Building a Simple LLM - PyTorch Notes & Code

PyTorch Optimizer Training Loop

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    if iter % eval_iters == 0:
        losses = estimate_loss()
        print(f'step: {iter}, train loss: {losses["train"]}, val loss: {losses["val"]}')

    # Sample a batch of training data
    xb, yb = get_batch('train')

    # Forward pass and compute loss
    logits, loss = model.forward(xb, yb)  # fixed typo 'foward' -> 'forward'

    # Clear previous gradients
    optimizer.zero_grad(set_to_none=True)

    # Backward pass
    loss.backward()

    # Update weights
    optimizer.step()
```

Notes:
- estimate_loss() returns dict with keys 'train' and 'val' losses.
- zero_grad(set_to_none=True) improves memory efficiency.
- Use model.train() before training loop for dropout/batchnorm if any.
- Use model.eval() and torch.no_grad() for validation.

Building a Simple LLM: Documentation

This documentation outlines key concepts and code foundations for building a basic character-level LLM using PyTorch.

1. PyTorch vs TensorFlow
- PyTorch: Dynamic, Pythonic, flexible for research, popular in academia.
- TensorFlow: More mature for deployment, used in industry.

This project uses PyTorch.

Building a Simple LLM - PyTorch Notes & Code

2. Setup

```
import torch
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Using:", device)
```

Note: Ryzen CPUs don't support CUDA; CUDA is Nvidia-specific.

3. Char-Level Language Modeling

Steps:

1. Convert text to chars
2. Create vocab of unique chars
3. Map each char to integer (stoi) and back (itos)
4. Use embeddings to map chars to vectors

Sample:

```
text = "hello world"
vocab = sorted(list(set(text)))
vocab_size = len(vocab)
stoi = { ch:i for i,ch in enumerate(vocab) }
itos = { i:ch for i,ch in enumerate(vocab) }
```

4. What is a Neural Network?

A neural net is a team of math workers solving problems by passing info layer to layer:

- Input layer takes data
- Hidden layers learn patterns
- Output layer gives predictions

5. Gradient Descent

How the net learns:

1. Make prediction
2. Calculate error (loss)
3. Use gradients to improve
4. Update weights to reduce loss

Like walking downhill to lowest error.

6. torch.nn and nn.Module

- torch.nn builds neural nets
- nn.Module is base class for models

Sample:

Building a Simple LLM - PyTorch Notes & Code

```
import torch.nn as nn
class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(10, 5)
        self.relu = nn.ReLU()
    def forward(self, x):
        return self.relu(self.linear(x))
```

7. Character Embedding

Sample:

```
embedding_dim = 4
embedding = nn.Embedding(vocab_size, embedding_dim)
input_str = "hello"
input_ids = torch.tensor([stoi[c] for c in input_str])
embedded = embedding(input_ids)
```

Maps each char index to dense vector.

8. Bigram Model with Dot Embedding

Bigram learns probability of one char after another:

```
logits = emb @ embedding.weight.T  # dot product for next char score
```

9. Important torch functions for tensor creation:

torch.tensor([...]), torch.zeros(...), torch.ones(...), torch.empty(...),
torch.arange(...), torch.linspace(...), torch.logspace(...),
torch.eye(...), torch.empty_like(...), torch.randint(...)

10. Desmos (optional tool)

https://www.desmos.com/calculator

Use to visualize equations, gradients, loss curves.

11. Is CUDA necessary?

- No, CPU training works for small LLMs.
- CUDA speeds training but not required.

Final thoughts:

You now have char-level data prep, embeddings, gradient descent understanding, and a base LLM in PyTorch.

Ready to build, train, and expand your model!

Bigram Language Model - Summary Document

Building a Simple LLM - PyTorch Notes & Code

Model Definition:

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

class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, index, targets):
        logits = self.token_embedding_table(index)  # (B, T, C)
        B, T, C = logits.shape
        logits = logits.view(B * T, C)              # Flatten logits
        targets = targets.view(B * T)               # Flatten targets
        loss = F.cross_entropy(logits, targets)     # Compute loss
        return logits, loss

    def generate(self, index, max_new_tokens):
        for _ in range(max_new_tokens):
            logits, _ = self.forward(index, index)  # Forward pass (targets unused in generation)
            logits = logits[:, -1, :]               # Last timestep
            probs = F.softmax(logits, dim=-1)       # Convert logits to probabilities
            index_next = torch.multinomial(probs, 1)  # Sample next token
            index = torch.cat((index, index_next), dim=1) # Append sampled token
        return index
```

Usage:

```
# Assuming vocab_size and device are defined
model = BigramLanguageModel(vocab_size)
m = model.to(device)

context = torch.zeros((1, 1), dtype=torch.long, device=device)
output_tensor = m.generate(context, max_new_tokens=500)
generated_text = decode(output_tensor[0].tolist())
print(generated_text)
```

Common Issues & Fixes:

Building a Simple LLM - PyTorch Notes & Code

- Typo: '==' instead of '=' for embedding assignment
- Missing targets during generation: pass targets as index

Output:
- Output is garbage before training
- After training, outputs form valid words/sentences

Next Steps:
- Prepare dataset
- Train model with optimizer and loop
- Evaluate improvements in generated text