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


block_size = 8
batch_size = 4
data_text = "hello world"
vocab = sorted(list(set(data_text)))
vocab_size = len(vocab)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
epochs = 1000
embed_size = 32



stoi = {ch: i for i, ch in enumerate(vocab)}
itos = {i: ch for ch, i in stoi.items()}
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])


data = torch.tensor(encode(data_text), dtype=torch.long).to(device)


def get_batch():
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i + block_size] for i in ix])
    y = torch.stack([data[i + 1:i + block_size + 1] for i in ix])
    return x, y


class TinyLLM(nn.Module):
    def __init__(self):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.fc = nn.Linear(embed_size * block_size, vocab_size)

    def forward(self, x):
        x = self.embed(x)               # (B, T, C)
        x = x.view(x.size(0), -1)       # flatten: (B, T*C)
        return self.fc(x)               # (B, vocab_size)



# --- Model Setup ---
model = TinyLLM().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)



# --- Training Loop ---
for epoch in range(epochs):
    x, y = get_batch()
    logits = model(x)
    loss = F.cross_entropy(logits, y[:, -1])  # Only last token predicted
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

```
Epoch 0, Loss: 1.9726
Epoch 100, Loss: 0.0096
Epoch 200, Loss: 0.0043
Epoch 300, Loss: 0.0025
Epoch 400, Loss: 0.0018
Epoch 500, Loss: 0.0013
Epoch 600, Loss: 0.0010
Epoch 700, Loss: 0.0008
Epoch 800, Loss: 0.0006
Epoch 900, Loss: 0.0005
```

```
# --- Generation Function ---
def generate(model, start='h', max_new_tokens=20):
    model.eval()
    idx = torch.tensor([encode(start)], dtype=torch.long).to(device)

    for _ in range(max_new_tokens):
        idx_cond = idx[:, -block_size:]
```

```
            # Padding if needed
            if idx_cond.shape[1] < block_size:
                padding = torch.zeros(1, block_size - idx_cond.shape[1], dtype=torch.long).to(device)
                idx_cond = torch.cat([padding, idx_cond], dim=1)

            logits = model(idx_cond)
            probs = F.softmax(logits, dim=-1)
            next_id = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, next_id), dim=1)

        return decode(idx[0].tolist())

# --- Generate text after training ---
print("Generated text:", generate(model))
```

```
Generated text: hhed  ddrldwlh  wdrld
```

Start coding or generate with AI.