HW3_1_LanguageModelArchitectures

February 15, 2025

This can be run on Google Colab using this link # Language model architectures, from RNNs to Transformers

In this notebook, you will get experience implementing the core logic within a few language model architectures, from RNNs to transformers.

0.1 Preliminaries

Set up here.

[12]:
```python
# First, we'll implement a simple RNN language model

import math
import random
from dataclasses import dataclass

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from datasets import load_dataset
from ipywidgets import FloatSlider, interactive
from torch.nn import functional as F
from torch.utils.data import DataLoader, Dataset
import time
from tqdm import tqdm

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {DEVICE}")

SEED = 2025
torch.manual_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)
```

```
Using device: cpu
```

0.2 Dataset

We will create a **character-based** autoregressive language model, which means, it will predict one character at a time.

To train it, we will need a dataset class that provides character sequences, then we will load some text into it.

We use the TinyStories dataset. (Eldan, 2023)

[13]:
```python
# This class takes a string of text and a sequence length and
# breaks it down to character sequences of the given length.
class CharLMDataset(Dataset):
    def __init__(self, text, seq_length=20):
        self.seq_length = seq_length
        self.vocab = sorted(set(text))
        self.stoi = {ch: i for i, ch in enumerate(self.vocab)}
        self.itos = {i: ch for i, ch in enumerate(self.vocab)}
        self.encoded_text = [self.stoi[ch] for ch in text]
        # Ensure we have enough tokens for both input and target sequences
        self.encoded_text = self.encoded_text[
            : len(self.encoded_text) - (len(self.encoded_text) % (seq_length +␣
 ↪1))
        ]

    def __len__(self):
        # Number of complete chunks
        return len(self.encoded_text) // self.seq_length

    def __getitem__(self, idx):
        # Get the idx-th chunk
        chunk = self.encoded_text[
            idx * self.seq_length : (idx + 1) * self.seq_length + 1
        ]
        x = torch.tensor(chunk[:-1], dtype=torch.long)
        y = torch.tensor(chunk[1:], dtype=torch.long)
        return x, y


# TinyStories (Eldan, 2023)[https://arxiv.org/abs/2305.07759] is a dataset of␣
 ↪short, simple stories generated
# by GPT3.5 and GPT4. There about 2B characters in the full dataset, but we'll␣
 ↪only use 10M
dataset_size = 10_000_000
batch_size = 32

tinystories = load_dataset("roneneldan/TinyStories")
full_text = "\n".join(tinystories["train"]["text"])[:dataset_size]
print(full_text[:1000])
```

```
dataset = CharLMDataset(full_text)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
  ↪drop_last=True)
```

```
One day, a little girl named Lily found a needle in her room. She knew it was
difficult to play with it because it was sharp. Lily wanted to share the needle
with her mom, so she could sew a button on her shirt.

Lily went to her mom and said, "Mom, I found this needle. Can you share it with
me and sew my shirt?" Her mom smiled and said, "Yes, Lily, we can share the
needle and fix your shirt."

Together, they shared the needle and sewed the button on Lily's shirt. It was
not difficult for them because they were sharing and helping each other. After
they finished, Lily thanked her mom for sharing the needle and fixing her shirt.
They both felt happy because they had shared and worked together.
Once upon a time, there was a little car named Beep. Beep loved to go fast and
play in the sun. Beep was a healthy car because he always had good fuel. Good
fuel made Beep happy and strong.

One day, Beep was driving in the park when he saw a big tree. The tree had many
leaves that were falling. B
```

1 Recurrent Neural Networks

1.1 Autoregressive Language Modeling

An autoregressive language model predicts the probability distribution of the next token given the previous tokens:

$$P(x_t|x_{t-1}, x_{t-2}, ..., x_1) = g_\theta(x_{t-1}, x_{t-2}, ..., x_1)$$

For the complete sequence, the joint probability factorizes as:

$$P(x_1, ..., x_T) = \prod_{t=1}^{T} P(x_t|x_{t-1}, ..., x_1)$$

Now, we need the neural network to plays the role of the abstract function g_θ, which means that it must take a sequence of tokens as input, and produce a categorical distribution over possible next-tokens as output. But every neural network has a fixed set of neurons. How can we make one that takes an arbitrarily long sequence of inputs?

The idea of a **Recurrent Neural Network** is to implement the abstract g_θ as a Recurrent neural network f_θ, that recursively consumes one token at a time. The idea is to use a hidden state vector h_t that summarizes the sequence history:

$$g_\theta(x_{1:t}) = \text{softmax}(W_{out} f_\theta(x_t, f_\theta(x_{t-1}, f_\theta(x_{t-2}, ... f_\theta(x_1, h_0)...))) + b_{out})$$

Each invocation of f_θ is now simple, consuming a single token and a single hidden state vector.

$ h_t = f_ (x_t, h_{t-1}) $

1.2 Token Encoding

Some details: to get tokens into a neural network, we need to encode them as vectors.

Each input token x_t is first mapped to a dense vector through an embedding layer:

$$e_t = E x_t$$

where $E \in \mathbb{R}^{d_{emb} \times |V|}$ is the embedding matrix and $|V|$ is the vocabulary size.

The model's hidden state h_t captures the sequence context.

1.3 Token Decoding

Then to get tokens out, we need to produce a distribution of output probabilities for the next token.

To predict the next token, we use a linear classifier that makes a prediction based on the hidden state:

$$logits = W_{fc} h_t + b_{fc}$$

$$P(x_{t+1} | x_t, ..., x_1) = softmax(logits)$$

where $W_{out} \in \mathbb{R}^{|V| \times d_{hidden}}$ projects the hidden state back to vocabulary-sized logits.

1.4 Elman RNN Architecture

How do we get one hidden state from another?

The vanilla or Elman RNN arcthiecture is given by this rule:

$$h_t = \tanh(W_{ih} x_t + b_{ih} + W_{hh} h_{t-1} + b_{hh})$$

where: - x_t is the input embedding at time t - h_t is the hidden state at time t - $W_{ih} \in \mathbb{R}^{d_{hidden} \times d_{emb}}$ is the input-to-hidden weight matrix - $W_{hh} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$ is the hidden-to-hidden weight matrix - $b_{ih}, b_{hh} \in \mathbb{R}^{d_{hidden}}$ are the corresponding bias vectors

The hidden state h_t serves as a memory that captures relevant information from previous tokens. The nonlinear tanh activation allows the network to learn complex temporal dependencies.

During generation, we: 1. Initialize h_0 to zeros 2. Sample each token from the predicted probability distribution 3. Update the hidden state using the sampled token 4. Repeat until reaching desired length

1.5 SimpleRNN

In the implementation of an Elman below, each invocation produces both h and $logits$.

Exercise 1 Implement the code that implements the recurrent generation.

[14]:
```
class SimpleRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(vocab_size, embed_size)

        # fill in the proper neural network dimensions
        self.W_ih = nn.Linear(embed_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)
        self.fc = nn.Linear(hidden_size, vocab_size)
        self.tanh = nn.Tanh()
        self.to(DEVICE)

    def forward(self, x_token, hidden=None):
        x_emb = self.embedding(x_token)
        # look at the Elman recurrence formula above and make the code match it.
        hidden = self.tanh(self.W_ih(x_emb) + self.W_hh(hidden))
        logits = self.fc(hidden)
        return logits, hidden

    def init_hidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size, device=DEVICE)

    def generate_text(self, dataset, length=100):
        self.eval()
        # Start with a random token
        token = torch.randint(0, len(dataset.vocab), (1,), device=DEVICE)
        generated_tokens = [token.item()]
        hidden = self.init_hidden(1)

        # Generate one token at a time
        for _ in range(length - 1):
            # Get predictions

            # YOUR CODE HERE
            #
            # 1. Call the neural network by adding the new token and re-feeding␣
 ↪the hidden state
            logits, hidden = self(token, hidden)
            # 2. Ensure the logits are converted to probability using softmax
            probs = nn.functional.softmax(logits, dim=-1).squeeze(0)
            # 3. Choose the next token according to the model's predicted␣
↪categorical probability distribution
            # HINT: torch.multinomial could be handy
            token = torch.multinomial(probs, num_samples=1)

            generated_tokens.append(token.item())

        # Convert tokens back to text using the dataset's vocabulary
        generated_text = "".join([dataset.itos[i] for i in generated_tokens])
        return generated_text
```

2 Backpropagation Through Time

2.1 The Chain Rule for RNNs

For a sequence of length T, the loss L depends on the model predictions at each timestep:

$$L = \frac{1}{T}\sum_{t=1}^{T} L_t(y_t, \hat{y}_t)$$

where $\hat{y}_t = \text{softmax}(W_{out}h_t + b_{out})$ and $h_t = f_\theta(x_t, h_{t-1})$.

The gradient with respect to parameters θ requires the chain rule through time:

$$\frac{\partial L}{\partial \theta} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial L_t}{\partial \theta}$$

For each timestep t, we must consider how θ affects the current prediction through all previous timesteps:

$$\frac{\partial L_t}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial \theta}$$

2.2 Implementation

The training code implements BPTT by: 1. Initializing hidden state h_0 for each sequence 2. Unrolling the RNN for T steps, accumulating loss at each step 3. Computing average loss over sequence length 4. Using autograd to compute gradients through the entire computational graph 5. Updating parameters with Adam optimizer

For each timestep t, the core computation is:

1. Forward pass through RNN: $\hat{y}_t = W_{out}h_t + b_{out}, \quad h_t = f_\theta(x_t, h_{t-1})$ Remember that `model(x_batch[:, t], h_prev)` will calculate both of these.
2. Compute cross-entropy loss: $L_t = -\sum_k y_{t,k} \log(\text{softmax}(\hat{y}_t)_k)$. In the code below, `criterion(logits, y_batch[:, t])` will calculate this.

3. Accumulate loss over sequence: $L = \frac{1}{T}\sum_{t=1}^{T} L_t$. In the code below, remember to add up the loss for every token; we have included the `loss / seq_len` for you at the end.

The hidden state carries gradients backward through time, allowing the model to learn long-term dependencies.

(However, this can lead to vanishing/exploding gradients over long sequences.)

Exercise 2: Implement the missing code in the RNN training code below.

```
[15]: def train_language_model(
          model, dataloader, lr=1e-3, print_every=100, max_iters=None, is_rnn=True
      ):

          optimizer = optim.Adam(model.parameters(), lr=lr)

          criterion = nn.CrossEntropyLoss()
          model.train()

          losses = []

          start_time = time.time()

          for step, (x_batch, y_batch) in enumerate(dataloader):
              if max_iters is not None and step >= max_iters:
                  break

              x_batch, y_batch = x_batch.to(DEVICE), y_batch.to(DEVICE)

              if is_rnn:
                  # RNN/GRU: process sequence step by step
                  h = model.init_hidden(x_batch.size(0))
                  loss = 0

                  # Unroll over seq_len
                  for t in range(x_batch.size(1)):
                      # YOUR CODE HERE
                      # 1. compute the logits for the next token and the next hidden␣
        ↪state
                      logits, h = model(x_batch[:, t], h)

                      # 2. compute the loss for the probability distribution for the␣
        ↪next token.
                      loss_t = criterion(logits, y_batch[:, t])

                      # 3. remember to accumulate the loss over all tokens.
                      loss += loss_t

                      # YOUR CODE HERE
```

```
            loss = loss / x_batch.size(1)
        else:

            # Transformer: process whole sequence at once

            logits, _ = model(x_batch)

            loss = criterion(logits.view(-1, logits.size(-1)), y_batch.view(-1))

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

        losses.append(loss.item())

        if (step + 1) % print_every == 0:

            elapsed = time.time() - start_time

            print(f"Step {step+1}, Loss: {loss.item():.4f}, Time: {elapsed:.
 ↪2f}s")
            start_time = time.time()

    return losses


def plot_losses(loss_dict):
    for description, losses in loss_dict.items():
        plt.plot(losses, label=description, alpha=0.7)

    plt.legend()
    plt.xlabel("Iteration")
    plt.ylabel("Loss")
    plt.title("Training Losses")
    plt.show()
```

2.3 Training loop below.

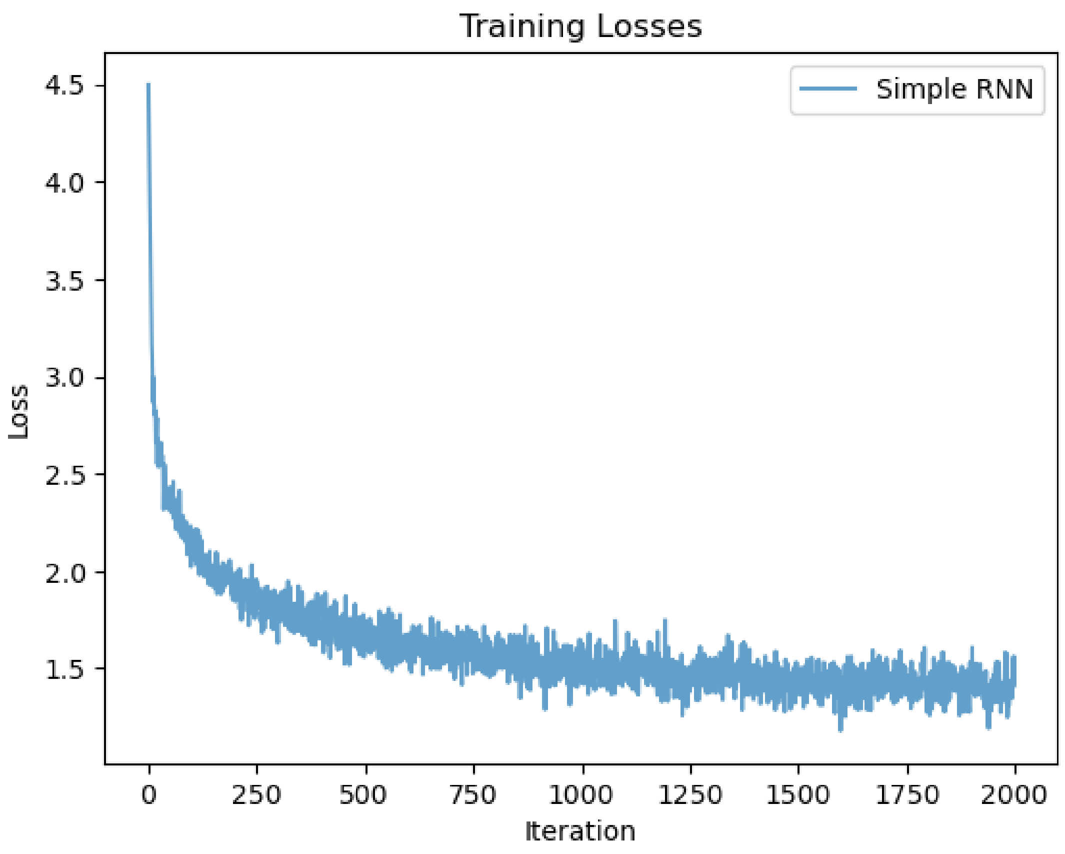
Experiment with the training loop below. Your model should have no trouble getting to a cross-entropy loss of about 1.3 or 1.4.

Does the training produce a model with reasonable results? If not, go back and debug!

[16]:
```
hidden_size = 256
embed_size = 64
simple_rnn = SimpleRNN(len(dataset.vocab), embed_size, hidden_size).to(DEVICE)
```

```
simple_rnn_losses = train_language_model(simple_rnn, dataloader, max_iters=2000)
plot_losses(
    {
        "Simple RNN": simple_rnn_losses,
    }
)
# Generate some sample text from the RNN. After maybe 2000 iterations, you␣
 ↪should be able to get a loss of 1.4, and see
# something that's recognizably English, but not perfect, with frequent␣
 ↪misspellings.
sample_text = simple_rnn.generate_text(dataset)
print(f"\nSample text from Simple RNN: {sample_text}")
print("-" * 80)
```

```
Step 100, Loss: 2.0229, Time: 4.13s
Step 200, Loss: 1.9412, Time: 4.22s
Step 300, Loss: 1.6307, Time: 4.22s
Step 400, Loss: 1.6567, Time: 4.41s
Step 500, Loss: 1.6818, Time: 4.15s
Step 600, Loss: 1.5694, Time: 4.20s
Step 700, Loss: 1.5973, Time: 4.36s
Step 800, Loss: 1.4731, Time: 4.20s
Step 900, Loss: 1.4911, Time: 4.39s
Step 1000, Loss: 1.5349, Time: 4.42s
Step 1100, Loss: 1.5041, Time: 4.52s
Step 1200, Loss: 1.3312, Time: 4.52s
Step 1300, Loss: 1.4290, Time: 4.33s
Step 1400, Loss: 1.4557, Time: 4.51s
Step 1500, Loss: 1.4019, Time: 4.44s
Step 1600, Loss: 1.3481, Time: 4.59s
Step 1700, Loss: 1.4320, Time: 4.48s
Step 1800, Loss: 1.4118, Time: 4.37s
Step 1900, Loss: 1.3799, Time: 4.43s
Step 2000, Loss: 1.4121, Time: 4.90s
```



```
Sample text from Simple RNN: R³€ry?" Tim said, "Dunk, kniase water bept know!"
Anna on her." Saras, bloaghtil down. She tore inc
--------------------------------------------------------------------------------
```

3 Extra Credit: Multi Layer RNN Architecture

3.1 Basic Idea

In a multi-layer RNN, each layer processes the output of the previous layer. For layer l, the hidden state update is:

$$h_t^{(l)} = f_\theta^{(l)}(h_t^{(l-1)}, h_{t-1}^{(l)})$$

where $h_t^{(l)}$ is the hidden state at time t for layer l.

3.2 Layer-wise Processing

The computation flows as follows:

1. Input embedding (layer 0):
$$h_t^{(0)} = Ex_t$$

2. First RNN layer:
$$h_t^{(1)} = \tanh(W_{ih}^{(1)} h_t^{(0)} + b_{ih}^{(1)} + W_{hh}^{(1)} h_{t-1}^{(1)} + b_{hh}^{(1)})$$

3. Subsequent layers (l > 1):
$$h_t^{(l)} = \tanh(W_{ih}^{(l)} h_t^{(l-1)} + b_{ih}^{(l)} + W_{hh}^{(l)} h_{t-1}^{(l)} + b_{hh}^{(l)})$$

4. Final output prediction:
$$\hat{y}_t = \mathrm{softmax}(W_{out} h_t^{(L)} + b_{out})$$

where L is the total number of layers.

3.3 Information Flow

The network now has two types of connections: - Temporal connections (within each layer across time steps) - Deep connections (between layers at each time step)

This allows the network to learn both: - Temporal features (through recurrent connections) - Hierarchical features (through layer stacking)

EXTRA CREDIT: Implement the MultiLayer RNN below, and then try training it and running it.

[17]:
```python
class MultiLayerRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers=2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)

        # Create ModuleLists for W_ih and W_hh
        # HINT: W_ih[0] should map from embed_size to hidden_size
        # subsequent W_ih[l] should map from hidden_size to hidden_size
        self.W_ih = nn.ModuleList(
            [
                nn.Linear(embed_size, hidden_size),
            ]
            + [
                nn.Linear(hidden_size, hidden_size)  # construct the needed␣
↪module
                for i in range(num_layers - 1)
            ]
        )
        self.W_hh = nn.ModuleList(
            [
                # YOUR CODE HERE
                nn.Linear(hidden_size, hidden_size)  # - construct the needed␣
↪module
                for i in range(num_layers)
            ]
        )

        self.fc = nn.Linear(hidden_size, vocab_size)
        self.tanh = nn.Tanh()

    def forward(self, x_token, hidden=None):
        if hidden is None:
            hidden = self.init_hidden(x_token.size(0))

        x = self.embedding(x_token)
        new_hidden = []

        for layer in range(self.num_layers):
            # Implement the recurrence for each layer
            layer_input = x if layer == 0 else new_hidden[-1]

            # YOUR CODE HERE: Apply the recurrence formula for this layer.
            # HINT: Use the formula from the markdown above
            # Compute h using self.W_ih[layer], self.W_hh[layer], and self.tanh,
            # (1) incorporating layer_input from layer stacking
            # (2) and the hidden[layer] from the temporal sequencing
            h = self.tanh(
                self.W_ih[layer](layer_input) + self.W_hh[layer](hidden[layer])
            )

            new_hidden.append(h)

        logits = self.fc(new_hidden[-1])
        return logits, new_hidden

    def init_hidden(self, batch_size):
        return [
            torch.zeros(batch_size, self.hidden_size, device=DEVICE)
            for _ in range(self.num_layers)
        ]

    def generate_text(self, dataset, length=100):
        self.eval()
        token = torch.randint(0, len(dataset.vocab), (1,), device=DEVICE)
        generated_tokens = [token.item()]
        hidden = self.init_hidden(1)

        for _ in range(length - 1):
            logits, hidden = self(token, hidden)
            # Convert logits to probabilities and sample the next token
            # HINT: Use torch.multinomial after applying softmax
            probs = nn.functional.softmax(logits, dim=-1).squeeze(0)   # YOUR␣
↪CODE HERE
            token = torch.multinomial(probs, num_samples=1)  # YOUR CODE HERE

            generated_tokens.append(token.item())

        generated_text = "".join([dataset.itos[i] for i in generated_tokens])
        return generated_text
```

3.4 (Extra Credit Continued) Multlayer training

(Extra credit): The training loop below can be use to test your multilayer model. You will need to fill in the missing core loop.

[18]:
```
def train_multilayer_rnn(model, dataloader, lr=1e-3, print_every=100,␣
↪max_iters=None):
    seq_len = dataloader.dataset.seq_length
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    model.train()
    losses = []
    start_time = time.time()

    for step, (x_batch, y_batch) in enumerate(dataloader):
        if max_iters is not None and step >= max_iters:
            break

        x_batch, y_batch = x_batch.to(DEVICE), y_batch.to(DEVICE)
        batch_size = x_batch.size(0)

        # Initialize hidden states for all layers
        h_prev = model.init_hidden(batch_size)
        loss = 0

        # Unroll over seq_len
        for t in range(seq_len):
            ## YOUR CODE HERE
            # This should all look quite similar to your previous training loop
            logits, h_prev = model(x_batch[:, t], h_prev)
            loss += criterion(logits, y_batch[:, t])
        loss = loss / seq_len
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

        if (step + 1) % print_every == 0:
            elapsed = time.time() - start_time
            print(
                f"(Multi-Layer RNN) Step {step+1}, Loss: {loss.item():.4f},␣
 ↪Time: {elapsed:.2f}s"
            )
            start_time = time.time()

    return losses
```

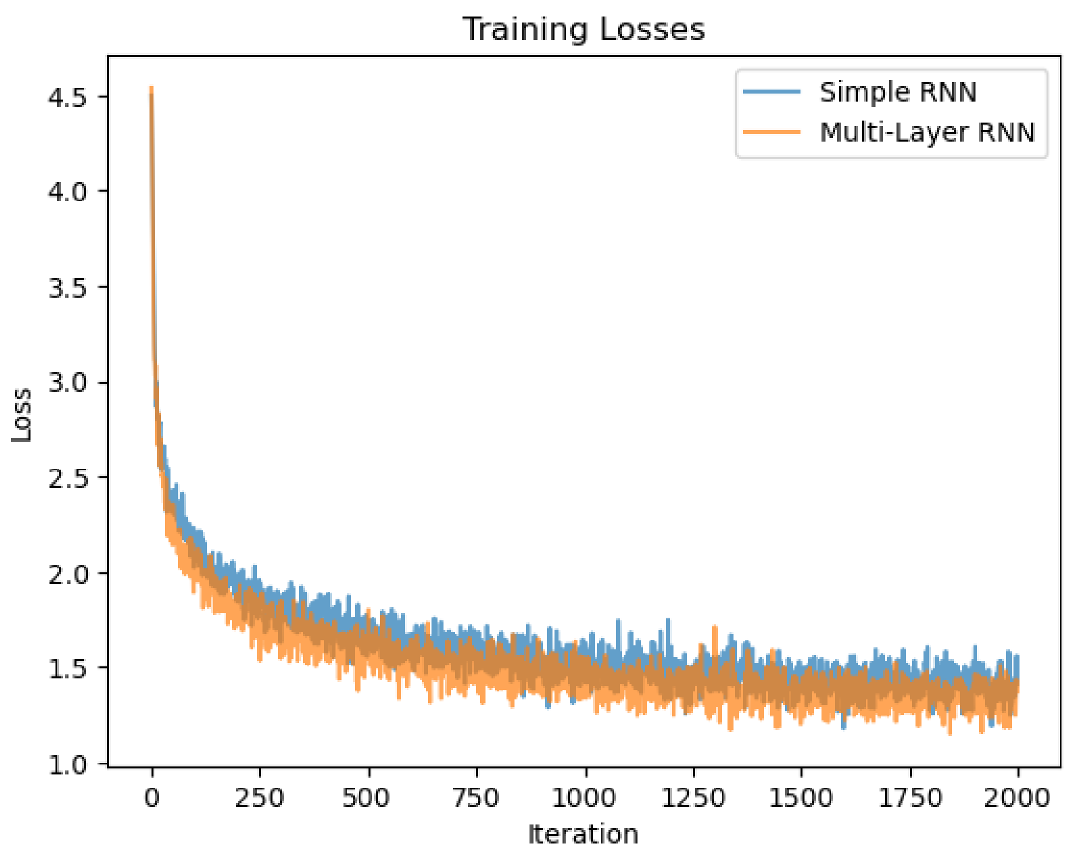
Extra credit, continued -

Experiment with different model sizes and other training configurations, and plot your findings.

```
[19]: hidden_size = 256
embed_size = 64
num_layers = 2  # We can experiment with different numbers of layers

multilayer_rnn = MultiLayerRNN(
    vocab_size=len(dataset.vocab),
    embed_size=embed_size,
    hidden_size=hidden_size,
    num_layers=num_layers,
).to(DEVICE)

multilayer_rnn_losses = train_multilayer_rnn(multilayer_rnn, dataloader,␣
 ↪max_iters=2000)

# Plot both models for comparison
plot_losses(
    {
        "Simple RNN": simple_rnn_losses,
        "Multi-Layer RNN": multilayer_rnn_losses,
    }
)

# Generate sample text from the multilayer RNN
sample_text = multilayer_rnn.generate_text(dataset)
print(f"\nSample text from Multi-Layer RNN: {sample_text}")
print("-" * 80)
```

```python
# Optional: Compare generations with different numbers of layers
for layers in [2, 3, 4]:
    model = MultiLayerRNN(
        vocab_size=len(dataset.vocab),
        embed_size=embed_size,
        hidden_size=hidden_size,
        num_layers=layers,
    ).to(DEVICE)
    losses = train_multilayer_rnn(model, dataloader, max_iters=2000)
    print(f"\nSample text from {layers}-layer RNN:")
    print(model.generate_text(dataset))
    print("-" * 80)
```

```
(Multi-Layer RNN) Step 100, Loss: 2.0780, Time: 7.63s
(Multi-Layer RNN) Step 200, Loss: 1.8867, Time: 7.41s
(Multi-Layer RNN) Step 300, Loss: 1.5485, Time: 7.90s
(Multi-Layer RNN) Step 400, Loss: 1.5876, Time: 7.44s
(Multi-Layer RNN) Step 500, Loss: 1.5599, Time: 6.96s
(Multi-Layer RNN) Step 600, Loss: 1.6362, Time: 6.73s
(Multi-Layer RNN) Step 700, Loss: 1.4040, Time: 6.78s
(Multi-Layer RNN) Step 800, Loss: 1.4700, Time: 6.63s
(Multi-Layer RNN) Step 900, Loss: 1.4503, Time: 6.82s
(Multi-Layer RNN) Step 1000, Loss: 1.5451, Time: 6.58s
(Multi-Layer RNN) Step 1100, Loss: 1.3475, Time: 6.59s
(Multi-Layer RNN) Step 1200, Loss: 1.3701, Time: 6.53s
(Multi-Layer RNN) Step 1300, Loss: 1.3650, Time: 6.50s
(Multi-Layer RNN) Step 1400, Loss: 1.4532, Time: 6.62s
(Multi-Layer RNN) Step 1500, Loss: 1.2029, Time: 6.53s
(Multi-Layer RNN) Step 1600, Loss: 1.3455, Time: 6.47s
(Multi-Layer RNN) Step 1700, Loss: 1.4698, Time: 6.43s
(Multi-Layer RNN) Step 1800, Loss: 1.3501, Time: 6.86s
(Multi-Layer RNN) Step 1900, Loss: 1.3297, Time: 6.56s
(Multi-Layer RNN) Step 2000, Loss: 1.3729, Time: 6.37s
```



```
Sample text from Multi-Layer RNN: Tim not was a splanted to be good becautiful
for the faces. She liked to help you'ge. They all saw a
--------------------------------------------------------------------------------
(Multi-Layer RNN) Step 100, Loss: 1.9506, Time: 6.37s
(Multi-Layer RNN) Step 200, Loss: 1.8869, Time: 6.36s
(Multi-Layer RNN) Step 300, Loss: 1.6373, Time: 6.35s
(Multi-Layer RNN) Step 400, Loss: 1.5996, Time: 6.63s
(Multi-Layer RNN) Step 500, Loss: 1.5618, Time: 6.65s
(Multi-Layer RNN) Step 600, Loss: 1.4855, Time: 6.70s
(Multi-Layer RNN) Step 700, Loss: 1.4709, Time: 7.01s
(Multi-Layer RNN) Step 800, Loss: 1.4284, Time: 6.94s
(Multi-Layer RNN) Step 900, Loss: 1.5277, Time: 7.16s
(Multi-Layer RNN) Step 1000, Loss: 1.3965, Time: 6.56s
(Multi-Layer RNN) Step 1100, Loss: 1.4818, Time: 6.40s
(Multi-Layer RNN) Step 1200, Loss: 1.3918, Time: 6.40s
(Multi-Layer RNN) Step 1300, Loss: 1.3585, Time: 6.43s
(Multi-Layer RNN) Step 1400, Loss: 1.3549, Time: 6.60s
(Multi-Layer RNN) Step 1500, Loss: 1.3192, Time: 6.43s
(Multi-Layer RNN) Step 1600, Loss: 1.2795, Time: 6.87s
```

```
(Multi-Layer RNN) Step 1700, Loss: 1.2567, Time: 6.49s
(Multi-Layer RNN) Step 1800, Loss: 1.3415, Time: 6.34s
(Multi-Layer RNN) Step 1900, Loss: 1.4720, Time: 6.42s
(Multi-Layer RNN) Step 2000, Loss: 1.3053, Time: 6.38s

Sample text from 2-layer RNN:
Can agair!" He said, "Thank outside.

Mumouse waited explore and dizing and he finastared find afte
--------------------------------------------------------------------------------
(Multi-Layer RNN) Step 100, Loss: 2.0534, Time: 8.68s
(Multi-Layer RNN) Step 200, Loss: 1.6748, Time: 8.70s
(Multi-Layer RNN) Step 300, Loss: 1.6672, Time: 8.70s
(Multi-Layer RNN) Step 400, Loss: 1.6335, Time: 8.65s
(Multi-Layer RNN) Step 500, Loss: 1.5736, Time: 8.64s
(Multi-Layer RNN) Step 600, Loss: 1.4525, Time: 8.63s
(Multi-Layer RNN) Step 700, Loss: 1.5138, Time: 8.63s
(Multi-Layer RNN) Step 800, Loss: 1.5224, Time: 8.65s
(Multi-Layer RNN) Step 900, Loss: 1.3732, Time: 8.61s
(Multi-Layer RNN) Step 1000, Loss: 1.4623, Time: 8.82s
(Multi-Layer RNN) Step 1100, Loss: 1.3229, Time: 9.03s
(Multi-Layer RNN) Step 1200, Loss: 1.4099, Time: 8.92s
(Multi-Layer RNN) Step 1300, Loss: 1.4121, Time: 8.83s
(Multi-Layer RNN) Step 1400, Loss: 1.2162, Time: 9.52s
(Multi-Layer RNN) Step 1500, Loss: 1.4313, Time: 8.85s
(Multi-Layer RNN) Step 1600, Loss: 1.4521, Time: 8.99s
(Multi-Layer RNN) Step 1700, Loss: 1.3198, Time: 9.80s
(Multi-Layer RNN) Step 1800, Loss: 1.2590, Time: 9.15s
(Multi-Layer RNN) Step 1900, Loss: 1.3316, Time: 8.77s
(Multi-Layer RNN) Step 2000, Loss: 1.4127, Time: 8.83s

Sample text from 3-layer RNN:
" did no learned he was curiends coldert infor or?"

Ben made the good himself and some noad the tre
--------------------------------------------------------------------------------
(Multi-Layer RNN) Step 100, Loss: 2.1079, Time: 11.06s
(Multi-Layer RNN) Step 200, Loss: 1.7892, Time: 11.17s
(Multi-Layer RNN) Step 300, Loss: 1.6788, Time: 11.35s
(Multi-Layer RNN) Step 400, Loss: 1.6297, Time: 11.14s
(Multi-Layer RNN) Step 500, Loss: 1.5641, Time: 11.10s
(Multi-Layer RNN) Step 600, Loss: 1.5444, Time: 10.97s
(Multi-Layer RNN) Step 700, Loss: 1.5652, Time: 10.95s
(Multi-Layer RNN) Step 800, Loss: 1.4042, Time: 11.14s
(Multi-Layer RNN) Step 900, Loss: 1.6407, Time: 11.26s
(Multi-Layer RNN) Step 1000, Loss: 1.4517, Time: 11.21s
(Multi-Layer RNN) Step 1100, Loss: 1.4365, Time: 11.18s
(Multi-Layer RNN) Step 1200, Loss: 1.4801, Time: 11.10s
```

```
(Multi-Layer RNN) Step 1300, Loss: 1.3987, Time: 10.82s
(Multi-Layer RNN) Step 1400, Loss: 1.2781, Time: 10.89s
(Multi-Layer RNN) Step 1500, Loss: 1.3506, Time: 10.85s
(Multi-Layer RNN) Step 1600, Loss: 1.3749, Time: 10.79s
(Multi-Layer RNN) Step 1700, Loss: 1.3561, Time: 11.20s
(Multi-Layer RNN) Step 1800, Loss: 1.3969, Time: 11.31s
(Multi-Layer RNN) Step 1900, Loss: 1.2993, Time: 10.96s
(Multi-Layer RNN) Step 2000, Loss: 1.4011, Time: 10.99s

Sample text from 4-layer RNN:
at was fix. Tom was a lion.
Once upon a time, there was a firl, come went to differ. She time, there
--------------------------------------------------------------------------------
```

This is the end of the extra-credit section. Back to the required part of the homework.

4 Computational Efficiency of the Classic RNN

Let's analyze the efficiency of the RNN. For an Elman RNN with: - Hidden dimension D - Vocabulary size V - Sequence length T - Number of layers L

For this exercise, you can assume that the embedding dimension is the same as the hidden dimension D.

1. How many FLOPs are needed to generate T tokens? (In this, and all the questions in this HW, you can ignore small calculations like biases and assume that there's a single batch. You can also assume that an AxB matrix and a BxC matrix take 2ABC FLOPs to be multiplied together (about half of those are scalar multiplications and half are scalar additions). Don't worry about the final projection to the vocabulary dimension. Even though we implemented a single layer, in the general case the model could implement L layers per step.)

   YOUR ANSWER: To process a single token and a single layer we require ***2*** matrix multiplications with dimension ***D*** x ***D*** and ***D*** x ***1***, which comes to ***(***$4D^2$***)*** FLOPS per token. Doing it for L layers and T tokens requires a total of ***(***$4TLD^2$***)*** FLOPs.

2. How many FLOPs are needed to train on a sequence of T tokens? (Not counting training the classifier layer W_{ff}.)

   YOUR ANSWER: For each layer and for a single token, we need to calculate the gradients with respect to W_{ih} and W_{hh}. Each of these is a ***D*** by ***D*** matrix multiplication which comes to ______($4D^2$)______ FLOPS. We also need to propagate the gradients with respect to the input (to train the embedding), which adds ***(***$2D^2$***)*** FLOPS for a single token, and and hidden states (to backpropgate through time) which is ***(***$2D^2$***)*** FLOPS per token. So for each layer and for a single token, we need ***(***$8D^2$***)*** FLOPS. We do this for L layers and T tokens, so the total is ______($8TLD^2$)______ FLOPs.

3. Counting FLOPS can overestimate how expensive neural networks are on a GPU, which can parallelize the operations in a matrix (or matrix-vector) operation very efficiently. Since a modern GPU can process a fairly good-sized matrix multiplication in a small amount of real-world wall-clock time, it can be more informative to count these matrix multiplications. How many sequential matrix multiplications are needed to generate T tokens?

YOUR ANSWER: For each layer and for a single token, we need ***2 (But can be done in parallel)*** matrix multiplications, So over all L layers and T tokens we need ***TL*** sequential matrix multiplications.

4. How many sequential matrix multiplications are needed to train on T tokens?

   YOUR ANSWER: For each layer and for a single token, we need ***4 (Can be done as 2 pairs in parallel)*** matrix multiplications, So over all L layers and T tokens we need ***2TL*** sequential matrix multiplications.

5 The GRU Cell

Can we improve on the RNN? We can try a GRU, or Gated Recurrent Unit (Cho et al., 2014)[https://arxiv.org/abs/1406.1078].

A Gated Recurrent Unit (GRU) is a gating mechanism for RNNs that helps control information flow. Let's understand its components:

5.1 Gate Mechanisms

The GRU has two gates and a memory update mechanism:

1. **Reset Gate** (r_t):
$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1})$$
   Controls how much of the previous hidden state to use in forming the new memory content.
2. **Update Gate** (z_t):
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1})$$
   Controls how much of the previous hidden state to keep vs. how much to update.
3. **New Content** (n_t):
$$n_t = \tanh(W_{xn}x_t + r_t \odot (W_{hn}h_{t-1}))$$
   Proposes new content based on input and gated previous state.
4. **Final Update** (h_t): The hidden state is updated as:
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1}$$

In the above: - σ is the sigmoid function - $\odot$ represents element-wise multiplication - W_* are learnable weight matrices - x_t is the input at time t - h_{t-1} is the previous hidden state

This allows the GRU to: - Learn long-term dependencies (through the update gate) - Adaptively reset its state (through the reset gate) - Propose relevant new content (through the memory content)

5.2 GRUCell code

Here is how the code looks:

```
[20]: class GRUCell(nn.Module):
          def __init__(self, input_size, hidden_size):
              super().__init__()
```

```
        self.hidden_size = hidden_size

        # Reset gate parameters
        self.W_xr = nn.Linear(input_size, hidden_size, bias=False)
        self.W_hr = nn.Linear(hidden_size, hidden_size, bias=False)

        # Update gate parameters
        self.W_xz = nn.Linear(input_size, hidden_size, bias=False)
        self.W_hz = nn.Linear(hidden_size, hidden_size, bias=False)

        # New content parameters
        self.W_xn = nn.Linear(input_size, hidden_size, bias=False)
        self.W_hn = nn.Linear(hidden_size, hidden_size, bias=False)

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, x, hidden):
        # Compute reset gate
        r_t = self.sigmoid(self.W_xr(x) + self.W_hr(hidden))

        # Compute update gate
        z_t = self.sigmoid(self.W_xz(x) + self.W_hz(hidden))

        # Compute new memory content
        n_t = self.tanh(self.W_xn(x) + r_t * self.W_hn(hidden))

        # Update hidden state
        h_t = (1 - z_t) * n_t + z_t * hidden

        return h_t, r_t, z_t, n_t
```

[21]:
```
def plot_gru_2d_step(
    data_input=0.0,
    control_input=0.0,
    data_hidden=0.0,
    control_hidden=0.0,
    reset_weight=0.0,
    update_weight=0.0,
    new_input_weight=3.0,
    new_hidden_weight=0.0,
):
    gru = GRUCell(input_size=2, hidden_size=2)
    # Set specific weights for control and data flow

    with torch.no_grad():
        # Zero out all weights first
        gru.W_xr.weight.zero_()
        gru.W_hr.weight.zero_()
        gru.W_xz.weight.zero_()
        gru.W_hz.weight.zero_()
        gru.W_xn.weight.zero_()
        gru.W_hn.weight.zero_()
        # Gates controlled by control dimension but affecting data dimension

        gru.W_xr.weight[0, 1] = reset_weight  # Reset gate

        gru.W_hr.weight[0, 1] = reset_weight

        gru.W_xz.weight[0, 1] = update_weight  # Update gate

        gru.W_hz.weight[0, 1] = update_weight

        # New memory weights for data dimension

        gru.W_xn.weight[0, 0] = new_input_weight  # From input

        gru.W_hn.weight[0, 0] = new_hidden_weight  # From hidden state

    # Create input and hidden tensors

    x = torch.tensor([[data_input, control_input]], dtype=torch.float32)
    h = torch.tensor([[data_hidden, control_hidden]], dtype=torch.float32)
    # Get GRU outputs

    with torch.no_grad():
        new_h, r, z, n = gru(x, h)
    # Create figure focusing only on data dimension

    fig, ax = plt.subplots(figsize=(12, 5))
    # Plot all relevant data dimension values

    labels = [
        "x[0]\nInput",
        "h[0]\nPrev Hidden",
        "r[0]\nReset Gate",
        "z[0]\nUpdate Gate",
        "n[0]\nNew Content",
        "h_new[0]\nFinal Output",
    ]
    values = [
        x[0, 0].item(),
        h[0, 0].item(),
        r[0, 0].item(),
        z[0, 0].item(),
        n[0, 0].item(),
        new_h[0, 0].item(),
    ]
    colors = ["blue", "gray", "lightblue", "lightgreen", "orange", "red"]
    # Create bars without labels on x-axis

    bars = ax.bar(range(len(values)), values, color=colors)
    ax.set_ylim(-2, 2)
    ax.set_title("GRU Data Flow (Data Dimension)")
    # Remove x-axis ticks

    ax.set_xticks([])
    # Add labels inside the plot

    for idx, (bar, label, value) in enumerate(zip(bars, labels, values)):
        # Position labels near the top of the plot

        y_pos = 1.8  # Position at top of plot

        # Add the label

        ax.text(idx, y_pos, label, ha="center", va="top")
        # Add the value on top if bar is positive, below if negative

        value_pos = 0.1 if value < 0 else -0.1

        ax.text(
            idx,
            value + value_pos,
            f"{value:.2f}",
            ha="center",
            va="bottom" if value < 0 else "top",
        )
    plt.tight_layout()
    plt.show()


# Create sliders with wider labels and organized in two columns

from ipywidgets import HBox, VBox

# Input sliders


data_input_slider = FloatSlider(
    value=0.0,
    min=-2.0,
    max=2.0,
    step=0.1,
    readout_format=".1f",
    description="Input x[0] (Data):",
    style={"description_width": "140px"},
)

control_input_slider = FloatSlider(
    value=3.0,
    min=-2.0,
    max=2.0,
    step=0.1,
    readout_format=".1f",
    description="Input x[1] (Control):",
    style={"description_width": "140px"},
)

data_hidden_slider = FloatSlider(
    value=0.0,
    min=-2.0,
    max=2.0,
    step=0.1,
    readout_format=".1f",
    description="Hidden x[0] (Data):",
    style={"description_width": "140px"},
)

control_hidden_slider = FloatSlider(
    value=0.0,
    min=-2.0,
    max=2.0,
    step=0.1,
    readout_format=".1f",
    description="Hidden x[1] (Control):",
    style={"description_width": "140px"},
)

# Weight sliders


reset_weight_slider = FloatSlider(
    value=3.0,
    min=-3.0,
    max=3.0,
    step=0.2,
    readout_format=".1f",
    description="Reset W_xr W_hr[0,1]:",
    style={"description_width": "160px"},
)

update_weight_slider = FloatSlider(
    value=-3.0,
    min=-3.0,
    max=3.0,
    step=0.2,
    readout_format=".1f",
    description="Update W_xz W_hz[0,1]:",
    style={"description_width": "160px"},
)

new_input_weight_slider = FloatSlider(
    value=3.0,
    min=-3.0,
    max=3.0,
    step=0.2,
    readout_format=".1f",
    description="New Content W_xn[0,0]:",
    style={"description_width": "160px"},
)

new_hidden_weight_slider = FloatSlider(
    value=3.0,
    min=-3.0,
    max=3.0,
    step=0.2,
    readout_format=".1f",
    description="New Content W_hn[0,0]:",
    style={"description_width": "160px"},
)

# Arrange sliders in four columns


col_1 = VBox([data_input_slider, control_input_slider])

col_2 = VBox([data_hidden_slider, control_hidden_slider])

col_3 = VBox([reset_weight_slider, update_weight_slider])

col_4 = VBox([new_input_weight_slider, new_hidden_weight_slider])

slider_layout = HBox([col_1, col_2, col_3, col_4])
```

```
interactive_plot = interactive(
    plot_gru_2d_step,
    data_input=data_input_slider,
    control_input=control_input_slider,
    data_hidden=data_hidden_slider,
    control_hidden=control_hidden_slider,
    reset_weight=reset_weight_slider,
    update_weight=update_weight_slider,
    new_input_weight=new_input_weight_slider,
    new_hidden_weight=new_hidden_weight_slider,
)

# Display both the sliders and the plot


VBox([slider_layout, interactive_plot.children[-1]])
```

```
[21]: VBox(children=(HBox(children=(VBox(children=(FloatSlider(value=0.0,
      description='Input x[0] (Data):', max=2.0,…
```

6 Understanding GRU Behavior By Interactive Experiments

The GRU demonstration above is set up to show how a GRU can selectively update, preserve, or modify information.

We have used a simplified two-dimenstional hidden state space.

The way we have set up the weight matrices, the data dimension (index 0) carries the actual values we want to process, while the control dimension (index 1) determines how the gates behave. For the first exercises, we will leave the Input x[1] (Control) to be a high value, 2.0. Then you will experiment with altering the weights, to see how they affect the relationship between inputs and outputs.

6.1 Exercise 1: Understanding Default Behavior

The default settings (New Content Weights = 3.0) are configured to make the hidden state contain an attenuated sum of the input and the previous hidden state.

Questions: 1. Move the "Input x[0] (Data)" slider. Does the (a) new content and (b) final output wiggle in the same direction, opposite direction, or not at all? > YOUR ANSWER: When the input varies, the new content wiggles ***in the same direction***. The final output wiggles ***in the same direction***. 2. Move the "Hidden x[0] (Data)" slider. Does the new content and final output respond similarly? > YOUR ANSWER: When the hidden state varies, the new content wiggles ***in the same direction***. The final output wiggles ***in the same direction***. 3. Set both input and hidden state data values high (e.g., 2.0). What happens to the output - is it high, low, or zero? > YOUR ANSWER: When both input and hidden state are high, the new content is ***high***. The final output is ***high***. Due to the attenuation, the maximum range of the new content is ***[-1.0, 1.0]***.

6.2 Exercise 2: Computing Differences

Task: Modify the "New Content" weights (W_xn[0,0] and W_hn[0,0]) to make the GRU compute the attenuated difference between input and hidden state, rather than their sum.

Success Criteria: - The new content should wiggle in the opposite direction of the hidden state - When both input and hidden state data are high, the new content should be close to zero - The final output should reflect this difference computation

Question: What weights did you use to achieve this? 4. Set both input and hidden state data values high (e.g., 2.0). What happens to the output - is it high, low, or zero? > YOUR ANSWER: To compute differences, set the weights ***W_hn[0,0] to -3.0, while keeping W_xn[0,0] as its default (3.0)***

6.3 Exercise 3: Perfect Memory

Task: Find settings of the weights that cause the GRU to (almost) completely ignore the new content and instead copy the old hidden state to the new hidden state.

Questions: 1. What weight settings achieve this? > YOUR ANSWER: For perfect memory, use the weight settings ***W_xz[0,1]=W_hz[0,1]=3.0***. 2. Which gate is primarily responsible for this behavior? > YOUR ANSWER: ***Update gate***. 3. How do you know that this setting is ignoring the new content and using just copying old hidden state? > YOUR ANSWER: ***In order to achieve that, we need z_t to be as close as possible to 1.0. Thus W_{xz} and W_{hz} need to be very high to yield such z_t value.***

6.4 Exercise 4: Pure Input Mode

Task: Find settings of the weights that make the GRU (almost) completely ignore the old hidden state and only use the new input, when computing the new content.

Questions: 1. What weight settings achieve this? > YOUR ANSWER: To ignore the old hidden state, use the weight settings ***W_xr[0,1]=W_hr[0,1]=-3.0 or W_hn[0,0] = 0.0***. 2. Which gate is primarily responsible for this behavior? > YOUR ANSWER: ***Reset Gate or New Content(?)***. 3. How can you verify your solution is correct? > YOUR ANSWER: Enable to make new content only dependent to the input (i.e. independent from h_{t-1}), we have to let $r_t \odot W_{hn} = \vec{0}$, this can achieve by having either r_t or W_{hn} to 0.

6.5 Final Observation: Control Channel

For all the above experiments, we kept the "Input [Control]" value high at 2.0.

Task: For each of the previous configurations you found, try setting the control input to 0.0 instead of 2.0.

Questions: 1. What happens to the gating behavior? > YOUR ANSWER: ***r[0]=z[0]=0.50, the final output became less dependent on the input, more on the previous hidden state***. 2. Why is this control channel important for the GRU's operation? > YOUR ANSWER: ***It is affecting how the cell integrates its inputs and hidden state***. 3. In a practical application, what might this control channel represent? > YOUR ANSWER: ***It might be useful for tasks like masked LM or image inpainting, the control channel can be used to filter out***

***the tokens/pixels that need to be masked out. But I'm not sure if leaving a separate channel merely for controlling the network's behavior is practical.*.**

7 Training an RNN using a GRU cell

Let's try the GRU on real data. We'll keep the same hyperparameters as the previous RNN model.

Exercise: Modify the code below to incorporate the GRUCell, and fill in the standard autoregressive generation loop. Then train the model and compare it to the RNN.

Compare the behavior of the GRU RNN with the vanilla Elman RNN.

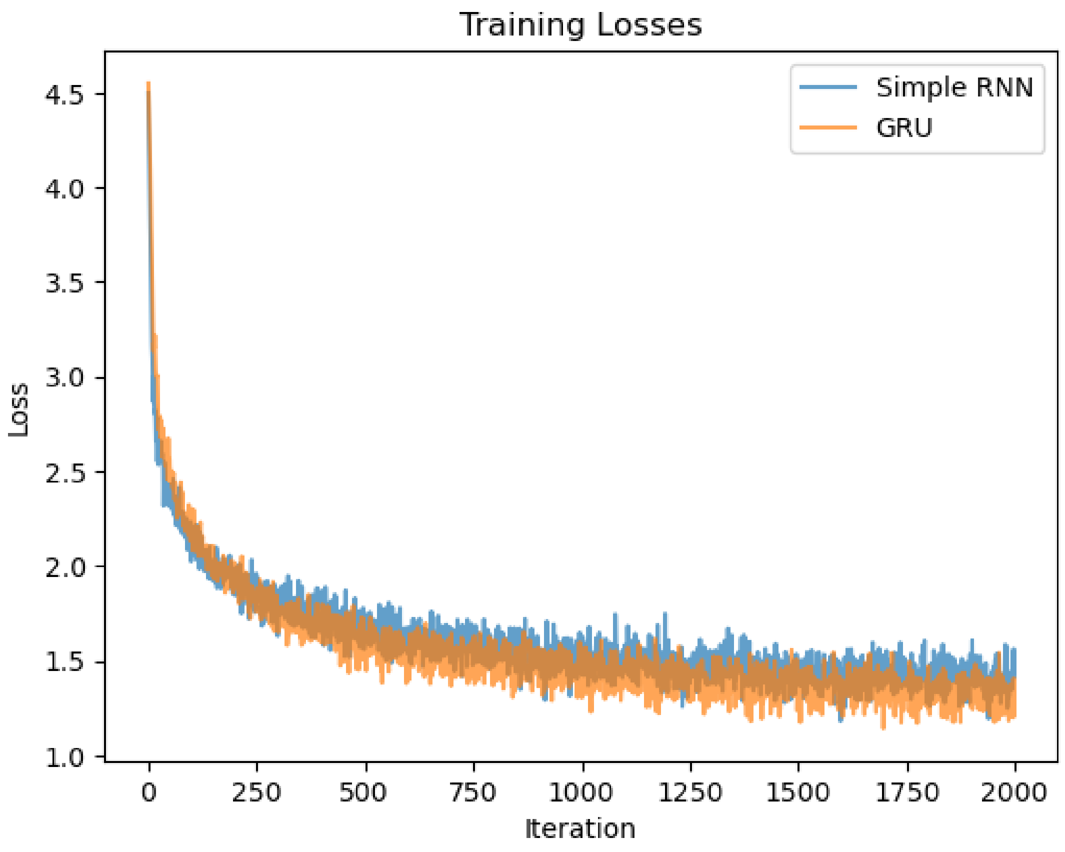
YOUR FINDINGS: Compared to the Elman RNN, the GRU RNN….

[22]:
```python
class GRULM(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.gru_cell = GRUCell(
            embed_size, hidden_size
        )  # construct a GRUCell with the right sizes.
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden=None):
        # x shape: (batch_size)
        if hidden is None:
            hidden = self.init_hidden(x.size(0))

        # Embed tokens
        x_emb = self.embedding(x)  # (batch_size, embed_size)

        # Process tokens
        h = hidden

        h, _, _, _ = self.gru_cell(x_emb, h)  # invoke gru_cell with the right␣
↪inputs.
        logits = self.fc(h)

        # # Stack outputs along sequence dimension
        # outputs = torch.stack(outputs, dim=1)
        return logits, h

    def init_hidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size, device=DEVICE)

    def generate_text(self, dataset, length=100):
        self.eval()
        # Start with a random token
        token = torch.randint(0, len(dataset.vocab), (1,), device=DEVICE)
        generated_tokens = [token.item()]
        hidden = self.init_hidden(1)

        # Generate one token at a time
        for _ in range(length - 1):
            # Get predictions
            # compute the token by filling in the standard autoregressive
            # generation sampling loop here.
            outputs, hidden = self.forward(token, hidden)
            token = torch.argmax(outputs, dim=1)
            generated_tokens.append(token.item())

        # Convert tokens back to text using the dataset's vocabulary
        generated_text = "".join([dataset.itos[i] for i in generated_tokens])
        return generated_text
```

Run the training code below; the GRU RNN should perform well - if it doesn't, try debugging your code!

[23]:
```
# Set up model with same hyperparameters as RNN
hidden_size = 256
embed_size = 64
gru_model = GRULM(len(dataset.vocab), embed_size, hidden_size).to(DEVICE)

# Use the same training function as before
gru_losses = train_language_model(gru_model, dataloader, max_iters=2000)

# Generate sample text from models and compare
print("\nSample text from GRU:")
sample_text = gru_model.generate_text(dataset)
print(sample_text)
print("-" * 80)

# Plot losses to compare models
plot_losses(
    {
        "Simple RNN": simple_rnn_losses,
        "GRU": gru_losses,
    }
)
```

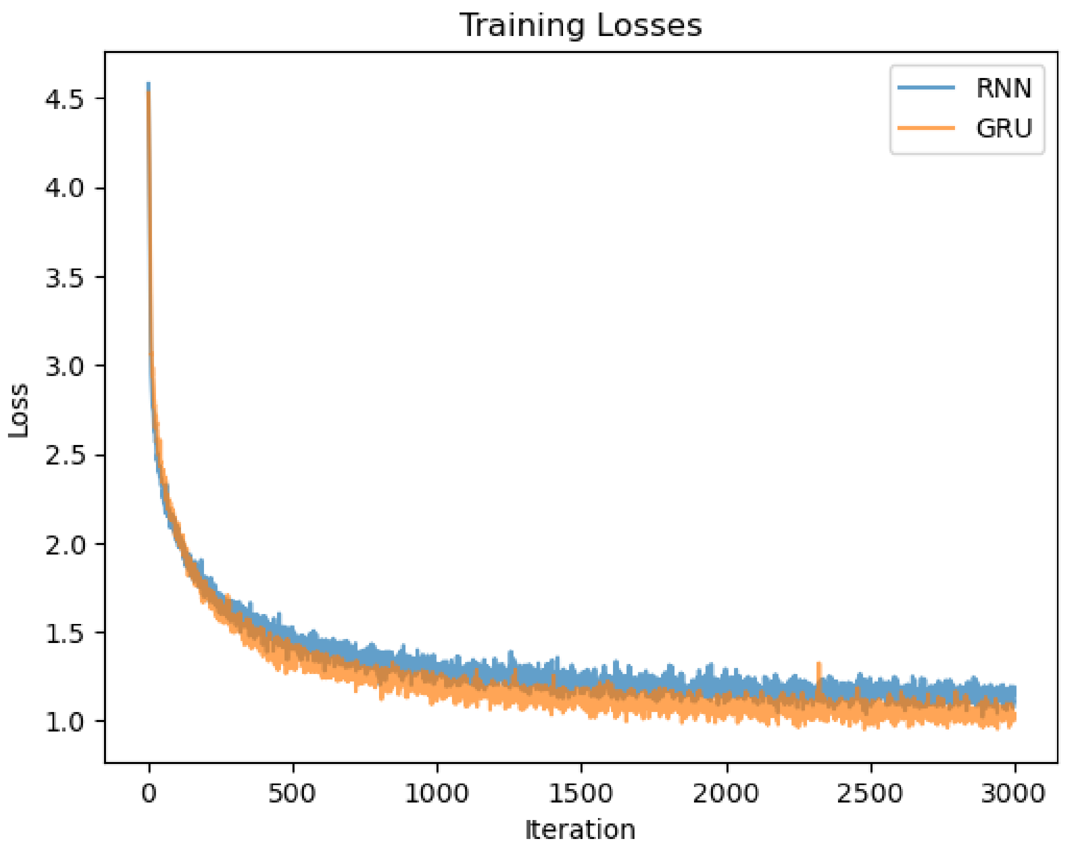
```
Step 100, Loss: 2.1771, Time: 8.54s
Step 200, Loss: 2.0011, Time: 8.19s
Step 300, Loss: 1.7338, Time: 8.31s
Step 400, Loss: 1.6901, Time: 8.44s
Step 500, Loss: 1.6534, Time: 8.48s
Step 600, Loss: 1.5041, Time: 8.37s
Step 700, Loss: 1.3647, Time: 8.53s
Step 800, Loss: 1.4246, Time: 8.27s
Step 900, Loss: 1.5397, Time: 8.24s
Step 1000, Loss: 1.4084, Time: 8.27s
Step 1100, Loss: 1.4403, Time: 8.41s
Step 1200, Loss: 1.4454, Time: 8.47s
Step 1300, Loss: 1.3582, Time: 8.40s
Step 1400, Loss: 1.3391, Time: 8.33s
Step 1500, Loss: 1.3056, Time: 8.30s
Step 1600, Loss: 1.3563, Time: 8.18s
Step 1700, Loss: 1.3707, Time: 8.36s
Step 1800, Loss: 1.2672, Time: 8.47s
Step 1900, Loss: 1.2554, Time: 8.36s
Step 2000, Loss: 1.2135, Time: 8.51s

Sample text from GRU:
4't saw a big beautiful to the park. He was so happy to the park. He was so
happy to the park. He wa
--------------------------------------------------------------------------------
```



Your loss curves should look pretty similar. What's going on? Whether the GRU is better or worse than an RNN can depend on many factors, including the dataset, the model architectures, and the hyperparameters. One especially common story about GRUs is that they're better at capturing long-term dependencies than RNNs. Let's see if that's true, by bumping up the sequence length.

[24]:
```
max_iters = 3_000

dataset = CharLMDataset(full_text, seq_length=100)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
 ↪drop_last=True)

# Train RNN
simple_rnn = SimpleRNN(len(dataset.vocab), embed_size, hidden_size).to(DEVICE)
rnn_losses = train_language_model(simple_rnn, dataloader, max_iters=max_iters)

sample_text = simple_rnn.generate_text(dataset)
print(f"\nSample text from RNN: {sample_text}")
print("-" * 80)

# Train GRU using same architecture and training function
simple_gru = GRULM(
    vocab_size=len(dataset.vocab), embed_size=embed_size,␣
 ↪hidden_size=hidden_size
).to(DEVICE)
gru_losses = train_language_model(simple_gru, dataloader, max_iters=max_iters)

sample_text = simple_gru.generate_text(dataset)
print(f"\nSample text from GRU: {sample_text}")
print("-" * 80)

# Plot loss curves
plot_losses(
    {
        "RNN": rnn_losses,
        "GRU": gru_losses,
    }
)
```

```
Step 100, Loss: 2.0991, Time: 17.78s
Step 200, Loss: 1.7105, Time: 17.57s
Step 300, Loss: 1.6494, Time: 18.21s
Step 400, Loss: 1.5493, Time: 18.50s
Step 500, Loss: 1.3999, Time: 19.31s
Step 600, Loss: 1.4510, Time: 18.56s
Step 700, Loss: 1.3841, Time: 18.29s
Step 800, Loss: 1.3028, Time: 18.35s
Step 900, Loss: 1.3180, Time: 18.29s
Step 1000, Loss: 1.3098, Time: 18.19s
Step 1100, Loss: 1.2586, Time: 18.39s
Step 1200, Loss: 1.2684, Time: 17.97s
Step 1300, Loss: 1.2582, Time: 17.70s
Step 1400, Loss: 1.3414, Time: 18.10s
Step 1500, Loss: 1.2402, Time: 17.75s
Step 1600, Loss: 1.1467, Time: 17.95s
Step 1700, Loss: 1.1636, Time: 17.80s
Step 1800, Loss: 1.1839, Time: 17.85s
Step 1900, Loss: 1.1773, Time: 18.57s
Step 2000, Loss: 1.2909, Time: 17.99s
Step 2100, Loss: 1.2152, Time: 17.91s
Step 2200, Loss: 1.1602, Time: 17.93s
Step 2300, Loss: 1.1550, Time: 17.91s
Step 2400, Loss: 1.1937, Time: 18.37s
Step 2500, Loss: 1.2403, Time: 18.60s
Step 2600, Loss: 1.1159, Time: 18.49s
Step 2700, Loss: 1.1429, Time: 18.55s
Step 2800, Loss: 1.1434, Time: 18.45s
Step 2900, Loss: 1.1877, Time: 18.41s
Step 3000, Loss: 1.1456, Time: 18.08s

Sample text from RNN: e decided to carrot. The cory!"

Ben said yellonâ€ t some muchangerful. At her big friends. One day,
--------------------------------------------------------------------------------
Step 100, Loss: 2.0351, Time: 39.84s
Step 200, Loss: 1.7412, Time: 39.90s
Step 300, Loss: 1.5615, Time: 38.84s
Step 400, Loss: 1.5135, Time: 38.85s
Step 500, Loss: 1.3264, Time: 39.05s
Step 600, Loss: 1.3761, Time: 38.86s
Step 700, Loss: 1.2997, Time: 39.10s
Step 800, Loss: 1.2243, Time: 38.75s
Step 900, Loss: 1.1529, Time: 38.73s
Step 1000, Loss: 1.1120, Time: 39.25s
Step 1100, Loss: 1.1694, Time: 39.51s
Step 1200, Loss: 1.1646, Time: 39.29s
Step 1300, Loss: 1.1729, Time: 38.84s
Step 1400, Loss: 1.1938, Time: 38.78s
Step 1500, Loss: 1.1150, Time: 38.71s
Step 1600, Loss: 1.0926, Time: 38.76s
Step 1700, Loss: 1.0522, Time: 38.81s
Step 1800, Loss: 1.0497, Time: 38.63s
Step 1900, Loss: 1.0669, Time: 38.66s
Step 2000, Loss: 1.0801, Time: 38.97s
Step 2100, Loss: 1.0605, Time: 38.84s
Step 2200, Loss: 1.0356, Time: 38.66s
Step 2300, Loss: 1.0110, Time: 38.80s
Step 2400, Loss: 1.0579, Time: 38.71s
Step 2500, Loss: 1.0169, Time: 38.65s
Step 2600, Loss: 1.0783, Time: 38.73s
Step 2700, Loss: 1.0362, Time: 38.71s
Step 2800, Loss: 1.0108, Time: 39.24s
Step 2900, Loss: 1.0176, Time: 38.83s
Step 3000, Loss: 1.0370, Time: 38.81s

Sample text from GRU: y was very happy. They were happy to help her and the bird
was very happy. They were happy to help h
--------------------------------------------------------------------------------
```



8 Transformer LM

Can we do even better? The Transformer (Vaswani et al., 2017)[https://arxiv.org/abs/1706.03762] seems like a good candidate. We'll adapt Andrej Karpathy's nanoGPT (https://github.com/karpathy/nanoGPT), itself based on GPT2, to our setting. nanoGPT uses a pre-LayerNorm architecture, GeLU activations, and simple learned positional encodings.

For another take on implementing a transformer, see the ARENA materials at https://arena-chapter1-transformer-interp.streamlit.app/[1.1]_Transformer_from_Scratch

Here is an implementation guide for the transformer below.

8.1 1. Main Forward Pass (Transformer class)

The forward pass combines token and positional embeddings, then processes them through transformer blocks:

Mathematical formulation: 1. Token embeddings: $E_{tok} = \text{Embedding}(idx) \in \mathbb{R}^{b \times t \times d}$ 2. Position embeddings: $E_{pos} = \text{Embedding}(\text{range}(t)) \in \mathbb{R}^{t \times d}$ 3. Combined: $x = E_{tok} + E_{pos}$

Implementation:

```python
# Get token embeddings
tok_emb = self.transformer.token_embedding(idx)  # shape: (b, t, n_embd)


# Get position embeddings
pos_emb = self.transformer.position_embedding(pos)  # shape: (t, n_embd)


# Combine embeddings
x = tok_emb + pos_emb

# Process through transformer blocks
for block in self.transformer.blocks:
    x = block(x)


# Final layer norm
x = self.transformer.ln_f(x)
```

8.2 2. Block Forward Pass

Each transformer block applies attention and MLP with residual connections:

Mathematical formulation: 1. Attention: $x = x + \text{Attention}(\text{LayerNorm}(x))$ 2. MLP: $x = x + \text{MLP}(\text{LayerNorm}(x))$

Implementation:

```python
# Attention with residual connection
x = x + self.attn(self.ln_1(x))


# MLP with residual connection
x = x + self.mlp(self.ln_2(x))
```

8.3 3. Attention Implementation

The self-attention mechanism computes weighted combinations of values based on query-key similarity:

Mathematical formulation: 1. Project to Q/K/V: $Q, K, V = WX$ where $W \in \mathbb{R}^{3d \times d}$ 2. Attention scores: $A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right)$ 3. Output: $y = AV$

Implementation:

```
# Manual implementation of attention (non-flash version)
# Scale dot product of Q and K
att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))


# Apply causal mask (sets future positions to -inf before softmax)
att = att.masked_fill(self.bias[:, :, :T, :T] == 0, float("-inf"))


# Apply softmax to get attention weights
att = F.softmax(att, dim=-1)


# Multiply by values
y = att @ v  # shape: (B, nh, T, hs)


# Reshape back: combine all heads
y = y.transpose(1, 2).contiguous().view(B, T, C)
```

8.4 4. Text Generation

For generating text, we: 1. Start with a sequence containing one random token 2. Repeatedly: - Get model predictions - Sample next token from the predicted distribution - Append to sequence

Implementation:

```
# Model is in eval mode
self.eval()


# Start with a random token
x = torch.randint(0, len(dataset.vocab), (1, 1), device=DEVICE)

for _ in range(max_new_tokens):
    # Get model predictions
    logits, _ = self(x)

    # Get probabilities for next token
    logits = logits[:, -1, :]
    probs = F.softmax(logits, dim=-1)

    # Sample next token
    x_next = torch.multinomial(probs, num_samples=1)
```

```
        # Append to sequence
        x = torch.cat((x, x_next), dim=1)


    # Convert to text
    text = "".join([dataset.itos[i.item()] for i in x[0]])
```

8.5 Implementation Tips

1. **Shape checking**: The most common issues arise from incorrect tensor shapes. Key shapes:
   - Input idx: (batch_size, seq_len)
   - Embeddings: (batch_size, seq_len, n_embd)
   - Attention: (batch_size, n_heads, seq_len, head_size)
2. **Causal masking**: The causal mask ensures tokens can only attend to previous positions:

   ```
   # Mask shape: (1, 1, seq_len, seq_len)
   mask = torch.tril(torch.ones(block_size, block_size))
   ```
3. **Multi-head attention**: Remember to:
   - Split attention heads: `view(B, T, n_head, head_size)`
   - Transpose for matrix multiplication: `transpose(1, 2)`
   - Combine heads at end: `view(B, T, n_embd)`

[30]:
```
class Transformer(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.transformer = nn.ModuleDict(
            dict(
                token_embedding=nn.Embedding(config.vocab_size, config.n_embd),
                position_embedding=nn.Embedding(config.block_size, config.
↪n_embd),
                blocks=nn.ModuleList([Block(config) for _ in range(config.
↪n_layer)]),
                ln_f=nn.LayerNorm(config.n_embd),
            )
        )
        self.config = config
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        self.transformer.token_embedding.weight = (
            self.lm_head.weight
        )  # https://paperswithcode.com/method/weight-tying

        # init all weights
        self.apply(self._init_weights)

    def _init_weights(self, module):
```

```
        if isinstance(module, (nn.Linear, nn.Embedding)):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx):
        device = idx.device
        b, t = idx.size()
        pos = torch.arange(0, t, dtype=torch.long, device=device)  # shape (t)

        # forward the GPT model itself
        # implement the transformer forward-pass.  As a guide, see the Main␣
 ↪Forward Pass above.
        # YOUR CODE HERE
        tok_emb = self.transformer.token_embedding(idx)
        pos_emb = self.transformer.position_embedding(pos)
        x = tok_emb + pos_emb
        for block in self.transformer.blocks:
            x = block(x)
        x = self.transformer.ln_f(x)

        logits = self.lm_head(x)

        # return tuple to mimic RNNs trained previously
        return logits, None

    @torch.no_grad()
    def generate_text(self, dataset, max_new_tokens=100):
        self.eval()
        # Start with a random token
        x = torch.randint(0, len(dataset.vocab), (1, 1), device=DEVICE)
        # add the standard autoregressive text generation loop here,
        # as adapted to the transformer.
        # HINT: consult 4. Text Generation above.

        for _ in range(max_new_tokens):
            logits, _ = self(x)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)

            x_next = torch.multinomial(input=probs, num_samples=1)
            x = torch.cat((x, x_next), dim=1)

        text = "".join([dataset.itos[i.item()] for i in x[0]])

        return text


class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.Sequential(
            nn.Linear(config.n_embd, 4 * config.n_embd),
            nn.GELU(),
            nn.Linear(4 * config.n_embd, config.n_embd),
        )

    def forward(self, x):
        # each block contains both Attention and MLP, in ResNet-style formation
        # Implement that forward pass here.
        # HINT: use the description of 2, Block Forward Pass, above as a guide.
        # YOUR CODE HERE
        x = self.attn(self.ln_1(x)) + x
        x = self.mlp(self.ln_2(x)) + x
        return x


class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=False)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=False)
        self.config = config
        # causal mask to ensure that attention is only applied to the left in␣
 ↪the input sequence
        self.register_buffer(
            "bias",
            torch.tril(torch.ones(config.block_size, config.block_size)).view(
                1, 1, config.block_size, config.block_size
            ),
        )

    def forward(self, x):
        B, T, C = (
            x.size()
        )  # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head␣
 ↪forward to be the batch dim
        q, k, v = self.c_attn(x).split(self.config.n_embd, dim=2)
        k = k.view(B, T, self.config.n_head, C // self.config.n_head).transpose(
            1, 2
        )  # (B, nh, T, hs)
        q = q.view(B, T, self.config.n_head, C // self.config.n_head).transpose(
            1, 2
        )  # (B, nh, T, hs)
        v = v.view(B, T, self.config.n_head, C // self.config.n_head).transpose(
            1, 2
        )  # (B, nh, T, hs)

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T)␣
 ↪-> (B, nh, T, T)
        if self.config.flash:
            # efficient attention using Flash Attention CUDA kernels
            y = F.scaled_dot_product_attention(
                q, k, v, attn_mask=None, dropout_p=0, is_causal=True
            )
        else:
            # manual implementation of attention
            # implement attention.
            # 1. Use k and q to compute the all-pairs attention weighting
            att = (q @ k.transpose(-2, -1)) * (
                1.0 / math.sqrt(k.size(-1))
            )  # YOUR CODE HERE
            # 2. Modify the attention weighting for autoregressive "causal"␣
 ↪visibility
            att = att.masked_fill(
                self.bias[:, :, :T, :T] == 0, float("-inf")
            )  # YOUR CODE HERE
            # 3. Apply softmax
            att = F.softmax(att, dim=-1)  # YOUR CODE HERE
            # 4. Use attention to gather and distribute the v data
            y = att @ v  # YOUR CODE HERE

        # 5. Reshape the data to re-assemble all head outputs for the hidden␣
 ↪state
        y = y.transpose(1, 2).contiguous().view(B, T, C)  # YOUR CODE HERE

        # output projection
        y = self.c_proj(y)

        return y


@dataclass
class TransformerConfig:
    vocab_size: int = len(dataset.vocab)
    # Here, we're choosing an embedding size to roughly match the GRUs␣
↪parameters
    n_embd: int = 128
    block_size: int = 100
    n_layer: int = 1
    n_head: int = 4
    flash: bool = True


# Check that your attention implementation is correct by comparing your␣
↪CausalSelfAttention to the PyTorch implementation
flash_transformer = Transformer(TransformerConfig(flash=True))
naive_transformer = Transformer(TransformerConfig(flash=False))

# Match weights between the two models
naive_transformer.load_state_dict(flash_transformer.state_dict())

random_input = torch.randint(0, len(dataset.vocab), (2, 8))
flash_output, _ = flash_transformer.forward(random_input)
naive_output, _ = naive_transformer.forward(random_input)

assert torch.allclose(
    flash_output, naive_output, atol=1e-6
), f"differences: {flash_output - naive_output}"
```

Now let's analyze the efficiency of our Transformer. With: - Model dimension D - Number of heads H - Head dimension J = D/H - Sequence length T - Number of layers L

1. How many FLOPs are needed to generate T tokens? (You can ignore small calculations like biases and assume that there's a single batch. You can also assume that an AxB matrix and a BxC matrix take 2ABC FLOPs to be multiplied together. Don't worry about the final projection to the vocabulary dimension, or about LayerNorm.)

   YOUR ANSWER: For a single layer, the MLP is: *($16TD^2$)* FLOPs.

   YOUR ANSWER: For a single layer, computing the K, Q, and V togther is: *($6JTD$)* FLOPs for each head and *($6TD^2$)* FLOPS for all heads together.

   YOUR ANSWER: For a single layer, computing the attention matrix is: *($2T^2J$)* FLOPs for each head and *($2T^2D$)* FLOPS for all heads together.

   YOUR ANSWER: For a single layer, processing all the values through attention is: *($2T^2J$)* FLOPs for each head and *($2T^2D$)* FLOPS for all heads together.

   YOUR ANSWER: After concatentating all head results, the final projection is *($2TD^2$)* FLOPS

   YOUR ANSWER: Therefore the total flops for a single layer of attention is *($8TD^2 + 4T^2D$)* FLOPS

   YOUR ANSWER: Adding up all layers the cost to generate T tokens is *($24LTD^2 +$

$4LT^2D$*)* FLOPS

2. How many FLOPs are needed to train on a sequence of T tokens?

   YOUR ANSWER: A single forward pass requires *(*$24LTD^2 + 4LT^2D$*)* FLOPS and the backward pass requires *(*$48LTD^2 + 8LT^2D$*)* FLOPS.

3. How many sequential matrix multiplications are needed to generate T tokens?

   YOUR ANSWER: For a single layer the MLP requires ***2*** sequential matrix multiplications.

   YOUR ANSWER: For a single layer, calculating the keys, queries and values can be done in parallel, but the attention matrix must be computed only after generating the keys, queries and values, and processing the values and doing the final projection must also happen after one another therefore, the attention sub-layer requires *4* sequential matrix multiplications.

   YOUR ANSWER: There the transformer requires ***6*** total sequential matrix multiplications per layer.

   YOUR ANSWER: Considering all layers all together, a foward pass requires ***6L*** sequential matrix multiplications.

4. How many sequential matrix multiplications are needed to train on T tokens, and how does this compare to the RNN?

   YOUR ANSWER: Considering all layers all together, a backward pass in a transformer requires ***6L*** sequential matrix multiplications.

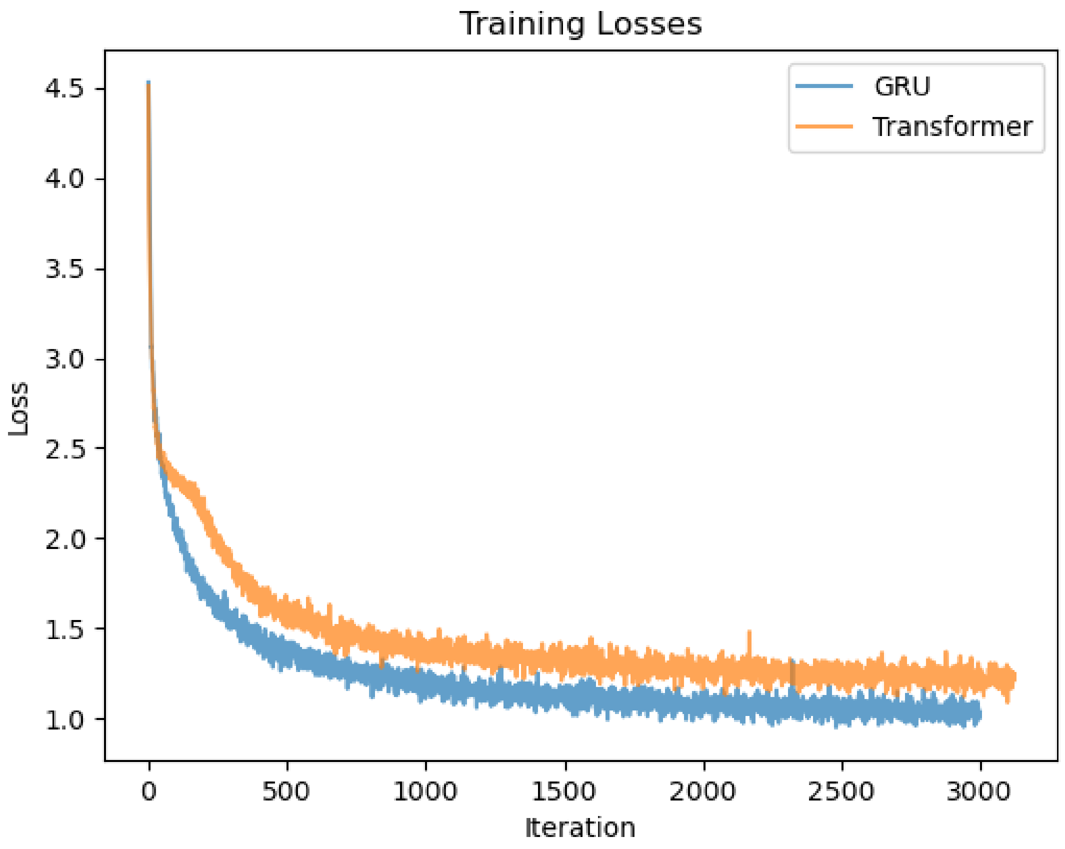
   YOUR ANSWER: This differs from the cost of training an RNN because ***computation for RNN training is mostly sequential, its training cost(~2TL) is correlated to the sequence length. Thus transformer is more cost-efficient to train if T»>L.***

```
[31]: # Now let's see how the Transformer performs on our dataset.

      seq_length = 100
      batch_size = 32
      lr = 1e-3

      dataset = CharLMDataset(full_text, seq_length=seq_length)
      dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
       ↪drop_last=True)

      transformer_model = Transformer(TransformerConfig()).to(DEVICE)

      print(f"Parameters: {sum(p.numel() for p in transformer_model.parameters())}")

      transformer_losses = train_language_model(
          transformer_model, dataloader, lr=lr, max_iters=15_000, is_rnn=False
      )
```

```python
print(len(transformer_losses), len(gru_losses), len(simple_rnn_losses))

# Generate sample text from Transformer
sample_text = transformer_model.generate_text(dataset)
print(f"\nSample text from Transformer: {sample_text}")
print("-" * 80)

# Plot the loss curves of all three models
plot_losses(
    {
        "GRU": gru_losses,
        "Transformer": transformer_losses,
    }
)
```

```
Parameters: 222592
Step 100, Loss: 2.3365, Time: 3.45s
Step 200, Loss: 2.1479, Time: 3.42s
Step 300, Loss: 1.8737, Time: 3.50s
Step 400, Loss: 1.6359, Time: 3.45s
Step 500, Loss: 1.5738, Time: 3.45s
Step 600, Loss: 1.4952, Time: 3.52s
Step 700, Loss: 1.4954, Time: 3.47s
Step 800, Loss: 1.4066, Time: 3.50s
Step 900, Loss: 1.3939, Time: 3.53s
Step 1000, Loss: 1.3766, Time: 3.50s
Step 1100, Loss: 1.3691, Time: 3.53s
Step 1200, Loss: 1.3800, Time: 3.54s
Step 1300, Loss: 1.3217, Time: 3.53s
Step 1400, Loss: 1.2909, Time: 3.58s
Step 1500, Loss: 1.3344, Time: 3.59s
Step 1600, Loss: 1.3836, Time: 3.55s
Step 1700, Loss: 1.3262, Time: 3.54s
Step 1800, Loss: 1.2609, Time: 3.51s
Step 1900, Loss: 1.1607, Time: 3.50s
Step 2000, Loss: 1.2429, Time: 3.51s
Step 2100, Loss: 1.2328, Time: 3.49s
Step 2200, Loss: 1.3039, Time: 3.53s
Step 2300, Loss: 1.2127, Time: 3.62s
Step 2400, Loss: 1.2524, Time: 3.60s
Step 2500, Loss: 1.2337, Time: 3.58s
Step 2600, Loss: 1.2200, Time: 3.54s
Step 2700, Loss: 1.2749, Time: 3.53s
Step 2800, Loss: 1.2098, Time: 3.51s
Step 2900, Loss: 1.2121, Time: 3.57s
Step 3000, Loss: 1.2130, Time: 3.49s
Step 3100, Loss: 1.2098, Time: 3.49s
```

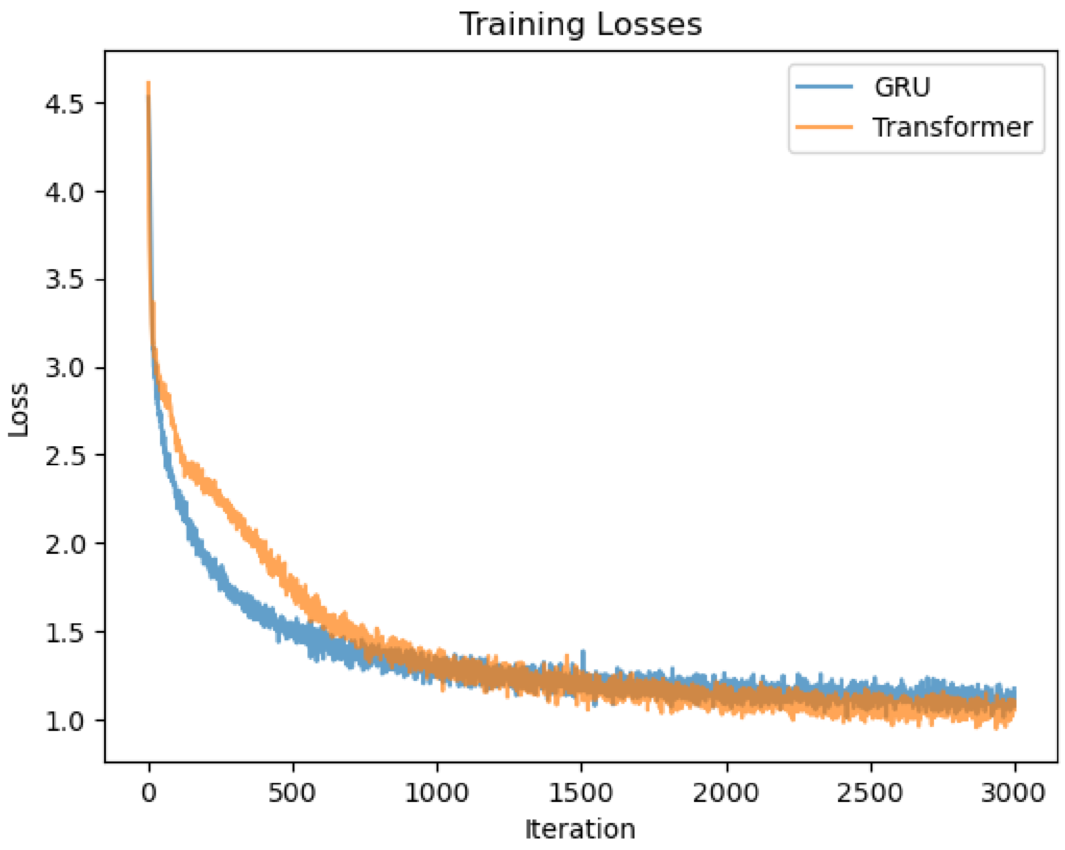
```
3124 3000 2000

Sample text from Transformer: Gift, whyter!"

But worried what it go with hill.

Fluffyed nent the room the toractor. She seon't
--------------------------------------------------------------------------------
```



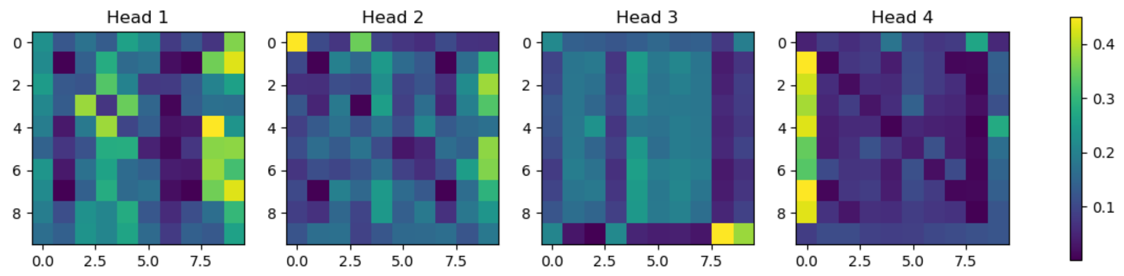
[32]:
```python
# Very little improvement! In a course on Deep Learning, though, it might make
 ↪sense to try some more layers. Here,
# we compare the GRU and Transformer, roughly parameter-matched, at 8 layers
 ↪each.

max_iters = 3_000

gru_model = GRULM(
    vocab_size=len(dataset.vocab),
    embed_size=32,
    hidden_size=192,
).to(DEVICE)
gru_losses = train_language_model(gru_model, dataloader, lr=lr,␣
  ↪max_iters=max_iters)
print(f"GRU parameters: {sum(p.numel() for p in gru_model.parameters()):.1e}")

sample_text = gru_model.generate_text(dataset)
print(f"\nSample text from GRU: {sample_text}")
print("-" * 80)

cfg = TransformerConfig(
    vocab_size=len(dataset.vocab),
    n_embd=128,  # Here, we're choosing an embedding size to roughly match the␣
  ↪GRUs parameters
    block_size=seq_length,
    n_layer=8,
    n_head=4,
    flash=True,
)

transformer_model = Transformer(cfg).to(DEVICE)
print(
    f"Transformer parameters: {sum(p.numel() for p in transformer_model.
  ↪parameters()):.1e}"
)

transformer_losses = train_language_model(
    transformer_model, dataloader, lr=lr, max_iters=max_iters, is_rnn=False
)

sample_text = transformer_model.generate_text(dataset)
print(f"\nSample text from Transformer: {sample_text}")
print("-" * 80)

plot_losses(
    {
        "GRU": gru_losses,
        "Transformer": transformer_losses,
    }
)

# There are two things to notice here. First, the GRU really struggles as you␣
  ↪add more layers. The Transformer's
# residual connections, layer normalization, and GELU activations help it␣
  ↪successfully propagate gradients. Second,
# the Transformer, despite having the same number of parameters, takes much,␣
  ↪much less time to train. Flash attention
```

```
# (Dao et al., 2022)[https://arxiv.org/abs/2205.14135] is a big part of this, 
 ↪but the Transformer is also just more
# efficient to train generally, especially with large batch sizes and longer 
 ↪sequences. Try disabling the flash
# attention to see how much slower the default implementation is.
```

```
Step 100, Loss: 2.2465, Time: 36.85s
Step 200, Loss: 1.8792, Time: 37.07s
Step 300, Loss: 1.7241, Time: 38.90s
Step 400, Loss: 1.6180, Time: 40.07s
Step 500, Loss: 1.4940, Time: 39.93s
Step 600, Loss: 1.4199, Time: 39.83s
Step 700, Loss: 1.3469, Time: 40.15s
Step 800, Loss: 1.4220, Time: 40.30s
Step 900, Loss: 1.3087, Time: 39.91s
Step 1000, Loss: 1.2408, Time: 39.61s
Step 1100, Loss: 1.2451, Time: 39.91s
Step 1200, Loss: 1.2759, Time: 39.82s
Step 1300, Loss: 1.2696, Time: 39.45s
Step 1400, Loss: 1.2762, Time: 39.54s
Step 1500, Loss: 1.1583, Time: 40.82s
Step 1600, Loss: 1.1948, Time: 39.99s
Step 1700, Loss: 1.2679, Time: 39.63s
Step 1800, Loss: 1.1398, Time: 40.09s
Step 1900, Loss: 1.1692, Time: 40.48s
Step 2000, Loss: 1.1660, Time: 39.69s
Step 2100, Loss: 1.1329, Time: 39.35s
Step 2200, Loss: 1.2466, Time: 39.43s
Step 2300, Loss: 1.1281, Time: 39.75s
Step 2400, Loss: 1.0897, Time: 39.28s
Step 2500, Loss: 1.1413, Time: 39.41s
Step 2600, Loss: 1.1321, Time: 36.92s
Step 2700, Loss: 1.1314, Time: 36.54s
Step 2800, Loss: 1.0746, Time: 36.36s
Step 2900, Loss: 1.0891, Time: 36.45s
Step 3000, Loss: 1.1730, Time: 36.61s
GRU parameters: 1.5e+05

Sample text from GRU: I have to see the bird was so happy to help her mom and
said, "I want to play with her mommy and dad
--------------------------------------------------------------------------------
Transformer parameters: 1.6e+06
Step 100, Loss: 2.5222, Time: 22.68s
Step 200, Loss: 2.3285, Time: 22.66s
Step 300, Loss: 2.1312, Time: 22.66s
Step 400, Loss: 1.9644, Time: 22.72s
Step 500, Loss: 1.7458, Time: 22.63s
Step 600, Loss: 1.5833, Time: 22.54s
Step 700, Loss: 1.5091, Time: 22.54s
Step 800, Loss: 1.4468, Time: 22.39s
Step 900, Loss: 1.3019, Time: 22.51s
Step 1000, Loss: 1.3432, Time: 22.52s
Step 1100, Loss: 1.3171, Time: 22.53s
Step 1200, Loss: 1.2442, Time: 22.51s
Step 1300, Loss: 1.2875, Time: 22.47s
Step 1400, Loss: 1.1959, Time: 22.55s
Step 1500, Loss: 1.1686, Time: 22.52s
Step 1600, Loss: 1.1368, Time: 22.53s
Step 1700, Loss: 1.0912, Time: 22.57s
Step 1800, Loss: 1.0949, Time: 22.51s
Step 1900, Loss: 1.1316, Time: 22.65s
Step 2000, Loss: 1.1089, Time: 22.49s
Step 2100, Loss: 1.0536, Time: 22.45s
Step 2200, Loss: 1.1039, Time: 22.51s
Step 2300, Loss: 1.1504, Time: 22.57s
Step 2400, Loss: 1.0683, Time: 22.55s
Step 2500, Loss: 1.0831, Time: 22.55s
Step 2600, Loss: 0.9918, Time: 22.45s
Step 2700, Loss: 1.0207, Time: 22.95s
Step 2800, Loss: 1.0990, Time: 22.62s
Step 2900, Loss: 0.9972, Time: 22.40s
Step 3000, Loss: 1.0518, Time: 22.33s

Sample text from Transformer: Figger her, smiled and their mother as thank to
play soon. They want to play with their tagether. Tim
--------------------------------------------------------------------------------
```



[33]:
```
# Transformers are ubiquitous. It would be nice to understand them better. One␣
↪way people have tried to this is to look
# at their attention patterns. Let's try to do this for our models.


# Add a helper function to visualize attention patterns
def plot_attention_patterns(model, text, dataset):
    """Visualize the attention patterns for a given input text"""
    # Convert text to tokens
    tokens = [dataset.stoi[c] for c in text]
    x = torch.tensor(tokens).unsqueeze(0).to(DEVICE)

    # Get attention weights
    with torch.no_grad():
        B, T = x.size()
        block = model.transformer.blocks[0]
        q, k, v = block.attn.c_attn(
            block.ln_1(model.transformer.token_embedding(x))
        ).split(model.config.n_embd, dim=2)
        k = k.view(B, T, model.config.n_head, -1).transpose(1, 2)
        q = q.view(B, T, model.config.n_head, -1).transpose(1, 2)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = F.softmax(att, dim=-1)

    # Plot heatmap for each attention head
    fig, axes = plt.subplots(1, model.config.n_head, figsize=(15, 3))
    for h in range(model.config.n_head):
        im = axes[h].imshow(att[0, h].cpu())
        axes[h].set_title(f"Head {h + 1}")
    plt.colorbar(im, ax=axes)
    plt.show()


plot_attention_patterns(transformer_model, "Lily said,", dataset)

# Experiment with some more input examples. Do you notice anything about how␣
  ↪the patterns change? Do particular heads
# behave differently from each other? How?
```


Head 1
Head 2
Head 3
Head 4