Data C182 Designing, Visualizing & Understanding DNN
Fall 2024 Eric Kim, Naveen Ashish **Discussion 04**

This discussion will cover CNN and RNN.

1. Convolutional Neural Networks

Convolutional neural networks[1] (CNN) are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance.

They perform operations similar to feed-forward neural networks that we have discussed, but explicitly account for spatial structure in the data, and so are very common for computer vision tasks where inputs are images. That said, CNNs can also be applied to non-image data with similar structure in the input, such as time series or text data (in which case they're taking advantage of temporal structure).

0.1 Convolution (Cross-Correlation) Operator

At the heart of CNNs is the convolution operator. In this discussion, what we refer to as a convolution is actually the **cross-correlation** operator here instead, which is the exact same but with the indexing of the weights in $\mathbf{w}$ inverted. For example, "convolutional" layers in the deep learning library Pytorch are also actually cross-correlations instead, and homework 1 will also similarly have you implement cross-correlation instead of the actual convolution.

To motivate the use of convolutions, we will work through an example of a 1-D convolution calculation to illustrate how convolutions work over a single spatial dimension. Suppose we have an input $\mathbf{x} \in \mathbb{R}^n$, and filter $\mathbf{w} \in \mathbb{R}^k$. We can compute the convolution of $\mathbf{x} \star \mathbf{w}$ as follows:

(a) Take your convolutional filter $\mathbf{w}$ and align it with the beginning of $\mathbf{x}$. Take the dot product of $\mathbf{w}$ and the $\mathbf{x}[0 : k-1]$ (using Python-style zero-indexing here) and assign that as the first entry of the output.

(b) Suppose we have stride s. Shift the filter down by s indices, and now take the dot product of $\mathbf{w}$ and $\mathbf{x}[s : k-1+s]$ and assign to the next entry of your output.

(c) Repeat until we run out of entries in $\mathbf{x}$.

Below, we illustrate a 1D convolution with stride 1.

$$\overbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input vector } \mathbf{x} \in \mathbb{R}^n} \star \overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}}^{\text{Convolutional filter } \mathbf{w} \in \mathbb{R}^k} = \overbrace{\begin{bmatrix} \sum_{i=1}^k w_i x_i \\ \sum_{i=1}^k w_i x_{i+1} \\ \vdots \\ \sum_{i=1}^k w_i x_{i+n-k} \end{bmatrix}}^{\text{Output vector } \mathbf{y} \in \mathbb{R}^{n-k+1}}$$

[1] Recommended reading: http://cs231n.github.io/convolutional-networks/

We see that the output vector is smaller than the input vector ($\mathbb{R}^{n-k-1}$ compared to $\mathbb{R}^n$). A common way to address this is **zero-padding**, in which we append zeros on both ends of the input vector before applying the convolution (note that there are other conventions for zero-padding as well).

Often, we'll be dealing with multiple spatial dimensions (2 spatial dimensions in the case of images). In this case, we would need to slide our filter along all spatial dimensions to construct the output.

Problem 1: Test your know knowledge of convolution dimensions

In this problem, we will run a series of convolution-related operations to better understand how dimensions are affected by convolutions.

(a) i. Suppose you have a $32 \times 32 \times 3$ image (a 32×32 image with 3 input channels). What are the resulting dimensions when you convolve with a $5 \times 5 \times 3$ filter with stride 1 and 0 padding?

ii. What if we zero-pad the input by 2?

iii. Suppose we now stack 10 of these $5 \times 5 \times 3$ filters and continue to zero pad the input by 2. What is the new shape of the output, and how many parameters are in our filters (not including any bias parameters)?

iv. What would be the spatial dimensions after applying a 1×1 convolution? Think about what this does.

Solution:

Solution 1: Test your know knowledge of convolution dimensions

i. The resulting spatial dimensions are 28×28 (with one output channel).

ii. The resulting spatial dimensions are 32×32, so we have preserved the same size as the input image.

iii. The resulting outputs are $32 \times 32 \times 10$, with 10 output channels. There are $5 \cdot 5 \cdot 3 = 75$ parameters per filter, so with 10 filters, we have 750 parameters in this layer. Note that, if we did choose to include a bias parameter, then there would be 76 parameters per filter, and so 760 in total.

iv. A 1×1 convolution does not change the spatial dimensions. For every spatial location, it performs a linear map of the the input channels pointwise over space. In practice, this is useful for changing the number of channels.

(b) (Convolutions as Matrix Multiplication) We note that convolutions are a linear operation. Recalling linear algebra, any linear map (between finite-dimensional spaces) can be expressed as a matrix, so we will see in this section how to write a convolution as a matrix multiplication.

Problem 2: Expressing convolutions as matrix multiplication

We shall again consider a 1D convolution. Consider an input $\mathbf{x} \in \mathbb{R}^4$ and filter $\mathbf{w} \in \mathbb{R}^3$. Letting $\bar{\mathbf{x}}$ denote the result of zero-padding the input by 1 on each end, what is the matrix W such that

$$\overbrace{W}^{\mathbb{R}^{4\times 6}} \overbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}^{\text{Zero padded input } \bar{\mathbf{x}} \in \mathbb{R}^6} = \bar{\mathbf{x}} * \mathbf{w}?$$

Solution:

Solution 2: Expressing convolutions as matrix multiplication

Computing the convolution, we see that

$$\bar{\mathbf{x}} * \mathbf{w} = \begin{bmatrix} x_1 w_2 + x_2 w_3 \\ x_1 w_1 + x_2 w_2 + x_3 w_3 \\ x_2 w_1 + x_3 w_2 + x_4 w_3 \\ x_3 w_1 + x_4 w_2 \end{bmatrix}.$$

Writing this out as a matrix multiplication, we obtain

$$W = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix}.$$

We can observe now that the resulting matrix will be very sparse (most entries are 0) if the filter size is much smaller than the input size, corresponding to the fact that such convolutions exploit spatial locality. We also observe that there is a lot of parameter reuse, as the convolutional filter weights are repeated many times throughout the explicit matrix.

This has several implications. First of all, this implies that convolutional layers are less expressive than fully-connected layers (as fully connected layers are represented by arbitrary matrices).

Another important implication stems from the fact that we have very optimized tools for computing matrix multiplications. While a naive implementation of a convolution will require looping over all the spatial dimensions, it will turn out that reformulating the convolution as a matrix multiplication will often be much faster due to these optimizations (for example, the Cythonized im2col function in part 4 of homework 1 essentially does this).

(c) (Backwards Pass for a Convolution) We'll consider the same 1D convolution as before, but without zero-padding for simplicity.

Problem 3: Backwards pass for convolutions

Let $\mathbf{y} = \mathbf{x} * \mathbf{w} \in \mathbb{R}^2$, where $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{x} \in \mathbb{R}^4$. Let $\boldsymbol{\nabla}_{\mathbf{y}} L$ denote the gradient of the loss with respect to the output of the convolution. Compute the gradients of L with respect to $\mathbf{x}$ and $\mathbf{w}$. Can you express the gradients as convolutions themselves?

Solution:

Solution 3: Backwards pass for convolutions

Let $\delta_i = \frac{\partial L}{\partial y_i}$. We can explicitly write out the partial derivatives with respect to each entry of $\mathbf{x}$.

$$\frac{\partial L}{\partial x_1} = w_1\delta_1$$
$$\frac{\partial L}{\partial x_2} = w_2\delta_1 + w_1\delta_2$$
$$\frac{\partial L}{\partial x_3} = w_3\delta_1 + w_2\delta_2$$
$$\frac{\partial L}{\partial x_4} = w_3\delta_2$$

We recognize this as convolution where we zero pad δ by 2 on each end, and convolve with the filter $\tilde{\mathbf{w}}$, where $\tilde{\mathbf{w}}$ reverses the entries of the filter $\mathbf{w}$. (Draw this out for students, explain why sliding the filter along means that we should convolve the output derivative with $\tilde{\mathbf{w}}$ instead of $\mathbf{w}$).

Now, we can similarly compute the partial derivatives for $\mathbf{w}$

$$\frac{\partial L}{\partial w_1} = \delta_1 x_1 + \delta_2 x_2$$
$$\frac{\partial L}{\partial w_2} = \delta_1 x_2 + \delta_2 x_3$$
$$\frac{\partial L}{\partial w_3} = \delta_1 x_3 + \delta_2 x_4$$

We see that $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{x} \star \boldsymbol{\nabla}_{\mathbf{y}} L$ with no zero-padding.

(d) (**backpropagation through a kernel**) The above rewriting allows us to use the same method to backpropagate through a convolution layer as if it is an affine layer. However, this *as-if* fails for the parameters of the kernel itself, because of weight sharing. That is, in the equivalent affine layer, some entries are forced to always remain the same. This question derives the gradients of the loss with respect to the kernel weights.

Let's consider a convolution layer with input matrix $\mathbf{X} \in \mathbb{R}^{n \times n}$,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}, \tag{1}$$

weight matrix $\mathbf{w} \in \mathbb{R}^{k \times k}$,

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,k} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & \cdots & w_{k,k} \end{bmatrix}, \tag{2}$$

and output matrix $\mathbf{Y} \in \mathbb{R}^{m \times m}$,

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,m} \end{bmatrix}. \tag{3}$$

For simplicity, we assume the number of the input channel (of $\mathbf{X}$ is) and the number of the output channel (of output $\mathbf{Y}$) are both 1, and the convolutional layer has no padding and a stride of 1. Then for all i, j,

$$y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}, \tag{4}$$

or

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}, \tag{5}$$

, where $*$ refers to the convolution operation. For simplicity, we omitted the bias term in this question. Suppose the final loss is $\mathcal{L}$, and the upstream gradient is $d\mathbf{Y} \in \mathbb{R}^{m,m}$,

$$d\mathbf{Y} = \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix}, \tag{6}$$

where $dy_{i,j}$ denotes $\dfrac{\partial \mathcal{L}}{\partial y_{i,j}}$.

Now, we **derive the gradient to the weight matrix** $d\mathbf{w} \in \mathbb{R}^{k,k}$,

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}, \tag{7}$$

where $dw_{h,l}$ denotes $\frac{\partial \mathcal{L}}{\partial w_{h,l}}$. Also, **derive the weight after one SGD step with a batch of a single image**.

Solution: This part is heavily inspired by Backpropagation in a convolutional layer.
The forward propagation rule is

$$y_{i,j} = \sum_{h=1}^{k}\sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}. \tag{8}$$

Use chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{h,l}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial w_{h,l}}, \tag{9}$$

and

$$\frac{\partial y_{i,j}}{\partial w_{h,l}} = x_{i+h-1,j+l-1}, \tag{10}$$

so we have

$$dw_{h,l} = \sum_{i=1}^{m}\sum_{j=1}^{m} x_{i+h-1,j+l-1} dy_{i,j} \tag{11}$$

$$d\mathbf{w} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix} * \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix} \tag{12}$$

$$= \mathbf{X} * d\mathbf{Y}. \tag{13}$$

For one SGD step with learning rate η, suppose the input and output are $\mathbf{X}$ and $\mathbf{Y}$, then

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta d\mathbf{w} = \mathbf{w_t} - \eta \mathbf{X} * d\mathbf{Y} \tag{14}$$

We can conclude that **during the training of CNN weights with SGD, the weights of a CNN are a weighted average of images patches in the dataset**. This is because the gradient, which is a weighted average of image patches in the dataset, is used to update the weights in SGD, so the trained weight is the initial weight plus a weighted average of the gradients.

(e) A **maxpooling** layer has 2 architectural hyperparameters: the stride step size(S) and the "filter size" (K). The maxpooling operation takes the maximum value in each $K \times K$ window of the input, and strides by S pixels each time. See Figure 1 for an example of maxpooling with $K = 2, S = 2$.

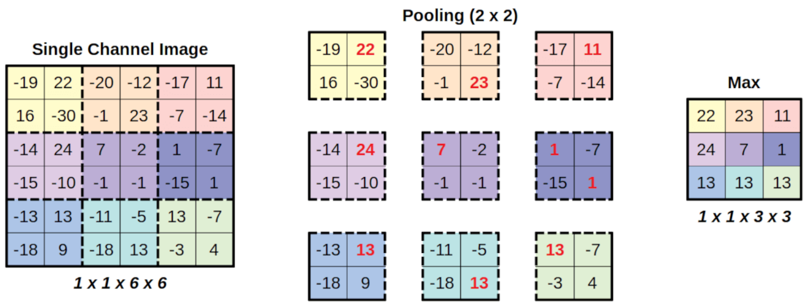

Figure 1: Example of maxpooling when $K = 2, S = 2$.

What is the output feature shape that this pooling layer produces?

Solution:
$W' = (W - K)/S + 1$
$H' = (H - K)/S + 1$
$C' = C$

(f) **For a network with only 2x2 max-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X} = [dx_{i,j}] = [\frac{\partial \mathcal{L}}{\partial x_{i,j}}]$? For a network with only 2x2 average-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X}$?**

HINT: Start with the simplest case first, where $\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$. *Further assume that top left value is selected by the max operation. i.e.*

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{15}$$

Then generalize to higher dimension and arbitrary max positions.

Solution:
In the simplest case, output $\mathbf{Y}$ has size 1x1. For the max pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } (i,j) = 1,1 \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

Combining all four partial derivatives, we have

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} dy_{11} & 0 \\ 0 & 0 \end{bmatrix} \tag{17}$$

For the average pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = 1/4 \tag{18}$$

$$d\mathbf{X} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \\ \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \end{bmatrix} \tag{19}$$

In the general setting, for max pooling, we can notice that x and y have a one to one mapping. Each x value is involved in calculation of exactly one y value. Let $k = i//2$, $l = j//2$, where $//$ performs the floordiv operation, let

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } x_{i,j} = \max(x_{i,j}, x_{i+1,j}, x_{i,j+1}, x_{i+1,j+1}) \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

, then

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} \tag{21}$$

$d\mathbf{X}$ is the matrix contructed by each $dx_{i,j}$. It will have similar pattern to the simplest case. For each 2x2 block, only one of the input pixel has gradient of magnitude one flowing back, and the rest three inputs have zero gradient.

For the average pooling general case,

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \frac{1}{4} \tag{22}$$

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} = \frac{dy_{kl}}{4} \tag{23}$$

Average pooling distributes the gradient evenly across each 2x2 input blocks.

(g) BatchNorm for CNNs is a bit different from BatchNorm for fully connected layers. The idea is that because CNN must treat a picture in the same way even if we shift the picture, we also should treat the neural activations inside the CNN the same way even if we shift the neurons. In other words, we treat the many outputs from a single convolutional kernel on a single picture as if they come from the same minibatch. That is, we apply BatchNorm per-channel, across all locations and all pictures in the minibatch.

```python
import numpy as np

def batchnorm(x, gamma, beta, epsilon=1e-8):
    # Mean and variance of each feature
    mu = np.mean(x, axis=0)  # shape (N,)
    var = np.var(x, axis=0)  # shape (N,)

    # Normalize the activations
    x_hat = (x - mu) / np.sqrt(var + epsilon)  # shape (B, N)

    # Apply the linear transform
    y = gamma * x_hat + beta  # shape (B, N)

    return y
```

```
def batchnorm_cnn(x, gamma, beta, epsilon=1e-8):
    # Calculate the mean and variance for each channel.
    mean = np.mean(x, axis=(0, 1, 2), keepdims=True)
    var = np.var(x, axis=(0, 1, 2), keepdims=True)

    # Normalize the input tensor.
    x_hat = (x - mean) / np.sqrt(var + epsilon)

    # Scale and shift the normalized tensor.
    y = gamma * x_hat + beta

    return y

# Alternative implementation using reshape
# Since it is just a special case of batchnorm for cnn
def batchnorm_cnn(x, gamma, beta, epsilon=1e-8):
    B, H, W, C = x.shape
    x_reshaped = x.reshape(B * H * W, C)
    y = batchnorm(x_reshaped, gamma, beta, epsilon)
    y = y.reshape(B, H, W, C)
    return y
```

Given this, how do we implement the backward pass for BatchNorm in a CNN?

Solution: The backward pass for BatchNorm in a CNN is similar to the backward pass for BatchNorm in a fully connected layer. It suffices to reshape the input (B, H, W, C) tensor to (BHW, C), apply the BatchNorm backward pass, and then reshape the output back to the original shape.

2. Recurrent Neural Network The world is full of sequential information, from video to language modelling to time series data. In particular, we would like to model these sequences using neural networks, and solve some major types of tasks that we would like to solve with sequence models.

0.2 Types of Problems

- **One-to-one** problems take a single input x and produce a single output y. Problems like classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output) fall under this category.
- **One-to-many** problems take a single input, and produce a sequence of output. Problems like image captioning (takes a single image as input, and produces a caption (a sequence of words) as output) fall under this category.
- **Many-to-many** problems take sequences of inputs and produce sequences of outputs. Problems like language translation (sequence of words in one language to sequence of words in another) fall under this category

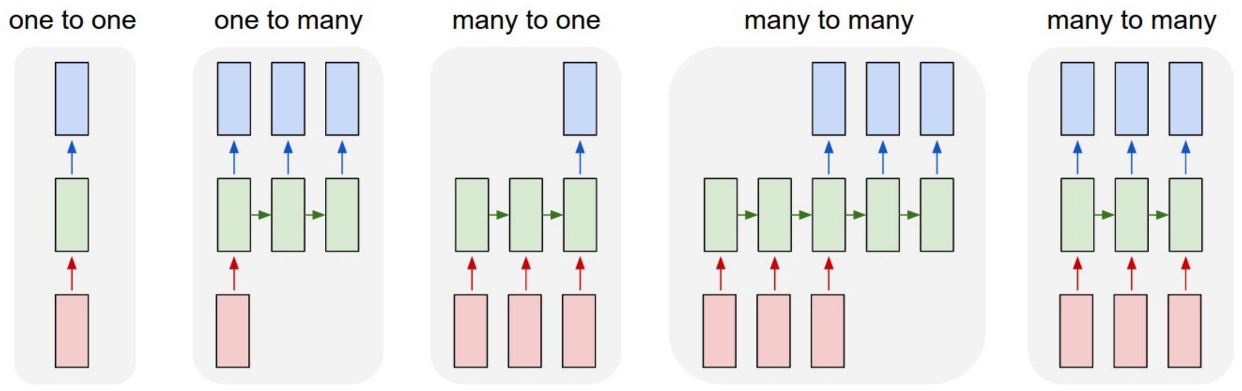

Figure 2: Types of problems we would like to solve using sequential models

0.3 Why the Recurrence?

As you read through this discussion worksheet, you don't process each word entirely on its own, but instead use your understanding from the previous words as well. Traditional neural networks do not have the capability to use its reasoning about previous events to infer later ones. For example, if we would like to classify what is happening at every frame in a movie, this can be framed as an image classification task where the network is provided the current image. However, it is unclear how a traditional neural network should incorporate knowledge from the previous frames in the film to inform later ones.

Recurrent neural networks (RNNs) address this issue, by using the idea of "recurrent connections." RNNs are networks with loops in them that allow information from previous inputs to persist as the network processes the future inputs. These recurrent connections allow information to propagate from "the past" (earlier in the sequence) to the future (later in the sequence).

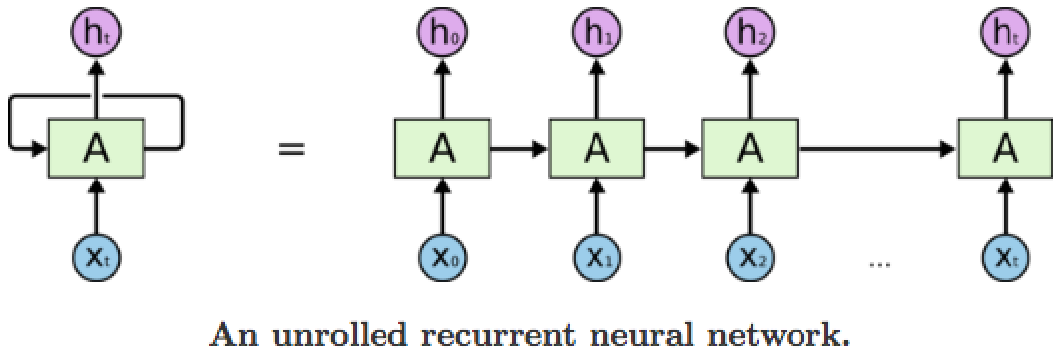

Figure 3: An example of a generic recurrent neural network. This shows how to "unroll" a network through time - instead of thinking about sequence modeling as a single network with shared weights

In Figure 3, we illustrate the RNN computation as it is unrolled through time. Each $i \in \{0, \ldots, t\}$ represents a new timestep in the network. By feeding in a state computed from earlier timesteps as an input together with the current input, information can persist throughout the time as the network "remembers" the past inputs it processed.

0.4 Vanilla RNN

In the following section, we will use the following notation. Denote the input sequence as $x_t \in \mathbb{R}^k$ for $t \in \{1, \dots, T\}$, and output of the network be $y_t \in \mathbb{R}^m$ for $t \in \{1, \dots, T\}$. In the following example, we construct a”vanilla" many-to-many RNN, consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$\begin{aligned} h_t &= \tanh(W_{h,h}h_{t-1} + W_{x,h}x_t + B_h) \\ y_t &= W_{h,y}h_t + B_y \end{aligned}$$

where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{\cdot,\cdot}$ be the set of weights (for example, $W_{x,h}$ represents weight matrix that accepts an input vector and produce a new hidden state), y_t be the output at timestep t and B_h and B_y be the bias terms. We can also represent it as the diagram below,

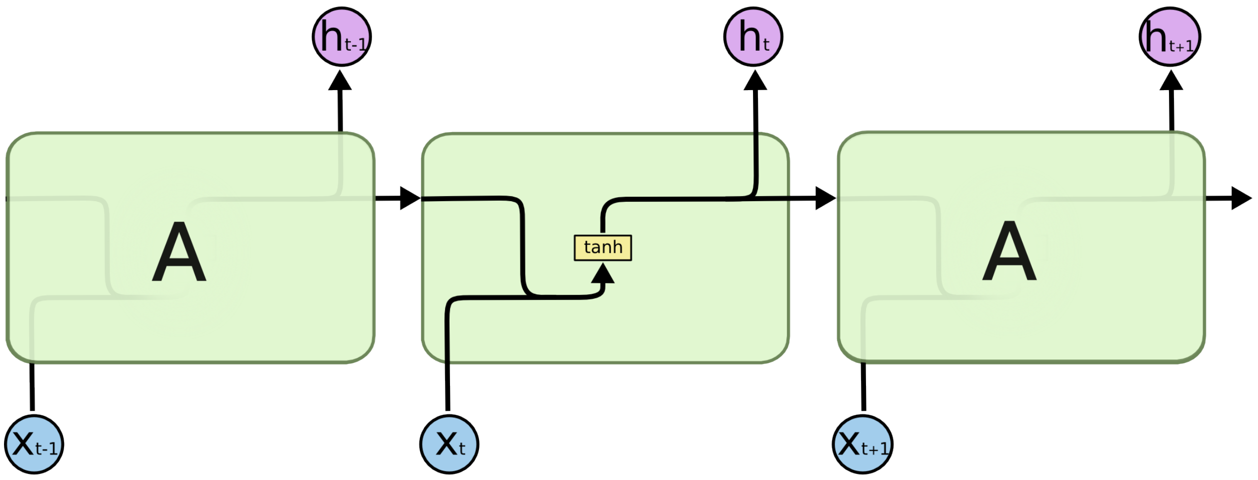

Figure 4: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t-1$ to time t

In this vanilla RNN, we update to a hidden state "h_t" based on the previous hidden state h_{t-1} and input at the current time x_t, and produce an output which that is a simple affine function of the hidden state. To compute the forward (and backward) passes of the network, we have to "unroll" the network, as shown in Figure 3. This "unrolling" process creates something that resembles a very deep feed forward network (with depth corresponding to the length of the input sequence), with shared affine parameters at each layer. Our gradient is computed by summing the losses from each time-step of the output.

Problem: Gradients in Vanilla RNN

Why are vanishing or exploding gradients an issue for RNNs?

Solution:

Solution: Gradients in Vanilla RNN

A major issue with the vanilla RNN is that they suffers from vanishing/exploding gradients similarly to issues with deep feedforward networks. At each timestep, the hidden state h_t is multiplied by W, at the last timestep, the value of h_t is effectively multiplied by $W^\top$. This means that depending on the singular values of the matrix W, the gradients of the loss with respect to W may become very large or very small as they pass back down the unrolled network. Additionally, the tanh activation at each step can also contribute to the vanishing gradient problem.

Problem: Coding RNNs Up!

Complete the class definition, started for you below,

```
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3)
        self.By = np.random.randn(3)
        self.hidden_state = np.zeros(3)

    def forward(self, x):
        # Processes the input at a single timestep and
        # updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```

Solution:

Solution: Coding RNNs Up!

```
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3,)
        self.By = np.random.rand(3,)

    def forward(self, x):
        self.hidden_state = np.tanh(np.dot(self.hidden_state, self.W_hh) \
                                    + np.dot(x, self.W_xh) + self.Bh)
        self.out = np.dot(self.W_hy, self.hidden_state) + self.By
        return self.output
```