Name: ________________________________ NetID: ____________

S&DS 365 / 665

Intermediate Machine Learning

Final Exam (Practice)

Saturday, May 7, 2022

Complete all of the problems. You have 2.5 hours (150 minutes) to complete the exam.

The exam is closed book, computer, phone, etc. You are allowed one double-sided $8\frac{1}{2} \times 11$ sheet of paper with hand-written notes.

1. ***Multinomial choice*** (20 points)

For each of the following questions, circle the *single best* answer.

1.1. To carry out Mercer kernel regression using training data X and Y to predict for a new set X', suppose that $\mathbb{K}_{ij} = K(X_i, X_j)$ and $\mathbb{K}'_{ij} = K(X'_i, X_j)$. Then the estimated regression function $\widehat{m}(X')$ is given by

(a) $(\mathbb{K} + \lambda I)^{-1} Y$

(b) $\mathbb{K}(\mathbb{K} + \lambda I)^{-1} Y$

$\boxed{\text{(c)}}$ $\mathbb{K}'(\mathbb{K} + \lambda I)^{-1} Y$

(d) $(\mathbb{K} + \lambda I)^{-1} Y \mathbb{K}'$

1.2. The main reason the ℓ_1 norm is used as a penalization term in the lasso is that

(a) the coefficients of a random set of variables are set to zero

$\boxed{\text{(b)}}$ the ℓ_1 norm leads to sparsity and convexity at the same time

(c) the ℓ_1 norm is better than the ℓ_2 norm

(d) the estimated coefficients will have less bias

1.3. Suppose that we have a kernel regression technique in one dimension with bandwidth parameter h for which the squared bias scales as $O(h^3)$ and the variance scales as $O\left(\frac{1}{nh}\right)$ as $h \to 0$ with $nh \to \infty$, for a sample of size n, under certain assumptions. If h scales as $h \approx n^{-1/3}$, what is the rate at which the risk (expected squared error) will decrease with sample size for this technique?

(a) $O(n^{-1/3})$

(b) $O(n^{-1/4})$

$\boxed{\text{(c)}}$ $O(n^{-2/3})$

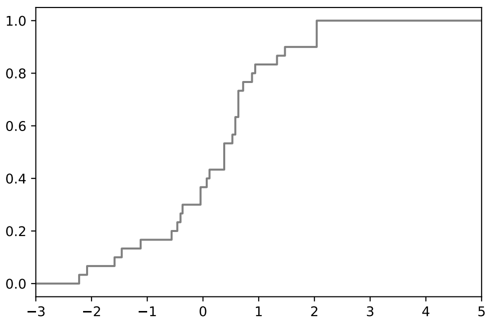
(d) $O(n^{-1})$

1.4. Consider the following code for sampling data $X_1, \dots, X_{\text{size}}$ from the marginal of the posterior of a Dirichlet process with prior $DP(\alpha, F_0)$ with $F_0 = N(\mu_0, \tau_0^2)$.

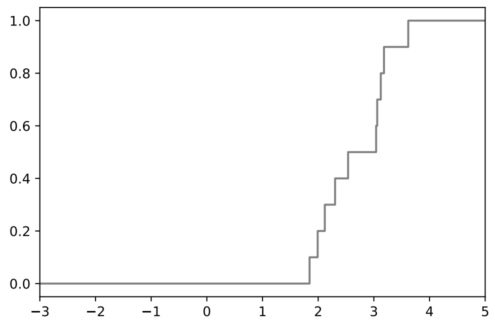
```
import numpy as np
from scipy.stats import norm

def sample_X_from_posterior_DP(X=[], size=10, alpha=1, mu0=3, tau0=1):
    X_sample = []
    for i in np.arange(size):
        m = i + len(X)
        z = np.random.choice([0,1], p=[alpha/(alpha+m), m/(alpha+m)])
        if z==0:
            x = norm.rvs(loc=mu0, scale=tau0)
        else:
            x = np.random.choice(X + X_sample)
        X_sample.append(x)

    return(X_sample)

X = norm.rvs(size=n, loc=0)
sample = sample_X_from_posterior_DP(X=X, alpha=alpha, size=size)
plot_empirical_cdf(sample)
```

Which of the following is most likely generated with `size=30, alpha=100, n=1`?

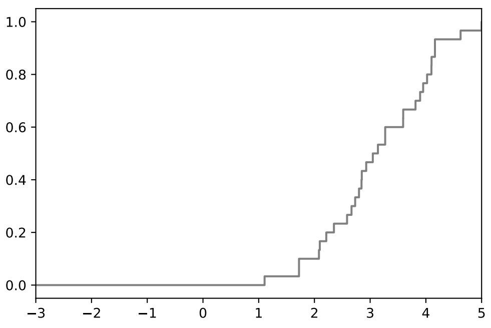
(a)



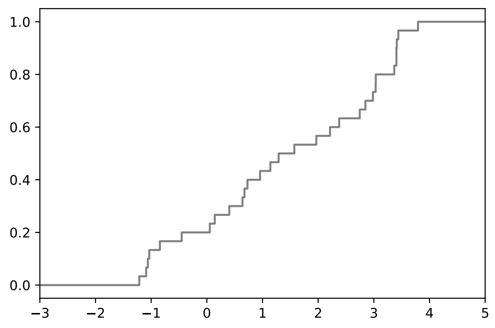
(b)



[✓ (c)]



(d)



1.5. Suppose F is drawn from a Dirichlet process $DP(\alpha, F_0)$ where $\alpha = 2$ and F_0 is a discrete uniform distribution on the 6 values $\{1, 2, 3, 4, 5, 6\}$, with equal probability $\frac{1}{6}$ for each (a fair 6-sided die).

We draw two values $X_1 = 1$ and $X_2 = 3$ from F. What is the probability that the next draw X_3 is also 3?

(a) $1/2$

(b) $2/3$

$\boxed{\text{(c)}}$ $1/3$

(d) 1

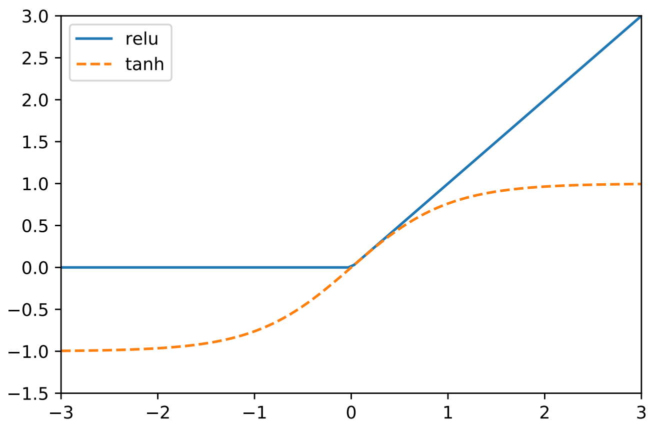
1.6 Consider a neural network for binary classification with 2-dimensional input $x = (x_1, x_2)^T$ defined by

$$\log\left(\frac{p(Y=1\,|\,x)}{p(Y=0\,|\,x)}\right) = \beta^T h(x)$$

with $h(x) = \varphi(Wx)$ where φ is an activation function and

$$W = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \beta = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Recall the relu and tanh activation functions look like this, with $\tanh(-x) = -\tanh(x)$:



.

[✓ TRUE] FALSE (1) If φ is relu and $x_1 = x_2$ then $p(Y=1\,|\,x) = \frac{1}{2}$.

[✓ TRUE] FALSE (2) If φ is tanh and $x_1 = x_2$ then $p(Y=1\,|\,x) = \frac{1}{2}$.

[✓ TRUE] FALSE (3) If φ is relu and $x = (2,1)^T$ then $p(Y=1\,|\,x) > \frac{1}{2}$.

[✓ TRUE] FALSE (4) If φ is tanh and $x = (2,1)^T$ then $p(Y=1\,|\,x) > \frac{1}{2}$.

[✓ TRUE] FALSE (5) The models with tanh and relu activations have the same decision boundaries.

1.7 The following questions concern recurrent neural networks (RNNs).

[✓ TRUE] FALSE (1) RNNs are used to model sequential data.

[✓ TRUE] FALSE (2) The number of parameters in an RNN with a single hidden layer is bounded by $c_1 H^2 + c_2 V H$ as a function of the number of hidden neurons H, where c_1 and c_2 are constants and V is the size of the input vocabulary.

[✓ TRUE] FALSE (3) Training an RNN uses a loss function that encourages the model to assign high probability to the training data.

[✓ TRUE] FALSE (4) The hidden neurons are deterministic functions of the input data.

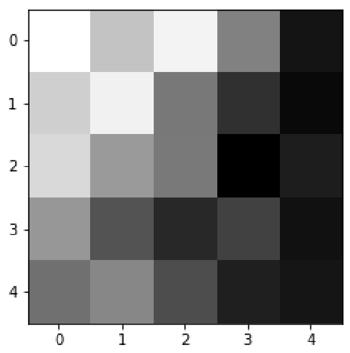
[✓ TRUE] FALSE (5) The network structure—including the number of hidden layers, the number of neurons and the choice of activation function—can be chosen using cross-validation.

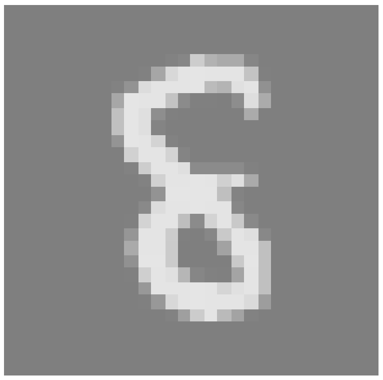
2. ***Convolutional neural networks*** (10 points)

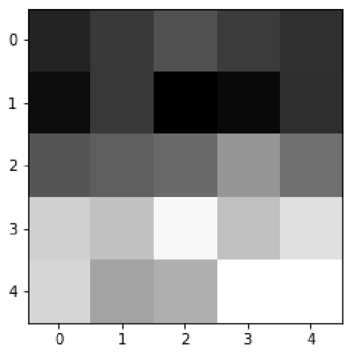
   Convolutional neural networks (CNNs) work by learning a set of kernel functions or "filters" that are swept across an image to create a "feature map." Some of the filters can be difficult to interpret; others are more interpretable.

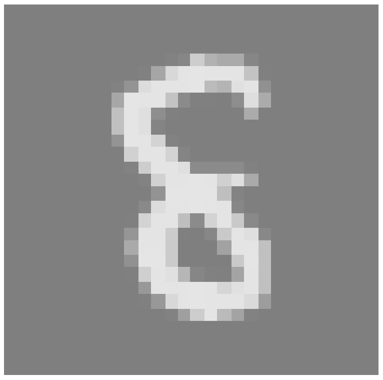
   (a) For each of the three 5×5 filters below, shade the regions with high values in their corresponding feature maps on the original images of the digit 8. Briefly describe your reasoning.

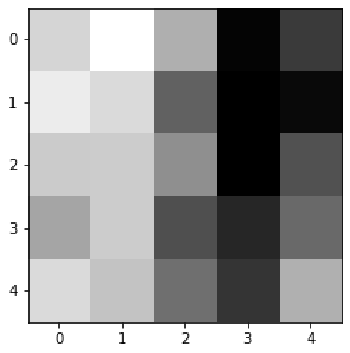
   For the original image, pixels in white have high values while pixels in black have zero values. For the filters, pixels in white have positive values while pixels in black have negative values.

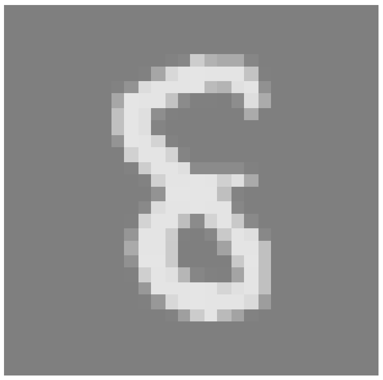

0
1
2
3
4
0
1
2
3
4




0
1
2
3
4
0
1
2
3
4




0
1
2
3
4
0
1
2
3
4



The response to the first filter is large at locations where the surrounding pixels roughly form an edge in the “northeast” direction, with white (large) values toward the upper left corner, and dark (small) values toward the lower right corner. For example, along the lower edge of the top of the 8.

The response to the second filter is large at locations where the surrounding pixels roughly form a horizontal edge, with white (large) values below, and dark (small) values above. For example, on the top of the bottom part of the lower loop of the 8.

The response to the third filter is large at locations where the surrounding pixels roughly form a vertical edge, with white (large) values to the left, and dark (small) values to the right. For example, on the right side of the lower loop of the 8.

(b) Suppose we were to have a convolutional layer constructed as

```
model.add(layers.Conv2D(24, (4, 4), input_shape=(20, 20, 32)))
```

What is the shape of each of the filters?

$(4, 4, 32)$

(c) What is the total number of trainable parameters for this layer? Show your work.

$24 \cdot (4 \cdot 4 \cdot 32 + 1) = 12{,}312$

3. ***Short answer*** (10 points)

   The following two subproblems ask you to explain the important concepts associated with two topics that have been central to the first part of the course.

   (a) ***Kernel methods***. Suppose you are given a dataset $\{(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)\} \subseteq \mathbb{R}^p \times \mathbb{R}$. Answer the following two questions. (1) What are two ways to use kernel methods to solve the regression problem, of predicting Y from X? (2) How can you use kernel methods to estimate the marginal distribution of X?

   (1) Kernel regression and Gaussian processes
   (2) Apply kernel density estimation to $X_1, \ldots, X_n$.

(b) ***Graph neural networks***. (1) Define the graph Laplacian for a weighted graph. (2) Explain how the Laplacian can be used to define the analogue of a convolutional neural network (CNN) to classify vectors assigning a value to each node in the graph. Be explicit in how the analogue of CNN kernels are defined, and how the discriminant function is constructed.

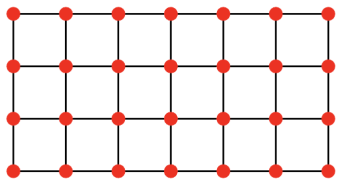
(1) $L = D - W$ where $W = [W_{ij}]$ is the matrix of edge weights and $D = \text{diag}(d_1, \ldots, d_p)$ is a diagonal matrix with $d_i = \sum_j W_{ij}$, for each of the p nodes in the graph.

(2) Construct kernels that are $p \times p$ matrices $K_w = \sum_{i=0}^{d} w_i L^i$ where w are weights and L is the Laplacian. If we have k kernels this gives a mapping $x \to h(x) \in \mathbb{R}^{kp}$ with $(d+1)k$ tunable parameters. This is the first layer in the net, analogous to a convolutional layer. The second, dense layer has parameters $\beta \in \mathbb{R}^{kp}$. Given a set of data $\{(X_i, y_i)\}$ where $X_i \in \mathbb{R}^p$ assigns a value to each node in the graph, this network is trained using stochastic gradient descent over the parameters w and β.

4. ***Gibbs sampling*** (10 points)

The Ising model, as discussed in class, is a model that is central to physics, social network analysis, and many other areas.

For this problem, we will consider a grid graph, such as the one shown below, where each node is connected to the nodes immediately above and below it, and to its left and right:



Each node position (x, y) in the graph, $Z_{(x,y)} \in \{-1, 1\}$ is a random variable that is either 1 or -1. The the joint probability takes the form

$$p(Z) \;\propto\; \exp\left(\sum_{((x,y),(x',y')) \in E} \beta Z_{(x,y)} Z_{(x',y')} \right)$$

where β is a scalar parameter, assigning a weight β to each edge in the grid graph.

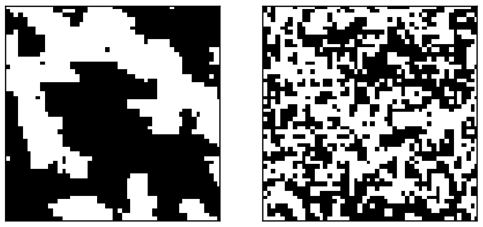
Recall that the Gibbs sampling algorithm draws from this distribution by repeatedly visiting nodes (x, y), and sampling $Z_{(x,y)}$ while holding all of the other $Z_{(x',y')}$ values fixed.

The following code partially implements Gibbs sampling for this model. Your job is to complete the implementation by providing two additional lines of code.

```
import numpy as np

def gibbs_step(Z, beta, x, y):
    exponent = 0
    for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
        if ((x + dx < 0) | (x + dx >= Z.shape[0]) |
            (y + dy < 0) | (y + dy >= Z.shape[1])):
            continue
        # finish the loop with one line
        # your first line of code here

    # set the probability p
    # your second line of code here

    # np.random.rand() draws a random uniform variable
    if np.random.rand() <= p:
        Z_new = 1
    else:
        Z_new = -1
    return Z_new

def run_gibbs_sampling(Z, beta, steps=10):
   for step in np.arange(steps):
       for x in np.arange(Z.shape[0]):
           for y in np.arange(Z.shape[1]):
               Z[x, y] = gibbs_step(Z, beta, x, y)
```

(a) Complete the implementation, by writing the two missing lines below.

```
exponent += beta * Z[x+dx, y+dy]
p = 1/(1+np.exp(-2*exponent))
```



(b) The figure above shows the result of running this algorithm on a 50×50 grid, for two different parameter values, $\beta = \frac{1}{4}$ and $\beta = \frac{3}{4}$. Which is which? Explain your answer.

$\beta = \frac{3}{4}$ is on the left, and $\beta = \frac{1}{4}$ is on the right.

The larger β is, the more likely it is that neighboring nodes i and j will have the same values of Z_i and Z_j.

(c) Now suppose that we are running a Gibbs sampling step on a node (x, y) that has four neighbors, with

$$\begin{aligned} Z[x-1, y] &= Z[x+1, y] = 1 \\ Z[x, y-1] &= Z[x, y+1] = -1. \end{aligned}$$

In this Gibbs sampling step, what is the probability that $Z[x, y]$ is set to 1?

(1) $1/(1+e^{2\beta})$

(2) $1/(1+e^{-2\beta})$

(3) $1/4$

$\boxed{(4)}$ $1/2$

5. ***Reinforcement learning*** (10 points)

The following three subproblems are on the topic of reinforcement learning.

(5.1) What is the Bellman equation for the Q-function? Write your answer assuming that the environment is stochastic. Briefly explain the meaning of the equation.

$$\begin{aligned} Q_*(s,a) &= \sum_{s',r} p(s',r \mid s,a)\Big\{r + \gamma \max_{a'} Q_*(s',a')\Big\} \\ &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1},a') \;\mid\; S_t = s, A_t = a\Big] \end{aligned}$$

$Q_*(s,a)$ is the expected long-term reward when taking action a in state s. The Bellman equation is a consistency equation, a necessary condition for an optimal Q-function.

In class we discussed the Q-learning algorithm, which has the following pseudocode:

```
$Q$-Learning

Parameters: step size $\alpha$, exploration probability $\varepsilon$, discount factor $\gamma$
Initialize $Q(s, a)$ arbitrarily, except $Q(\text{terminal}, \cdot) = 0$
**Loop** for each episode:
    Initialize $S$
    **Loop** for each step of episode:
        Choose action $A$ from $S$ using $Q$ with $\varepsilon$-greedy policy
        Take action $A$; observe reward $R$ and new state $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_a Q(S', a) - Q(S, A)\right)$
        $S \leftarrow S'$
    **Until** $S$ is terminal
```

Our implementation of this algorithm on the Taxi problem was the following:

```
for _ in np.arange(episodes):
  state = env.reset()
  done = False
  while not done:
    if random.uniform(0, 1) < epsilon:
      action = env.action_space.sample()
    else:
      action = np.argmax(q_table[state])

    next_state, reward, done, _ = env.step(action)

    old_value = q_table[state, action]
    next_max = np.max(q_table[next_state])
    new_value = old_value + alpha*(reward + gamma*next_max - old_value)
    q_table[state, action] = new_value

    state = next_state
```

An algorithm that is closely related to Q-learning is called "Sarsa" and has the following pseudo-code:

```
Sarsa

Parameters: step size $\alpha$, exploration probability $\varepsilon$, discount factor $\gamma$
Initialize $Q(s, a)$ arbitrarily, except $Q(\text{terminal}, \cdot) = 0$
Loop for each episode:
    Initialize $S$
    Choose action $A$ from $S$ using $Q$ with $\varepsilon$-greedy policy
    Loop for each step of episode:
        Take action $A$; observe reward $R$ and new state $S'$
        Choose action $A'$ from $S'$ using $Q$ with $\varepsilon$-greedy policy
        $Q(S, A) \leftarrow Q(S, A) + \alpha\,(R + \gamma Q(S', A') - Q(S, A))$
        $S \leftarrow S'$; $A \leftarrow A'$
    Until $S$ is terminal
```

(5.2) (1) What is the main difference between Sarsa and Q-learning? (2) Would you expect them to perform differently or the same for the Taxi problem? (3) Could "Deep Sarsa Learning" be used in a way that is analogous to Deep Q-Learning when the state space is large? Explain.

(1) Sarsa is an "on-policy" method because the ε-greedy policy using the Q-function is used to evaluate the error. In Q-learning, the value function (max of the Q function) is used to evaluate the error.

(2) Sarsa can have trouble converging if ε is too large, because the objective function is then random and will have high variance. With careful tuning of the parameters though, it will converge and perform the same as Q-learning.

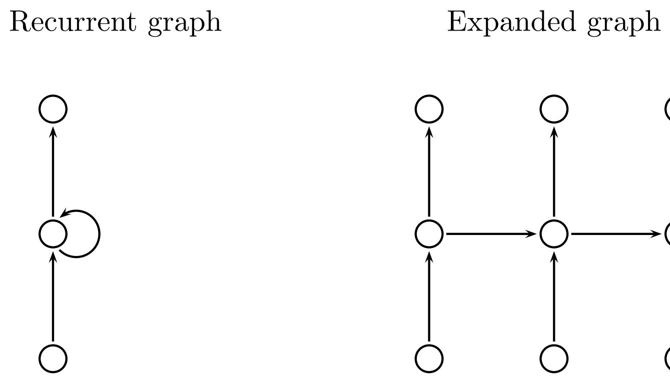
(3) Yes, the same procedure as for deep Q-learning can be used, but where the objective function uses the ε-greedy policy to compute the target.

(5.3) Give a Python implementation of the Sarsa algorithm on the Taxi problem.

```
def get_action(state):
  if random.uniform(0, 1) < epsilon:
     action = env.action_space.sample()
  else:
     action = np.argmax(s_table[state])
  return action


for _ in tqdm(np.arange(episodes)):
  state = env.reset()
  done = False
  action = get_action(state)

  while not done:
    next_state, reward, done, _ = env.step(action)
    next_action = get_action(next_state)

    old_value = s_table[state, action]
    next_value = s_table[next_state, next_action]
    new_value = old_value + alpha*(reward + gamma*next_value - old_value)
    s_table[state, action] = new_value

    state = next_state
    action = next_action
```

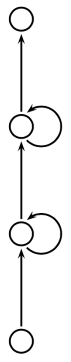
6. ***Recurrent neural networks*** (5 points)

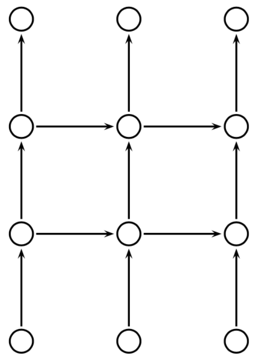
When we talked about recurrent neural networks in class, we used “expanded graphs” like the one on the right below. This shows three steps of an RNN with one hidden layer. The bottom nodes are the inputs, the middle nodes are the hidden neurons, and the top nodes are the outputs.



This can also be represented with a “recurrent graph” as shown on the left, which is not a feedforward graph. In this example, the self-connection between the hidden neuron means that it depends on itself at the previous time. The two graphs show the same information in different ways.

(a) For the expanded graph below, draw the corresponding recurrent graph.





(b) In this last problem we ask you to consider how you might approach a new machine learning problem.

Consider a “Tweet embedding” problem to map posts on Twitter to a high dimensional vector. That is, suppose we have a large database of tweets—short text strings of up to 280 characters—and wish to map each tweet to a vector so that similar tweets are mapped to nearby points in the embedding space.

Describe how you would approach this problem *using recurrent neural networks*.

There are many ways of doing this. One is to train an RNN on a database of tweets, then let the embedding be the state h_T where T is the number of charaters in the tweet, or let $h = \sum_{t=1}^{T} h_t$.
Another is to train two RNNs, one going left-to-right, the other going right-to-left. Let the embedding be $h = h_{T/2} + h'_{T/2}$ where $h_{T/2}$ is the state halfway through the left-to-right RNN and $h'_{T/2}$ is the state halfway through the right-to-left RNN.

Extra work space

Extra work space