

Lecture 4 Language Modeling

Fei Tan

Department of Economics
Chaifetz School of Business
Saint Louis University

E6930 Introduction to Neural Networks

July 23, 2024

Makemore

- ▶ Autoregressive character-level language model
 - ▶ input: a text file of words
 - ▶ output: similar words as input
- ▶ Application to name generation

```
# Input: 32K common names from ssa.gov in 2018  
$ python makemore.py -i names.txt -o names
```

```
# Output: name-like words
```

```
dontell  
khylum  
camatena  
aeriline  
najlah  
sherrith
```

The Road Ahead...

- ① Bigram
- ② Multilayer Perceptron
- ③ Transformer

Bigram Counts

0	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	
6640	556	541	470	1042	692	134	268	2332	1650	175	968	2528	1634	175	63	82	60	842	1118	687	381	834	161	882	2050	435	
h	ba	bb	bc	bd	be	bf	bg	bh	bi	bj	bk	bl	bm	bn	bo	bp	bq	br	bs	bt	bu	bv	bw	bx	by	bz	
114	321	38	1	65	655	1	41	217	1	103	1	1	1	105	1	11	1	76	5	35	1	17	1	1	1		
97	815	0	42	1	551	0	2	664	271	3	316	116	0	0	380	1	11	1	76	5	35	1	17	1	1		
516	1503	1	3	149	1283	1	25	118	674	0	3	60	30	31	378	0	1	1	424	29	4	32	17	23	0		
679	121	153	384	1271	82	125	152	818	55	178	2548	769	2675	269	83	12	1958	861	580	89	483	50	132	1070	181		
80	242	0	0	123	44	1	1	160	0	2	20	1	4	60	0	0	114	6	18	10	1	4	14	2			
108	310	1	19	354	1	22	360	190	1	0	2	0	0	27	0	0	201	95	1	85	1	1	1	1	1		
2409	2244	8	2	674	12	12	729	1	9	29	185	117	138	287	1	1	204	31	71	166	39	10	213	12			
2489	2445	110	509	440	1653	101	478	95	82	76	445	1345	427	2126	588	53	52	849	1316	541	109	269	18	89	779	277	
1	1473	1	4	4	440	0	0	45	119	1	2	6	1	5	2	479	1	0	11	7	2	202	5	16	1	10	6
363	1731	1	2	2	895	1	1	307	509	1	2	20	139	26	344	1	1	109	95	17	50	2	34	0	379	12	
1314	2623	52	25	138	2921	22	6	19	2480	1	6	24	1345	60	14	692	15	3	18	94	77	324	72	16	1	1588	10
116	2590	112	51	24	818	1	0	1256	1	1	5	168	20	452	38	0	97	35	4	139	3	2	0	287	11		
678	2577	8	0	704	1359	11	273	26	1725	44	58	195	19	1906	496	0	2	44	278	443	96	55	11	6	465	145	
855	149	140	114	190	132	34	44	171	69	16	68	619	261	2411	115	95	3	1059	504	118	275	176	114	45	103	54	
33	209	2	1	1	1	1	204	61	1	1	1	16	1	1	59	59	0	151	16	17	4	0	0	0	12	0	
28	13	0	0	1	1	1	0	0	13	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	
1377	2356	41	99	187	1697	1	76	121	3033	25	90	413	162	140	869	14	16	425	190	208	252	80	21	3	773	23	
1169	1201	21	60	9	884	2	52	1285	684	2	82	279	90	24	531	51	1	55	461	765	185	14	24	0	215	52	
483	1027	1	17	0	716	2	2	647	532	3	0	134	4	22	667	1	1	352	35	374	78	15	11	2	341	105	
155	163	103	103	136	169	19	47	58	121	14	93	301	154	275	10	16	10	414	474	82	3	37	86	14	13	45	
88	642	1	0	1	568	1	0	1	911	0	3	14	0	8	153	0	0	48	0	0	7	7	0	0	121	0	
51	280	1	0	149	149	2	1	23	148	0	6	13	2	58	36	0	0	22	0	8	25	0	0	0	3	19	
164	103	1	0	36	3	0	1	102	0	0	39	1	1	41	0	0	0	31	31	70	5	0	0	0	30	19	
2007	2143	27	115	272	301	12	30	22	192	23	86	1104	148	1826	271	15	6	291	401	104	141	106	4	28	23	78	
160	860	4	2	373	0	1	43	364	2	2	2	123	35	4	110	2	0	32	25	4	4	73	2	2	147	45	

Preprocessing

```
import torch

# Training dataset
xs, ys = [], []
for w in words:      # list of names
    chs = ['.''] + list(w) + ['.'']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1 = stoi[ch1] # char to index
        ix2 = stoi[ch2]
        xs.append(ix1)
        ys.append(ix2)
xs = torch.tensor(xs)
ys = torch.tensor(ys)

# Weight initialization
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g,
                requires_grad=True)
```

Training

```
import torch.nn.functional as F

# Gradient descent (maximum likelihood estimation)
for k in range(10000):
    # Forward pass
    xenc = F.one_hot(xs, num_classes=27).float() #
        one-hot encoding to avoid ordering
    logits = xenc @ W # softmax classifier
    counts = logits.exp()
    probs = counts / counts.sum(1, keepdims=True)
    loss = -probs[torch.arange(num), ys].log().
        mean() + 0.01*(W**2).mean()
    print(loss.item())

    # Backward pass
    W.grad = None # zero gradients
    loss.backward()
    W.data += -0.1 * W.grad
```

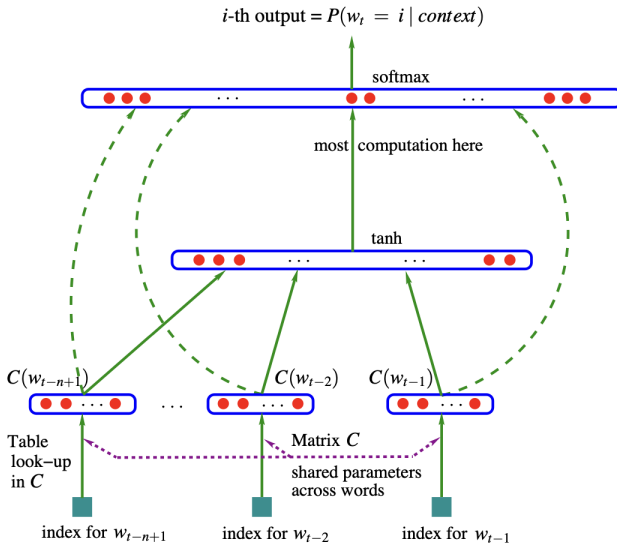
Prediction

```
# Sampling
for i in range(5):
    out = []
    ix = 0
    while True:
        xenc = F.one_hot(torch.tensor([ix]),
                           num_classes=27).float()
        logits = xenc @ W
        counts = logits.exp()
        p = counts / counts.sum(1, keepdims=True)
        ix = torch.multinomial(p, num_samples=1,
                               replacement=True, generator=g).item()
        out.append(itos[ix])
        if ix == 0:
            break
    print(''.join(out)) # mor. axx. minaymoryles.
                        kondlaisah. anchthizarie.
```

The Road Ahead...

- ① Bigram
- ② Multilayer Perceptron
- ③ Transformer

Model Architecture



PyTorch-Like API

```
class Linear:    # torch.nn.Linear
    def __init__(self, fan_in, fan_out, bias=True):
        self.weight = torch.randn((fan_in, fan_out
                                   ), generator=g)    # or Kaiming init
        self.bias = torch.zeros(fan_out) if bias
                else None
    def __call__(self, x):
        self.out = x @ self.weight
        if self.bias is not None:
            self.out += self.bias
        return self.out
    def parameters(self):
        return [self.weight] + ([ self.bias is
                                None else [self.bias]])

class Tanh:      # torch.nn.Tanh
    def __call__(self, x):
        self.out = torch.tanh(x)
        return self.out
```

PyTorch-Like API (Cont'd)

```
class BatchNorm1d:  # torch.nn.BatchNorm1d
    def __init__(self, dim, eps=1e-5, momentum
                  =0.1):
        self.eps = eps
        self.momentum = momentum
        self.training = True
        # Trained with backprop
        self.gamma = torch.ones(dim)
        self.beta = torch.zeros(dim)
        # Trained with running momentum update
        self.running_mean = torch.zeros(dim)
        self.running_var = torch.ones(dim)

    def parameters(self):
        return [self.gamma, self.beta]
    ...
```

PyTorch-Like API (Cont'd)

```
class BatchNorm1d:  # torch.nn.BatchNorm1d
    def __call__(self, x):
        if self.training:
            xmean = x.mean(0, keepdim=True)
            xvar = x.var(0, keepdim=True)
        else:
            xmean = self.running_mean
            xvar = self.running_var
        xhat = (x - xmean) / torch.sqrt(xvar +
            self.eps)  # normalization
        self.out = self.gamma * xhat + self.beta
        if self.training:
            with torch.no_grad():
                self.running_mean = (1 - self.
                    momentum) * self.running_mean
                    + self.momentum * xmean
                self.running_var = (1 - self.
                    momentum) * self.running_var +
                    self.momentum * xvar
        return self.out
```

Preprocessing

```
for w in words:
    context = [0] * block_size
    for ch in w + ' ':
        ix = stoi[ch]
        X.append(context)
        Y.append(ix)
        context = context[1:] + [ix]
X = torch.tensor(X)
Y = torch.tensor(Y)

C = torch.randn((vocab_size, n_embd), generator=g)
layers = [
    Linear(n_embd * block_size, n_hidden, bias=
        False), BatchNorm1d(n_hidden), Tanh(),
    Linear(n_hidden, n_hidden, bias=False),
        BatchNorm1d(n_hidden), Tanh(),
    ...
    Linear(n_hidden, vocab_size, bias=False),
        BatchNorm1d(vocab_size)]
```

Training

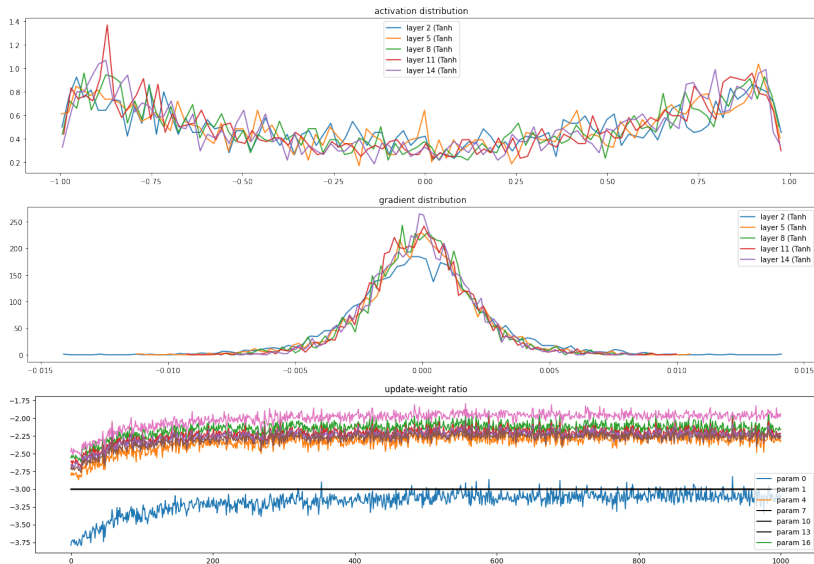
```
for i in range(200000):
    # Forward pass
    ix = torch.randint(0, X.shape[0], (batch_size,)) # minibatch
    emb = C[X[ix]]
    x = emb.view(emb.shape[0], -1)
    for layer in layers:
        x = layer(x)
    loss = F.cross_entropy(x, Y[ix])

    # Backward pass
    for layer in layers:
        layer.out.retain_grad()
    for p in parameters:
        p.grad = None
    loss.backward()
    lr = 0.1 if i < 100000 else 0.01
    for p in parameters:
        p.data += -lr * p.grad
```

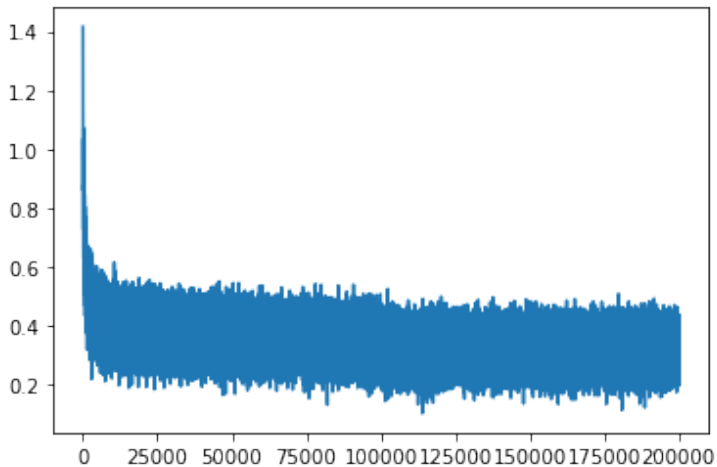
Prediction

```
for _ in range(5):
    out = []
    context = [0] * block_size
    while True:
        emb = C[torch.tensor([context])]
        x = emb.view(emb.shape[0], -1)
        for layer in layers:
            x = layer(x)
        probs = F.softmax(x, dim=1)
        ix = torch.multinomial(probs, num_samples
                               =1, generator=g).item()
        context = context[1:] + [ix]
        out.append(ix)
        if ix == 0:
            break
    print(''.join(itos[i] for i in out)) # carpah
    . garlileif. jmrrix. thty. sacansa.
```

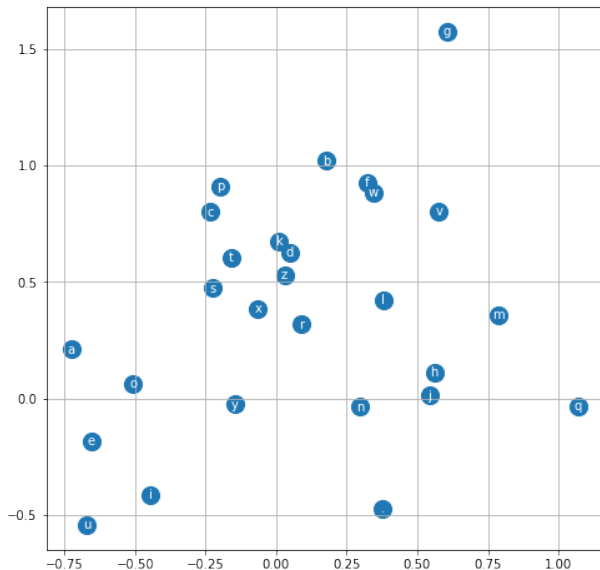
Diagnoses



Loss Trace (log10)



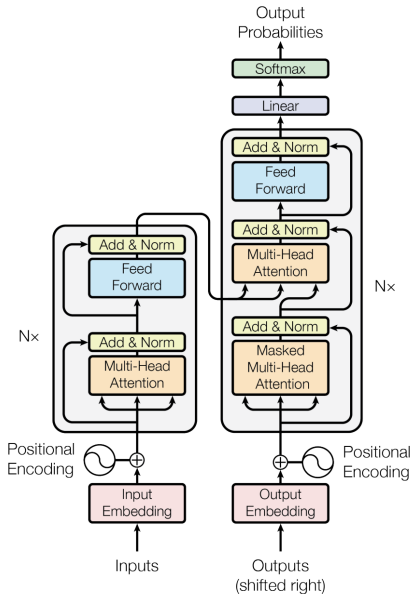
Character Embeddings



The Road Ahead...

- ① Bigram
- ② Multilayer Perceptron
- ③ Transformer

Model Architecture



Attention

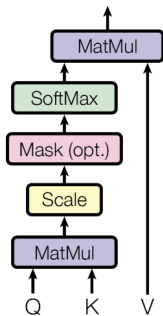
Scaled dot-product attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

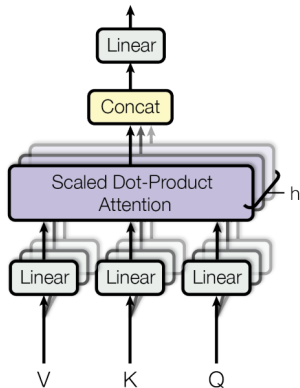
- ▶ Key concepts
 - ▶ *query* (Q) represents current word, *key* (K) captures other words, *value* (V) carries information
 - ▶ encoder vs. decoder, self-attention vs. cross-attention
- ▶ Why attention mechanism?
 - ▶ eliminate need for recurrence or convolution
 - ▶ superior performance while maintaining parallelization

Attention (Cont'd)

Scaled Dot-Product Attention



Multi-Head Attention



Single-Head Attention

```
class Head(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size,
                               bias=False)
        self.query = nn.Linear(n_embd, head_size,
                                bias=False)
        self.value = nn.Linear(n_embd, head_size,
                                bias=False)
        self.register_buffer('tril', torch.tril(
            torch.ones(block_size, block_size)))
        # not updated during backprop
        self.dropout = nn.Dropout(dropout)
    ...
```

Single-Head Attention (Cont'd)

```
class Head(nn.Module):
    def forward(self, x):
        B,T,C = x.shape # (batch, time, channels)
        k = self.key(x)
        q = self.query(x)
        wei = q @ k.transpose(-2,-1) * k.shape
            [-1]**-0.5 # attention scores
        wei = wei.masked_fill(self.tril[:T, :T] ==
            0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        out = wei @ v # (batch, time, head size)
        return out
```


Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size)
                                     for _ in range(num_heads)])
        self.proj = nn.Linear(head_size *
                               num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads
                        ], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

Feed Forward

```
class FeedFoward(nn.Module):
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout))

    def forward(self, x):
        return self.net(x)
```

Transformer Block

```
class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head,
                                     head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

```
class GPTLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(
            vocab_size, n_embd)
        self.position_embedding_table = nn.
            Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd
            , n_head=n_head) for _ in range(
                n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd,
            vocab_size)

    ...
```

GPT (Cont'd)

```
class GPTLanguageModel(nn.Module):
    def forward(self, idx, targets):
        B, T = idx.shape
        tok_emb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(
            torch.arange(T))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)
        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)
        loss = F.cross_entropy(logits, targets)
        return logits, loss
```

Prediction

```
# wget https://raw.githubusercontent.com/karpathy/  
char-rnn/master/data/tinyshakespeare/input.txt
```

First Citizen:

Before we proceed any further, hear me speak.

All:

Speak, speak.

```
# Output: Shakespeare-like text
```

VALHASINA:

Nobleman; go, then both groans to us.

AUFIDIUS:

O those prepaion!

References

- ▶ github.com/karpathy/makemore – An autoregressive character-level language model for making more things
- ▶ github.com/karpathy/nanoGPT – The simplest, fastest repository for training/finetuning medium-sized GPTs
- ▶ github.com/karpathy/llm.c – LLM training in simple, raw C/CUDA
- ▶ Bengio et al. (2003), “A Neural Probabilistic Language Model”, *Journal of Machine Learning Research*
- ▶ Vaswani et al. (2017), “Attention is All You Need”, [arXiv:1706.03762](https://arxiv.org/abs/1706.03762)