EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Review: Autoencoder & RNN

1. Autoencoders

State whether each of the statements below is True or False and explain why. If the type of autoencoder is not specified, you may consider all autoencoder types (vanilla, denoising, masked, etc.)

(a) There is no point in checking autoencoder reconstruction performance on a validation set because we will ultimately evaluate whether representations are useful by training on downstream tasks.

(b) If you train two different autoencoder variants on the same dataset, the one which produces lower validation loss will perform better on the downstream task.

(c) The autoencoder decoder is not used after pretraining.

(d) Using autoencoder representations can can sometimes produce worse performance on a downstream task than using raw inputs.

(e) Autoencoder representations can be useful even if the representation was trained on a very different dataset than the downstream task.

(f) Using an autoencoder representation (rather than using raw inputs) is most useful when you have few labels for your downstream task.

(g) With images, it is often more effective to mask patches than to mask individual pixels.

(h) We can think of masked and denoising autoencoders as vanilla autoencoders with data augmentation applied.

(i) If you trained an autoencoder with noise or masking, you should also apply noise/masking to inputs when using the representations for downstream tasks.

(j) Autoencoders always encode inputs into fixed-size lower-dimensional representations.

2. RNN Recap

A vanilla RNN layer implements the function

$$h_t = \sigma(W^h h_{t-1} + W^x x_t + b)$$

where W^h, W^x, and b are learned parameter matrices, x is the input sequence, and σ is a nonlinearity such as tanh. The RNN layer "unrolls" across a sequence, passing a hidden state between timesteps and returning an array of hidden states at all timesteps.

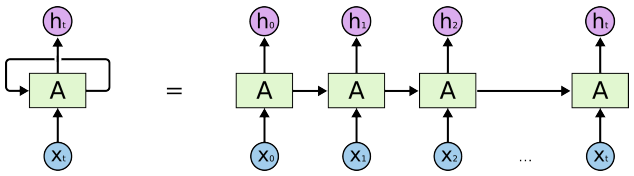
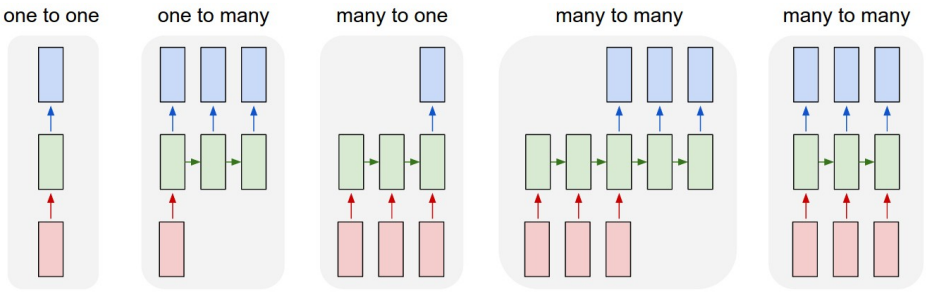


Figure 1: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

For output, you will use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

$$\hat{y}_t = W^f h_t + b^f$$

We'll compute one prediction for each timestep. RNNs can be used for many kinds of prediction problems, as shown below.



We here consider a simple averaging task. The input X consists of a sequence of numbers, and the label y is a running average of all numbers seen so far.

We will consider two tasks with this dataset:

- Task 1: predict the running average at all timesteps
- Task 2: predict the average at the last timestep only

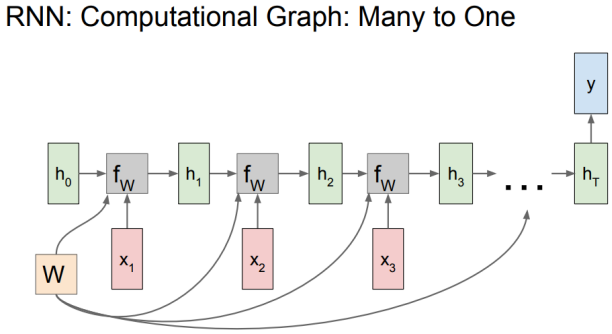


Figure 2: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(a) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 2, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{1}$$

When you backpropagate a loss through many timesteps, the later terms in this sum often end up with either very small or very large magnitude - called vanishing or exploding gradients respectively. Either problem can make learning with long sequences difficult.

In the original Notebook Section 1.D, it plots the magnitude at each timestep of $\frac{\partial \mathcal{L}}{\partial h_t}$. Play around with this visualization tool and try to generate exploding and vanishing gradients.

If the network has no nonlinearities, under what conditions would you expect the exploding or vanishing gradients with for long sequences? Why? (Hint: it might be helpful to write out the formula for $\frac{\partial \mathcal{L}}{\partial h_t}$ and analyze how this changes with different t). **Do you see this pattern empirically using the visualization tool in Section 1.D in the notebook** with last_step_only=True?

(b) Compare the magnitude of hidden states and gradients when using ReLU and tanh nonlinearities in Section 1.D in the notebook. **Which activation results in more vanishing and exploding gradients? Why?** (This does not have to be a rigorous mathematical explanation.)

(c) **What happens if you set last_target_only = False in Section 1.D in the notebook? Explain why this change affects vanishing gradients. Does it help the network's ability to learn dependencies across long sequences?** (The explanation can be intuitive, not mathematically rigorous.)

3. Beam Search

When making predictions with an autoregressive sequence model, it can be intractable to decode the true most likely sequence of the model, as doing so would require exhaustively searching the tree of all $O(M^T)$ possible sequences, where M is the size of our vocabulary, and T is the max length of a sequence. We could decode our sequence by greedily decoding the most likely token each timestep, and this can work to some extent, but there are no guarantees that this sequence is the actual most likely sequence of our model.

Instead, we can use beam search to limit our search to only candidate sequences that are the most likely so far. In beam search, we keep track of the k most likely predictions of our model so far. At each timestep, we expand our predictions to all of the possible expansions of these sequences after one token, and then we keep only the top k of the most likely sequences out of these. In the end, we return the most likely sequence out of our final candidate sequences. This is also not guaranteed to be the true most likely sequence, but it is usually better than the result of just greedy decoding.

The beam search procedure can be written as the following pseudocode:

Algorithm 1 Beam Search

for each time step t **do**
  for each hypothesis $y_{1:t-1,i}$ that we are tracking **do**
    find the top k tokens $y_{t,i,1}$,...,$y_{t,i,k}$
  end for
  sort the resulting k^2 length t sequences by their total log-probability
  store the top k
  advance each hypothesis to time $t+1$
end for

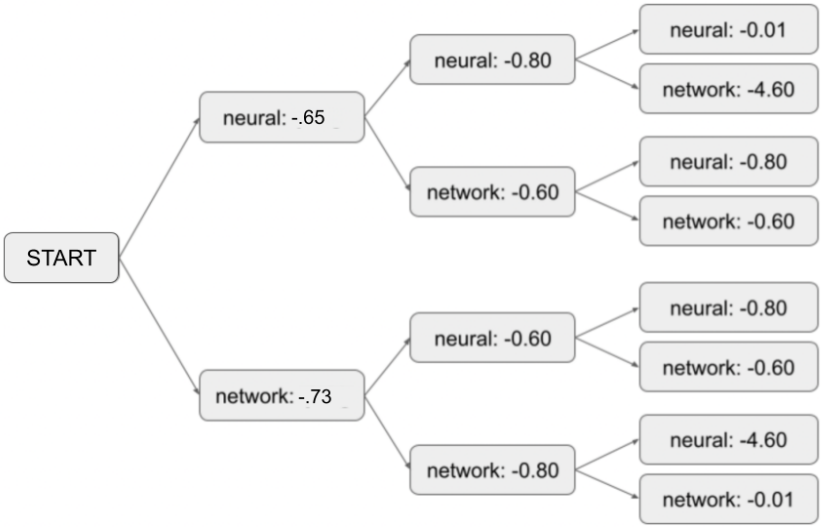


Figure 3: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

We are running the beam search to decode a sequence of length 3 using a beam search with $k = 2$. Consider predictions of a decoder in Figure 3, where each node in the tree represents the next token **log probability** prediction of one step of the decoder conditioned on previous tokens. The vocabulary consists of two words: "neural" and "network".

(a) **At timestep 1, which sequences is beam search storing?**

(b) **At timestep 2, which sequences is beam search storing?**

(c) **At timestep 3, which sequences is beam search storing?**

(d) **Does beam search return the overall most-likely sequence in this example? Explain why or why not.**

(e) **What is the runtime complexity of generating a length-T sequence with beam size k with an RNN?** Answer in terms of T and k and M. (Note: an earlier version of this question said to write it in terms of just T and k. This answer is also acceptable.)

Contributors:

- Olivia Watkins.
- CS 182 Staff from past semesters.
- Kumar Krishna Agrawal.
- Dhruv Shah.

