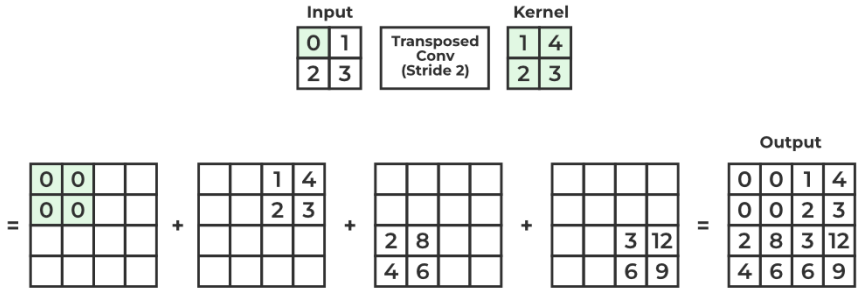
EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

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).**

Solution: Without padding and stride 1, the output is simply $\mathbf{y} = [kx_1, kx_2, kx_3]^T$, which is just a scaled version of the input.

With stride 2, the output is $\mathbf{y} = [kx_1, 0, kx_2, 0, kx_3]^T$, which looks like a scaled version of the input with zeros inserted between each entry.

With stride 3, the output is $\mathbf{y} = [kx_1, 0, 0, kx_2, 0, 0, kx_3]^T$. Again, this is a scaled version of the input, but this time with more zeros inserted between each entry.

(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.**

Solution: With stride 1, the output is the following length-5 vector:

$$\mathbf{y} = \begin{bmatrix} k_1x_1 \\ k_2x_1 + k_1x_2 \\ k_3x_1 + k_2x_2 + k_1x_3 \\ k_3x_2 + k_2x_3 \\ k_3x_3 \end{bmatrix}$$

With stride 3, the output is the following length-9 vector:

$$\mathbf{y} = \begin{bmatrix} k_1x_1 \\ k_2x_1 \\ k_3x_1 \\ k_1x_2 \\ k_2x_2 \\ k_3x_2 \\ k_1x_3 \\ k_2x_3 \\ k_3x_3 \end{bmatrix}$$

Note that a transpose convolution with stride s can be thought of as inserting $s-1$ zeros between each entry of the input, then performing a standard convolution with the transpose kernel. However, in order for the shapes to work out, in the stride 3 case, it is necessary to add zero-padding of 2. Concretely, we upsample x to get

$$\mathbf{x'} = [0, 0, x_1, 0, 0, x_2, 0, 0, x_3, 0, 0]^T$$

Then, we can convolve $\mathbf{x'}$ with the transpose kernel $\mathbf{k}^T$ to get the same result as above.

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?**

Solution:

When the entire trajectories are given, the problem is solvable with least squares. Both the ordinary least squares and ridge regression could work depending on the data.

Particularly, we learn A,B by least squares:

$$\begin{bmatrix} x_2^T \\ x_3^T \\ \vdots \\ x_T^T \end{bmatrix} = \begin{bmatrix} x_1^T & u_1^T \\ x_2^T & u_2^T \\ \vdots & \vdots \\ x_{T-1}^T & u_{T-1}^T \end{bmatrix} \begin{bmatrix} A^T \\ B^T \end{bmatrix}.$$

Learn C similarly:

$$\begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_T^T \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_T^T \end{bmatrix} C^T.$$

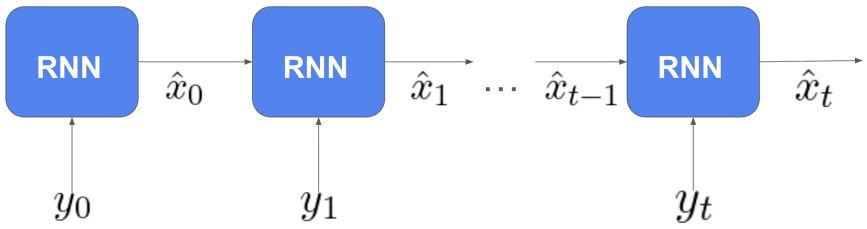
(b) Assume that the $\mathbf{u}_t = \mathbf{0}$ for simplicity. We know from Kalman filtering that if we know the $\mathbf{A}, \mathbf{B}, \mathbf{C}$ matrices, it is possible to create a steady-state filter which approximately tracks that state $\mathbf{x}_t$ from (1) given only the sequence of observations $\mathbf{y}_t$ 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.**

Solution: In the fully-observed system-ID problem of part (1), we simply used the known $\mathbf{x}_t$ values at each time step. However, for the filter we cannot simply put $\widehat{\mathbf{x}}_t$ in place of $\mathbf{x}_t$ in the training data. This is because the filter is a recursive model that depends on the previous state $\widehat{\mathbf{x}}_{t-1}$ to predict the next state $\widehat{\mathbf{x}}_t$.

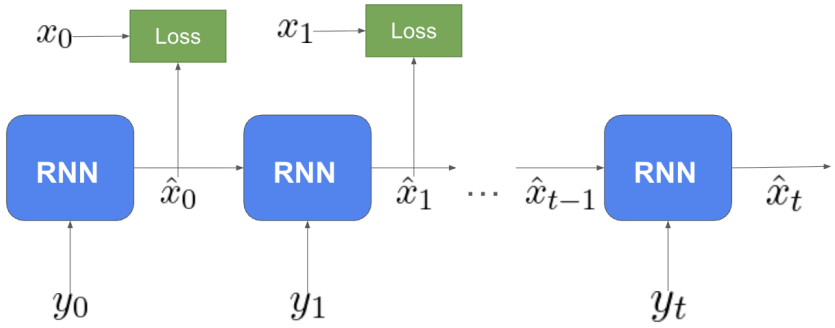
(c) Continuing the previous part (continue to assume that the $\mathbf{u}_t = \mathbf{0}$ for simplicity), **how can you train on full trajectories of $(\mathbf{x}_t, \mathbf{y}_t)$ in order to learn $\mathbf{A'}, \mathbf{B'}$ from (3)?** Also, **modify the computation graph below to show what extra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



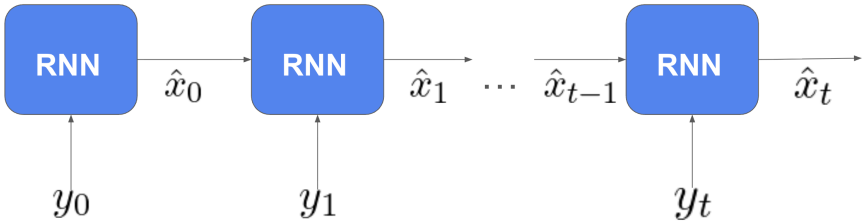
Note: in this case, for simplicity, do not attempt to setup any loss function involving trying to reconstruct the $\mathbf{y}_t$. Simply treat these as known inputs for this part.

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 $\mathbf{B'}$.

Solution: At every timestep, we can train the filter to minimize loss $||\mathbf{x}_t - \widehat{\mathbf{x}}_t||_2^2$. This bypasses the issue in the previous part because we generate $\widehat{\mathbf{x}}_t$ recursively using $\widehat{\mathbf{x}}_{t-1}$.



(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.**



Solution:

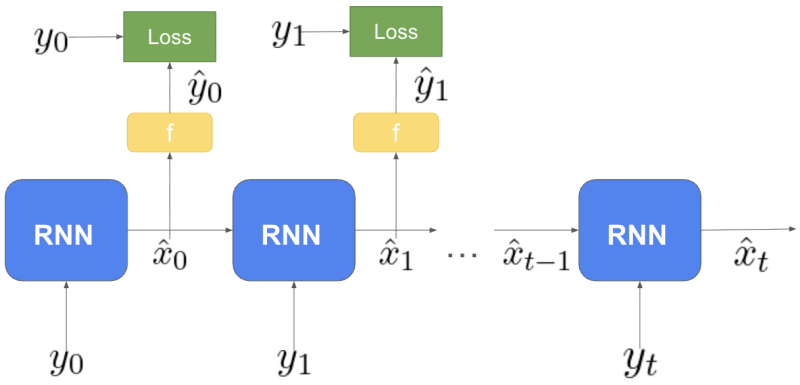
Since we assumed that the weights are timestep-invariant, RNN is a suitable model to learn the transitions given that both $\mathbf{u}_t$ and $\mathbf{x}_t$ are unknown. As we see in part (a), $\mathbf{y}_t$ is dependent on $\mathbf{u}_{t-1}$ and $\mathbf{x}_{t-1}$, we can rewrite the transition as the follows:

$$\mathbf{y}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}) + \mathbf{w}_t$$

where f is a neural network that determines the transition. Here all we know is about $\mathbf{y}_t$. We can manipulate the algebra to show the dependency between $\mathbf{y}_t$ to make it into a one-step prediction setup:

$$\begin{aligned}
\mathbf{y}_t &= \mathbf{C}\mathbf{x}_t + \mathbf{w}_t \\
&= \mathbf{C}(\mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_{t-1}) + \mathbf{w}_t \\
&= \mathbf{C}(\mathbf{A} \cdot (\mathbf{C}^{-1}(\mathbf{y}_{t-1} - \mathbf{w}_{t-1})) + \mathbf{B}\mathbf{u}_{t-1}) + \mathbf{w}_t \\
&= \mathbf{C}\mathbf{A}\mathbf{C}^{-1}\mathbf{y}_{t-1} + \mathbf{C}\mathbf{B}\mathbf{u}_{t-1} + \text{noise}
\end{aligned}$$

which shows the dependency of $\mathbf{y}_t$ and the label at the previous timestep $\mathbf{y}_{t-1}$. This leads to the typical training paradigm of RNN, as depicted in the diagram below.



(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.**

Solution: From the recursion $x_{t+1} = Ax_t + Bu_t$, we get

$$x_1 = Bu_0, \quad x_2 = ABu_0 + Bu_1, \quad x_3 = A^2Bu_0 + ABu_1 + Bu_2, \ldots$$

Hence,

$$y_0 = 0, \tag{4}$$
$$y_1 = CBu_0, \tag{5}$$
$$y_2 = CBu_1 + CABu_0, \tag{6}$$
$$y_3 = CBu_2 + CABu_1 + CA^2Bu_0, \tag{7}$$
$$\vdots \tag{8}$$

each y_t is given explicitly by summing all terms of the form $CA^{t-i-1}Bu_i$. This summation matches a discrete convolution $\{y\} = K * \{u\}$ with kernel K given by

$$K = (0, CB, CAB, CA^2B, \ldots).$$

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?

Solution: To check validation for autoencoder training, we do the same training and validation split and compare the reconstruction loss on training and validation sets. Checking performance on a validation set is useful for sanity checking that the model is training correctly (val loss should go down), diagnosing overfitting (big gap between the validation loss and the training loss), and early stopping (figuring out which training checkpoint to actually use).

(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?**

Solution: This is not always true. For instance, if you add noise or masking or make the bottleneck smaller, reconstruction loss may go up, but these could still produce more useful representations.

(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?**

Solution: We only use the encoder for downstream training.

(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?**

Solution: True: Unsupervised pretraining provides the largest gains when you have little task-specific data (and are therefore at risk of overfitting), though it can still help somewhat even when plenty of data is available.

(e) We can think of masked and denoising autoencoders as vanilla autoencoders with data augmentation applied. **Is this true?** **Solution:** True: like other data augmentations, these can help prevent overfitting and produce more robust representations. Masking, however, sometimes involves additional changes to the training procedure which are not used with other augmentations (e.g. some transformer architectures completely remove masked tokens from the encoder.)

(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?** **Solution:** False: this is uncommon (though it is done occasionally when you want data augmentation on the downstream task). Like many regularizers (like data augmentation and dropout), the modification is applied during training only.

(g) Autoencoders always encode inputs into fixed-size lower-dimensional representations. **Is this true?** **Solution:** False. Masked and denoising autoencoder representations may not be lower-dimensional, and representations can be variable-sized (e.g. a transformer's representation contains one vector for each input token.

Contributors:

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

