CSCE 689-609
Training LLMs

Jeff Huang
jeff@cse.tamu.edu
o2lab.github.io

babyGPT demo

Training data
(1.1M shakespeare text) → **Tokenizer** → a (long) sequence of token ids

babyGPT model

```
n_layer = 6
n_head = 6
n_embd = 384
vocab_size = 65
```

 # parameters: 10.65M

```
wte: token embedding weight
wpe: position encoding weight

LayerNorm
CausalSelfAttention
LayerNorm
MLP
```

Training sequence length (i.e. context window): 256
Batch (i.e. block size): a number of sequences: 8

Key Data Structures

```c
typedef struct {
    GPT2Config config;
    // the weights (parameters) of the model, and their sizes
    ParameterTensors params;
    size_t param_sizes[NUM_PARAMETER_TENSORS];
    float* params_memory;
    size_t num_parameters;
    // gradients of the weights
    ParameterTensors grads;
    float* grads_memory;
    // buffers for the AdamW optimizer
    float* m_memory;
    float* v_memory;
    // the activations of the model, and their sizes
    ActivationTensors acts;
    size_t act_sizes[NUM_ACTIVATION_TENSORS];
    float* acts_memory;
    size_t num_activations;
    // gradients of the activations
    ActivationTensors grads_acts;
    float* grads_acts_memory;
    // other run state configuration
    int batch_size; // the batch size (B) of current forward pass
    int seq_len; // the sequence length (T) of current forward pass
    int* inputs; // the input tokens for the current forward pass
    int* targets; // the target tokens for the current forward pass
    float mean_loss; // after a forward pass with targets, will be popu
} GPT2;
```

```c
typedef struct {
    float* wte; // (V, C)
    float* wpe; // (maxT, C)
    float* ln1w; // (L, C)
    float* ln1b; // (L, C)
    float* qkvw; // (L, 3*C, C)
    float* qkvb; // (L, 3*C)
    float* attprojw; // (L, C, C)
    float* attprojb; // (L, C)
    float* ln2w; // (L, C)
    float* ln2b; // (L, C)
    float* fcw; // (L, 4*C, C)
    float* fcb; // (L, 4*C)
    float* fcprojw; // (L, C, 4*C)
    float* fcprojb; // (L, C)
    float* lnfw; // (C)
    float* lnfb; // (C)
```

```c
#define NUM_ACTIVATION_TENSORS 23
typedef struct {
    float* encoded; // (B, T, C)
    float* ln1; // (L, B, T, C)
    float* ln1_mean; // (L, B, T)
    float* ln1_rstd; // (L, B, T)
    float* qkv; // (L, B, T, 3*C)
    float* atty; // (L, B, T, C)
    float* preatt; // (L, B, NH, T, T)
    float* att; // (L, B, NH, T, T)
    float* attproj; // (L, B, T, C)
    float* residual2; // (L, B, T, C)
    float* ln2; // (L, B, T, C)
    float* ln2_mean; // (L, B, T)
    float* ln2_rstd; // (L, B, T)
    float* fch; // (L, B, T, 4*C)
    float* fch_gelu; // (L, B, T, 4*C)
    float* fcproj; // (L, B, T, C)
    float* residual3; // (L, B, T, C)
    float* lnf; // (B, T, C)
    float* lnf_mean; // (B, T)
    float* lnf_rstd; // (B, T)
    float* logits; // (B, T, V)
    float* probs; // (B, T, V)
    float* losses; // (B, T)
} ActivationTensors;
```

Key Steps

```
// do a training step
clock_gettime(CLOCK_MONOTONIC, &start);
dataloader_next_batch(&train_loader);
gpt2_forward(&model, train_loader.inputs, train_loader.targets, B, T);
gpt2_zero_grad(&model);
gpt2_backward(&model);
gpt2_update(&model, 1e-4f, 0.9f, 0.999f, 1e-8f, 0.0f, step+1);
clock_gettime(CLOCK_MONOTONIC, &end);
double time_elapsed_s = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
printf("step %d: train loss %f (took %f ms)\n", step, model.mean_loss, time_elapsed_s * 1000);
```

gpt2_forward

```
// forward pass
ParameterTensors params = model->params; // for brevity
ActivationTensors acts = model->acts;
float* residual;
encoder_forward(acts.encoded, inputs, params.wte, params.wpe, B, T, C); //
for (int l = 0; l < L; l++) {

    residual = l == 0 ? acts.encoded : acts.residual3 + (l-1) * B * T * C;

    // get the pointers of the weights for this layer
    float* l_ln1w = params.ln1w + l * C;
```

```
    // now do the forward pass
    layernorm_forward(l_ln1, l_ln1_mean, l_ln1_rstd, residual, l_ln1w, l_ln1b, B, T, C);
    matmul_forward(l_qkv, l_ln1, l_qkvw, l_qkvb, B, T, C, 3*C);
    attention_forward(l_atty, l_preatt, l_att, l_qkv, B, T, C, NH);
    matmul_forward(l_attproj, l_atty, l_attprojw, l_attprojb, B, T, C, C);
    residual_forward(l_residual2, residual, l_attproj, B*T*C);
    layernorm_forward(l_ln2, l_ln2_mean, l_ln2_rstd, l_residual2, l_ln2w, l_ln2b, B, T, C);
    matmul_forward(l_fch, l_ln2, l_fcw, l_fcb, B, T, C, 4*C);
    gelu_forward(l_fch_gelu, l_fch, B*T*4*C);
    matmul_forward(l_fcproj, l_fch_gelu, l_fcprojw, l_fcprojb, B, T, 4*C, C);
    residual_forward(l_residual3, l_residual2, l_fcproj, B*T*C);
```

crossentropy_forward

```c
void crossentropy_forward(float* losses,
                          float* probs, int* targets,
                          int B, int T, int Vp) {
    // output: losses is (B,T) of the individual losses at each position
    // input: probs are (B,T,Vp) of the probabilities
    // input: targets is (B,T) of integers giving the correct index in logits
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            // loss = -log(probs[target])
            float* probs_bt = probs + b * T * Vp + t * Vp;
            int ix = targets[b * T + t];
            losses[b * T + t] = -logf(probs_bt[ix]);
        }
    }
```

gpt2_backward

```c
crossentropy_softmax_backward(grads_acts.logits, grads_acts.losses, acts.probs, model->targets, B, T, V, Vp);
matmul_backward(grads_acts.lnf, grads.wte, NULL, grads_acts.logits, acts.lnf, params.wte, B, T, C, Vp);
float* residual = acts.residual3 + (L-1) * B * T * C; // last layer's residual
float* dresidual = grads_acts.residual3 + (L-1) * B * T * C; // write to last layer's residual
layernorm_backward(dresidual, grads.lnfw, grads.lnfb, grads_acts.lnf, residual, params.lnfw, acts.lnf_mean, acts.lnf_rstd, B, T, C);

for (int l = L-1; l >= 0; l--) {

    residual = l == 0 ? acts.encoded : acts.residual3 + (l-1) * B * T * C;
    dresidual = l == 0 ? grads_acts.encoded : grads_acts.residual3 + (l-1) * B * T * C;

    // get the pointers of the weights for this layer
    float* l_ln1w = params.ln1w + l * C;
```

```c
    // backprop this layer
    residual_backward(dl_residual2, dl_fcproj, dl_residual3, B*T*C);
    matmul_backward(dl_fch_gelu, dl_fcprojw, dl_fcprojb, dl_fcproj, l_fch_gelu, l_fcprojw, B, T, 4*C, C);
    gelu_backward(dl_fch, l_fch, dl_fch_gelu, B*T*4*C);
    matmul_backward(dl_ln2, dl_fcw, dl_fcb, dl_fch, l_ln2, l_fcw, B, T, C, 4*C);
    layernorm_backward(dl_residual2, dl_ln2w, dl_ln2b, dl_ln2, l_residual2, l_ln2w, l_ln2_mean, l_ln2_rstd, B, T, C);
    residual_backward(dresidual, dl_attproj, dl_residual2, B*T*C);
    matmul_backward(dl_atty, dl_attprojw, dl_attprojb, dl_attproj, l_atty, l_attprojw, B, T, C, C);
    attention_backward(dl_qkv, dl_preatt, dl_att, dl_atty, l_qkv, l_att, B, T, C, NH);
    matmul_backward(dl_ln1, dl_qkvw, dl_qkvb, dl_qkv, l_ln1, l_qkvw, B, T, C, 3*C);
    layernorm_backward(dresidual, dl_ln1w, dl_ln1b, dl_ln1, residual, l_ln1w, l_ln1_mean, l_ln1_rstd, B, T, C);
```

crossentropy_softmax_backward

```c
void crossentropy_softmax_backward(float* dlogits,
                                   float* dlosses, float* probs, int* targets,
                                   int B, int T, int V, int Vp) {
    // backwards through both softmax and crossentropy
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            float* dlogits_bt = dlogits + b * T * Vp + t * Vp;
            float* probs_bt = probs + b * T * Vp + t * Vp;
            float dloss = dlosses[b * T + t];
            int ix = targets[b * T + t];
            // note we only loop to V, leaving the padded dimensions
            // of dlogits untouched, so gradient there stays at zero
            for (int i = 0; i < V; i++) {
                float p = probs_bt[i];
                float indicator = i == ix ? 1.0f : 0.0f;
                dlogits_bt[i] += (p - indicator) * dloss;
            }
        }
    }
}
```

Automatic Differentiation (autograd)

See train_gpt2.py

torch.nn.Module: PyTorch's autograd engine automatically computes the gradients needed for backpropagation when you perform the forward pass.

1. **Forward pass**: When you call the `forward` function of the model, the input tensor (`x`) goes through various operations (like linear layers, activation functions). During this process, PyTorch records each of these operations, forming a computation graph.
2. **Backward pass**: Once you have the loss (final output of the network compared with ground truth), calling `loss.backward()` initiates the backward pass.

micrograd

https://github.com/karpathy/micrograd/blob/master/micrograd/engine.py

```
def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data * other.data, (self, other), '*')

    def _backward():
        self.grad += other.data * out.grad
        other.grad += self.data * out.grad
    out._backward = _backward

    return out
```

micrograd

Suppose we have a computational graph where `out` is some intermediate value, and we are eventually computing a loss L that depends on `out`. During backpropagation, we compute the gradient of the loss L with respect to `self` and `other`.

- self.grad represents $\frac{\partial L}{\partial \text{self}}$ (i.e., how the loss L changes with respect to `self`).
- out.grad represents $\frac{\partial L}{\partial \text{out}}$ (i.e., how the loss L changes with respect to `out`).

The chain rule tells us how to propagate this gradient:

$$\frac{\partial L}{\partial \text{self}} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial \text{self}}$$

Since $\text{out} = \text{self} \cdot \text{other}$, the derivative of `out` with respect to `self` is simply `other.data` (because the derivative of $x \cdot y$ with respect to x is y).

Thus:

$$\frac{\partial L}{\partial \text{self}} = \text{other.data} \cdot \text{out.grad}$$

gpt2_update

https://pytorch.org/docs/stable/generated/torch.optim.**AdamW**.html

```
void gpt2_update(GPT2 *model, float learning_rate, float beta1, float beta2, float eps, float weight_decay, int t) {
    // reference: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

    // lazily allocate the memory for m_memory and v_memory
    if (model->m_memory == NULL) {
        model->m_memory = (float*)calloc(model->num_parameters, sizeof(float));
        model->v_memory = (float*)calloc(model->num_parameters, sizeof(float));
    }

    for (size_t i = 0; i < model->num_parameters; i++) {
        float param = model->params_memory[i];
        float grad = model->grads_memory[i];

        // update the first moment (momentum)
        float m = beta1 * model->m_memory[i] + (1.0f - beta1) * grad;
        // update the second moment (RMSprop)
        float v = beta2 * model->v_memory[i] + (1.0f - beta2) * grad * grad;
        // bias-correct both moments
        float m_hat = m / (1.0f - powf(beta1, t));
        float v_hat = v / (1.0f - powf(beta2, t));

        // update
        model->m_memory[i] = m;
        model->v_memory[i] = v;
        model->params_memory[i] -= learning_rate * (m_hat / (sqrtf(v_hat) + eps) + weight_decay * param);
    }
}
```

Adam vs AdamW

- **Adam**: The update rule for Adam is as follows:

$$\theta_t = \theta_{t-1} - \eta \cdot \left(\frac{m_t}{\sqrt{v_t} + \epsilon} + \lambda \cdot \theta_{t-1} \right)$$

  Here, λ is the L2 regularization term, and it affects both the momentum and the weights.

- **AdamW**: The update rule for AdamW is slightly different:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} - \lambda \cdot \theta_{t-1}$$

  In this case, weight decay (λ) is **separated** from the gradient update, which results in better and more efficient regularization.

AdamW Optimizer

https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

Reviews: https://openreview.net/forum?id=Bkg6RiCqY7

a useful and influential finding

ICLR 2019 Conference Paper939 Area Chair1

05 Dec 2018, 08:49 (modified: 20 Dec 2018, 19:08) ICLR 2019 Conference Paper939 Meta Review Readers: Everyone Show Revisions

Metareview: Evaluating this paper is somewhat awkward because it has already been through multiple reviewing cycles, and in the meantime, the trick has already become widely adopted and inspired interesting follow-up work. Much of the paper is devoted to reviewing this follow-up work. I think it's clearly time for this to be made part of the published literature, so I recommend acceptance. (And all reviewers are in agreement that the paper ought to be accepted.)

The paper proposes, in the context of Adam, to apply literal weight decay in place of L2 regularization. An impressively thorough set of experiments are given to demonstrate the improved generalization performance, as well as a decoupling of the hyperparameters.

Previous versions of the paper suffered from a lack of theoretical justification for the proposed method. Ordinarily, in such cases, one would worry that the improved results could be due to some sort of experimental confound. But AdamW has been validated by so many other groups on a range of domains that the improvement is well established. And other researchers have offered possible explanations for the improvement.

Recommendation: Accept (Poster)

HW2: Reproducing ChatGPT

https://github.com/parasol-aser/hw-reproduce-chatgpt

Due Oct 12, Saturday