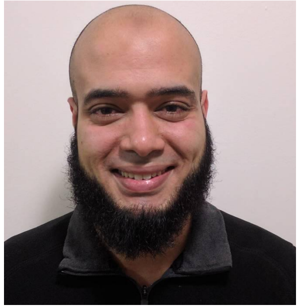
# Machine Learning

# Char Generation

**Mostafa S. Ibrahim**
*Teaching, Training and Coaching for more than a decade!*

*Artificial Intelligence & Computer Vision Researcher*
*PhD* from Simon Fraser University - Canada
*Bachelor / MSc* from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



# Next Character Generation

- As a simple demonstration for RNNs (in PyTorch), let’s do a simple next character generation
- Assume our data consists of some statements, e.g. Shakespeare
- We would like our model to understand it and be able to **generate** it or generate **similar** statements (but this requires a lot of data / good model)
- For simplicity, assume we have 3 statements
  - **Get Skilled in Machine Learning**
  - **By CS-Get Skilled Academy**
  - **Instructor Mostafa Saad Ibrahim**
- After the training, we would like to give the model a prefix and then it **generates** a possible statement. Example:
  - **Prefix**: Instructor **Generation**: Mostafa Saad Ibrahim

# RNN Logic

- Let's create a simple network with a single RNN layer
- Given a sentence, it is a sequence of characters
- We will train the network such that, **given a prefix** of the sequence, it predicts the next character
- Example: assume the sequence is the word: mostafa#saad
  - Input m output o [now append o to m]
  - Input mo output s [now append s to mo]
  - Input mos output t [now append t to mos]
  - Input most output a [now append a to most]
  - Input mosta output f [now append f to mosta]
  - Input mostaf output a [now append a to mostaf]
  - Input mostafa output # [now append # to mostafa]
  - Input mostafa# output s [now append s to mostafa#] and so on

```python
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.rnn = nn.RNN(input_size, hidden_size, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    # Forward pass
    def forward(self, x, hidden):
        #out, hidden = self.rnn(x)           # default
        out, hidden = self.rnn(x, hidden)   # allows u resume from a state
        out = self.fc(out)
        return out, hidden

    def nohistory_hidden_state(self, batch_size):
        # return tensor of zeros as a begin
        return torch.zeros(self.n_layers, batch_size, self.hidden_size)
```

# Prepare the data

- Assume all unique characters are 20 (let's call vocabulary size)
- We will represent each character in 2 ways
  - An index in the given text (below in all_sequences_data)
  - A one-hot-encoding: this will be the vector to feed to the RNN!
- Assume our data is xab
  - We build 2 maps
  - Character to integer
    - x=0, a=1, b=2
  - Integer to character
    - 0=x, 1=a, 2=b

```
# lets use some letter as EOF like $
sequences = [
    "Get Skilled in Machine Learning$$$$",
    "By CS-Get Skilled Academy$$$$",
    "Instructor Mostafa Saad Ibrahim$$$$"
]
all_sequences_data = ''.join(sequences)
chars = tuple(set(all_sequences_data))
vocab_size = len(chars)

# Character to index and index to character mappings
char2int = {ch: ii for ii, ch in enumerate(chars)}
int2char = {ii: ch for ii, ch in enumerate(chars)}
```

# Network Preparation

- Nothing especial. Let's use one RNN (n_layers=1)
- For simplicity, keep the batch size = 1

```
# Prepare the model and optimizer
hidden_size = 128
n_layers = 1
batch_size = 1
n_epochs = 100
learning_rate = 0.01

model = CharRNN(vocab_size, hidden_size, vocab_size, n_layers)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()
```

# Preparing the data

- For simplicity, assume our input string is 01256347
    - Assume, we also append letter $ to indicate EOS (end of sequence)
- We will feed the sequence letter by letter
    - The ground truth is its next letter
    - (0, 1), (1, 2), (2, 5), (5, 6), (6, 3), (3, 4), (4, 7), (7, $)
- To feed a letter, we will use one-hot-encoding over the vocabulary size
    - Assume we have 9 letters (01256347 and the #)

# Preparing the data: 01256347$

- For sequence: (0, 1), (1, 2), (2, 5), (5, 6), (6, 3), (3, 4), (4, 7), (7, $)
  - We build one hot encoding
  - For example for (4, 7)
  - Assume 4 has index 4
  - Then 4 is: 000010000
- We can feed the whole sequence to the RNN module in a single call
  - It will keep updating the hidden sequence
- Or, you can divide into blocks/batches
  - But keep passing the last hidden state to continue processing

| Pair | Vector X | Target Y |
|---|---|---|
| 0 ⇒ 1 | 100000000 | 1 |
| 1 ⇒ 2 | 010000000 | 2 |
| 2 ⇒ 5 | 001000000 | 5 |
| 5 ⇒ 6 | 000001000 | 6 |
| 6 ⇒ 3 | 000000100 | 3 |
| 3 ⇒ 4 | 000100000 | 4 |
| 4 ⇒ 7 | 000010000 | 7 |
| 7 ⇒ $ | 000000010 | 8 (for $) |

# Training: Epochs and Examples

- Iterate on every sequence. For each sequence prepare the zero hidden state (no previous history)
- Let the cu

```
sequences = [
    "Get Skilled in Machine Learning$$$$",
    "By CS-Get Skilled Academy$$$$",
    "Instructor Mostafa Saad Ibrahim$$$$"
]
```

```
# Training the model
model.train()
for epoch in range(n_epochs):
    for data in sequences:
        # the whole document is a single sequence (one hidden state)
        hidden = model.nohistory_hidden_state(batch_size)
```

# Training: Batches

- For very long sequences, we better break into batches
  - However, you must keep using the updating same hidden state
  - Below, I use a single batch (whole string), as the sentence is short
- The code just breaks the sequence into batches of seq_length, no magic
  - If the last batch < seq_length, it is dropped

```
hidden = model.nohistory_hidden_state(batch_size)
# Below, I feed the whole sequence as a single batch (better performance for a few examples)
# We also can divide sequences to subsequences (but all on SAME hidden state): e.g. seq_length = 10
seq_length = len(data)  # explore values like 10 and 20
for batch in range(0, len(data) - seq_length + 1, seq_length):
    X = torch.zeros(batch_size, seq_length, vocab_size)        # 1x35x30  (35 seq len, 30 voc size)
    y = torch.zeros(batch_size, seq_length, dtype=torch.long)   # 1x30
    s1, s2 = '', ''
    for i in range(seq_length):
        s1, s2 = data[batch + i], data[batch + i + 1 if batch + i + 1 < len(data) else 0]
        X[0, i, char2int[s1]] = 1
        y[0, i] = char2int[s2]
```

# Training: Optimizer

- Normal optimizer and loss steps

```
optimizer.zero_grad()
output, hidden = model(X, hidden)
loss = criterion(output.squeeze(0), y.squeeze(0))
loss.backward()
optimizer.step()
hidden.detach_()    # to avoid pytorch error for seq_length < len(data)

answers = torch.max(output.squeeze(0), dim=1)[1]
train_acc = torch.sum(answers == y.squeeze(0)) / y.squeeze(0).size()[0]
```

# Inference

- Give a prefix and see generation

```python
# Generate some text that starts with this prefix
print(generate(model, size=50, prefix_str='Get Skilled'))
print(generate(model, size=50, prefix_str='By'))
print(generate(model, size=50, prefix_str='Instructor'))
print(generate(model, size=50, prefix_str='lls'))
```

if __name__ == '__main__' › for epoch in range(n_epochs) › for data in sequences › for batch in

simple_rnn1_char_gen_pytorch (1)

```
Epoch 98, Loss: 0.005 - Accuracy: 1.00
Epoch 98, Loss: 0.010 - Accuracy: 1.00
Epoch 99, Loss: 0.001 - Accuracy: 1.00
Epoch 99, Loss: 0.007 - Accuracy: 1.00
Epoch 99, Loss: 0.005 - Accuracy: 1.00
Epoch 99, Loss: 0.009 - Accuracy: 1.00
Epoch 100, Loss: 0.001 - Accuracy: 1.00
Epoch 100, Loss: 0.007 - Accuracy: 1.00
Epoch 100, Loss: 0.005 - Accuracy: 1.00
Epoch 100, Loss: 0.009 - Accuracy: 1.00
 in Machine Learning$
 CS-Get Skilled Academy$
 Mostafa Saad Ibrahim$
 LSkilled Academy$
```

# Generation: Build Hidden

- First, we take the prefix, feed it into the network and get the last hidden state
- The output of the last step can generate the FIRST character

```
def generate(model, prefix_str, size, eof='$'):
    model.eval()
    chars = []

    # Build the initial hidden from the given prefix
    hidden = model.nohistory_hidden_state(1)
    for char in prefix_str:
        # Build a one-hot-encoding for the current char
        char_tensor = torch.zeros(1, 1, vocab_size)
        char_tensor[0, 0, char2int[char]] = 1
        out, hidden = model(char_tensor, hidden)
        #print(char, int2char[out.argmax().item()])

    # Now hidden represents all input letters and its out can predict a letter
```

# Generation: Extract the next character

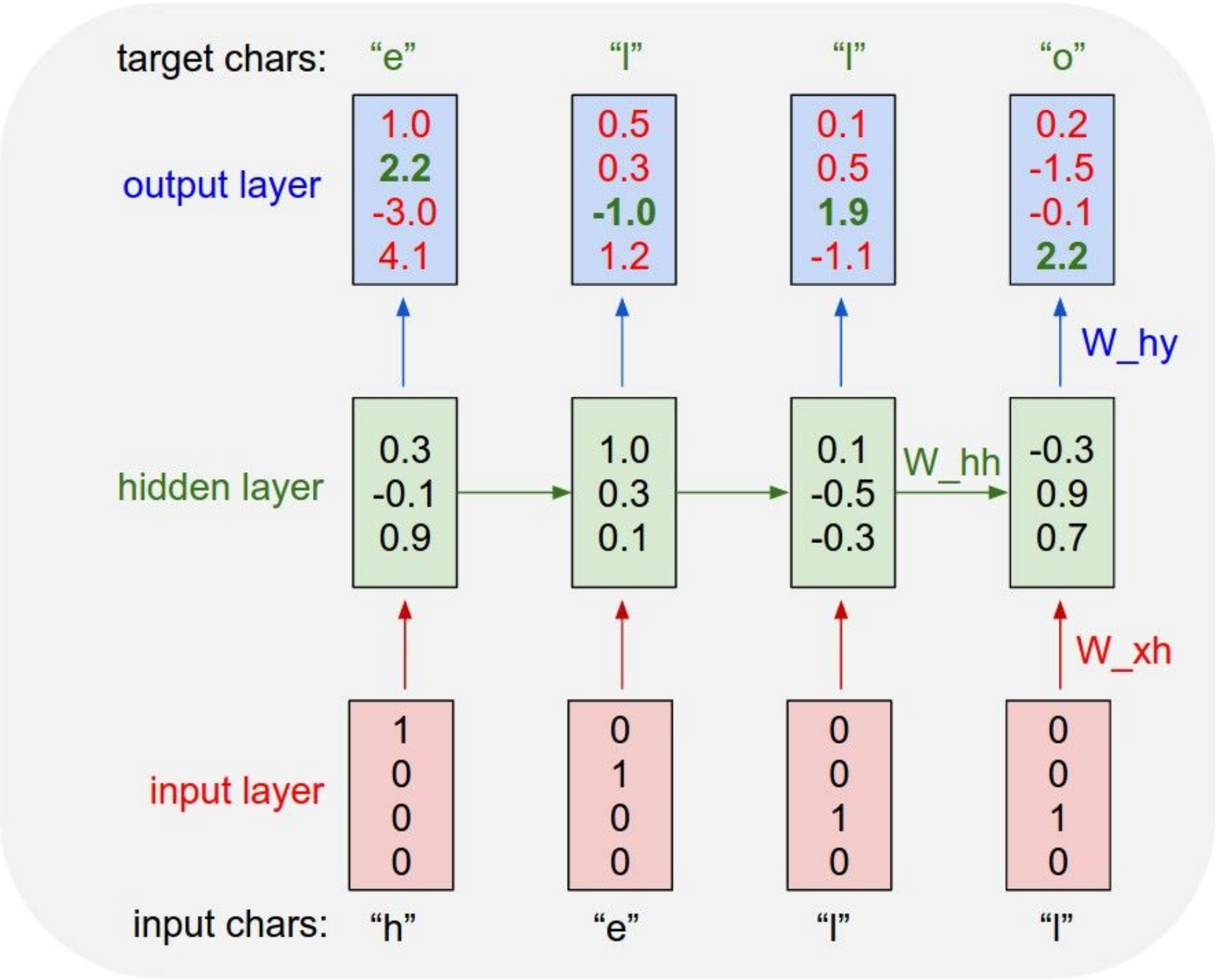
- Given the logits (out), we can convert to probabilities with softmax
- You expect to just get the argmax? But this is a definite answer
- To generate, we use this distribution to generate randomly according to this distribution. This what ChatGPT can use to get more new text!
    - [0.15, 0.8, 0.5] ⇒ with 80% chance, will sample idx 1

```
def generate_letter():
    # given logits, compute the probabilities and sample a letter
    p = torch.nn.functional.softmax(out[0, 0], dim=0).detach().numpy()

    # select a random index (generation) based on the distribution (weights)
    # This allows us to generate several possible answers, like chatgpt
    char_idx = np.random.choice(vocab_size, p=p)
    #char_idx = out.argmax().item() # this just select a single most probable answe

    char = int2char[char_idx]
    return char
```

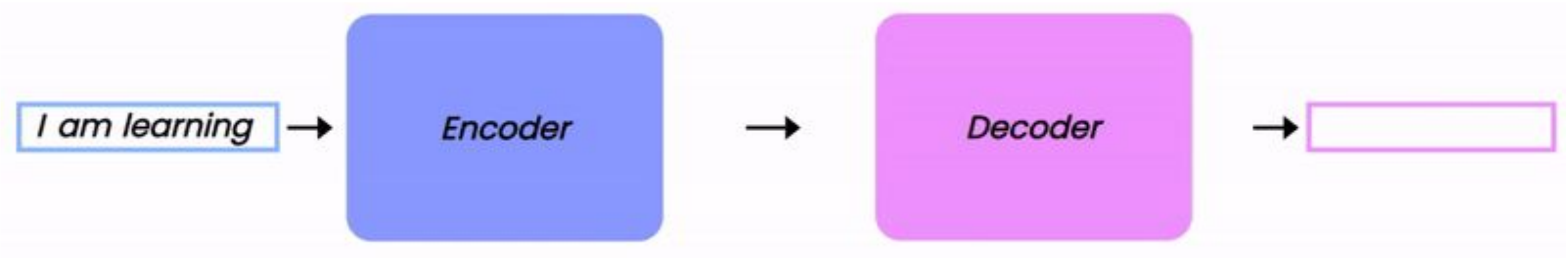
# Generation of new data

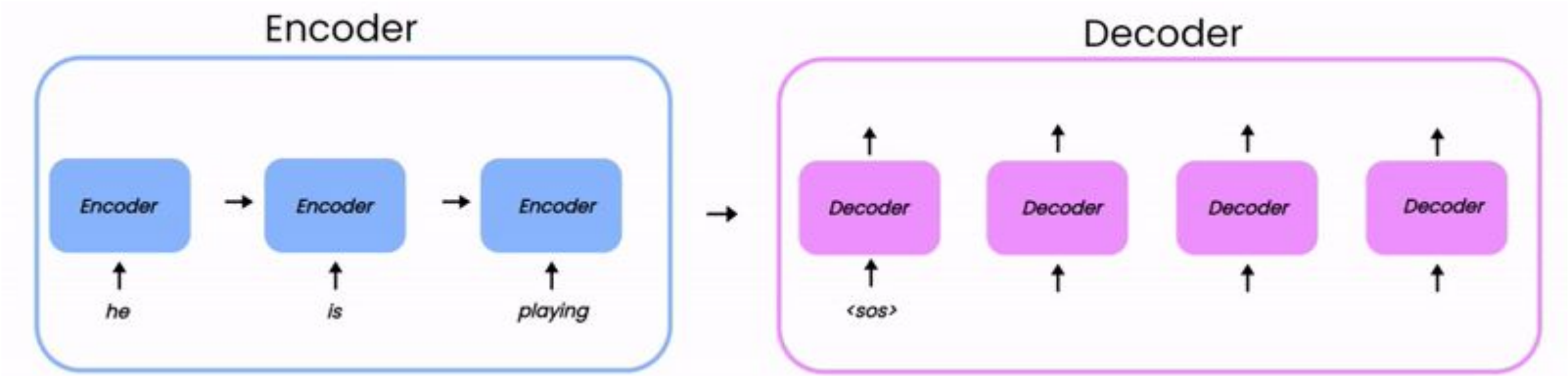
- Now, we will run for the given size steps and keep generating letters
- Whatever we generate we **feed back again** to the network

```
for _ in range(size):
    # use the last out logits to generate a new letter
    char = generate_letter()
    chars.append(char)

    if char == eof:
        break

    char_tensor = torch.zeros(1, 1, vocab_size)
    char_tensor[0, 0, char2int[char]] = 1
    # Predict the next letter given the current one and its history
    out, hidden = model(char_tensor, hidden)

return ''.join(chars)
```


target chars:
"e"
"l"
"l"
"o"
output layer
1.0
2.2
-3.0
4.1
0.5
0.3
-1.0
1.2
0.1
0.5
1.9
-1.1
0.2
-1.5
-0.1
2.2
W_hy
hidden layer
0.3
-0.1
0.9
1.0
0.3
0.1
0.1
-0.5
-0.3
W_hh
-0.3
0.9
0.7
W_xh
input layer
1
0
0
0
0
1
0
0
0
0
1
0
0
0
1
0
input chars:
"h"
"e"
"l"
"l"

Img [src]()




- The encoder output represents about the last hidden step. We call it **context vector**.
- The decoder use the context to parse the whole sequence out
- This is example of **sequence-to-sequence**. Being sequential make it hard to parallelize!

## Machine Translation

# Relevant Materials

- The Unreasonable Effectiveness of Recurrent Neural Networks
- Neural Machine Translation Using Sequence to Sequence Model

*"Acquire knowledge and impart it to the people."*

*"Seek knowledge from the Cradle to the Grave."*