

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:

$$X(t+1) = b_0 + b_1X(t-1) + b_2X(t-2)$$

Here, $X(t+1)$ is the prediction, b_0 , b_1 , and b_2 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

```
1 # one hot encoded x values, input to the network
2 xenc = F.one_hot(xs, num_classes=27).float()
3 # gradient descent
4
5 for k in range(1):
6     # forward pass
7     logits = xenc @ W
8     # basically the probability of next value
9     loss = F.cross_entropy(logits, yb)
10
11     W.grad = None
12     loss.backward()
13     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., ($\langle . \rangle$, 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.

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 from torch.optim import SGD
5 class CharLanguageModel(nn.Module):
6     def __init__(self, vocab_size, n_embd, n_hidden, block_size, max_steps, batch_size):
7         super().__init__()
8         self.vocab_size = vocab_size
9         self.n_embd = n_embd
10        self.n_hidden = n_hidden
11        self.block_size = block_size
12        self.max_steps = max_steps
13        self.batch_size = batch_size
14        self.chars = 'abcdefghijklmnopqrstuvwxyz'
15        self.stoi = {s:i for i,s in enumerate(self.chars)}
16        self.itos = {i:s for s,i in self.stoi.items()}
17        self.C = torch.randn((vocab_size, n_embd), requires_grad=True)
18        self.fc1 = nn.Linear(n_embd * block_size, n_hidden)
19        self.bn = nn.BatchNorm1d(n_hidden)
20        self.fc2 = nn.Linear(n_hidden, vocab_size)
21        self.optimizer = SGD(self.parameters(), lr=0.1)
22
23    def forward(self, Xb):
24        emb = self.C[Xb]
25        embcat = emb.view(emb.shape[0], -1)
26        hprebn = self.fc1(embcat)
27        hprebn = self.bn(hprebn)
28        h = torch.tanh(hprebn)
29        logits = self.fc2(h)
30        return logits
31
32    def backward(self, loss):
33        self.optimizer.zero_grad()
34        loss.backward()
35        self.optimizer.step()
36
37    def train_model(self, Xtr, Ytr):
38        for i in range(self.max_steps):
39            ix = torch.randint(0, Xtr.shape[0], (self.batch_size,))
40            Xb, Yb = Xtr[ix], Ytr[ix]
41            logits = self.forward(Xb)
42            loss = F.cross_entropy(logits, Yb)
43            self.backward(loss)
44            if i % 10000 == 0:
45                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	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	aa	ab	ac	ad	ae	af	ag	ah	ai	aj	ak	al	am	an	ao	ap	aq	ar	as	at	au	av	aw	ax	ay	az
114	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
97	ca	cb	cc	cd	ce	cf	cg	ch	ci	cj	ck	cl	cm	cn	co	cp	cq	cr	cs	ct	cu	cv	cw	cx	cy	cz
516	da	db	dc	dd	de	df	dg	dh	di	dj	dk	dl	dm	dn	do	dp	dq	dr	ds	dt	du	dv	dw	dx	dy	dz
3983	ea	eb	ec	ed	ee	ef	eg	eh	ei	ej	ek	el	em	en	eo	ep	eq	er	es	et	eu	ev	ew	ex	ey	ez
80	fa	fb	fc	fd	fe	ff	fg	fh	fi	fj	fk	fl	fm	fn	fo	fp	fq	fr	fs	ft	fu	fv	fw	fx	fy	fz
108	ga	gb	gc	gd	ge	gf	gg	gh	gi	gj	gk	gl	gm	gn	go	gp	gq	gr	gs	gt	gu	gv	gw	gx	gy	gz
2409	ha	hb	hc	hd	he	hf	hg	hh	hi	hj	hk	hl	hm	hn	ho	hp	hq	hr	hs	ht	hu	hv	hw	hx	hy	hz
2489	ia	ib	ic	id	ie	if	ig	ih	ii	ij	ik	il	im	in	io	ip	iq	ir	is	it	iu	iv	iw	ix	iy	iz
71	ja	jb	jc	jd	je	jf	jg	jh	ji	jj	jk	jl	jm	jn	jo	jp	jq	jr	js	jt	ju	jv	jw	jx	iy	jz
363	ka	kb	kc	kd	ke	kf	kg	kh	ki	kj	kk	kl	km	kn	ko	kp	kq	kr	ks	kt	ku	kv	kw	kx	ky	kz
1314	la	lb	lc	ld	le	lf	lg	lh	li	lj	lk	ll	lm	ln	lo	lp	lq	lr	ls	lt	lu	lv	lw	lx	ly	lz
516	ma	mb	mc	md	me	mf	mg	mh	mi	mj	mk	ml	mm	mn	mo	mp	mq	mr	ms	mt	mu	mv	mw	mx	my	mz
6763	na	nb	nc	nd	ne	nf	ng	nh	ni	nj	nk	nl	nm	nn	no	np	nq	nr	ns	nt	nu	nv	nw	nx	ny	nz
855	oa	ob	oc	od	oe	of	og	oh	oi	oj	ok	ol	om	on	oo	op	oq	or	os	ot	ou	ov	ow	ox	oy	oz
33	pa	pb	pc	pd	pe	pf	pg	ph	pi	pj	pk	pl	pm	pn	po	pp	pq	pr	ps	pt	pu	pv	pw	px	py	pz
28	qa	qb	qc	qd	qe	qf	qg	qh	qi	qj	qk	ql	qm	qn	qo	qp	qq	qr	qs	qt	qu	qv	qw	qx	qy	qz
1377	ra	rb	rc	rd	re	rf	rg	rh	ri	rj	rk	rl	rm	rn	ro	rp	rq	rr	rs	rt	ru	rv	rw	rx	ry	rz
1169	sa	sb	sc	sd	se	sf	sg	sh	si	sj	sk	sl	sm	sn	so	sp	sq	sr	ss	st	su	sv	sw	sx	sy	sz
483	ta	tb	tc	td	te	tf	tg	th	ti	tj	tk	tl	tm	tn	to	tp	tq	tr	ts	tt	tu	tv	tw	tx	ty	tz
155	ua	ub	uc	ud	ue	uf	ug	uh	ui	uj	uk	ul	um	un	uo	up	uq	ur	us	ut	uu	uv	uw	ux	uy	uz
88	va	vb	vc	vd	ve	vf	vg	vh	vi	vj	vk	vl	vm	vn	vo	vp	vq	vr	vs	vt	vu	vv	vw	vx	vy	vz
51	wa	wb	wc	wd	we	wf	wg	wh	wi	wj	wk	wl	wm	wn	wo	wp	wq	wr	ws	wt	wu	wv	ww	wx	wy	wz
164	xa	xb	xc	xd	xe	xf	xg	xh	xi	xj	xk	xl	xm	xn	xo	xp	xq	xr	xs	xt	xu	xv	xw	xx	xy	xz
2007	ya	yb	yc	yd	ye	yf	yg	yh	yi	yj	yk	yl	ym	yn	yo	yp	yq	yr	ys	yt	yu	yv	yw	yx	yy	yz
160	za	zb	zc	zd	ze	zf	zg	zh	zi	zj	zk	zl	zm	zn	zo	zp	zq	zr	zs	zt	zu	zv	zw	zx	zy	zz

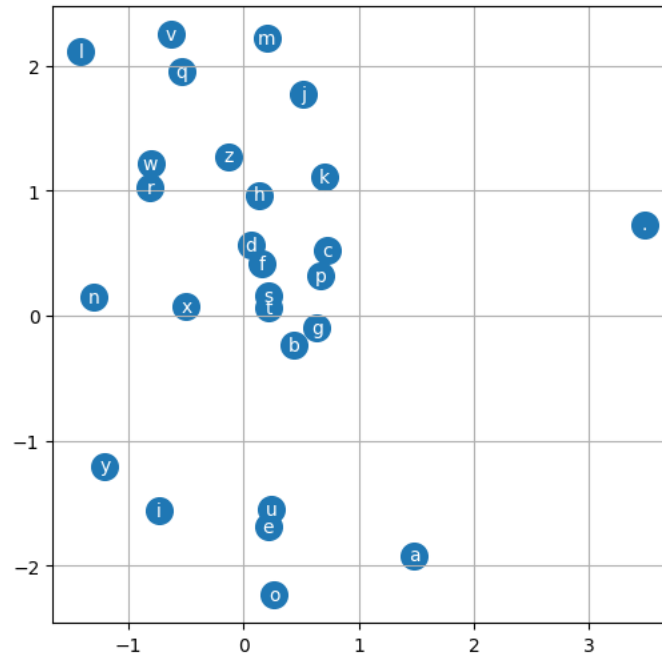


Figure 2: 2D Embedding

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

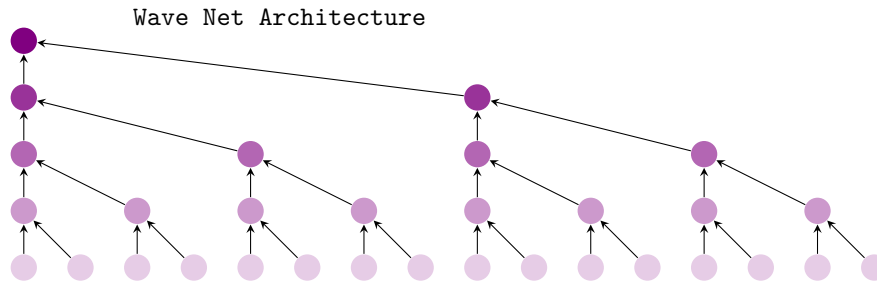
Previously, we used '3' characters, embedded them, squashed them together, and fed them all into 'tanh'. This potentially squashed too much information too quickly, causing the model to stagnate.

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.



```

1 # hierarchical network
2 n_embd = 24 # the dimensionality of the character embedding vectors
3 n_hidden = 128 # the number of neurons in the hidden layer of the MLP
4 model = Sequential([
5     Embedding(vocab_size, n_embd),
6     FlattenConsecutive(2), Linear(n_embd * 2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh()
7     ,
8     FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh()
9     ,
10    FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh()
11    ,
12    Linear(n_hidden, vocab_size),
13 ])

```

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.
raziaah.
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

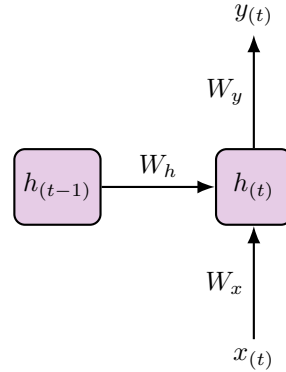


Figure 3: Recurrent neural network

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.

```
1 self.start = nn.Parameter(torch.zeros(1, N_EMBD2)) # the starting hidden state
2 self.wte = nn.Embedding(VOCAB_SIZE, N_EMBD) # token embeddings table
3 self.xh_to_h = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
4
5 # embed all the integers up front and all at once for efficiency
6 emb = self.wte(idx) # (b, t, N_EMBD)
7
8 # sequentially iterate over the inputs and update the RNN state each tick
9 hprev = self.start.expand((b, -1)) # expand out the batch dimension
10 hiddens = []
11 for i in range(t):
12     xt = emb[:, i, :] # (b, N_EMBD)
13     xh = torch.cat([xt, hprev], dim=1) # add rows horizontally
14     ht = torch.tanh(self.xh_to_h(xh)) # (b, N_EMBD2)
15     hprev = ht
16     hiddens.append(ht)
17
18 # decode the outputs
19 hidden = torch.stack(hiddens, 1) # (b, t, N_EMBD2)
20 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 $z = (U_z.x_t + W_z.h_{t-1})$ 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 $r = (U_r.x_t + W_r.h_{t-1})$ 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 $h_t = \tanh(U_z.x_t + W_s.(h_{t-1} \odot r))$ 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

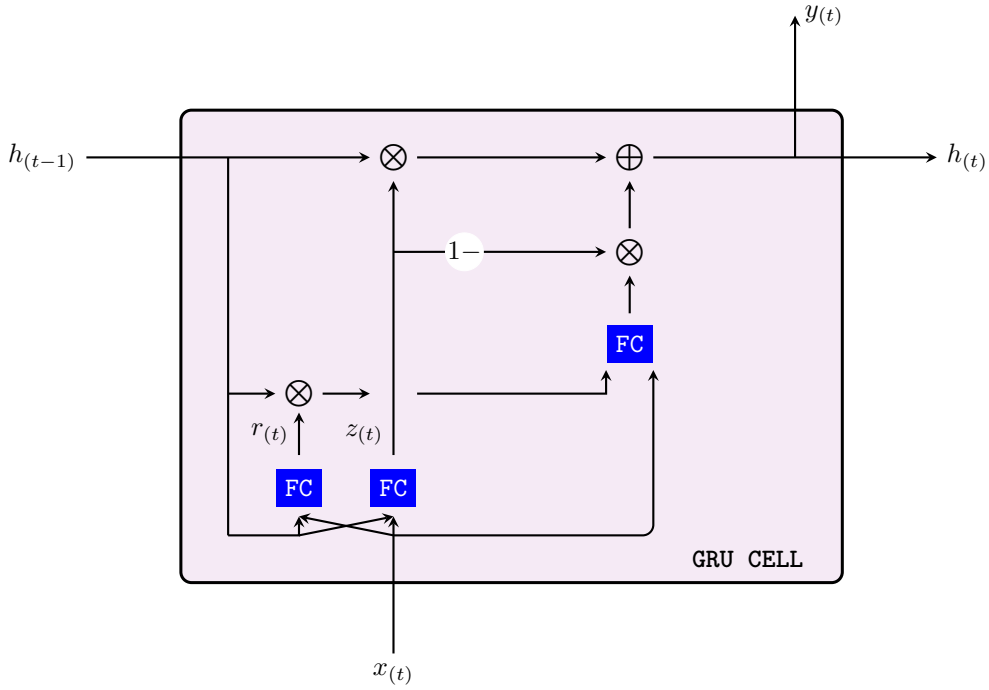


Figure 4: Gated Recurrent Unit

```

1 # input, forget, output, gate
2 self.xh_to_z = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
3 self.xh_to_r = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
4 self.xh_to_hbar = nn.Linear(N_EMBD + N_EMBD2, N_EMBD2)
5 self.start = nn.Parameter(torch.zeros(1, N_EMBD2)) # the starting hidden state
6 self.wte = nn.Embedding(VOCAB_SIZE, N_EMBD) # token embeddings table
7 # first use the reset gate to wipe some channels of the hidden state to zero
8 # embed all the integers up front and all at once for efficiency
9 emb = self.wte(idx) # (b, t, N_EMBD)
10
11 # sequentially iterate over the inputs and update the RNN state each tick
12 hprev = self.start.expand((b, -1)) # expand out the batch dimension
13 hiddens = []
14 for i in range(t):
15     xt = emb[:, i, :] # (b, N_EMBD)
16     xh = torch.cat([xt, hprev], dim=1)
17     r = torch.sigmoid(self.xh_to_r(xh)) # reset gate - squashing to zero
18     hprev_reset = r * hprev
19     # calculate the candidate new hidden state hbar
20     xhr = torch.cat([xt, hprev_reset], dim=1)
21     hbar = torch.tanh(self.xh_to_hbar(xhr)) # new candidate hidden state
22     # calculate the switch gate that determines if each channel should be updated at all
23     z = torch.sigmoid(self.xh_to_z(xh))
24     # blend the previous hidden state and the new candidate hidden state
25     ht = (1 - z) * hprev + z * hbar # (b, N_EMBD2)
26     hprev = ht
27     hiddens.append(ht)
28     # decode the outputs
29     hidden = torch.stack(hiddens, 1) # (b, t, N_EMBD2)
30     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.

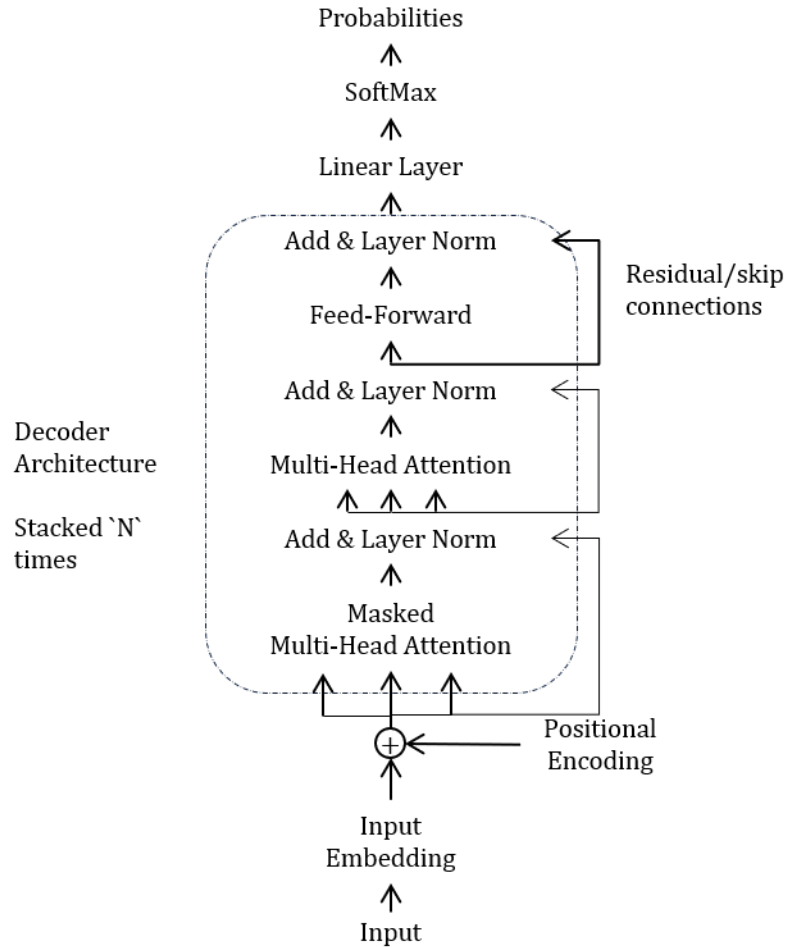


Figure 5: Transformer, Decoder only architecture

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 .

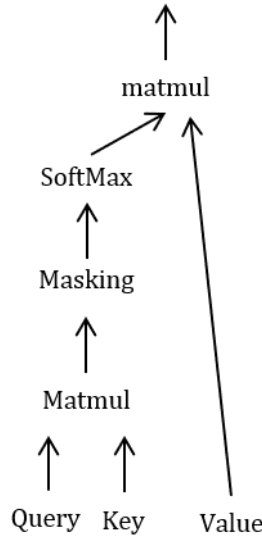


Figure 6: Attention Mechanism in Transformer

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(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.

```

1 def forward(self, x):
2     # creating an attention matrix
3     B,T,C = x.shape
4     k = self.key(x) # (B,T,C)
5     q = self.query(x) # (B,T,C)
6     # compute attention scores ("affinities")
7     wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B, T, T)
8     wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
9     wei = F.softmax(wei, dim=-1) # (B, T, T)
10    wei = self.dropout(wei)
11    # perform the weighted aggregation of the values
12    # basically this wei Tensor is our
13    # `ATTENTION` tensor which multiplied with value gives output vector
14    v = self.value(x) # (B,T,C)
15    out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
16    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.

```

1 self.net = nn.Sequential(
2     nn.Linear(n_embd, 4 * n_embd), # layers like these work better
3     nn.ReLU(),
4     nn.Linear(4 * n_embd, n_embd),
5     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:

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

where:

- Q , K , and 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.

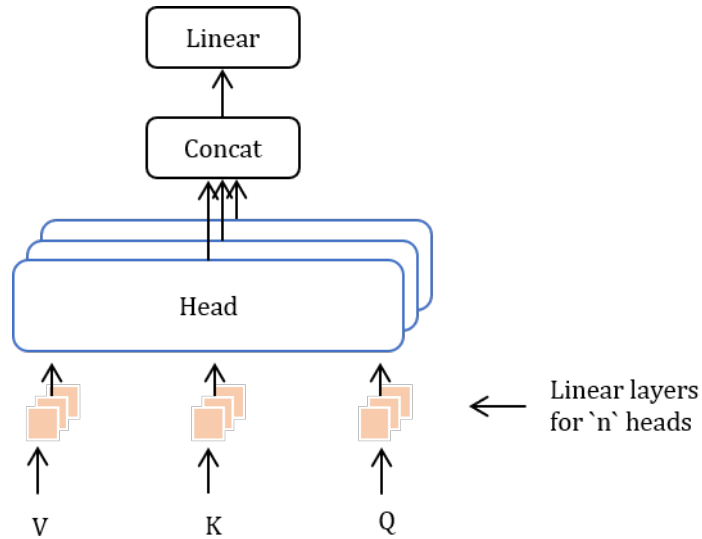


Figure 7: Multi-Head Attention