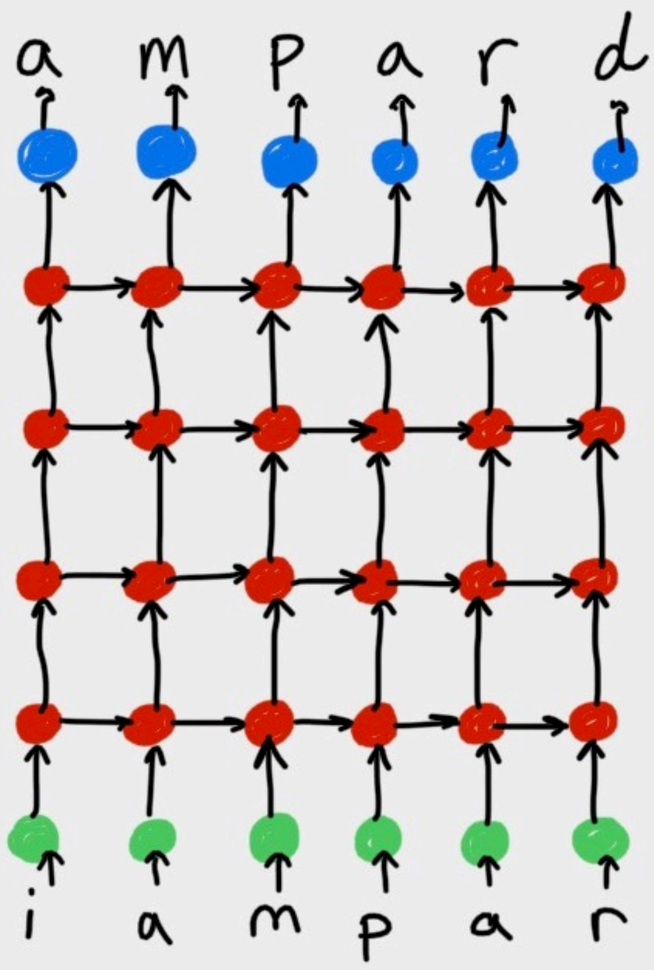
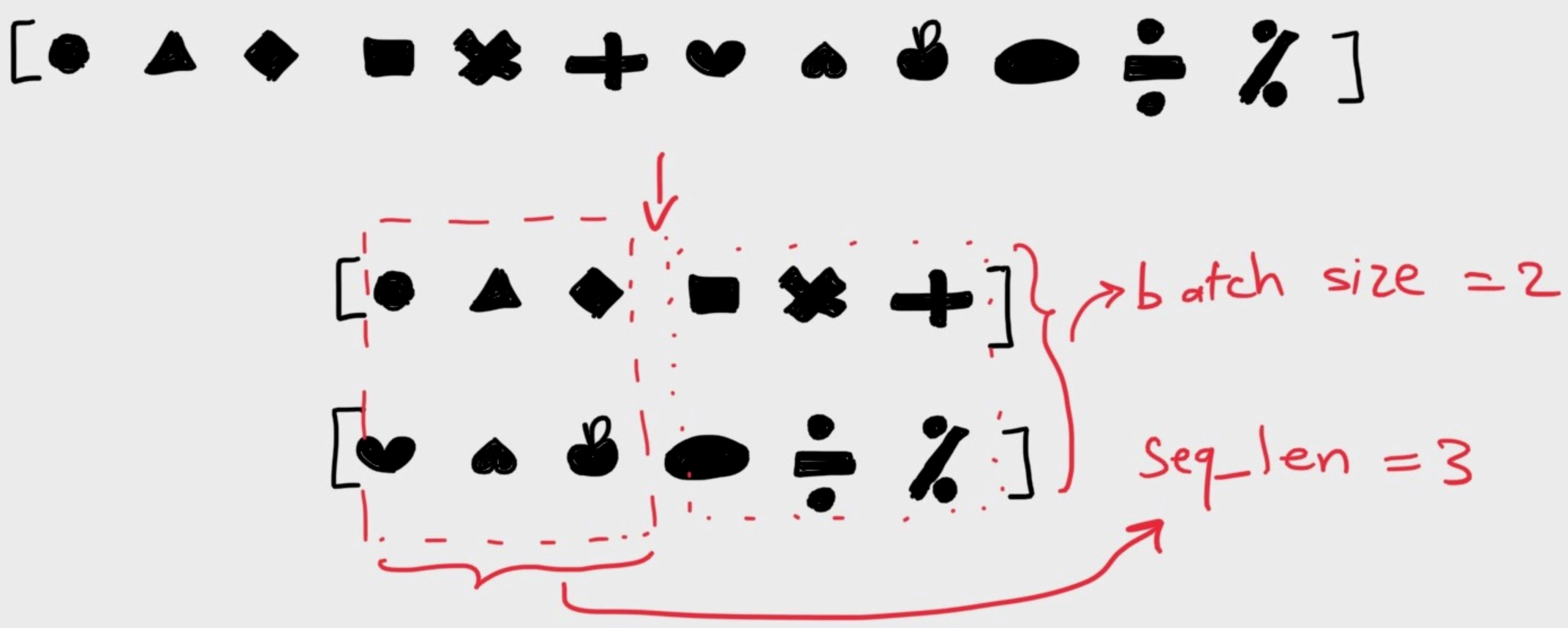
An Exercise in LSTM — (PyTorch)

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

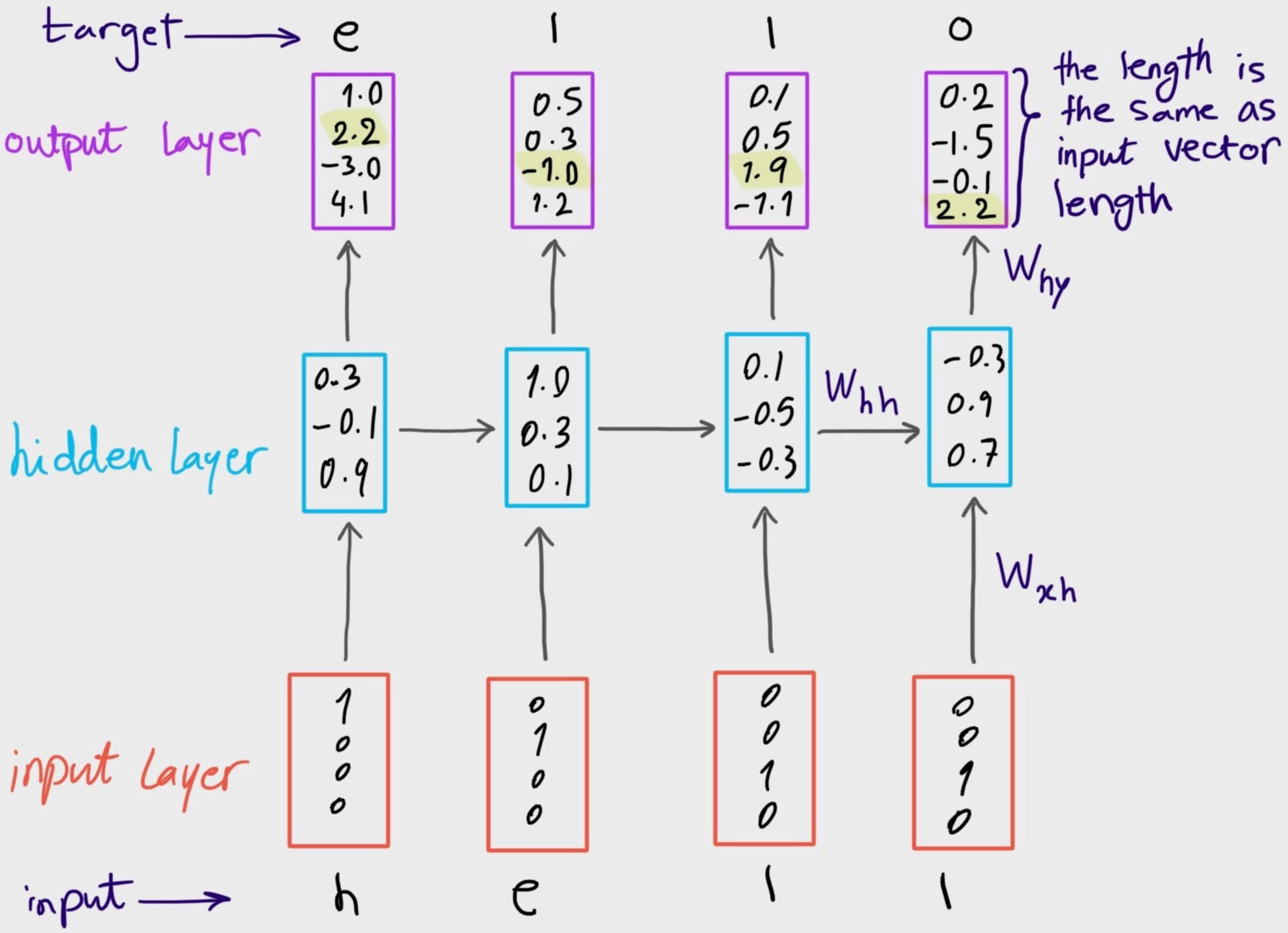


- We should 'batch' the sequences to better use matrix mult capabilities.



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
```

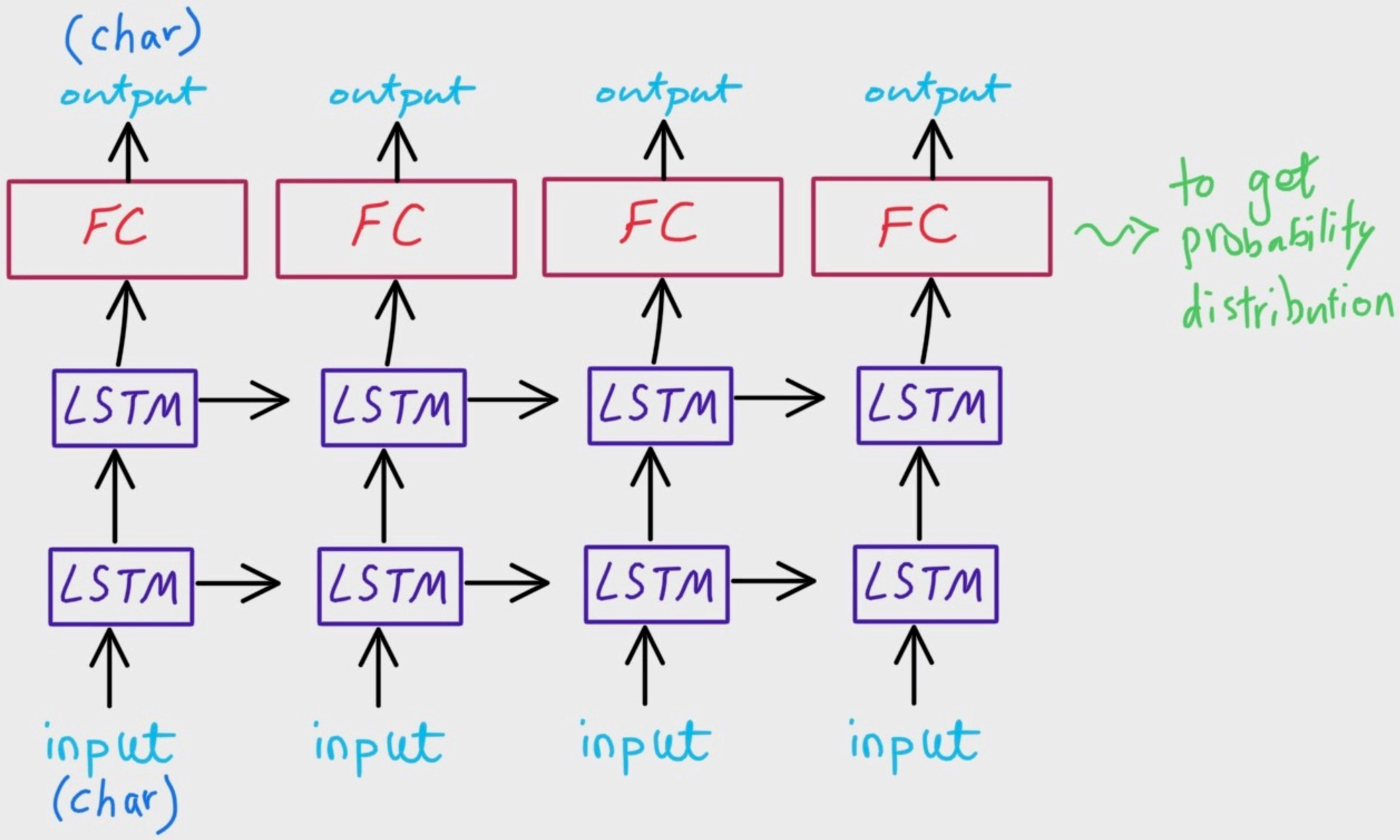
Making Batches

```
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)

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

Some Columns

all rows

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):
```

make tensor:

```
    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()
```

detach hidden state from history:

```
    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()
```

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)
```

prime: how to start

pass the previous character and get a new one.

```
Sample(loaded, 2000, top_k=5, prime="And Lven said")
```