MATH80629A — Machine Learning I: Large Scale Data Analysis and Decision Making

Solutions to the Fall 2020 exam.

Q. 1

---

a.

i. [2 Points]

Unsupervised learning.

This is the dimensionality reduction task, which involves representing given high-dimensional data in a lower-dimensional space.

The labels for the input data points are not available.

[+1: Correct answer, +1: Reasoning]

ii. [2 Points]

Supervised learning.

The input data points are the English sentences, which are available along with their target French translations.

[+1: Correct answer, +1: Reasoning]

iii. [2 Points]

Reinforcement learning.

The princess is the agent that interacts with her environment to carry out sequential decisions. She must learn a policy— to take appropriate actions based on her state in order to gain reward and advance through the levels to save the prince.

[+1: Correct answer, +1: Reasoning]

iv. [2 Points]

Supervised learning.

In order to learn to recognize objects in the game, it is not necessary to carry out the sequential decision making process as in the previous case.

Instead, having a dataset of the frames from the game along with the annotations of the bounding boxes for the game objects would suffice. Thus, the task is a supervised learning one.

[+1: Correct answer, +1: Reasoning]

[+1: Other answer, +1: Appropriate reasoning]

---

b. [2 Points]

There are multiple tasks where supervised learning and unsupervised learning approaches can be combined for better performing the task at hand.

For example, consider the task of supervised classification where the input data is extremely high-dimensional. Learning a supervised classifier directly on such data can prove to be difficult and computationally expensive. Instead, one can use unsupervised learning techniques of dimensionality reduction, such as principle component analysis (PCA), in order to first learn a lower-dimensional representation of the data. One can then train a supervised classifier on this representation, which can prove to be easier and computationally feasible.

However, there are multiple correct answers here. Instead of the approach above, we can also use a mix of supervised and unsupervised data to design a **semi-supervised** approach for better performing the task at hand.

[+2: Any correct example with explanation (binary marking)]

---

c. [4 Points]

A regularizer is any approach, or any expression, that can augment a machine learning approach in order to avoid overfitting to the training data and generalize better to the unseen testing data. It can constrain the (effective) capacity of a model without changing the model class itself in order to achieve this effect. Regularizers help inject

into the learning process our prior knowledge/bias in order to prefer certain (simpler) solutions over the other (complex) solutions.

In the case of deep networks, dropout is an example of a regularizer. Dropout involves randomly "turning off" a fraction of neurons during the training of a deep network in order to stop the co-learning in neurons. Another example is early stopping, which involves monitoring the performance of a deep network on a separate validation set as training progresses, and stopping the training when this performance starts to deteriorate. Early stopping stops the training of the deep network before it starts to overfit to the training data.

In the case of supervised learning, we are given with a dataset $\left\{\mathbf{x}^i, y^i\right\}_{i=1}^{n}$ and our aim is to learn a function f having trainable parameters $\mathbf{w}$, which inputs the data point $\mathbf{x}$ in order to predict its label $\hat{y} = f(\mathbf{x}; \mathbf{w})$. This is done via empirical risk minimization.

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(f\left(\mathbf{x}^i; \mathbf{w}\right), y^i\right)$$

One of the widely used regularizers in this scenario is the so-called L^2 regularizer, which is defined as the sum of the squares of the trainable parameters.

$$R(\mathbf{w}) = \|\mathbf{w}\|_2^2$$

The regularized empirical risk minimization learns the weights by minimizing the empirical risk as well as the aforesaid regularizer scaled by a hyper-parameter λ.

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(f\left(\mathbf{x}^i; \mathbf{w}\right), y^i\right) + \lambda \|\mathbf{w}\|_2^2$$

Too small a value of λ can result in overfitting and too large a value of λ can result in underfitting, which makes the choice of λ crucial for improving generalization.

[+2: Explanation of regularizers, +2 Example with explanation]

---

d. [3 Points]

The term i.i.d. stands for independent and identically distributed. It refers to the assumption that for a given dataset $\{\mathbf{z}^i\}$, its data points $\mathbf{z}^i$ follow the same underlying distribution and that all the data points are independent of each other.

There are multiple reasons for which the i.i.d. assumption is important.

Firstly, it allows us to understand the distribution of an entire dataset in terms of only the distribution followed by each sample. However, more importantly, it is required to

ascertain that the model learned by a machine learning approach is indeed correct and is the best possible model for unseen data.

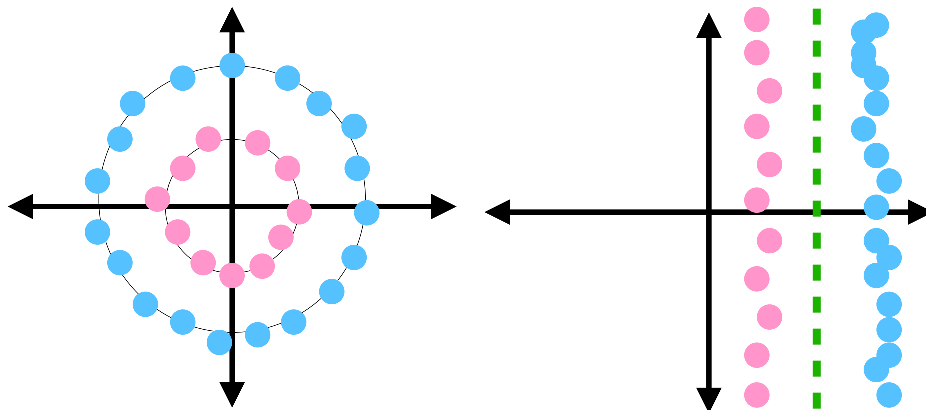
Note that we train a machine learning approach with a training dataset and evaluate it on a separate testing dataset. With the i.i.d. assumption, we can be certain that the training and evaluation of the model is being carried out on the same data distribution. In the absence of the i.i.d. assumption, the distribution of the training set would be systematically different from the testing set and thus, there would not be any guarantees that a model trained on the training dataset would indeed generalize well on an unseen testing dataset.

[+1: IID definition, +2 Explanation]

---

e. [3 Points] [Only on some versions of the exam]

Representation learning refers to the machine learning paradigm of discovering useful representations of the given data, thereby facilitating the task at hand. Often, the available data is not best suitable to carry out the task at hand. Thus, representation learning involves extracting features, or useful information, from this data that can make the downstream task at hand easier. Alternatively, by extracting good representations, we can learn a simpler model on the given dataset.



Dataset A: No linear classifier　　Linearly separable representation

Consider a simple example of learning a linear classifier on the dataset A. This dataset can not be classified correctly with a linear classifier and thus, we would require a higher capacity model to perform well on it. However, if we change the representation of the data points by converting them from Cartesian coordinates to polar coordinates, we can see that the data becomes linearly separable. Thus, the representation of the data is important; good representations can enable learning simpler models of the

data. Deep learning has the peculiarity that instead of using hand-crafted features of the data for performing the downstream task, the model *learns* the best features for the task. For instance, in a convolutional neural network (CNN) for performing the task of image classification, the initial layers learn features of the data, which are then classified with a learned classifier. Each layer of the CNN learns progressively complex representations of the data by composing the features output by the previous layer.

[+2: Explanation, +1 Example]

Q. 2

---

a. [4 Points]

Whereas $K-$means does **hard assignment** of points to clusters (i.e., each point is assigned to a single cluster), GMMs perform **soft assignment** (i.e., each point is given a probability distribution over cluster assignments). This can be especially useful for points at boundaries where we want our model to reflect greater "uncertainty". In addition, GMMs provide **modelling flexibility** since the entire covariance matrix for each cluster can be learned. This allows for modeling non-spherical cluster shapes. In contrast, $K-$means clusters are always spherical (equivalent to having a diagonal covariance), which may be a strong assumption to make depending on the data.

**[+2: Hard/soft assignment (1/2: if soft-clustering without description),
+2: Modeling flexibility (+1: entire covariance matrix, +1: non-spherical cluster)]**

---

b. [6 Points]

There are two important changes that we must consider in order to make the algorithm work for other types of data— *i.* We need to use an **appropriate distance metric** for the given data type. This change should be done is both line 7 and line 9. *ii.* We need to update the cluster centers to ensure that they are valid members of the data distribution that we have. This change should be done in line 3 and line 9.

In the specific case of integer data, treating it as float is incorrect. A valid alternative would be to use the $K-$medoids approach that ensures the argmin over cluster centers is instead carried out with an argmin over the data points. In the case of integer data, we can use L^1-distance (also called the Manhattan distance).

In general, a somewhat creative answer would be to encode the given data into real-valued vectors and then perform $K-$means approach in the encoded space.

[+3: Change of metric in BOTH line 7 and line 9 (2/3: Insufficient description, 1/3: Stating only data encoding), +1: Suggest a precise alternative metric, +2: Updating cluster centers correctly (1/2: Stating that the cluster centers may not remain valid data points but not resolving the issue)]

---

c. [6 Points]

Yes, $K-$means can be distributed using Apache Spark.

Note that in line 7, updating the cluster responsibilities for any data point x_i depends only on the data point itself and the cluster centers. This computation does NOT depend on any other data point x_j. Thus, this calculation can be distributed in a straight-forward manner as a map operation over every data point. Once we have updated all the responsibilities, we then need to update the cluster centers in line 9. Once again, note that updates for any cluster center μ_i does NOT depend on any other cluster centers μ_j. This update depends only on the cluster center itself and the data points assigned to it. Thus, this calculation can also be distributed as a reduce operation, aggregating over the assigned data points for a cluster center.

Given m number of computers, we can expect roughly a speedup of a factor of m at each of the parallelizable operations (lines 7 and 9). However, this would be true as long as the number of clusters K is larger than the number of computers m, which may be a reasonable assumption for large datasets.

For the interested reader, a longer discussion of this question can be found in this nicely written report:

https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/bodoia.pdf

[+1: Yes/No, +3: Description of the answer (2/3: Insufficient/vague answers, 0/3: Use MLib), +2: Discussion of potential gains in performance]

Q. 3

a.

i. [3 Points]

Interpretation 1:

Suppose we want to find the neural network with no hidden layers such that it can process 100-dimensional inputs and produce 2-dimensional outputs.

Then, there can only be 1 set of weights and biases $\mathbf{W}^1$ in the neural network.

The number of trainable parameters in this set is:

$$(100 + 1) \times 2 = 202 \text{ (with biases) or } 100 \times 2 = 200 \text{ (without biases)}$$

Interpretation 2:

Suppose we want to find the neural network that can process 100-dimensional inputs and produce 2-dimensional outputs with the least number of trainable parameters. Then, the neural network must have 1 hidden layer containing 1 neuron. Thus, there can only be two sets of weights and biases $\mathbf{W}^1$, $\mathbf{W}^2$ in the neural network.

The number of parameters in $\mathbf{W}^1$:

$$(100 + 1) \times 1 = 101 \text{ (with biases) or } 100 \times 1 = 100 \text{ (without biases)}$$

The number of parameters in $\mathbf{W}^2$:

$$(1 + 1) \times 2 = 4 \text{ (with biases) or } 1 \times 2 = 2 \text{ (without biases)}$$

So, the total number of parameters is:

$$101 + 4 = 105 \text{ (with biases) or } 100 + 2 = 102 \text{ (without biases)}$$

[+2: Explanation, +1: Final answer; Answers with/without biases are accepted]

ii. [3 Points]

In this case, the neural network processes 100-dimensional inputs with two hidden layers having 20 neurons each and then produces a 2-dimensional outputs. Thus, there are three sets of weights and biases $\mathbf{W}^1, \mathbf{W}^2, \mathbf{W}^3$ in the neural network.

The number of parameters in $\mathbf{W}^1$:

$$(100 + 1) \times 20 = 2020 \text{ (with biases) or } 100 \times 20 = 2000 \text{ (without biases)}$$

The number of parameters in $\mathbf{W}^2$:

$$(20 + 1) \times 20 = 420 \text{ (with biases) or } 20 \times 20 = 400 \text{ (without biases)}$$

The number of parameters in $\mathbf{W}^3$:

$$(20 + 1) \times 2 = 42 \text{ (with biases) or } 20 \times 2 = 40 \text{ (without biases)}$$

So, the total number of parameters is:

$$2020 + 420 + 42 = 2482 \text{ (with biases)}$$

$$\text{or } 2000 + 400 + 40 = 2440 \text{ (without biases)}$$

[+2: Explanation, +1: Final answer; Answers with/without biases are accepted]

---

b.

i. [8 Points]

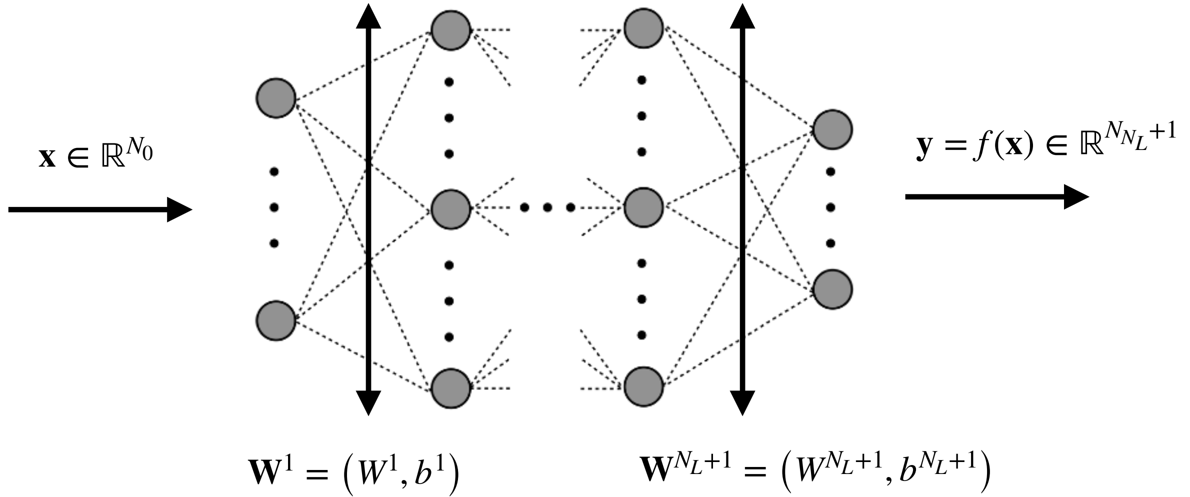
The given data consists of ratings given by a user to a particular item and thus, we can model the given ratings data using a **feed-forward neural network (FFNN)** that inputs the information about the user and the item in order to predict the rating that the user would give for that item.

Note that the users and the products are **categorical variables**, taking discrete values that need to be fed as inputs to a neural network. Thus, we must **encode** the user and item IDs in order to obtain their vector representations. This can be simply achieved by using the **one-hot encoding** separately for the users and for the items. Suppose there are N_U number of users and N_I number of items. Then, a user with ID u and an item with ID i are encoded as vectors $\overrightarrow{u}$ and $\vec{i}$ as follows:

$$\overrightarrow{u} = [0 \ \dots \ 0 \ \underbrace{1}_{u-\text{th location}} \ 0 \ \dots \ 0]^\top \in \mathbb{R}^{N_U} \text{ and } \overrightarrow{i} = [0 \ \dots \ 0 \ \underbrace{1}_{i-\text{th location}} \ 0 \ \dots \ 0]^\top \in \mathbb{R}^{N_I}$$

Further, it is important to encoder the users and the items separately in order to **disentangle** their contribution to the prediction of the rating.

Note that our network must produce a rating r based on the inputs $\overrightarrow{u}$ and $\overrightarrow{i}$. Note that the rating must satisfy: $r \in [0,1]$.



Now, we consider a general FFNN block as shown above. The block has an input layer, N_L number of hidden layers, and one output layer. Thus, there are $N_L + 1$ sets of weights and biases $\mathbf{W}^1, \dots, \mathbf{W}^{N_L+1}$ with each set $\mathbf{W}^i$ consisting of weight matrix W^i and bias b^i. Suppose the number of input layer neurons be N_0. Let $W^i \in \mathbb{R}^{N_{i-1} \times N_i}$ and $b^i \in \mathbb{R}^{N_i}$, where N_i represents the number of neurons in the $i-$th layer. The output size will then be N_{N_L+1}. Now, suppose for each layer, let $\mathbf{a}^i$ denote its input. Thus, $\mathbf{a}^{i+1}$ denotes the output of the $i-$th layer, which is the input to the $(i+1)-$th layer. For the block shown above, we have $\mathbf{a}^1 = \mathbf{x} \in \mathbb{R}^{N_0}$ and $\mathbf{a}^{N_{N_L+1}} = \mathbf{y} \in \mathbb{R}^{N_{N_L+1}}$. Note that each layer computes the following function of its inputs:

$$\mathbf{a}^{i+1} = \phi^i \left(W^i \cdot \mathbf{a}^i + b^i\right) \ \ \forall i \in 1,\dots,N_L + 1$$

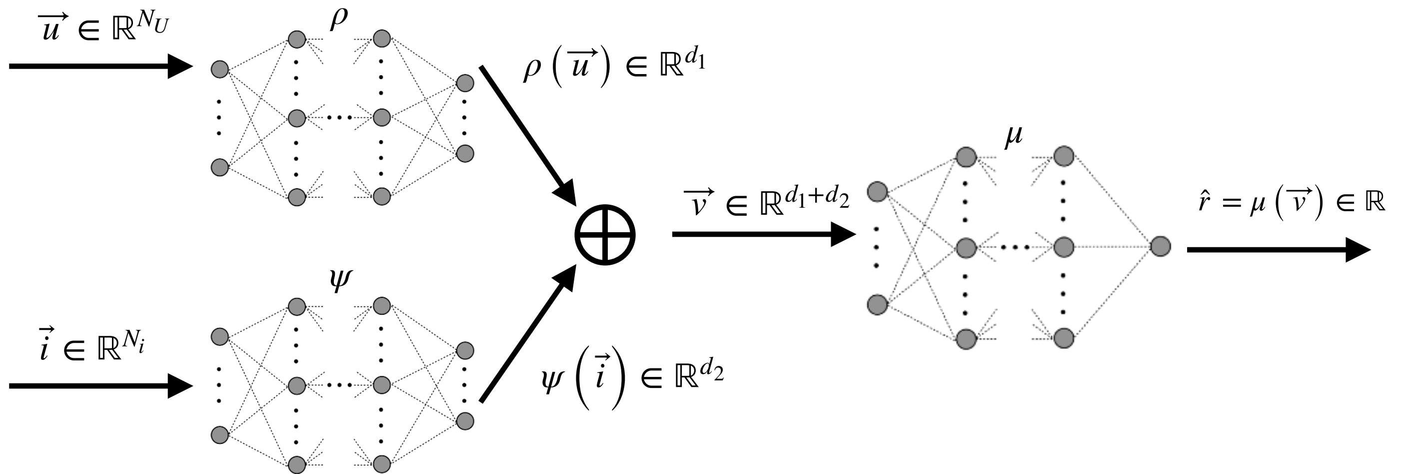
Here, ϕ^i represents the non-linearity of the $i-$th layer. Some of the widely used choices for the non-linearity are sigmoid, tanh, ReLU, etc.

Finally, note that the hyperparameters such as the number of hidden layers, the number of neurons per layer, the choices of non-linearities are dependent on and are tuned based on the task at hand.

Now, having seen a FFNN block, we consider the following NN for the given task.

This neural net has two input blocks. The first block inputs the user encoding $\overrightarrow{u}$ and produces its d_1-dimensional feature representation $\rho\left(\overrightarrow{u}\right) \in \mathbb{R}^{d_1}$. The second block inputs the item encoding $\vec{i}$ and produces its d_2-dimensional feature representation $\psi\left(\vec{i}\right) \in \mathbb{R}^{d_2}$. These two representations are then **concatenated** in order to produce a representation of the user-item pair $\left(\overrightarrow{u}, \vec{i}\right)$ as $\overrightarrow{v} = \rho\left(\overrightarrow{u}\right) \oplus \psi\left(\vec{i}\right) \in \mathbb{R}^{d_1+d_2}$. This concatenated $\left(d_1 + d_2\right)-$dimensional representation is fed to another block, which then outputs a scalar corresponding to the predicted rating $\hat{r}$ as $\hat{r} = \mu\left(\overrightarrow{v}\right) \in \mathbb{R}$.

Note that the choices of the aforesaid hyperparameters can be decided based on the given dataset but the output layer activation of the block μ must be **sigmoid** in order to output a $[0,1]-$ranged scalar at the output.



Now, we can train this network using the given dataset $D = \left\{\left(u^n, i^n\right), r^n\right\}_{n=1}^{N}$ by minimizing the empirical risk objective, which is defined as follows—

$$L = \frac{1}{N}\sum_{n=1}^{N} \parallel \hat{r}^n - r^n \parallel^2 = \frac{1}{N}\sum_{i=1}^{N} \left\| \mu\left(\rho\left(\overrightarrow{u}^n\right) \oplus \psi\left(\vec{i}^n\right)\right) - r^n \right\|^2$$

All the trainable weights can then be learned by using (stochastic) gradient descent on this objective. After training the model, for any new combination of user u^* and item i^*, we can predict the corresponding rating $r^* = \mu\left(\rho\left(\overrightarrow{u}^*\right) \oplus \psi\left(\vec{i}^*\right)\right)$ that the user would give for the item. This gives us the desired model for the ratings based on the user and item information.

[+1: Description of inputs: users/items and their encoding, +1: Disentangling user/item representation or factorization, +1: Description of output: sigmoid/[0, 1]-ranged scalar, +1: Formalization/mathematical details, +4: Sketch and description; other correct and consistent answers accepted, marked accordingly]

ii. [8 Points]

Unlike the previous question, we are given with temporal data for different users in terms of the items that they bought and the order in which they did so. Thus, we must use **recurrent neural network** architectures to model the given data. We can use any of the following— vanilla RNN, LSTM, or even GRU. We will describe an RNN below.

Similar to the previous question, since the user and item data is categorical, we must **encode** it in order to feed it into our neural network. However, we can also learn trainable **embeddings** for these categorical variables. We will describe the solution with a simple one-hot encoding as described in the previous question. For brevity, we will represent this encoding/embedding as $e(\cdot)$.

Now, note that the dataset that we have has the following form:

$$D = \left\{ \left(u^n, \left\langle i^{n,1}, i^{n,2}, \ldots, i^{n,m_n} \right\rangle \right) \right\}_{n=1}^{N}$$

Here, each data point consists of a user u^n and the (ordered) tuple of the items that the user considered. Note that for each of the data points, the lengths of the tuple m_n can in principle be different. We can model this dataset similar in the following manner— For each data point, we initialize our RNN architecture with the information about the user and then based on the items purchased by that user till time t, we try to predict the item that the user might purchase at the next time step $(t+1)$. For each data point, we perform this task for every time step $t \leq m_n$. This idea is summarized below.

Firstly, note that we can not afford to learn a different RNN for every user as we expect to model a large number of users. Thus, we must somehow **condition our RNN** for a particular user in order to predict the items considered by this user. One of the easiest ways to do this would be to initialize the hidden state h^0 of the RNN with the user's encoding/embedding: $h^0 = e\left(u^n\right)$. Now, at the first time-step, we provide as an input a special embedding titled $\langle BOS \rangle$, which serves to represent the starting token before the first item is considered and is used as an input to predict the item $\hat{i}^{n,1}$ at time $t = 1$. At each time step t, we update the hidden state of the RNN based on the current input and the previous hidden state. At each time step $t \geq 2$, we provide as

input $e\left(i^{n,t-1}\right)$, which is the encoding/embedding of the item considered by the user u^n and try to predict the next item $\hat{i}^{n,t}$ based on this input and the current hidden state h^t. Note that performing this prediction requires the information of all the previous items and the user itself, which is correctly encoded in the hidden state h^t at time t. One important detail in the prediction of the next item is that the RNN outputs a vector that depends on hidden state h^t and input $e\left(t^{n,t-1}\right)$, which must then be used to predict the next item ID. Therefor, we process this output of RNN using a small FFNN ρ having a **softmax** layer at the output. This layer would produce a distribution over all the items based on the output of the RNN at every time step t and the argmax over this distribution would give the required item ID.

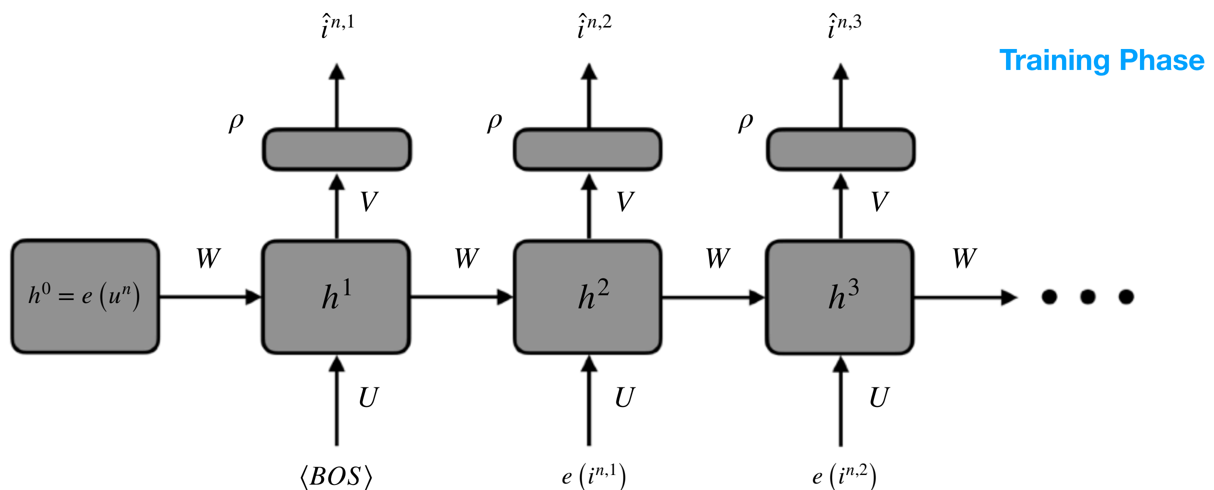
Formally, the RNN performs following computation at every time step $t \geq 1$—

$$h^t = \tanh\left(U \cdot e\left(i^{n,t-1}\right) + W \cdot h^{t-1}\right) \text{ and } \hat{i}^{n,t} = \rho\left(V \cdot h^t\right)$$

Now, the minimization objective would consist of the **cross-entropy loss** for the item prediction at each time step for all the given input data points. In particular, we have—

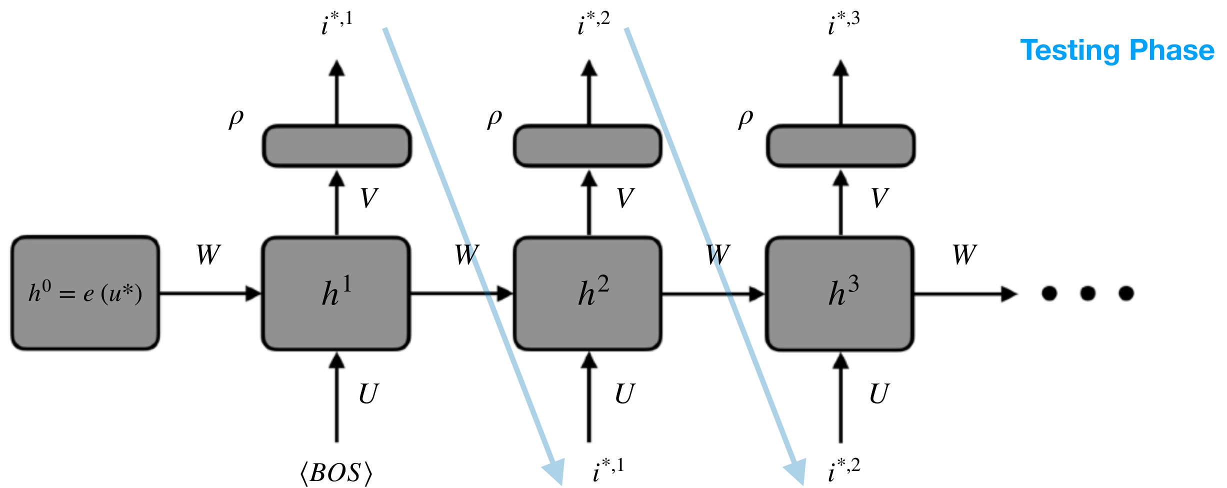
$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{m_n} \text{cross-entropy}\left(\hat{i}^{n,t}, i^{n,t}\right)$$

We perform (stochastic) gradient descent on this objective to learn trainable weights.



Once the training is complete, we can use the same recurrent neural network to make recommendations for different users as follows. Suppose we are given a user u^*. We first initialize the hidden state h^0 of the RNN with the user's encoding/embedding: $h^0 = e\left(u^*\right)$. We then input the trained RNN with the special embedding $\langle BOS \rangle$ and obtain the item prediction $i^{*,1}$ for time step $t = 1$ at the output. We take this prediction

and feed it back to the RNN as input at time step $t = 2$ and obtain as output the next item prediction $i^{*,2}$. We continue this process to make subsequent predictions.



[+1: Stating RNN, +1: Conditioning of the RNN based on the user, +1: Description of inputs: item embedding/encoding, +1: Per time step details/unfolding of the RNN, +2: Sketch and description of the approach, +2: Training versus testing details; Other correct and consistent answers accepted and marked accordingly]

Q. 4

---

a. [2 Points]

In Markov decision processes (MDPs) having infinite episode lengths, the returns obtained by the agent will consist of an infinite number of summands. Thus, it is possible that the returns, and therefore the associated values, at certain states/state-actions may diverge (blow up to infinities). In order to mitigate this problem, a discount factor γ $\left(0 \leq \gamma < 1\right)$ is often used while computing the returns in MDPs. Note that $\gamma = 1$ results in no discounting. With discounting, we multiply the reward k time steps in the future by γ^k, thereby effectively decreasing its contribution.

The intuition behind the discount factor is that an agent should weigh the rewards in the immediate future more than the the rewards in the very distant future. However,

more importantly, the discount factor ensures that the returns are bounded, which leads to consistent estimates of value functions.

Formally, suppose that all rewards are bounded, i.e, at any state s and action a, the corresponding reward r satisfies that $|r| \leq R$ for some fixed $R \geq 0$. Consider a discounted return expression— $G = R^t + \gamma \cdot R^{t+1} + \ldots + \gamma^k \cdot R^{t+k} + \ldots$

Then, $|G| = \left| \sum_{k \geq 0} \gamma^k \cdot R^{t+k} \right| \leq \sum_{k \geq 0} \left| \gamma^k \cdot R^{t+k} \right| = \sum_{k \geq 0} \gamma^k \cdot \left| R^{t+k} \right|$

$$\leq \sum_{k \geq 0} \gamma^k \cdot R = R \cdot \left(\sum_{k \geq 0} \gamma^k \right) = \frac{R}{1 - \gamma}$$

Thus, returns G are always well-defined and never diverge when discounted!

[2/2: Finiteness of returns; 1/2: Stating preference for immediate rewards]

---

b.

i. [5 Points]

The major goal of Monte-Carlo Exploring Starts algorithm is to **discover the optimal policy** by *i.* updating Q-values for state-action pairs based on sampling trajectories with the current policy, *ii.* updating the policy to the greedy policy based on the current estimate of the Q-values, and *iii.* repeating steps *i.* and *ii.* until convergence.

The “Monte-Carlo” part of the name refers to the use of sampling of trajectories in order to estimate the expectations in Q-values. The “Exploring Starts” part is related to ensuring that we visit all possible state-action pairs in order to have good estimate of the corresponding Q-values.

In this algorithm, we begin with a randomly initialized policy π and randomly initialized estimates of Q-values. We create lists corresponding to the returns obtained for all possible state-action pairs (s', a') and initialize them to empty lists. We start randomly at any state s and randomly choose action a from the state of actions that can be taken at s. Then, following the policy π, we sample a trajectory with this given initialization. We update the returns list for those state-action pairs that are visited in this trajectory by appending to these lists the returns for the first occurrence of the state-action pair. We then update the Q-values for all these state-action pairs as the average of all the returns in the corresponding returns list.

[+3: Discover optimal policy, +1: Learn Q for each possible (s, a), +1: Greedy update for the policy; -1: Impreciseness related to state vs. state-action values.]

ii. [3 Points]

Note that we must visit all possible states and take all possible actions in those states in order to have good estimates of the Q-values for all possible state-action pairs. The "Exploring Starts" part of the name of the algorithm refers to starting a particular trajectory by randomly sampling a state from all possible states and randomly taking any action from all possible actions at that state. This ensures that we visit all possible state-action pairs in our algorithm, thereby having good Q-value estimates for them.

However, for several reasons, this assumption can be unreasonable in many real-world examples. One of the problems is the difficulty in enumerating possible states or initializing at a randomly sampled state. For instance, in the game of chess, how do we know that a particular placement of the chess pieces is indeed legitimate and can be reached from the standard starting point? Thus, we need a mechanism to ensure that all possible state-action pairs can be reached without explicitly starting a trajectory in a particular state-action pair. This can be achieved by using an $\epsilon-$soft policy.

In our algorithm, we replace the greedy policy update with another policy that takes with high probability the optimal action but also takes with low probability a random non-optimal action. This ensures that we eventually reach all possible state-action pairs. Formally, we only need to change the greedy policy update with the $\epsilon-$soft policy update, which is described below.

For each state s appearing in the trajectory:

$$a* = \arg\max_{a \in A(s)} Q(s, a)$$

For all actions a that are possible in state s $\left(\text{i.e., } \forall a \in A(s)\right)$:

$$\pi\left(a \mid s\right) = 1 - \epsilon + \frac{\epsilon}{\left|A(s)\right|} \text{ if } a = a*$$

$$\pi\left(a \mid s\right) = \frac{\epsilon}{\left|A(s)\right|} \text{ if } a \neq a*$$

[+2: Explanation of ES, +1: $\epsilon-$Soft policy and its description]

iii. [10 Points]

We have 4 states: A, B, C, D and two actions: L, R. The policy π is initialized randomly and the values $Q(s, a)$ are initialized to 0 $\forall\ s, a$. For each state-action pair (s, a), let Return (s, a) denote the list of corresponding returns. Initially, these lists are empty.

After episode 1:

Return $(B, L) = \left[\left(5 + 5 + 2 + 5 + 0 = 17\right)\right]$

Return $(A, L) = \left[\left(5 + 2 + 5 + 0 = 12\right)\right]$

(Note that we must consider the return only for the first occurrence of the state-action pair (A, L))

Return $(C, L) = \left[\left(2 + 5 + 0 = 7\right)\right]$

Return $(C, R) = \left[(0 = 0)\right]$

The rest of the return lists remain unchanged.

Thus, the new Q-values are—

$Q(B, L) =$ average $\left(\text{Return}(B, L)\right) = 17$

$Q(A, L) =$ average $\left(\text{Return}(A, L)\right) = 12$

$Q(C, L) =$ average $\left(\text{Return}(C, L)\right) = 7$

$Q(C, R) =$ average $\left(\text{Return}(C, R)\right) = 0$

The rest of the state-action pairs did not occur in the episode and thus, their returns lists did not change. Thus, their Q-values remain unchanged at 0—

$Q(A, R) = Q(B, R) = Q(D, L) = Q(D, R) = 0$

Thus, the policy will update as follows—

$\pi(A) = \arg\max_{a \in \{L, R\}} Q(A, a) = L$ as $Q(A, L) = 12 > Q(A, R) = 0$

$\pi(B) = \arg\max_{a \in \{L, R\}} Q(B, a) = L$ as $Q(B, L) = 17 > Q(B, R) = 0$

$\pi(C) = \arg\max_{a \in \{L, R\}} Q(C, a) = L$ as $Q(C, L) = 7 > Q(C, R) = 0$

$\pi(D)$ remains unchanged to the action chosen in the random initialization

After episode 2:

Return $(C, R) = \Big[(0 = 0), \big(2 + 5 + 2 = 9\big)\Big]$

Return $(A, L) = \Big[\big(5 + 2 + 5 + 0 = 12\big), \big(5 + 2 = 7\big)\Big]$

Return $(C, L) = \Big[\big(2 + 5 + 0 = 7\big), (2 = 2)\Big]$

The rest of the return lists remain unchanged.

Thus, the new Q-values are—

$Q\,(B, L) = \text{average}\,\big(\text{Return}\,(B, L)\big) = 17$

$Q\,(A, L) = \text{average}\,\big(\text{Return}\,(A, L)\big) = \dfrac{12 + 7}{2} = 9.5$

$Q\,(C, L) = \text{average}\,\big(\text{Return}\,(C, L)\big) = \dfrac{7 + 2}{2} = 4.5$

$Q\,(C, R) = \text{average}\,\big(\text{Return}\,(C, R)\big) = \dfrac{0 + 9}{2} = 4.5$

The rest of the state-action pairs did not occur in the the second episode as well and thus, their returns lists did not change. Thus, their Q-values remain unchanged at 0—

$Q\,(A, R) = Q\,(B, R) = Q\,(D, L) = Q\,(D, R) = 0$

Thus, the policy will update as follows—

$\pi\,(A) = \arg\max_{a \in \{L, R\}} Q\,(A, a) = L$ as $Q(A, L) = 9.5 > Q(A, R) = 0$

$\pi\,(B) = \arg\max_{a \in \{L, R\}} Q\,(B, a) = L$ as $Q(B, L) = 17 > Q(B, R) = 0$

$\pi\,(C) = \arg\max_{a \in \{L, R\}} Q\,(C, a) = \text{random-choice}\,\{L, R\}$ as $Q(C, L) = Q(C, R) = 4.5$ and the ties are broken randomly

$\pi\,(D)$ remains unchanged to the action chosen in the random initialization

[+4: Correct calculation of each of the 4 $Q(s, a)$ values that update, +1: Correct calculation of policy, +1: Stating the rest of $Q(s, a)$ values remain at 0 and that $\pi\,(D)$ remains unchanged at random initialization, +4: Calculations (3/4 if minor calculation/conceptual error, 2/4 if major conceptual error, 0/4 if no justification)]

---

c.

Note that the (autonomous) driving car agent is significantly more complex than a simple gridworld agent and thus, the former should not be reduced to the latter!

i. [2 Points]

Note that a car needs to decide at every time step whether to accelerate or decelerate (brake) as well as the direction in which to go. The accelerator being pressed or not and the breaks being applied or not helps capture the acceleration/deceleration. Further, the direction of motion of the car can be captured by the steering angle. Thus, in a first-level model, we can define the tuple (a, b, θ) as the action, where a represents if the accelerator is pressed, b represents whether the brakes are applied, and θ represents the steering angle.

Note that this is a simplified model because in a real car, we must also incorporate the gears and their contribution to the change of the state of the car.

[+2: Any correct and sufficient description of actions; 1/2: If mapped to gridworld]

ii. [4 Points]

A state captures all the relevant information about the agent and the environment and uniquely characterizes them at a particular time step. Thus, it is important to notice that the state must contain the information about the internal variables related to the car as well as the external factors.

The internal variables associated with the car may include the amount of fuel left in the car f, the velocity of the car v, the relative/absolute location x of the car in a convenient coordinate system, etc.

The external variables may include the locations $\left(c^i\right)_i$ of other cars with respect to our car, the different objects (e.g., traffic signs, pedestrians, etc.) in the environment of the car along with their locations $\left(o^j\right)_j$ and so on.

Thus, the state is a tuple that would capture all the relevant factors described above. For instance, in our example, the state would be defined as– $\left(f, v, x, \left(c^i\right)_i, \left(o^j\right)_j\right)$.

[+2: Internal factors, +2: External factors; 3/4: Insufficient description mentioning both internal and external factors; 2/4: Correct answer mapped to gridworld; 1/4: Insufficient description mapped to gridworld; 0/4: Insufficient description]

iii. [3 Points]

Note that the reward is a scalar that is maximized by the agent in order to achieve its goal. Further, the reward is, in principle, a function of the state and action; i.e., for any state s, taking an action a in it leads to a reward of $r(s, a)$ for the agent.

Thus, when we need to define the reward in our case, we ought to define a scalar-valued function for each possible combination of state and action. This is a challenging task in general but nonetheless, we can broadly describe how the reward should look like in different scenarios of interest. Intuitively, there are a lot of factors that would contribute to the design of reward and they can be described based on what we want our agent to learn.

For instance, if our car takes an action that keeps it driving correctly on the road in a specific lane, we give our car a *small* positive reward of, say, 1. If the car is traveling on a road and takes an action that leads it to break a traffic rule (e.g., over-speeding, changing lanes without indicators, etc.), we give a *medium* negative reward of, say, -10. If the car takes an action and then hits another car or a pedestrian, we give a *large* negative reward of, say, -500. If the car correctly reaches the target destination, we give a *large* positive reward of, say, 1000.

Note that the choice of values will result in reward-shaping, which is an important factor in defining the reinforcement learning problem.

[3/3: Example of scalar valued rewards and description; 2/3: Explanation of the factors contributing to the reward but missing that the reward is a scalar; 1/3: Insufficient description; Any reasonable reward description is accepted]