EECS 182/282A

Today: RNNs
Self-supervision

Reading: Prince through Ch 11 + Ch 13
Note: Little in this lecture is in the Prince textbook. (Just 9.3.7)
Older RNN material: deeplearningbook.org
Project Info Released: Start Forming Teams!

Architecture Order In Class:

MLPs → CNNs → Graph NN → $\boxed{\text{RNN/State-space}}$ → Transformers

RNNs: Recurrent Neural Nets

History: Before 2018, concensus view: Two big successes in NN:
Images & Vision using CNNs
Language & Speech using RNNs

Key issue in language & speech (along with control, time-series, etc.) sequential data

A signal-processing perspective

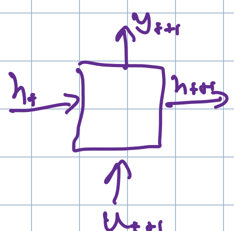
SP	Neural Nets	Weight-sharing
FIR Filter (finite impulse response)	CNN	Across Space
IIR Filter (infinite impulse response) Finite-dimensional hidden state inside filter. Sequential & Causal	RNN	Across Time

Inside an IIR filter from $\vec{u}_t \to \vec{y}_t$, there is a hidden state $\vec{h}_t$. (Initialized to zero)

e.g.

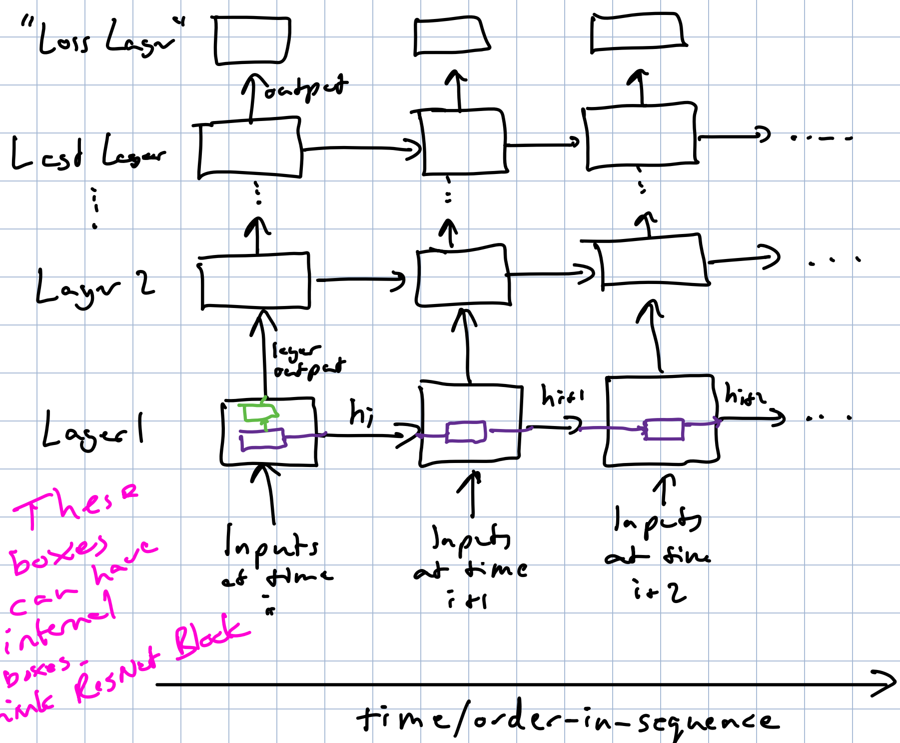
$$\vec{h}_{t+1} = W_h \vec{h}_t + B\vec{u}_{t+1} \quad \text{(Update State Causally)}$$

$$\vec{y}_{t+1} = C\vec{h}_{t+1} + D\vec{u}_{t+1} \quad \text{(Generate Filter Output)}$$



Treat this box as the counterpart of a conv for sequence problems.

Approach: Keep all the design ideas & principles from CNNs.
Just replace space with time and enforce causality.



These boxes can have internal boxes - Think ResNet Block

Many Ideas Just Carry Over
Vertical Residual Connections

Vertical "1x1" MLPs

Vertical Normalizations
Note: Can't average with the future
Just local channels

Key Design Question:

Where do the nonlinearities go?

Note: Use vertical direction for layers
Use horizontal for "time"

Traditional RNN perspective: Put a nonlinearity in the state update

Start with linear dynamics:

$$\vec{h}_{t+1} = W_h \vec{h}_t + B\vec{u}_{t+1} + \vec{b}_h$$
$$\vec{y}_{t+1} = C\vec{h}_{t+1} + \vec{b}_y$$

biases

nonlinearity. typically sigmoid, tanh, etc.

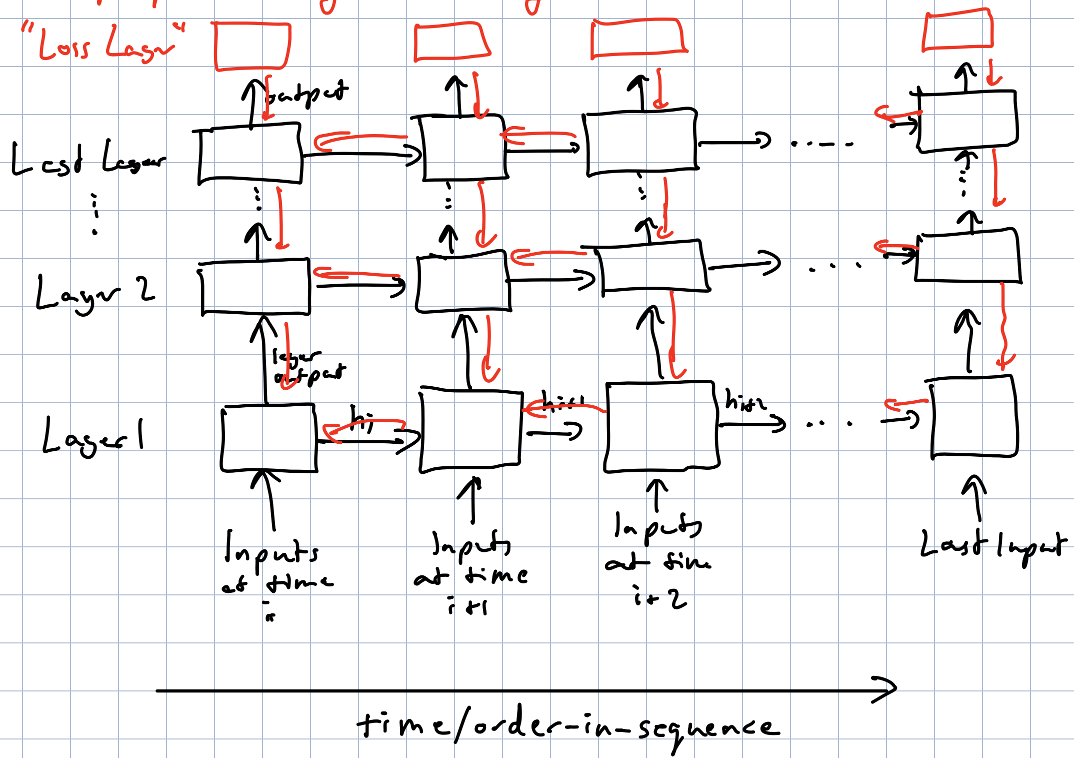
Add a nonlinearity

$$\vec{h}_{t+1} = \sigma\left(W_h \vec{h}_t + B\vec{u}_{t+1} + \vec{b}_h\right)$$
$$\vec{y}_{t+1} = C\vec{h}_{t+1} + \vec{b}_y \quad \leftarrow \text{Linear Layer}$$

Traditional "Vanilla" RNNs treat the hidden state as the layer's output.

A reason why traditional RNNs use saturating nonlinearities is that this blocks exploding activations across time.

Backprop during training:



Long Paths for Backprop means that gradients can die out going that far.
Saturating Nonlinearities makes this even worse.

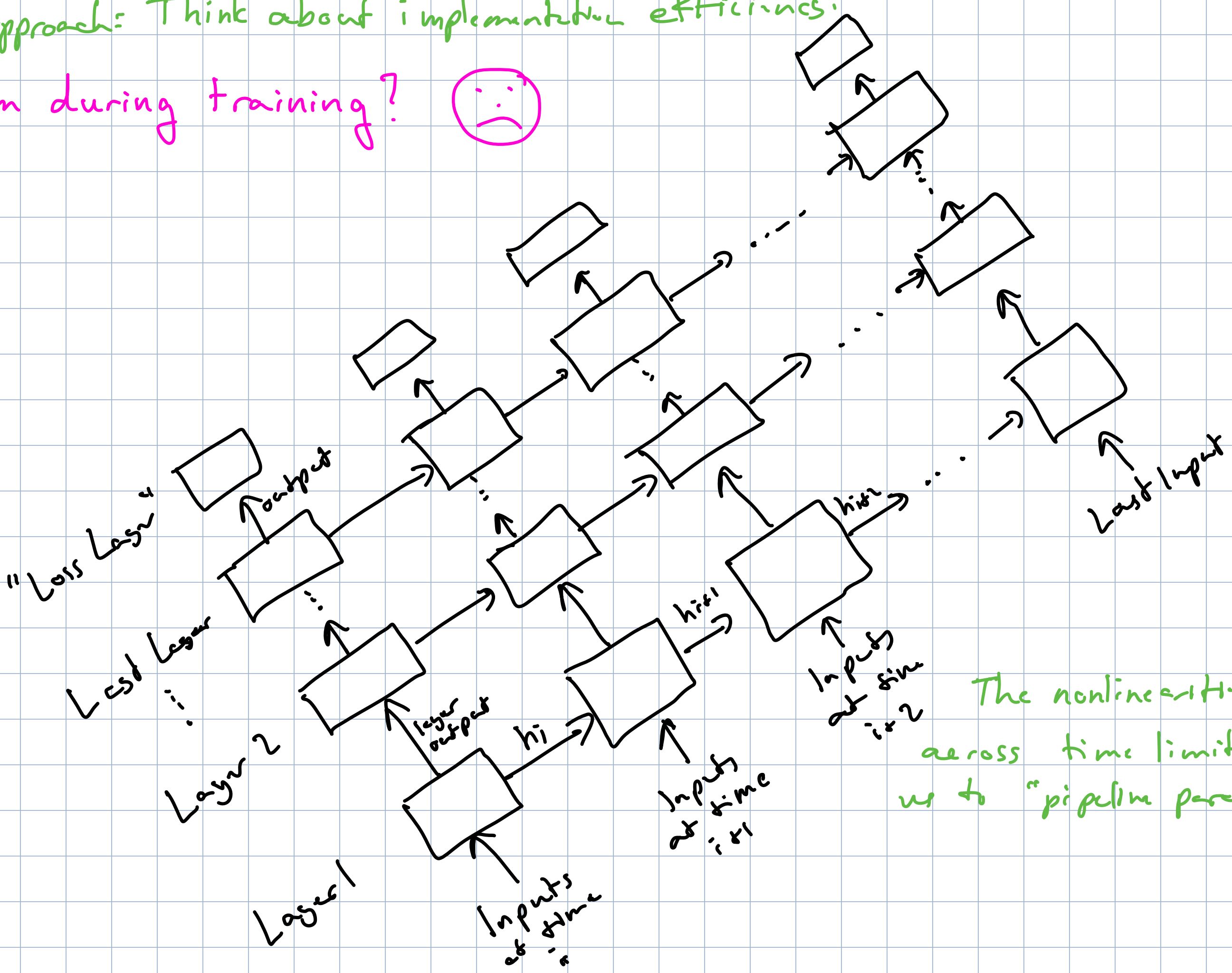
So What's the Problem with Traditional RNNs?

Traditional Answer: Hard to learn long-range dependencies.

Many traditional answers and approaches... (LSTMs, GRUs, etc...)

Radical Approach: Think about implementation efficiency:

Parallelism during training?



The nonlinearities across time limits us to "pipeline parallelism"

Step Back for clarity: Think about Kalman Filters...

Underlying true linear process:

$$\vec{x}_{t+1} = A\vec{x}_t + B_u \vec{u}_{t+1} + \vec{w}_{t+1}$$

$$\vec{y}_{t+1} = C\vec{x}_{t+1} + D\vec{u}_{t+1} + \vec{v}_{t+1}$$

where $\vec{w}_i, \vec{v}_i$ are iid $N(0, K_w)$ and $N(0, K_v)$

If all we observe are $\vec{u}_t, \vec{y}_t$, and we know A, B, C, D, K_w, K_v, the optimal steady-state filter to estimate $\vec{x}_t$ (in a mean-squared sense) has the form:

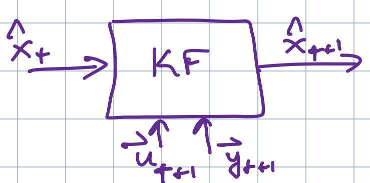
$$\hat{x}_{t+1} = A\hat{x}_t + B_u \vec{u}_{t+1} + F(\vec{y}_{t+1} - \underbrace{C(A\hat{x}_t + B_u\vec{u}_{t+1}) - D\vec{u}_{t+1}}_{\text{Predictable Parts}})$$

$$= \tilde{A}\hat{x}_t + \tilde{B}\vec{u}_{t+1} + F(\vec{y}_{t+1})$$

where $\tilde{A} = (I - FC)A$ and $F = PC^T(CPC^T + K_v)^{-1}$

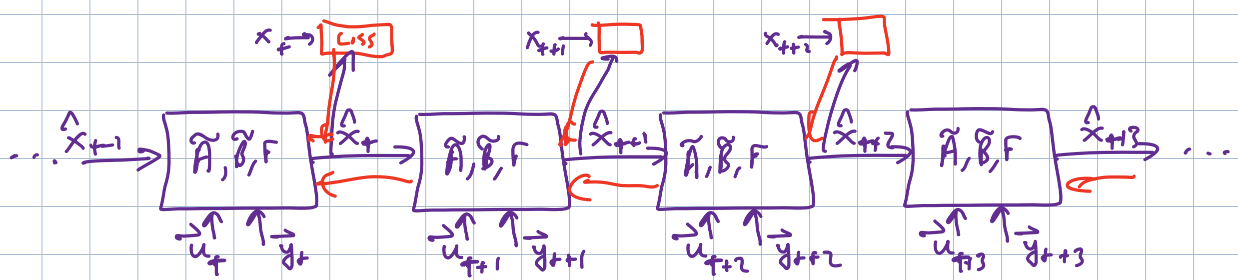
$\tilde{B} = (I - FC)B_u - FD$ when P solves D.A.R.E (Discrete Algebraic Riccati Equation)

$$P = APA^T + K_w - APC^T(CPC^T + K_v)^{-1}CPA^T$$



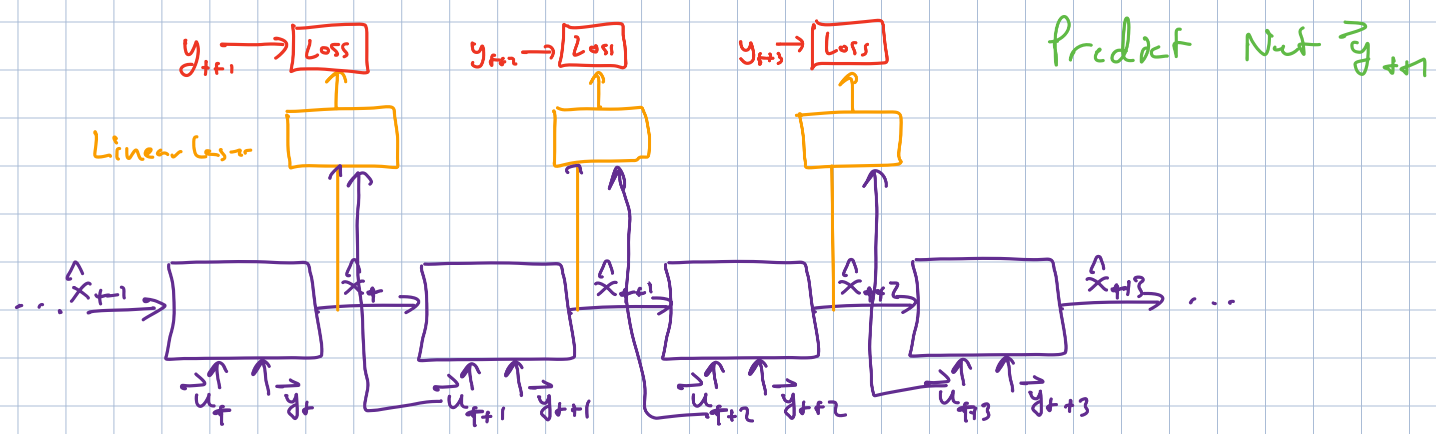
But what if we don't know A, B, C, D, K_w, K_v... And all we have is training data traces: $(\vec{x}_j[t], \vec{u}_j[t], \vec{y}_j[t])_{t=1}^{T_j}$ where j is different such traces (possibly of different lengths)

1) Deem $\tilde{A}, \tilde{B}, F$ learnable parameters. (Shared weights across time)
2) Put squared loss on $\hat{x}[t]$ i.e. $\|\hat{x}[t] - x_j[t]\|^2$
3) Run with $u_j[t]$ $y_j[t]$ as inputs



Now, what if we only had traces $(\vec{u}_j[t], \vec{y}_j[t])_{t=1}^{T_j}$ — No states. Just input & output

Question: Can we learn $\tilde{A}, \tilde{B}, F$ for KF? Why or Why not?
No. Can change coordinates for $\vec{x}$ and this is unobservable from



This basically works! We learn the essence of the KF but not the specific coordinate system for $\hat{x}$. Practically, this means that we can learn the coordinate system with just a little bit of $\vec{x}$ data.

General Principle: Self-supervision

"I need labels to train, I don't have labels, so make my own labels from data"

Lessons from example above:
1) We can learn a partial pattern that can be useful
2) Might need scaffolding parts of my NN.
3) Generic idea of "next-thing" prediction in causal sequence modeling

Step Back: Connect to unsupervised learning in classic ML.

Two approaches:
1) Dimensionality Reduction ⟵ Start here
2) Clustering

Recall Dimensionality Reduction.

Think about PCA

All we have are $\{\vec{x}_i\}_{i=1}^{N} \leftarrow$ d-dim   Unlabeled Data From Interesting Distribution

Classic Recipe (Neglecting Means):

1) Construct
$$X = \begin{bmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vdots \\ \vec{x}_N^T \end{bmatrix}$$

2) Compute SVD $X = U \Sigma V^T = \sum_i \sigma_i \vec{u}_i \vec{v}_i^T$

3) Keep top k $\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k$ singular vectors to use for dim-reduction

4) Given some other problem $\vec{x} \longrightarrow \left.\begin{bmatrix} \vec{v}_1^T \vec{x} \\ \vdots \\ \vec{v}_k^T \vec{x} \end{bmatrix}\right\}$ k-dim Features

Why was this reasonable?

Recall Eckhart-Young-Mirsky Theorem for Frobenius Norm

Given X, $\hat{X} = \sum_{i=1}^{k} \sigma_i \vec{u}_i \vec{v}_i^T$ is the rank-k matrix that minimizes
$$\| X - \hat{X} \|_F^2$$