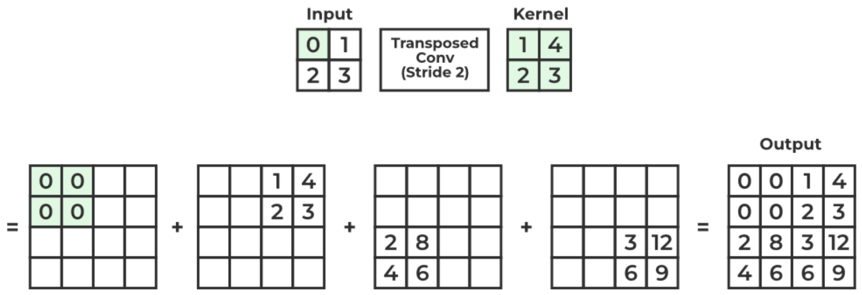
EECS 182 Fall 2025 — Deep Neural Networks — Anant Sahai and Gireeja Ranade

Discussion 8

1. Transpose Convolutions

Recall that a transpose convolution operation increase the spatial dimensions (upsamples) of an input feature map to produce a larger output, as shown in the following figure:



We will practice performing transpose convolutions on a 1D discrete signal $\mathbf{x} = [x_1, x_2, x_3]^T$.

(a) Consider the a transpose convolution with size-1 kernel $\mathbf{k} = [k]$. **Find the result of a transpose convolution of $\mathbf{x}$ with $\mathbf{k}$ with stride 1 and no padding. Then, repeat with stride 2 and stride 3 (again with no padding).**

(b) Now, consider a size-3 kernel $\mathbf{k} = [k_1, k_2, k_3]^T$. **Find the result of a transpose convolution of $\mathbf{x}$ with $\mathbf{k}$ using stride 1 and no padding. Repeat with stride 3.**

2. Setting up sequence prediction as RNN

The idea of recurrent information processing is quite basic and predates deep neural networks. Consider the following formulation:

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t \tag{1}$$

$$\mathbf{y}_t = \mathbf{C}\mathbf{x}_t + \mathbf{w}_t \tag{2}$$

This recurrent form has $\mathbf{x}_t$ denoting the (soon to be hidden) state at timestep t, $\mathbf{y}_t$ is the measurement (or label), $\mathbf{u}_t$ is some driving noise (assume it to be zero-mean iid Gaussian) at timestep t, and $\mathbf{w}_t$ is the similarly zero-mean iid Gaussian observation noise at each timestep. Here $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are the weights that determine the state evolution and observations at **all** time steps.

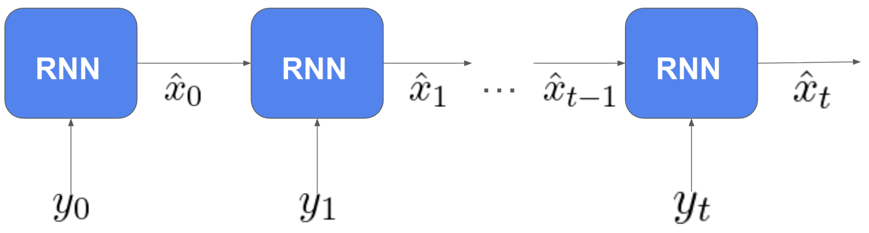
(a) In the case where all weights $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are just matrices and we have access to full trajectories of $(\mathbf{x}_t, \mathbf{u}_t, \mathbf{y}_t)$, **how can you setup the problem of learning $\mathbf{A}, \mathbf{B}, \mathbf{C}$ from this training data?**

(b) Assume that the $\mathbf{u}_t = \mathbf{0}$ for simplicity. We know from Kalman filtering that if we know the $\mathbf{A}, \mathbf{B}, \mathbf{C}$ matrices, it is possible to create a steady-state filter which approximately tracks that state $\mathbf{x}_t$ from (1) given only the sequence of observations $\mathbf{y}_t$ as long as the (A, C) matrix is observable — note "observability" here is a technical condition that is dual to "controllability" and says that $[C, CA, CA^2, \ldots]$ is full rank. Such a filter can be given in recursive form as:

$$\widehat{\mathbf{x}}_t = \mathbf{A}'\widehat{\mathbf{x}}_{t-1} + \mathbf{B}'\mathbf{y}_t \tag{3}$$

Now, suppose we actually had training-time access to full trajectories of $(\mathbf{x}_t, \mathbf{y}_t)$ and want to learn $\mathbf{A}', \mathbf{B}'$ as in (3). **Comment on why simply using the approach of the previous part does not work and we can't use the ground-truth $\mathbf{x}_t$ to break the long chain of dependency in the state evolution of the filter across the sequence in the way that we were able to do for simple system identification with perfectly observed states.**

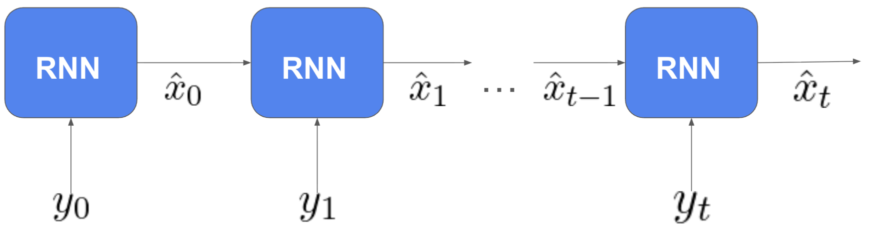
(c) Continuing the previous part (continue to assume that the $\mathbf{u}_t = \mathbf{0}$ for simplicity), **how can you train on full trajectories of $(\mathbf{x}_t, \mathbf{y}_t)$ in order to learn $\mathbf{A}', \mathbf{B}'$ from (3)?** Also, **modify the computation graph below to show what extra computations are required to compute the loss function.**



Note: in this case, for simplicity, do not attempt to setup any loss function involving trying to reconstruct the $\mathbf{y}_t$. Simply treat these as known inputs for this part.

While the filter doesn't see $\mathbf{x}_t$ at training time, we can define a loss function on how close $\widehat{\mathbf{x}}_t$ is to $\mathbf{x}_t$, and optimize the loss function to learn $\mathbf{A}'$ and $\mathbf{B}'$.

(d) Now consider the case where we only have measurements of $\mathbf{y}_t$ for all time steps. (Continue to assume that the $\mathbf{u}_t = \mathbf{0}$ for simplicity.) This is common in applications such as **object tracking**, where you only have a series of video frames, and the model has to track the movement of an object throughout the frames. **First, show how an RNN model can be setup to predict the next $\mathbf{y}_{t+1}$ in the sequence and how this RNN can be trained.** Afterwards, **modify the same computation graph to include the new loss function.**



(e) To simplify computation, now assume that $\mathbf{w}_t = \mathbf{0}$. Given that the sequence length is T (input: $(\mathbf{u}_1, \cdots, \mathbf{u}_T)$, output: $(\mathbf{y}_1, \cdots, \mathbf{y}_T)$) and assuming the initial state $\mathbf{x}_0 = 0$, **show that the output $\mathbf{y}_t$ can be expressed as a *convolution* of the input sequence $\{\mathbf{u}_i\}_1^T$ with a kernel $K = \{K_i\}_0^T$:**

$$\mathbf{y}_t = \sum_{i=0}^{T} K_{t-i}\mathbf{u}_i,$$

where any K_j for $j < 0$ is set to $\mathbf{0}$ to ensure causality. **Find K.**

3. Autoencoders

If the type of autoencoder is not specified, you may consider all autoencoder types (vanilla, denoising, masked, etc.)

(a) **How can you do validation for autoencoder training?** Think about how we traditionally split our training set into a train and validation set. How can you use your validation set?

(b) If you train two different autoencoder variants on the same dataset, **is it true that the one which produces lower validation loss will necessarily perform better on the downstream task?**

(c) After training the autoencoder, the learned representations are often used for downstream learning. When doing so, **what part of the autoencoder is used? The encoder, decoder, or both?**

(d) Using the representations by a learned autoencoder (rather than using raw inputs) is most useful when you have only a few data samples for your downstream task. **Is this true?**

(e) We can think of masked and denoising autoencoders as vanilla autoencoders with data augmentation applied. **Is this true?**

(f) If you trained an autoencoder with noise or masking, you should also apply noise/masking to inputs when using the representations for downstream tasks. **Is this true?**

(g) Autoencoders always encode inputs into fixed-size lower-dimensional representations. **Is this true?**

Contributors:

- Lance Mathias.
- Kevin Li.
- Anant Sahai.
- Naman Jain.
- Olivia Watkins.