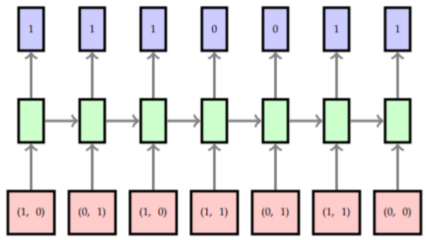
1 Binary Addition (2.5pts): In this problem, you will design a recurrent neural network to implement binary addition. The inputs are provided as binary sequences, starting with the least significant bit. The sequences are padded with at least one zero at the end. Here is an example:

$$0101101 + 0111010 = 1100111$$

where the inputs and outputs would be represented as:

- **Input 1:** 1, 0, 1, 1, 0, 1, 0
- **Input 2:** 0, 1, 0, 1, 1, 1, 0
- **Correct Output:** 1, 1, 1, 0, 0, 1, 1

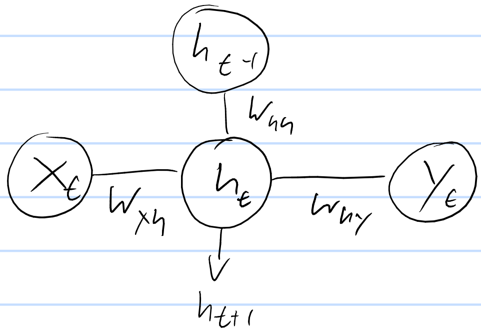
At each time instance, the RNN would have two inputs and one output. Therefore, the pattern of inputs and outputs for the above example would be:



Design, by hand, the weights and biases for a RNN that can perform binary addition. The RNN should consist of two inputs, three hidden nodes and one output. You can assume that all the nodes use the hard threshold activation function. In particular, find the weights $\boldsymbol{W}_{xh}$, $\boldsymbol{W}_{hh}$ and $\boldsymbol{W}_{hy}$, the bias vector $\boldsymbol{b}_h$ and scalar b_y.

Solution on next page!

Problem 1



$$Y_t = \Theta(W_{hy} h_t + b_y)$$

$$h_t = \Theta(W_{hh} h_{t-1} + W_{xh} X_t + b_h)$$

where $\Theta(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$

$$x_t \in \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

Example:

t		6 5 4 3 2 1 0
x_1		0 1 0 1 1 0 1
x_2	+	0 1 1 1 1 0 1 0
x_1+x_2	h_1	0 0 1 0 1 1 1 1
	h_2	0 1 0 1 0 0 0 0
	h_3	1 1 1 0 0 0 0 0
y		1 1 0 0 1 1 1

Idea: h_1, h_2 will be the bits of $x_1 + x_2$.
h_3 will be the carry-over bit:

$$h_{t,3} = \begin{cases} 1, & h_{t-1,2} = 1 \\ 1, & h_{t-1,1} = 1 \text{ and } h_{t-1,3} = 1 \\ 0 & \text{otherwise} \end{cases}$$

We want $y = h_1 \text{ XOR } h_3 \Rightarrow W_{hy} = (-1 \;\; 0 \;\; -1)$

$1 < b_y < 2$, let's say $b_y = 1.5$

We want $h_1 = x_1 \text{ XOR } x_2$
and $h_2 = x_1 \text{ AND } x_2$ $\Rightarrow W_{xh} = \begin{pmatrix} -1 & -1 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}$, $b_{y,1} = 1.5$, $b_{y,2} = -1.5$

$-2 < b_{y,2} < -1$

$$h_{t,3} = \begin{cases} 1, & h_{t-1,2} = 1 \\ 1, & h_{t-1,1} = 1 \text{ and } h_{t-1,3} = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\Rightarrow W_{hh} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}, \quad b_{y,3} = -1.5$$

$-2 < b_{y,2} < -1$

or any other number $> |b_{y,3}|$

2 LSTM Gradient (3.5pts): In this problem you will derive the backpropagation-through-time equations for a univariate version of the Long-Term Short-Term Memory (LSTM) architecture we saw in class. Consider the case when the bias terms are assumed to be zero. So the LSTM computations are:

$$
\begin{aligned}
i^{(t)} &= \sigma(w_{xi}x^{(t)} + w_{hi}h^{(t-1)}) && \text{input} \\
f^{(t)} &= \sigma(w_{xf}x^{(t)} + w_{hf}h^{(t-1)}) && \text{forget} \\
o^{(t)} &= \sigma(w_{xo}x^{(t)} + w_{ho}h^{(t-1)}) && \text{output} \\
g^{(t)} &= tanh(w_{xg}x^{(t)} + w_{hg}h^{(t-1)}) && \text{input activation} \\
c^{(t)} &= f^{(t)}c^{(t-1)} + i^{(t)}g^{(t)} && \text{cell state} \\
h^{(t)} &= o^{(t)}tanh(c^{(t)}) && \text{output}
\end{aligned}
$$

gates

1. **(3pts)** Derive the backpropagation-through-time equations for the activations and the gates:

$$
\begin{aligned}
\overline{h^{(t)}} &= \ldots \\
\overline{c^{(t)}} &= \ldots \\
\overline{g^{(t)}} &= \ldots \\
\overline{o^{(t)}} &= \ldots \\
\overline{f^{(t)}} &= \ldots \\
\overline{i^{(t)}} &= \ldots
\end{aligned}
$$

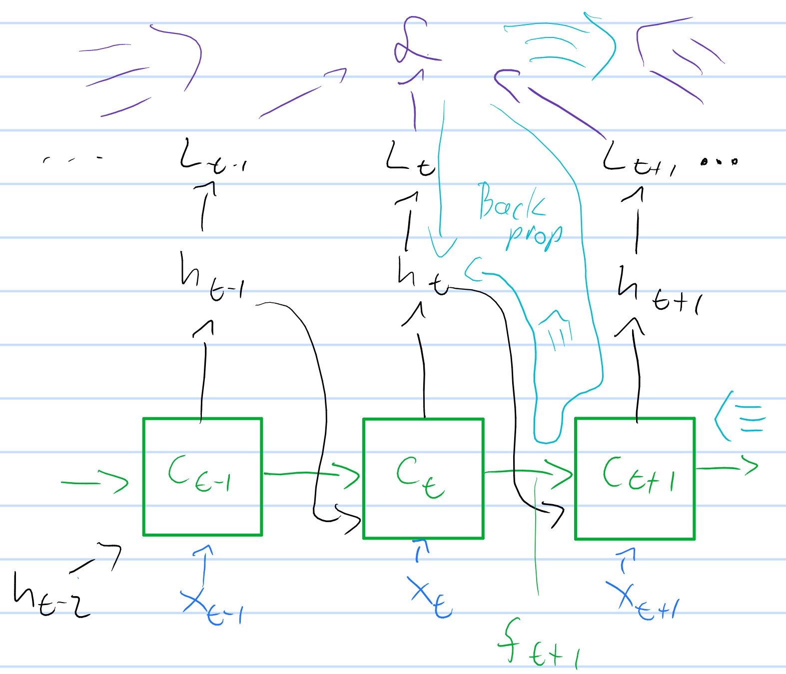
We simply use the regular backpropagation algorithm on the unfolded graph.

I prefer to write $i_t = i^{(t)}$, $o_t = o^{(t)}$, etc.

Before we start, recall the derivatives:

$\frac{d}{dx}\tanh(x) = 1 - \tanh^2 x$, $\frac{d}{dx}\sigma(x) = \sigma(x)\cdot(1-\sigma(x))$

We have $\mathcal{L} = \sum_{t=1}^{T} L_t$ with some differentiable loss $L_t(h_t, y_t)$ where the y_t are labels.



$$\bar{h}_t = \frac{\partial \mathcal{L}}{\partial h_t} = \frac{\partial}{h_t}\left(\sum_{k=1}^{T} L_k\right) = \sum_{k=t}^{T} \frac{\partial L_k}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \bar{c}_{t+1}\frac{\partial c_{t+1}}{\partial h_t}$$

only $\geq t$ contribute

$$= \frac{\partial L_t}{\partial h_t} + \bar{C}_{t+1}\, C_t \frac{\partial}{\partial h_t}\sigma(W_{xf}x_{t+1} + W_{hf}h_t)$$

$$= \frac{\partial L_t}{\partial h_t} + \bar{C}_{t+1}\, C_t W_{hf}\, f_{t+1}(1 - f_{t+1})$$

$$\bar{C}_t = \sum_{k=t}^{T}\frac{\partial L_k}{\partial C_t} = \bar{h}_t \frac{\partial h_t}{\partial C_t} + \sum_{k=t+1}^{T}\frac{\partial L_k}{\partial C_t} = \bar{h}_t\, o_t\left(1 - \tanh^2 C_t\right) + \bar{C}_{t+1}\, f_{t+1}$$

$$\bar{g}_t = \frac{\partial \mathcal{L}}{\partial C_t}\frac{\partial C_t}{\partial g_t} = \bar{C}_t \cdot i_t$$

$$\bar{o}_t = \bar{h}_t \frac{\partial h_t}{\partial o_t} = \bar{h}_t \tanh(C_t)$$

$$\bar{f}_t = \bar{C}_t \frac{\partial C_t}{\partial f_t} = \bar{C}_t\, C_{t-1}$$

$$\bar{i}_t = \bar{C}_t \frac{\partial C_t}{\partial i_t} = \bar{C}_t\, g_t$$

2. (**0.5pt**) Derive the update equation for the weight w_{xi}:

$$\overline{w_{xi}} = \dots$$

$$\overline{w_{xi}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial w_{xi}} = \sum_{t=1}^{T}\sum_{k=1}^{T} \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial w_{xi}}$$

gate activation $= w_{xi} x_t + w_{hi} h_{t-1}$

$$= \sum_{t=1}^{T} \overline{i_t}\,\frac{\partial i_t}{\partial w_{xi}} = \sum_{i=1}^{T} \overline{c_t}\, g_t\, x_t \cdot \sigma(\hat{i}_t)(1-\sigma(\hat{i}_t))$$

$$= \sum_{i=1}^{T} \underbrace{\overline{c_t}\, g_t}_{\overline{i_t}}\, x_t\, i_t\,(1-i_t)$$

3. (**Extra Credit: 0.5pt**) Based on your answers above, explain why the gradient does not explode if the values of the forget gates are very close to 1 and the values of the input and output gates are very close to 0. **Hint:** Your answer should involve both $\overline{h^{(t)}}$ and $\overline{c^{(t)}}$.

$f_t \approx 1$, $i_t \ll 1$, $o_t \ll 1$

In this case $g_t x_t i_t (1-i_t) =: v_t$

where v_t is some small constant. It's roughly the same for each t: $v_t \approx v$

So: $\overline{w_{xi}} \approx v \sum_{t=1}^{T} \overline{c_t}$

$$= v \sum_{t=1}^{T} \Big[\underbrace{\overline{h_t}\, o_t\,(1-\tanh^2 c_t)}_{\text{small } \tilde{v}_t} + \overline{c_{t+1}}\, \underbrace{f_{t+1}}_{\approx 1} \Big]$$

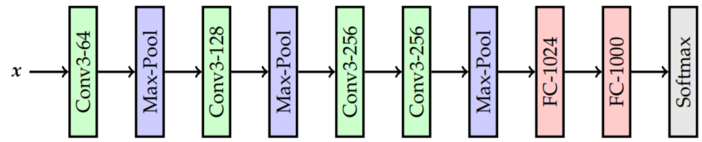
$$\approx v \sum_{t=1}^{T} \overline{c_{t+1}} \approx v\,T\,\overline{c_T}$$

Does not explode!

Because $\overline{c_t} \approx \overline{c_{t+1}}$, only the small contributions $\frac{\partial L_t}{\partial h_t}$ in $\overline{h_t}$ accumulate.

3 Convolutional Neural Networks (2pts):

1. **(1pt)** Consider a CNN with 4 conv layers like in the diagram below. All 4 conv layers have kernel size of 3×3. The number after the hyphen specifies the number of output channels or units of a layer (e.g. Conv3-64 layer has 64 output channels and FC-1024 has 1024 output units). All the Max Pool in the diagram has size of 2×2. Assume zero padding of 1 for conv layers and stride 2 for Max Pool.



Size of the input image is 224×224 with 3 channels. Calculate the total number of parameters in the network including the bias units.

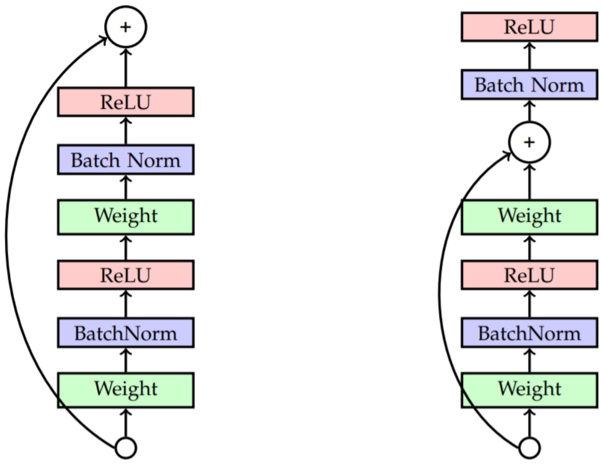
As usual: we want to figure out the number of parameters for each layer and then simply sum them up. I'll try to create a table, to keep track:

Layer	Output shape	#parameters
X	(224, 224, 3)	0
Conv3-64	(224, 224, 64)	#filters channels bias kernel $64 \cdot (3 \cdot 3 \cdot 3 + 1) = 1792$
Max Pool	(112, 112, 64)	0
Conv3-128	(112, 112, 128)	$128 \cdot (64 \cdot 3 \cdot 3 + 1) = 73856$
Max Pool	(56, 56, 128)	0
Conv3-256	(56, 56, 256)	$256 \cdot (128 \cdot 3 \cdot 3 + 1) = 295168$
Conv3-256	(56, 56, 256)	295168
Max Pool	(28, 28, 256) => 20704	0
FC-1024	1024	$1024 \cdot (20704 + 1) = 21201920$
FC-1000	1000	$1000 \cdot (1024 + 1) = 1025000$
Softmax	1000	0

So in total: $1792 + 73856 + 2 \cdot 295168 + 21201920 + 1025000$

$= 22892904$

2. (**1pt**) Consider the following two ResNet architectures for the placement of the batch norm and the residual connection. "Weight" layer can be either a matrix multiplication or convolution. Which architecture is easier to learn in terms of exploding / vanishing gradient? Provide a brief justification for your answer.



From the BN paper (arXiv:1502.03167 **[cs.LG]**) :

Batch Normalization can be applied to any set of activations in the network. Here, we focus on transforms that consist of an affine transformation followed by an element-wise nonlinearity:

$$z = g(W\mathrm{u} + \mathrm{b})$$

where W and b are learned parameters of the model, and $g(\cdot)$ is the nonlinearity such as sigmoid or ReLU. This formulation covers both fully-connected and convolutional layers. We add the BN transform immediately before the nonlinearity, by normalizing $\mathrm{x} = W\mathrm{u} + \mathrm{b}$. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $W\mathrm{u} + \mathrm{b}$ is more likely to have a symmetric, non-sparse distribution, that is "more Gaussian" (Hyvärinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution.

Here we have:

residual

$z = W_+ g(BN(Wu)) + x_+$ left

Vs $z = g(BN(W_+ Wu + x_+))$ right

=> Terefore, the left architecture should be better suited to eliminate the internal covariate shift.

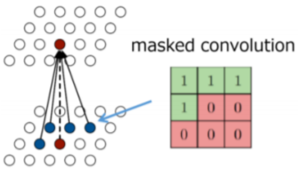
Indeed, this is what is widely usesd.

E.g. : arXiv:1512.03385 **[cs.CV]** and https://openreview.net/pdf?id=S1xU74med4

However, this to me seems like something that requires specific case studies or complicated proof. It is not obvious which architecture will work better after all in practice.

4 Autoregressive Generative Models (2pt): In this question we will consider autoregressive models that model the distribution of images in an autoregressive manner. For simplicity, assume that each conditional distribution is a Gaussian with a fixed variance, so the model predicts the mean of the next pixel given all the previous pixels.

1. **(1pt)** Here we will consider PixelCNN, which models the distribution of images using a convolutional architecture. PixelCNN masks each convolutional filter to only see the pixels that appear before the current pixel in a raster scan order. See figure below for a visualization. Several of such layers are stacked sequentially.



   (a) Consider a d layer PixelCNN model, what is the total number of ~~connections~~ weights? Give your answers in terms of d, k, H, W. You only need to give $\mathcal{O}(\cdot)$, not an exact count.

Assuming that k is the size of the kernels, $H \times W$ the size of the image and d the number of sequentially stacked layers.

For a normal (unmasked) CNN we would have

$$O(dk^2) \text{ weights.}$$

Here, the only diffrence is the mask, so instead of k^2 it's $\frac{k^2}{2}(-1)$, but $O(\frac{k^2}{2} - 1) = O(k^2)$.

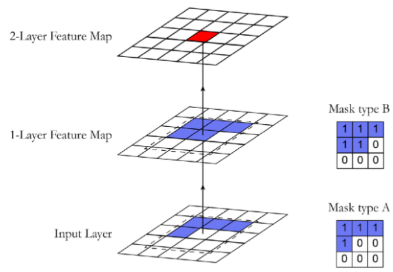
only first layer

So it's still

$$\boxed{O(dk^2) \text{ weights.}}$$

(b) Suppose that in each step we compute as many matrix-vector multiplications in parallel as we can, what is the minimum number of the sequential operations to compute the output of a Pixel CNN in terms of d, k, H, W. You only need to give $\mathcal{O}(\cdot)$, not an exact count.

After training, when we actually want to generate, each pixel has to be predicted sequentially. Considering that PixelCNN uses type B filters for deeper layers, which depend on the output of the previous layer:



Each filter position in each layer has to be evaluated so $O(dHW)$ total operations.

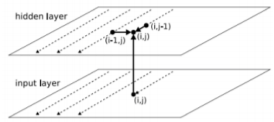
During training, all pixels can be evaluated in parallel. However, all layers except the first (type A) have to be computed sequentially for each pixel.

So during training: $O(d-1) = O(d)$ sequential operations

During evaluation, we can use the same parallel approach as I describe in 2b for the RNN.
So $O(\sqrt{HW})$ sequential operations per layer.

For multiple layers, we can do the same thing in 3D, then: $O(\sqrt{dHW})$ sequential operations.

2. (**1pt**) Here we will consider Multidimensional RNN (MDRNN). This is like the RNNs we discussed in the lecture, except that instead of a 1-D sequence, we have a 2-D grid structure. Analogous to how ordinary RNNs have an input vector and a hidden vector for every time step, MDRNNs have an input vector and hidden vector for every grid square. Each hidden unit receives bottom-up connections from the corresponding input square, as well as recurrent connections from its north and west neighbors as shown in the figure below.



The activations are computed as:

$$h^{(i,j)} = \phi(\mathbf{W}_{in}^T x^{(i,j)} + \mathbf{W}_W^T x^{(i-1,j)} + \mathbf{W}_N^T x^{(i,j-1)}) \tag{1}$$

Denote the number of input channels and the number of recurrent neurons at each layer to be k. The input image size is $H \times W$. For simplicity, we assume there are no bias parameters.

(a) Consider a d layer MDRNN model, what is the total number of ~~connections~~ connections? Give your answers in terms of d, k, H, W. You only need to give $\mathcal{O}(\cdot)$, not an exact count.

During training: x_{ij} are the input image pixels

Generation: $x_{ij} = y_{t-1}$, the predicted pixel from previous step.

$W_x \in \mathbb{R}^{k \times k}$, $x \in \mathbb{R}^k$, $h \in \mathbb{R}^k$

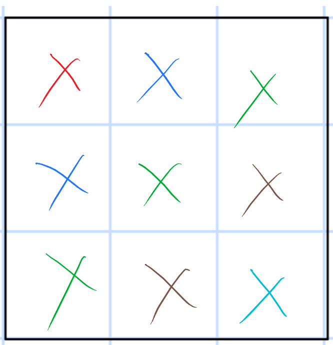
All 3 weight matrices are shared

So we have $O(3k^2) = \boxed{O(k^2) \text{ weights!}}$

(b) Suppose that in each step we compute as many matrix-vector multiplications in parallel as we can, what is the minimum number of the sequential operations to compute the output of a ~~Pixel CNN~~ MDRNN in terms of d, k, H, W. You only need to give $\mathcal{O}(\cdot)$, not an exact count.

There are $H \cdot W$ steps in total. However, we can compute the next step once the north north and west neighbors have been computed. So we can traverse it diagonally and only $\approx 2\sqrt{HW} - 1$ sequential steps are needed. Exact for $H = W$

E.g. for a 3x3 grid:



1.
2.
3.
4.
5.

$5 = 2\sqrt{2 \cdot 2} - 1$

So: $\boxed{O(\sqrt{HW}) \text{ sequential steps!}}$

3. (**Extra Credit: 1pt**) What is the benefit of using PixelCNN over MDRNN. Discuss the pros and cons of the two models in terms of their computational, memory complexity, parallelization potential and the size of their context windows.

PixelCNN	MDRNN
+ Fast to train in parallel	- Slow to train because of sequential dependency
- Smaller context window due to blind spots (see Gated PixelCNN for improvement)	+ Uses less memory
- Memory intensive	+ Maximum possible receptive field
	+ => best performance