Lecture 3

Feed-forward Neural Networks

Lars A. Lochstoer

UCLA Anderson School of Management

Overview

The Universal Approximation Theorem

- Neural nets as universal function approximators

Deep learning and efficiency: The role of multiple layers

- What does multiple layers buy us?
- Functional compositions

Feed-forward networks

- Review of structure
- What functions to approximate: SSM, SR, SSM+GARCH
- Approximating state variables using finite-windows
- Estimation: stochastic gradient descent, backpropagation, vanishing gradient problem

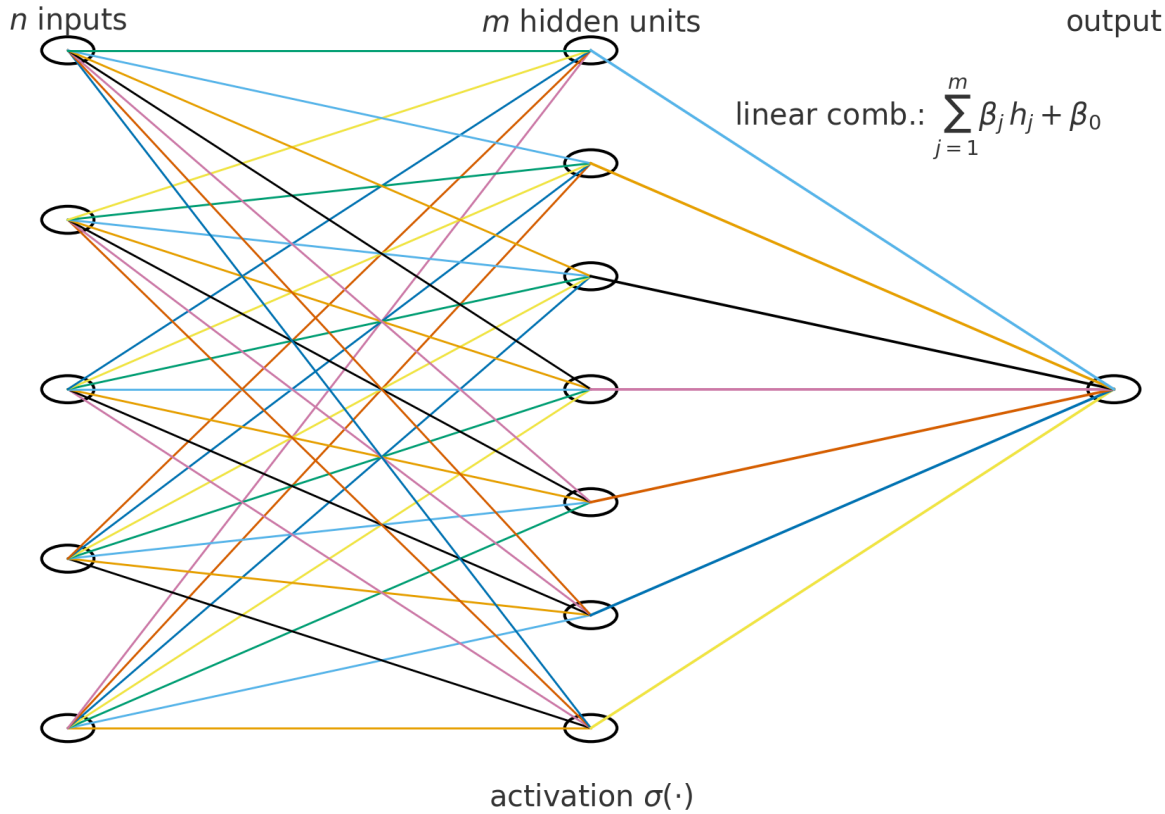
Why Neural Nets?

The Universal Approximation Theorem

One-Layer Networks: A Review

Recall the structure of a one-hidden-layer network

- n_0 inputs (data)
- n_l units in hidden layer l
- Activation function $\sigma(\cdot)$ (e.g., sigmoid, ReLU, Tanh)



One-Layer Networks: Mathematical Representation

Data space: $x \in \mathbb{R}^{n_0}$. Target: $f : K \to \mathbb{R}$ continuous on compact $K \subset \mathbb{R}^{n_0}$.

Single-hidden-layer network with n_1 units:

$$N_{n_1}(x) = \sum_{j=1}^{n_1} \beta_j \, \sigma(w_j^\top x + b_j) + \beta_0,$$

parameters $\theta = (\{\beta_j\}_{j=0}^{n_1}, \{w_j, b_j\}_{j=1}^{n_1})$. Activation function $\sigma : \mathbb{R} \to \mathbb{R}$.

Approximation criterion: $\|f - N_{n_1}\|_\infty = \sup_{x \in K} |f(x) - N_{n_1}(x)|$.

Goal: for any $\varepsilon > 0$, find m and θ with $\|f - N_{n_1}\|_\infty < \varepsilon$.

Universal Approximation Theorem

Theorem

Let $K \subset \mathbb{R}^{n_0}$ compact and $C(K)$ continuous real-valued functions on K.

If σ is continuous and nonpolynomial (e.g., sigmoid, tanh, ReLU), then the set

$$\mathcal{N} = \left\{ \sum_{j=1}^{n_1} \beta_j \, \sigma(w_j^\top x + b_j) + \beta_0 : \; n_1 \in \mathbb{N} \right\}$$

is dense in $C(K)$ under $\| \cdot \|_\infty$.

- "Dense in $C(K)$": neural net can approximate any continuous function on K to arbitrary accuracy!

Equivalently, $\forall f \in C(K)$, $\forall \varepsilon > 0$, $\exists N_{n_1} \in \mathcal{N}$ with $\| f - N_{n_1} \|_\infty < \varepsilon$.

Remarks: density does not imply uniqueness or parameter identifiability; rates of convergence depend on smoothness of f and choice of σ.

Proof sketch: The ReLU case

ReLU can create a continuous piecewise-linear approximation to a continuous function.

- Easiest to see using a specific example

Let $f(x) = \sin(x)$, $x \in [0, 2\pi]$. Choose uniform knots $0 = x_0 < x_1 < \cdots < x_{n_1} = 2\pi$, spacing $h = 2\pi / n_1$.

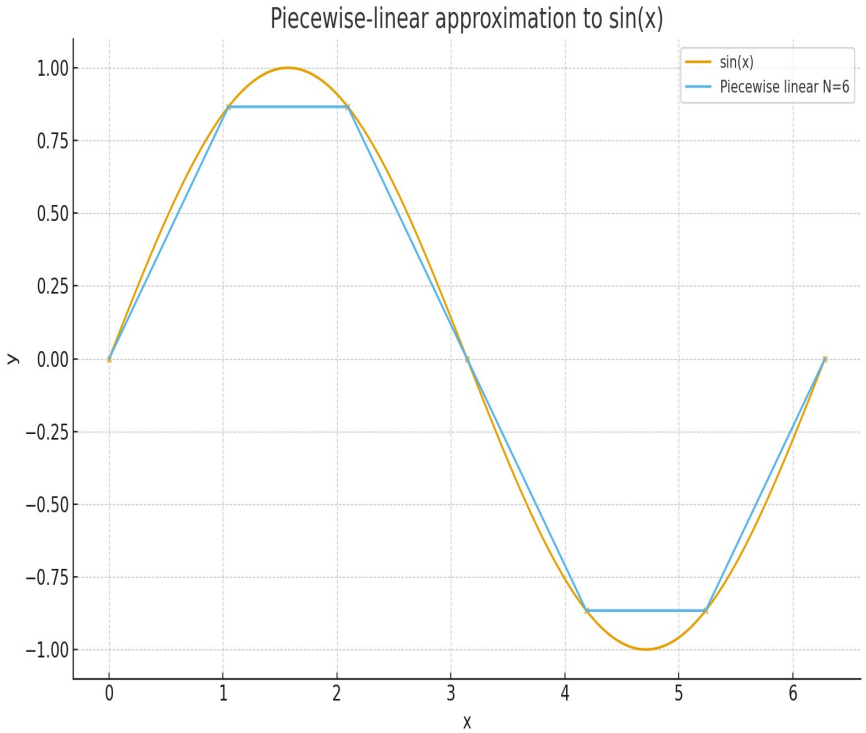
Linear spline S_h defined by node values $y_j = f(x_j)$: for $x \in [x_j, x_{j+1}]$,

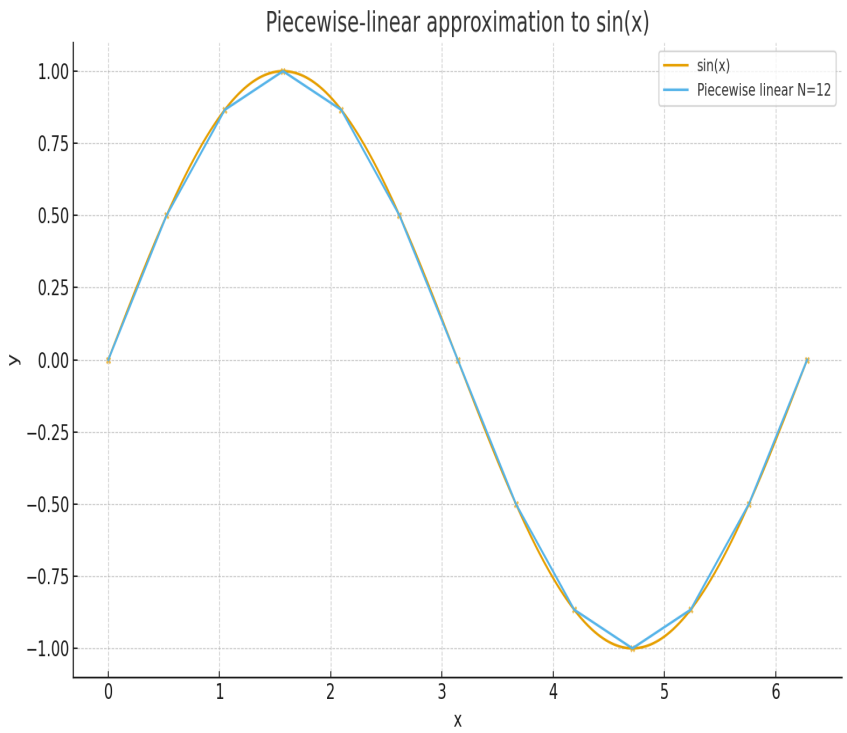
$$S_h(x) = \frac{x_{j+1} - x}{h} y_j + \frac{x - x_j}{h} y_{j+1}.$$

- Ie, pick points on true function, linearly interpolate between these points.
- Next slide shows this and hints that with more points (units) we get less approximation error!

Linear interpolation of continuous function

Intuitively (and it's mathematical fact), the approximation error becomes arbitrarily small as the number of ReLU units goes to infinity





Why Deep Learning?

Functional Approximations and Efficiency

What does multiple hidden layers buy us?

Efficiency.

Functional compositions: $(f \circ g)(x) = f(g(x))$

- Layers act like functional compositions
- Thus, if the function you are trying to approximate can be expressed as a functional composition, multiple hidden layers is likely more efficient than a single hidden layer

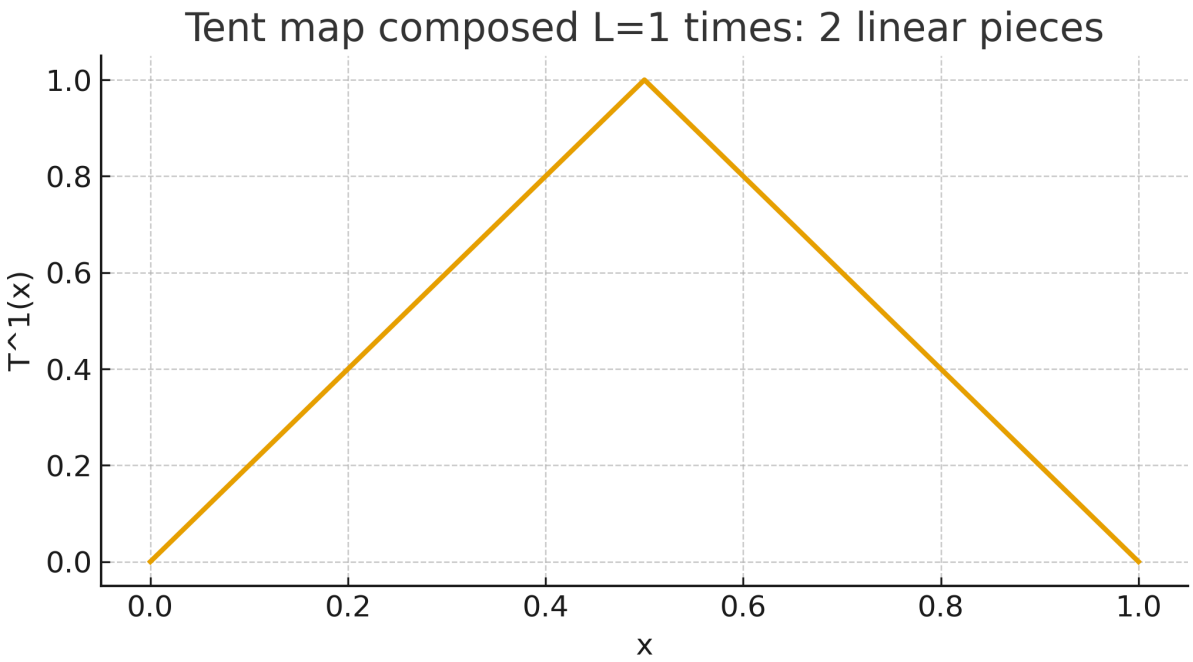
Efficiency: smaller out-of-sample forecast error (e.g., expected mean squared error)

- Recall two types of error: variance and bias
- Multiple layers often have fewer parameters when calibrated to the same approximation error (ie. same bias) as a one-layer network
  - Thus, less estimation error and more efficient

Simple example of multilayer efficiency

Consider a tent map (function):

$$T(x) = \max(0, 1 - 2|x - 1/2|) \quad \text{on } x \in [0, 1]$$



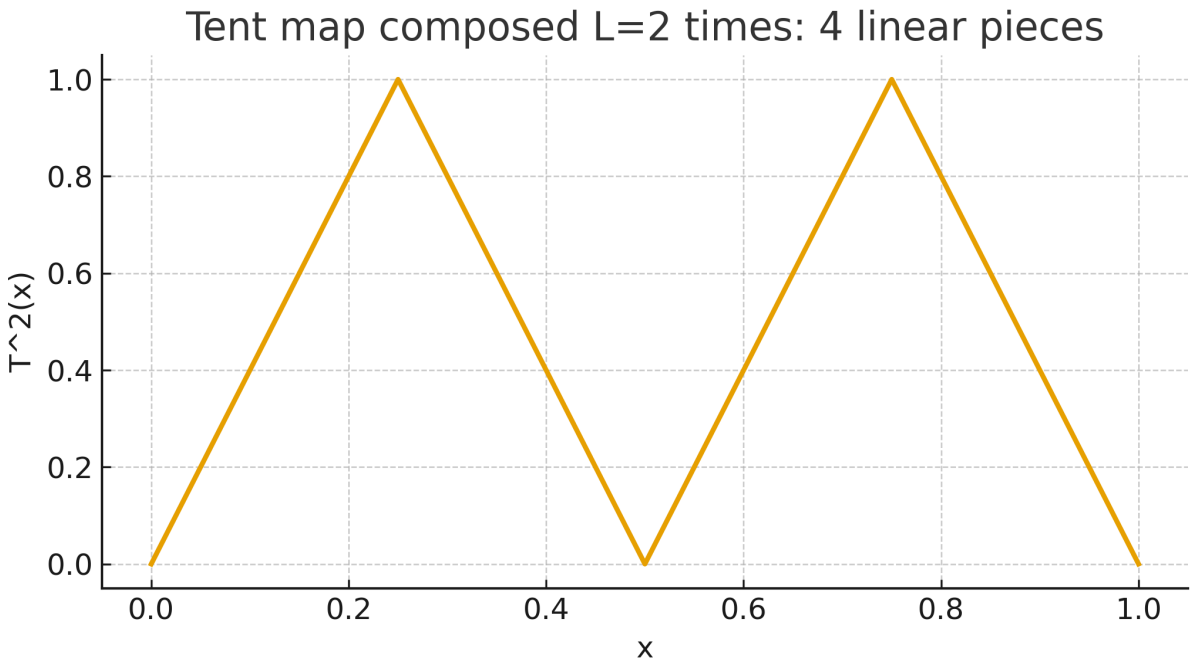
Can be fit with 3 ReLu units

- Think long 1 call with K = 0, short two calls with K = 0.5, long one call with K = 1

Simple example: functional composition

Next, consider a case where function is a composition of the same function:

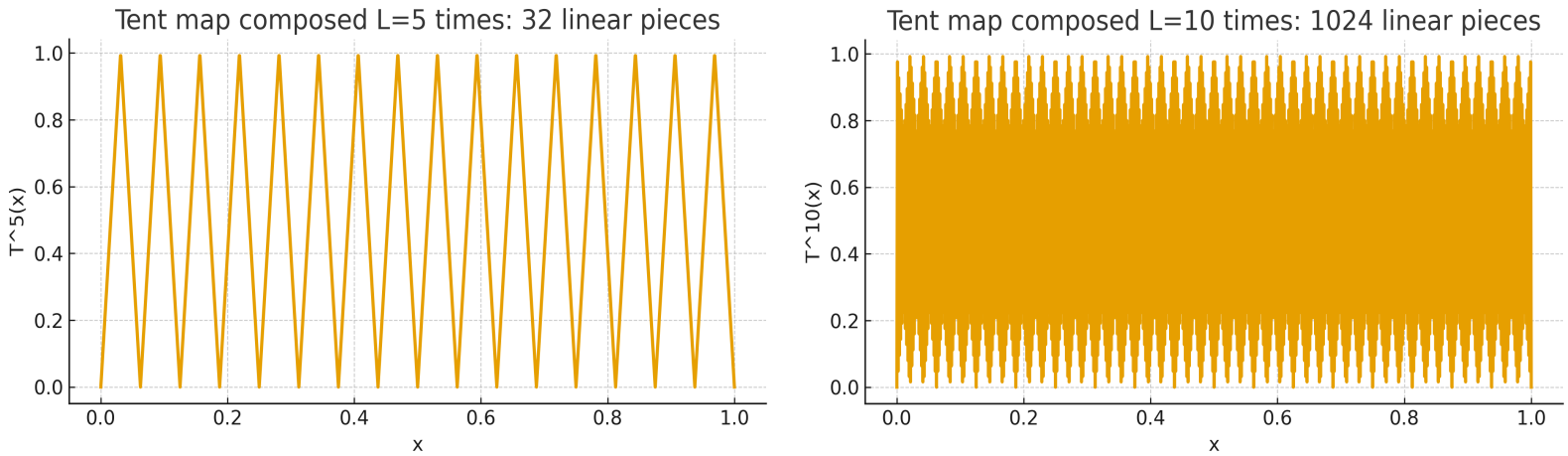
$$T^{(2)} = (T \circ T)(x) = T(T(x))$$



Simple example: functional composition (cont'd)

General L-time composition (again, on same function which is special case_)

$$T^{(L)} = \underbrace{(T \circ T \circ \cdots \circ T)}_{L \text{ times}}(x)$$



- Original function has 2 linear pieces, L layer composition has 2^L linear pieces
  - Exponential growth in complexity it allows for!

Simple example: fitting with ReLU

One-layer ReLU:

$$f(x) = c_0 + c_1 x + \sum_{i=1}^{n_1} a_i \max(0, x - b_i)$$

- Thus, $2n_1 + 2$ parameters, where n_1 is the number of units in the layer
- You basically need one unit per kink in the function
- You need intercept and slope (c_0 and c_1) to fit average level and first slope from left (possible to add another unit instead of the $c_1 x$ term, but this is less efficient as it introduces 2 parameters instead of 1)

Multi-layer approach

- Repeating triangles: need 2 ReLU units per layer

Simple example: fitting with ReLU (cont'd)

With $L = 10$ there are $2^{10} = 1024$ linear pieces and 1023 kinks (with endpoints not counted)

- A one-hidden-layer network thus needs 1023 ReLU units; $2n_1 + 2 = 2048$ parameters

Multilayer approach: 10 layers, 2 ReLU units per layer, plus final output layer

- Layer 1: two units – 2 weights w_i and intercepts b_i
- Layer 2-10: two units with two inputs from prior layer: 4 weights w_i and 2 intercepts b_i
- Final output, linear average: 2 weights and 1 intercept

In sum: 61 parameters with 10-layer network, 2048 with 1-layer

- Clearly $61 < 2048$, which is the source of efficiency gain!

Common functions that use multiple layers

Certain common functions are more efficiently approximated by multiple hidden layers.

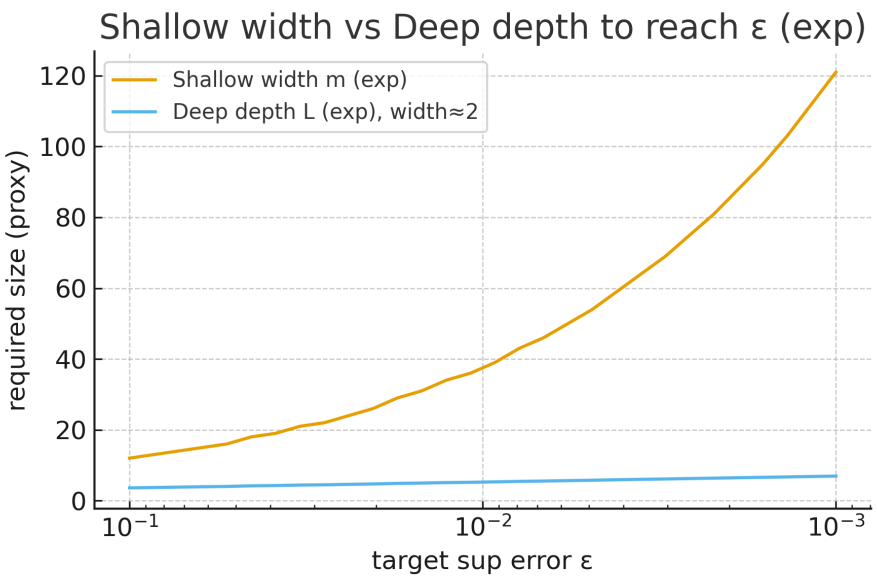
- $\exp(\cdot)$ and $\ln(\cdot)$ are prominent examples

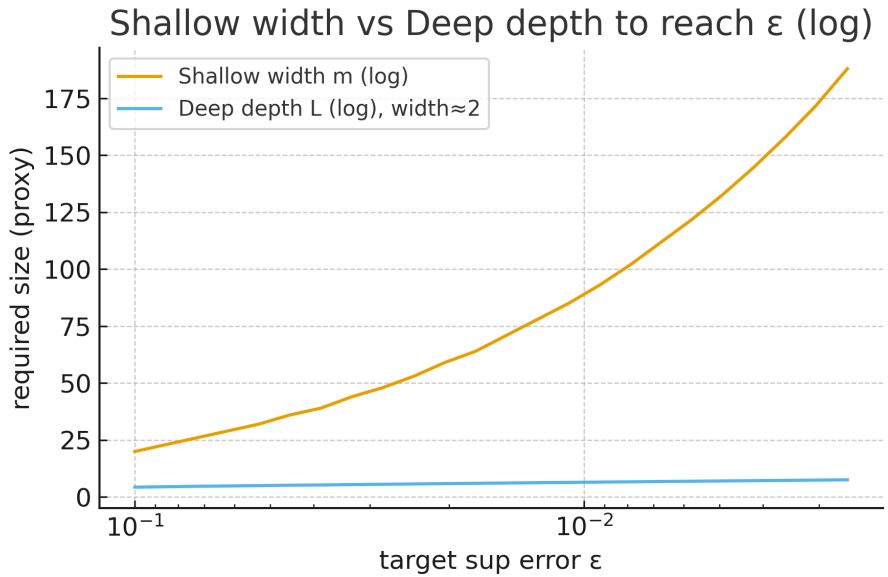
Curvature means many units required in single layer

- As previous example shows, can get in the order of n^L (often written $O\left(n^L\right)$) breakpoints with n units per layer, where L is number of layers
  - The ReLU naturally gives rise to a linear spline approximation and many break points are needed for close approximations
- Plots on next slide show efficiency gain as a function of approximation error ε

Common functions: deep learning efficiency

- Think of the size of the model as related to the number of parameters needed to achieve a certain approximation error
- As we go further to the right on the x-axis, the approximation error becomes smaller
- The one-layer network (sometimes refer to as 1-D) explodes in size, while the multi-layer approach does not.





Form of activation function

The above discussion assumed ReLU activation function $\sigma(\cdot)$

- Convenient as piecewise linear is easy to understand

Other forms are also common

- Which is best depends on function we are trying to approximate
- Typically, we use cross-validation and MSE to choose number of layers and type of activation function
- Still, there are choices we can make beyond this approach if we know something about the function we want to approximate

Other common activation functions include:

$$\text{sigmoid: } \frac{e^u}{1+e^u},$$

$$\text{tanh: } \frac{e^u - e^{-u}}{e^u + e^u}.$$

What types of functions are we trying to approximate?

Benchmarks

Before we start fitting a bunch of data and look at out-of-sample MSEs....

- Let's think about some properties of the type of functions we are looking to approximate

Three examples:

1. SSM: simple linear Gaussian case
2. SR: simple 2-state switching model
3. SSM-GARCH: SSM model with time-varying variance of shocks

What do optimal forecasts look like in these models, and how can FFNNs (feed-forward neural nets) approximate this?

Linear state-space model forecasts

Standard state-space models have forecasts of the form:

$$\begin{aligned} E_t\left(y_{t+1}\right) &= Z a_{t+1|t} + d \\ &= \phi_0 + \sum_{j=1}^{\infty} \phi_j y_{t+1-j}. \end{aligned}$$

(here prior is not present as we have implictly assumed stationary dynamics and an infinite history)

- That is, the forecast as a $AR\left(\infty\right)$ (or $VAR\left(\infty\right)$ in vector case)

ReLU can fit a linear function with two units:
$a + bx = a + b\max\left(0, x\right) - b\max\left(0, -x\right)$

- Note: input vector to FFNN should be *all historical data* given the infinite lag requirement
- In practice, choose window length J, which implicitly sets $\phi_j = 0$ for $j > J$
  - Decent approximation if system not too persistent
  - Thus: at time t feed the FFNN y_t, y_{t-1}, ..., y_{t-J+1}

Switching regime model forecasts

Consider simple 2-state model $y_t|s_t \sim N\left(\mu_{s_t}, \sigma^2_{s_t}\right)$ with transition probability matrix Π

- From previous lecture note:

$$E_t\left(y_{t+1}\right) = \mathbf{p}_t'\Pi\mu,$$

where $\mu' = \begin{bmatrix} \mu_1 & \mu_2 \end{bmatrix}$ and $\mathbf{p}_t' = \begin{bmatrix} p_t\left(s_t = 1\right) & p_t\left(s_t = 2\right) \end{bmatrix}$

So we need to express beliefs as a function of the history of observables

- Recall, update for belief that $s_t = j$:

$$\begin{aligned} p_t(j) &= \frac{1}{c_t} f_j(y_t)\, p_{t|t-1}(j) \\ &= \frac{1}{c_t} f_j(y_t) \sum_{i=1}^{2} \pi_{ij}\, p_{t-1}(i), \end{aligned}$$

where $c_t = \sum_{m=1}^{2} f_m(y_t)\, p_{t|t-1}(m)$ is the normalizing constant so probabilities sum to 1 and $\pi_{ij} = \Pr\left(S_t = j|S_{t-1} = i\right)$

Forecast function based on observables

We can iterate backwards on the updating equation:

$$p_t(j) = \frac{1}{c_t} f_j(y_t) \sum_{i=1}^{2} \pi_{ij} \frac{1}{c_{t-1}} f_i(y_{t-1}) p_{t-1|t-2}(i)$$

- Keep going and we get a function that is a sum of exponentials of lagged y^t: multiple layers will be efficient!

Note: if $f_j(y_t) > f_i(y_t)$ the data is saying state j was more likely

- Thus, lagged sequence of data tells us if we should weight μ_1 or μ_2 more in our current forecast
- Again, need full history of y^t as input vector, but in practice choose a window of length J where J depends on the persistence of the system

SSM-GARCH model forecasts

Model: $y_t = Zx_t + d + \varepsilon_t$, $x_t = \phi x_{t-1} + \eta_t$, with $\varepsilon_t \sim N(0, h_t)$ and $\eta_t \sim N(0, q)$

- Note that observation equation has time-varying volatility of noise, h_t

Optimal one-step forecast: $E_t(y_{t+1}) = Z\phi a_{t|t} + d$

- The forecast admits an $AR(\infty)$-type representation

$$E_t(y_{t+1}) = c_t + \sum_{j=0}^{\infty} \psi_{j,t} y_{t-j},$$

  where c_t and $\psi_{j,t}$ are functions of the history of the variances of the noise terms, h^t, as well as the model parameters (Z, ϕ, q)
- Interpretation: larger past h_{t-j} reduces the effective weight $\psi_{j,t}$ assigned to y_{t-j}. The vector h^t therefore acts like an "attention" vector that down-weights noisier observations

Understanding FFNNs

A Monte-Carlo Perspective

Notation for deep feed-forward network

Input x is $n_0 \times 1$ vector of data.

Output of each hidden layer $l = \{1, ..., L\}$

$$\mathbf{a}^{[l]} = \sigma^{[l]} \left(\mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \right),$$

where the activation function is applied elementwise.

- Layer l has n_l units. Thus, $\mathbf{a}^{[l-1]}$ is $n_{l-1} \times 1$, $\mathbf{W}^{[l]}$ is $n_l \times n_{l-1}$, $\mathbf{a}^{[l]}$ and $\mathbf{b}^{[l]}$ are $n_l \times 1$
- It's also useful to define: $\mathbf{h}^{[l]} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$

Output layer:

$$\mathbf{y} = \mathbf{U} \mathbf{a}^{[L]},$$

where $\mathbf{U}$ is $n_{L+1} \times n_L$. Here $n_{L+1} \geq 1$.

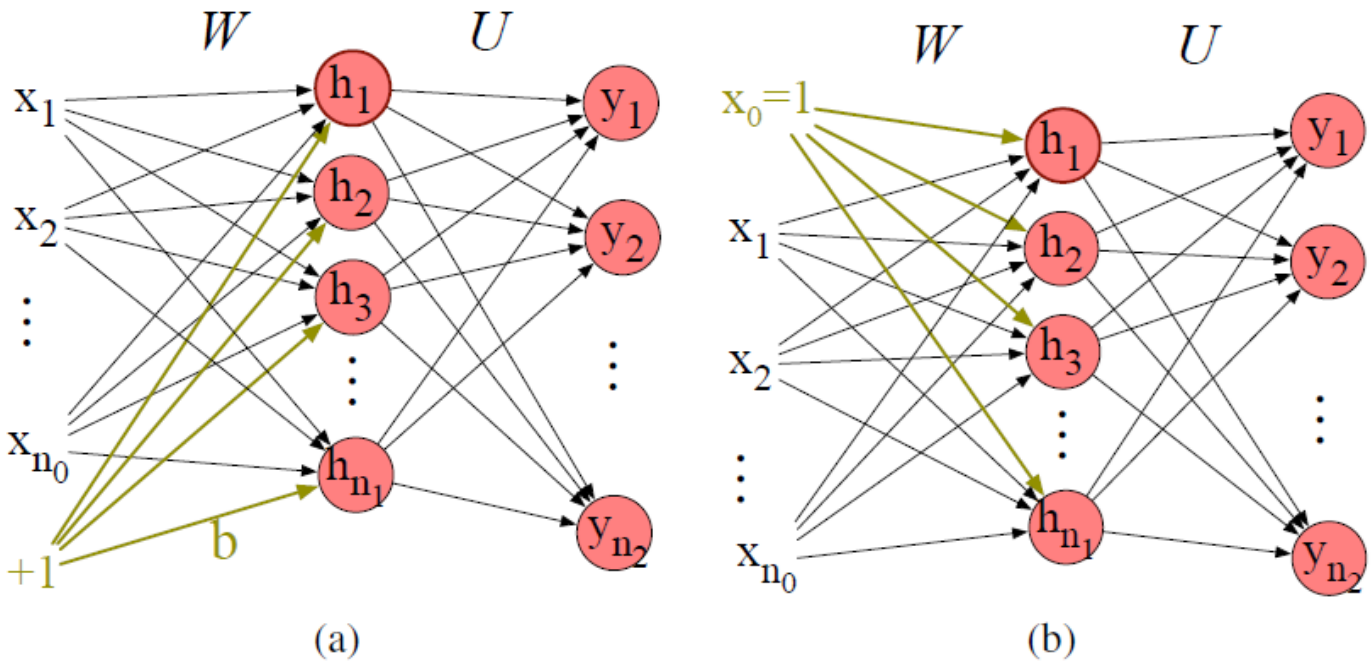
- Often, if $n_{L+1} > 1$, we are looking for a probability distribution where we take the additional step

$$\tilde{y}^{(k)} = \text{softmax} \left(y^{(k)} \right) = \frac{\exp \left(y^{(k)} \right)}{\sum_{j=1}^{n_{L+1}} \exp \left(y^{(j)} \right)},$$

where superscript (j) denotes row j

Canonical 2-Layer feed-forward network

(a) with bias node; (b) bias node included in x vector



In this plot, the output layer has many variables (e.g., for a probability distribution)

Estimating the weights

Often Stochastic Gradient Descent is used to estimate the function

- Use random subset of data (a mini-batch of length b) to update gradient/parameter guess
- An "Epoch" is one full pass over training set (number of iterations per Epoch T/b if T is size of sample)
- Typical Epoch settings is 20-300 depending on case
- Early Stopping means stop when objective function fails to improve

Update rule:

$$\theta_{t+1} = \theta_t - \eta g_{B_t}(\theta_t),$$

where $g_{B_t}(\theta_t)$ is the mini-batch gradient for iteration t and η is the learning rate.

- θ_t is a vector with all the parameters

Overview of Monte Carlo exercise

- **Goal:** learn a feed-forward NN that maps a short window of past observations to the one-step-ahead forecast.
- **Data:** 2-state Gaussian switching-regime (HMM "emissions") with asymmetric transitions.
- **Training:** only uses y (and optionally a noisy proxy z); loss is MSE on realized y_{t+1}.
- **Evaluation:** compare NN forecast against *true* conditional mean from HMM filter and an AR(1) baseline.

Monte Carlo: true model (HMM emissions)

State process: $s_t \in \{0, 1\}$, Markov with

$$\Pi = \begin{bmatrix} 0.95 & 0.05 \\ 0.20 & 0.80 \end{bmatrix}, \qquad \mathbf{p}_{t|t-1} = \Pi \mathbf{p}_{t-1}.$$

Observation:

$$y_t \mid s_t = j \sim \mathcal{N}(\mu_j, \sigma^2), \qquad \mu = (-1,\, 1),\ \sigma = 1.$$

Filtering and forecast: with posterior $\mathbf{p}_t = \Pr(s_t = \cdot \mid y_{1:t})$,

$$m_t \equiv \mathbb{E}[y_{t+1} \mid y_{1:t}] = \mathbf{p}_t' \Pi \mu, \qquad p_t(i) \ \propto \ p_{t|t-1}(i) f_i(y_t).$$

Irreducible risk vs realized y_{t+1}:

$$\mathrm{Var}(y_{t+1} \mid y_{1:t}) = \sigma^2 + (\mu_1 - \mu_0)^2 \, \mathbf{p}_{t+1|t}' (1 - \mathbf{p}_{t+1|t}).$$

Neural net specification

Inputs (window $J = 2$): $X_t = [y_{t-1}, y_t]$.

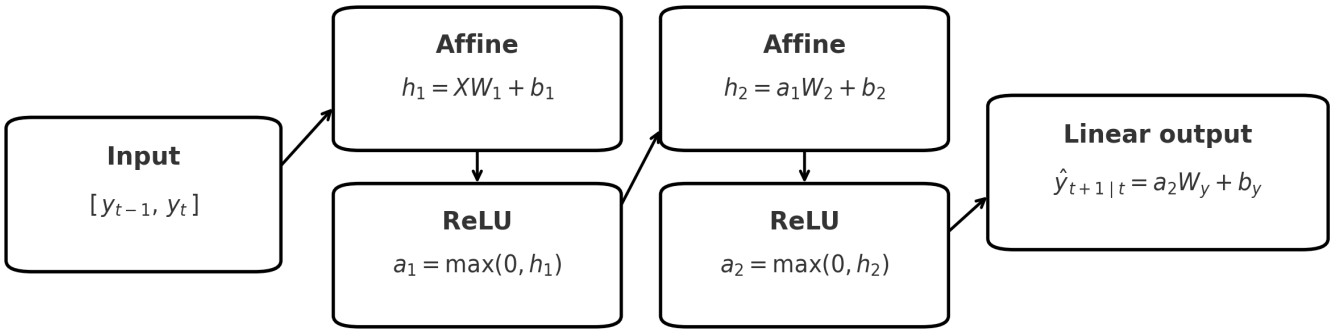
Architecture:

$$X \xrightarrow{\text{Aff+ReLU}} a_1 \xrightarrow{\text{Aff+ReLU}} a_2 \xrightarrow{\text{Aff}} \hat{y}_{t+1|t}.$$

Hidden sizes $n_1 = n_2 = 64$; ReLU activations; linear output.

Loss (training on realized y_{t+1}):

$$\mathcal{L} = \frac{1}{T}\sum_t (\hat{y}_t - y_t^{\text{obs}})^2 + \frac{\lambda}{2}\sum \|W\|_F^2.$$



Estimation of Feed-forward Neural Net

Input: at each t, input y_t, y_{t-1}, ..., y_{t-J+1}

Numerical optimizer: Typically, Stochastic Gradient Descent and MSE loss function plus L1 (Lasso) and/or L2 (Ridge) constraint

Stochastic Gradient Descent:

- Gradient based method (derivative of objective function wrt parameters in network)
  - Use backpropagation to compute gradient (or derivative vector, or score)
- Stochastic: Choose a random mini-batch (random sub-set of data) for each iteration of gradient descent
- Epoch: is one complete pass through the data.
  - Multiple epochs typically chosen to fit model until some early stopping criterion (e.g., parameters stops updating meaningfully) is achieved

Backpropagation: goal

Goal. Efficiently compute the gradient $\nabla_\theta \mathcal{L}$ of a scalar loss $\mathcal{L}$ w.r.t. millions of parameters θ so we can *learn* via gradient methods.

Why it is needed.

- Directly differentiating the full composition is costly and error-prone.
- Finite differences need $O(\#\theta)$ forward passes and are noisy.
- Backpropagation uses the *chain rule* organized as dynamic programming to get all partials in time proportional to one forward/backward sweep.

Recall:

$$\frac{\partial f\left(g\left(x\right)\right)}{\partial x} = \frac{\partial f\left(g\left(x\right)\right)}{\partial g\left(x\right)} \frac{\partial g\left(x\right)}{\partial x}$$

Backpropagation: core idea

Core idea. Treat the network as a computation graph of local functions. For a layer output $a^{(\ell)}$ feeding later nodes, the sensitivities of the loss with respect to node values are given by (using Chain Rule directly here)

$$\underbrace{\frac{\partial \mathcal{L}}{\partial h^{(\ell)}}}_{\text{upstream "blame"}} = \sum_{k \in \text{children of } \ell} \underbrace{\frac{\partial \mathcal{L}}{\partial h^{(k)}}}_{\text{already known}} \underbrace{\frac{\partial h^{(k)}}{\partial h^{(\ell)}}}_{\text{local Jacobian}}.$$

Propagate these sensitivities *backward* from the loss to inputs; then obtain parameter gradients by one more local multiplication:

$$\nabla_{W^{(\ell)}} \mathcal{L} = \left(a^{(\ell-1)}\right)^\top \underbrace{\frac{\partial \mathcal{L}}{\partial h^{(\ell)}}}_{\Delta^{(\ell)}}, \qquad \nabla_{b^{(\ell)}} \mathcal{L} = \mathbf{1}^\top \Delta^{(\ell)}.$$

- To understand last equation, elementwise we have

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(\ell)}} = \sum_r \frac{\partial \mathcal{L}}{\partial h_r^{(\ell)}} \frac{\partial h_r^{(\ell)}}{\partial W_{ij}} = \frac{\partial \mathcal{L}}{\partial h^{(\ell)}} a_j^{(\ell-1)} \quad \text{where } \frac{\partial h_r^{(\ell)}}{\partial W_{ij}} = \mathbf{1}_{\{r=i\}} a_j^{(\ell-1)}$$

Backpropagation: purpose

What it accomplishes.

- *Efficiency:* Computes $\nabla_\theta \mathcal{L}$ in $O(\text{forward cost})$, independent of $\#\theta$ up to constants.
- *Correct attribution:* Each weight learns how a small change would reduce $\mathcal{L}$ given the current batch.
- *Scalability:* Works for any differentiable graph (ReLU, normalization, residuals), enabling deep nets.

Econometric analogy. Backprop is a *reverse recursion of scores*: local "scores" (sensitivities) are propagated backward, like filtering vs. smoothing forward computes predictions, backward attributes error to earlier transformations.

Backpropagation: algorithm, notation, and tips

Define: $\|W_1\|_F^2 = \operatorname{trace}(W_1^\top W_1) = \sum_i \sum_j (W_1)_{ij}^2$.

Mini-batch Stochastic Gradient Descent (one step).

1. *Forward:* compute $h_1, a_1, h_2, a_2, \hat{y}$.
2. *Loss:* $\mathcal{L} = \frac{1}{N}\sum(\hat{y}-y)^2 + \frac{\lambda}{2}\sum\|W\|_F^2$.
3. *Backward:*

$$\Delta = \tfrac{2}{N}(\hat{y}-y), \quad \Delta_{a_2} = \Delta W_y^\top, \quad \Delta_{h_2} = \Delta_{a_2} \odot \mathbb{I}[h_2 > 0],$$

$$\Delta_{a_1} = \Delta_{h_2} W_2^\top, \quad \Delta_{h_1} = \Delta_{a_1} \odot \mathbb{I}[h_1 > 0].$$

   Accumulate $\nabla_{W_\cdot}, \nabla_{b_\cdot}$ as on the derivation slide.
4. *Update:* $W \leftarrow W - \eta\,\nabla_W, \quad b \leftarrow b - \eta\,\nabla_b$.

Notes.

- ReLU mask $\mathbb{I}[h > 0]$ zeros gradients where units are inactive.
- Weight decay: add λW to ∇_W only (not to biases).
- Standardize X; pick small η; clip gradients if needed (to bound step size)
- All formulas are vectorized over the batch; avoid loops.

Backpropagation: derivation for the MLP

Network (mini-batch size N).

$$
\begin{aligned}
&X \in \mathbb{R}^{N \times J} \quad \text{(window length } J\text{)}, \\
&h_1 = X W_1 + \mathbf{1} b_1, \quad a_1 = \sigma(h_1), \\
&h_2 = a_1 W_2 + \mathbf{1} b_2, \quad a_2 = \sigma(h_2), \\
&\hat{y} = a_2 W_y + \mathbf{1} b_y \in \mathbb{R}^{N \times 1}, \\
&\sigma(u) = \max\{0, u\} \quad \text{(ReLU)}.
\end{aligned}
$$

Loss (MSE + L_2).

$$
\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{y}_i - y_i\right)^2 + \frac{\lambda}{2} \left(\| W_1 \|_F^2 + \| W_2 \|_F^2 + \| W_y \|_F^2 \right).
$$

Backpropagation: derivation for the MLP (cont'd)

Output gradient.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{2}{N}(\hat{y} - y) \;\equiv\; \Delta.$$

Readout layer.

$$\nabla_{W_y} = a_2^\top \Delta + \lambda W_y, \qquad \nabla_{b_y} = \mathbf{1}^\top \Delta, \qquad \Delta_{a_2} = \Delta W_y^\top.$$

Second hidden layer.

$$\Delta_{h_2} = \Delta_{a_2} \odot \mathbb{I}[h_2 > 0], \quad \nabla_{W_2} = a_1^\top \Delta_{h_2} + \lambda W_2, \quad \nabla_{b_2} = \mathbf{1}^\top \Delta_{h_2}, \quad \Delta_{a_1} = \Delta_{h_2} W_2^\top$$

First hidden layer.

$$\Delta_{h_1} = \Delta_{a_1} \odot \mathbb{I}[h_1 > 0], \quad \nabla_{W_1} = X^\top \Delta_{h_1} + \lambda W_1, \quad \nabla_{b_1} = \mathbf{1}^\top \Delta_{h_1}.$$

Shapes: $W_1 : J \times n_1$, $W_2 : n_1 \times n_2$, $W_y : n_2 \times 1$.

Backpropagation and SGD

Gradients:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{2}{N}(\hat{y} - y), \quad \frac{\partial \mathcal{L}}{\partial h_2} = \left(\frac{\partial \mathcal{L}}{\partial \hat{y}} W_y^\top\right) \odot \mathbf{1}_{h_2 > 0},$$

$$\nabla_{W_2} = a_1^\top \frac{\partial \mathcal{L}}{\partial h_2} + \lambda W_2, \quad \nabla_{W_1} = X^\top \left((\frac{\partial \mathcal{L}}{\partial h_2} W_2^\top) \odot \mathbf{1}_{h_1 > 0} \right) + \lambda W_1.$$

SGD update: $\theta \leftarrow \theta - \eta \nabla_\theta$ on mini-batches.

Code excerpt (note code has a and h reversed, sorry!**):**

```
dy = (2.0/N)*(y_hat - y_true)
dWy = h2.T @ dy + weight_decay*Wy
dh2 = dy @ Wy.T
da2 = dh2 * relu_grad(a2)
dW2 = h1.T @ da2 + weight_decay*W2
...
W2 -= lr*dW2; b2 -= lr*db2; Wy -= lr*dWy; by -= lr*dby
```

Evaluation protocol

True forecast (teacher) for comparison only:

$$m_t = \mathbb{E}[y_{t+1} \mid y_{1:t}] = \mu^\top(\pi_t P), \quad \text{taken from } \texttt{hmm\_filter}.$$

Alignment: windows ending at t map to m_t and to targets y_{t+1} at index $t+1$.

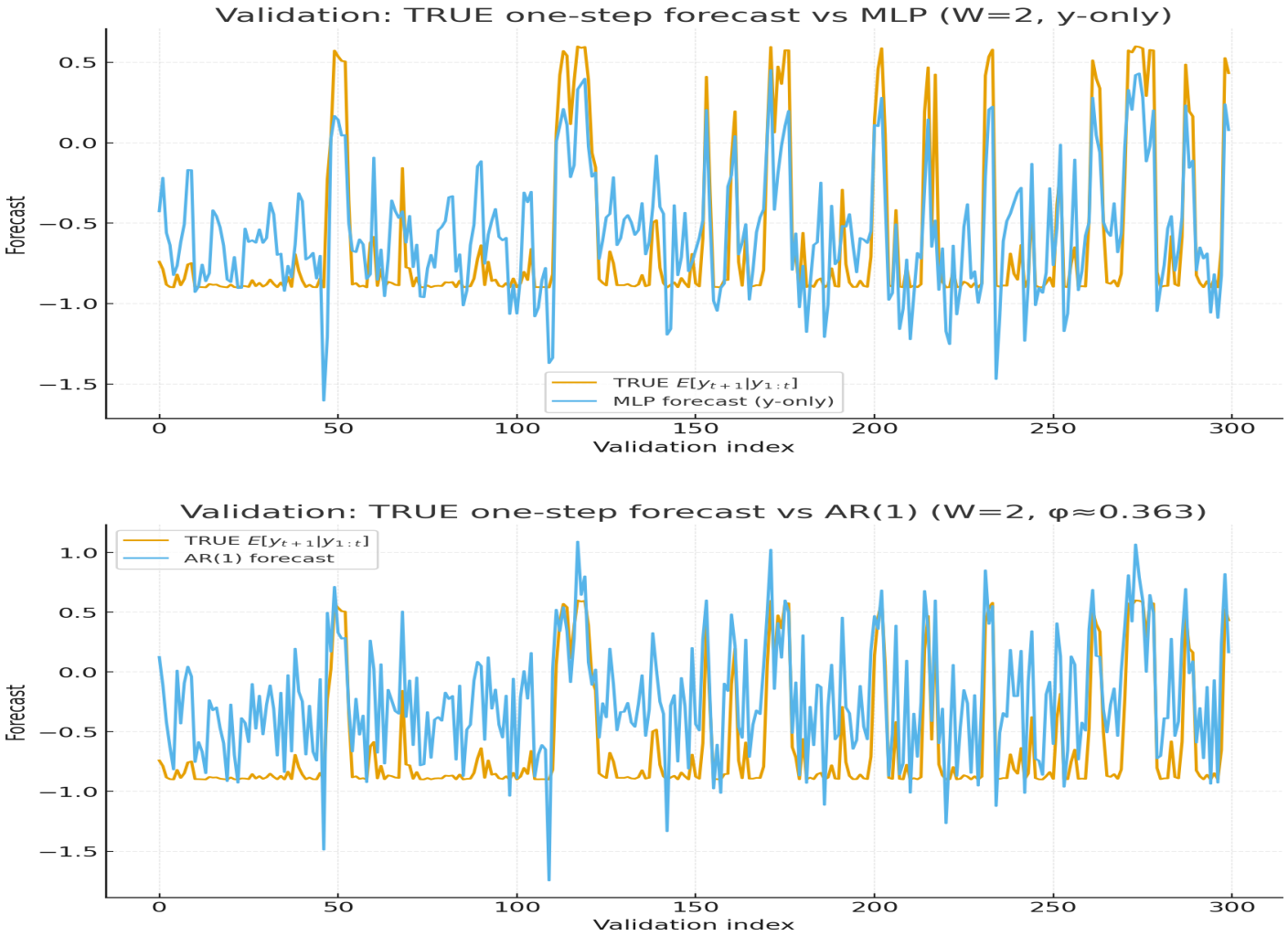
Metrics vs m_t:

$$\mathrm{MSE} = \frac{1}{N}\sum(\hat{y} - m)^2, \qquad R^2 = 1 - \frac{\sum(\hat{y} - m)^2}{\sum(m - \bar{m})^2}.$$

Plots:

1. m_t vs NN forecast.
2. m_t vs AR(1) forecast.

Results from SR Monte Carlo exercise



Reading: NN forecast tracks regime-driven shifts in the true mean better than AR(1). Discrepancies remain near switches due to residual mixture uncertainty.

Results from SR Monte Carlo exercise

Validation metrics vs true conditional mean m_t

Model	MSE	R^2
MLP	0.079	0.684
AR(1)	0.237	0.050

Reading: NN forecast tracks regime-driven shifts in the true mean better than AR(1) due to capacity for handling nonlinear models.

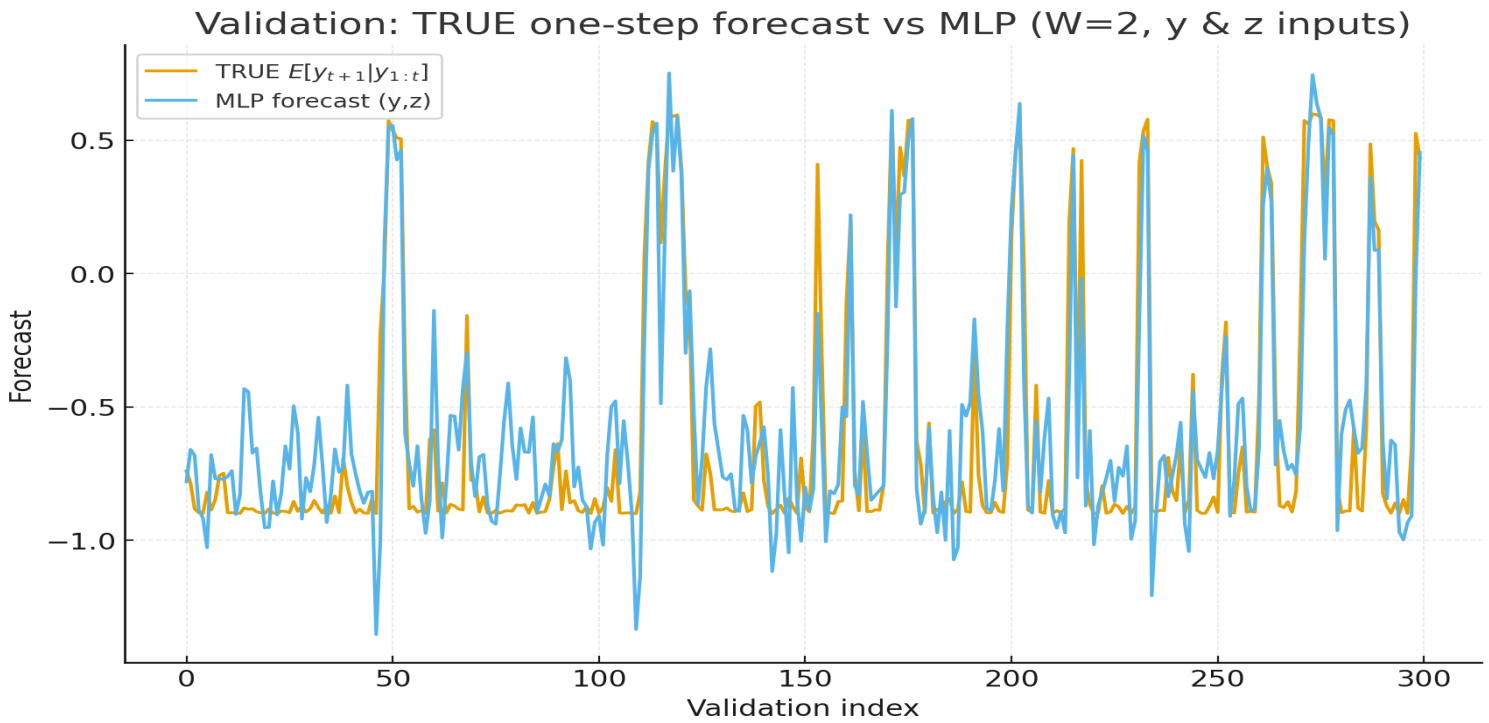
Adding a noisy regime proxy

Observe: $z_t = p_t(1) + \eta_t$, $\eta_t \sim \mathcal{N}(0, \sigma_z^2)$, where $p_t(1) = \Pr(s_t = 1 \mid y_{1:t})$

- Noisy but informative signal of what state we are in

Augmented input: $[y_{t-1}, y_t, z_{t-1}, z_t]$.

Training unchanged: MSE on y_{t+1}.



Results using additional signal

With signal model performs much better

Validation metrics vs true conditional mean m_t

Model	MSE	R^2
MLP	0.018	0.912
AR(1)	0.250	-0.227

Reading: NN forecast tracks regime-driven shifts in the true mean better than AR(1) due to capacity for handling nonlinear models.

Warning: Vanishing gradients problem

Setup.

- Depth-L MLP with ReLU/sigmoid/tanh hidden layers:

$$h^{(\ell)} = a^{(\ell-1)} W_\ell + \mathbf{1} b_\ell, \quad a^{(\ell)} = \sigma\big(h^{(\ell)}\big), \quad \ell = 1, \dots, L.$$

Backprop chain.

- For loss $\mathcal{L}$ and layer ℓ:

$$\frac{\partial \mathcal{L}}{\partial W_\ell} = \big(a^{(\ell-1)}\big)^\top \Delta^{(\ell)}, \quad \Delta^{(\ell)} = \Delta^{(\ell+1)} W_{\ell+1}^\top \odot \sigma'\big(h^{(\ell)}\big).$$

Unrolling from the output ($\Delta^{(L+1)}$ known):

$$\Delta^{(\ell)} = \Delta^{(L+1)} \left(\prod_{k=\ell+1}^{L} W_k^\top D^{(k-1)} \right), \quad D^{(k)} = \operatorname{diag}\big(\sigma'^{(k)}\big).$$

Vanishing gradients (cont'd)

Why vanishing.

- If $\|\cdot\|$ is any sub-multiplicative matrix norm,

$$\|\Delta^{(\ell)}\| \leq \|\Delta^{(L+1)}\| \prod_{k=\ell+1}^{L} \|W_k\| \, \|D^{(k-1)}\|.$$

- For sigmoid/tanh, $|\sigma'(u)| \leq 1/4$ (sigmoid) and $|\sigma'(u)| \leq 1$ with saturation ≈ 0 (tanh). Hence $\|D^{(k)}\| \ll 1$ often, so the product decays $\longrightarrow 0$ as depth grows.

Saturation view.

- In saturated regions, $\sigma'^{(k)} \approx 0 \Rightarrow \Delta^{(\ell)} \approx 0$. Parameters in early layers receive near-zero updates.

Plainly: when you multiply together derivatives that are less than $|1|$ in a backpropagation chain, they very quickly go to zero, which means the optimizer doesn't find an effect as it blocks learning about the parameter

Mitigations: ReLU and batch normalization

ReLU non-vanishing region.

$$\sigma(u) = \max\{0, u\}, \quad \sigma'(u) = \begin{cases} 1, & u > 0 \\ 0, & u < 0 \ (\text{“off”}) \end{cases}$$

In the active region $u > 0$, $D^{(k)} = \mathbf{I}$. Then along active paths,

$$\Delta^{(\ell)} \approx \Delta^{(L+1)} \prod_{k=\ell+1}^{L} W_k^\top,$$

so gradients do not systematically shrink due to activations.

- *Caveat:* dead ReLUs ($u < 0$ persistently) yield zero gradients.

Batch normalization (BN).

- Technique that uses normalization of pre-activations $h^{(\ell)}$ in a mini-batch to ensure gradient does not vanish

In sum...

Neural nets are "simply" flexible approximating functions

Feed-forward networks have nice properties

- Estimating network, forecasting, backpropagation can be done in parallell
- I.e., perfect for GPU computing

Downside: FFNN have no endogenous state-variables

- SR-SSM have sufficient statistics (state and latent variable beliefs)
  - Yield low dimensional represention
  - But, comes from a sequential structure so not parallelizable

HW 3 you will work with FFNNs

- Next lecture: Recurrent Neural Nets (RNNs) and other network structures that allow endogenous state-variables working in a sequential fashion