AUTO REGRESSIVE LANGUAGE MODEL

Khethan R G

January 2024

Introduction

Autoregressive Model - An Autoregressive model in the context of neural networks is a type of model that uses observations from previous time steps as input to predict the value at the next time step. It's a form of regression model that uses lagged variables as input.

Let's consider a simple example. Suppose we have a time series data and we want to predict the value for the next time step (t+1) given the observations at the last two time steps (t-1) and (t-2). As a regression model, this would look as follows:

```
X(t+1) = b0 + b1*X(t-1) + b2*X(t-2)
```

Here, X(t+1) is the prediction, b0, b1, and b2 are coefficients found by optimizing the model on training data, and X(t-1) and X(t-2) are input values from previous time steps. Because the regression model uses data from the same input variable at previous time steps, it is referred to as an autoregression (regression of self).

A *character level language model* is a model which considers each character in a word while predicting the next character. A *Bi-Gram* neural network is a weak but good beginner language model in which we always work with just 2 characters (in a row) at a time i.e., we are only looking at the one character that was given and try to predict the next character in the sequence.

Manual Method

We basically create a 2D tensor where the values represent the likelihood of the next character (from the column), given the present character (in the row) by looking in this table.

	a	b	...	z	start	end
a			...			
b			...			
.			...			
z			...			
start			...			
end			...			

We first create a dataset which consists of start and end words and looks somewhat like the below image. This is basically the weight matrix. Although here we have some values in the image, these values are initially some random values between range -(1, 1) and then

```python
# one hot encoded x values, input to the network
xenc = F.one_hot(xs, num_classes=27).float()
# gradient descent

for k in range(1):
    # forward pass
    logits = xenc @ W
    # basically the probability of next value
    loss = F.cross_entropy(logits, yb)

    W.grad = None
    loss.backward()
    W.data += -50 * W.grad
```

Listing 1: Basic Weight Method

We use this part of the code to update this matrix which helps in finding the next possible value, given the previous value. But this has a lot of disadvantages, like:

- Works only if we have one character from the previous context.
- The predictions also are not very good because only one character is taken to predict.
- If we take "more than one character" to predict the next character then things quickly blow up i.e., the N matrix which is of size (27, 27) is present only for one previous character, since there are 27 possibilities for the prediction i.e., (<.>, a, b, c, ..., z).
- If we take say 2 previous characters for prediction of the next character, then $27^2 = 729$ number of possibilities exist.

- Prediction array = 27^n where n = number of characters used for prediction.

Multi-Layer Perceptron

To overcome this limitation, we need to use a more sophisticated approach that can capture the features of previous words and use them to predict the features of the next word. For example, we can use a neural network that learns a vector representation for each word, and then uses a recurrent or attention mechanism to combine the vectors of the previous words into a context vector. This context vector can then be used to generate the next word, either by sampling from a soft-max distribution or by using a decoder network.

Using a feature representation also allows for a larger context than n-gram models. N-gram models are limited by the size of the n-gram order, which is usually between 3 and 5. Using a larger n-gram order would result in a huge number of parameters and a sparse data problem. However, using a feature representation, we can use a context that contains many more previous words (e.g., 10) without increasing the number of parameters or the data sparsity.

We want to create an embedding space where each character can have a good vector representation that indicates the location of this character in the space, like the below image, but instead of 2 dimensions, we can have any number of dimensions as we want.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD
class CharLanguageModel(nn.Module):
    def __init__(self, vocab_size, n_embd, n_hidden, block_size, max_steps, batch_size)
        super().__init__()
        self.vocab_size = vocab_size
        self.n_embd = n_embd
        self.n_hidden = n_hidden
        self.block_size = block_size
        self.max_steps = max_steps
        self.batch_size = batch_size
        self.chars = '.abcdefghijklmnopqrstuvwxyz'
        self.stoi = {s:i for i,s in enumerate(self.chars)}
        self.itos = {i:s for s,i in self.stoi.items()}
        self.C = torch.randn((vocab_size, n_embd), requires_grad=True)
        self.fc1 = nn.Linear(n_embd * block_size, n_hidden)
        self.bn = nn.BatchNorm1d(n_hidden)
        self.fc2 = nn.Linear(n_hidden, vocab_size)
        self.optimizer = SGD(self.parameters(), lr=0.1)

def forward(self, Xb):
    emb = self.C[Xb]
    embcat = emb.view(emb.shape[0], -1)
    hprebn = self.fc1(embcat)
    hprebn = self.bn(hprebn)
    h = torch.tanh(hprebn)
    logits = self.fc2(h)
    return logits

def backward(self, loss):
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

def train_model(self, Xtr, Ytr):
    for i in range(self.max_steps):
        ix = torch.randint(0, Xtr.shape[0], (self.batch_size,))
        Xb, Yb = Xtr[ix], Ytr[ix]
        logits = self.forward(Xb)
        loss = F.cross_entropy(logits, Yb)
        self.backward(loss)
        if i % 10000 == 0:
            print(f'{i:7d}/{self.max_steps:7d}: {loss.item():.4f}')
```

Listing 2: A character language model

In the above code, we embed the 27 characters that we have into a 10-dimensional space to obtain a ***Tensor***, ***C*** having 27 characters embedded in 10-dimensional space. Then we pass these embedding

Figure 1: Probability matrix

.. 0	.a 4410	.b 1306	.c 1542	.d 1690	.e 1531	.f 417	.g 669	.h 874	.i 591	.j 2422	.k 2963	.l 1572	.m 2538	.n 1146	.o 394	.p 515	.q 92	.r 1639	.s 2055	.t 1308	.u 78	.v 376	.w 307	.x 134	.y 535	.z 929
a. 6640	aa 556	ab 541	ac 470	ad 1042	ae 692	af 134	ag 168	ah 2332	ai 1650	aj 175	ak 568	al 2528	am 1634	an 5438	ao 63	ap 82	aq 60	ar 3264	as 1118	at 687	au 381	av 834	aw 161	ax 182	ay 2050	az 435
b. 114	ba 321	bb 38	bc 1	bd 65	be 655	bf 0	bg 0	bh 41	bi 217	bj 1	bk 0	bl 103	bm 0	bn 4	bo 105	bp 0	bq 0	br 842	bs 8	bt 2	bu 45	bv 0	bw 0	bx 0	by 83	bz 0
c. 97	ca 815	cb 0	cc 42	cd 1	ce 551	cf 0	cg 2	ch 664	ci 271	cj 3	ck 316	cl 116	cm 0	cn 0	co 380	cp 1	cq 11	cr 76	cs 5	ct 35	cu 35	cv 0	cw 0	cx 3	cy 104	cz 4
d. 516	da 1303	db 1	dc 3	dd 149	de 1283	df 5	dg 25	dh 118	di 674	dj 9	dk 3	dl 60	dm 30	dn 31	do 378	dp 0	dq 1	dr 424	ds 29	dt 4	du 92	dv 17	dw 23	dx 0	dy 317	dz 1
e. 3983	ea 679	eb 121	ec 153	ed 384	ee 1271	ef 82	eg 125	eh 152	ei 818	ej 55	ek 178	el 3248	em 769	en 2675	eo 269	ep 83	eq 14	er 1958	es 861	et 580	eu 69	ev 463	ew 50	ex 132	ey 1070	ez 181
f. 80	fa 242	fb 0	fc 0	fd 0	fe 123	ff 44	fg 1	fh 1	fi 160	fj 0	fk 2	fl 20	fm 0	fn 4	fo 60	fp 0	fq 0	fr 114	fs 6	ft 18	fu 10	fv 0	fw 4	fx 0	fy 14	fz 2
g. 108	ga 330	gb 3	gc 0	gd 19	ge 334	gf 1	gg 25	gh 360	gi 190	gj 3	gk 0	gl 32	gm 6	gn 27	go 83	gp 0	gq 0	gr 201	gs 30	gt 31	gu 85	gv 1	gw 26	gx 0	gy 31	gz 1
h. 2409	ha 2244	hb 8	hc 2	hd 24	he 674	hf 2	hg 2	hh 1	hi 729	hj 9	hk 29	hl 185	hm 117	hn 138	ho 287	hp 1	hq 1	hr 204	hs 31	ht 71	hu 166	hv 39	hw 10	hx 0	hy 213	hz 20
i. 2489	ia 2445	ib 110	ic 509	id 440	ie 1653	if 101	ig 428	ih 95	ii 82	ij 76	ik 445	il 1345	im 427	in 2126	io 588	ip 53	iq 52	ir 849	is 1316	it 541	iu 109	iv 269	iw 8	ix 89	iy 779	iz 277
j. 71	ja 1473	jb 1	jc 4	jd 4	je 440	jf 0	jg 0	jh 45	ji 119	jj 2	jk 2	jl 9	jm 5	jn 2	jo 479	jp 1	jq 0	jr 11	js 7	jt 2	ju 202	jv 5	jw 6	jx 0	jy 10	jz 0
k. 363	ka 1731	kb 2	kc 2	kd 2	ke 895	kf 1	kg 0	kh 307	ki 509	kj 2	kk 20	kl 139	km 9	kn 26	ko 344	kp 0	kq 0	kr 109	ks 95	kt 17	ku 50	kv 2	kw 34	kx 0	ky 379	kz 2
l. 1314	la 2623	lb 52	lc 25	ld 138	le 2921	lf 22	lg 6	lh 19	li 2480	lj 6	lk 24	ll 1345	lm 60	ln 14	lo 692	lp 15	lq 3	lr 18	ls 94	lt 77	lu 324	lv 72	lw 16	lx 0	ly 1588	lz 10
m. 516	ma 2590	mb 112	mc 51	md 24	me 818	mf 1	mg 0	mh 5	mi 1256	mj 7	mk 1	ml 5	mm 168	mn 20	mo 452	mp 38	mq 0	mr 97	ms 35	mt 4	mu 139	mv 3	mw 2	mx 0	my 287	mz 11
n. 6763	na 2977	nb 8	nc 213	nd 704	ne 1359	nf 11	ng 273	nh 26	ni 1725	nj 44	nk 58	nl 195	nm 19	nn 1906	no 496	np 5	nq 2	nr 44	ns 278	nt 443	nu 96	nv 55	nw 11	nx 6	ny 465	nz 145
o. 855	oa 149	ob 140	oc 114	od 190	oe 132	of 34	og 44	oh 171	oi 69	oj 16	ok 68	ol 619	om 261	on 2411	oo 115	op 95	oq 3	or 1059	os 504	ot 118	ou 275	ov 176	ow 114	ox 45	oy 103	oz 54
p. 33	pa 209	pb 2	pc 1	pd 0	pe 197	pf 1	pg 0	ph 204	pi 61	pj 1	pk 1	pl 16	pm 1	pn 1	po 59	pp 39	pq 0	pr 151	ps 16	pt 17	pu 4	pv 0	pw 0	px 0	py 12	pz 0
q. 28	qa 13	qb 0	qc 0	qd 0	qe 1	qf 0	qg 0	qh 0	qi 13	qj 0	qk 0	ql 1	qm 2	qn 0	qo 2	qp 0	qq 0	qr 1	qs 2	qt 0	qu 206	qv 0	qw 3	qx 0	qy 0	qz 0
r. 1377	ra 2356	rb 41	rc 99	rd 187	re 1697	rf 9	rg 76	rh 121	ri 3033	rj 25	rk 90	rl 413	rm 162	rn 140	ro 869	rp 14	rq 16	rr 425	rs 190	rt 208	ru 252	rv 80	rw 21	rx 3	ry 773	rz 23
s. 1169	sa 1201	sb 21	sc 60	sd 9	se 884	sf 2	sg 2	sh 1285	si 684	sj 2	sk 82	sl 279	sm 90	sn 24	so 531	sp 51	sq 1	sr 55	ss 461	st 765	su 185	sv 14	sw 24	sx 0	sy 215	sz 10
t. 483	ta 1027	tb 1	tc 17	td 0	te 716	tf 2	tg 2	th 647	ti 532	tj 3	tk 0	tl 134	tm 4	tn 22	to 667	tp 0	tq 0	tr 352	ts 35	tt 374	tu 78	tv 15	tw 11	tx 2	ty 341	tz 105
u. 155	ua 163	ub 103	uc 103	ud 136	ue 169	uf 19	ug 47	uh 58	ui 121	uj 14	uk 93	ul 301	um 154	un 275	uo 10	up 16	uq 10	ur 414	us 474	ut 82	uu 3	uv 37	uw 86	ux 34	uy 13	uz 45
v. 88	va 642	vb 1	vc 0	vd 1	ve 568	vf 0	vg 0	vh 1	vi 911	vj 0	vk 3	vl 14	vm 0	vn 8	vo 153	vp 0	vq 0	vr 48	vs 0	vt 0	vu 7	vv 7	vw 0	vx 0	vy 121	vz 0
w. 51	wa 280	wb 1	wc 0	wd 8	we 149	wf 2	wg 1	wh 23	wi 148	wj 0	wk 6	wl 13	wm 2	wn 58	wo 36	wp 0	wq 0	wr 22	ws 20	wt 8	wu 25	wv 0	ww 2	wx 0	wy 73	wz 1
x. 164	xa 103	xb 1	xc 4	xd 5	xe 36	xf 3	xg 0	xh 1	xi 102	xj 0	xk 0	xl 39	xm 1	xn 1	xo 41	xp 0	xq 0	xr 0	xs 31	xt 70	xu 5	xv 0	xw 3	xx 38	xy 30	xz 19
y. 2007	ya 2143	yb 27	yc 115	yd 272	ye 301	yf 12	yg 30	yh 22	yi 192	yj 23	yk 86	yl 1104	ym 148	yn 1826	yo 271	yp 15	yq 6	yr 291	ys 401	yt 104	yu 141	yv 106	yw 4	yx 28	yy 23	yz 78
z. 160	za 860	zb 4	zc 2	zd 2	ze 373	zf 0	zg 1	zh 43	zi 364	zj 2	zk 2	zl 123	zm 35	zn 4	zo 110	zp 2	zq 0	zr 32	zs 4	zt 4	zu 73	zv 2	zw 3	zx 1	zy 147	zz 45

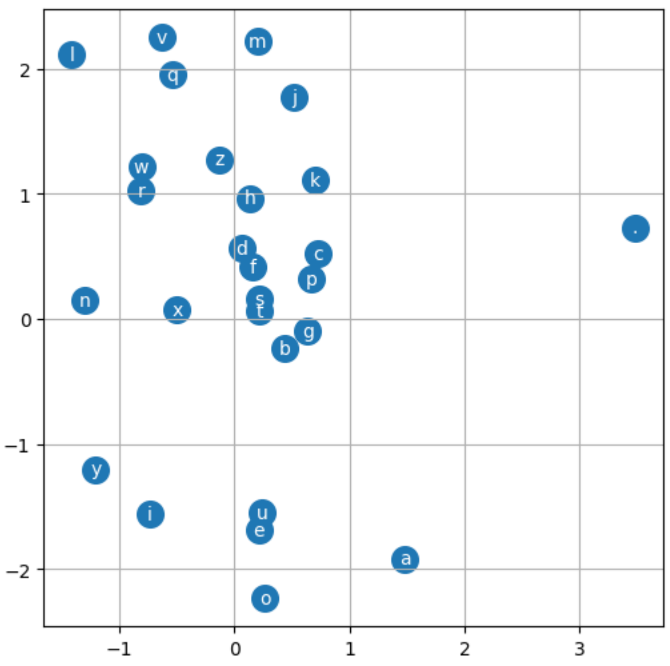


Figure 2: 2D Embedding

into a linear layer, apply an activation function, normalize it using batch norm, pass through another linear layer and finally use soft max to get the prediction with maximum probability.

Wave-Net Architecture

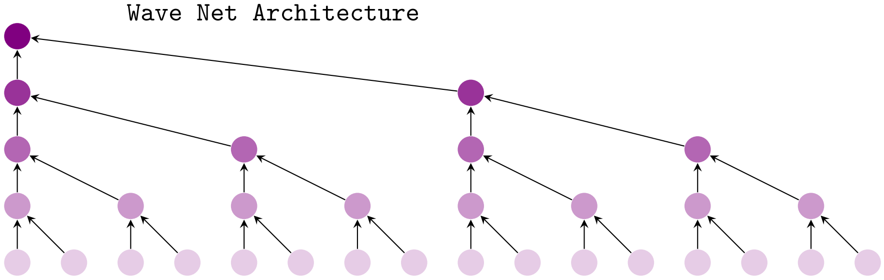
Previously, we used ‘3‘ characters, embedded them, squashed them together, and fed them all into ‘tanh‘. This potentially squashed too much information too quickly, causing the model to stagnate.

Now, we’re making it deeper with a tree-like structure, arriving at a convolutional neural network architecture similar to WaveNet (2016) from DeepMind. In the WaveNet paper, the same hierarchical architecture is implemented more efficiently using causal dilated convolutions, but we don’t do that here.

- **Causal convolutions** ensure that output at a given time step depends only on past and current inputs, not future ones.
- This leads to **better inference** compared to fully sequential models.
- This architecture can capture **short-term dependencies** within words and **long-term dependencies** across sentences.

Our plan is similar to the approach in the paper. We merge/concatenate 2 data points. For example, say we have a single input that, after mapping from string to integers, looks like ‘1,2,3,4,5,4,2,0‘. Our context length here is ‘8‘ characters to predict the next character. Now we must embed these characters using some embedding dimension, resulting in a single row of size ‘8*embd_dim‘. This was our previous approach. It squashes a lot of info too quickly, so the model takes some time to train. But from WaveNet, we take this ‘8*embd_dim‘ and reshape it to ‘(1,2)*embd_dim,(3,4)*embd_dim,(5,4)*embd_dim ,(2,0)*embd_dim‘. Here, we are essentially building a deeper network where at each layer, the input only depends on ‘2‘ previous values present below it. The below diagram gives the basic idea.

Wave Net Architecture



```python
# hierarchical network
n_embd = 24 # the dimensionality of the character embedding vectors
n_hidden = 128 # the number of neurons in the hidden layer of the MLP
model = Sequential([
  Embedding(vocab_size, n_embd),
  FlattenConsecutive(2), Linear(n_embd * 2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  Linear(n_hidden, vocab_size),
])
```

Listing 3: WaveNet model

The basic idea is that we are progressively fusing 2 char at first layer, 2 two-chars at second layer and 2 four-chars at third layer and so on.

This is how the fusing happens, we are not squashing everything here at a time, we are merging inputs part by part which helps in preserving information.

```
Embedding : (4, 8, 24)
FlattenConsecutive : (4, 4, 48)
Fusing previous 2 data points

Linear : (4, 4, 128)
BatchNorm1d : (4, 4, 128)
Tanh : (4, 4, 128)
FlattenConsecutive : (4, 2, 256)
Fusing previous 2 data points

Linear : (4, 2, 128)
BatchNorm1d : (4, 2, 128)
Tanh : (4, 2, 128)
FlattenConsecutive : (4, 256)
Fusing previous 2 data points

Linear : (4, 128)
BatchNorm1d : (4, 128)
Tanh : (4, 128)
Linear : (4, 27)
Total layers in the model, 15
```

This approach also provides better training and validation loss, and also generates more readable text like shown below:

```
naston.
aashna.
raziah.
edonie.
phoelox.
dawtine.
unnalee.
```

RNN

Although WaveNet architecture that we implemented was good, it can still only work for generating names, and can't be used to generate textual sentences, this is where Recurrent neural networks come into picture.

A *Recurrent Neural Network* (RNN) is a type of neural network that is powerful for modeling sequence data such as time series or natural language. Unlike traditional neural networks, where all the inputs and outputs are independent of each other, RNNs have the ability to remember previous inputs while processing.

The fundamental processing unit in an RNN is a Recurrent Unit/Cell, which has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing. This is also referred to as Memory State.

They are originally three layer networks. The above figure shows a simple one-to-one vanilla

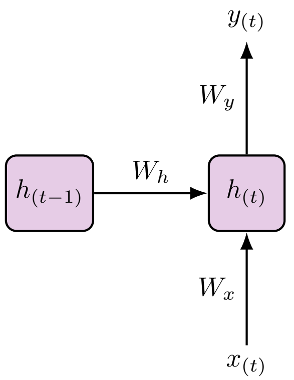


Figure 3: Recurrent neural network

RNN i.e., a single input gives a single output.

In the figure, x_t is taken as the input to the network at time step 't' and h_t represents the hidden state at the same time step. Calculation of h_t is based as per the equation: $h_t = f(U * x_t + W * h_{t1})$

Thus, h_t is calculated based on the current input and the previous time step's hidden state. The function f is taken to be a non-linear transformation, tanh and U, V, W account for weights that are shared across time. In the context of NLP, x_t typically comprises of one-hot encodings or embeddings.

Vanishing Gradient Problem In practice, however, these simple RNN networks suffer from the infamous vanishing gradient problem, which makes it really hard to learn and tune the parameters of the earlier layers in the network. This limitation was overcome by various networks such as long short-term memory (LSTM), gated recurrent units (GRUs), and residual networks (ResNets), where the first two are the most used RNN variants in NLP applications.

```python
self.start = nn.Parameter(torch.zeros(1, N_EMBD2))  # the starting hidden state
self.wte = nn.Embedding(VOCAB_SIZE, N_EMBD)  # token embeddings table
self.xh_to_h = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)

# embed all the integers up front and all at once for efficiency
emb = self.wte(idx)  # (b, t, N_EMBD)

# sequentially iterate over the inputs and update the RNN state each tick
hprev = self.start.expand((b, -1))  # expand out the batch dimension
hiddens = []
for i in range(t):
    xt = emb[:, i, :]  # (b, N_EMBD)
    xh = torch.cat([xt, hprev], dim=1)  # add rows horizontally
    ht = torch.tanh(self.xh_to_h(xh))  # (b, N_EMBD2)
    hprev = ht
    hiddens.append(ht)

# decode the outputs
hidden = torch.stack(hiddens, 1)  # (b, t, N_EMBD2)
logits = self.lm_head(hidden)
```

Listing 4: RNN Forward pass

Gated Recurrent Units (GRU)

A gated RNN variant called GRU used for sequential data processing. GRU comprises of two gates, reset gate and update gate. They train faster and perform better on less training data. Being less complex, GRU can be a more efficient RNN than LSTM. The working of GRU is as follows:

Update Gate (z):

$$z_t = \sigma(W_z * [h_{t-1}, x_t])$$

, or $z = (U_z.x_t + W_z.h_{t1})$ Here, z_t is the update gate, and W_z is the weight matrix for the update gate. It determines how much of the past knowledge needs to be passed along into the future.

Reset Gate (r):

$$r_t = \sigma(W_r * [h_{t-1}, x_t])$$

, or $r = (U_r.x_t + W_r.h_{t1})$ Here, r_t is the reset gate, σ is the sigmoid function, W_r is the weight matrix for the reset gate, h_{t-1} is the hidden state from the previous time step, and x_t is the input at the current time step.

It determines how much of the past knowledge to forget. It uses a sigmoid activation function to squash the values between 0 and 1.

Candidate Hidden State:

$$\tilde{h}_t = \tanh(W * [r_t \odot h_{t-1}, x_t])$$

, or $h_t = tanh(U_z.x_t + W_s.(h_{t1} \odot r))$ Here, $\tilde{h}_t$ is the candidate hidden state, $\tanh$ is the hyperbolic tangent function, W is the weight matrix, r_t is the reset gate, $\odot$ denotes element-wise multiplication, h_{t-1} is the hidden state from the previous time step, and x_t is the input at the current time step.

The candidate hidden state is calculated using the reset gate and the input.

Final Hidden State:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

, or $h_t = (1 - z) \odot s_t + z \odot h_{t-1}$ Here, h_t is the final hidden state, z_t is the update gate, h_{t-1} is the hidden state from the previous time step, and $\tilde{h}_t$ is the candidate hidden state

The final hidden state is a combination of the previous hidden state and the candidate hidden state, controlled by the update gate. The equation for the final hidden state is as follows:

These equations allow the GRU to effectively learn from sequential data by maintaining a form of memory through its hidden states

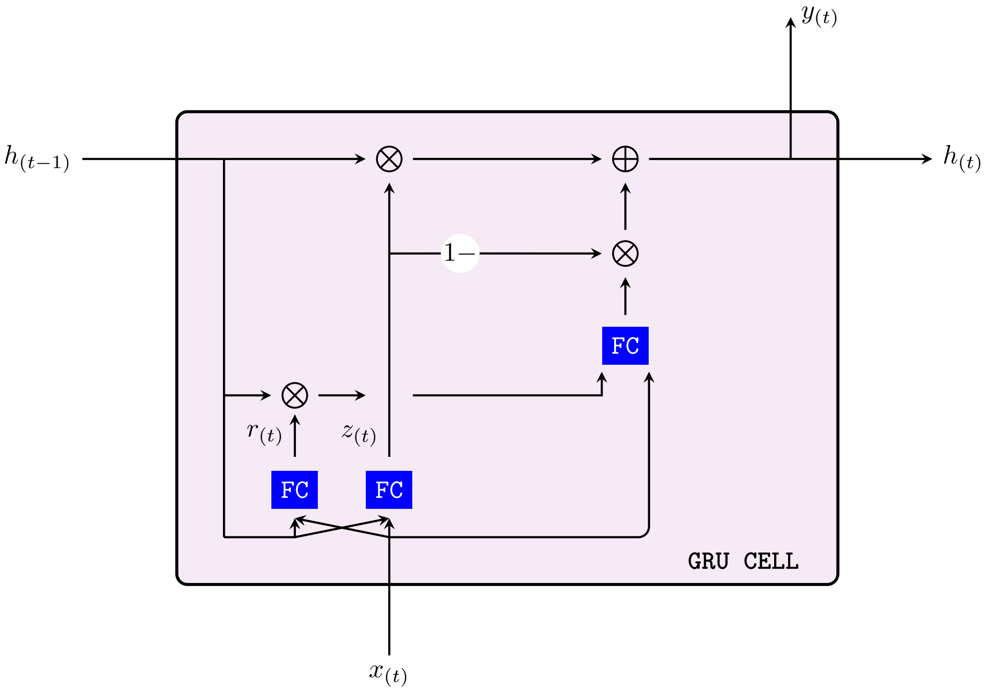


Figure 4: Gated Recurrent Unit

```
# input, forget, output, gate
self.xh_to_z = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
self.xh_to_r = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
self.xh_to_hbar = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
self.start = nn.Parameter(torch.zeros(1, N_EMBD2))  # the starting hidden state
self.wte = nn.Embedding(VOCAB_SIZE, N_EMBD)  # token embeddings table
# first use the reset gate to wipe some channels of the hidden state to zero
# embed all the integers up front and all at once for efficiency
emb = self.wte(idx)  # (b, t, N_EMBD)

# sequentially iterate over the inputs and update the RNN state each tick
hprev = self.start.expand((b, -1))  # expand out the batch dimension
hiddens = []
for i in range(t):
    xt = emb[:, i, :]  # (b, N_EMBD)
    xh = torch.cat([xt, hprev], dim=1)
    r = torch.sigmoid(self.xh_to_r(xh))  # reset gate - squashing to zero
    hprev_reset = r * hprev
    # calculate the candidate new hidden state hbar
    xhr = torch.cat([xt, hprev_reset], dim=1)
    hbar = torch.tanh(self.xh_to_hbar(xhr))  # new candidate hidden state
    # calculate the switch gate that determines if each channel should be updated at all
    z = torch.sigmoid(self.xh_to_z(xh))
    # blend the previous hidden state and the new candidate hidden state
    ht = (1 - z) * hprev + z * hbar # (b, N_EMBD2)
    hprev = ht
    hiddens.append(ht)
    # decode the outputs
    hidden = torch.stack(hiddens, 1)  # (b, t, N_EMBD2)
    logits = self.lm_head(hidden)
```

Listing 5: GRU Forward pass

Disadvantages:

- They can't train on longer data, and since our data is small, LSTM also produces similar results.
- Can't be used for training in parallel.

Transformer: Decoder Only

We only use decoder architecture here because we are just generating text in the same language. There is no need to encode anything.

Decoder:

The decoder is composed of a stack of $N = 6$ identical layers (where $N = 6$ is the base model, and $N = 12$ means a large model). N here represents the number of heads.

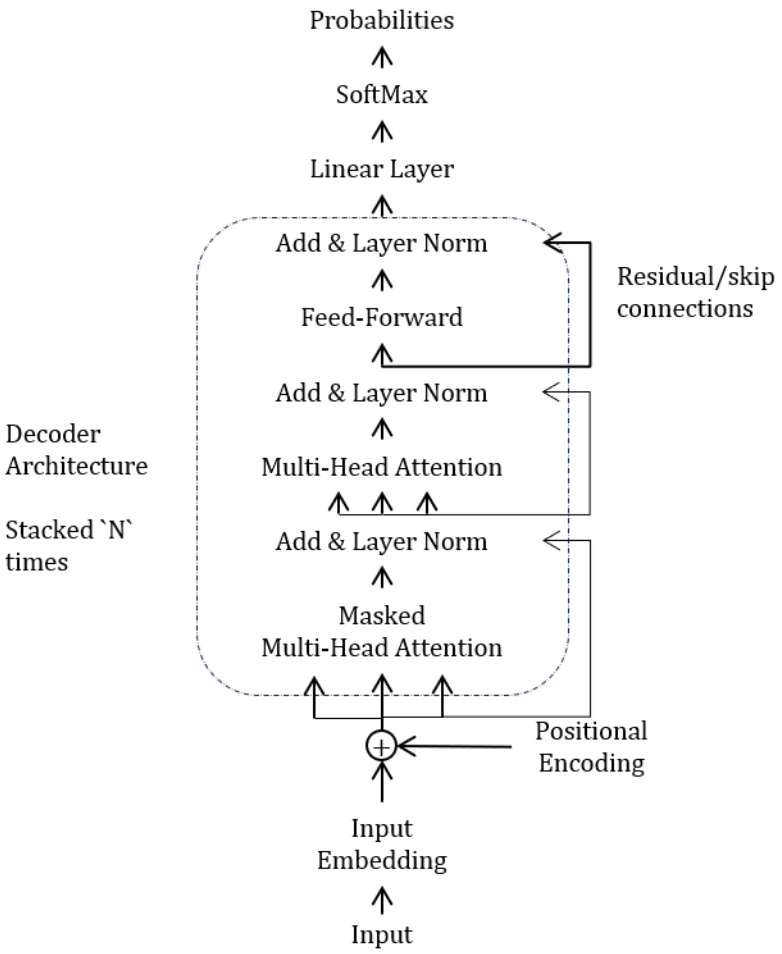


Figure 5: Transformer, Decoder only architecture

Each layer has 2 sub-layers. Although the image below shows 3 sub layers here only 2 were implemented because it's a simple model. The first is a masked multi-head self-attention mechanism, where matrices KEY, VALUE, and QUERY are created based on the input. This attention mechanism ensures that predictions for position i depend only on the known outputs at positions less than i.

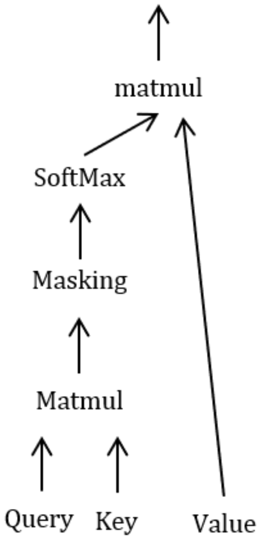


Figure 6: Attention Mechanism in Transformer

This is like a smart *key-value* lookup dictionary.

Attention is permutation invariant i.e., since these matrix multiplication occur simultaneously and not sequentially they don't care about position, but language is not! So we encode position of each word and add it to the token embedding as well.

The o/p from different layers are first concatenated and then normalized using layer norm, layerNorm(concat(N, attention(x)) and then a residual or skip connection is employed wherein the data before this layer is taken and added to the o/p of this layer to avoid vanishing gradients.

```python
def forward(self, x):
  # creating an attention matrix
    B,T,C = x.shape
    k = self.key(x)   # (B,T,C)
    q = self.query(x) # (B,T,C)
    # compute attention scores ("affinities")
    wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B, T, T)
    wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
    wei = F.softmax(wei, dim=-1) # (B, T, T)
    wei = self.dropout(wei)
    # perform the weighted aggregation of the values
    # basically this wei Tensor is our
    # `ATTENTION` tensor which multiplied with value gives output vector
    v = self.value(x) # (B,T,C)
    out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
    return out
```

Listing 6: Attention Mechanism Code

Following this sub-layer, a simple feed-forward network is used to learn how to create affinities between words properly.

```python
self.net = nn.Sequential(
    nn.Linear(n_embd, 4 * n_embd),  # layers like these work better
    nn.ReLU(),
    nn.Linear(4 * n_embd, n_embd),
    nn.Dropout(dropout),
```

Listing 7: Feed Forward part

This masking ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Attention Equation:

The attention equation in the Transformer model is defined as follows:

$$\texttt{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \texttt{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^\top}{\sqrt{d_k}}\right) \cdot \mathbf{V}$$

where:

- $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are the query, key, and value matrices, respectively.
- d_k is the dimension of the key vectors. Here we neglected this.

Multi-Head Attention:

Instead of performing a single attention function with keys, values, and queries, it's beneficial to linearly project them N times with different learned linear projections. We then perform the attention function in parallel, yielding N v-dimensional matrices that are concatenated, and dot product is performed to get the final output having the same size as the input vector.

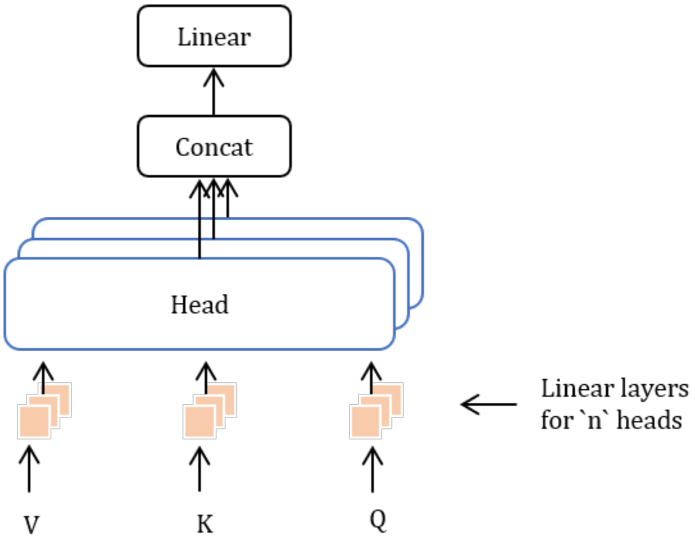


Figure 7: Multi-Head Attention