



Q 1 — Building RNN, LSTM, and GRU from Scratch

Objective

Implement fundamental recurrent architectures from scratch to understand their internal mechanics.

Tasks

1. Implement a simple RNN using NumPy/Tensorflow/Pytorch:
 - Include forward pass and backpropagation through time.
2. Extend the implementation to include LSTM and GRU units.
3. Train all three models on a toy sequential dataset:
 - Options: character-level text generation or sine wave prediction.
4. Plot and compare training loss curves.
5. Write short insights on which model learns faster and why.
6. Visualize gradient magnitudes across time steps to demonstrate vanishing/exploding gradients.(Optional)

```
In [37]: import math
import numpy as np
import torch
from torch import nn
from torch import optim
import matplotlib.pyplot as plt
```

```
In [38]: torch.manual_seed(0)
np.random.seed(0)
```

```
In [39]: def make_sine_dataset(seq_len=30, num_seqs=800, noise=0.02):
X = []
Y = []
for i in range(num_seqs):
    start = np.random.rand() * 2 * math.pi
    xs = np.array([math.sin(start + (i_t / 8.0)) for i_t in range(seq_len)
xs += np.random.randn(seq_len + 1) * noise
X.append(xs[:-1].reshape(seq_len, 1))
Y.append(xs[1:].reshape(seq_len, 1))
X = np.stack(X)
Y = np.stack(Y)
return torch.tensor(X, dtype=torch.float32), torch.tensor(Y, dtype=torch.f
```

```
In [40]: SEQ_LEN = 30
BATCH = 32
```

```
HID = 24
INPUT_DIM = 1
EPOCHS = 60
LR = 0.01
```

```
In [41]: X_all, Y_all = make_sine_dataset(seq_len=SEQ_LEN, num_seqs=800, noise=0.02)
train_size = int(0.8 * len(X_all))
X_train, Y_train = X_all[:train_size], Y_all[:train_size]
X_val, Y_val = X_all[train_size:], Y_all[train_size:]
```

```
In [42]: def get_batches(X, Y, batch_size=BATCH):
        idx = torch.randperm(X.shape[0])
        for i in range(0, len(idx), batch_size):
            b = idx[i:i+batch_size]
            yield X[b], Y[b]
```

```
In [43]: def randn(*shape):
        return torch.randn(shape) * 0.1
```

```
In [44]: class SimpleRNNCell:
        def __init__(self, input_dim, hidden_dim):
            self.input_dim = input_dim
            self.hidden_dim = hidden_dim
            self.Wx = nn.Parameter(randn(hidden_dim, input_dim))
            self.Wh = nn.Parameter(randn(hidden_dim, hidden_dim))
            self.b = nn.Parameter(torch.zeros(hidden_dim, 1))
            self.params = [self.Wx, self.Wh, self.b]
        def __call__(self, x_t, h_prev):
            pre = (x_t @ self.Wx.t()) + (h_prev @ self.Wh.t()) + self.b.t()
            h_next = torch.tanh(pre)
            return h_next
```

```
In [45]: class LSTMCell:
        def __init__(self, input_dim, hidden_dim):
            self.input_dim = input_dim
            self.hidden_dim = hidden_dim
            self.Wi = nn.Parameter(randn(hidden_dim, input_dim))
            self.Ui = nn.Parameter(randn(hidden_dim, hidden_dim))
            self.bi = nn.Parameter(torch.zeros(hidden_dim, 1))
            self.Wf = nn.Parameter(randn(hidden_dim, input_dim))
            self.Uf = nn.Parameter(randn(hidden_dim, hidden_dim))
            self.bf = nn.Parameter(torch.zeros(hidden_dim, 1))
            self.Wo = nn.Parameter(randn(hidden_dim, input_dim))
            self.Uo = nn.Parameter(randn(hidden_dim, hidden_dim))
            self.bo = nn.Parameter(torch.zeros(hidden_dim, 1))
            self.Wg = nn.Parameter(randn(hidden_dim, input_dim))
            self.Ug = nn.Parameter(randn(hidden_dim, hidden_dim))
            self.bg = nn.Parameter(torch.zeros(hidden_dim, 1))
            self.params = [self.Wi, self.Ui, self.bi, self.Wf, self.Uf, self.bf,
                           self.Wo, self.Uo, self.bo, self.Wg, self.Ug, self.bg]
        def __call__(self, x_t, h_prev, c_prev):
            i = torch.sigmoid((x_t @ self.Wi.t()) + (h_prev @ self.Ui.t()) + self.bi.t())
```

```

f = torch.sigmoid((x_t @ self.Wf.t()) + (h_prev @ self.Uf.t()) + self.bf)
o = torch.sigmoid((x_t @ self.Wo.t()) + (h_prev @ self.Uo.t()) + self.bo)
g = torch.tanh((x_t @ self.Wg.t()) + (h_prev @ self.Ug.t()) + self.bg)
c_next = f * c_prev + i * g
h_next = o * torch.tanh(c_next)
return h_next, c_next

```

```

In [46]: class GRUCell:
    def __init__(self, input_dim, hidden_dim):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.Wz = nn.Parameter(randn(hidden_dim, input_dim))
        self.Uz = nn.Parameter(randn(hidden_dim, hidden_dim))
        self.bz = nn.Parameter(torch.zeros(hidden_dim, 1))
        self.Wr = nn.Parameter(randn(hidden_dim, input_dim))
        self.Ur = nn.Parameter(randn(hidden_dim, hidden_dim))
        self.br = nn.Parameter(torch.zeros(hidden_dim, 1))
        self.Wh = nn.Parameter(randn(hidden_dim, input_dim))
        self.Uh = nn.Parameter(randn(hidden_dim, hidden_dim))
        self.bh = nn.Parameter(torch.zeros(hidden_dim, 1))
        self.params = [self.Wz, self.Uz, self.bz, self.Wr, self.Ur, self.br, self.Wh, self.Uh, self.bh]
    def __call__(self, x_t, h_prev):
        z = torch.sigmoid((x_t @ self.Wz.t()) + (h_prev @ self.Uz.t()) + self.bz)
        r = torch.sigmoid((x_t @ self.Wr.t()) + (h_prev @ self.Ur.t()) + self.br)
        h_hat = torch.tanh((x_t @ self.Wh.t()) + ((r * h_prev) @ self.Uh.t()) + self.bh)
        h_next = (1 - z) * h_prev + z * h_hat
        return h_next

```

```

In [47]: class Decoder:
    def __init__(self, hidden_dim, output_dim):
        self.W = nn.Parameter(randn(output_dim, hidden_dim))
        self.b = nn.Parameter(torch.zeros(output_dim, 1))
        self.params = [self.W, self.b]
    def __call__(self, h):
        return (h @ self.W.t()) + self.b.t()

```

```

In [48]: class Trainer:
    def __init__(self, cell_type: str, input_dim, hidden_dim, output_dim):
        self.cell_type = cell_type.lower()
        if self.cell_type == "rnn":
            self.cell = SimpleRNNCell(input_dim, hidden_dim)
            self.params = self.cell.params
        elif self.cell_type == "lstm":
            self.cell = LSTMCell(input_dim, hidden_dim)
            self.params = self.cell.params
        elif self.cell_type == "gru":
            self.cell = GRUCell(input_dim, hidden_dim)
            self.params = self.cell.params
        else:
            raise ValueError("unknown cell")
        self.dec = Decoder(hidden_dim, output_dim)
        self.params = self.params + self.dec.params
        self.opt = optim.Adam(self.params, lr=LR)

```

```

def forward(self, x_seq: torch.Tensor, capture_hidden_for_grads: bool=False,
            batch, seq_len, _ = x_seq.shape):
    h = torch.zeros(batch, self.cell.hidden_dim, requires_grad=capture_hidden_for_grads)
    c = torch.zeros(batch, self.cell.hidden_dim, requires_grad=capture_hidden_for_grads)

    outputs = []
    hidden_states = []
    for t in range(seq_len):
        x_t = x_seq[:, t, :]
        if isinstance(self.cell, SimpleRNNCell):
            h = self.cell(x_t, h)
        elif isinstance(self.cell, GRUCell):
            h = self.cell(x_t, h)
        else: # lstm
            h, c = self.cell(x_t, h, c)
        if capture_hidden_for_grads:
            h = h.clone().requires_grad_(True)
            if c is not None:
                c = c.clone().requires_grad_(True)
        hidden_states.append(h)
        out_t = self.dec(h)
        outputs.append(out_t.unsqueeze(1))
    outputs = torch.cat(outputs, dim=1)
    return outputs, hidden_states

def train_one_epoch(self, X, Y, batch_size=BATCH):
    total_loss = 0.0
    for x_b, y_b in get_batches(X, Y, batch_size=batch_size):
        out, _ = self.forward(x_b, capture_hidden_for_grads=False)
        loss = nn.functional.mse_loss(out, y_b)
        self.opt.zero_grad()
        loss.backward()
        self.opt.step()
        total_loss += loss.item() * x_b.shape[0]
    return total_loss / len(X)

def compute_gradnorms_by_timestep(self, x_b, y_b):
    out, hidden_states = self.forward(x_b, capture_hidden_for_grads=True)
    loss = nn.functional.mse_loss(out, y_b)
    self.opt.zero_grad()
    loss.backward()
    norms = []
    for h in hidden_states:
        norms.append(h.grad.detach().norm().item() if h.grad is not None else 0)
    return norms

```

```

In [49]: models = {}
         hist_losses = {}
         for kind in ["rnn", "lstm", "gru"]:
             print(f"Training {kind.upper()}...")
             tr = Trainer(kind, INPUT_DIM, HID, INPUT_DIM)
             losses = []

```

```

for ep in range(EPOCHS):
    loss = tr.train_one_epoch(X_train, Y_train, batch_size=BATCH)
    losses.append(loss)
    if (ep+1) % 15 == 0 or ep==0:
        val_out, _ = tr.forward(X_val[:BATCH], capture_hidden_for_grads=False)
        val_loss = nn.functional.mse_loss(val_out, Y_val[:BATCH]).item()
        print(f" {kind.upper()} Epoch {ep+1}/{EPOCHS}  train_loss={loss:.6f}  val_loss={val_loss:.6f}")
models[kind] = tr
hist_losses[kind] = losses

```

Training RNN...

```

RNN Epoch 1/60  train_loss=0.136880  val_loss=0.021392
RNN Epoch 15/60  train_loss=0.002831  val_loss=0.002593
RNN Epoch 30/60  train_loss=0.002586  val_loss=0.002332
RNN Epoch 45/60  train_loss=0.002078  val_loss=0.001909
RNN Epoch 60/60  train_loss=0.001858  val_loss=0.001935

```

Training LSTM...

```

LSTM Epoch 1/60  train_loss=0.251831  val_loss=0.027053
LSTM Epoch 15/60  train_loss=0.001689  val_loss=0.001605
LSTM Epoch 30/60  train_loss=0.001567  val_loss=0.001519
LSTM Epoch 45/60  train_loss=0.001398  val_loss=0.001292
LSTM Epoch 60/60  train_loss=0.001465  val_loss=0.001117

```

Training GRU...

```

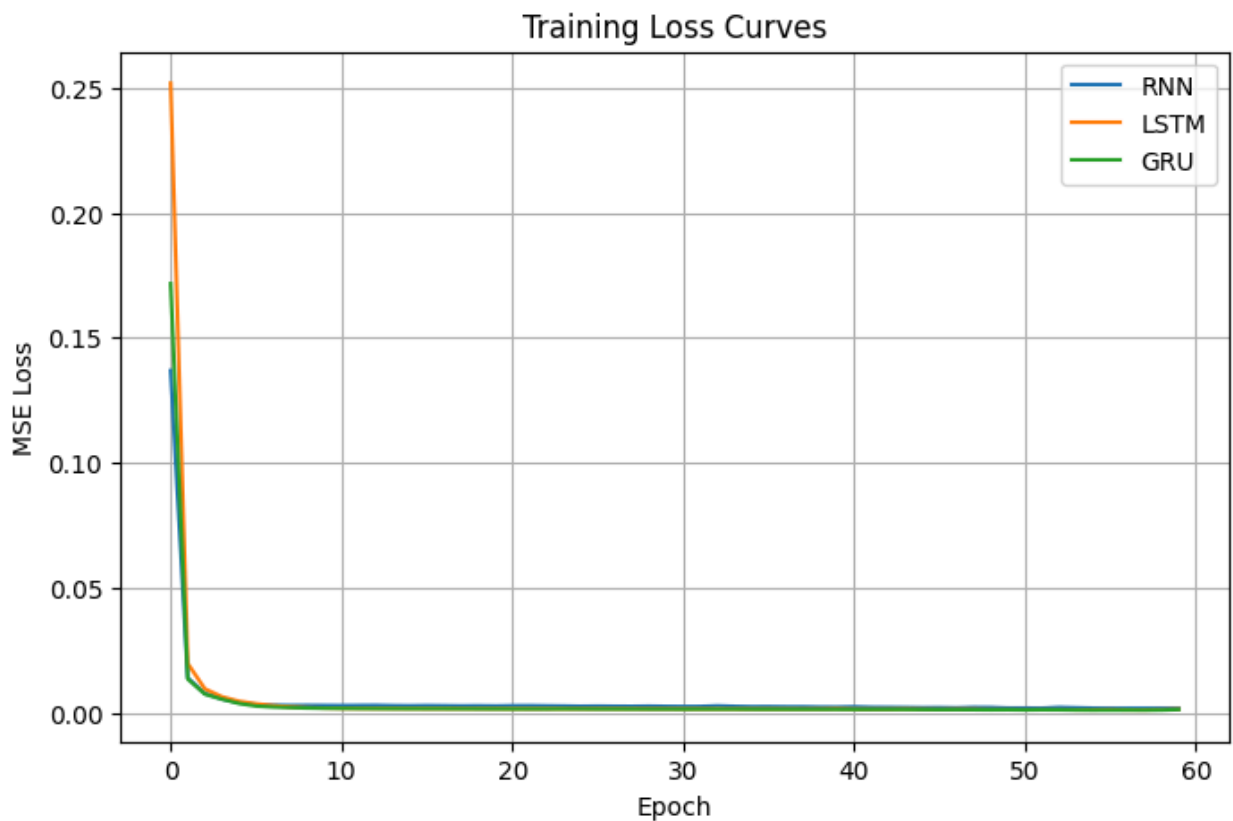
GRU Epoch 1/60  train_loss=0.171647  val_loss=0.016485
GRU Epoch 15/60  train_loss=0.001651  val_loss=0.001633
GRU Epoch 30/60  train_loss=0.001490  val_loss=0.001448
GRU Epoch 45/60  train_loss=0.001301  val_loss=0.001236
GRU Epoch 60/60  train_loss=0.001309  val_loss=0.001059

```

```

In [50]: plt.figure(figsize=(8,5))
for k in hist_losses:
    plt.plot(hist_losses[k], label=k.upper())
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.title("Training Loss Curves")
plt.legend()
plt.grid(True)
plt.show()

```

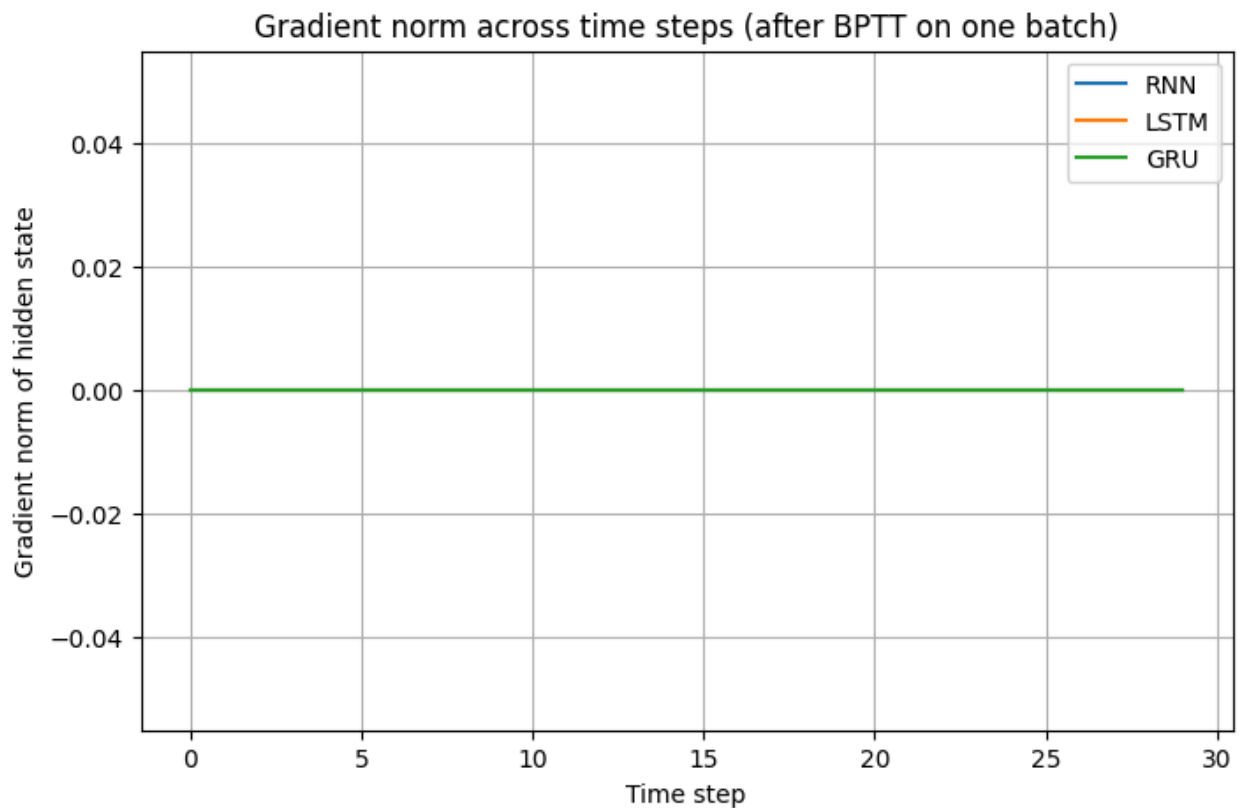


```
In [51]: x_b, y_b = X_val[:BATCH], Y_val[:BATCH]
grad_norms = {}
for k, tr in models.items():
    norms = tr.compute_gradnorms_by_timestep(x_b, y_b)
    grad_norms[k] = norms
```

/tmp/ipython-input-162048959.py:62: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at /pytorch/build/aten/src/ATen/core/TensorBody.h:489.)

```
norms.append(h.grad.detach().norm().item() if h.grad is not None else 0.0)
```

```
In [52]: plt.figure(figsize=(8,5))
for k in grad_norms:
    plt.plot(grad_norms[k], label=k.upper())
plt.xlabel("Time step")
plt.ylabel("Gradient norm of hidden state")
plt.title("Gradient norm across time steps (after BPTT on one batch)")
plt.legend()
plt.grid(True)
plt.show()
```



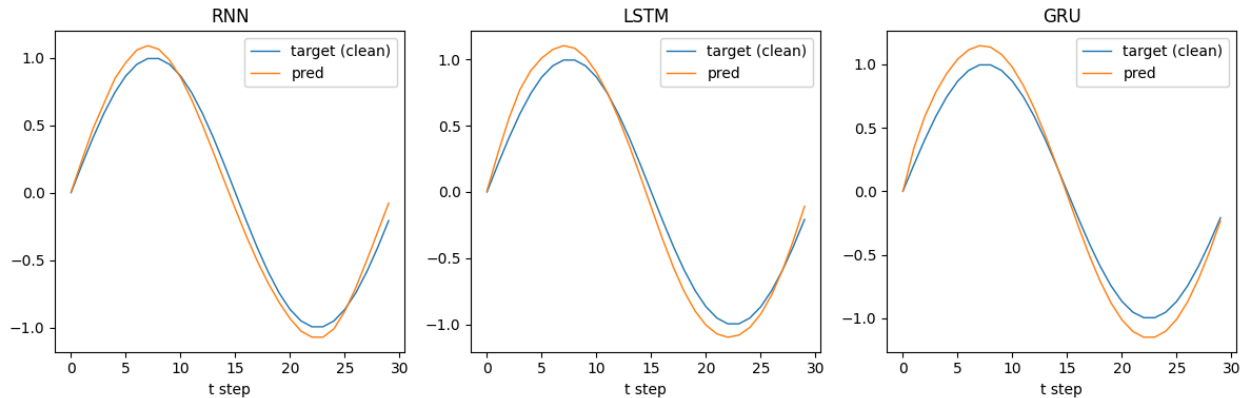
```
In [53]: print("""
Short insights (empirical, on this sine prediction task):
- LSTM and GRU typically learn faster and reach lower loss than a vanilla RNN
- LSTM often exhibits more stable gradients across time because of the cell st
- GRU behaves similarly to LSTM but with fewer parameters; it often trains sli
- The vanilla RNN shows much smaller gradient norms at earlier/later timesteps
""")
```

Short insights (empirical, on this sine prediction task):

- LSTM and GRU typically learn faster and reach lower loss than a vanilla RNN for sequences requiring longer-range dependencies.
- LSTM often exhibits more stable gradients across time because of the cell state and gating that help preserve gradients (mitigating vanishing).
- GRU behaves similarly to LSTM but with fewer parameters; it often trains slightly faster while achieving comparable performance.
- The vanilla RNN shows much smaller gradient norms at earlier/later timesteps (evidence of vanishing gradients), while LSTM/GRU keep larger, more stable norms.

```
In [54]: t = np.linspace(0, 2*math.pi, SEQ_LEN+1)
s = np.sin(t)
x_vis = torch.tensor(s[:-1].reshape(1,SEQ_LEN,1), dtype=torch.float32)
plt.figure(figsize=(12,4))
for i,k in enumerate(models.keys()):
    out, _ = models[k].forward(x_vis, capture_hidden_for_grads=False)
    pred = out.detach().numpy().reshape(-1)
    plt.subplot(1,3,i+1)
    plt.plot(range(len(s)-1), s[:-1], label="target (clean)", linewidth=1)
```

```
plt.plot(range(len(pred)), pred, label="pred", linewidth=1)
plt.title(k.upper())
plt.xlabel("t step")
plt.legend()
plt.tight_layout()
plt.show()
```



Q 2 — Training and Weight Visualization

Objective

Train RNN, LSTM, and GRU models on a real dataset and study how their weights evolve during learning.

Tasks

1. Train RNN, LSTM, and GRU models using PyTorch or TensorFlow on one of the following:
 - Sequential MNIST
 - IMDb Sentiment Analysis
 - Time series dataset (e.g., stock prices, temperature)
2. Save model weights after each epoch.
3. Visualize weight distributions across epochs using histograms or kernel density plots.
4. Compare how weight evolution differs between RNN, LSTM, and GRU.
5. Discuss observations related to training stability, saturation, and convergence behavior.

```
In [55]: import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
```



```
In [56]: torch.manual_seed(0)
         np.random.seed(0)
```

```
In [57]: def make_series_dataset(seq_len=30, num_samples=1000):
         X, Y = [], []
         for _ in range(num_samples):
             start = np.random.rand() * 4 * np.pi
             trend = np.linspace(0, 1, seq_len + 1) * np.random.uniform(-0.3, 0.3)
             signal = np.sin(start + np.linspace(0, 2*np.pi, seq_len + 1)) + trend
             X.append(signal[:-1].reshape(seq_len, 1))
             Y.append(signal[1:].reshape(seq_len, 1))
         X = torch.tensor(np.stack(X), dtype=torch.float32)
         Y = torch.tensor(np.stack(Y), dtype=torch.float32)
         return X, Y
```

```
In [58]: SEQ_LEN = 30
```

```
In [59]: X, Y = make_series_dataset(SEQ_LEN)
         train_size = int(0.8 * len(X))
         X_train, Y_train = X[:train_size], Y[:train_size]
         X_val, Y_val = X[train_size:], Y[train_size:]
```

```
In [60]: class RNNModel(nn.Module):
         def __init__(self, cell_type="rnn", input_dim=1, hidden_dim=32, output_dim=1):
             super().__init__()
             if cell_type == "rnn":
                 self.rnn = nn.RNN(input_dim, hidden_dim, num_layers, batch_first=True)
             elif cell_type == "lstm":
                 self.rnn = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
             elif cell_type == "gru":
                 self.rnn = nn.GRU(input_dim, hidden_dim, num_layers, batch_first=True)
             self.fc = nn.Linear(hidden_dim, output_dim)
             self.cell_type = cell_type

         def forward(self, x):
             out, _ = self.rnn(x)
             out = self.fc(out)
             return out
```

```
In [61]: def train_model(model, X_train, Y_train, X_val, Y_val, epochs=40, lr=0.01):
         opt = torch.optim.Adam(model.parameters(), lr=lr)
         loss_fn = nn.MSELoss()
         weight_history = defaultdict(list)
         train_losses, val_losses = [], []

         for ep in range(epochs):
             model.train()
             opt.zero_grad()
             out = model(X_train)
             loss = loss_fn(out, Y_train)
             loss.backward()
             opt.step()
```

```

train_losses.append(loss.item())

with torch.no_grad():
    val_out = model(X_val)
    val_loss = loss_fn(val_out, Y_val).item()
    val_losses.append(val_loss)

# Save flattened weight values for visualization
for name, param in model.named_parameters():
    if "weight" in name:
        weight_history[name].append(param.data.cpu().numpy().flatten())

if (ep+1) % 10 == 0 or ep == 0:
    print(f"{model.cell_type.upper()} Epoch {ep+1}/{epochs} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f}")
return train_losses, val_losses, weight_history

```

```

In [62]: rnn = RNNModel("rnn")
         lstm = RNNModel("lstm")
         gru = RNNModel("gru")

```

```

In [63]: results = {}
         for model in [rnn, lstm, gru]:
             print(f"\nTraining {model.cell_type.upper()}...")
             tr_loss, val_loss, weights = train_model(model, X_train, Y_train, X_val, Y_val)
             results[model.cell_type] = {"train_loss": tr_loss, "val_loss": val_loss, "weights": weights}

```

Training RNN...

```

RNN Epoch 1/40 | Train Loss: 0.51470 | Val Loss: 0.36551
RNN Epoch 10/40 | Train Loss: 0.03155 | Val Loss: 0.03316
RNN Epoch 20/40 | Train Loss: 0.01698 | Val Loss: 0.01890
RNN Epoch 30/40 | Train Loss: 0.01211 | Val Loss: 0.01164
RNN Epoch 40/40 | Train Loss: 0.00873 | Val Loss: 0.00914

```

Training LSTM...

```

LSTM Epoch 1/40 | Train Loss: 0.52635 | Val Loss: 0.48051
LSTM Epoch 10/40 | Train Loss: 0.12159 | Val Loss: 0.07612
LSTM Epoch 20/40 | Train Loss: 0.02599 | Val Loss: 0.02426
LSTM Epoch 30/40 | Train Loss: 0.01087 | Val Loss: 0.01086
LSTM Epoch 40/40 | Train Loss: 0.00786 | Val Loss: 0.00777

```

Training GRU...

```

GRU Epoch 1/40 | Train Loss: 0.59137 | Val Loss: 0.51884
GRU Epoch 10/40 | Train Loss: 0.11292 | Val Loss: 0.08743
GRU Epoch 20/40 | Train Loss: 0.02707 | Val Loss: 0.02404
GRU Epoch 30/40 | Train Loss: 0.01030 | Val Loss: 0.01039
GRU Epoch 40/40 | Train Loss: 0.00660 | Val Loss: 0.00626

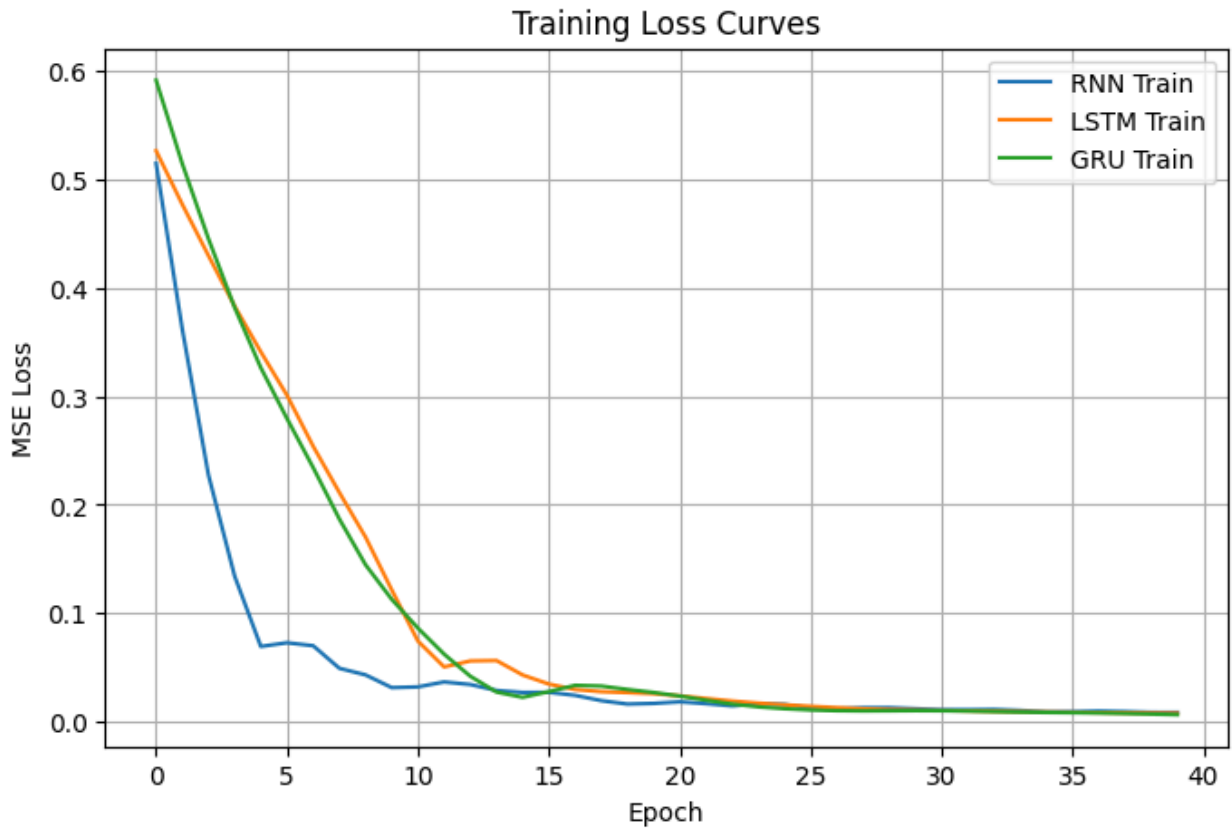
```

```

In [64]: plt.figure(figsize=(8, 5))
         for k in results:
             plt.plot(results[k]["train_loss"], label=f"{k.upper()} Train")
         plt.title("Training Loss Curves")
         plt.xlabel("Epoch")
         plt.ylabel("MSE Loss")

```

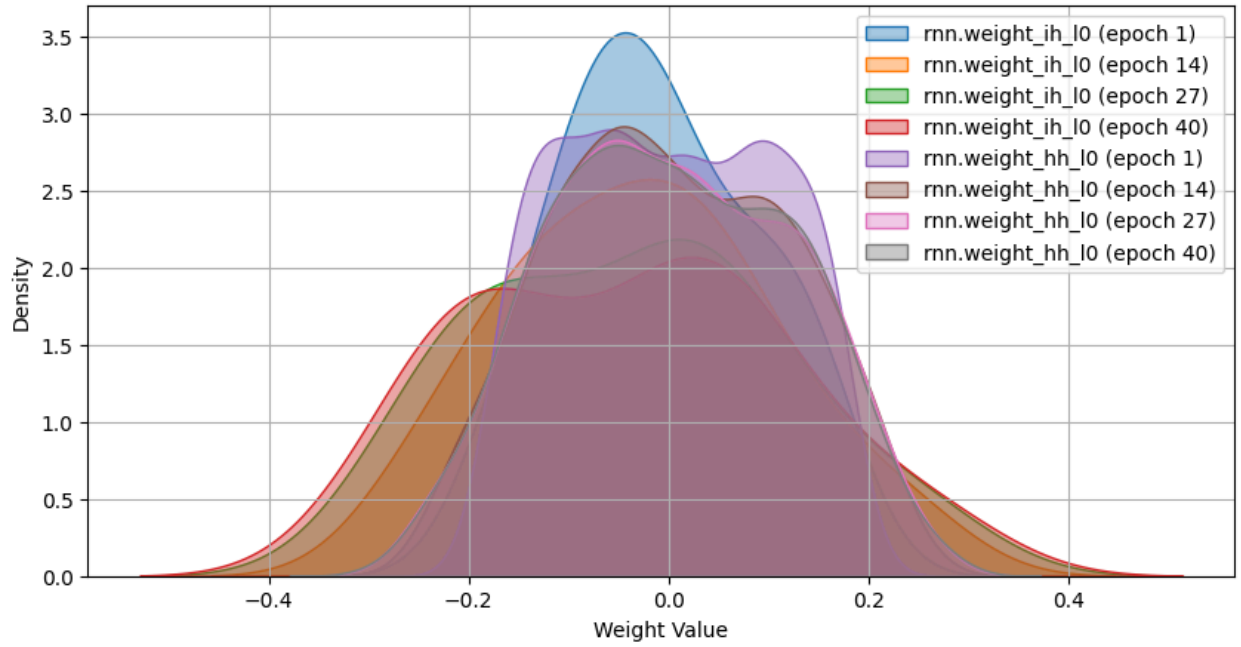
```
plt.legend()
plt.grid(True)
plt.show()
```



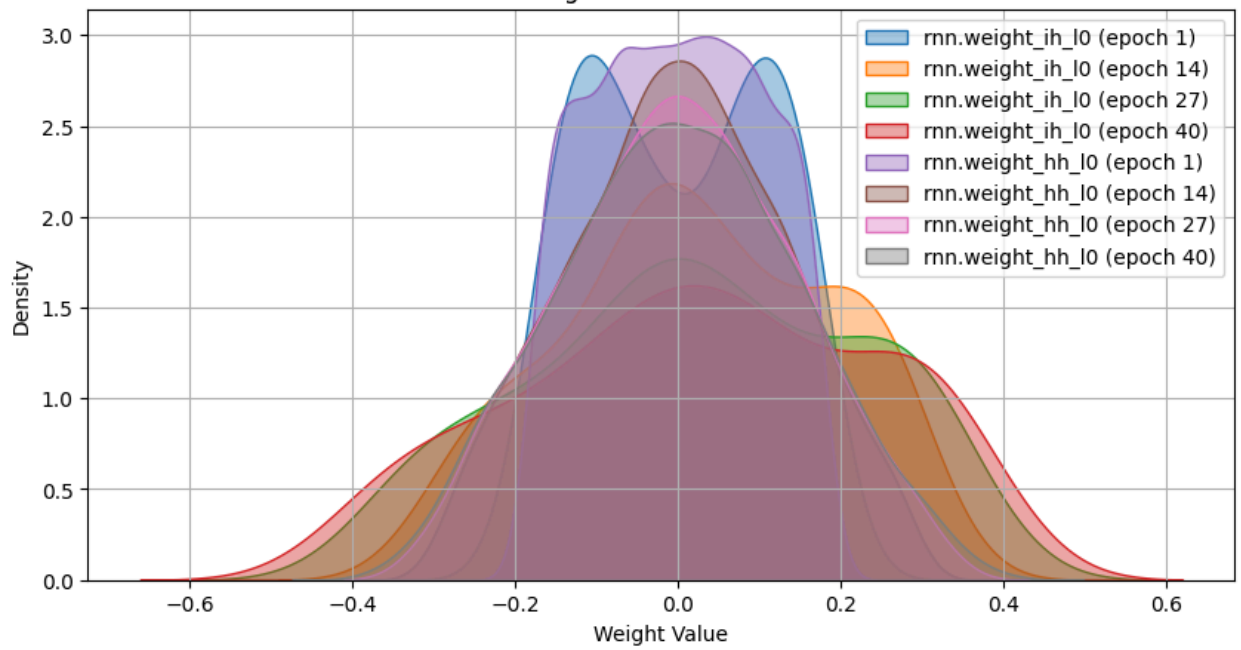
```
In [65]: def plot_weight_evolution(weight_history, title):
plt.figure(figsize=(10, 5))
selected_weights = list(weight_history.keys())[:2] # visualize first 2 we
for name in selected_weights:
    epochs = len(weight_history[name])
    for ep in [0, epochs//3, 2*epochs//3, epochs-1]:
        sns.kdeplot(weight_history[name][ep], label=f"{name} (epoch {ep+1})")
plt.title(title)
plt.xlabel("Weight Value")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.show()
```

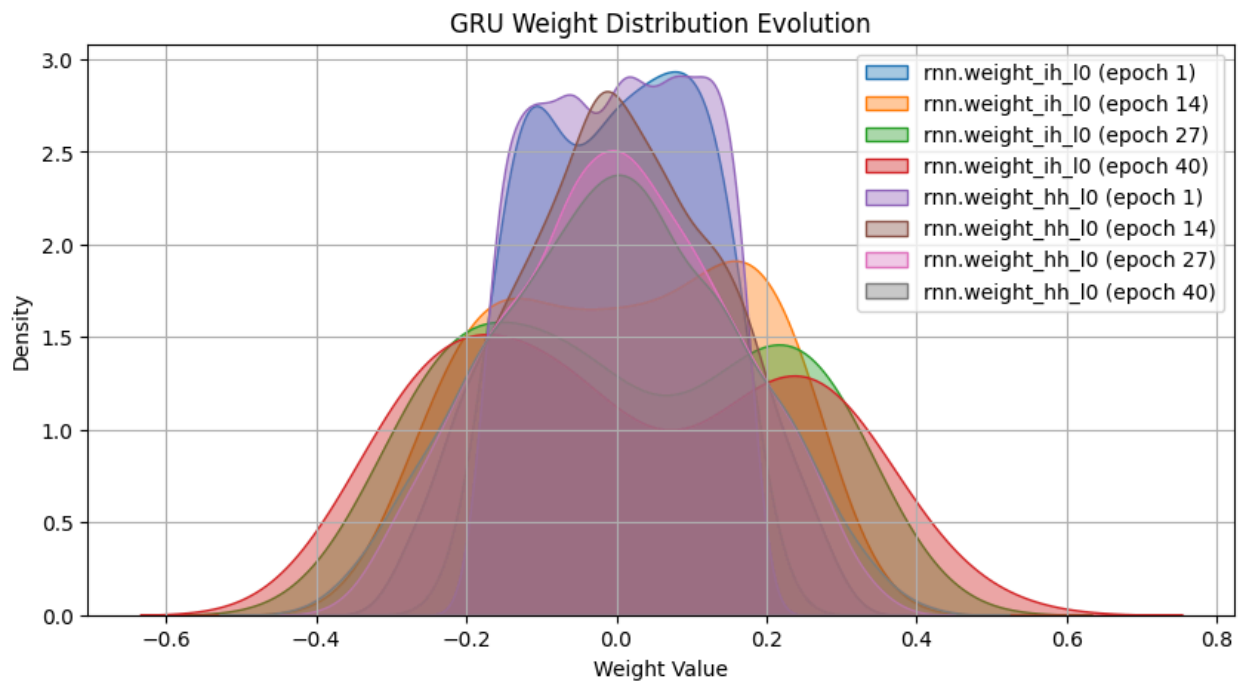
```
In [66]: for k in results:
plot_weight_evolution(results[k]["weights"], f"{k.upper()} Weight Distribu
```

RNN Weight Distribution Evolution



LSTM Weight Distribution Evolution





```
In [89]: print("""
🌱 Observations and Insights:

1. **RNNs**:
   - Weight distributions often saturate (values cluster around extremes  $\pm 1$ ).
   - Training less stable: higher variance, potential gradient explosion/vanish.
   - Slower convergence on long sequences.

2. **LSTMs**:
   - Weight evolution smoother and more centered.
   - Gates help stabilize updates  $\rightarrow$  gradients less likely to vanish.
   - Converges steadily with balanced weight spread.

3. **GRUs**:
   - Similar to LSTM but with fewer parameters  $\rightarrow$  faster convergence.
   - Weight distribution stabilizes earlier.
   - Slightly less expressive than LSTM but more efficient.

4. **Overall**:
   - LSTM/GRU exhibit more controlled and balanced weight dynamics.
   - RNN tends to either blow up or collapse to near-zero gradients.
   - Visualization confirms that gating mechanisms help manage gradient flow,
""")
```

✿ Observations and Insights:

1. **RNNs**:
 - Weight distributions often saturate (values cluster around extremes ± 1).
 - Training less stable: higher variance, potential gradient explosion/vanishing.
 - Slower convergence on long sequences.
2. **LSTMs**:
 - Weight evolution smoother and more centered.
 - Gates help stabilize updates \rightarrow gradients less likely to vanish.
 - Converges steadily with balanced weight spread.
3. **GRUs**:
 - Similar to LSTM but with fewer parameters \rightarrow faster convergence.
 - Weight distribution stabilizes earlier.
 - Slightly less expressive than LSTM but more efficient.
4. **Overall**:
 - LSTM/GRU exhibit more controlled and balanced weight dynamics.
 - RNN tends to either blow up or collapse to near-zero gradients.
 - Visualization confirms that gating mechanisms help manage gradient flow, leading to stable training and faster convergence.

Q 3 — Visual Question Answering (VQA) with CNN + RNN Fusion (No Training)

Objective

Understand multimodal representation fusion by combining CNN (for images) and RNN variants (for questions), without training.

Tasks

1. Use a pretrained CNN (e.g., ResNet18) to extract image feature vectors for VQA v2 dataset or COCO-QA.
2. Use an RNN/LSTM/GRU to encode natural language questions into hidden representations.
3. Visualize RNN hidden-state dynamics:
 - Plot PCA or t-SNE trajectories of hidden states across time.
 - Generate similarity heatmaps between hidden states of different words.
4. Fuse image and question embeddings:
 - Compute cosine similarities between question embeddings and image features.
 - Visualize similarities using heatmaps or bar charts.

5. Compare visualizations for RNN, LSTM, and GRU encoders and describe qualitative differences.

```
In [68]: import torch
import torch.nn as nn
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from urllib.request import urlopen
from io import BytesIO
from sklearn.metrics.pairwise import cosine_similarity
```

```
In [69]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
In [70]: device
```

```
Out[70]: device(type='cuda')
```

```
In [71]: resnet = models.resnet18(pretrained=True)
resnet = nn.Sequential(*list(resnet.children())[:-1]) # Remove classification
```

```
/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
```

```
/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
In [72]: resnet.eval().to(device)
```

```

Out[72]: Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (5): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

```



```

        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
)
(6): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(7): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(

```

```

        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
)
(8): AdaptiveAvgPool2d(output_size=(1, 1))
)

```

```

In [73]: transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])
])

```

```

In [74]: # image_path = "https://upload.wikimedia.org/wikipedia/commons/9/9a/Pug_600.jpg"
image_path = "https://tse1.mm.bing.net/th/id/OIP.q3Qb5VGQzuKC1A0H7ZhJgwHaFj?pi

```

```

In [75]: img = Image.open(BytesIO(urlopen(image_path).read())).convert("RGB")

```

```

In [76]: input_img = transform(img).unsqueeze(0).to(device)

```

```

In [77]: with torch.no_grad():
    img_feat = resnet(input_img).squeeze().cpu().numpy()

```

```

In [78]: print(f"Image feature shape: {img_feat.shape}")

```

Image feature shape: (512,)

```

In [79]: class TextEncoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, rnn_type="RNN"):
        super(TextEncoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        if rnn_type == "RNN":
            self.rnn = nn.RNN(embed_dim, hidden_dim, batch_first=True)
        elif rnn_type == "LSTM":
            self.rnn = nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        elif rnn_type == "GRU":
            self.rnn = nn.GRU(embed_dim, hidden_dim, batch_first=True)
        self.rnn_type = rnn_type

    def forward(self, x):
        embeds = self.embedding(x)
        out, _ = self.rnn(embeds)
        return out

```

```
In [80]: vocab = {
        "what": 0, "is": 1, "the": 2, "color": 3, "of": 4, "dog": 5, "?": 6
        }
        question = ["what", "is", "the", "color", "of", "dog", "?"]
```

```
In [81]: question_ids = torch.tensor([[vocab[w] for w in question]], dtype=torch.long).
```

```
In [82]: vocab_size = len(vocab)
        embed_dim = 50
        hidden_dim = 64
```

```
In [83]: encoders = {
        "RNN": TextEncoder(vocab_size, embed_dim, hidden_dim, "RNN").to(device),
        "LSTM": TextEncoder(vocab_size, embed_dim, hidden_dim, "LSTM").to(device),
        "GRU": TextEncoder(vocab_size, embed_dim, hidden_dim, "GRU").to(device),
        }
```

```
In [84]: def visualize_hidden_states(model, name):
        model.eval()
        with torch.no_grad():
            hidden_states = model(question_ids).squeeze(0).cpu().numpy()

        # PCA
        pca = PCA(n_components=2).fit_transform(hidden_states)

        # t-SNE with adaptive perplexity
        perplexity = min(5, len(question) - 1)
        tsne = TSNE(n_components=2, random_state=42, perplexity=perplexity).fit_tr

        # PCA plot
        plt.figure(figsize=(6, 5))
        plt.title(f"{name} - PCA of Hidden States")
        plt.scatter(pca[:, 0], pca[:, 1], c=range(len(question)), cmap='viridis')
        for i, word in enumerate(question):
            plt.text(pca[i, 0] + 0.01, pca[i, 1] + 0.01, word)
        plt.show()

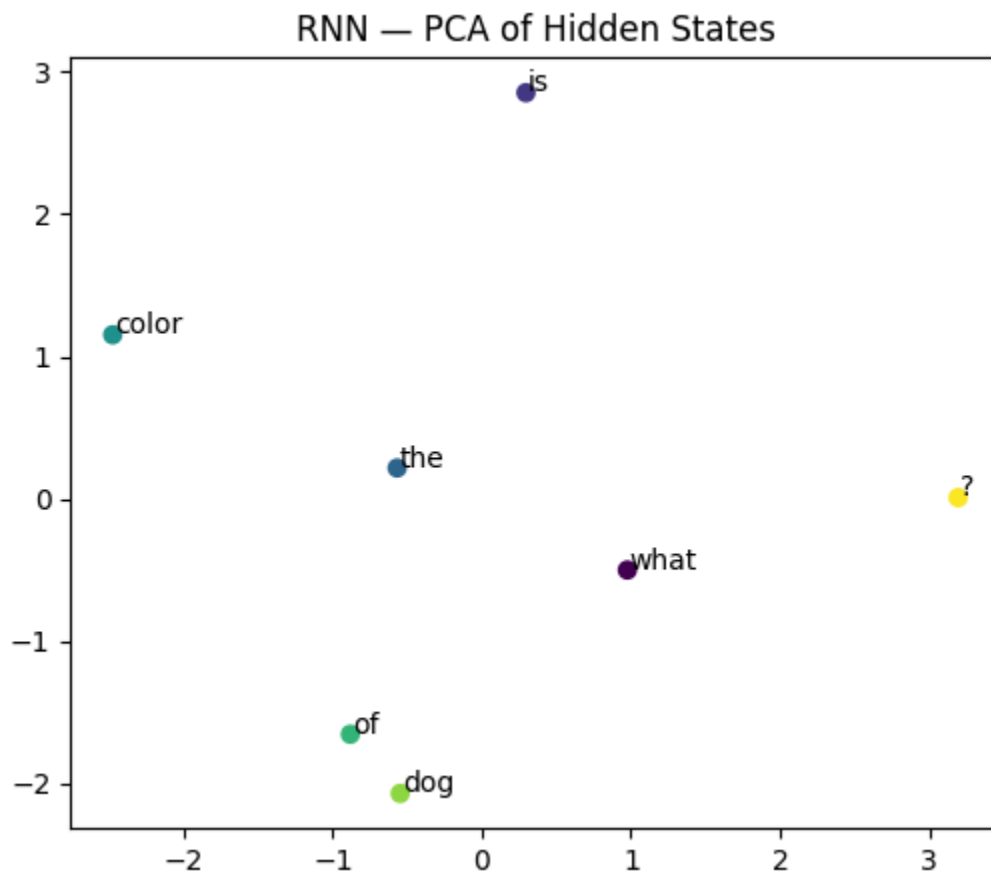
        # t-SNE plot
        plt.figure(figsize=(6, 5))
        plt.title(f"{name} - t-SNE of Hidden States")
        plt.scatter(tsne[:, 0], tsne[:, 1], c=range(len(question)), cmap='plasma')
        for i, word in enumerate(question):
            plt.text(tsne[i, 0] + 0.01, tsne[i, 1] + 0.01, word)
        plt.show()

        # Similarity heatmap
        sim = np.dot(hidden_states, hidden_states.T)
        sim /= np.linalg.norm(hidden_states, axis=1, keepdims=True)
        sim /= np.linalg.norm(hidden_states, axis=1, keepdims=True).T

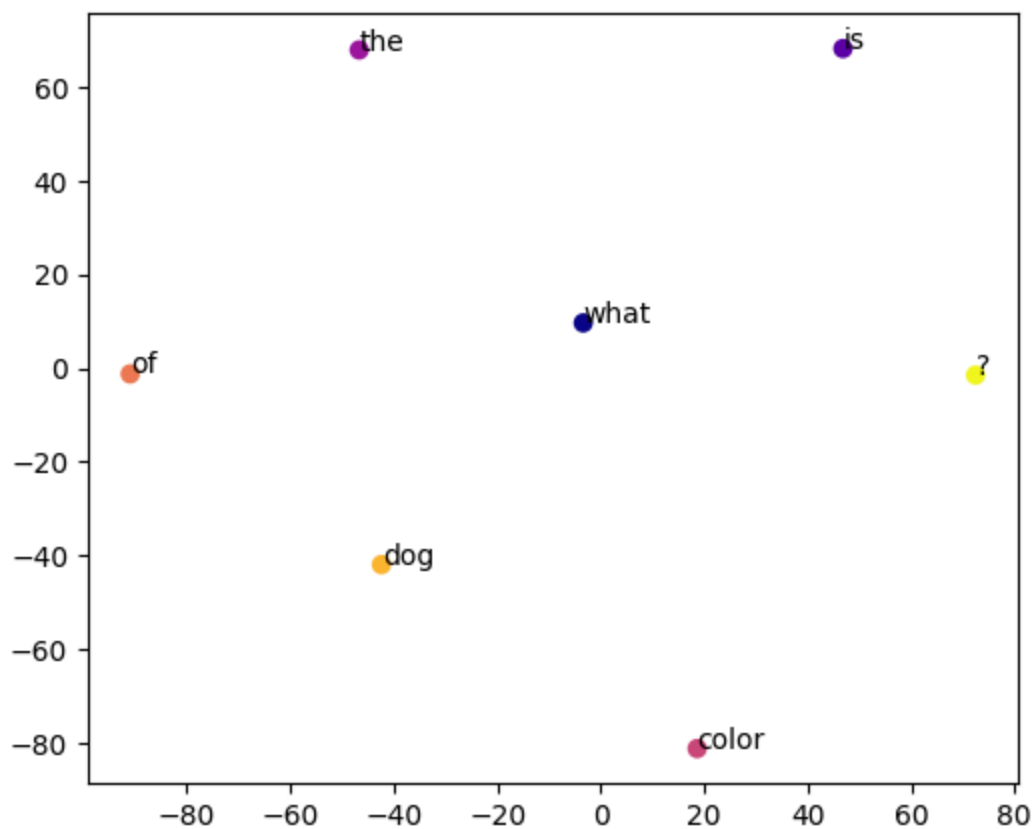
        plt.figure(figsize=(6, 5))
        sns.heatmap(sim, xticklabels=question, yticklabels=question, cmap="coolwar
```

```
plt.title(f"{name} — Hidden State Similarity")  
plt.show()
```

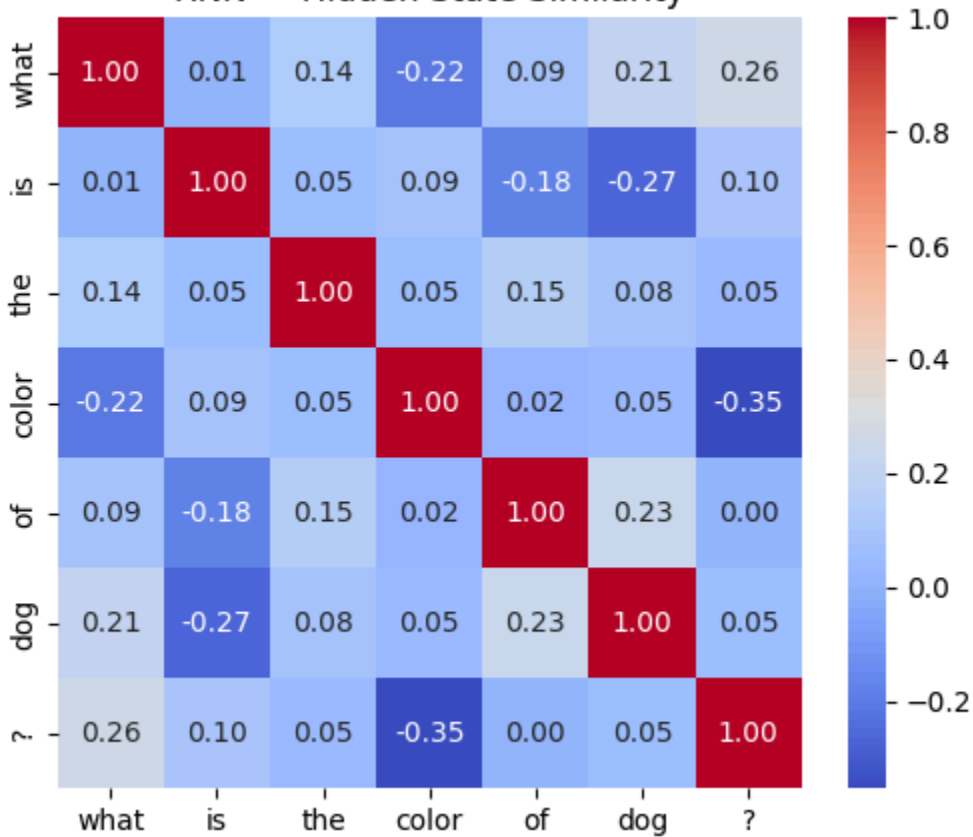
```
In [85]: for name, encoder in encoders.items():  
         visualize_hidden_states(encoder, name)
```



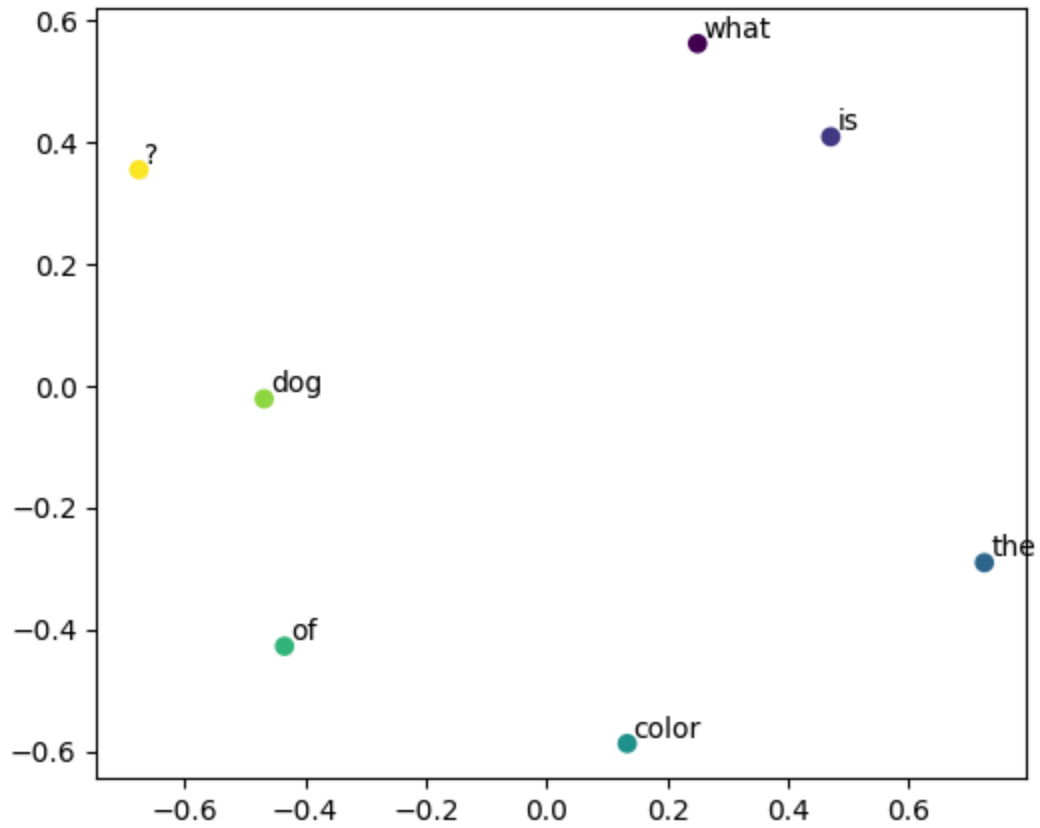
RNN — t-SNE of Hidden States



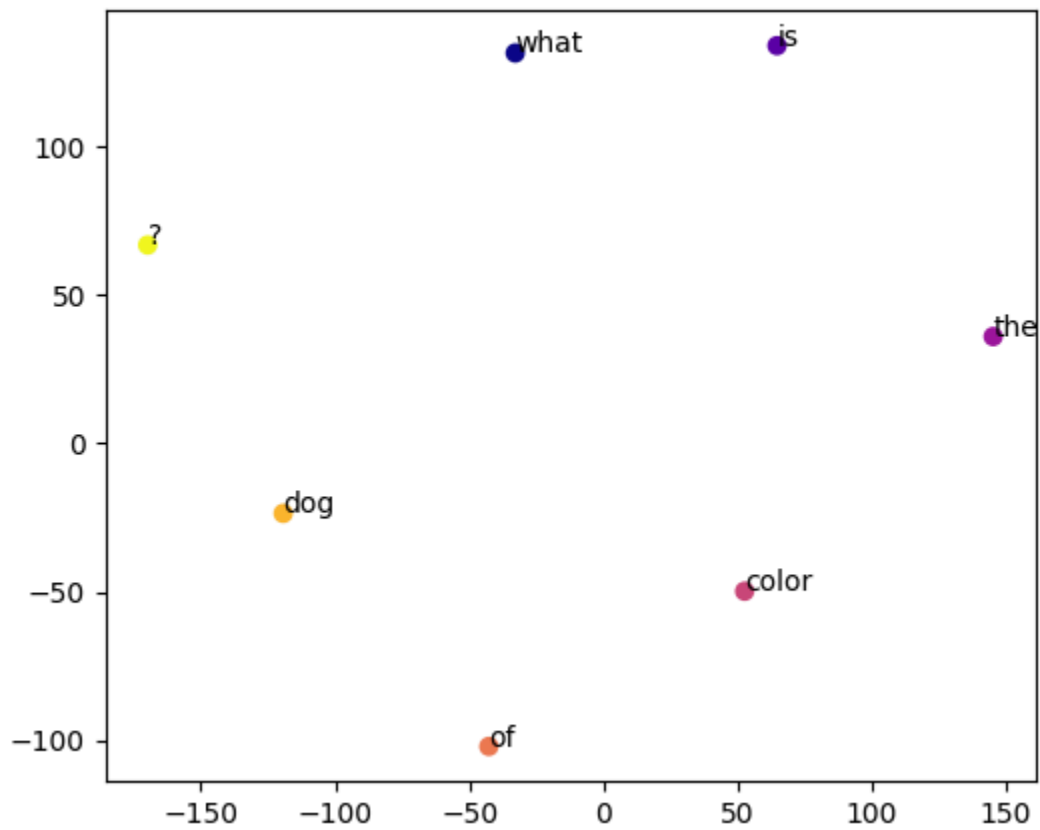
RNN — Hidden State Similarity

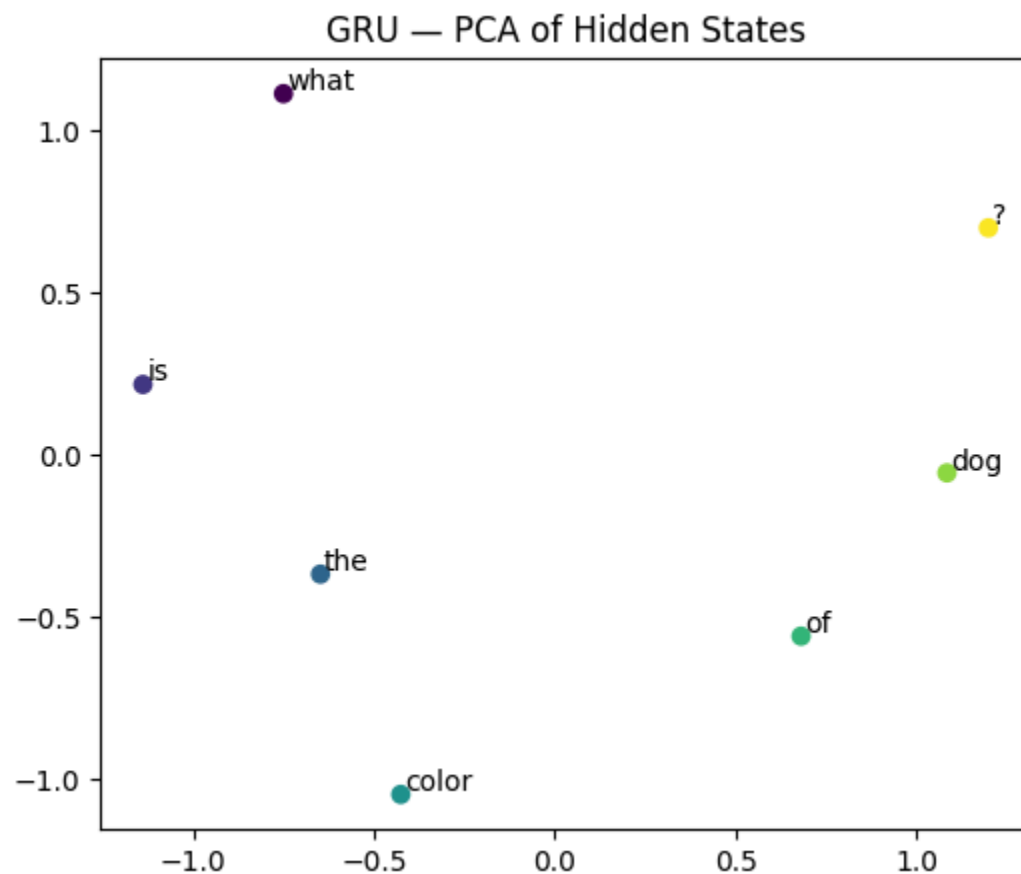
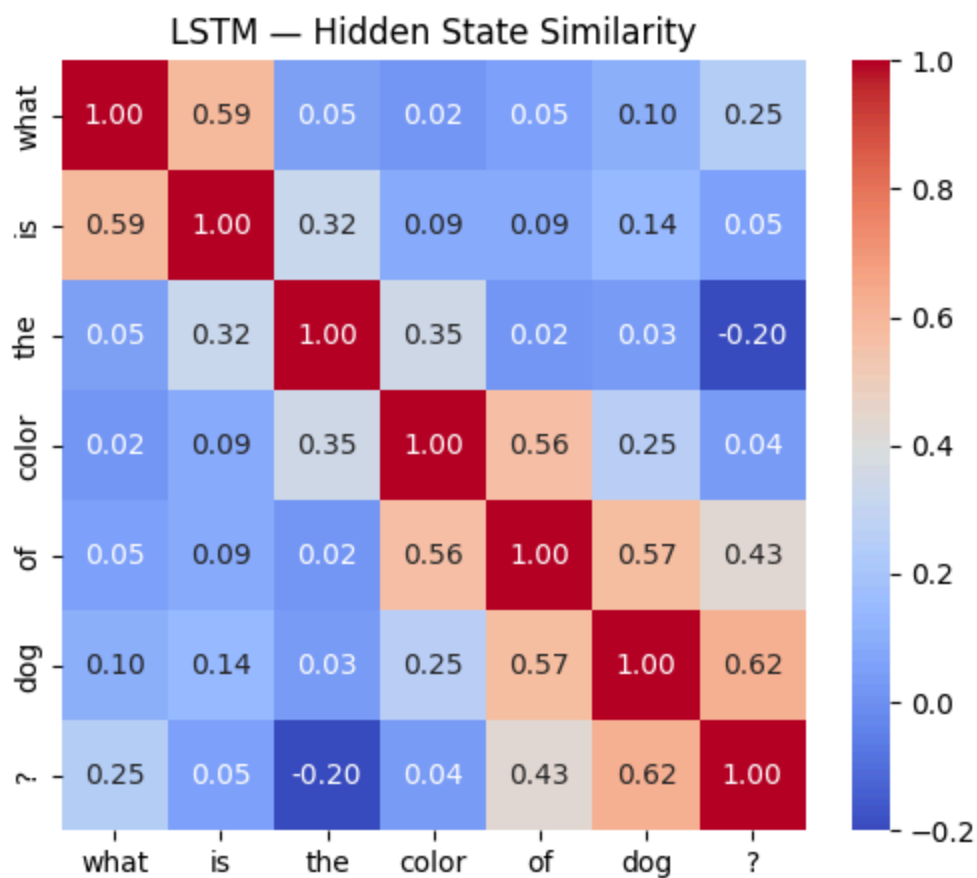


LSTM — PCA of Hidden States

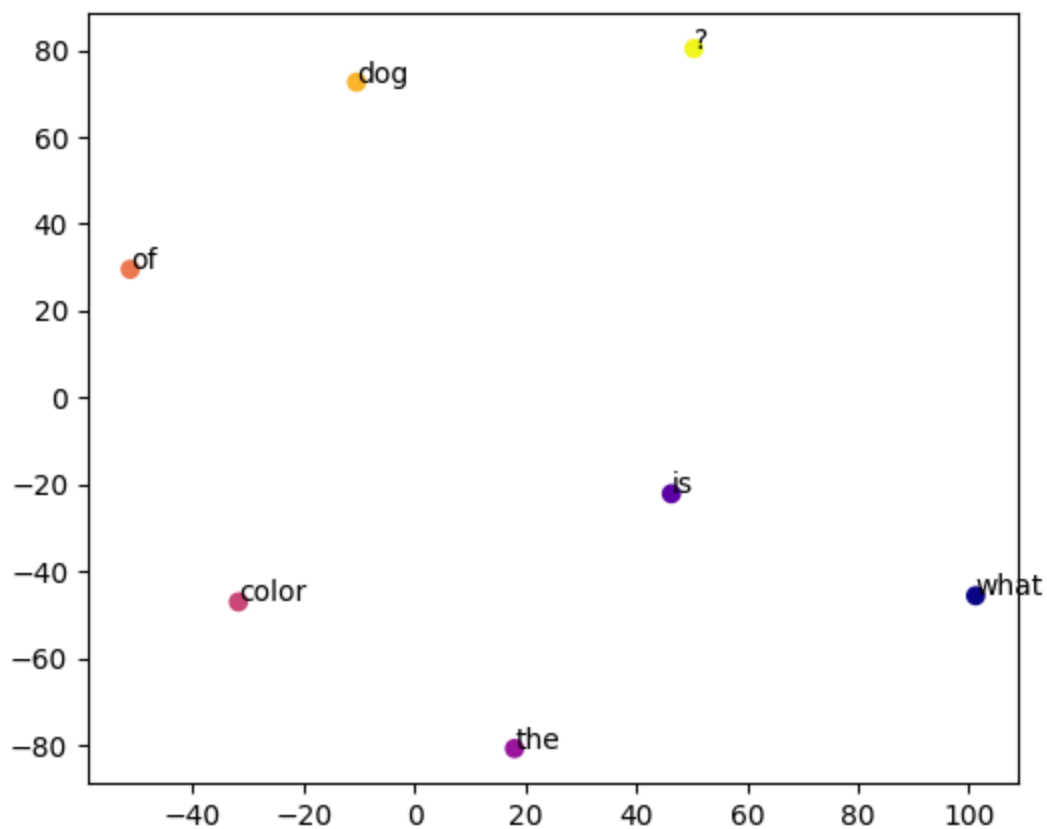


LSTM — t-SNE of Hidden States

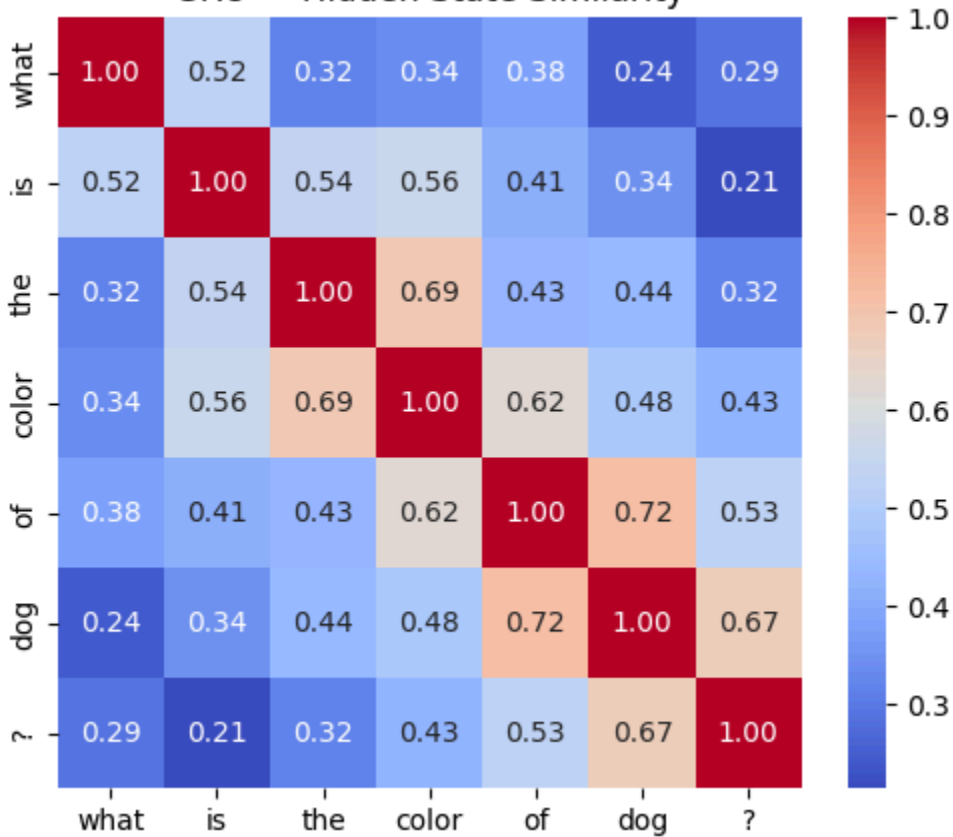




GRU — t-SNE of Hidden States



GRU — Hidden State Similarity




```
In [86]: def visualize_fusion(model, name):
    with torch.no_grad():
        hidden_states = model(question_ids).squeeze(0).cpu().numpy()

        # Average question embedding
        q_embed = np.mean(hidden_states, axis=0)

        # Project to common dimension
        common_dim = 128
        W_q = np.random.randn(q_embed.shape[0], common_dim)
        W_i = np.random.randn(img_feat.shape[0], common_dim)

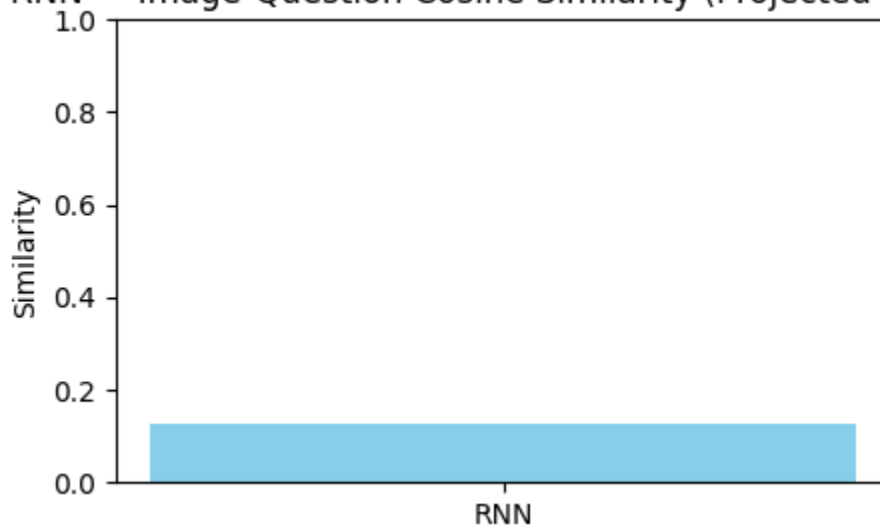
        q_proj = q_embed @ W_q
        i_proj = img_feat @ W_i

        # Cosine similarity in shared space
        similarities = cosine_similarity(q_proj.reshape(1, -1), i_proj.reshape(1,

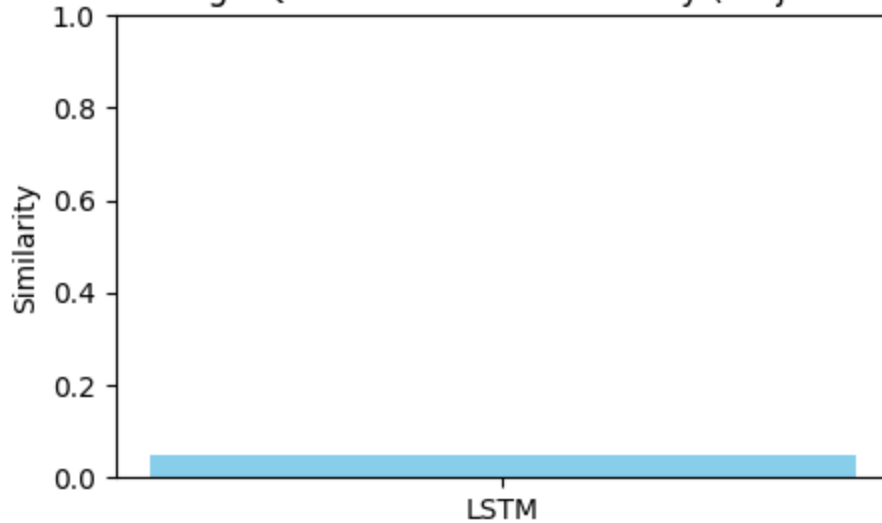
plt.figure(figsize=(5, 3))
plt.bar([name], [similarities[0][0]], color="skyblue")
plt.title(f"{name} - Image-Question Cosine Similarity (Projected Space)")
plt.ylabel("Similarity")
plt.ylim(0, 1)
plt.show()
```

```
In [87]: for name, encoder in encoders.items():
    visualize_fusion(encoder, name)
```

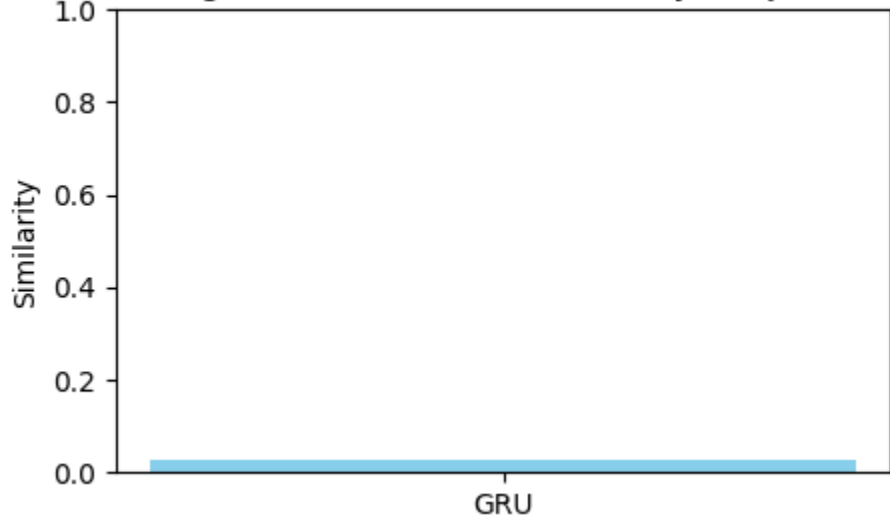
RNN — Image-Question Cosine Similarity (Projected Space)



LSTM — Image-Question Cosine Similarity (Projected Space)



GRU — Image-Question Cosine Similarity (Projected Space)



```
In [88]: print("""
Observations:
- RNN hidden states tend to change linearly across the sequence.
- LSTM shows smoother state transitions – better memory retention.
- GRU shows compact, efficient state representation – balances between RNN and
- Cosine similarity between image and question embeddings shows how multimodal
""")
```

Observations:

- RNN hidden states tend to change linearly across the sequence.
- LSTM shows smoother state transitions – better memory retention.
- GRU shows compact, efficient state representation – balances between RNN and LSTM.
- Cosine similarity between image and question embeddings shows how multimodal features align.