EECS 182 Deep Neural Networks — Fall 2023 Anant Sahai — Homework 8

This homework is due on October 21, at 10:59PM.

1. Backprop through a Simple RNN

Consider the following 1D RNN with no nonlinearities, a 1D hidden state, and 1D inputs u_t at each timestep. (Note: There is only a single parameter w, no bias). This RNN expresses unrolling the following recurrence relation, with hidden state h_t at unrolling step t given by:

$$h_t = w \cdot (u_t + h_{t-1}) \tag{1}$$

The computational graph of unrolling the RNN for three timesteps is shown below:

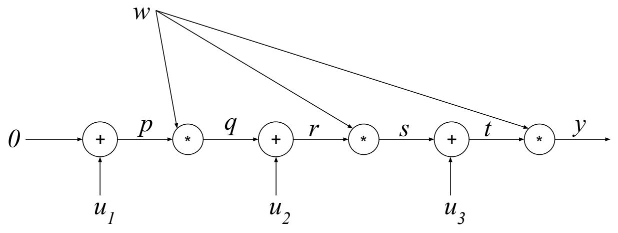


Figure 1: Illustrating the weight-sharing and intermediate results in the RNN.

where w is the learnable weight, u_1, u_2, and u_3 are sequential inputs, and p, q, r, s, and t are intermediate values.

(a) **Fill in the blanks for the intermediate values during the forward pass, in terms of w and the u_i's:**

$$p = u_1 \qquad q = w \cdot u_1 \qquad r = u_2 + q = u_2 + w \cdot u_1$$

$$s = w \cdot r = w \cdot u_2 + w^2 \cdot u_1$$

$t =$ ______________________________

$y =$ ______________________________

(b) **Using the expression for y from the previous subpart, compute $\frac{dy}{dw}$.**

(c) **Fill in the blank for the missing partial derivative of y with respect to the nodes on the backward pass.** You may use values for p, q, r, s, t, y computed in the forward pass and downstream derivatives already computed.

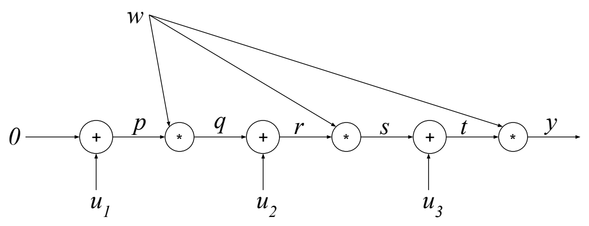
$$\frac{\partial y}{\partial t} = w \qquad \frac{\partial y}{\partial s} = w \qquad \frac{\partial y}{\partial r} = \frac{\partial y}{\partial s} \cdot w \qquad \frac{\partial y}{\partial q} = \frac{\partial y}{\partial r} \cdot 1$$

$$\frac{\partial y}{\partial p} = \underline{\hspace{6cm}}$$

(d) **Calculate the partial derivatives along each of the three outgoing edges from the learnable w in Figure 1, replicated below.** (e.g., the right-most edge has a relevant partial derivative of t in terms of how much the output y is effected by a small change in w as it influences y through this edge. You need to compute the partial derivatives for the other two edges yourself.)

You can write your answers in terms of the p, q, r, s, t and the partial derivatives of y with respect to them.

Use these three terms to find the total derivative $\frac{dy}{dw}$.



(HINT: You can use your answer to part (b) to check your work.)

2. Implementing RNNs (and optionally, LSTMs)

This problem involves filling out this notebook.

Note that implementing the LSTM portion of this question is optional and out-of-scope for the exam.

(a) **Implement Section 1A in the notebook**, which constructs a vanilla RNN layer. This 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.

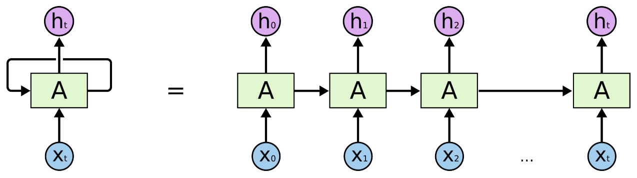


Figure 2: Source: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Copy the outputs of the *"Test Cases"* code cell and paste it into your submission of the written assignment.

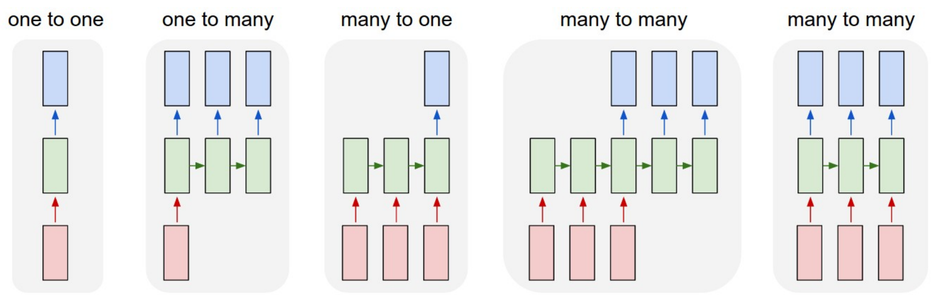
(b) **Implement Section 1.B of the notebook**, in which you'll use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

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

We'll compute one prediction for each timestep.

Copy the outputs of the "*Tests*" code cell and paste it into your submission of the written assignment.

(c) RNNs can be used for many kinds of prediction problems, as shown below. In this notebook we will look at many-to-one prediction and aligned many-to-many prediction.



We will use 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

Implement Section 1.C in the notebook, in which you'll look at the synthetic dataset shown and implement a loss function for the two problem variants.

Copy the outputs of the "*Tests*" code cell and paste it into your submission of the written assignment.

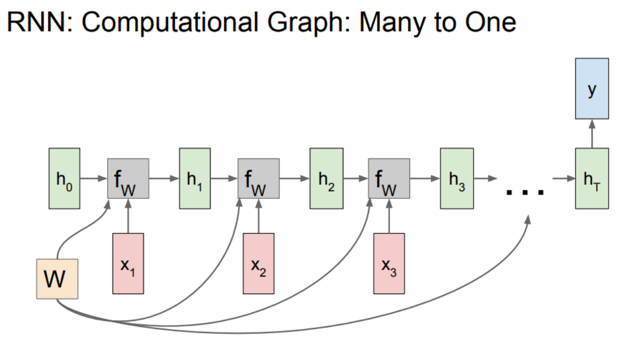


Figure 3: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(d) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 3, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{2}$$

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.

Implement Notebook Section 1.D, which 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.

Include a screenshot of your visualization in the written assignment submission.

(e) **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?

(f) 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.)

(g) **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.)

(h) (Optional) **Implement Section 1.8 of the notebook** in which you implement a LSTM layer. LSTMs pass a cell state between timesteps as well as a hidden state. **Explore gradient magnitudes using the visualization tool you implemented earlier and report on the results.**

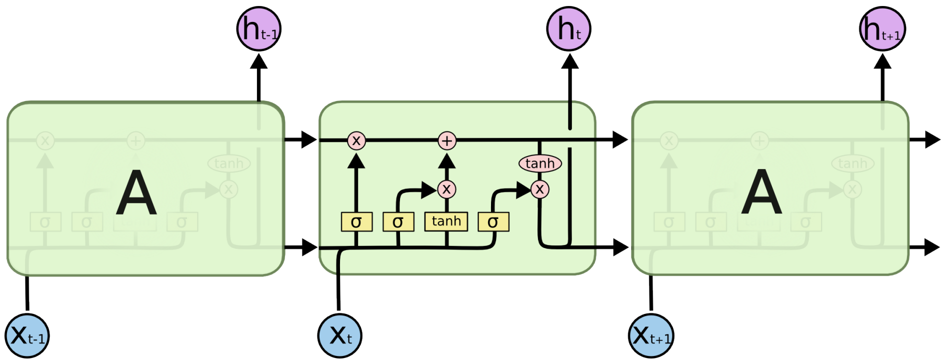


Figure 4: Image source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

The LSTM forward pass is shown below:

$$\begin{aligned} f_t &= \sigma(x_t U^f + h_{t-1} W^f + b^f) \\ i_t &= \sigma(x_t U^i + h_{t-1} W^i + b^i) \\ o_t &= \sigma(x_t U^o + h_{t-1} W^o + b^o) \\ \tilde{C}_t &= \text{tanh}(x_t U^g + h_{t-1} W^g + b^g) \\ C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\ h_t &= \text{tanh}(C_t) \circ o_t \end{aligned}$$

where $\circ$ represents the Hadamard Product (elementwise multiplication) and σ is the sigmoid function.

(i) (Optional) When using an LSTM, you should still see vanishing gradients, but the gradients should vanish less quickly. **Interpret why this might happen by considering gradients of the loss with respect to the cell state.** (Hint: consider computing $\frac{\partial \mathcal{L}}{\partial C_{T-1}}$ using the terms $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$).

(j) (Optional)

Consider a ResNet with simple resblocks defined by $h_{t+1} = \sigma(W_t h_t + b_t) + h_t$. **Draw a connection between the role of a ResNet's skip connections and the LSTM's cell state in facilitating gradient propagation through the network.**

(k) (Optional) We can create multi-layer recurrent networks by stacking layers as shown in Figure 5. The hidden state outputs from one layer become the inputs to the layer above.

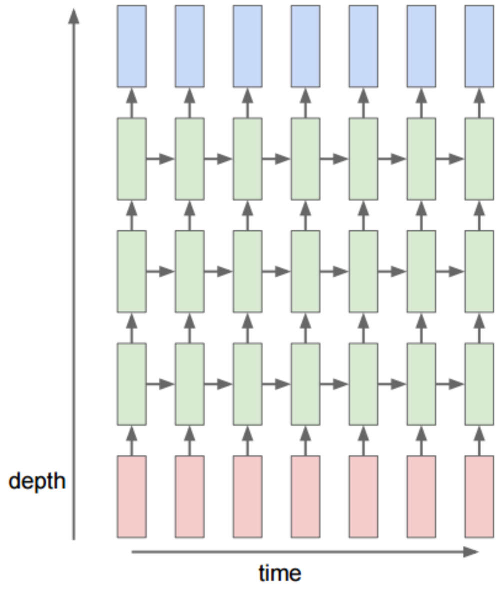


Figure 5: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

Implement notebook Section 1.K and run the last cell to train your network. You should be able to reach training loss < 0.001 for the 2-layer networks, and <.01 for the 1-layer networks.

3. RNNs for Last Name Classification

Please follow the instructions in this notebook. You will train a neural network to predict the probable language of origin for a given last name / family name in Latin alphabets.

(a) Although the neural network you have trained is intended to predict the language of origin for a given last name, it could potentially be misused. **In what ways do you think this could be problematic in real-world applications**?

4. Auto-encoder : Learning without Labels

So far, with supervised learning algorithms we have used labelled datasets $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ to learn a mapping f_θ from input x to labels y. In this problem, we now consider algorithms for *unsupervised learning*, where we are given only inputs x, but no labels y. At a high-level, unsupervised learning algorithms leverage some structure in the dataset to define proxy tasks, and learn *representations* that are useful for solving downstream tasks.

Autoencoders present a family of such algorithms, where we consider the problem of learning a function $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ from input x to a *intermediate representation* z of x (usually $k \ll m$). For autoencoders, we use reconstructing the original signal as a proxy task, i.e. representations are more likely to be useful for downstream tasks if they are low-dimensional but encode sufficient information to approximately reconstruct the input. Broadly, autoencoder architectures have two modules:

- An encoder $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ that maps input x to a representation z.

- A decoder $g_\phi : \mathbb{R}^k \to \mathbb{R}^m$ that maps representation z to a reconstruction $\hat{x}$ of x.

In such architectures, the parameters (θ, ϕ) are learnable, and trained with the learning objective of minimizing the reconstruction error $\ell(\hat{x}_i, x_i) = \|x - \hat{x}\|_2^2$ using gradient descent.

$$\theta_{\text{enc}}, \phi_{\text{dec}} = \underset{\Theta,\Phi}{\text{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{x}_i, x_i)$$
$$\hat{x} = g_\phi(f_\theta(x))$$

Note that above optimization problem does not require labels $\mathbf{y}$. In practice however, we would want to use the *pretrained* models to solve the *downstream* task at hand, e.g. classifying MNIST digits. *Linear Probing* is one such approach, where we fix the encoder weights, and learn a simple linear classifier over the features $z = \text{encoder}(x)$.

(a) Designing AutoEncoders

Please follow the instructions in this notebook. You will train autoencoders, denoising autoencoders, and masked autoencoders on a synthetic dataset and the MNIST dataset. Once you finished with the notebook,

- Answer the following questions in your submission of the written assignment:

(i) **Show your visualization** of the vanilla autoencoder with different latent representation sizes.

(ii) Based on your previous visualizations, answer this question: **How does changing the latent representation size of the autoencoder affect the model's performance in terms of reconstruction accuracy and linear probe accuracy? Why?**

(b) PCA & AutoEncoders

In the case where the encoder f_θ, g_ϕ are linear functions, the model is termed as a *linear autoencoder*. In particular, assume that we have data $x_i \in \mathbb{R}^m$ and consider two weight matrices: an encoder $W_1 \in \mathbb{R}^{k\times m}$ and decoder $W_2 \in \mathbb{R}^{m\times k}$ (with $k < m$). Then, a linear autoencoder learns a low-dimensional embedding of the data $\mathbf{X} \in \mathbb{R}^{m\times n}$ (which we assume is zero-centered without loss of generality) by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \frac{1}{n}||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 \tag{3}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > 0$ are the k largest eigenvalues of $\frac{1}{n}\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components, and is common in this setting. Therefore, the top-k eigenvalues of $\mathbf{X}$ are $S = \text{diag}(\sigma_1, \ldots, \sigma_\text{k})$, with corresponding eigenvectors are the columns of $\mathbf{U}_k \in \mathbb{R}^{m\times k}$. A well-established result from [1] shows that principal components are the unique optimal solution to linear autoencoders (up to sign changes to the projection directions). In this subpart, we take some steps towards proving this result.

(i) **Write out the first order optimality conditions that the minima of Eq. 3 would satisfy.**

(ii) **Show that the principal components $\mathbf{U}_k$ satisfy the optimality conditions outlined in (i).**

5. Self-supervised Linear Autoencoders

[1]Baldi, Pierre, and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima" (1989)

We consider linear models consisting of two weight matrices: an encoder $W_1 \in \mathbb{R}^{k\times m}$ and decoder $W_2 \in \mathbb{R}^{m\times k}$ (assume $1 < k < m$). The traditional autoencoder model learns a low-dimensional embedding of the n points of training data $\mathbf{X} \in \mathbb{R}^{m\times n}$ by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \frac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 \tag{4}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > \sigma_{k+1}^2 \geq 0$ are the $k+1$ largest eigenvalues of $\frac{1}{n}\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components.

Consider an ℓ_2-regularized linear autoencoder where the objective is:

$$\mathcal{L}_\lambda(W_1, W_2; \mathbf{X}) = \tfrac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 + \lambda\|W_1\|_F^2 + \lambda\|W_2\|_F^2. \tag{5}$$

where $\|\cdot\|_F^2$ represents the Frobenius norm squared of the matrix (i.e. sum of squares of the entries).

(a) You want to use SGD-style training in PyTorch (involving the training points one at a time) and self-supervision to find W_1 and W_2 which optimize (5) by treating the problem as a neural net being trained in a supervised fashion. **Answer the following questions and briefly explain your choice:**

(i) **How many linear layers do you need?**

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3

(ii) **What is the loss function that you will be using?**

- ☐ `nn.L1Loss`
- ☐ `nn.MSELoss`
- ☐ `nn.CrossEntropyLoss`

(iii) **Which of the following would you need to optimize (5) exactly as it is written? (Select all that are needed)**

- ☐ Weight Decay
- ☐ Dropout
- ☐ Layer Norm
- ☐ Batch Norm
- ☐ SGD optimizer

(b) **Do you think that the solution to (5) when we use a small nonzero λ has an inductive bias towards finding a W_2 matrix with approximately orthonormal columns? Argue why or why not?**
(Hint: Think about the SVDs of $W_1 = U_1\Sigma_1V_1^\top$ and $W_2 = U_2\Sigma_2V_2^\top$. You can assume that if a $k\times m$ or $m\times k$ matrix has all k of its nonzero singular values being 1, then it must have orthonormal rows or columns. Remember that the Frobenius norm squared of a matrix is just the sum of the squares of its singular values. Further think about the minimizer of $\frac{1}{\sigma^2} + \sigma^2$. Is it unique?)

6. Exploring Deep Learning Tooling

Deep learning in practice often requires the use of various tooling in order make the learning workflow efficient. Among these include tools for creating graphs and organizing experiments. These kinds of tools can aid in careful ablation studies, and can accelerate research. We will explore the use of two different tools in this question: tensorboard and wandbai.

(a) Complete the first notebook: tensorboard.ipynb. Provide your graphs here. What is easy about tensorboard? What do you dislike? Would it still be easy to use when we need to run massive amounts of experiments? How organized is it?

(b) Complete the second notebook: wandb.ipynb. No need to provide your graphs. What does wandb have that tensorboard does not?

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Saagar Sanghavi.
- Dhruv Shah.
- Olivia Watkins.
- Jerome Quenum.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.
- Linyuan Gong.
- Kumar Krishna Agrawal.
- Sheng Shen.
- Liam Tan.