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.

Solution: False. Checking performance on a validation set is still useful for sanity checking that the model is training correctly (val loss should go down), diagnosing overfitting, and early stopping.

(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.

Solution: This is not always true. For instance, if you add noise or masking or make the bottleneck smaller, reconstruction loss may go up, but these could still produce more useful representations.

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

Solution: True, we only use the encoder for downstream training.

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

Solution: True. This may occur, for instance, if the pretraining dataset was too different from the task dataset, or if the autoencoder bottleneck is small enough that it throws away features which are important for that downstream task.

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

Solution: True, often autoencoders are trained on large, diverse datasets. If these pretraining datasets are diverse enough that they are a superset of the task data, or at least are diverse enough that they learn some general patterns which are relevant to the task data, the AE representation may be useful.

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

Solution: True: Unsupervised pretraining provides the largest gains when you have little task-specific data (and are therefore at risk of overfitting), though it can still help somewhat even when plenty of data is available.

(g) With images, it is often more effective to mask patches than to mask individual pixels. **Solution:** True. If you masked pixels, the model could get decent loss by just copying neighboring pixels. This representation would not learn much about the overall image structure

(h) We can think of masked and denoising autoencoders as vanilla autoencoders with data augmentation applied. **Solution:** True: like other data augmentations, these can help prevent overfitting and produce more robust representations. Masking, however, sometimes involves additional changes to the training procedure which are not used with other augmentations (e.g. some transformer architectures completely remove masked tokens from the encoder.)

(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. **Solution:** False: this is uncommon (though it is done occasionally when you want data augmentation on the downstream task). Like many regularizers (like data augmentation and dropout), the modification is applied during training only.

(j) Autoencoders always encode inputs into fixed-size lower-dimensional representations. **Solution:** False. Masked and denoising autoencoder representations may not be lower-dimensional, and representations can be variable-sized (e.g. a transformer's representation contains one vector for each input token.

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.

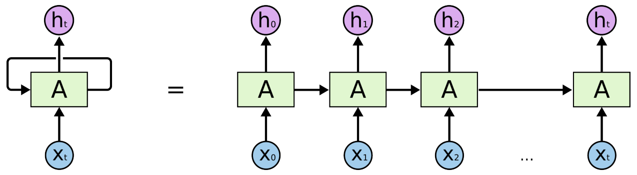
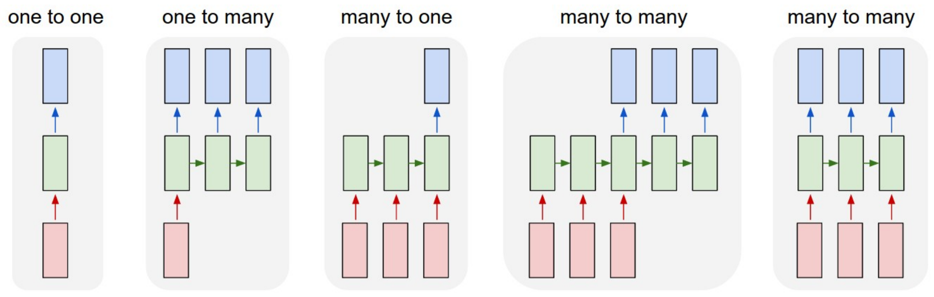


Figure 1: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

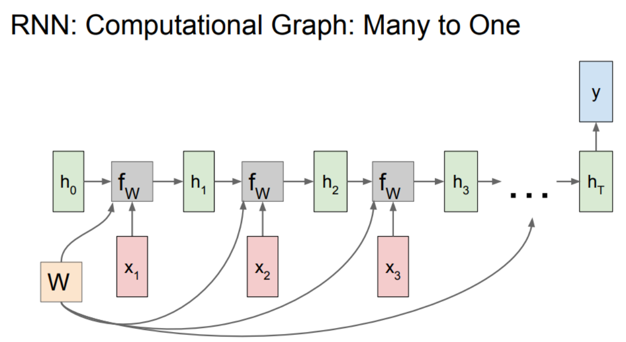


Figure 2: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(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?

Solution: If we use the MSE loss on a single example (x, y), the gradient $\frac{\partial \mathcal{L}}{\partial h_t} = 2(\hat{y} - y)W^f(W^h)^{T-t}$. (To clarify, the exponents f and h are matrix indicators, but $t - i$ is an exponent.) If the magnitude of the largest eigenvalue of W^h is much greater than 1, the gradient will explode, and if it's much less than 1, the gradient will start to vanish. Gradients are only stable when the largest eigenvalue magnitude is close to 1.

We see the expected pattern empirically.

(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.)

Solution: Hidden states: tanh restricts hidden state values to (-1, 1), so hidden state magnitudes remain small. With ReLU, hidden state values can easily explode.

Gradients: When tanh inputs are large, gradients are close to zero. This results in fewer exploding gradients but more vanishing gradients. Exploding gradients are still possible, however, when the largest eigenvalue of W_h has magnitude > 1, but the hidden states remain close to zero. ReLU activations, in contrast, result in frequent exploding hidden state sizes and gradients, similar to in the no-activation RNN.

(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.)

Solution: For every timestep k, the model's prediction produces loss $\mathcal{L}_k$. The gradient $\partial L_k/\partial h_k$ is high-magnitude, resulting in high-magnitude logged gradients in the visualization tool, but for any timestep $t << k$, $\partial L_k/\partial h_t$ will still be small. This means the network will still struggle to pass gradients over long sequences, making it hard to learn long-range dependencies.

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
```

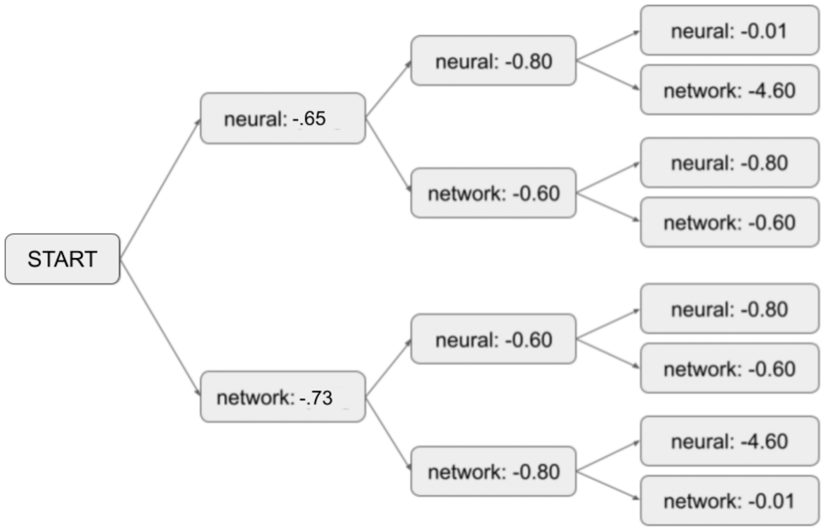


Figure 3: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

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?**

Solution: There are only two options, so our beam search keeps them both: “neural” (log prob = -.65) and “network” (log prob = -.73).

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

Solution: We consider all possible two word sequences, but we then keep only the top two, “neural network” (with log prob = -.65 - .6 = -1.25) and “network neural” (with log prob = -.73 - .6 = -1.33).

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

Solution: We consider three word sequences that start with “neural network” and “network neural”, and the top two are “neural network network” (with log prob = -.65 - .6 - .6 = -1.85) and “network neural network” (with log prob = -.73 - .6 - .6= -1.93).

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

Solution: No, the overall most-likely sequence is “neural neural neural” with log prob = -.65 - .8 - .01= -1.46). These sequences don’t get returned since they get eliminated from consideration in step 2, since “neural neural” is not in the $k = 2$ most likely length-2 sequences.

(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.)

Solution:

- Step RNN forward one step for one hypothesis = $O(M)$ (since we compute one logit for each vocab item, and none of the other RNN operations rely on M, T, or K).
- Do the above, and select the top k tokens for one hypothesis. We do this by sorting the logits: $O(M \log M)$. (Note: there are more efficient ways to select the top K, for instance using a min heap. We just use this way since the code implementation is simple.) Combined with the previous step, this is $O(M \log M + M) = O(M \log M)$.
- Do the above for all k current hypotheses $O(KM \log M)$.
- Do all above + choose the top K of the K^2 hypotheses currently stored: we do this by sorting the array of K^2 items: $O(K^2 \log(K^2)) = O(K^2 \log(K))$ (since $\log(K^2) = 2\log(K)$). (Note: there are also more efficient ways to do this). Combining this with the previous steps, we get $O(KM \log M + K^2 \log(K))$. When one term is strictly larger than another we can take the max of the two. Since $M \geq K$, we could also write this as $O(KM \log M)$.
- Repeat this for T timesteps: $O(TKM \log M)$.

Contributors:

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