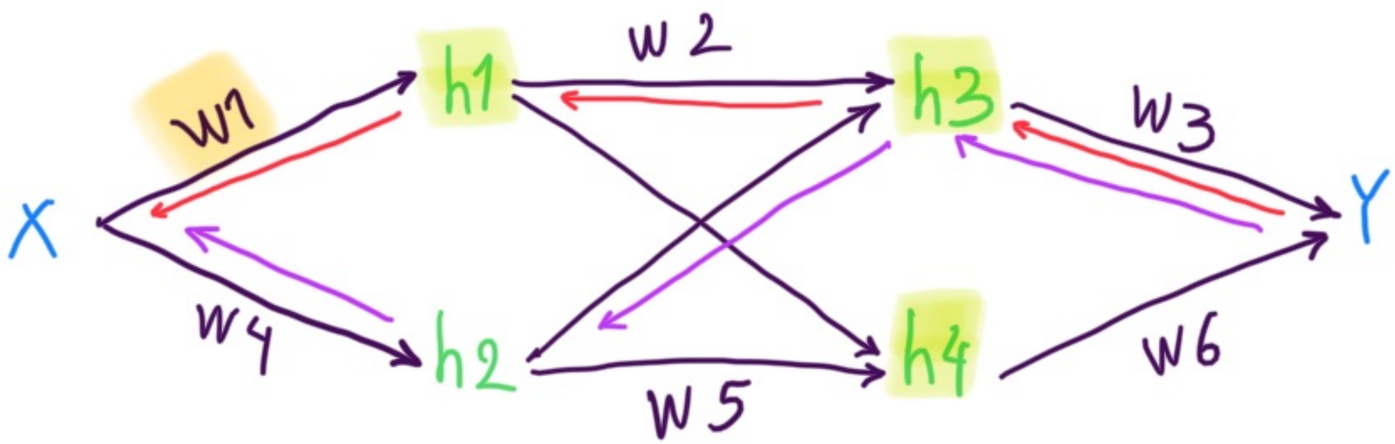
Some Terms and Background

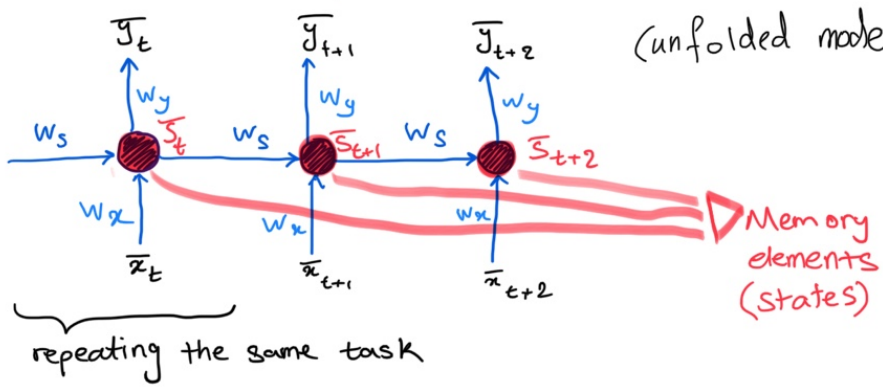
- Why is mini-batch training good?
  1) reduces the complexity of training process
  2) reduces noise

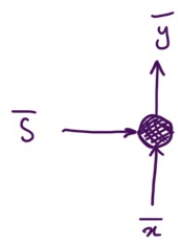
- What is the update rule for weight matrix W1?



$$\frac{\partial y}{\partial w_1} = \underbrace{\frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial w_1}}_{\text{red path}} + \underbrace{\frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial w_1}}_{\text{purple path}}$$

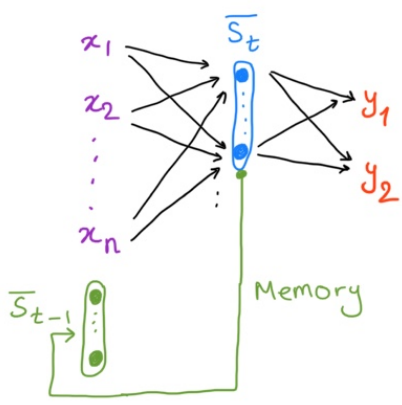
Recurrent Neural Nets





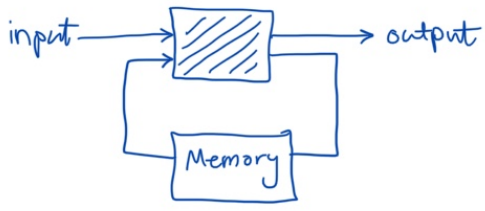
$\bar{y}$ depends on both { $\bar{x}$ input; $\bar{s}$ memory (past input)

RNN illustrated differently:

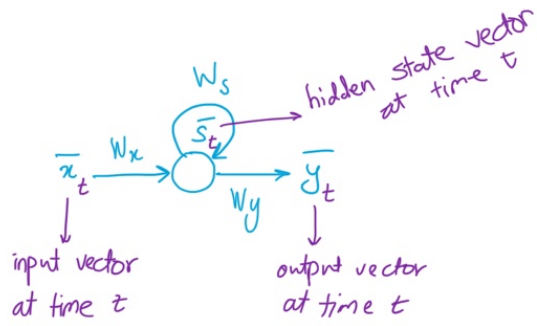


← Simple (Elman) RNN

RNN as State Machine



or, more simply:
(folded model)

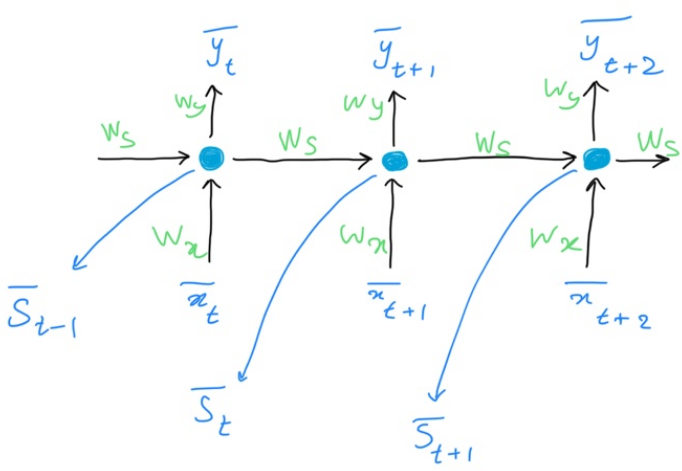


activation function

$$\bar{s}_t = \phi(\bar{x}_t \cdot W_x + \bar{s}_{t-1} \cdot W_s)$$

$$\bar{y}_t = \bar{s}_t \cdot W_y$$

Unfolded representation is cleaner and easier to understand.



* RNNs can handle varying lengths of inputs.

Training RNNs

In RNN: Back Propagation Through Time (BPTT).

$$\bar{S}_t = \Phi\left(\bar{x}_t \cdot W_x + \bar{S}_{t-1} \cdot W_S\right)$$

e.g., $$\bar{S}_t = \tanh\left(\bar{x}_t \cdot W_x + \bar{S}_{t-1} \cdot W_S\right)$$

$$\bar{y}_t = \underset{\text{Softmax}}{\sigma}\left(\bar{S}_t \cdot W_y\right)$$

$$E_t = \left(\bar{d}_t - \bar{y}_t\right)^2$$ error function is MSE

How does BPTT work?

$t = 3$

$$E_3 = (\bar{d}_3 - \bar{y}_3)^2$$

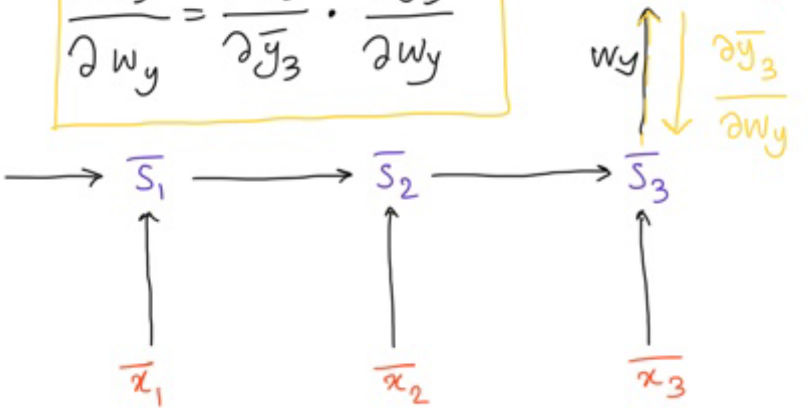
$\bar{d}_3$: desired output; $\bar{y}_3$: Calculated output

but we also need info from $t=2$ and $t=1$.

W_y

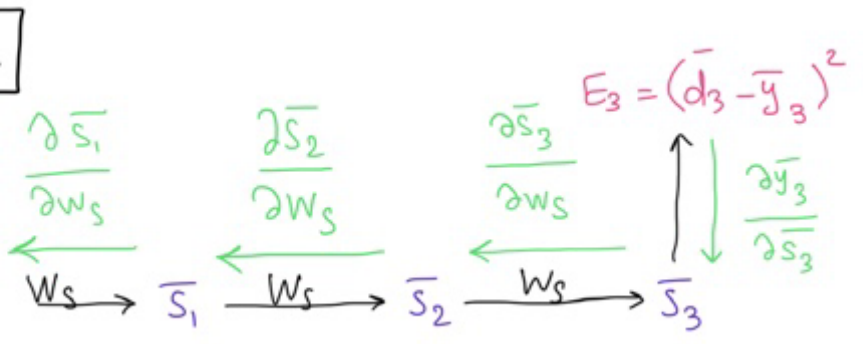
$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial W_y}$$

$E_3 = (\bar{d}_3 - \bar{y}_3)^2$



W_S

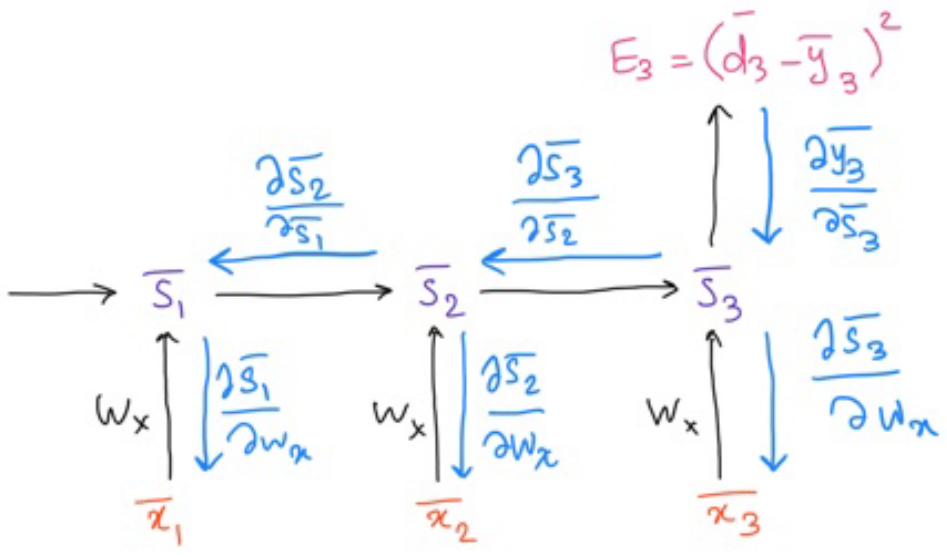
$E_3 = (\bar{d}_3 - \bar{y}_3)^2$



$$\frac{\partial E_3}{\partial W_S} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{S}_3} \cdot \frac{\partial \bar{S}_3}{\partial W_S} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{S}_3} \cdot \frac{\partial \bar{S}_3}{\partial \bar{S}_2} \cdot \frac{\partial \bar{S}_2}{\partial W_S} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{S}_3} \cdot \frac{\partial \bar{S}_3}{\partial \bar{S}_2} \cdot \frac{\partial \bar{S}_2}{\partial \bar{S}_1} \cdot \frac{\partial \bar{S}_1}{\partial W_S}$$

More generally: $$\frac{\partial E_N}{\partial W_S} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{S}_i} \cdot \frac{\partial \bar{S}_i}{\partial W_S}$$

W_x



$$\frac{\partial E_3}{\partial w_x} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial w_x}$$

$$+ \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial w_x}$$

$$+ \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial w_x}$$

More generally : $\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \cdot \frac{\partial \bar{s}_i}{\partial w_x}$

- for training : updating weights every N steps

  mini-batch

- what happens if we have too many steps?
  up to 8-10 steps, this BPTT works, but beyond that, the vanishing gradient problem happens. → LSTM is born!

- Gradient clipping : → avoiding Exploding Gradient Problem.

  At each timestep t :

  $\delta = \frac{\partial y}{\partial W_{ij}} >$ threshold ?

  If so, normalize the gradients.

Implement RNN in PyTorch

(a simple time-series prediction)

```
# Initial Reqs
from torch import nn
import numpy as np
import matplotlib.pyplot as plt

# stuff for visualizing the data
plt.figure (figsize = (8, 5))
seq_length = 20 → # of data points in each batch
time_steps = np.linspace (0, np.pi, seq_length + 1)
data = np.sin (time_steps)
data.resize ((seq_length + 1, 1)) → adds a dimension

x = data [:-1]
y = data [1:]

plt.plot (time_steps[1:], x, 'r.', label='input, x')
plt.plot (time_steps [1:], y, 'b.', label='target, y')
plt.legend (loc='best')
plt.show ()

# Define RNN
class RNN (nn.Module):
    def __init__ (self, input_size, output_size,
                  hidden_dim, n_layers):
        super (RNN, self).__init__ ()
        self.hidden_dim = hidden_dim

Actual RNN! ← self.rnn = nn.RNN (input_size, hidden_dim,
                                 n_layers, batch_first=True)
        self.fc = nn.Linear (hidden_dim, output_size)

batch size X
seq_len X
input_size

    def forward (self, x, hidden):
        batch_size = x.size (0)

hidden state
n_layers X
batch_size X
hidden_dim

        r_out, hidden = self.rnn (x, hidden)

RNN output
batch size X time_step X hidden_dim

        r_out = r_out.view (-1, self.hidden_dim)

new shape :
batch_size * timeStep X hidden dim
We are doing some sort of flattening for FC layer.

        output = self.fc (r_out)

        return output, hidden

# Let's test the RNN !
my_rnn = RNN (input_size=1, output_size=1,
              hidden_dim=10, n_layers=2)

time_steps = np.linspace (0, np.pi, seq_length)
data = np.sin (time_steps)
data.resize ((seq_length, 1))

add a dim for batch.

test_input = torch.Tensor (data).unsqueeze (0)

out, h = my_rnn (test_input, None)
```

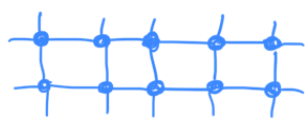
```
# Training our RNN
input_size = 1
output_size = 1
hidden_dim = 32
n_layers = 1                                   } hyperparams

rnn = RNN(input_size, output_size,
          hidden_dim, n_layers)

Loss & Opt.
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=.01)
                  standard for RNNs

# Training Loop
def train(rnn, n_steps, print_every):
    hidden = None

    for i, step in enumerate(range(n_steps)):
        time_steps = np.linspace(step * np.pi,
                                 (step+1)*np.pi,
                                 seq_length + 1)
        data = np.sin(time_steps)
        data.resize((seq_length+1, 1))

        x = data[:-1]
        y = data[1:]

        x_tensor = torch.Tensor(x).unsqueeze(0)
        y_tensor = torch.Tensor(y)

        prediction, hidden = rnn(x_tensor, hidden)

        # ~~~~~~~~~ #
        # ~ representing memory ~ #
        hidden = hidden.data
avoid backpropagating through the entire history

        loss = criterion(prediction, y_tensor)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Some fancy visualization
        if i % print_every == 0:
            plt.plot(time_steps[1:], x, 'r.')
            plt.plot(time_steps[1:],
                     prediction.data.numpy().flatten(),
                     'b.')
            plt.show()

n_steps = 75
print_every = 15
trained_rnn = train(rnn, n_steps, print_every)
```

LSTM

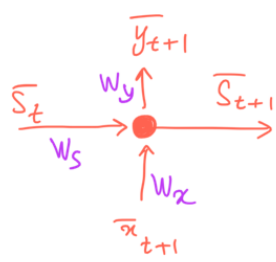
or long short-term memory cell

let's Compare LSTM and RNN

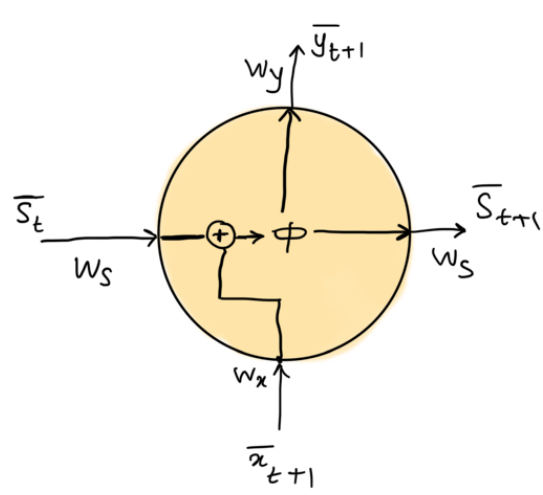
RNN



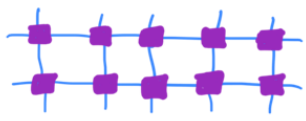
Zooming In



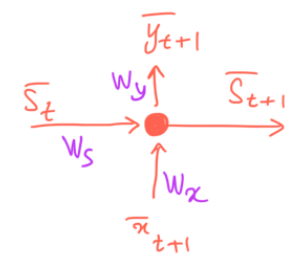
Zooming In



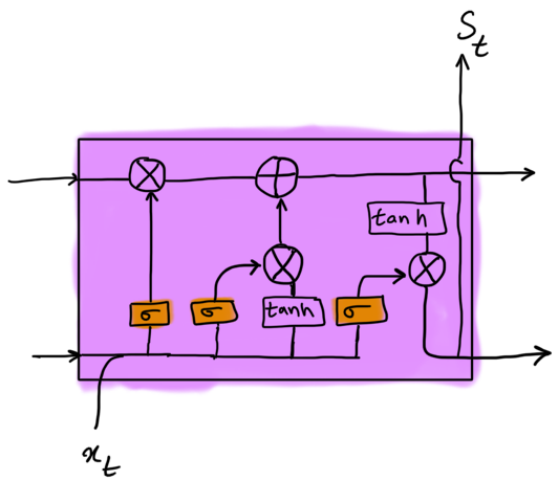
LSTM



Zooming In



Zooming In

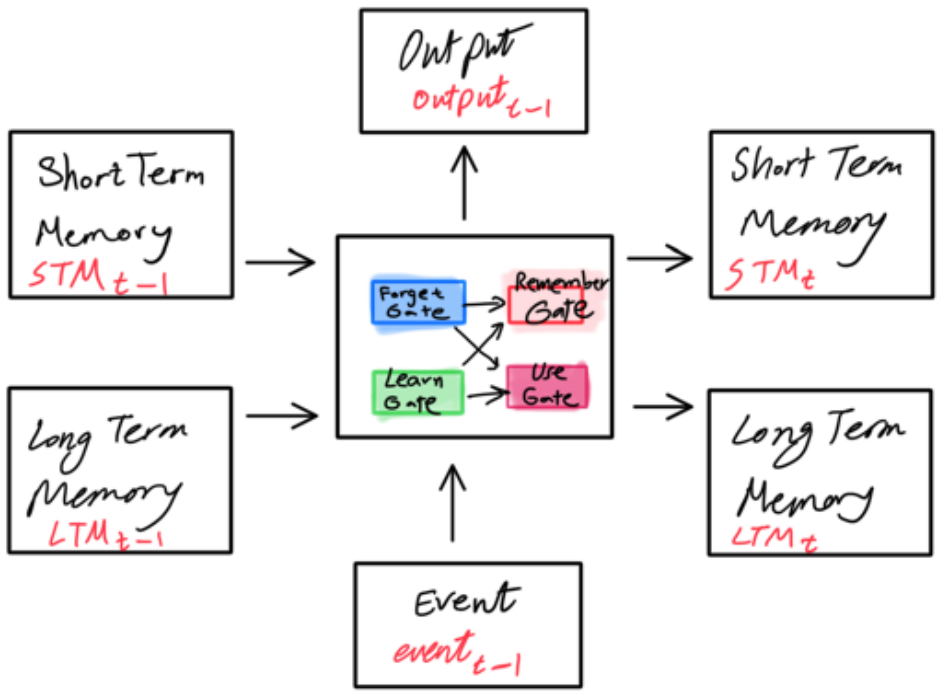


Gating functions

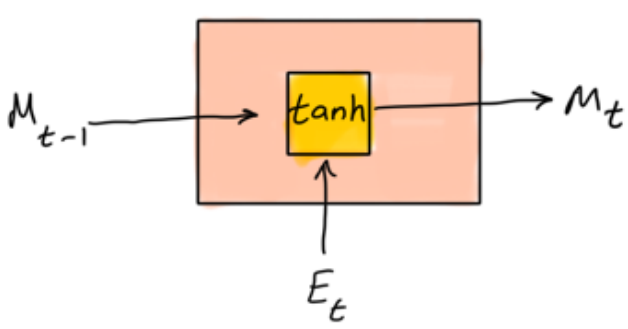
- $\sigma(x) \approx 1$ — All data passes through
- $\sigma(x) \approx 0$ — No data passes through

They decide what data to retain.

LSTM Explained (visually!)

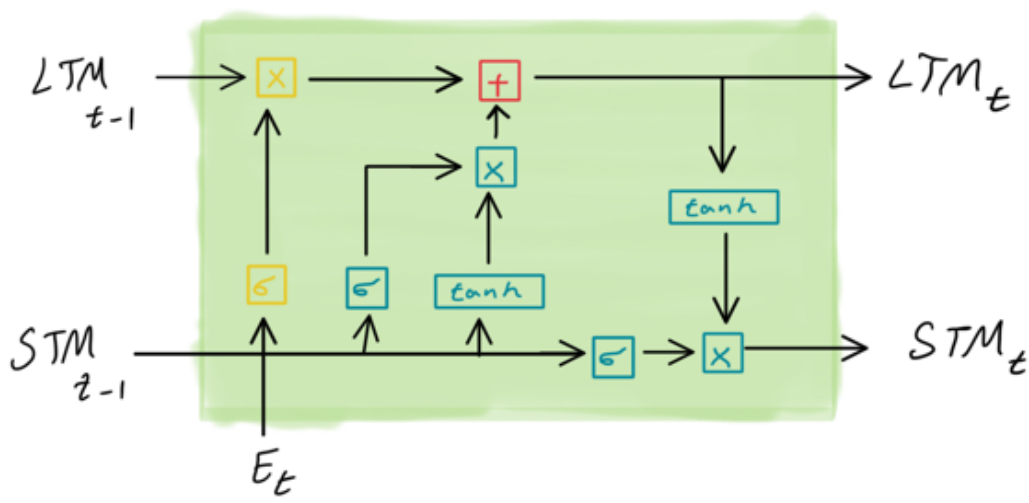


Reminder for RNN

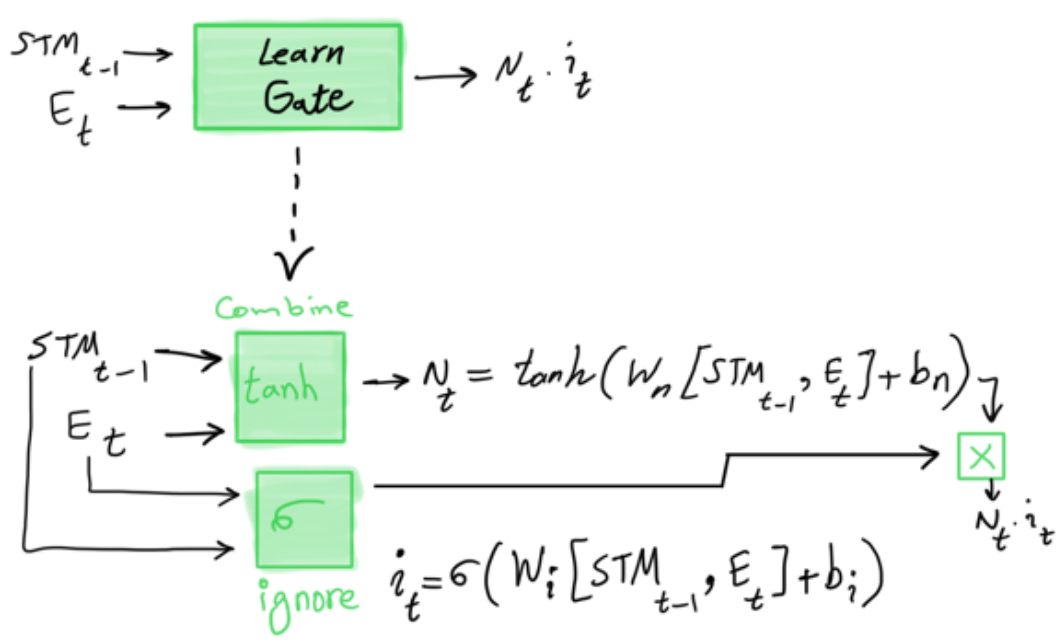


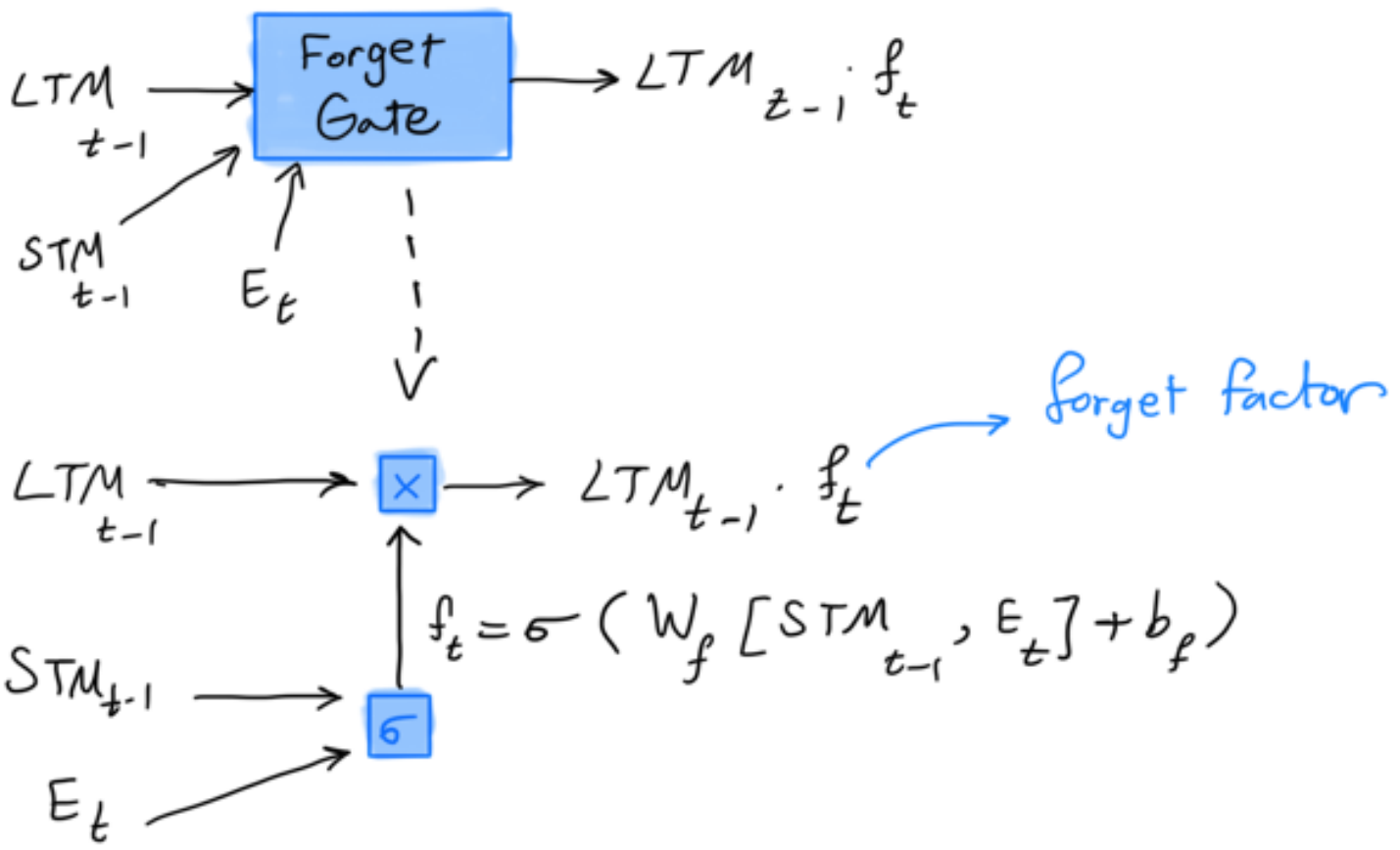
$$M_t = \tanh\left(W[STM_{t-1}, E_t] + b\right)$$

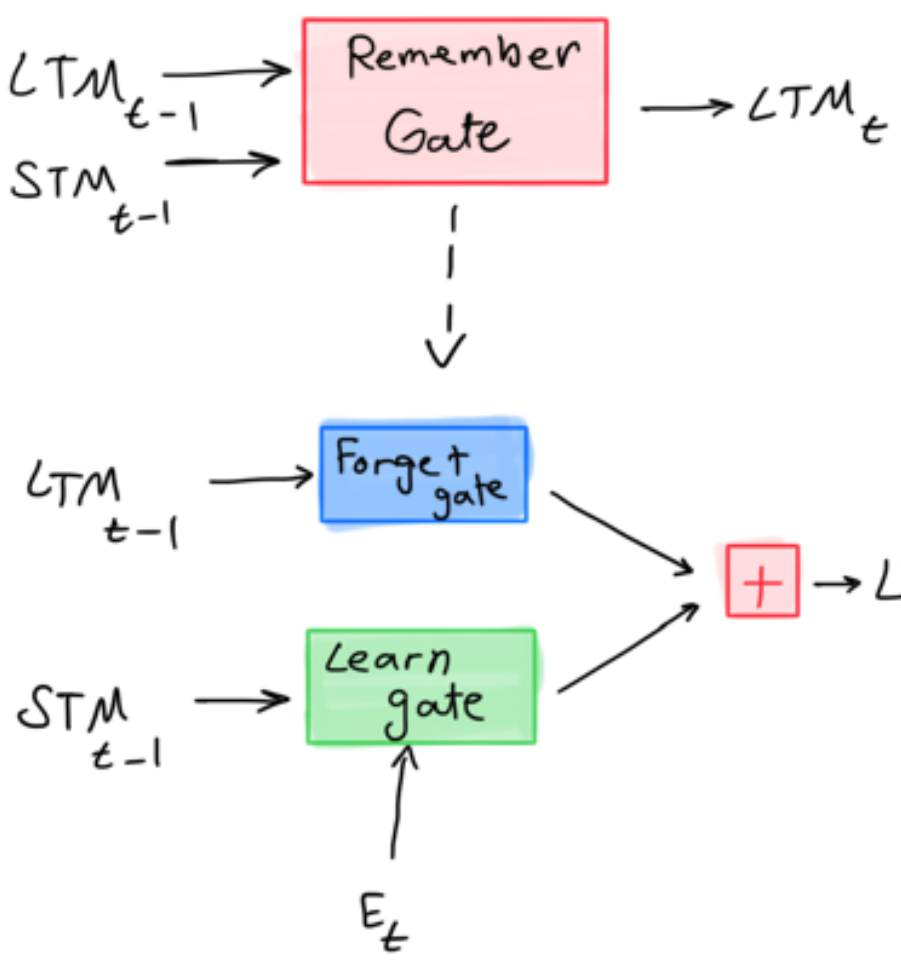
So now let's see how LSTM looks like:

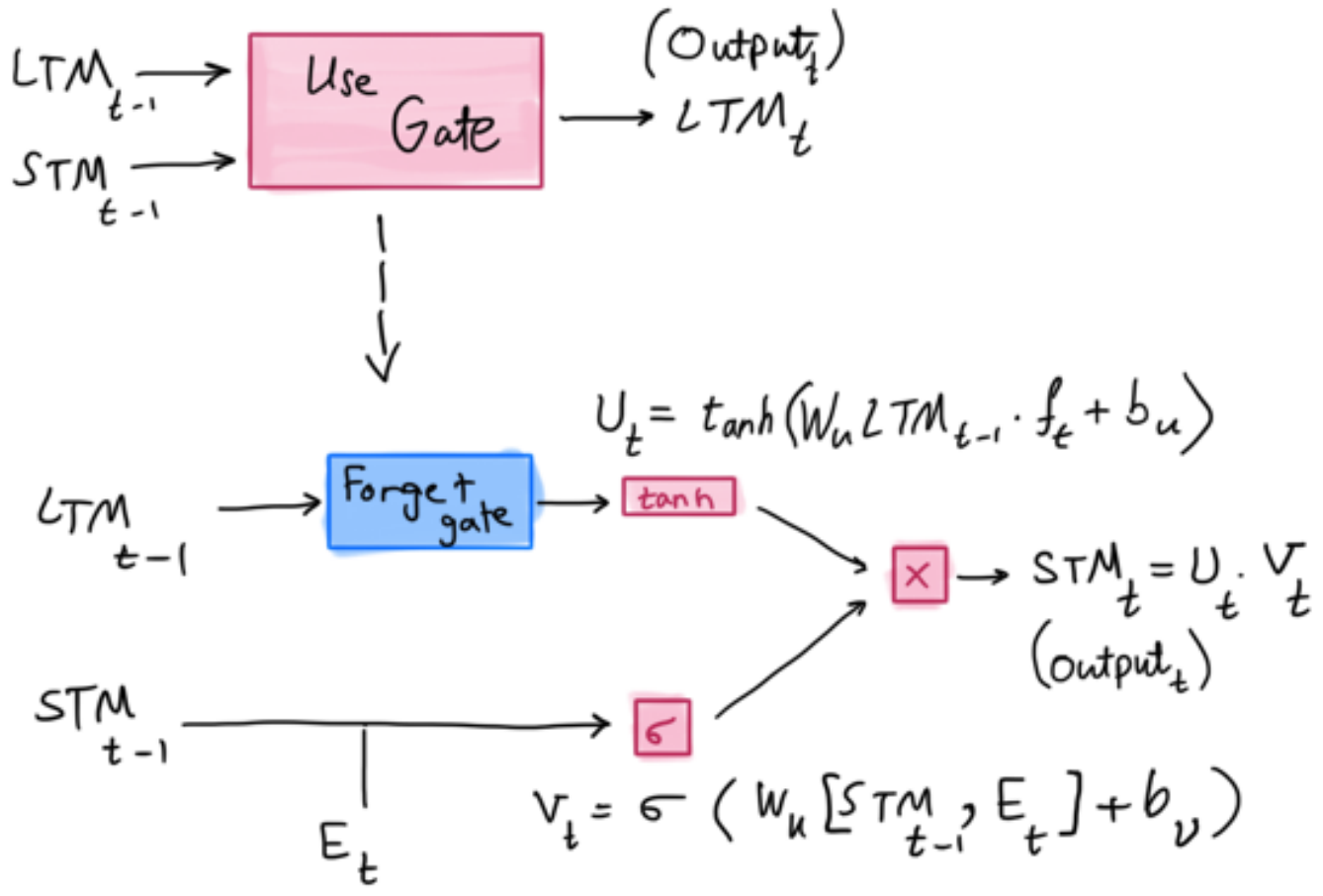


Let's look at each component individually

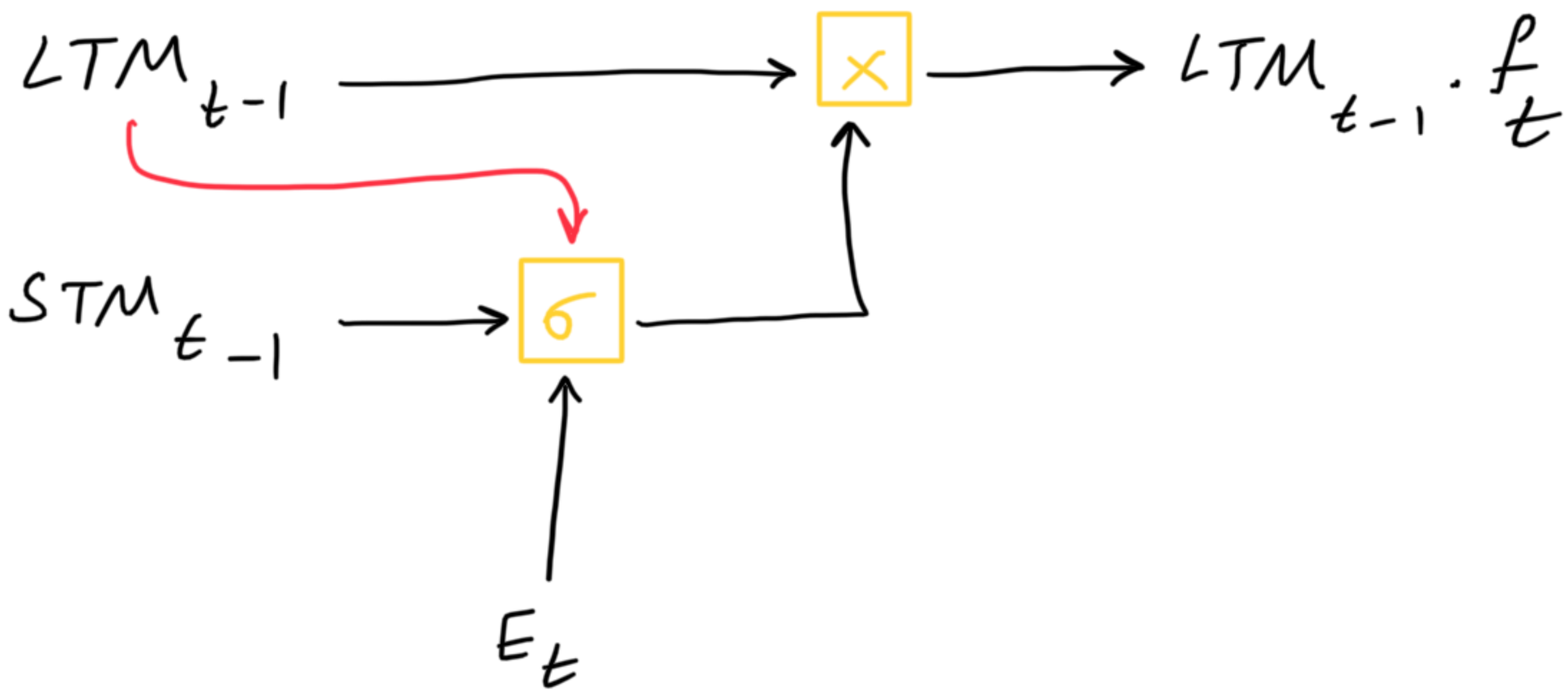



Forget Gate
LTM t-1
STM t-1
E t
LTM t-1 · f t
forget factor
LTM t-1
STM t-1
E t
f t = σ (W f [STM t-1 , E t] + b f)


Remember Gate
LTM t-1
STM t-1
LTM t
Forget gate
Learn gate
LTM t-1
STM t-1
E t
LTM t = LTM t-1 . f t + N t . i t

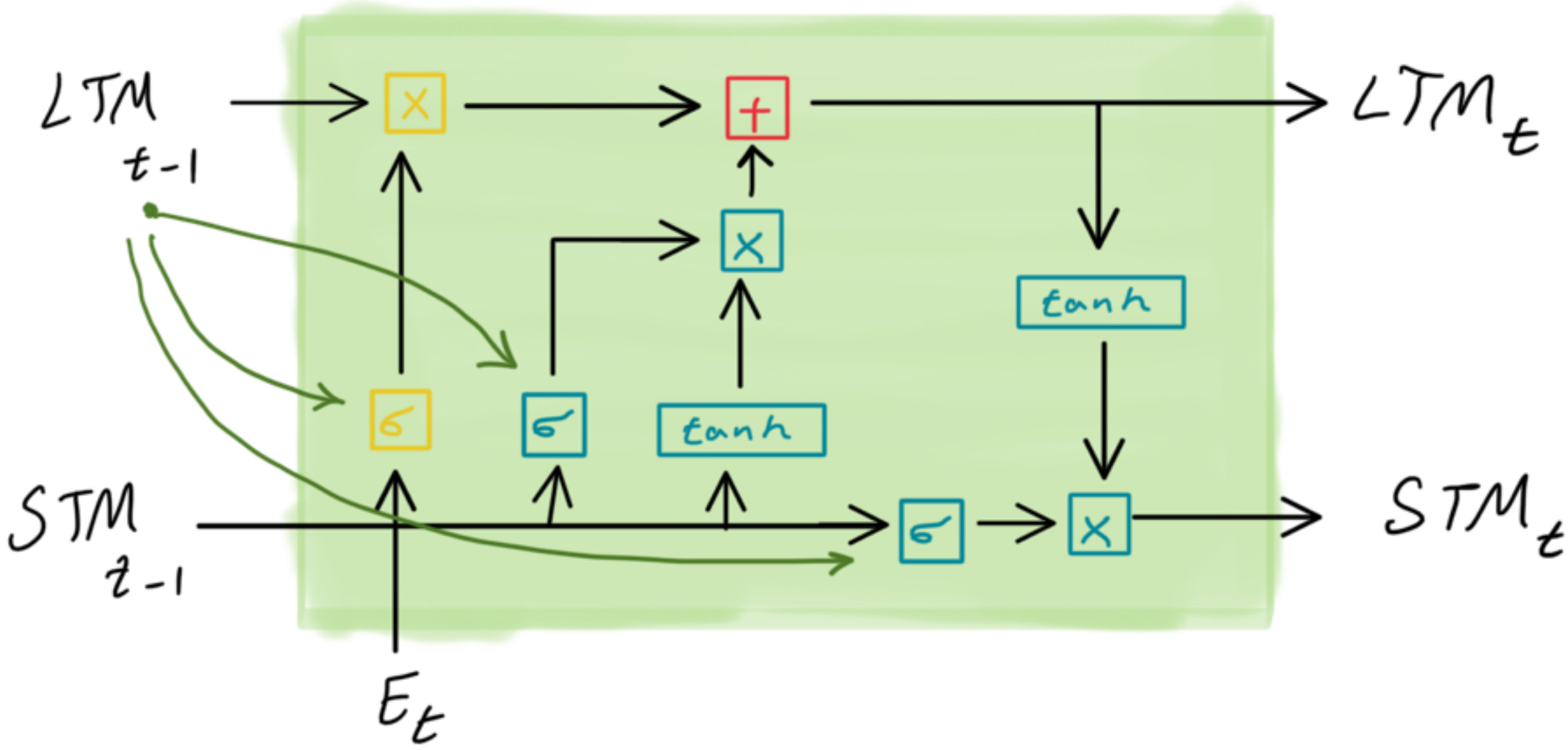

Use Gate
(Output t)
LTM t-1
STM t-1
LTM t
U t = tanh (W u LTM t-1 · f t + b u)
Forget gate
tanh
LTM t-1
STM t-1
E t
V t = σ (W u [STM t-1 , E t] + b v)
STM t = U t . V t
(Output t)

Peephole Connections



w/ The added edge: $f_t = \sigma\left(W_f[LTM_{t-1}, STM_{t-1}, E_t] + b_f\right)$

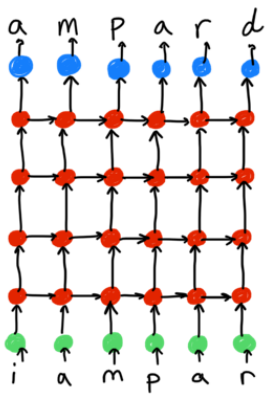
we are trying to add LTM_{t-1} to the formula for computing the forget factor.



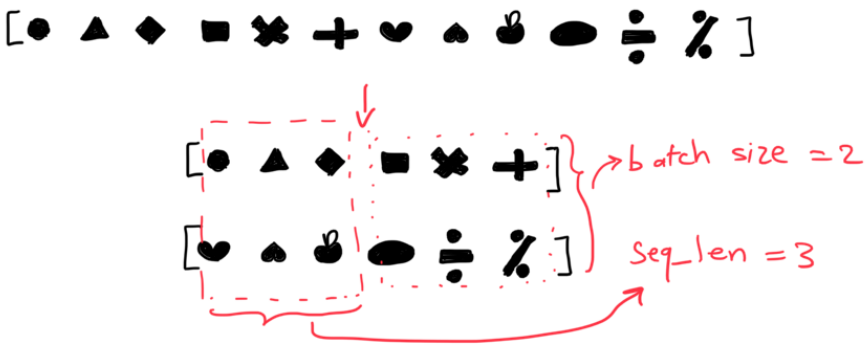
We can add these peephole connection everywher in the LSTM cell!

An Exercise in RNN — (PyTorch)

* learning about a text one character at a time and generating new text one character at a time (character-wise RNN)

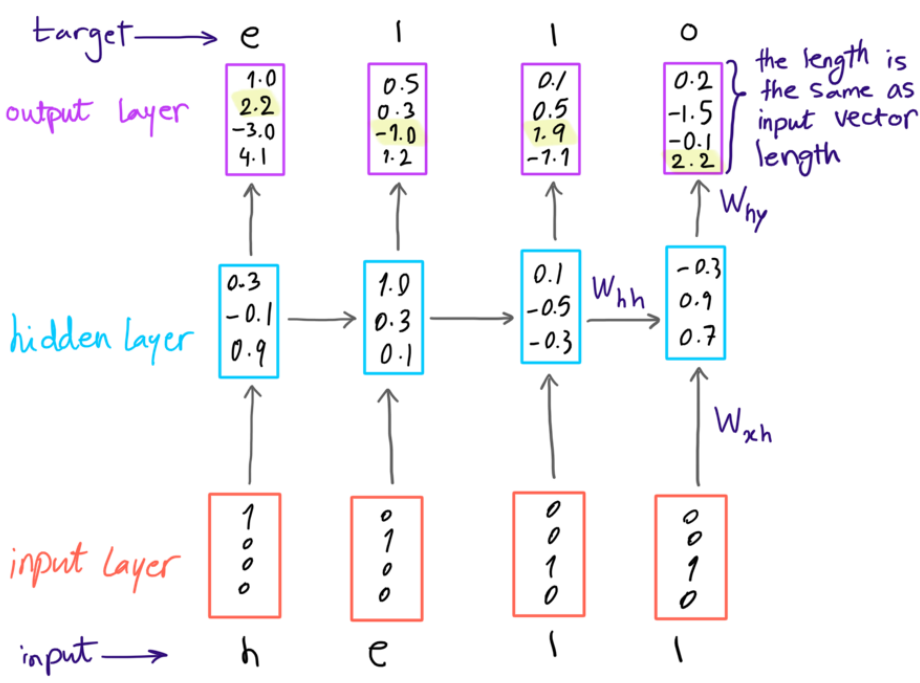


- We should 'batch' the sequences to better use matrix mult capabilities.



Now in PyTorch

What we want to implement:



Code

```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
```

Loading the data

```
with open('file.txt', 'r') as f:
    text = f.read()
```

Tokenization

```
chars = tuple(set(text))
int2char = dict(enumerate(chars))
char2int = {ch: ii for ii, ch in int2char.items()}
encoded = np.array([char2int[ch] for ch in text])
```

Pre-processing

```
def one-hot (arr, n_labels):
    oh = np.zeros ((np.multiply(*arr.shape), n_labels),
                                        dtype = np.float32)
    oh[np.arange(oh.shape[0]), arr.flatten()] = 1.0
    oh = oh.reshape ((*arr.shape, n_labels))
    return oh
```

Making Batches

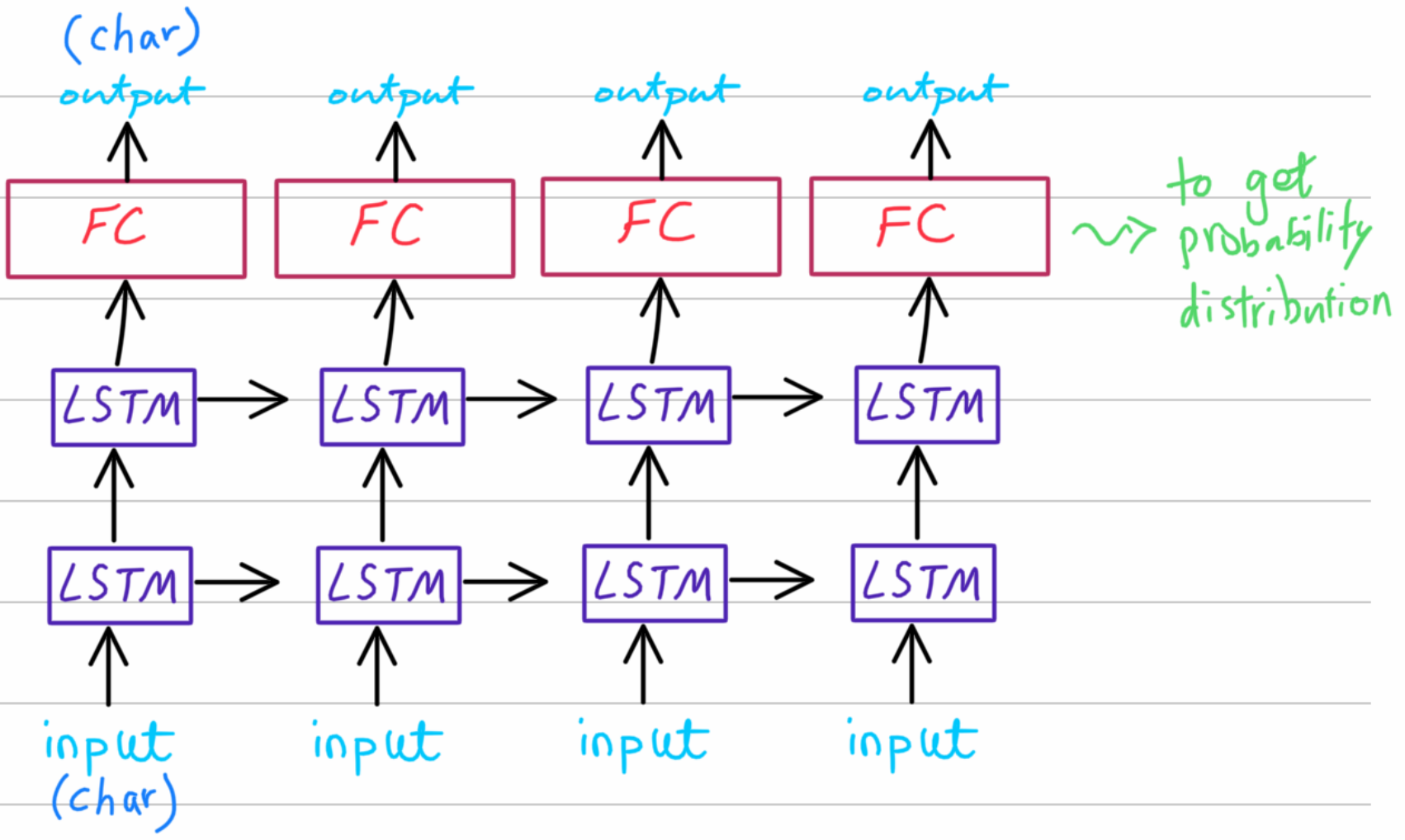
```
def get_batches (arr, batch_size, seq-length):
    n_batches = len(arr) // (batch_size * seq_len)
    arr = arr[: n_batches * batch_size * seq_len]
    arr = arr.reshape ((batch_size, -1))

    for n in range(0, arr.shape[1], seq_length):
        x = arr[:, n:n+seq_length]
        y = np.zeros_like(x)
```

all rows

Some Columns

```
        try:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, n+seq_length]
        except IndexError:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, 0]
        yield x, y
```

The model




```
class CharRNN(nn.Module):

    def __init__(self, tokens, n_hidden=256, n_layers=2,
                 drop_prob=0.5, lr=0.001):

        super().__init__()
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

        # Making the Dictionary
        self.chars = tokens
        self.int2char = dict(enumerate(self.chars))
        self.char2int = {ch: ii for ii, ch in self.int2char.items()}
```

LSTM Layer in PyTorch

```
nn.LSTM(input_size, n_hidden, n_layers,
        dropout=drop_prob, batch_first=True)
```

layers

```
self.lstm = nn.LSTM(len(self.chars), n_hidden,
                    n_layers, dropout=drop_prob,
                    batch_first=True)
```

dropout between LSTM layers

```
self.dropout = nn.Dropout(drop_prob)
self.fc = nn.Linear(n_hidden, len(self.chars))
```

```
def forward(self, x, hidden):
    r_output, hidden = self.lstm(x, hidden)
    out = self.dropout(r_output)
    out = out.view(-1, self.n_hidden)
```

stacking LSTM hidden outputs

```
    out = self.fc(out)
    return out, hidden
```

```
def init_hidden(self, batch_size):
    w = next(self.parameters()).data
    if train_on_gpu:
        hidden = (w.new(self.n_layers, batch_size, self.n_hidden).
                  zero_().cuda(),
                  w.new(self.n_layers, batch_size, self.n_hidden).
                  zero_().cuda())
    else:
        hidden = (w.new(self.n_layers, batch_size, self.n_hidden).
                  zero_(),
                  w.new(self.n_layers, batch_size, self.n_hidden).
                  zero_())
    return hidden
```

Training

```
def train(model, data, n_epochs=2, batch_size=32,
          seq_length=128, lr=0.001, clip=5,
          val_frac=0.1):
```

clip → gradient clipping

val_frac → fraction of holdout data for validation

```
    model.train()
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
```

```
val_idx = int(len(data)*(1-val_frac))
data, val_data = data[:val_idx], data[val_idx:]

if train_on_gpu:
    model.cuda()

Counter = 0
n_chars = len(model.chars)
```

```
for e in range(epochs):
    h = model.init_hidden(batch_size)

    for x,y in get_batches(data, batch_size, seq_length):
        Counter += 1
        x = one_hot(x, n_chars)
        input, targets = torch.from_numpy(x),
                         torch.from_numpy(y)

        if train_on_gpu:
            inputs, targets = inputs.cuda(), targets.cuda()

        h = tuple([each.data for each in h])
```

Create new vars for hidden state to avoid backprop on the entire history

```
        model.zero_grad()
        output, h = model(inputs, h)
        loss = Criterion(output, targets.view(batch_size *
                                              seq_length))
        loss.backward()
```

prevent gradient exploding →

```
        nn.utils.clip_grad_norm_(model.parameters, clip)

        opt.step()
```

```
if counter % print_every == 0:
    val_h = model.init_hidden(batch_size)
    losses = []
    model.eval()
    for x,y in get_batches(val_data, batch_size,
                           seq_length):
        x = one_hot(x, n_chars)
        x,y = torch.from_numpy(x),
              torch.from_numpy(y)

        val_h = tuple([each.data for each in val_h])
        inputs, targets = x,y
        if train_on_gpu:
            inputs, targets = inputs.cuda(), targets.cuda()
        out, val_h = model(inputs, val_h)

        loss = criterion(out, targets.view(batch_size *
                                           seq_length))
        losses.append(loss.item())

    model.train()
```

Let's train now!

```
n_hidden = 512
n_layers = 2
model = CharRNN(chars, n_hidden, n_layers)

batch_size = 128
seq_length = 100
epochs = 2

train(model, encoded, epochs=epochs, batch_size=batch_size,
      seq_length=seq_length, lr=0.001)
```

Save the Model

```
model_name = f'model_{epoch}_{n_hidden}_{n_layers}'
checkpoint = {'n_hidden': model.n_hidden,
              'n_layers': model.n_layers,
              'state_dict': model.state_dict(),
              'tokens': model.chars}
with open(model_name, 'wb') as f:
    torch.save(checkpoint, f)
```

Use the trained Model

```
def predict(model, char, h=None, top_k=None):
    x = np.array([[net.char2int[char]]])
    x = one_hot(x, len(net.chars))
    inputs = torch.from_numpy(x)

    if train_on_gpu:
        inputs = inputs.cuda()

    h = tuple([each.data for each in h])
    out, h = model(inputs, h)

    p = F.softmax(out, dim=1).data
    if train_out_gpu:
        p = p.cpu()

    if top_k is None:
        top_ch = np.arange(len(model.chars))
    else:
        p, top_ch = p.topk(top_k)
        top_ch = top_ch.numpy().squeeze()
```

make tensor

detach hidden state

from history

only consider if the predicted int is in dict.

select the next likely character

```
p = p.numpy().squeeze()
char = np.random.choice(top_ch, p=p/p.sum())

return net.int2char[char], h
```

How to use prediction to do text generation?

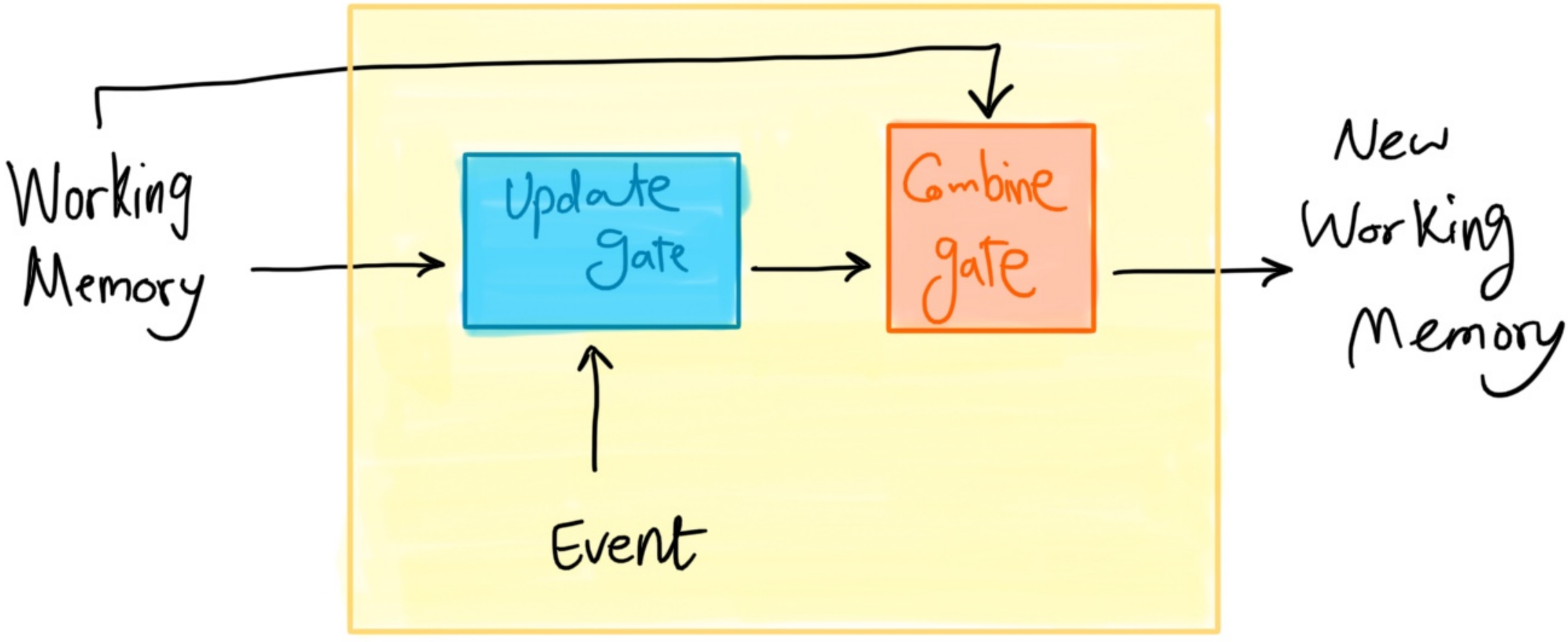
```
def sample(model, size, prime='The', top_k=None):
    if train_on_gpu:
        model.cuda()
    else:
        model.cpu()

    chars = [ch for ch in prime]
    h = model.init_hidden(1)
    for ch in prime:
        char, h = predict(model, ch, h, top_k=top_k)
    chars.append(char)

    for ii in range(size):
        char, h = predict(model, chars[-1], h, top_k=top_k)
        chars.append(char)

    return ''.join(chars)
```

how to start

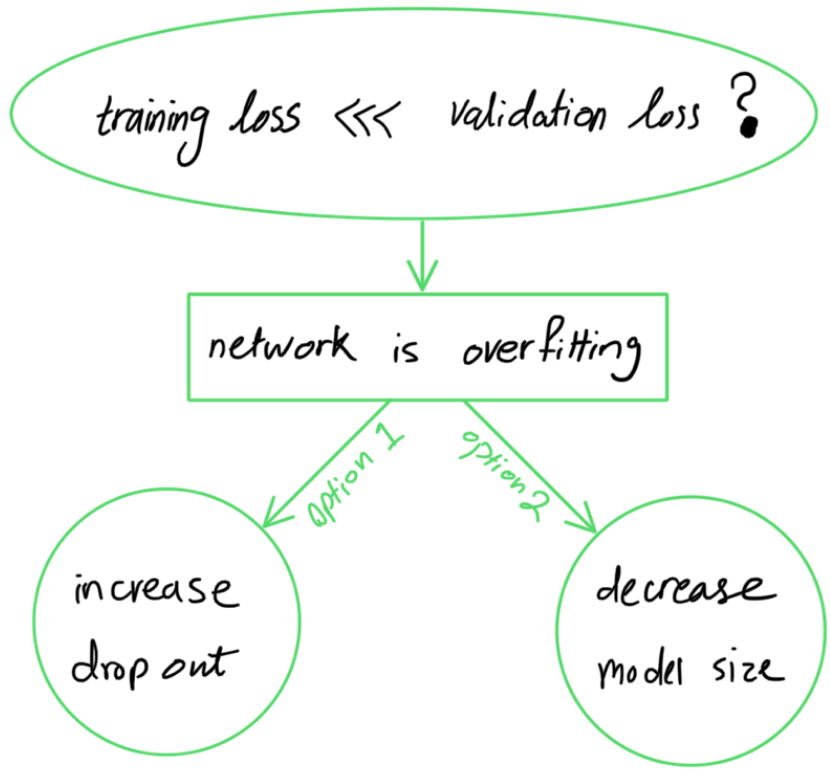
pass the previous character and get a new one.

Sample (loaded, 2000, top-k=5, prime="And Uven said")

Gated Recurrent Unit (GRU)



Selecting Reasonable Hyperparameters



- the number of model parameters should be about same magnitude as the size of dataset.

100 MB dataset (~ 100 million chars)