EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Homework 6

This homework is due on Friday, March 10, 2023, 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:

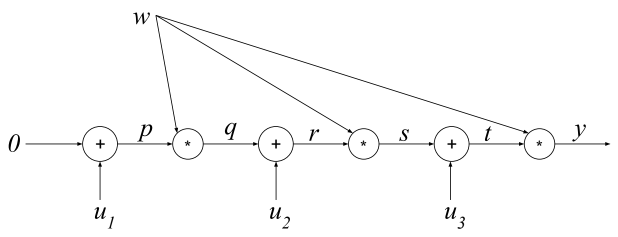


Figure 1: Illustrating the weight-sharing and intermediate results in the RNN.

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:**

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

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

$$t = \underline{\hspace{6cm}}$$

Solution: $t = u_3 + s = u_3 + w \cdot u_2 + w^2 \cdot u_1$

$$y = \underline{\hspace{6cm}}$$

Solution: $y = w \cdot t = w \cdot u_3 + w^2 \cdot u_2 + w^3 \cdot u_1$

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

Solution: $\frac{dy}{dw} = u_3 + 2wu_2 + 3w^2u_1$

(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.

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

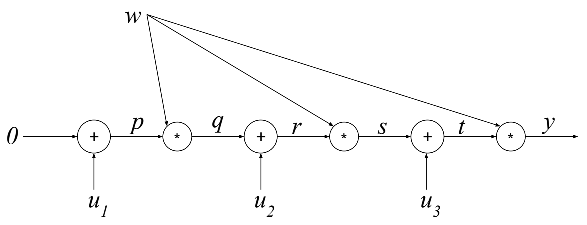
$$\frac{\partial y}{\partial p} = \underline{\hspace{6cm}}$$

Solution: $\frac{\partial y}{\partial p} = \frac{\partial y}{\partial q} \cdot w = w \cdot w \cdot w = w^3$

(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.)

Solution: Along the right edge, we have $t = u_3 + wu_2 + w^2u_1$ (This is provided for you).
Along the middle edge, we have $r \cdot \frac{\partial y}{\partial s} = r \cdot w = (u_2 + wu_1)w = wu_2 + w^2u_1$
Along the left edge, we have $p \cdot \frac{\partial y}{\partial q} = p \cdot w^2 = u_1w^2$
Adding all of these up, we have

$$\frac{dy}{dw} = t + r \cdot \frac{\partial y}{\partial s} + p \cdot \frac{\partial y}{\partial q} \tag{2}$$

$$= (u_3 + wu_2 + w^2u_1) + (u_2 + wu_1)w + w^2u_1 \tag{3}$$

$$= 3w^2u_1 + 2wu_2 + u_3 \tag{4}$$

Which is same as answer to (b) so everything checks out.

2. Beam Search

This problem will also be covered in discussion.

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
```

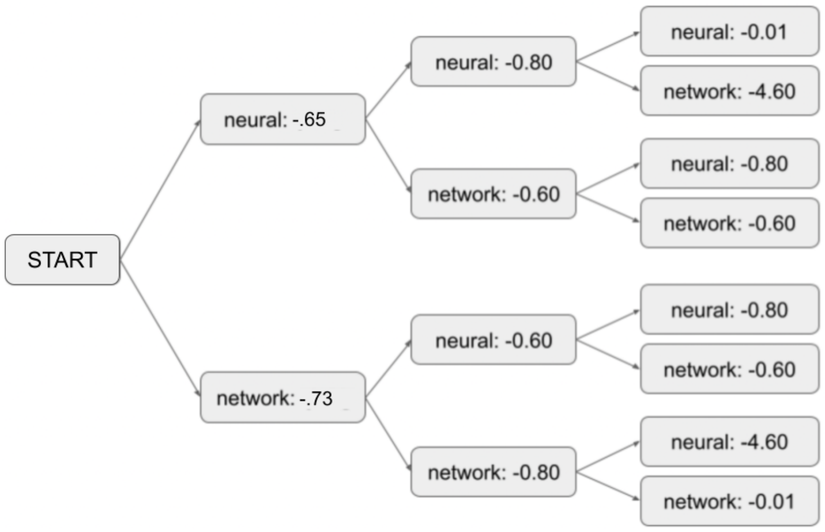


Figure 2: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

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 2, 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: $O(TKM \log M)$.

3. 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.

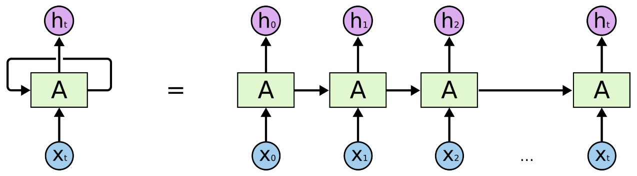


Figure 3: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

Solution: See the solution notebook. Each max error should be less than `1e-4`.

(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.

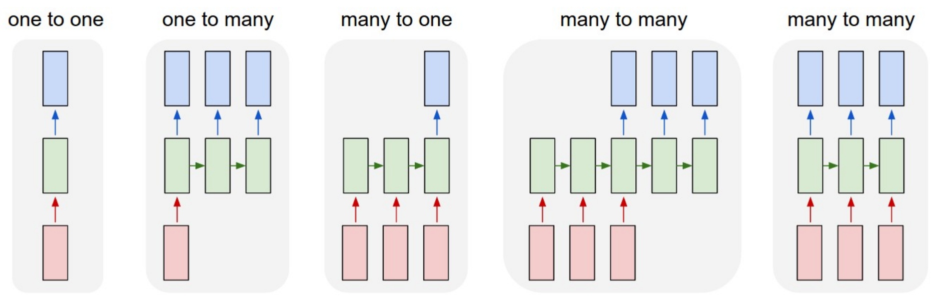
$$\hat{y}_t = W^f h_t + b^f$$

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.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

(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.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

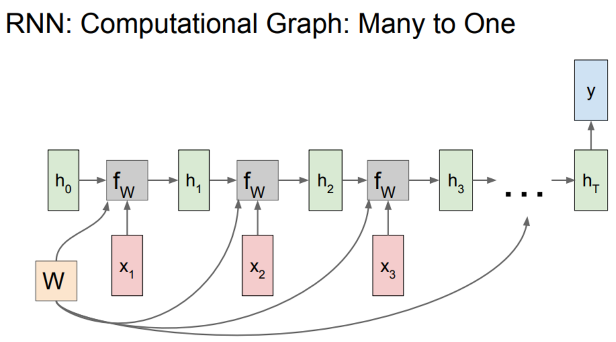


Figure 4: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(d) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 4, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{5}$$

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.

Solution: See the solution notebook.

(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?

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.

(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.)

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.

(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.)

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.

(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.** **Solution:** Hidden state outputs remain between +/- 1. Gradients don't explode, but they still vanish. Typically, they don't vanish as fast as vanilla RNNs.

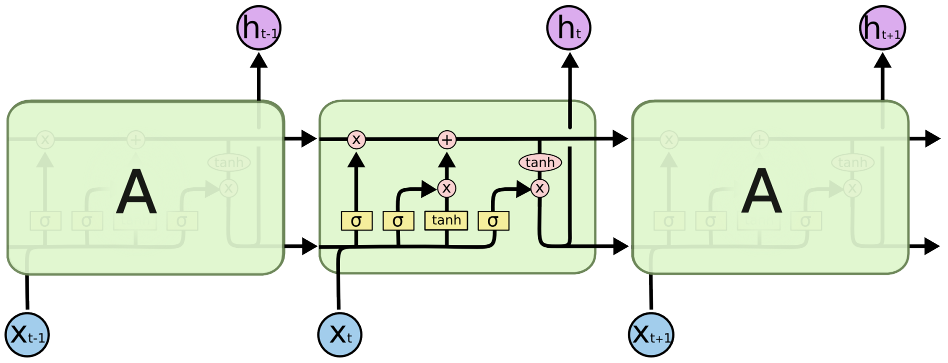


Figure 5: Image source: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

The LSTM forward pass is shown below:

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

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 $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$).

Solution: In an LSTM, information can be stored in the cell state without needing to persist through a chain of matrix multiplications.

$$\frac{\partial \mathcal{L}}{\partial C_{T-1}} = \frac{\partial \mathcal{L}}{\partial h_T}\left(\frac{\partial h_T}{\partial C_T}\frac{\partial C_T}{\partial C_{T-1}} + \frac{\partial h_T}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial C_{T-1}}\right)$$

Unlike $\frac{\partial h_T}{\partial h_{T-1}}$, which results in decaying gradients due to matrix multiplication, $\frac{\partial C_T}{\partial C_{T-1}}$ is a diagonal matrix with f_t on the diagonal. If the network sets f_t close to 1, then $\frac{\partial C_T}{\partial C_{T-1}}$ will be close to the identity, allowing information stored in the cell state to "pass through" without decay. In practice, however, many elements of f_t are not close to 1, which results in some gradient decay.

In addition, LSTMs also contain a hidden state in which to store information, and this "pathway" through the network will decay like in a vanilla RNN, potentially resulting in some overall gradient decay.

(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.**

Solution: ResNet skip connections and LSTM cell states both allow information to pass through the network without requiring a matrix multiplication at each layer. As a result, they both mitigate vanishing gradients.

(k) (Optional) We can create multi-layer recurrent networks by stacking layers as shown in Figure 6. The hidden state outputs from one layer become the inputs to the layer above.

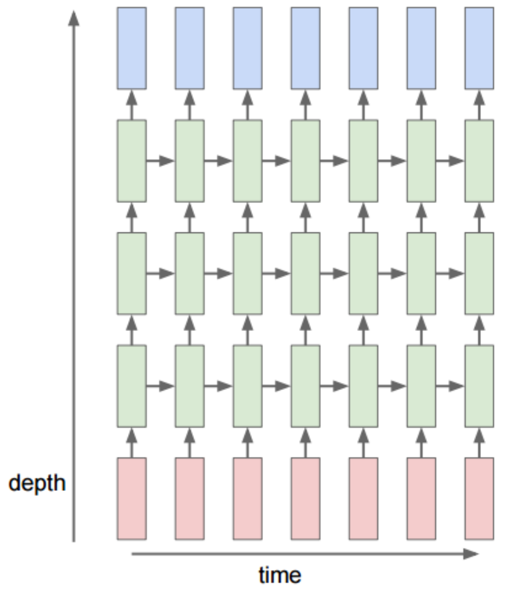


Figure 6: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

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.

4. 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. Once you finished with the notebook, download `submission_log.json` and submit it to "Homework 6 (Code)" in Gradescope.

(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**?

Solution: The model's predictions could be used to make assumptions about a person's nationality or ethnicity, which could lead to discrimination or bias. It's important to note that the model should only be used to make predictions about the language of origin for a given name and not to make assumptions about a person's identity. The model could also be used to infer personal information about individuals, such as their cultural background, which could be considered an invasion of privacy.

5. Read a Blog Post: How to train your Resnet

In previous homeworks, we saw how memory and compute constraints on GPUs put limits on the architecture and the hyperparameters (e.g., batch size) we can use to train our models. To train better models, we could scale up by using multiple GPUs, but most distributed training techniques scale sub-linearly and often we simply don't have as many GPU resources at our disposal. This raises a natural question - how can we make model training more efficient on a single GPU?

The blog series How to train your Resnet (https://myrtle.ai/learn/how-to-train-your-resnet/) explores how to train ResNet models efficiently on a single GPU. It covers a range of topics, including architecture, weight decay, batch normalization, and hyperparameter tuning. In doing so, it provides valuable insights into the training dynamics of neural networks and offers lessons that can be applied in other settings.

Read the blog series and answer the questions below.

(a) **What is the baseline training time and accuracy the authors started with? What was the final training time and accuracy achieved by the authors?**

Solution: The baseline was Ben Johnson's run, which reached 94% accuracy in 341s. With the optimizations made by the authors, they achieved 94.1% in 26s.

(b) **Comment on what you have learnt.** ($\approx$ 100 words)

Solution: There is no "right" answer. This is an open ended question, and each student may have a different answer. Key takeaways can include (but are not limited to) tradeoffs between small batch vs large batch training, computing batch norms can be slow if not using optimized code, the importance of resource-constraining ML benchmarks, weight-decay dynamics, the importance of batch norm etc.

(c) **Which approach taken by the authors interested you the most? Why?** ($\approx$ 100 words)

Solution: Open ended question, there's no "right" answer.

6. Convolutional Networks

Note: Throughout this problem, we will use the convention of NOT flipping the filter before dragging it over the signal. This is the standard notation with neural networks (ie, we assume the filter given to us is already flipped)

(a) **List two reasons we typically prefer convolutional layers instead of fully connected layers when working with image data.**

Solution: The basic reason is so that we can get an inductive bias that is suited for the regularities often present in image data so that we can learn without having to use inordinate amounts of training data. Valid reasons for using convolutional layers to do this include:

- Convolutions with finite sized filters respect the kind of locality and by layering them, we can reflect the heirarchical part/whole structure decompositions that we see in images.
- Convolutions are nicely compatible with weight-sharing which builds in a basic invariance/equivariance to translations which is reasonable since in most image-related contexts, the desired answers for a translated image are closely related to the original image. The weight-sharing allows the numbers of parameters to be much smaller.
- Convolutions build in the idea that left/right and up/down can be different and allow different weights for those.
- There is a long history of using filterbanks and convolutional structures in hand-designed approaches to computer vision and image processing so it is natural to simply replace those with learned weights in similar structures.
- There is biological inspiration based on what we know about how eyes work for the locality properties and the kinds of edge-detecting responses that convolutions capture naturally.

- Furthermore, other people have used convolutional layers to great success when working with vision and in Deep Learning, it is natural to try to build on the successes of others even when we don't understand why they succeeded.

(b) Consider the following 1D signal: $[1, 4, 0, -2, 3]$. After convolution with a length-3 filter, no padding, stride=1, we get the following sequence: $[-2, 2, 11]$. **What was the filter?**

(Hint: Just to help you check your work, the first entry in the filter that you should find is 2. However, if you try to use this hint directly to solve for the answer, you will not get credit since this hint only exists to help you check your work.)

Solution: If we represent the original filter as $[h_1, h_2, h_3]$, we can set up the system of equations:

$$\begin{aligned} h_1 + 4h_2 + 0h_3 &= -2 \\ 4h_1 + 0h_2 - 2h_3 &= 2 \\ 0h_1 - 2h_2 + 3h_3 &= 11 \end{aligned}$$

Solving, we get $[h_1, h_2, h_3] = [2, -1, 3]$

This could be done using Gaussian elimination or substitution. By substitution, we see $h_1 = -2 - 4h_2$ and $h_3 = \frac{11}{3} + \frac{2}{3}h_2$. Plugging into the second equation we get $-8 - 16h_2 - \frac{22}{3} - \frac{4}{3}h_2 = 2$ which simplifies to $\frac{52}{3} = -\frac{52}{3}h_2$ which solves to $h_2 = -1$. Back substituting gives us $h_1 = 2$ and $h_3 = \frac{9}{3} = 3$.

The Gaussian elimination approach involves similar numbers.

(c) Transpose convolution is an operation to help us upsample a signal (increase the resolution). For example, if our original signal were $[a, b, c]$ and we perform transpose convolution with pad=0 and stride=2, with the filter $[x, y, z]$, the output would be $[ax, ay, az + bx, by, bz + cx, cy, cz]$. Notice that the entries of the input are multipled by each of the entries of the filter. Overlaps are summed. Also notice how for a fixed filtersize and stride, the dimensions of the input and output are swapped compared to standard convolution. (For example, if we did standard convolution on a length-7 sequence with filtersize of 3 and stride=2, we would output a length-3 sequence).

If our 2D input is $\begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix}$ and the 2D filter is $\begin{bmatrix} +1 & -1 \\ 0 & +1 \end{bmatrix}$ **What is the output of transpose convolution with pad=0 and stride=1?**

Solution: Transpose convolution is designed to do an operation that can upsample a signal while simultaneously filtering it. That application is easiest to understand in the context of a stride that is greater than 1 — the original input signal is turned into a scaled sequence of "impulses" separated by the stride and then the filter acts on each of those impulses to smear them out and let them interact (linearly) with the other impulses to result in the final signal.

For this problem, the stride was set to 1 to give a nontrivial problem without making the arithmetic and size of the output too big.

The first conceptual part of this question is simply understanding what the output dimensions should be. If this were a standard convolution with pad=0, then we would only get a 1×1 sized output. But this is a "transpose convolution" and so pad=0 means that we don't want to drop any of the outputs that would naturally arise from the transpose convolution operation. In this case, this means that we will get a 3×3 output. We can "type-check" this by looking at what we would get with a regular transpose convolution starting with a 3×3 and applying a 2×2 filter with zero pad and stride 1 — we indeed would get a 2×2 back.

Given that the input is $\begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix}$ we can look at the filter responses to each of the terms as

$$-1\begin{bmatrix} +1 & -1 & 0 \\ 0 & +1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + 2\begin{bmatrix} 0 & +1 & -1 \\ 0 & 0 & +1 \\ 0 & 0 & 0 \end{bmatrix} + 3\begin{bmatrix} 0 & 0 & 0 \\ +1 & -1 & 0 \\ 0 & +1 & 0 \end{bmatrix} + 1\begin{bmatrix} 0 & 0 & 0 \\ 0 & +1 & -1 \\ 0 & 0 & +1 \end{bmatrix} \tag{6}$$

$$= \begin{bmatrix} -1 & +1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & +2 & -2 \\ 0 & 0 & +2 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ +3 & -3 & 0 \\ 0 & +3 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & +1 & -1 \\ 0 & 0 & +1 \end{bmatrix} \tag{7}$$

$$= \begin{bmatrix} -1 & 3 & -2 \\ 3 & -3 & 1 \\ 0 & 3 & 1 \end{bmatrix}. \tag{8}$$

Notice above how the filter matrix has just been shifted each time by the relevant stride (1) times the position of the corresponding element in the input matrix. This is the "dragging" operation at the root of all convolutions as applied for transpose convolution.

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? Write it down here where you'll need to remember it for the self-grade form.**

Contributors:

- Saagar Sanghavi.
- CS 182 Staff from past semesters.
- Olivia Watkins.
- Kumar Krishna Agrawal.
- Dhruv Shah.
- Jerome Quenum.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.

- Past EECS 282 and 227 Staff.
- Linyuan Gong.
- Romil Bhardwaj.