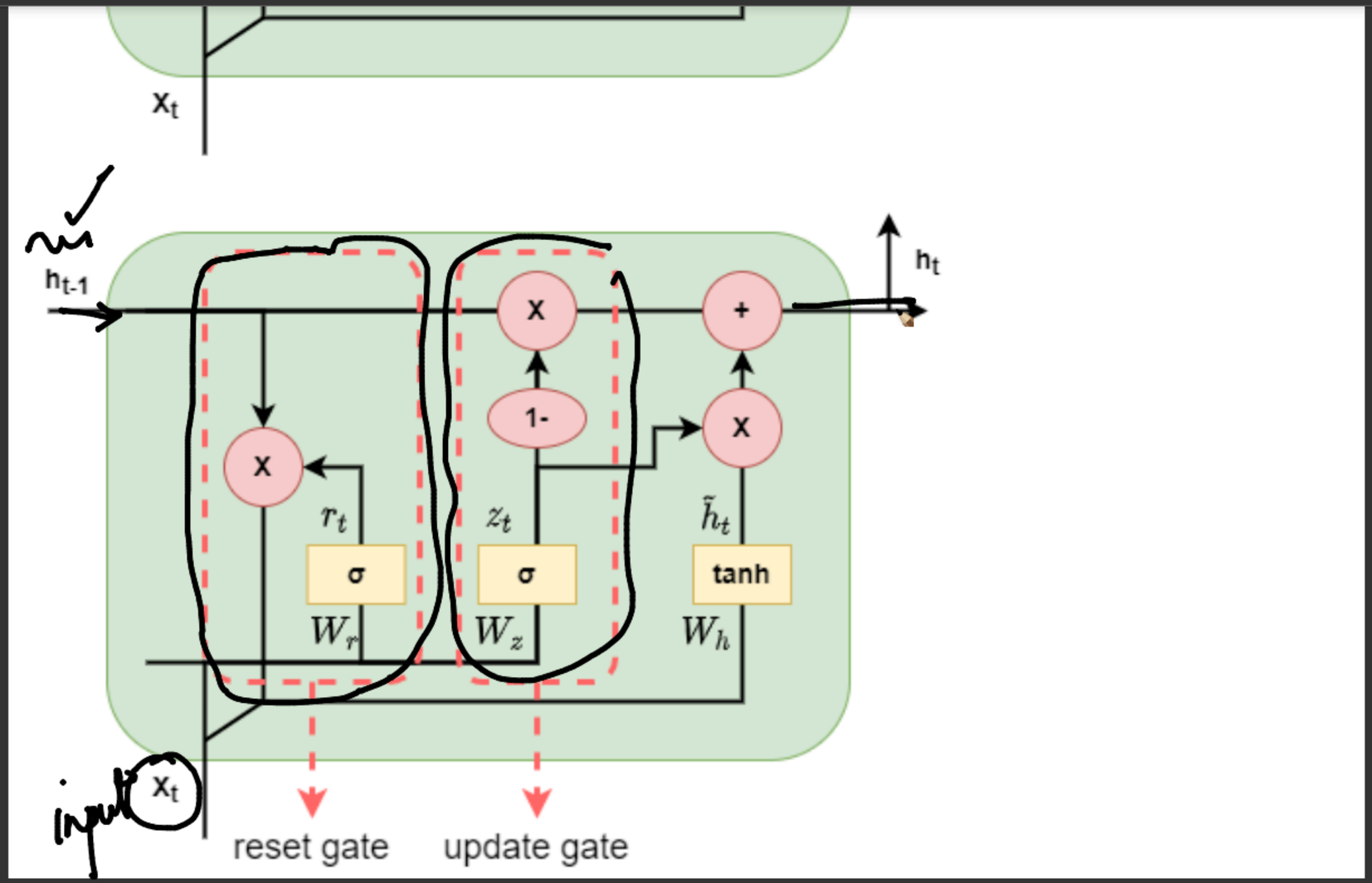
Another Variant of LSTM is the GRU

- It has 3 weight matrices W_r, W_z, W_h which is 1 weight matrix less than LSTM
- There is only one hidden state h_t in GRU unlike a separate cell state and hidden state in LSTM.

Note: fan_{in}, fan_{out} mean the number of neurons coming in and going out respectively.

A closer look at LSTM

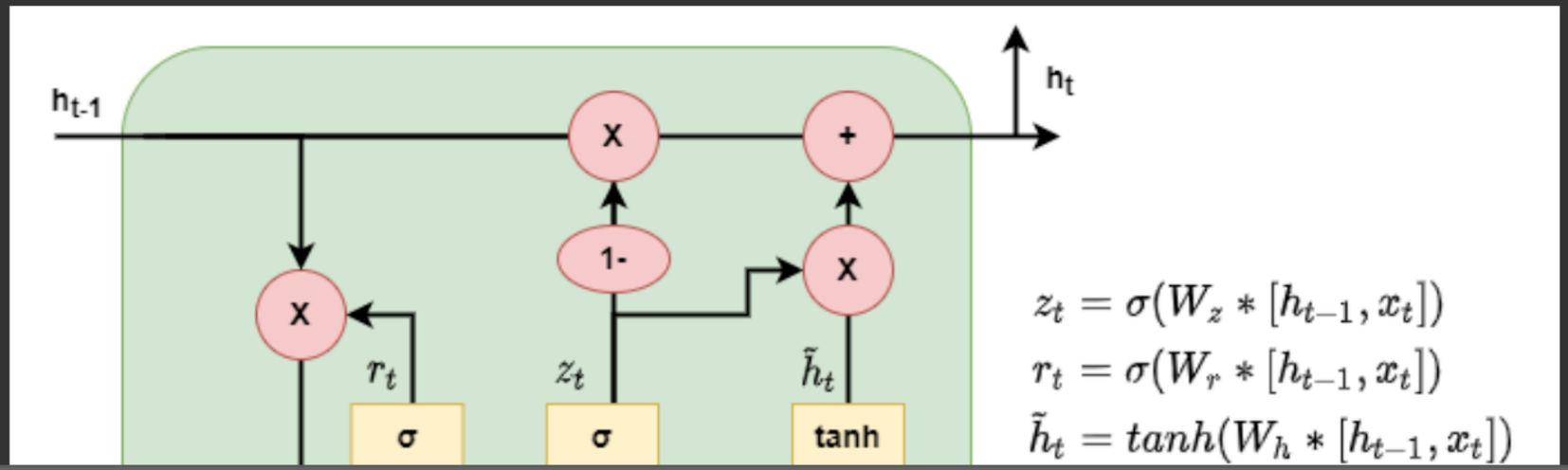




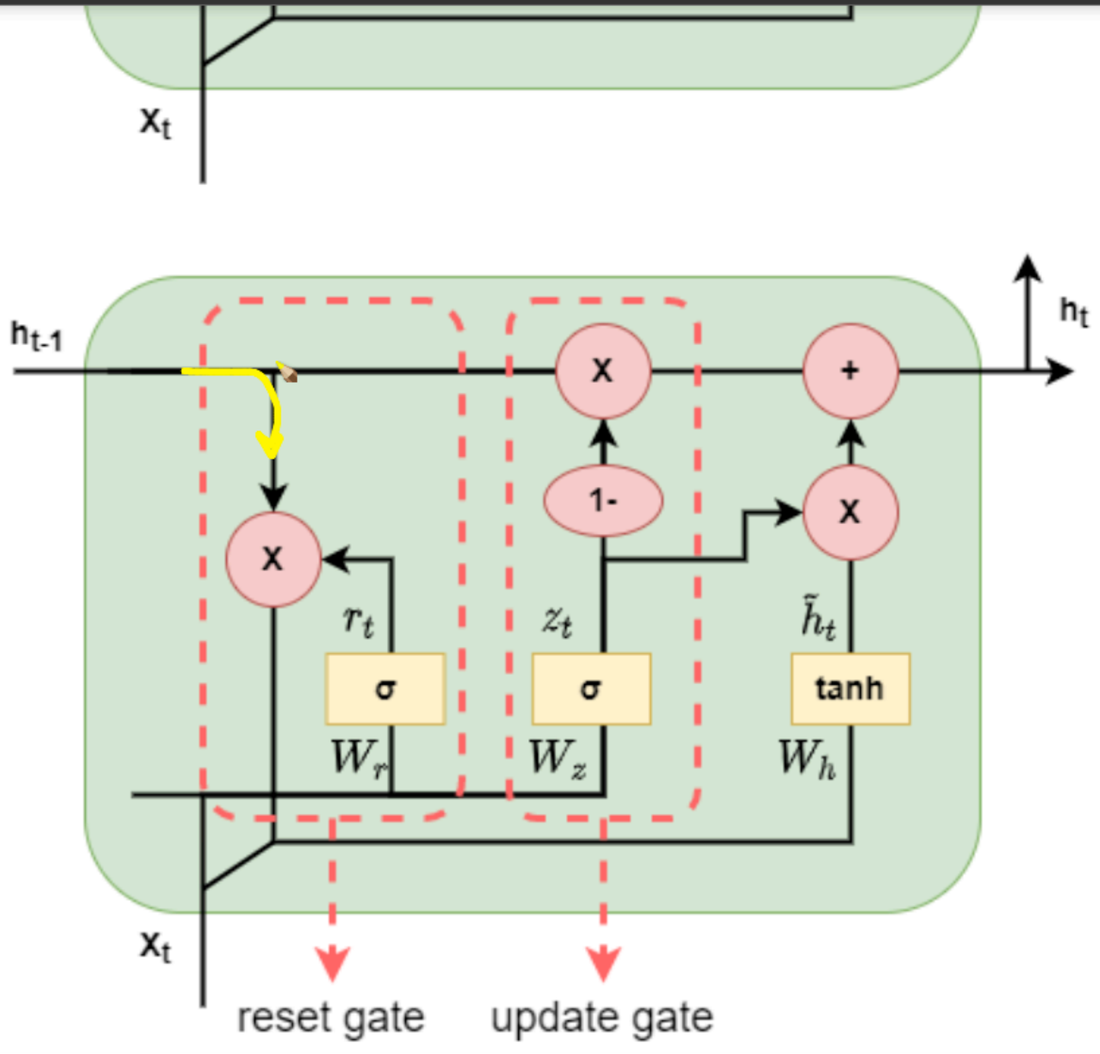
Variants of LSTM - GRU

Another Variant of LSTM is the GRU

- It has 3 weight matrices W_r, W_z, W_h which is 1 weight matrix less than LSTM
- There is only one hidden state h_t in GRU unlike a separate cell state and hidden state in LSTM.
- It has 2 gates a reset gate and a update gate.
- The update gate works similar to forget gate in LSTM as it determines what information to forget and what new information to update.
- The reset gate determines how much past information to forget



$$z_t = \sigma(W_z * [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r * [h_{t-1}, x_t])$$
$$\tilde{h}_t = tanh(W_h * [h_{t-1}, x_t])$$

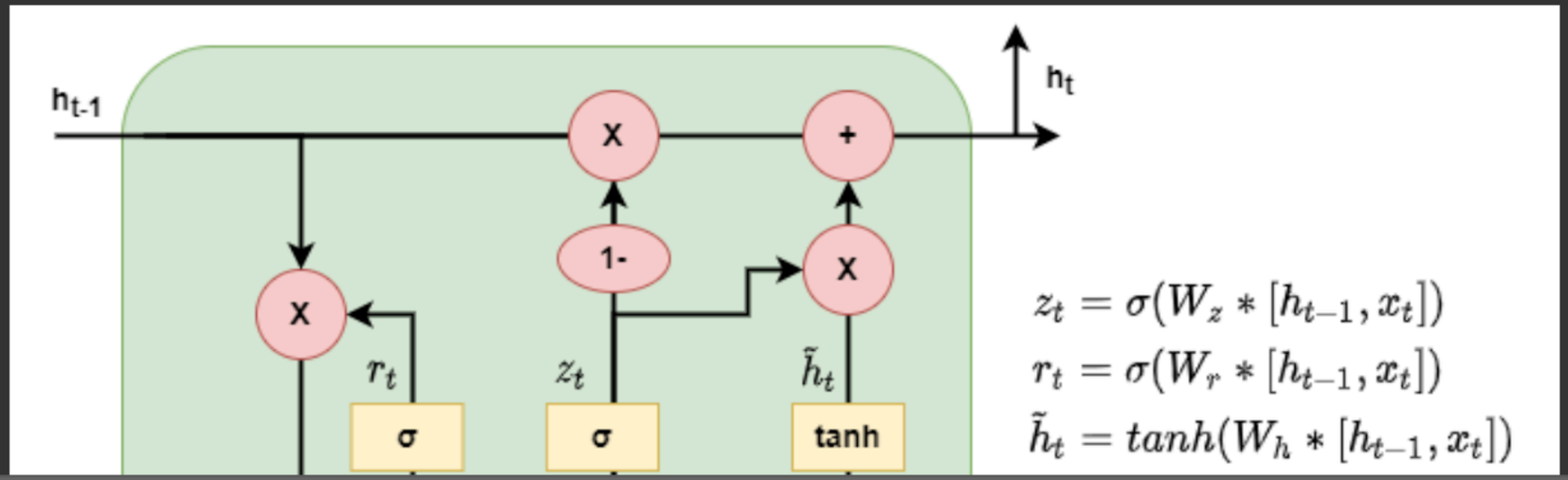


Variants of LSTM - GRU

GRU - adv

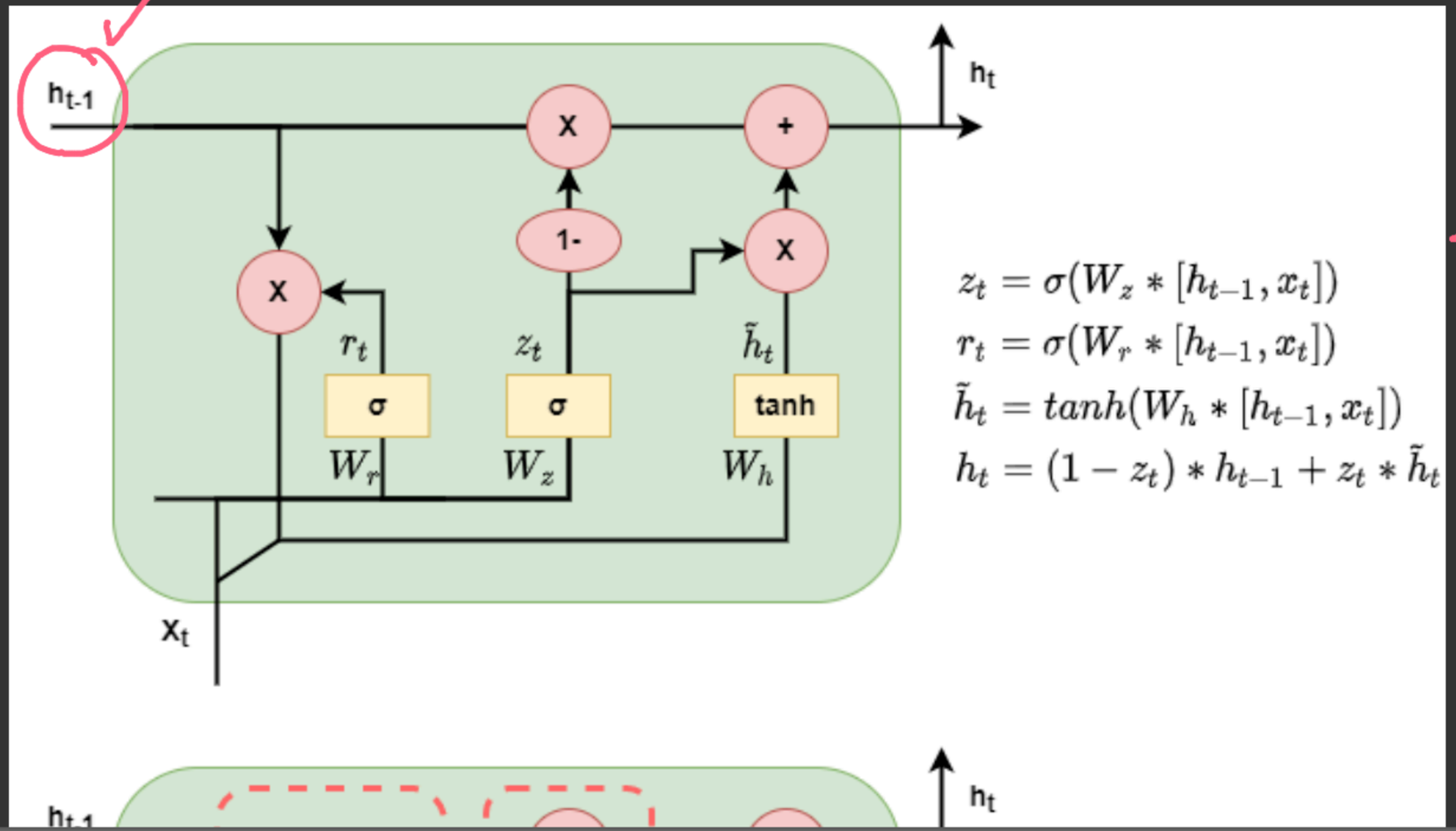
Another Variant of LSTM is the GRU

- It has 3 weight matrices W_r, W_z, W_h which is 1 weight matrix less than LSTM
- There is only one hidden state h_t in GRU unlike a separate cell state and hidden state in LSTM.
- It has 2 gates a reset gate and a update gate.
- The update gate works similar to forget gate in LSTM as it determines what information to forget and what new information to update.
- The reset gate determines how much past information to forget



$$z_t = \sigma(W_z * [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r * [h_{t-1}, x_t])$$
$$\tilde{h}_t = tanh(W_h * [h_{t-1}, x_t])$$

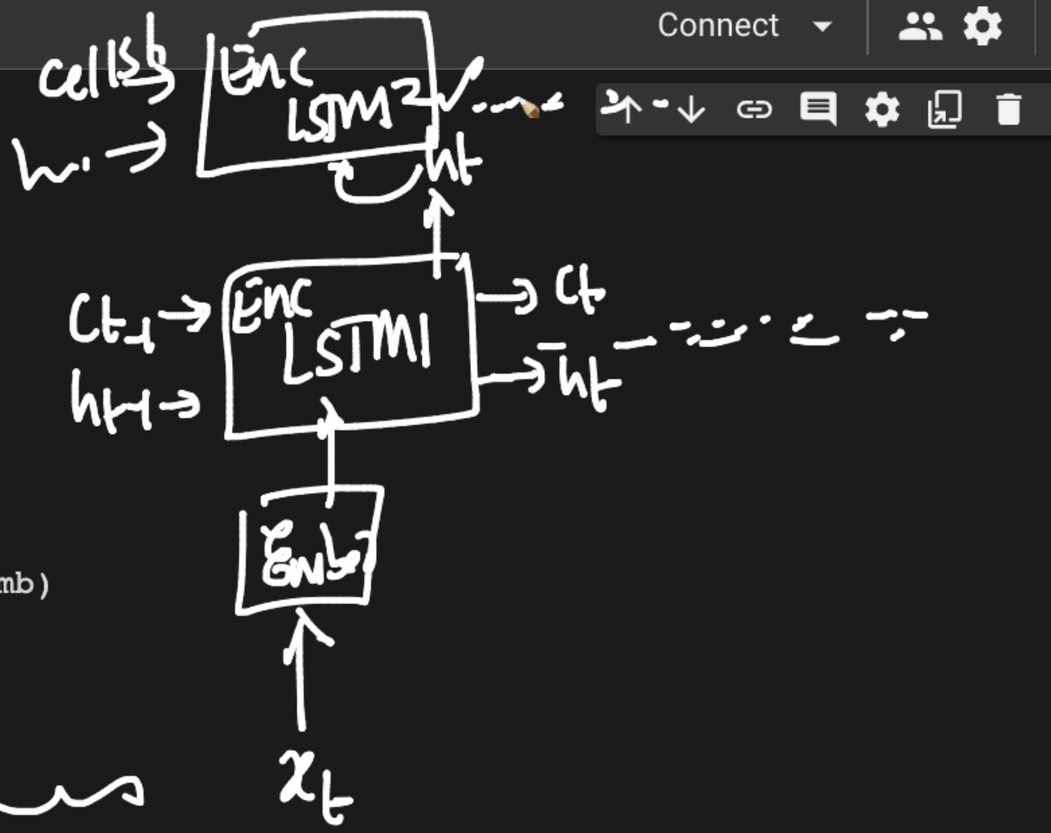
- The update gate works similar to forget gate in LSTM as it determines what information to forget and wha
- The reset gate determines how much past information to forget

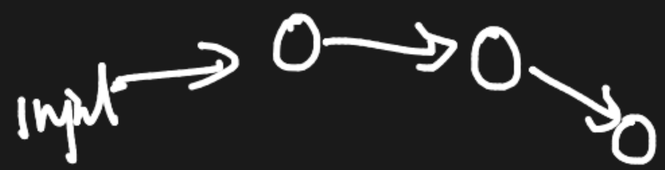


$$z_t = \sigma(W_z * [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r * [h_{t-1}, x_t])$$
$$\tilde{h}_t = tanh(W_h * [h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

fewer params
GRU
vs
LSTM
h_t, C_t, more params

```
# Encoder
encoder_inputs = Input(shape=(max_text_len, ))

# Embedding layer
enc_emb = Embedding(x_voc, embedding_dim,
                    trainable=True)(encoder_inputs)

# Encoder LSTM 1
encoder_lstm1 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output1, state_h1, state_c1) = encoder_lstm1(enc_emb)

# Encoder LSTM 2
encoder_lstm2 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output2, state_h2, state_c2) = encoder_lstm2(encoder_output1)

# Encoder LSTM 3
encoder_lstm3 = LSTM(latent_dim, return_state=True,
                     return_sequences=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder states as the initial state
```





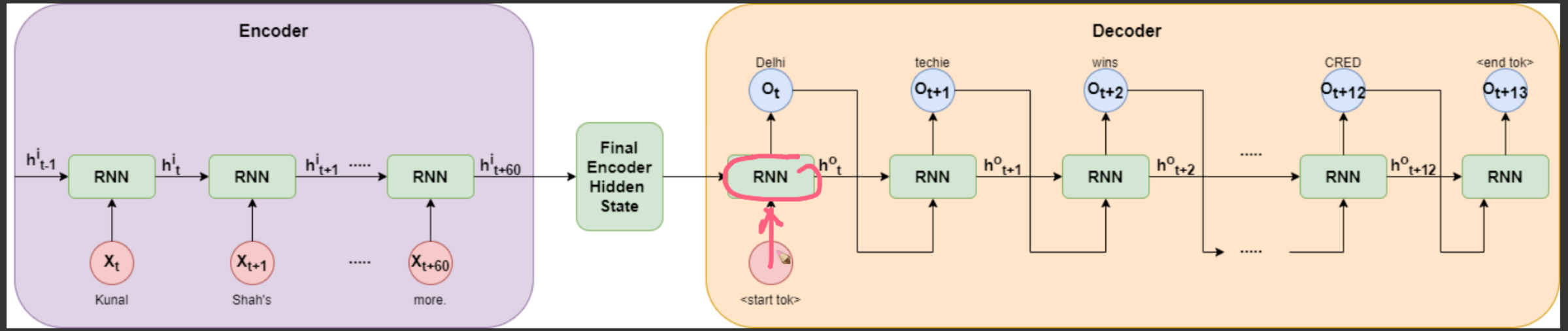
```
(encoder_output1, state_h1, state_c1) = encoder_lstm1(enc_emb)

# Encoder LSTM 2
encoder_lstm2 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output2, state_h2, state_c2) = encoder_lstm2(encoder_output1)

# Encoder LSTM 3
encoder_lstm3 = LSTM(latent_dim, return_state=True,
                     return_sequences=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder_states as the initial state
decoder_inputs = Input(shape=(None, ))

# Embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim, trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

# Decoder LSTM
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True, dropout=0.4,
                    recurrent_dropout=0.2)
```

What kind of problem is it?

- **Many to Many:** From the problem statement we can understand that it has multiple input and output, so it's a many to many problem.

Which architecture works well for this?

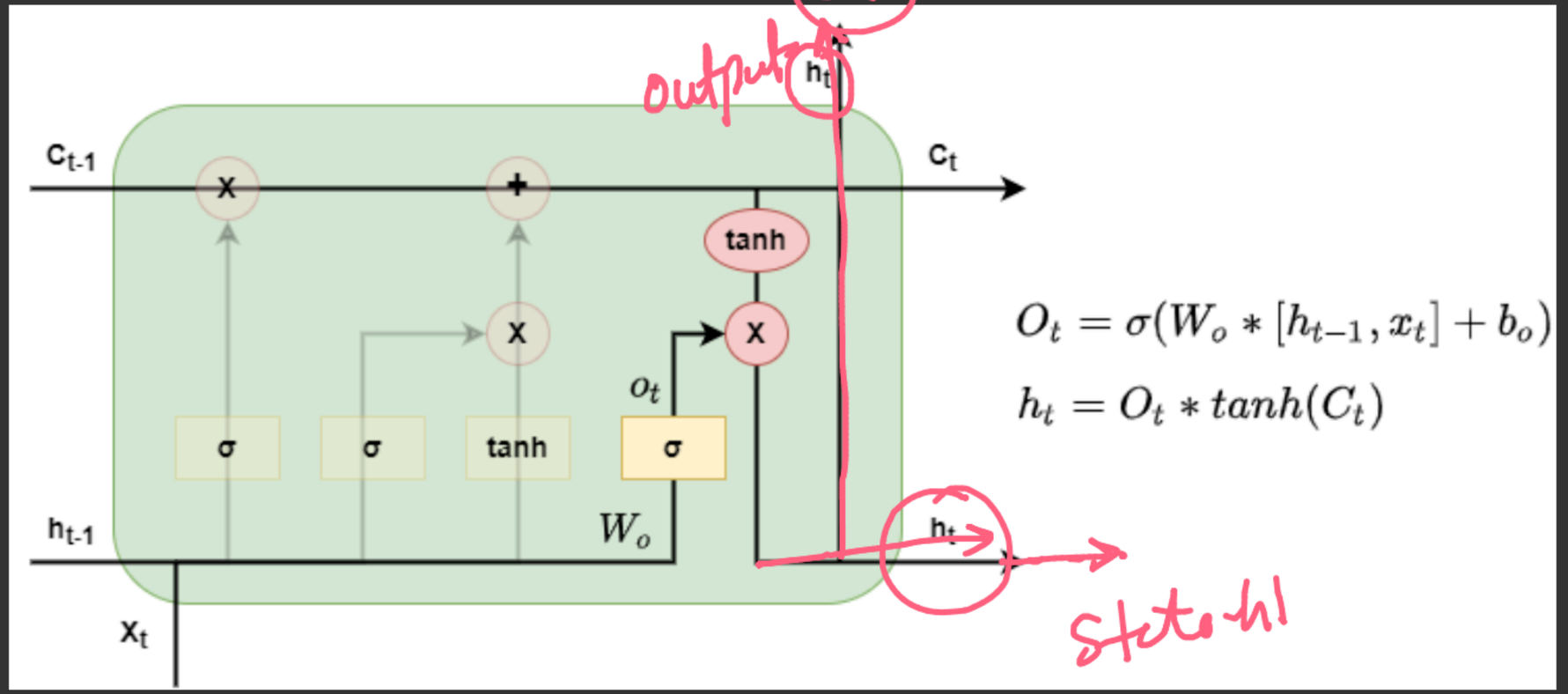
- **Encoder Decoder:** Since the input and the output lengths are not the same we will use the Encoder Decoder architecture



What is Long Term dependencies?

```
encoder_inputs = Input(shape=(max_text_len, ))

# Embedding layer
enc_emb = Embedding(x_voc, embedding_dim,
                    trainable=True)(encoder_inputs)

# Encoder LSTM 1
encoder_lstm1 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output1, state_h1, state_c1) = encoder_lstm1(enc_emb)

# Encoder LSTM 2
encoder_lstm2 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output2, state_h2, state_c2) = encoder_lstm2(encoder_output1)

# Encoder LSTM 3
encoder_lstm3 = LSTM(latent_dim, return_state=True,
                     return_sequences=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder_states as the initial state
```

Output gate

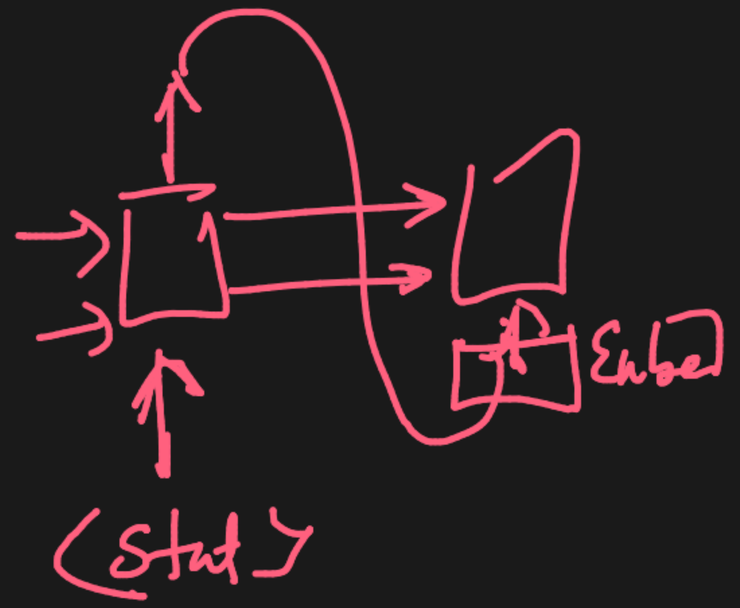


$$O_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = O_t * tanh(C_t)$$

- The output gate regulates the present hidden state h_t and it decides what relevant information has to be passed to the next hidden state
- It takes h_{t-1}, X_t as input to the sigmoid function

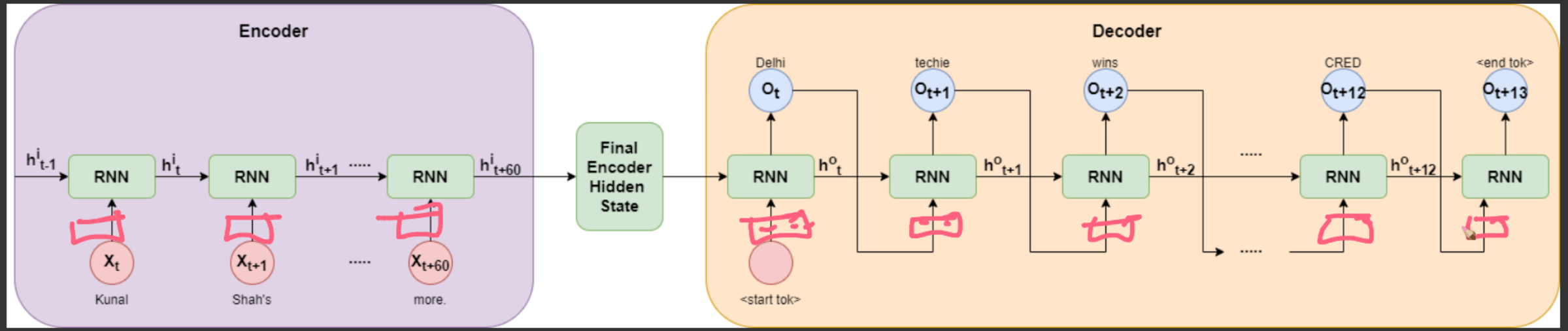
```
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder_states as the initial state
decoder_inputs = Input(shape=(None, ))

# Embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim, trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

# Decoder LSTM
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True, dropout=0.4,
                    recurrent_dropout=0.2)
(decoder_outputs, decoder_fwd_state, decoder_back_state) = \
    decoder_lstm(dec_emb, initial_state=[state_h, state_c])

# Dense layer
decoder_dense = TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```



```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic G
```

Which architecture works well for this?

- **Encoder Decoder:** Since the input and the output lengths are not the same we will use the Encoder Decoder architecture



What is Long Term dependencies?

For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

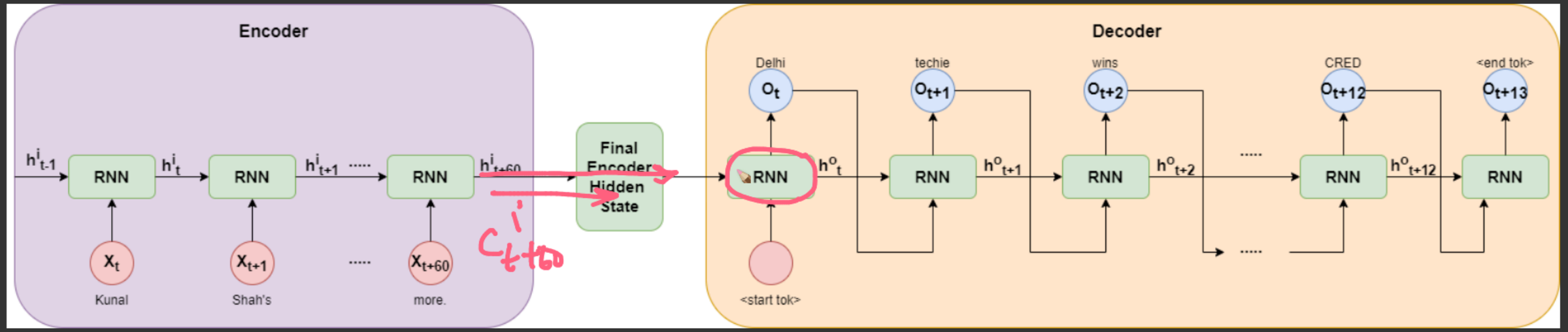
Sentence: I visited **Paris** last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and stayed at hotel near my office location. The weather was too cold for someone from a subcontinent. The trip lasted for 2 weeks and I travelled by metro for the most of my commute as the city is well connected. They were quite a lot of market area for shopping to buy souvenirs too. During the

```
                    recurrent_dropout=0.4)
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder_states as the initial state
decoder_inputs = Input(shape=(None, ))

# Embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim, trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

# Decoder LSTM
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True, dropout=0.4,
                    recurrent_dropout=0.2)
(decoder_outputs, decoder_fwd_state, decoder_back_state) = \
    decoder_lstm(dec_emb, initial_state=[state_h, state_c])

# Dense layer
decoder_dense = TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```

- **Many to Many:** From the problem statement we can understand that it has multiple input and output, so it's a many to many problem.

Which architecture works well for this?

- **Encoder Decoder:** Since the input and the output lengths are not the same we will use the Encoder Decoder architecture



What is Long Term dependencies?

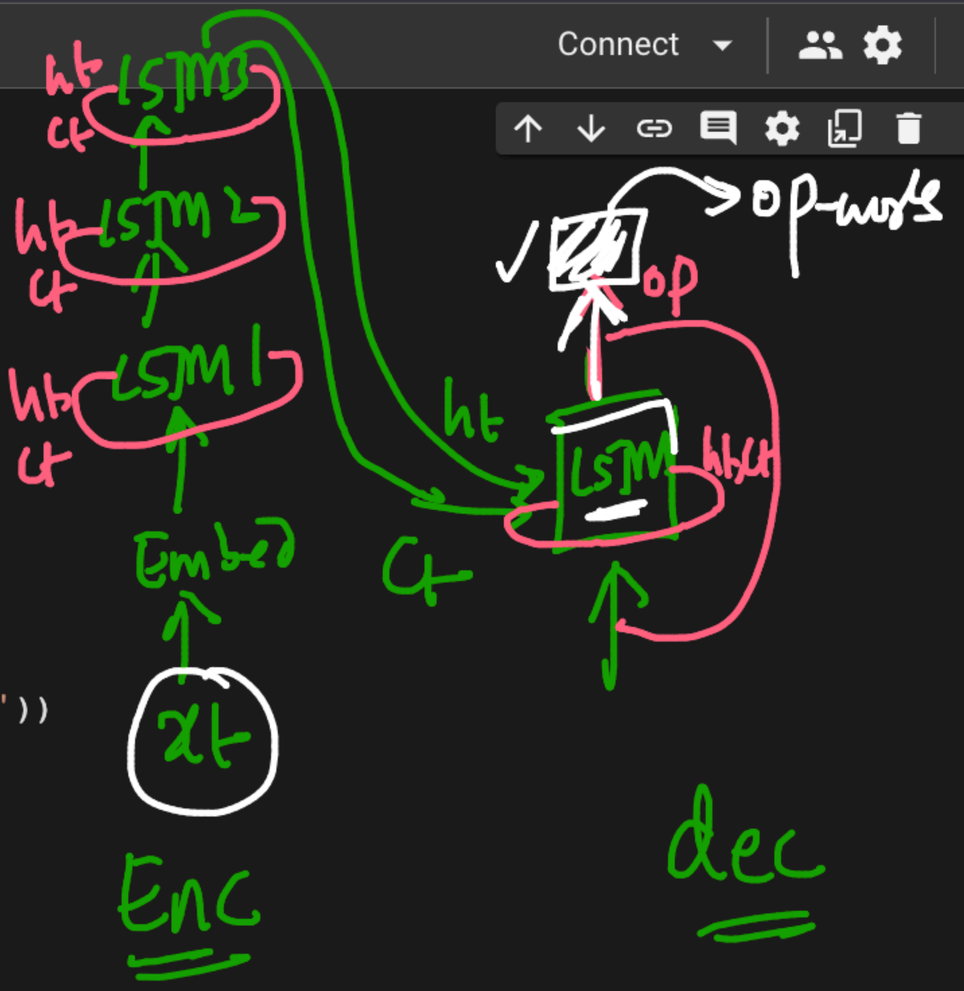
For the sake of understanding let's assume that we are performing a next word prediction task given the previous words.

Sentence: I visited Paris last year during winter as a part of the business trip. For the first 2 days I suferred from jet lag and staved at hotel

```
# Embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim, trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

# Decoder LSTM
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True, dropout=0.4,
                    recurrent_dropout=0.2)
(decoder_outputs, decoder_fwd_state, decoder_back_state) = \
    decoder_lstm(dec_emb, initial_state=[state_h, state_c])

# Dense layer
decoder_dense = TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic G
WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
WARNING:tensorflow:Layer lstm_3 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
Model: "model"
```

```
# Embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim, trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

# Decoder LSTM
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True, dropout=0.4,
                    recurrent_dropout=0.2)
(decoder_outputs, decoder_fwd_state, decoder_back_state) = \
    decoder_lstm(dec_emb, initial_state=[state_h, state_c])

# Dense layer
decoder_dense = TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```




```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic G
WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
WARNING:tensorflow:Layer lstm_3 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic
Model: "model"
```

```
# Embedding layer
enc_emb = Embedding(x_voc, embedding_dim,
                    trainable=True)(encoder_inputs)

# Encoder LSTM 1
encoder_lstm1 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output1, state_h1, state_c1) = encoder_lstm1(enc_emb)

# Encoder LSTM 2
encoder_lstm2 = LSTM(latent_dim, return_sequences=True,
                     return_state=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_output2, state_h2, state_c2) = encoder_lstm2(encoder_output1)

# Encoder LSTM 3
encoder_lstm3 = LSTM(latent_dim, return_state=True,
                     return_sequences=True, dropout=0.4,
                     recurrent_dropout=0.4)
(encoder_outputs, state_h, state_c) = encoder_lstm3(encoder_output2)

# Set up the decoder, using encoder_states as the initial state
decoder_inputs = Input(shape=(None, ))
```