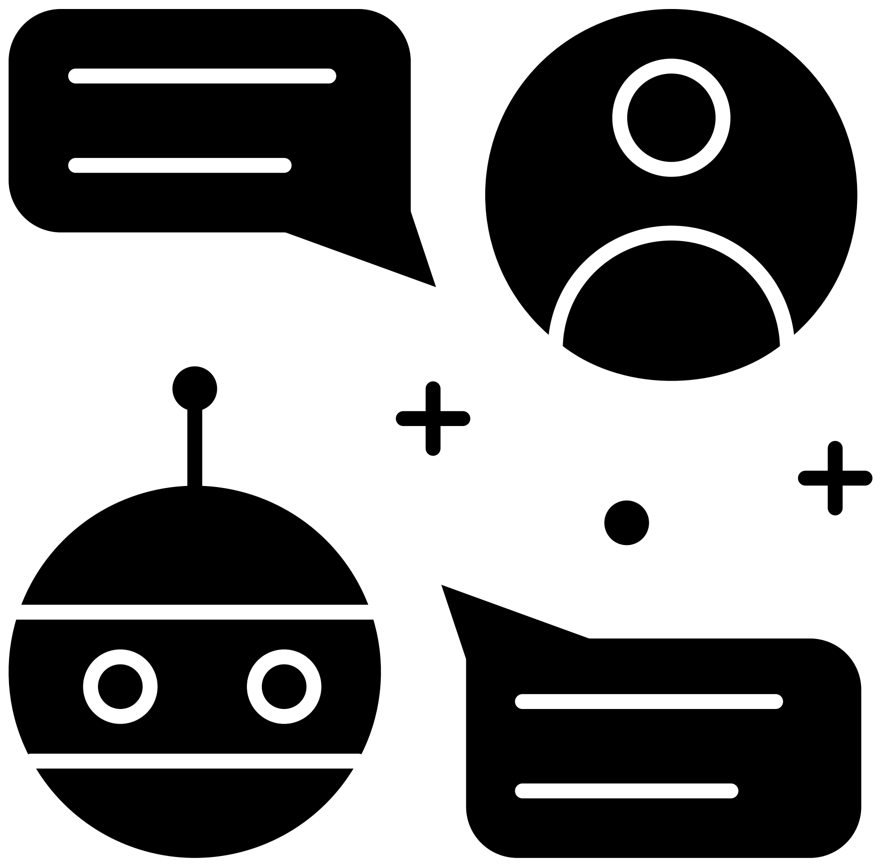
Course: Build a Small Language Model from scratch

Module 5: Training, Evaluation. Generation



Instructor: Nusrat Jahan Lia

Training Process

GPU vs. CPU Training

- Multi-core CPUs are better at multitasking than single-core CPUs, but they still process data sequentially.

- GPUs handle data differently, through a process known as parallel computing. Instead of processing tasks sequentially, GPUs break down problems into component parts and use their multitude of cores to work on different parts of a problem concurrently.

Memory Management

- Large models + big batches can cause CUDA Out of Memory errors.
- Reduce batch size or sequence length (BLOCK_SIZE) to fit into GPU memory.

Float16 vs. Float32 Precision

- torch.float32 → default, stable, but uses more memory.
- torch.float16 → half memory, faster, but needs GPU with mixed precision support.

Batch Loading

The get_batch() function is crucial for efficient data loading during training:

1. **Random Sampling:** Selects random sequences from the dataset to prevent overfitting to data order
2. **Input-Target Pair Creation:** Creates (x, y) pairs where y is x shifted by one position
3. **Memory Mapping:** Efficiently loads large datasets without loading everything into RAM

```
def get_batch(split:str, block:int=BLOCK_SIZE, batch:int=BATCH_SIZE, device=torch.device("cuda" if
torch.cuda.is_available() else "cpu")):

   data = np.memmap(Path(DRIVE_DIR)/f"{split}.bin", dtype=BIN_DTYPE_np, mode="r")
    ix = torch.randint(len(data)-block, (batch,))
    x = torch.stack([torch.from_numpy(data[i:i+block].astype(np.int64)) for i in ix])
    y = torch.stack([torch.from_numpy(data[i+1:i+1+block].astype(np.int64)) for i in ix])
    if device.type == "cuda":
        x = x.pin_memory().to(device, non_blocking=True)
        y = y.pin_memory().to(device, non_blocking=True)
    else:
        x, y = x.to(device), y.to(device)
    return x, y
```

Memory Mapping Benefits:

- Allows working with datasets larger than available RAM
- OS handles caching and memory management
- Faster than traditional file I/O for random access

Optimization Setup

AdamW Optimizer Configuration:

```
opt = torch.optim.AdamW(model.parameters(), lr=LR, betas=(0.9,0.95), weight_decay=0.1)
```

Anyone knows adam?

"dimension"?

- l. 336: "Both architectures are optimized with Adam". Who/what is "Adam"? I think this is a very serious typo that the author should have removed from the submission.

9:04 PM · 7/24/25 · **242K** Views

159 374 2.8K 288

The AdamW optimizer is a variant of the Adam optimizer that improves model generalization by decoupling weight decay from the gradient update in the optimization process.

For step-by-step math, Let:

- θt= parameters at step t
- $g\_t=\nabla\_\theta \times L\_t$ = gradient of the loss with respect to the parameters θ, at step t
- m_t = first moment estimate (mean of gradients)
- v_t = second moment estimate (mean of squared gradients)
- $\beta 1,\beta 2$ = exponential decay rates for the moment estimates
- η = learning rate
- λ= weight decay coefficient

Step 1 — Update biased first moment estimate

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

Here, β1=0.9 means 90% of the old momentum is kept, and 10% comes from the new gradient.

Step 2 — Update biased second moment estimate

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Here, β2=0.95 means we smooth squared gradients over recent history : this helps stabilize updates when gradient magnitudes vary.

Step 3 — Bias correction

Because m_t and v_t start at zero, early steps are biased towards zero. We fix that:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad , \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Step 4 — Apply update with decoupled weight decay

In AdamW, weight decay is applied directly to the parameters after the gradient update. The update rule is:

$$\theta t \leftarrow \theta t - \eta \left[\frac{m_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta t \right]$$

Here, η is the learning rate, λ is the weight decay factor, and θt represents the parameters. This decoupled weight decay term λθt ensures that regularization is applied independently of the gradient update, **which is the key difference from Adam.**

Parameter Breakdown:

- Learning Rate (lr): Controls step size during optimization (typically 3e-4 to 6e-4 for GPT models)
- Beta1 (0.9): Momentum term for first-order moments (gradient)
- Beta2 (0.95): Momentum term for second-order moments (gradient squared)
- Weight Decay (0.1): L2 regularization to prevent overfitting

Why AdamW over Adam:

- **Decoupled Weight Decay:** AdamW applies weight decay directly to parameters, not through gradients
- **Better Generalization:** Often leads to better test performance
- **Stable Training:** More robust optimization for large language models
- **Industry Standard:** Used in GPT-3, BERT, and other successful models

Learning Rate Scheduling

Warmup Phase

```
if it < WARMUP_ITERS:
    return LR * it / WARMUP_ITERS
```

Purpose of Warmup:

- Gradient Stability: Large models can have unstable gradients in early training
- Parameter Initialization: Gives model time to adjust from random initialization
- Loss Landscape: Smooths optimization trajectory in early phases

Typical Warmup Duration:

- Small models: 100-500 iterations
- Large models: 2000-10000 iterations
- Rule of thumb: ~1-5% of total training steps

Mathematical Explanation: During warmup, learning rate increases linearly from 0 to maximum value: current_lr = max_lr × (current_step / warmup_steps)

Cosine Annealing

```
pct = (it - WARMUP_ITERS)/max(1, MAX_ITERS-WARMUP_ITERS)
    return 0.1*LR + 0.9*LR*0.5*(1+math.cos(math.pi*pct))
```

Smoothly decays LR from max → min using a cosine curve.

Final LR is 10% of the maximum.

Benefits of Cosine Annealing:

- Smooth Decay: Gradual reduction prevents training instability
- Fine-tuning Phase: Low learning rates at end allow precise optimization
- Exploration vs Exploitation: High LR for exploration, low LR for exploitation

Training Loop Implementation

Forward Pass → Loss Computation → Backward Pass:

```
x, y = get_batch("train", BLOCK_SIZE, BATCH_SIZE, Device)
logits, loss = model(x, y)  # Forward pass
opt.zero_grad(set_to_none=True)  # Clear gradients
loss.backward()  # Backward pass
torch.nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)  # Clip gradients
opt.step()  # Update parameters
```

Step-by-Step Breakdown:

1. Data Loading: Get random batch of input-target pairs
2. Forward Pass: Compute model predictions and loss
3. Gradient Clearing: Reset gradients from previous iteration
4. Backward Pass: Compute gradients via backpropagation
5. Gradient Clipping: Prevent exploding gradients
6. Parameter Update: Apply optimizer step

Gradient Clipping

```
torch.nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)
```

Why Gradient Clipping is Essential:

- Exploding Gradients: Large gradients can destabilize training
- RNN/Transformer Issue: Long sequences can compound gradient problems
- Training Stability: Ensures consistent optimization steps

Clipping Methods:

- Norm-based: Scales gradients if their norm exceeds threshold
- Value-based: Clips individual gradient values
- Global vs Layer-wise: Apply globally or per-layer

Optimal Clip Values:

- Transformers: 1.0 (standard)
- RNNs: 0.25-5.0
- Experiment with different values if training is unstable

Evaluation Loop Implementation

```
@torch.no_grad()  # Critical for memory efficiency
def eval_loss():
    model.eval()  # Disable dropout, batch norm updates
    losses = torch.zeros(EVAL_ITERS)

    for k in range(EVAL_ITERS):
        x, y = get_batch(split, BLOCK_SIZE, BATCH_SIZE, Device)
        _, loss = model(x, y)
        losses[k] = loss.item()

    return losses.mean().item()
```

Cross-entropy loss measures the difference between predicted probability distribution and actual token distribution:

$\text{Loss} = -\Sigma(\text{actual} \times \log(\text{predicted}))$

Key Design Decisions:

- @torch.no_grad(): Prevents gradient computation, saves memory
- Multiple Iterations: Reduces variance in loss estimates
- model.eval(): Ensures consistent evaluation conditions
- Average Reporting: Single number easier to track than distributions

Healthy Training Patterns:

- Both losses decrease initially
- Training loss continues decreasing
- Validation loss plateaus or slightly increases
- Gap between them indicates generalization ability

Warning Signs:

- Severe Overfitting: Large gap between train/val loss
- Underfitting: Both losses plateau at high values
- Unstable Training: Erratic loss trajectories
- Data Leakage: Validation loss lower than training loss

Model Checkpointing

```
ckpt = {
    "model": model.state_dict(),    # All model parameters
    "cfg": cfg.__dict__,           # Model architecture config
    "iter": it,                    # Training iteration number
    "meta": meta,                  # Dataset metadata
}
```

Optional But Useful:

- Optimizer state (for resuming training)
- Learning rate schedule state
- Random number generator states
- Training and validation loss history

Text Generation

1. Start with prompt tokens
2. Predict probability distribution for next token
3. Sample/select next token from distribution
4. Add token to context
5. Repeat until stopping criteria met

Sampling Strategies

- **Greedy** → Always pick highest probability token (can be repetitive).
- **Sampling** → Adds randomness by sampling from probability distribution.
- **Temperature** → Controls randomness:
    - **temp = 1.0** → normal randomness.
    - **temp < 1.0** → more deterministic.
    - **temp > 1.0** → more diverse but less coherent.

Generation Function Architecture

```
@torch.no_grad()
def generate(prompt: str, max_new_tokens: int = 50):
    model.eval()  # Disable training-specific behaviors

    # Tokenize input prompt
    ids = Tokenizer(prompt, return_tensors="pt")["input_ids"].to(Device)
    out = ids

    # Generate tokens iteratively
    for _ in range(max_new_tokens):
        # Manage context window
        idx_cond = out[:, -BLOCK_SIZE:]

        # Get predictions
        logits, _ = model(idx_cond)

        # Sample next token
        next_id = torch.multinomial(
            torch.softmax(logits[:, -1, :] / 1.0, dim=-1), 1
        )

        # Append to sequence
        out = torch.cat([out, next_id], dim=1)

    return Tokenizer.decode(out[0].tolist(),skip_special_tokens=True)
```

Decoding Process: Token ID to Text Conversion, avoiding special tokens

We’ve done it !!

Find the full code here:

Bangla-Small-Language-Model