BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

NLP LAB - 7

[HIDDEN MARKOV MODEL (HMM) & CONDITIONAL RANDOM FIELD (CRF)]

DATE: 03/11/2023 – 05/11/2023
TIME: 02 Hours

References:

1. https://compneuro.neuromatch.io/tutorials/W3D2_HiddenDynamics/student/W3D2_Tutorial2.html
2. https://towardsdatascience.com/conditional-random-field-tutorial-in-pytorch-ca0d04499463

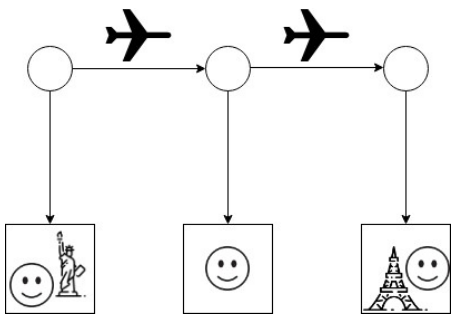
Hidden Markov Model (HMM)

This notebook introduces the Hidden Markov Model (HMM), a simple model for sequential data.
We will see:

- what an HMM is and when you might want to use it;
- the so-called "three problems" of an HMM; and
- how to implement an HMM in PyTorch.

A hypothetical scenario

To motivate the use of HMMs, imagine that you have a friend who gets to do a lot of travelling. Every day, this jet-setting friend sends you a selfie from the city they're in, to make you envious.

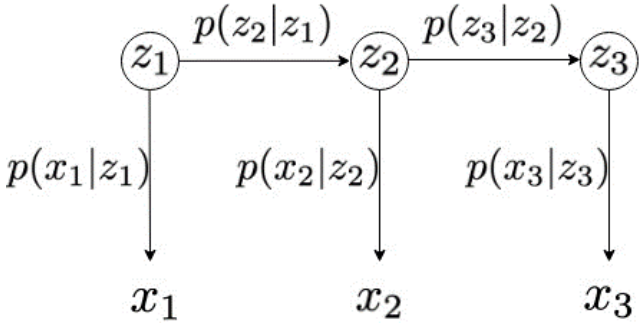


How would you go about guessing which city the friend is in each day, just by looking at the selfies?
If the selfie contains a really obvious landmark, like the Eiffel Tower, it will be easy to figure out where the photo was taken. If not, it will be a lot harder to infer the city.
But we have a clue to help us: the city the friend is in each day is not totally random. For example, the friend will probably remain in the same city for a few days to sightsee before flying to a new city.

The HMM Setup

The hypothetical scenario of the friend travelling between cities and sending you selfies can be modeled using an HMM.
An HMM models a system that is in a particular state at any given time and produces an output that depends on that state.
At each timestep or clock tick, the system randomly decides on a new state and jumps into that state. The system then randomly generates an observation. The states are "hidden": we can't observe them. (In the cities/selfies analogy, the unknown cities would be the hidden states, and the selfies would be the observations.)

Let's denote the sequence of states as $z = z_1, z_2, \ldots, z_T$, where each state is one of a finite set of N states, and the sequence of observations as $x = x_1, x_2, \ldots, x_T$. The observations could be discrete, like letters, or real-valued, like audio frames.



An HMM makes two key assumptions:

- **Assumption 1:** The state at time t depends *only* on the state at the previous time t−1.
- **Assumption 2:** The output at time t depends *only* on the state at time t.

These two assumptions make it possible to efficiently compute certain quantities that we may be interested in.

Components of HMM

An HMM has three sets of trainable parameters.

- The **transition model** is a square matrix A, where $A_{s,s'}$ represents $p(z_t = s|z_{t-1} = s')$, the probability of jumping from state s' to state s.
- The **emission model** $b_s(x_t)$ tells us $p(x_t|z_t = s)$, the probability of generating x_t when the system is in state s. For discrete observations, which we will use in this notebook, the emission model is just a lookup table, with one row for each state, and one column for each observation. For real-valued observations, it is common to use a Gaussian mixture model or neural network to implement the emission model.

- The **state priors** tell us $p(z_1 = s)$, the probability of starting in state s. We use π to denote the vector of state priors, so π_s is the state prior for state s.

Let's program an HMM class in PyTorch.

```
import torch
import numpy as np

class HMM(torch.nn.Module):
  """
  Hidden Markov Model with discrete observations.
  """
  def __init__(self, M, N):
    super(HMM, self).__init__()
    self.M = M # number of possible observations
    self.N = N # number of states

    # A
    self.transition_model = ... # TODO

    # b(x_t)
    self.emission_model = ... # TODO

    # pi
    self.unnormalized_state_priors = ... # TODO

    # use the GPU
    self.is_cuda = torch.cuda.is_available()
    if self.is_cuda: self.cuda()

class TransitionModel(torch.nn.Module):
  def __init__(self, N):
    ... # TODO
    self.unnormalized_transition_matrix = ... # TODO

class EmissionModel(torch.nn.Module):
  def __init__(self, N, M):
    ... # TODO
    self.unnormalized_emission_matrix = ... # TODO
```

To sample from the HMM, we start by picking a random initial state from the state prior distribution.
Then, we sample an output from the emission distribution, sample a transition from the transition distribution, and repeat.
(Notice that we pass the unnormalized model parameters through a softmax function to make them into probabilities.)

```
def sample(self, T=10):
  state_priors =
torch.nn.functional.softmax(self.unnormalized_state_priors, dim=0)
  transition_matrix =
torch.nn.functional.softmax(self.transition_model.unnormalized_transiti
on_matrix, dim=0)
  emission_matrix =
torch.nn.functional.softmax(self.emission_model.unnormalized_emission_m
atrix, dim=1)

  # sample initial state
  z_t =
torch.distributions.categorical.Categorical(state_priors).sample().item
()
  z = []; x = []
  z.append(z_t)
  for t in range(0,T):
    # sample emission
    x_t =
torch.distributions.categorical.Categorical(...).sample().item() # TODO
    x.append(x_t)

    # sample transition
    z_t =
torch.distributions.categorical.Categorical(...).sample().item() # TODO
    if t < T-1: z.append(z_t)

  return x, z

# Add the sampling method to our HMM class
HMM.sample = sample
```

Let's try hard-coding an HMM for generating fake words. (We'll also add some helper functions for encoding and decoding strings.)

We will assume that the system has one state for generating vowels and one state for generating consonants, and the transition matrix has $0s$ on the diagonal, in other words, the system cannot stay in the vowel state or the consonant state for one than one timestep; it has to switch.

Since we pass the transition matrix through a softmax, to get $0s$ we set the unnormalized parameter values to $-\infty$.

```
import string
alphabet = string.ascii_lowercase

def encode(s):
  """
  Convert a string into a list of integers
  """
  x = [alphabet.index(ss) for ss in s]
  return x

def decode(x):
  """
  Convert list of ints to string
  """
  s = "".join([alphabet[xx] for xx in x])
  return s

# Initialize the model
model = ... # TODO
```

```
# Hard-wiring the parameters!
# Let state 0 = consonant, state 1 = vowel
for p in model.parameters():
    p.requires_grad = False # needed to do lines below
model.unnormalized_state_priors[0] = 0.    # Let's start with a
consonant more frequently
model.unnormalized_state_priors[1] = -0.5
print("State priors:",
torch.nn.functional.softmax(model.unnormalized_state_priors, dim=0))

# In state 0, only allow consonants; in state 1, only allow vowels
vowel_indices = torch.tensor([alphabet.index(letter) for letter in
"aeiou"])
consonant_indices = torch.tensor([alphabet.index(letter) for letter in
"bcdfghjklmnpqrstvwxyz"])
model.emission_model.unnormalized_emission_matrix[0, vowel_indices] = -
np.inf
model.emission_model.unnormalized_emission_matrix[1, consonant_indices]
= -np.inf
print("Emission matrix:",
torch.nn.functional.softmax(model.emission_model.unnormalized_emission_
matrix, dim=1))

# Only allow vowel -> consonant and consonant -> vowel
model.transition_model.unnormalized_transition_matrix[0,0] = -np.inf  #
consonant -> consonant
model.transition_model.unnormalized_transition_matrix[0,1] = 0.       #
vowel -> consonant
model.transition_model.unnormalized_transition_matrix[1,0] = 0.       #
consonant -> vowel
model.transition_model.unnormalized_transition_matrix[1,1] = -np.inf  #
vowel -> vowel
print("Transition matrix:",
torch.nn.functional.softmax(model.transition_model.unnormalized_transit
ion_matrix, dim=0))
```

Try sampling from our hard-coded model:

```
# Sample some outputs
for _ in range(4):
  sampled_x, sampled_z = model.sample(T=5)
  print("x:", decode(sampled_x))
  print("z:", sampled_z)
```

The Three Problems

In a classic tutorial on HMMs, Lawrence Rabiner describes "three problems" that need to be solved before you can effectively use an HMM. They are:

- Problem 1: How do we efficiently compute $p(\mathbf{x})$?
- Problem 2: How do we find the most likely state sequence $\mathbf{z}$ that could have generated the data?
- Problem 3: How do we train the model?

In the rest of the notebook, we will see how to solve each problem and implement the solutions in PyTorch.

Problem-1: How do we compute $p(x)$

Why?

Why might we care about computing $p(\mathbf{x})$? Here's two reasons.

- Given two HMMs, θ_1 and θ_2, we can compute the likelihood of some data $\mathbf{x}$ under each model, $p_{\theta_1}(\mathbf{x})$ and $p_{\theta_2}(\mathbf{x})$, to decide which model is a better fit to the data. (For example, given an HMM for English speech and an HMM for French speech, we could compute the likelihood given each model, and pick the model with the higher likelihood to infer whether the person is speaking English or French.)
- Being able to compute $p(\mathbf{x})$ gives us a way to train the model, as we will see later.

How?

Given that we want $p(\mathbf{x})$, how do we compute it?

We've assumed that the data is generated by visiting some sequence of states $\mathbf{z}$ and picking an output x_t for each z_t from the emission distribution $p(x_t|z_t)$. So if we knew $\mathbf{z}$, then the probability of $\mathbf{x}$ could be computed as follows:

$$p(\mathbf{x}|\mathbf{z}) = \prod_t p(x_t|z_t)p(z_t|z_{t-1})$$

However, we don't know $\mathbf{z}$; it's hidden. But we do know the probability of any given $\mathbf{z}$, independent of what we observe. So we could get the probability of $\mathbf{x}$ by summing over the different possibilities for $\mathbf{z}$, like this:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z}) = \sum_{\mathbf{z}} \prod_t p(x_t|z_t)p(z_t|z_{t-1})$$

The problem is: if you try to take that sum directly, you will need to compute N^T terms. This is impossible to do for anything but very short sequences. For example, let's say the sequence is of length $T = 100$ and there are $N = 2$ possible states. Then we would need to check $N^T = 2^{100} \approx 10^{30}$ different possible state sequences.

We need a way to compute $p(\mathbf{x})$ that doesn't require us to explicitly calculate all N^T terms. For this, we use the forward algorithm.

The Forward Agorithm:

> for $s = 1 \to N$:
> $\alpha_{s,1} := b_s(x_1) \cdot \pi_s$
> for $t = 2 \to T$:
> for $s = 1 \to N$:
> $\alpha_{s,t} := b_s(x_t) \cdot \sum_{s'} A_{s,s'} \cdot \alpha_{s',t-1}$
> $p(\mathbf{x}) := \sum_s \alpha_{s,T}$
> return $p(\mathbf{x})$

The forward algorithm is much faster than enumerating all N^T possible state sequences: it requires only $O(N^2T)$ operations to run, since each step is mostly multiplying the vector of forward variables by the transition matrix. (And very often we can reduce that complexity even further, if the transition matrix is sparse.)

There is one practical problem with the forward algorithm as presented above: it is prone to underflow due to multiplying a long chain of small numbers, since probabilities are always between 0 and 1. Instead, let's do everything in the log domain. In the log domain, a multiplication becomes a sum, and a sum becomes a `logsumexp`.

The Forward Algorithm (Log Domain):

> for $s = 1 \to N$:
> $\log \alpha_{s,1} := \log b_s(x_1) + \log \pi_s$
> for $t = 2 \to T$:
> for $s = 1 \to N$:
> $\log \alpha_{s,t} := \log b_s(x_t) + \operatorname{logsumexp}_{s'} (\log A_{s,s'} + \log \alpha_{s',t-1})$
> $\log p(\mathbf{x}) := \operatorname{logsumexp}_s (\log \alpha_{s,T})$
> return $\log p(\mathbf{x})$

Now that we have a numerically stable version of the forward algorithm, let's implement it in PyTorch.

```
def HMM_forward(self, x, T):
  """
  x : IntTensor of shape (batch size, T_max)
  T : IntTensor of shape (batch size)

  Compute log p(x) for each example in the batch.
  T = length of each example
  """
  if self.is_cuda:
    x = x.cuda()
    T = T.cuda()

  batch_size = x.shape[0]; T_max = x.shape[1]
  log_state_priors = ... # TODO
  log_alpha = ... # TODO
  if self.is_cuda: log_alpha = log_alpha.cuda()

  log_alpha[:, 0, :] = self.emission_model(x[:,0]) + log_state_priors
  for t in range(1, T_max):
    log_alpha[:, t, :] = self.emission_model(x[:,t]) +
self.transition_model(log_alpha[:, t-1, :])

  # Select the sum for the final timestep (each x may have different
length).
  log_sums = ... # TODO # HINT: Use logexpsum
  log_probs = torch.gather(log_sums, 1, T.view(-1,1) - 1)
  return log_probs
```

```
def emission_model_forward(self, x_t):
  log_emission_matrix = ... # TODO
  out = log_emission_matrix[:, x_t].transpose(0,1)
  return out
```

```
def transition_model_forward(self, log_alpha):
  """
  log_alpha : Tensor of shape (batch size, N)
  Multiply previous timestep's alphas by transition matrix (in log
domain)
  """
  log_transition_matrix = ... # TODO

  # Matrix multiplication in the log domain
  out = log_domain_matmul(log_transition_matrix,
log_alpha.transpose(0,1)).transpose(0,1)
  return out
```

```
def log_domain_matmul(log_A, log_B):
  """
  log_A : m x n
  log_B : n x p
  output : m x p matrix

  Normally, a matrix multiplication
  computes out_{i,j} = sum_k A_{i,k} x B_{k,j}

  A log domain matrix multiplication
  computes out_{i,j} = logsumexp_k log_A_{i,k} + log_B_{k,j}
  """
  m = log_A.shape[0]
  n = log_A.shape[1]
  p = log_B.shape[1]

  # log_A_expanded = torch.stack([log_A] * p, dim=2)
  # log_B_expanded = torch.stack([log_B] * m, dim=0)
    # fix for PyTorch > 1.5 by egaznep on Github:
  log_A_expanded = torch.reshape(log_A, (...)) # TODO
  log_B_expanded = torch.reshape(log_B, (...)) # TODO

  elementwise_sum = ... # TODO
  out = ... # TODO

  return out

TransitionModel.forward = transition_model_forward
EmissionModel.forward = emission_model_forward
HMM.forward = HMM_forward
```

Try running the forward algorithm on our vowels/consonants model from before:

```
x = torch.stack( [torch.tensor(encode("cat"))] )
T = torch.tensor([3])
print(model.forward(x, T))

x = torch.stack( [torch.tensor(encode("aba")),
torch.tensor(encode("abb"))] )
T = torch.tensor([3,3])
print(model.forward(x, T))
```

When using the vowel ↔ consonant HMM from above, notice that the forward algorithm returns $-\infty$ for $\mathbf{x} = abb$. That's because our transition matrix says the probability of vowel → vowel and consonant → consonant is 0, so the probability of abb happening is 0, and thus the log probability is $-\infty$.

Side Note: Deriving the forward algorithm:

If you're interested in understanding how the forward algorithm actually computes $p(\mathbf{x})$, read this section; if not, skip to the next part on "Problem 2" (finding the most likely state sequence).

To derive the forward algorithm, start by deriving the forward variable:

$$
\begin{aligned}
\alpha_{s,t} &= p(x_1, x_2, \ldots, x_t, z_t = s) \\
&= p(x_t | x_1, x_2, \ldots, x_{t-1}, z_t = s) \cdot p(x_1, x_2, \ldots, x_{t-1}, z_t = s) \\
&= p(x_t | z_t = s) \cdot p(x_1, x_2, \ldots, x_{t-1}, z_t = s) \\
&= p(x_t | z_t = s) \cdot \left(\sum_{s'} p(x_1, x_2, \ldots, x_{t-1}, z_{t-1} = s', z_t = s) \right) \\
&\qquad\qquad = p(x_t | z_t = s) \\
&\quad \cdot \left(\sum_{s'} p(z_t = s | x_1, x_2, \ldots, x_{t-1}, z_{t-1} = s') \cdot p(x_1, x_2, \ldots, x_{t-1}, z_{t-1} = s') \right) \\
&= \underbrace{p(x_t | z_t = s)}_{\text{emission model}} \cdot \left(\sum_{s'} \underbrace{p(z_t = s | z_{t-1} = s')}_{\text{transition model}} \cdot \underbrace{p(x_1, x_2, \ldots, x_{t-1}, z_{t-1} = s')}_{\text{forward variable for previous timestep}} \right) \\
&= b_s(x_t) \cdot \left(\sum_{s'} A_{s,s'} \cdot \alpha_{s',t-1} \right)
\end{aligned}
$$

Let's see how to get to each line of this equation from the previous line.

Line 1 is the definition of the forward variable $\alpha_{s,t}$.

Line 2 is the chain rule $p(A, B) = p(A|B) \cdot p(B)$, where A is x_t and B is all the other variables.

In Line 3, we apply Assumption 2: the probability of observation x_t depends only on the current state z_t.

In Line 4, we marginalize over all the possible states in the previous timestep $t - 1$.

In Line 5, we apply the chain rule again.

In Line 6, we apply Assumption 1: the current state depends only on the previous state.

In Line 7, we substitute in the emission probability, the transition probability, and the forward variable for the previous timestep, to get the complete recursion.

The formula above can be used for $t = 2 \rightarrow T$. At $t = 1$, there is no previous state, so instead of the transition matrix A, we use the state priors π, which tell us the probability of starting in each state. Thus for $t = 1$, the forward variables are computed as follows:

$$\begin{aligned}\alpha_{s,1} &= p(x_1, z_1 = s) \\ &= p(x_1 | z_1 = s) \cdot p(z_1 = s) \\ &= b_s(x_1) \cdot \pi_s\end{aligned}$$

Finally, to compute $p(\mathbf{x}) = p(x_1, x_2, \ldots, x_T)$, we marginalize over $\alpha_{s,T}$, the forward variables computed in the last timestep:

$$p(\mathbf{x}) = \sum_s p(x_1, x_2, \ldots, x_T, z_T = s) = \sum_s \alpha_{s,T}$$

You can get from this formulation to the log domain formulation by taking the log of the forward variable, and using these identities:

- $\log(a \cdot b) = \log a + \log b$
- $\log(a + b) = \log\left(e^{\log a} + e^{\log b}\right) = \text{logsumexp}(\log a, \log b)$

Problem-2: How do we compute $\text{argmax}_{\mathbf{z}} p(\mathbf{z}|\mathbf{x})$?

Given an observation sequence $\mathbf{x}$, we may want to find the most likely sequence of states that could have generated $\mathbf{x}$. (Given the sequence of selfies, we want to infer what cities the friend visited.) In other words, we want $\text{argmax}_{\mathbf{z}} p(\mathbf{z}|\mathbf{x})$.

We can use Bayes' rule to rewrite this expression:

$$\begin{aligned}\underset{\mathbf{z}}{\text{argmax}}\, p(\mathbf{z}|\mathbf{x}) &= \underset{\mathbf{z}}{\text{argmax}}\, \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})} \\ &= \underset{\mathbf{z}}{\text{argmax}}\, p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\end{aligned}$$

Hmm! That last expression, $\underset{\mathbf{z}}{\text{argmax}}\, p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$, looks suspiciously similar to the intractable expression we encountered before introducing the forward algorithm,

$$\sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$$

And indeed, just as the intractable ***sum*** over all $\mathbf{z}$ can be implemented efficiently using the forward algorithm, so too this intractable ***argmax*** can be implemented efficiently using a similar divide-and-conquer algorithm: the legendary Viterbi algorithm!

The Viterbi Algorithm:

> for $s = 1 \rightarrow N$:
>
> $\quad \delta_{s,1} := b_s(x_1) \cdot \pi_s$
>
> $\quad \psi_{s,1} := 0$
>
> for $t = 2 \rightarrow T$:
>
> $\quad$ for $s = 1 \rightarrow N$:
>
> $\quad\quad \delta_{s,t} := b_s(x_t) \cdot \left(\max_{s'} A_{s,s'} \cdot \delta_{s',t-1} \right)$
>
> $\quad\quad \psi_{s,t} := \operatorname{argmax}_{s'} A_{s,s'} \cdot \delta_{s',t-1}$
>
> $z_T^* := \operatorname{argmax}_s \delta_{s,T}$
>
> for $t = T - 1 \rightarrow 1$:
>
> $\quad z_t^* := \psi_{z_{t+1}^*,t+1}$
>
> $\mathbf{z}^* := \{z_1^*, \ldots, z_T^*\}$
>
> return $\mathbf{z}^*$

The Viterbi algorithm looks somewhat gnarlier than the forward algorithm, but it is essentially the same algorithm, with two tweaks: 1) instead of taking the sum over previous states, we take the max; and 2) we record the argmax of the previous states in a table, and loop back over this table at the end to get $\mathbf{z}^*$, the most likely state sequence. (And like the forward algorithm, we should run the Viterbi algorithm in the log domain for better numerical stability.)

Let's add the Viterbi algorithm to our PyTorch model:

```
def viterbi(self, x, T):
  """
  x : IntTensor of shape (batch size, T_max)
  T : IntTensor of shape (batch size)
  Find argmax_z log p(x|z) for each (x) in the batch.
  """
  if self.is_cuda:
    x = x.cuda()
    T = T.cuda()

  batch_size = x.shape[0]; T_max = x.shape[1]
  log_state_priors = ... # TODO
  log_delta = ... # TODO
  psi = torch.zeros(...).long() # TODO
  if self.is_cuda:
    log_delta = log_delta.cuda()
    psi = psi.cuda()

  log_delta[:, 0, :] = ... # TODO: Use emission model and log state
priors
  for t in range(1, T_max):
    max_val, argmax_val = self.transition_model.maxmul(log_delta[:, t-
1, :])
    log_delta[:, t, :] = self.emission_model(x[:,t]) + max_val
    psi[:, t, :] = argmax_val

  # Get the log probability of the best path
  log_max = log_delta.max(dim=2)[0]
  best_path_scores = torch.gather(...) # TODO

  # This next part is a bit tricky to parallelize across the batch,
  # so we will do it separately for each example.
  z_star = []
  for i in range(0, batch_size):
    z_star_i = [...] # TODO
    for t in range(T[i] - 1, 0, -1):
      z_t = psi[i, t, z_star_i[0]].item()
      z_star_i.insert(0, z_t)

    z_star.append(z_star_i)

  return z_star, best_path_scores # return both the best path and its
log probability
```

```
def transition_model_maxmul(self, log_alpha):
  log_transition_matrix = ... # TODO

  out1, out2 = maxmul(...) # TODO
  return out1.transpose(0,1), out2.transpose(0,1)
```

```
def maxmul(log_A, log_B):
  """
  log_A : m x n
  log_B : n x p
  output : m x p matrix

  Similar to the log domain matrix multiplication,
  this computes out_{i,j} = max_k log_A_{i,k} + log_B_{k,j}
  """
  m = log_A.shape[0]
  n = log_A.shape[1]
  p = log_B.shape[1]

  log_A_expanded = torch.stack([log_A] * ..., dim=2) # TODO
  log_B_expanded = torch.stack([log_B] * ..., dim=0) # TODO

  elementwise_sum = ... # TODO
  out1,out2 = torch.max(elementwise_sum, dim=1)

  return out1,out2

TransitionModel.maxmul = transition_model_maxmul
HMM.viterbi = viterbi
```

Try running Viterbi on an input sequence, given the vowel/consonant HMM:

```
x = torch.stack( [torch.tensor(encode("aba")),
torch.tensor(encode("abb"))] )
T = torch.tensor([3,3])
print(model.viterbi(x, T))
```

For $\mathbf{x} = aba$, the Viterbi algorithm returns $\mathbf{z}^* = \{1,0,1\}$. This corresponds to "vowel, consonant, vowel" according to the way we defined the states above, which is correct for this input sequence. Yay!

For $\mathbf{x} = abb$, the Viterbi algorithm still returns a $\mathbf{z}^*$, but we know this is gibberish because "vowel, consonant, consonant" is impossible under this HMM, and indeed the log probability of this path is $-\infty$.

Let's compare the "forward score" (the log probability of all possible paths, returned by the forward algorithm) with the "Viterbi score" (the log probability of the maximum likelihood path, returned by the Viterbi algorithm):

```
print(model.forward(x, T))
print(model.viterbi(x, T)[1])
```

The two scores are the same! That's because in this instance there is only one possible path through the HMM, so the probability of the most likely path is the same as the sum of the probabilities of all possible paths.

In general, though, the forward score and Viterbi score will always be somewhat close. This is because of a property of the logsumexp function: $\text{logsumexp}(\mathbf{x}) \approx \max(\mathbf{x})$. (logsumexp is sometimes referred to as the "smooth maximum" function.)

```
x = torch.tensor([1., 2., 3.])
print(x.max(dim=0)[0])
print(x.logsumexp(dim=0))
```

Problem-3: How do we train the model?

Earlier, we hard-coded an HMM to have certain behavior. What we would like to do instead is have the HMM learn to model the data on its own. And while it is possible to use supervised learning with an HMM (by hard-coding the emission model or the transition model) so that the states have a particular interpretation, the really cool thing about HMMs is that they are naturally unsupervised learners, so they can learn to use their different states to represent different patterns in the data, without the programmer needing to indicate what each state means.

Like many machine learning models, an HMM can be trained using maximum likelihood estimation, i.e.:

$$\theta^* = argmin_{\,\theta} - \sum_{\mathbf{x^i}\log} p_\theta(\mathbf{x^i})$$

where $\mathbf{x^1}, \mathbf{x^2}$, are training examples.

The standard method for doing this is the Expectation-Maximization (EM) algorithm, which for HMMs is also called the "Baum-Welch" algorithm. In EM training, we alternate between an "E-step", where we estimate the values of the latent variables, and an "M-step", where the model parameters are updated given the estimated latent variables. (Think k −means, where you guess which cluster each data point belongs to, then reestimate where the

clusters are, and repeat.) The EM algorithm has some nice properties: it is guaranteed at each step to decrease the loss function, and the E-step and M-step may have an exact closed form solution, in which case no pesky learning rates are required.

But because the HMM forward algorithm is differentiable with respect to all the model parameters, we can also just take advantage of automatic differentiation methods in libraries like PyTorch and try to minimize $-\log p_\theta(\mathbf{x})$ directly, by backpropagating through the forward algorithm and running stochastic gradient descent. That means we don't need to write any additional HMM code to implement training: `loss.backward()` is all you need.

Here we will implement SGD training for an HMM in PyTorch. First, some helper classes:

```
import torch.utils.data
from collections import Counter
from sklearn.model_selection import train_test_split

class TextDataset(torch.utils.data.Dataset):
  def __init__(self, lines):
    self.lines = lines # list of strings
    collate = Collate() # function for generating a minibatch from
strings
    self.loader = torch.utils.data.DataLoader(self, batch_size=1024,
num_workers=1, shuffle=True, collate_fn=collate)

  def __len__(self):
    return len(self.lines)

  def __getitem__(self, idx):
    line = self.lines[idx].lstrip(" ").rstrip("\n").rstrip("
").rstrip("\n")
    return line
```

```
class Collate:
  def __init__(self):
    pass

  def __call__(self, batch):
    """
    Returns a minibatch of strings, padded to have the same length.
    """
    x = []
    batch_size = len(batch)
    for index in range(batch_size):
      x_ = batch[index]

      # convert letters to integers
      x.append(...) # TODO: Encode

    # pad all sequences with 0 to have same length
    x_lengths = [...] # TODO: List comprehension
    T = max(x_lengths)
    for index in range(batch_size):
      x[index] += ... # TODO
      x[index] = torch.tensor(x[index])

    # stack into single tensor
    x = torch.stack(x)
    x_lengths = torch.tensor(x_lengths)
    return (x,x_lengths)
```

Let's load some training/testing data. By default, this will use the unix "words" file, but you could also use your own text file.

```
!wget
https://raw.githubusercontent.com/lorenlugosch/pytorch_HMM/master/data/
train/training.txt
```

```
filename = "training.txt"

with open(filename, "r") as f:
  lines = f.readlines() # each line of lines will have one word

alphabet = list(Counter(("".join(lines))).keys())
train_lines, valid_lines = train_test_split(lines, test_size=0.1,
random_state=42)
train_dataset = ... # TODO
valid_dataset = ... # TODO

M = len(alphabet)
```

We will use a Trainer class for training and testing the model:

```
from tqdm import tqdm # for displaying progress bar

class Trainer:
  def __init__(self, model, lr):
    self.model = model
    self.lr = lr
    self.optimizer = torch.optim.Adam(model.parameters(), lr=self.lr,
weight_decay=0.00001)

  def train(self, dataset):
    train_loss = 0
    num_samples = 0
    self.model.train()
    print_interval = 50
    for idx, batch in enumerate(tqdm(dataset.loader)):
      x,T = batch
      batch_size = len(x)
      num_samples += batch_size
      log_probs = ... # TODO
      loss = ... # TODO
      self.optimizer.zero_grad()
      loss.backward()
      self.optimizer.step()
      train_loss += loss.cpu().data.numpy().item() * batch_size
      if idx % print_interval == 0:
        print("loss:", loss.item())
        for _ in range(5):
          sampled_x, sampled_z = self.model.sample()
          print(decode(sampled_x))
          print(sampled_z)
    train_loss /= num_samples
    return train_loss
```

```
  def test(self, dataset):
    test_loss = 0
    num_samples = 0
    ... # TODO: Call model eval
    print_interval = 50
    for idx, batch in enumerate(dataset.loader):
      x,T = batch
      batch_size = len(x)
      num_samples += batch_size
      log_probs = ... # TODO
      loss = ... # TODO
      test_loss += loss.cpu().data.numpy().item() * batch_size
      if idx % print_interval == 0:
        print("loss:", loss.item())
        sampled_x, sampled_z = self.model.sample()
        print(decode(sampled_x))
        print(sampled_z)
    test_loss /= num_samples
    return test_loss
```

Finally, initialize the model and run the main training loop. Every 50 batches, the code will produce a few samples from the model. Over time, these samples should look more and more realistic.

```
# Initialize model
model = HMM(N=64, M=M)

# Train the model
num_epochs = 10
trainer = Trainer(model, lr=0.01)

for epoch in range(num_epochs):
        print("========= Epoch %d of %d =========" % (epoch+1,
num_epochs))
        train_loss = ... # TODO
        valid_loss = ... # TODO

        print("========= Results: epoch %d of %d =========" % (epoch+1,
num_epochs))
        print("train loss: %.2f| valid loss: %.2f\n" % (train_loss,
valid_loss) )
```

You may wish to try different values of N and see what the impact on sample quality is.

```
x = torch.tensor(encode("quack")).unsqueeze(0)
T = torch.tensor([5])
print(model.viterbi(x,T))

x = torch.tensor(encode("quick")).unsqueeze(0)
T = torch.tensor([5])
print(model.viterbi(x,T))

x = torch.tensor(encode("qurck")).unsqueeze(0)
T = torch.tensor([5])
print(model.viterbi(x,T)) # should have lower probability---in English
only vowels follow "qu"

x = torch.tensor(encode("qiick")).unsqueeze(0)
T = torch.tensor([5])
print(model.viterbi(x,T)) # should have lower probability---in English
only "u" follows "q"
```

HMMs used to be very popular in natural language processing, but they have largely been overshadowed by neural network models like RNNs and Transformers. Still, it is fun and instructive to study the HMM; some commonly used machine learning techniques like Connectionist Temporal Classification are inspired by HMM methods. HMMs are still used in conjunction with neural networks in speech recognition, where the assumption of a one-hot state makes sense for modelling phonemes, which are spoken one at a time.

Conditional Random Fields (CRF)

The bag of words (BoW) approach works well for multiple text classification problems. This approach assumes that presence or absence of word(s) matter more than the sequence of the words. However, there are problems such as entity recognition, part of speech identification where word sequences matter as much, if not more. Conditional Random Fields (CRF) comes to the rescue here as it uses word sequences as opposed to just words.

Part-of-Speech (POS) Tagging

In POS tagging, the goal is to label a sentence (a sequence of words or tokens) with tags like ADJECTIVE, NOUN, PREPOSITION, VERB, ADVERB, ARTICLE.
For example, given the sentence "Bob drank coffee at Starbucks", the labeling might be "Bob (NOUN) drank (VERB) coffee (NOUN) at (PREPOSITION) Starbucks (NOUN)".
So, we will build a Conditional Random Field to label sentences with their parts of speech.

```
# install crf and nltk in python if not installed
!pip install python-crfsuite
!pip install nltk
!pip install lxml
```

Upload the dataset file to your Colab session, or place it in the appropriate directory, if you are running locally.

A few words about the Dataset:
To train a named entity recognition model, we need some labelled data. The dataset that will be used below is the Reuters-128 dataset, which is an English corpus in the NLP Interchange Format (NIF). It contains 128 economic news articles. The dataset contains information for 880 named entities with their position in the document and a URI of a DBpedia resource identifying the entity. It was created by the Agile Knowledge Engineering and Semantic Web research group at Leipzig University, Germany. More details can be found in their paper.

Step-1: Prepare the Dataset for Training from the XML format

In order to prepare the dataset for training, we need to label every word (or token) in the sentences to be either irrelevant or part of a named entity. Since the data is in XML format, we can make use of BeautifulSoup to parse the file and extract the data as follows:

```
from bs4 import BeautifulSoup as bs
from bs4.element import Tag
import codecs

# Read data file and parse the XML
with codecs.open("reuters.xml", "r", "utf-8") as infile:
    soup = bs(infile, "html5lib")

docs = []
for elem in soup.find_all("document"):
    texts = []

    # Loop through each child of the element under
"textwithnamedentities"
    for c in elem.find("textwithnamedentities").children:
        if type(c) == Tag:
            if c.name == "namedentityintext":
                label = "N"  # part of a named entity
            else:
                label = "I"  # irrelevant word
            for w in c.text.split(" "):
                if len(w) > 0:
                    texts.append((w, label))
    docs.append(texts)
    #Prepare labels as "N" representing part of a named entity and "I"
for irrelevant word

docs[0]
```

Step-2: Generating Part-of-Speech Tags

```
import nltk
nltk.download('averaged_perceptron_tagger')

data = []
for i, doc in enumerate(docs):

    # Obtain the list of tokens in the document
    tokens = [...] # TODO

    # Perform POS tagging
    tagged = ... # TODO

    # Take the word, POS tag, and its label
    data.append([(w, pos, label) for (w, label), (word, pos) in
zip(doc, tagged)])
data[0]
```

Step-3: Generating Features

To train a CRF model, we need to create features for each of the tokens in the sentences. One particularly useful feature in NLP is the part-of-speech (POS) tags of the words. They indicates whether a word is a noun, a verb or an adjective. (In fact, a POS tagger is also usually a trained CRF model.)

Below are some of the commonly used features for a word w in named entity recognition:

- The word w itself (converted to lowercase for normalisation)
- The prefix/suffix of w (e.g. -ion)
- The words surrounding w, such as the previous and the next word
- Whether w is in uppercase or lowercase
- Whether w is a number, or contains digits
- The POS tag of w, and those of the surrounding words
- Whether w is or contains a special character (e.g. hypen, dollar sign)

We can use NLTK's POS tagger to generate the POS tags for the tokens in our documents as follows:

```
def word2features(doc, i):
    word = doc[i][0]
    postag = doc[i][1]

    # Common features for all words
    features = [
        'bias',
        'word.lower=' + word.lower(),
        'word[-3:]=' + word[-3:],
        'word[-2:]=' + word[-2:],
        'word.isupper=%s' % word.isupper(),
        'word.istitle=%s' % word.istitle(),
        'word.isdigit=%s' % word.isdigit(),
        'postag=' + postag
    ]

    # Features for words that are not
    # at the beginning of a document
    if i > 0:
        word1 = doc[i-1][0]
        postag1 = doc[i-1][1]
        features.extend([
            '-1:word.lower=' + word1.lower(),
            '-1:word.istitle=%s' % word1.istitle(),
            '-1:word.isupper=%s' % word1.isupper(),
            '-1:word.isdigit=%s' % word1.isdigit(),
            '-1:postag=' + postag1
        ])
    else:
        # Indicate that it is the 'beginning of a document'
        features.append('BOS')
```

```
    # Features for words that are not
    # at the end of a document
    if i < len(doc)-1:
        word1 = doc[i+1][0]
        postag1 = doc[i+1][1]
        features.extend([
            '+1:word.lower=' + word1.lower(),
            '+1:word.istitle=%s' % word1.istitle(),
            '+1:word.isupper=%s' % word1.isupper(),
            '+1:word.isdigit=%s' % word1.isdigit(),
            '+1:postag=' + postag1
        ])
    else:
        # Indicate that it is the 'end of a document'
        features.append('EOS')

    return features
```

Step-4: Training the Model

To train the model, we need to first prepare the training data and the corresponding labels. Also, to be able to investigate the accuracy of the model, we need to separate the data into training set and test set. Below are some codes for preparing the training data and test data, using the train_test_split function in scikit-learn.

```
from sklearn.model_selection import train_test_split

# A function for extracting features in documents
def extract_features(doc):
    return [...] # TODO

# A function fo generating the list of labels for each document
def get_labels(doc):
    return [...] # TODO

X = [extract_features(doc) for doc in data]
y = [get_labels(doc) for doc in data]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)
```

In pycrfsuite, A CRF model in can be trained by first creating a trainer, and then submit the training data and corresponding labels to the trainer. After that, set the parameters and call train() to start the training process. For the complete list of parameters, one can refer to the documentation of CRFSuite. With the very small dataset in this example, the training with max_iterations=200 can be finished in a few seconds. Below is the code for creating the trainer and start training the model:

```
import pycrfsuite
trainer = pycrfsuite.Trainer(verbose=True)

# Submit training data to the trainer
for xseq, yseq in zip(X_train, y_train):
    trainer.append(xseq, yseq)

# Set the parameters of the model
trainer.set_params({
    # coefficient for L1 penalty
    'c1': 0.1,
    # coefficient for L2 penalty
    'c2': 0.01,
    # maximum number of iterations
    'max_iterations': 200,
    # whether to include transitions that are possible, but unobserved
    'feature.possible_transitions': True
})

# Provide a file name as a parameter to the train function, such that
# the model will be saved to the file when training is finished
trainer.train('crf.model')
```

Step-5: Checking the Results

Once we have the model trained, we can apply it on our test data and see whether it gives reasonable results. Assuming that the model is saved to a file named crf.model. The following block of code shows how we can load the model into memory, and apply it on to our test data.

```
tagger = pycrfsuite.Tagger()
tagger.open('crf.model')
y_pred = [...] # TODO: Tag using tagger for X_Test

# Let's take a look at a random sample in the testing set
i = 12
for x, y in zip(y_pred[i], [x[1].split("=")[1] for x in X_test[i]]):
    print("%s (%s)" % (y, x))
```

To study the performance of the CRF tagger trained above in a more quantitative way, we can check the precision and recall on the test data. This can be done very easily using the classification_report function in scikit-learn. However, given that the predictions are sequences of tags, we need to transform the data into a list of labels before feeding them into the function.

```
import numpy as np
from sklearn.metrics import classification_report

# Create a mapping of labels to indices
labels = {"N": 1, "I": 0}

# Convert the sequences of tags into a 1-dimensional array
predictions = np.array([...]) # TODO
truths = np.array([labels[...] for row in y_test for ... in row])# TODO

# Print out the classification report
print(classification_report(
    truths, predictions,
    target_names=["I", "N"]))
```

Exercise

1. In the section of HMM,
    a. Complete all the TODOs (blanks) wherever mentioned, including the following algorithms in the code
    b. Read the Forward Algorithm (Log Domain) and complete the code
    c. Read the Viterbi Algorithm and add the code to the PyTorch Model
2. In the second section of CRF,
    a. Complete all the TODOs (blanks) wherever mentioned
    b. Print the loss and classification report, by changing any hyperparameter/ hyperparameters (for at least 3 values, say first is `lr=0.001`; second is a different value for `c1`, `c2`, `#epochs`, `lr=0.01`; and third is original values of `c1`, `c2`, with `lr=5e-3`)