Lecture 4

Alternative Neural Net Structures:

Application to Problems in Financial Economics

Lars A. Lochstoer

UCLA Anderson School of Management

Overview

Economic constraints and deep learning

- Reframing the problem
- Bespoke neural nets

PyTorch

- The nn module
- Building bespoke neural nets: the ICNN model and option pricing

Recurrent neural nets

- Latent state-variable approach
- Gating and GRU
- LSTM

Economic Constraints and Deep Learning

Structure and efficiency

Adding structure to a problem can increase out-of-sample performance

- Parameter/architecture restrictions often leads to less estimation error, smaller parameter space
- This assumes the restrictions are empirically relevant
  - i.e., informative about real-world outcomes, distinguish signal from noise

In financial economics applications often natural constraints. Examples include:

1. Derivative pricing
   - No-arbitrage restrictions
   - These are important constraints any pricing model have to obey
     - ★ Otherwise you will post prices that you will lose money on immediately

2. Portfolio choice
   - Preferences put restrictions on portfolio weights
   - E.g., no short-sales, no derivatives, Value-at-Risk constraint, etc

Example: option pricing

In HW 3, we fit a neural net to the implied variance surface with no constraints on the model parameters

- But, these surfaces should obey standard no-arbitrage bounds!

Bounds to check include:

1. Positive risk-neutral probabilities: Call option convex in strike price $\left(\partial^2 C/\partial K^2 \geq 0\right)$
2. Calendar spread: Longer maturity option cannot be cheaper than short maturity (for calls on non-dividend paying underlying)

For parsimony, we will focus on 1. in the following example

Bespoke nn: Input-Convex Neural Net (ICNN)

Neural network with convexity of mapping $x \longmapsto f_\theta(x)$ guaranteed by activation functions and net setup, as well as restricting the sign of certain weights

Consider an ICNN for finding plain vanilla call option prices, $C(K, z)$

- K denotes option strike, z denotes other relevant inputs to determine the price (e.g., $z = (T, F_T)$)
- Example of two-layer network for $f_\theta(K; z)$ that imposes convexity in K is

$$\begin{aligned} u &= -K \\ a_1 &= \sigma\left(C_1 u + W_1 z + b_1\right), \quad C_1 \geq 0, \\ a_2 &= \sigma\left(T_2 a_1 + C_2 u + W_2 z + b_2\right), \quad T_2, C_2 \geq 0, \\ f_\theta(K; z) &= w_3' a_2 + b_3, \quad w_3 > 0, \end{aligned}$$

where $\sigma(\cdot)$ is ReLU (a convex function)

- All weights hitting u (including through $T_2 a_1$) are nonnegative, so each layer is convex and non-increasing in $K = -u$
- This is achieved through nonnegativity constraints on parameters C_1, C_2, T_2, w_3 and the use of ReLU

Estimation

The loss function is

$$\mathcal{L} = \sum_i \left(f_\theta \left(K_i ; z \right) - C^{data} \left(K_i \right) \right)^2 ,$$

where K_i refers to strike price i and $C^{data} \left(K_i \right)$ refers to the call option price in the data for strike K_i

The weight constraints are imposed inside the neural net estimation by re-parameterizing the weights

- Nonnegative of α: e.g., $\text{softplus}(\alpha)$
- Nonpositive of β: e.g., $-\text{softplus}(\beta)$

But, how do we impose such constraints inside a multi-layer neural net?

- Code it up yourself using ADAM to find parameters
  - ADAM is a refinement of Stochastic Gradient Descent (adaptive variant)
- Or, use flexibility built into PyTorch package!

PyTorch

(https://docs.pytorch.org/docs/stable/index.html)

PyTorch: Open source, deep learning

Some of the basic PyTorch components include:

- **Tensors** - N-dimensional arrays that serve as PyTorch's fundamental data structure. They support automatic differentiation, hardware acceleration, and provide a comprehensive API for mathematical operations.

- **Autograd** - PyTorch's automatic differentiation engine that tracks operations performed on tensors and builds a computational graph dynamically to be able to compute gradients.

- **Neural Network API** - A modular framework for building neural networks with pre-defined layers, activation functions, and loss functions. *The nn.Module base class provides a clean interface for creating custom network architectures with parameter management.*

- **DataLoaders** - Tools for efficient data handling that provide features like batching, shuffling, and parallel data loading. They abstract away the complexities of data preprocessing and iteration, allowing for optimized training loops.

PyTorch: install

Create pytorch_env

Install Spyder or similar there

- import torch (https://docs.pytorch.org/docs/stable/torch.html)
  - data structures for multi-dimensional tensors and mathematical operations on these
- import torch.nn (https://docs.pytorch.org/docs/stable/nn.html)
  - basic building blocks for neural nets (graphs)
  - includes specification of layers, activation function, etc.

Works with CPU, shines with GPU

- Tensor class built with GPU in mind

Implementing ICNN with PyTorch

Recall tree structure:

- Two-layer network for $f_\theta(K; z)$ that imposes convexity in K:

$$
\begin{aligned}
u &= -K \\
a_1 &= \sigma\left(C_1 u + W_1 z + b_1\right), \quad C_1 \geq 0, \\
a_2 &= \sigma\left(T_2 a_1 + C_2 u + W_2 z + b_2\right), \quad T_2, C_2 \geq 0, \\
f_\theta(K; z) &= w_3' a_2 + b_3, \quad w_3 > 0,
\end{aligned}
$$

- Loss function:

$$
\mathcal{L} = \sum_i \left(f_\theta(K_i; z) - C^{data}(K_i) \right)^2,
$$

  where parameter constraints are imposed inside neural net itself.
- $\sigma(\cdot)$ is ReLU

Simplest model

Let's estimate this model in the simplest case: one neural net per maturity

- That means the only relevant input across options is the strike price K
- Also, let's only use calls so only use the data with cp_flag = c
- The model is then:

$$\begin{aligned} a_1 &= \sigma\left(C_1 u + b_1\right), \quad C_1 \geq 0, \\ a_2 &= \sigma\left(T_2 a_1 + C_2 u + b_2\right), \quad T_2, C_2 \geq 0, \\ f_\theta\left(K; z\right) &= w_3' a_2 + b_3, \quad w_3 > 0, \end{aligned}$$

- Loss function:

$$\mathcal{L} = \sum_i \left(f_\theta\left(K_i\right) - C^{data}\left(K_i\right)\right)^2,$$

where parameter constraints are imposed inside neural net itself.

Implementing in PyTorch: torch.nn.module

Every model in PyTorch is effectively subclass of nn.Module.

Define neural network through:

1. Initialization (__init__): Define the layers and components of your network.
2. Forward Pass (forward): Specify how data flows through the layers of your network.
3. Parameter Management: Automatically tracks and optimizes model parameters.

Also provides:

- Lots of activation functions
- Several loss functions
- Optimizers
- +++

Implementing our specific neural net

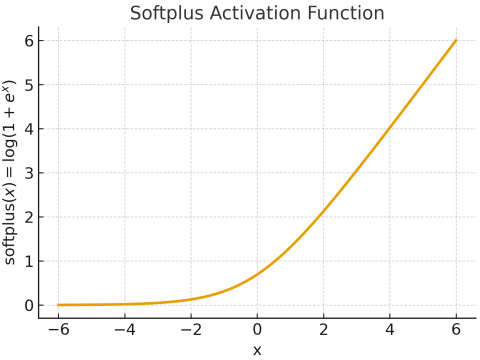
The instance of nn.module we specify

```
class ICNN1D_Mono(nn.Module):
    """
    u = -K_scaled
    a1 = ReLU( C1*u + b1 ),              C1 >= 0
    a2 = ReLU( T2*a1 + C2*u + b2 ),      T2,C2 >= 0
    g(u) = w3^T a2 + b3,                 w3 >= 0
    f(K) = g(-K_scaled)
    """
    def __init__(self, h1=64, h2=64):
        super().__init__()
        f32 = torch.float32
        self.C1_raw = nn.Parameter(torch.randn(h1, dtype=f32))
        self.b1     = nn.Parameter(torch.zeros(h1, dtype=f32))

        self.T2_raw = nn.Parameter(torch.randn(h2, h1, dtype=f32))
        self.C2_raw = nn.Parameter(torch.randn(h2, dtype=f32))
        self.b2     = nn.Parameter(torch.zeros(h2, dtype=f32))

        self.w3_raw = nn.Parameter(torch.randn(h2, dtype=f32))   # -> >= 0
        self.b3     = nn.Parameter(torch.tensor(0.0, dtype=f32))

        self.pos = nn.Softplus(beta=1.0, threshold=20.0)         # R -> (0,∞)
        self.act = nn.ReLU()

    def forward(self, K_scaled):           # (N,1)
        u  = -K_scaled
        C1 = self.pos(self.C1_raw)         # >= 0
        T2 = self.pos(self.T2_raw)         # >= 0
        C2 = self.pos(self.C2_raw)         # >= 0
        w3 = self.pos(self.w3_raw)         # >= 0

        a1 = self.act(u @ C1.unsqueeze(0) + self.b1)                # (N,H1)
        a2 = self.act(a1 @ T2.T + u @ C2.unsqueeze(0) + self.b2)    # (N,H2)
        g  = (a2 @ w3.unsqueeze(1)) + self.b3                       # (N,1)
        return g
```

Comments on code in last slide

self.pos defines a module instance .pos that is callable like a function

- We will use this to enforce positive coefficient constraints
- We apply the softplus to the argument, where softplus is



Comments on code two slides ago

self.parameter defines parameters of neural net

- These are what the optimizer will work with, eg C1_raw
- C1 is the softplus applied to C1_raw, which is a positive vector

self.act defines .act to apply the ReLU activation function

def *forward*: defines the forward pass of the net

- h1 and h2 is the number of units in the two layers where

  a1 = self.act(u @ C1.unsqueeze(0) + self.b1) # (N,H1)
  a2 = self.act(a1 @ T2.T + u @ C2.unsqueeze(0) + self.b2) # (N,H2)
  g = (a2 @ w3.unsqueeze(1)) + self.b3 # (N,1)

defines the neural net in sequence

Next, estimating the model

```python
def train_one_maturity(df_m, exdate, verbose=False):
    x = torch.tensor(df_m["K_scaled"].to_numpy(np.float32)).reshape(-1,1)
    y = torch.tensor(df_m["y_obs"].to_numpy(np.float32)).reshape(-1,1)

    dataset = TensorDataset(x, y)
    n = len(dataset)
    n_val = max(1, int(VAL_SPLIT * n))
    n_trn = n - n_val
    train_set, val_set = random_split(dataset, [n_trn, n_val],
                                      generator=torch.Generator().manual_seed(SEED))
    train_loader = DataLoader(train_set, batch_size=BATCH, shuffle=True)
    val_loader   = DataLoader(val_set,   batch_size=4096, shuffle=False)

    model = ICNN1D_Mono(h1=H1, h2=H2).to(DEVICE)
    opt = torch.optim.Adam(model.parameters(), lr=LR_INIT)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        opt, mode="min", factor=0.5, patience=80, min_lr=1e-6
    )
    mse = nn.MSELoss()

    best_val, best_state, no_imp = float("inf"), None, 0

    # ---- Adam phase ----
    for epoch in range(1, MAX_EPOCHS+1):
        model.train()
        trn = 0.0
        for xb, yb in train_loader:
            xb = xb.to(DEVICE); yb = yb.to(DEVICE)
            opt.zero_grad(set_to_none=True)
            yhat = model(xb)
            loss = mse(yhat, yb)
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
            opt.step()
            trn += loss.item() * len(xb)
        trn /= max(1, n_trn)

        model.eval()
        with torch.no_grad():
            vloss = 0.0
            for xb, yb in val_loader:
                xb = xb.to(DEVICE); yb = yb.to(DEVICE)
                vloss += mse(model(xb), yb).item() * len(xb)
            vloss /= max(1, n_val)
```

Comments on code in last slide

Input data as tensors

- Arrays for torch module, designed for GPU computing
- I only have CPU so that's what it will use

Create:

- Batches
- Sets Adam optimizer (for of stochastic gradient descent)

Loops through Epochs

- model.train() sets model in training mode (keeps track of batch normalizations etc, which are not needed for evaluation mode)
- loss is mse (nn.MSEloss())
- gets gradient through backward propagation in loss.backward()
- Updates parameters in opt.step()

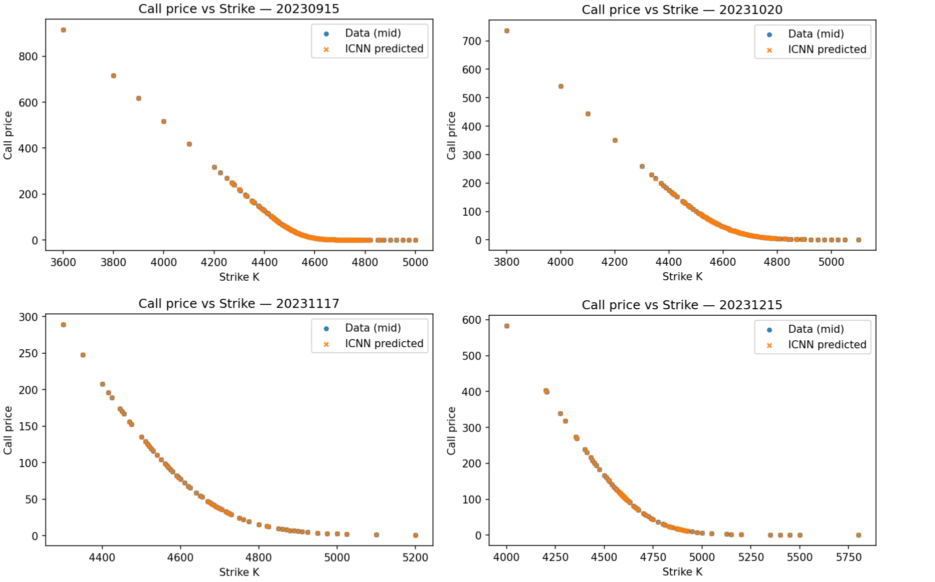
Some details on estimation

- Use midpoint prices, only calls, volume > 0, $|\ln K/S| < 0.3$
- Estimate one neural net per maturity (for now)
- Set H1 $= 64$, H2 $= 32$ (Layers per hidden layer)
- Split 20% validation, 80% training sample

Results:

Maturity	*N*	*RMSE*	R^2
20230915	110	0.0896	1.000
20231020	94	0.0257	1.000
20231117	64	0.0256	1.000
20231215	68	0.0317	1.000

Fit per maturity



Fitting full model for both K and T

If we want to price options for arbitrary K and T, we need a price function $f_\theta\left(K, T; z\right)$

Achieve this by modeling a "T-gated" network

- Let coefficients be a function of maturity T
  - This extra function is a "head" as it is like a branch that "heads off" from the trunk of the neural net
- T then governs the "flow" of a coefficient (it's magnitude) and the coefficients $C\left(T\right)$ that modulate the inputs (u and a) are then the gate

Fitting full model for both K and T

The gates modulate the backbone, the heads are the hyper networks that achieve this:

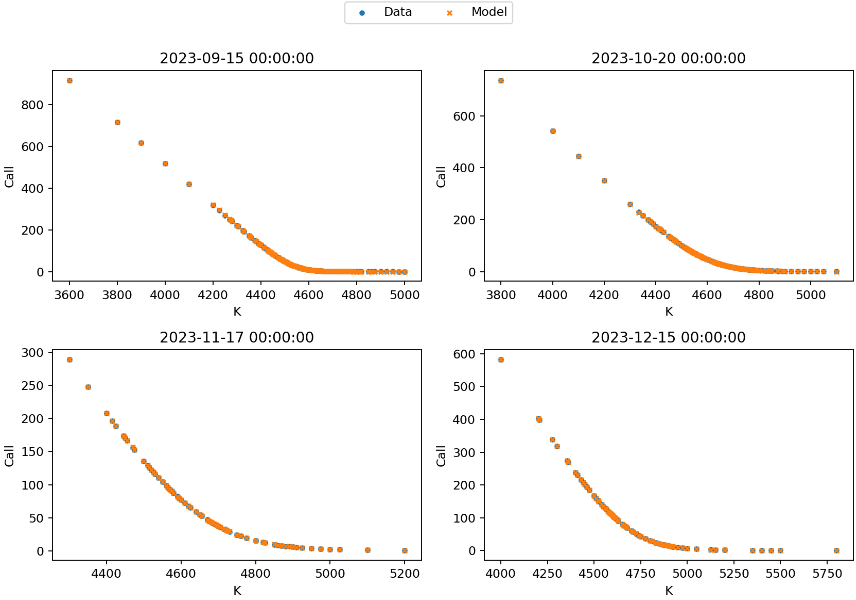
$$
\begin{aligned}
a_1 &= \sigma\left(C_1\left(T\right) \odot u + V_1\left(T\right) + b_1\right), \quad C_1 \geq 0, \\
a_2 &= \sigma\left(T_2\left(a_1 \odot G_1\left(T\right)\right) + C_2 u + V_2\left(T\right) + b_2\right), \quad T_2, C_2 \geq 0, \\
f_\theta\left(K; z\right) &= w_3' a_2 + v_3\left(T\right) + b_3, \quad w_3 > 0,
\end{aligned}
$$

where

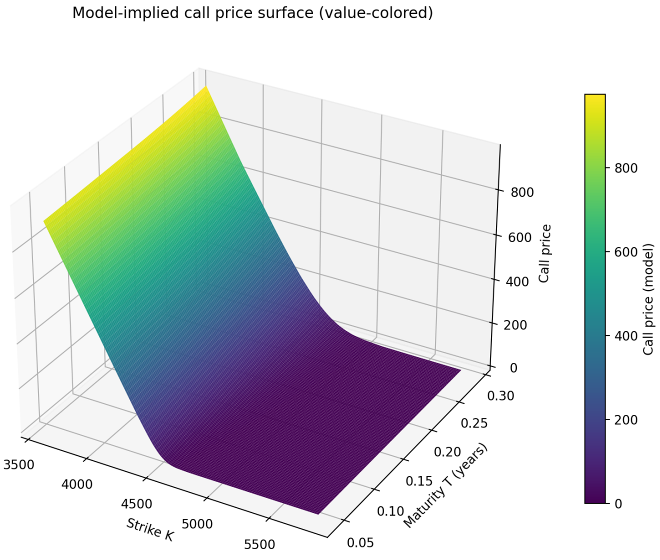
$$
\begin{aligned}
C_1\left(T\right) &= \text{softplus}\left(\alpha_1 + A_1 T\right), \quad G_1\left(T\right) = \text{softplus}\left(\alpha_2 + A_2 T\right), \\
V_l\left(T\right) &= \beta_l + B_l T, \quad l \in \{1, 2, 3\}, \quad w_3 = \text{softplus}\left(\alpha_3 + A_3 T\right).
\end{aligned}
$$

- In sum: Coefficients are modulated by T through affine functions.
- This enables the model to fit call prices per maturity and provide a model where one can feed in arbitrary K and T
- See code on BruinLearn. Note, without parameter constraints on the "heads", there may be calendar arbitrage

Fit per maturity: model with both K and T



Call price surface: model with both K and T



Recurrent Neural Nets
GRU and LSTM

State variables and economic modeling

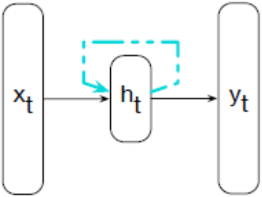
We typically have settings where there is a vector of state variables that determine the conditional distribution of future outcomes

- E.g., the latent state in SSM, the filtering probabilities in SR
- Forecasts are functions of the current value of these state variables
  - Simpler than using the full history of observations + priors

Can neural nets learn such state variables?

- Yes, the general class of recurrent neural networks (RNNs) can
  - GRU
  - LSTM

Baseline RNN



Here, x_t are inputs at time t, h_t is the hidden layer outputs, and y_t is the final output layer result

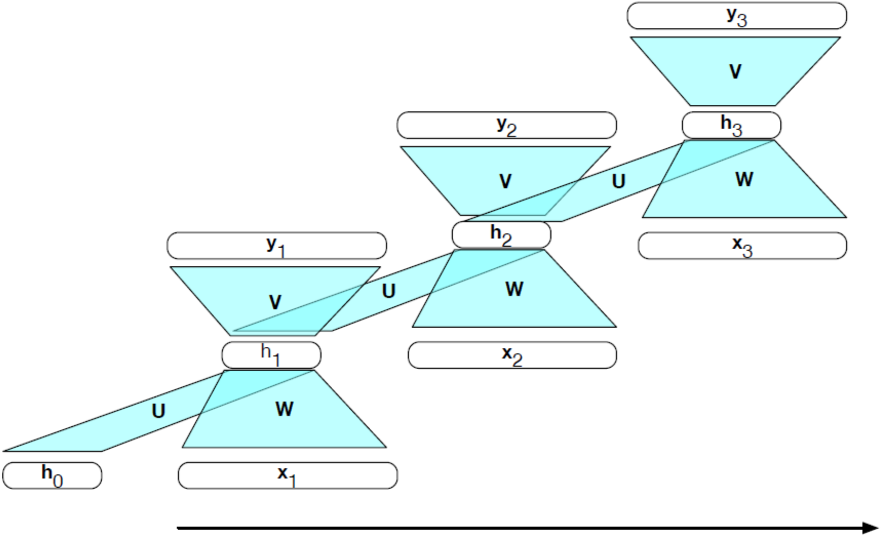
- Note the loop: h_t is an output at time t, but an input at time $t+1$
- Generic mathematical description of network:

$$h_t = g\left(Uh_{t-1} + Wx_t\right),$$
$$y_t = f\left(Vh_t\right)$$

- Here there is a strong sense of a sequence in time t (see next slide)
  - Thus, RNNs are depth-T deep learning networks with layers unrolled in time

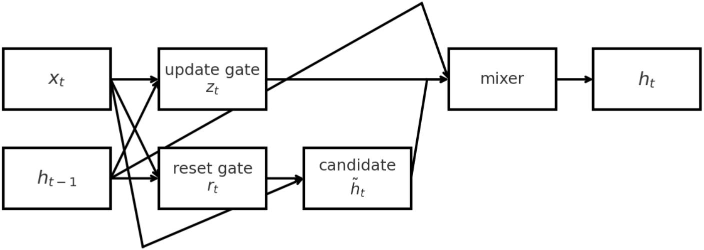
Baseline RNN: Unroll in time

Sequential nature of network plotted unrolled in time:



More complicated RNN: The GRU

GRU: gated recurrent unit



GRU = update gate + reset gate + candidate + mixer

- Update gate: How much to forget vs keep
- Reset gate: How much past to expose the candidate $\tilde{h}_t$ to
- Mixer: convex combination of the hold state and the new proposal

The GRU in equations

$$
\begin{aligned}
z_t &= \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) && \text{update gate} \\
r_t &= \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) && \text{reset gate} \\
\tilde{h}_t &= \tanh\left(W_h x_t + U_h\left(r_t \odot h_{t-1}\right) + b_h\right) && \text{candidate} \\
h_t &= \left(1 - z_t\right) \odot h_{t-1} + z_t \odot \tilde{h}_t && \text{mixer}
\end{aligned}
$$

- Update gate: How much to forget vs keep
  - if $z_t = 1$, forget previous state, full update
- Reset gate: How much past to expose the candidate $\tilde{h}_t$ to
  - if $r_t = 1$ full use of past in finding candidate (the $r_t \odot h_{t-1}$ term)
- Mixer: convex combination of the old state and the new proposal

GRU approximating the HMM

To see how this model approximates our baseline 2-state SR model more efficiently than a feed-forward network, note that the state variable $\mathbf{p}_t\ (s_t)$ is all that is needed for forecasting any future outcome

- Thus, there is an endogenous state-variable that is computed using Bayes rule and optimal filtering of the observations y^t
  - This state variable is a *sufficient statistic*
- 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 $\pi_{ij} = \Pr\left(S_t = j | S_{t-1} = i\right)$

The GRU will aim to approximate $h_t \approx h\left(p_t\left(s_t = 1\right)\right)$

- $h\left(\cdot\right)$ here just denotes some function to be determined so that the information in h_t can be used to back out the sufficient statistic $p_t\left(s_t = 1\right)$

Generic GRU to learn DGP

DGP = data-generating process

- input: x_t =standardize(y_t)
- hidden state $h_t \in \mathbb{R}^m$, say $m = 8$
- GRU as before:

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right), \quad r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right),$$
$$\tilde{h}_t = \tanh\left(W_h x_t + U_h\left(r_t \odot h_{t-1}\right) + b_h\right), \quad h_t = \left(1 - z_t\right) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

- Output head:

$$p_t\left(1\right) = sigmoid\left(W_p h_t + b_p\right)$$

- Loss function:

$$\mathcal{L}\left(\theta\right) = -\sum_{t=1}^{T-1} \ln q_\theta\left(y_{t+1} | h_t\right),$$

where

$$q_\theta\left(y_{t+1} | h_t\right) = p_t\left(1\right) N\left(y_{t+1}; \mu_1, \sigma_1^2\right) + \left(1 - p_t\left(1\right)\right) N\left(y_{t+1}; \mu_2, \sigma_2^2\right)$$

Why this works: intuition

Linearize: $\tanh(u) \approx u$, if u is small:

$$h_t \approx \left[(I - Z_t) + Z_t U_h R_t\right] h_{t-1} + Z_t \left(W_h x_t + b_h\right),$$

where $Z_t = diag\left(z_t\right)$ and $R_t = diag\left(r_t\right)$

The weight on the past belief is a function of z_t (update) and r_t (reset)

The weight on new observations, x_t, depends on z_t (and parameters)

- If we are sure we are in regime 1 and this regime is very persistent, small weight on data as we are unlikely to learn anything new
- If we are very unsure about what regime we are in, high weight on data

Example code for previous GRU: page 1

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

y = torch.randn(500)

class SRGRU(nn.Module):
    def __init__(self, m, m1, m2, r1, r2):
        super().__init__()
        self.gru  = nn.GRU(input_size=1, hidden_size=m, num_layers=1, batch_first=True)
        self.w_pt = nn.Linear(m, 1)                     # logit for state 2
        self.mu   = nn.Parameter(torch.tensor([m1, m2], dtype=torch.float))
        self.rho  = nn.Parameter(torch.tensor([r1, r2], dtype=torch.float))  # unconstrained

    def head(self, H):                                  # H: [B, L-1, m]
        pt2 = torch.sigmoid(self.w_pt(H))               # [B, L-1, 1]
        pt  = torch.cat([1-pt2, pt2], dim=-1)           # [B, L-1, 2]
        sigma = F.softplus(self.rho) + 1e-3             # [2], positive stds
        return pt, self.mu, sigma

    def forward(self, y):                               # y: [B, L, 1]
        H, _ = self.gru(y[:, :-1, :])                   # [B, L-1, m]
        pt, mu, sigma = self.head(H)                    # params for y_{1..L-1}
        return pt, mu, sigma
```

Example code for previous GRU: page 2

```python
def logmixnorm(y_next, pt, mu, sigma):
    # y_next: [B, L-1], pt: [B, L-1, 2], mu/sigma: [2]
    y = y_next.unsqueeze(-1)                              # [B, L-1, 1]
    z = (y - mu) / sigma                                  # [B, L-1, 2]
    log_comp = -0.5*(z**2) - torch.log(sigma) - 0.5*torch.log(torch.tensor(2*3.141592653589793))
    return torch.logsumexp(torch.log(pt) + log_comp, dim=-1)      # [B, L-1]

def nll(model, y):
    pt, mu, sigma = model(y)                              # pt: [B, L-1, 2]
    y_next = y[:, 1:, :].squeeze(-1)                      # [B, L-1]
    return -(logmixnorm(y_next, pt, mu, sigma).mean())

# Training
model = SRGRU(m=8, m1=0.0, m2=1.0, r1=0.0, r2=0.0)
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

y_batch = y.view(1, -1, 1).float()
opt.zero_grad()
loss = nll(model, y_batch)
loss.backward()
opt.step()
```

The LSTM: use cases

What if the DGP has multifrequency dynamics? For instance:

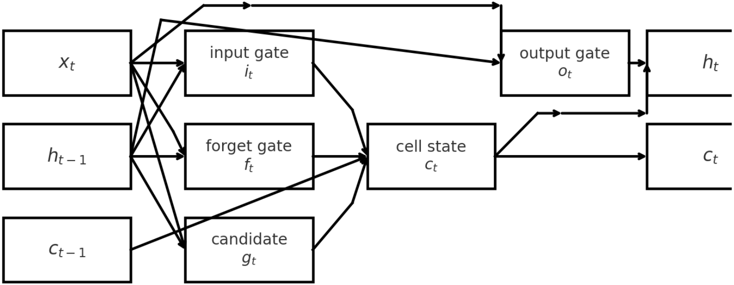
$$y_t = \mu_{s_t} + \sigma_t \varepsilon_t,$$
$$h_{t+1} = \alpha + \beta h_t + \eta_{t+1}, \quad \sigma_t^2 = e^{h_t},$$

where ε_t, η_t are iid standard Normals.

- μ_{s_t} and h_t operate on different time scales
- LSTM can capture these different frequencies

For such models, the Long Short-Term Memory (LSTM) network is often better

The LSTM: architecture



Vanilla LSTM: edges {x_t,h_{t-1}}→{i_t,f_t,o_t,g_t}; {i_t,f_t,g_t}→c_t; c_{t-1}→c_t; o_t→h_t; c_t→h_t.

The cell or context vector c_t captures very persistent features, while h_t captures faster moving features through time-varying exposures o_t

The LSTM: architecture

$$
\begin{aligned}
i_t &= \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) && \text{input gate} \\
f_t &= \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) && \text{forget gate} \\
o_t &= \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) && \text{output gate} \\
g_t &= \tanh\left(W_g x_t + U_g h_{t-1} + b_g\right) && \text{candidate} \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t && \text{cell (context) state} \\
h_t &= o_t \odot \tanh\left(c_t\right) && \text{exposed state}
\end{aligned}
$$

The cell or context vector c_t captures very persistent features, while h_t captures faster moving features through time-varying exposures o_t

Conclusion on RNNs

RNNs are natural candidates for time-series applications

- Notion of sequence of time $h_{t-1} \longrightarrow h_t \longrightarrow h_{t+1}$
- Sufficient statistics (state variables) are modeled as latent endogenous variables of the system
  - In the end, as always, some function of past and current observations (and parameters)

Feed-forward nets need us to use input variables with a window of time series length J to approximate the sufficient stats

What type of network you choose is problem-specific

- Want good out of sample performance along with fast computing time
- Think first, then try several architectures that capture features you want