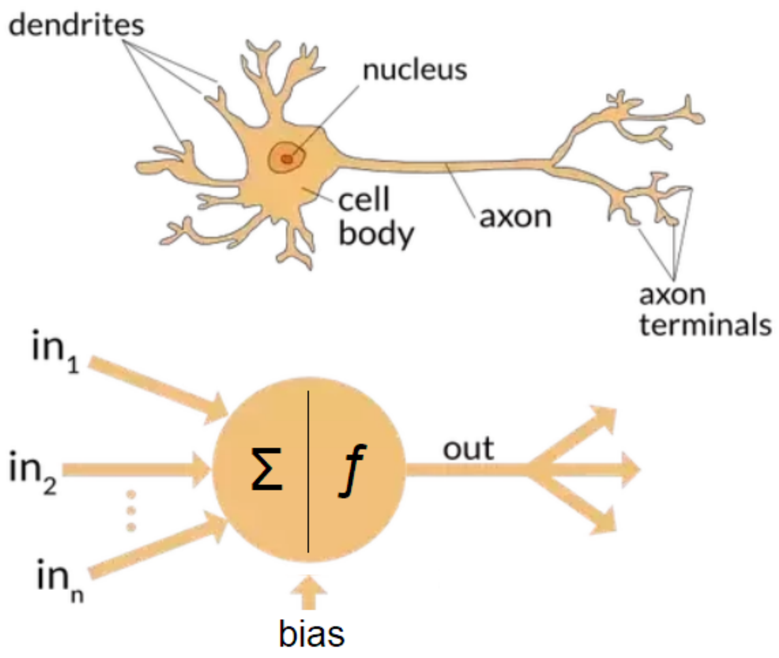
The Quest For Neural Networks Heroship

Becoming a neural networks “hero” is a lofty goal to achieve in 13 hours of lecture, but an enticing offer for someone looking to venture into the world of machine learning. After spending the last summer helping incorporate AI-powered tools into the workflow at the company I interned with, I was eager to learn more about the history of artificial intelligence and the implementation of neural networks. Andrej Karpathy, an influential AI researcher and one of the founding members of OpenAI, offers a free lecture series called Neural Networks: Zero to Hero, which I decided to study. While I’m certainly no hero yet, the course provided me with a strong foundational understanding of the core concepts that comprise neural networks, as well as meaningful practice implementing these in PyTorch. In this post, I intend to share my learning from the course, focusing on concepts and leaving out most of the math and code. The aim of this post is not to act as a substitute for Andrej’s incredibly detailed lectures, but to document the knowledge I’ve gained and serve as a primer for other beginners to the topic. If you have the time, I highly recommend working through the course (the total lecture time is about 13 hours, but it took me closer to 30 hours, accounting for time spent note-taking, coding and replaying difficult sections).

Defining Key Terms

Before starting the lecture, I primed myself with some introductory research. This paper was especially helpful in providing context. Below are some of the basics:

A **neural network** is any collection of neurons arranged in such a way that their interactions yields fruitful results. While neural networks aren’t specific to machine learning by definition (the human brain is the original neural network), the term is now most often associated with the artificial neural network. Artificial neural networks (ANN) were pioneered when Warren McCulloch and and Walter Pitts developed an algorithm to imitate human brain activity in the 1940s (more on the history of neural networks here).



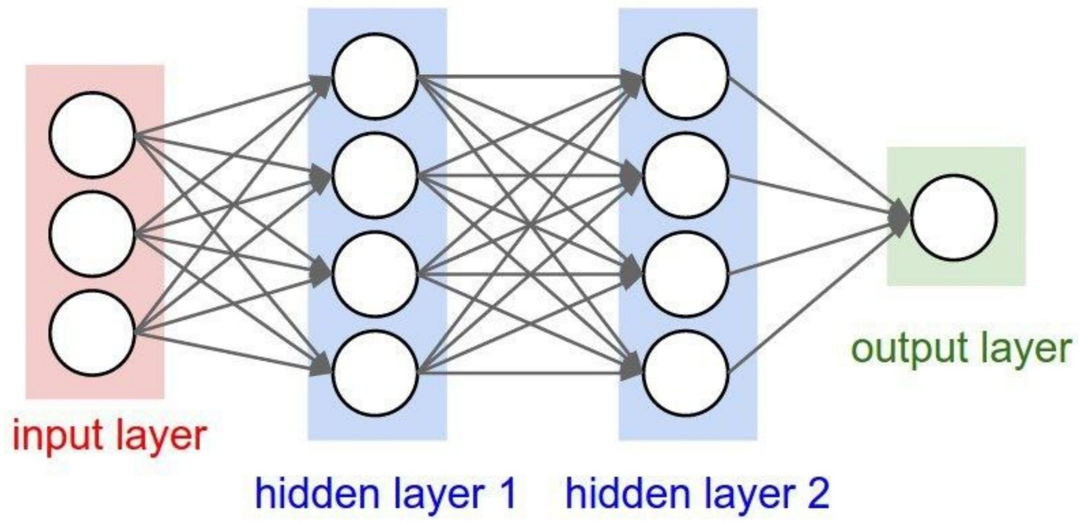
Neurons, artificial and biological. Source

Neurons in an ANN are, in essence, mathematical equations. They take a collection of inputs, perform some operation on these inputs, and produce a single output, which can then be transmitted to other neurons. Neurons in an ANN are similar to nerve cells in many ways:

- both receive many input signals, assign weights to these, and process them
- both transmit a single output to receivers when enough input is present

- data processing occurs on a local level (within each neuron)
- “memory” is distributed throughout the system (long-term memory is stored in synapse weights, short-term memory in impulses)
- both networks' synapse connections are strengthened by experience
- certain groups of neurons getting activated will cause other neurons to fire.

In an ANN, neurons are arranged into **layers**, where the collection of neurons receiving inputs are the **input layer**, the intermediary neurons that do most of the processing are the **hidden layer**, and the final layer producing a result is the **output layer**. Neural networks form conclusions and make decisions by assigning **weights** to each input, which determines how much impact each input has. Weights and **biases** are the parameters that, through trial and error, are adjusted during training until the network arrives at a correct model.



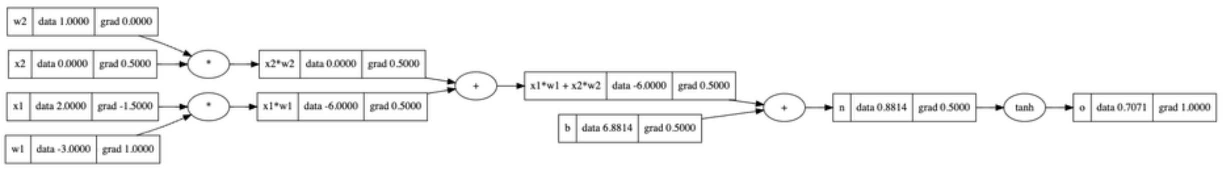
A simple neural net. Source

So, the defining characteristics of an ANN are its **architecture** (how neurons are arranged in layers and interconnected), the **training algorithm** (i.e. the technique for calculating and adjusting weights on connections), and its **activation function** (the function that determines whether a neuron should be activated or not based on the given inputs). The activation in the neurons of the output layer will determine the model’s “answer” to whatever question was posed. Activation functions often also normalize output to the proper desired range (i.e. 0 to 1).

Backpropagation as a Training Method

With some basic terms defined, we can now examine how these networks are trained at a baseline level. I mentioned that neurons are basically mathematical expressions. In fact, their most important characteristic in the context of training a network is that any operation that takes place in a neuron must be differentiable. This allows us to calculate **gradients** at each neuron with respect to the network’s **loss function**. The loss function measures the overall performance of a neural network by comparing target outcomes and actual outcomes, so the goal is always to minimize the loss function. Gradients represent the impact that each parameter has on the loss function, and are calculated using local derivatives at each neuron and the chain rule. The gradient provides information that allows the network to adjust these parameters to decrease the network’s loss. Simply put, we use derivatives to measure what impact each neuron has on the output. If the impact aligns with our target outcome, we increase that neuron’s impact, otherwise we reduce that neuron’s impact (using weights and biases). This is how the model learns.

To get a gradient at each neuron in the network, we need some way to calculate the derivative of each parameter with respect to its impact on the loss function. Using a process called **backpropagation**, this process is relatively simple.


w2 | data 1.0000 | grad 0.0000
x2 | data 0.0000 | grad 0.5000
x1 | data 2.0000 | grad -1.5000
w1 | data -3.0000 | grad 1.0000
*
*
x2*w2 | data 0.0000 | grad 0.5000
x1*w1 | data -6.0000 | grad 0.5000
+
x1*w1 + x2*w2 | data -6.0000 | grad 0.5000
b | data 6.8814 | grad 0.5000
+
n | data 0.8814 | grad 0.5000
tanh
o | data 0.7071 | grad 1.0000

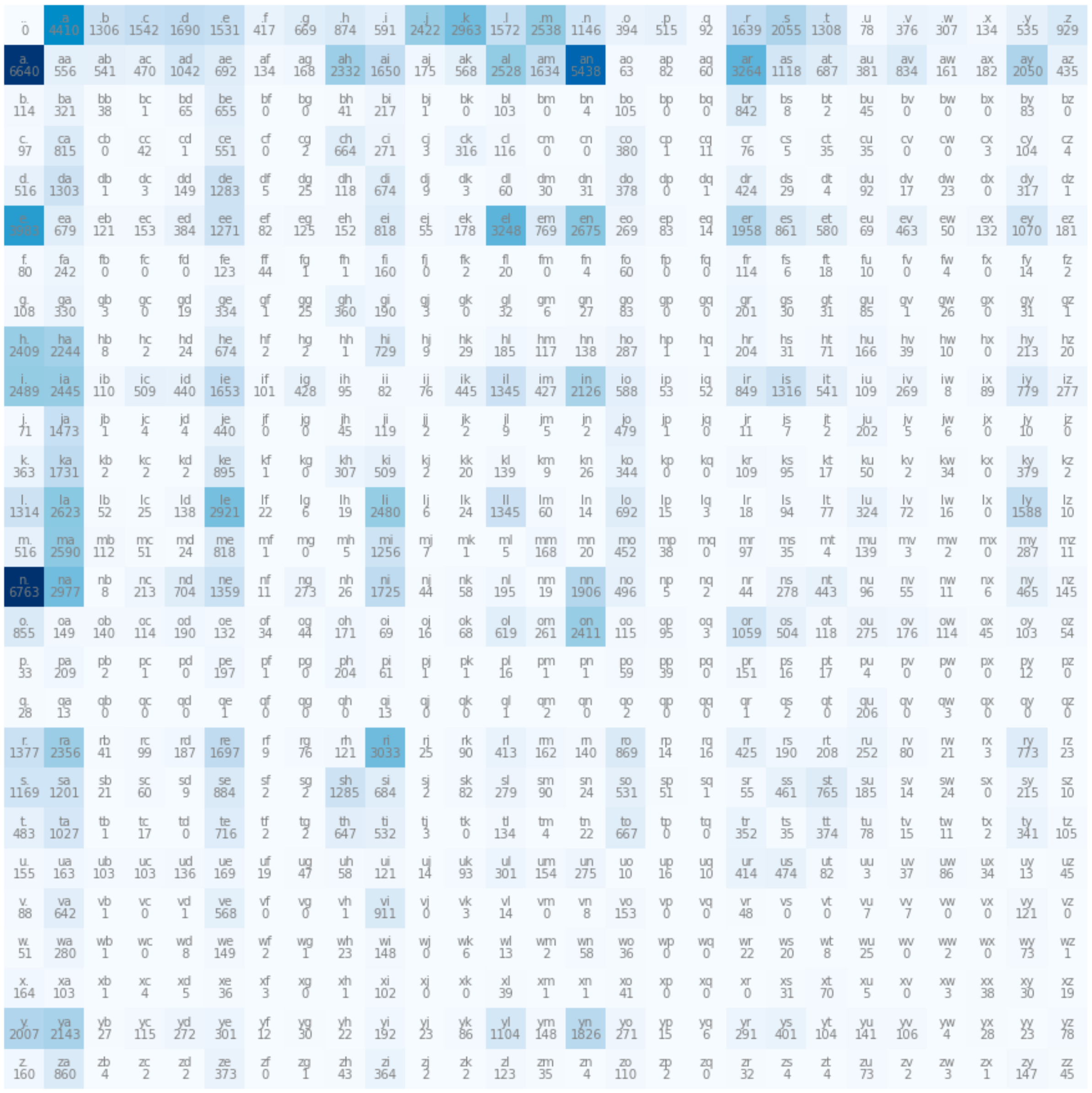
Take this neuron from Andrej's first lecture, which has two inputs (`x1` and `x2`), two weights (`w1` and `w2`), one bias `b` and the activation function `tanh`. The derivative of output `o` with respect to itself is `1`. That's our base case. Now we can go back one step; `do/dn` is the derivative of `tanh(n)` at `n = 0.8814`, which is equal to `1-tanh(0.8814)^2 = 0.5`. So we can fill in the gradients of `o` and `n` with `1` and `0.5`, respectively (these are already filled in in the diagram above). The next four gradients are simple to calculate: since each of the four values are being added, the local derivative is `1` for each of them. We use the chain rule with the ongoing overall derivative, multiplying `0.5*1` in each case to get a gradient of `0.5`. To get the leftmost gradients, we calculate local derivatives `d(w2)/d(x2w2) = x2 = 0`, `d(x2)/d(x2w2) = w2 = 1`, `d(w1)/d(x1w1) = x1 = 2`, and finally `d(x1)/d(x1w1) = w1 = -3`. Now we can get the gradient by multiplying the local derivative by the output's gradient, so `w2`'s gradient is `0*0.5=0`, `x2`'s is `1*0.5=0.5`, `w1`'s gradient is `2*0.5=1` and `x1`'s is `-3*0.5=-1.5`.

There are many steps involved in calculating gradients and implementing backpropagation, but the arithmetic is simple. We begin with the rightmost local derivative, then we recursively iterate left, or backpropagate, multiplying each node's local derivative by its output's gradient. These gradients are the core element we need to train our neural network. This is an isolated example in one neuron, but we now know that if we need the final output `o` to increase, we can achieve this by making the weight `w1` or the bias `b` more positive. These parameters can then be recursively adjusted until the output reaches the desired value. Backpropagation is the core mechanism that enables the training of neural networks. Using backpropagation, the parameters in a neural network can essentially be randomly initialized, and they will eventually converge to the correct values during model training.

A single-layer neural network

As mentioned above, a neural network can be as straightforward as a single layer of neurons that modify inputs by applying some weights and some mathematical expression. To understand how these are implemented, Andrej uses an example problem, where the goal is to generate more names or name-like strings using a model that is trained on a long list of existing names. This problem can be solved using a neural network, but a neural network isn't the easiest solution. We can also use a probability distribution matrix and a character-level **bigram** language model, where we store pairs of subsequent characters and their corresponding frequencies, then use these to calculate the probability that one letter will follow another. With this probability distribution, we can make reasonably accurate guesses about which letter should follow the current letter; for example, the letter `z` is highly unlikely to follow the letter `q`, so the language model would almost never pick `z` next when the current letter is `q`. Conversely, the letter `a` is a common starting letter for names, so the model should assign a high probability that it follows the "start" character `.` (which is also the end character in this implementation and the graph below).

Below is a graph of these frequencies from a large dataset of words used in the exercises. Each cell denotes the frequency that the second character follows the first character in the dataset. If we divide each cell by the sum of its row (the total number of times the first character appeared in the dataset), we have a probability distribution showing how likely it is that each of the second characters were to follow the first character. Using this probability distribution is one way to generate words. For each character, starting at the first, we use the probability distribution to "pick" the next character, so characters that appeared next to each other in the dataset are likely to appear next to each other in the output.



.. 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

Distribution matrix from the second lecture

This is relatively sound theoretically, but you can imagine that choosing a character only based on the previous output isn't a very effective method. This is also not a neural network – since we aren't training the model, we're simply using existing probabilities to guess what a good output might be. Here are some sample "names" the model generated in the second lecture:

```
mor.
axx.
minaymoryles.
kondlaisah.
anchshizarie.
```

These names are indisputably not very name-like. However, in order to improve our output, we must be able to quantify its quality in some standard way. To do this, we'll use the concept of **likelihood**; how likely an output is to be produced given a model's probability distribution. So the likelihood of something like `emma`, which is in the dataset, should be reasonably high and something like `xqxzi` which certainly isn't in the dataset, should be near-zero. The likelihood of the dataset is the product of all the individual probabilities that each element in the dataset could be generated. To make this more stable and easier to work with, instead of multiplying all of the probabilities, we sum the `log` of the individual probabilities, giving us the **log likelihood**. Since probabilities are between 0 and 1, making the logs of the probabilities negative, we negate their sum (the log likelihood) and use the **negative log likelihood**. To arrive at our final measure for output quality (the loss function), we divide the negative log likelihood by the number of values in the dataset and are left with an average negative log likelihood, which is a metric to describe how likely the model would be to reproduce the training set.

We can use this to create a neural network with the bigram language model. To train it, we follow a process similar to the one in the first example, iteratively backpropagating and modifying the weights in accordance with their gradients, thus minimizing the average negative log likelihood. To codify inputting a character into the model to calculate the next character, we use “one-hot” encodings of each character, where an integer is encoded as a matrix where each value is zeroed-out except the the value at the index corresponding to the integer’s value, which is 1. So the letter `d` which has index `3` would be represented as the matrix

`[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`

Each element in this vector represents the position for one of the 27 characters of the alphabet, plus an extra for the start/stop character `.`. To create the foundation for training the model, we can create a randomly-initialized 27-by-27 matrix representing the weights: 27 neurons, each takes 27 inputs. So when we multiply the input matrix (1x27) by the weights matrix (27x27), we get a 1x27 output matrix. Each element in the output matrix represents the **logit** (log probability) for that character, and is typically “squished” using the softmax activation function, to represent a probability, in this case the probability of this character being chosen.

At the beginning, these outputted probabilities won’t be accurate or useful. However, all of the operations described above are differentiable, so we can follow a similar process as done in the first example to tune the weights. To train the network, we iteratively backpropagate and modify the weights in accordance with their gradients with respect to the loss function, thus minimizing the average negative log likelihood. Eventually, the weights matrix will converge to a more “correct” matrix that does a better job of assigning probabilities for the next character given an input matrix representing the current character.

Unfortunately, since the bigram model only takes into account the previous character, even a neural network trained on lots of data will output mediocre results (in fact, once it’s fully trained, it will be an equivalent model to the earlier, counting-based probability model using frequencies). To improve these results, we must increase the size of context we are considering, by adding more layers to our neural network. The important takeaway is that we are able to start off with a completely random collection of weights, assign these weights to modify input characters, and get probability estimates for the next character. Then, simply by calculating the derivatives of each parameter with respect to the output’s accuracy, we were able to iteratively improve the weights until we had a model that performed as well as it could given only a single character as context. This example also introduces one of the central challenges to implementing a neural network: how best to represent the input in a numerical format that can be fed into the network as input. In this example, a vector with a one-hot encoding worked well, but this encoding may vary greatly from network to network.

Building a Multi-Layer Perceptron

The next step in creating a more intelligent model is to give it more knowledge or context. To build a multi-layer perceptron, we will expand on the bigram model by adding two more layers and slightly altering our method for representing inputs.

To provide more context to the model, we now define a **block size**, or how many previous characters are given to determine the next character (in “real” large language models, the block size would determine how many words or chunks of words are used to predict the next). Our training set is now the set of all substrings (of `block_size` length) of names in the list of words, and the label for each substring is the character following it. For simplicity, let's assume `block_size = 3`.

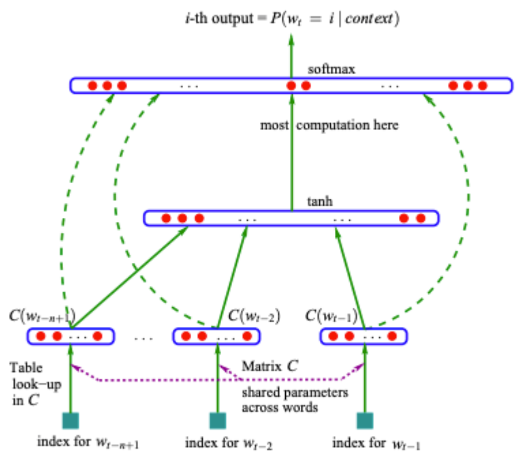
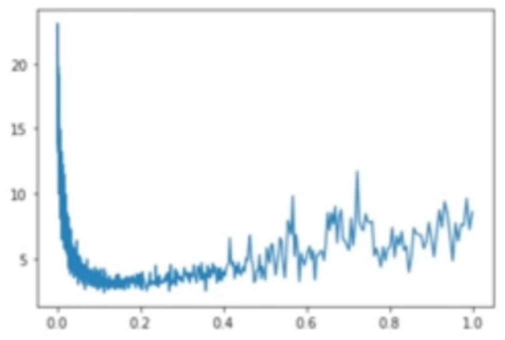


Diagram of a Multi-Layer Perceptron (Source)

Another feature introduced in this section is an **embeddings layer**, where we define an embeddings lookup table that stores an embedding for each character, thus converting each input (the 3-character string) to an `n`-dimensional embedding. This is done to normalize inputs and codify their values in a standard way. The embeddings are then adjusted as the network trains. Our new network's architecture is as follows: the embeddings layer outputs embeddings that squash the inputs into a pre-defined space. Then, this embedding is the input to the hidden layer, the size of which we can choose. A design choice like the number of neurons in the hidden layer is called a **hyperparameter**, and is left to the developer to define and later optimize to improve the model's performance. The hidden layer must have the same number of outputs as each neuron in the output layer takes as inputs. Finally, the output layer will have 27 neurons, one for each character. The hidden layer and the output layer each have a matrix of weights, which are used to process the inputs and are modified to train the model. The logits in the output layer are then exponentiated and normalized to sum to 1, so they can be used as probabilities.

This model is ready to train. However, we now have over 10,000 parameters and our dataset has more than 30,000 values. Calculating the output, backpropagating through each parameter, and adjusting the weights for each parameter is very computationally expensive, and while feasible for a network of this size, is not scalable. Instead, we choose **minibatches**, or smaller subsets of the dataset, to train the model. So for each pass, we choose a random, fixed-size subset of the data, and use the loss function with respect to that data to tune the weights. The size of the minibatch is another hyperparameter; the tradeoff for the developer is the compute time per pass vs. the noise introduced by using incomplete datasets to train the model.

We must also choose a reasonable **learning rate**, which is the multiplier used to multiply the gradient to adjust weights. With a very small learning rate, the model will learn at an unwieldy, slow rate. With a very high learning rate, the loss will be unstable, as the parameters are being over-adjusted each pass through the network. To find a good learning rate, we can plot the learning rate against the loss, and find a rate at which the loss is decreasing consistently without much noise. In the graph below, the learning rate at 0.1 seems to be a good choice. **Learning rate decay** is another common practice, where the learning rate is decreased at a later point in training when the loss is already small, and the initial learning rate has become too high.



Learning rate and loss. From lecture 3

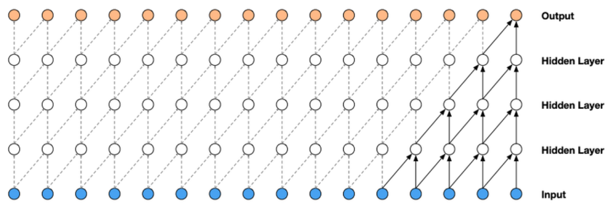
Additional concepts that are key to be cognizant of when training a neural network are **overfitting** and **underfitting**. Overfitting occurs when the neural network is trained *too much* on a certain dataset, and starts to essentially memorize that data. Then, when sampling from this neural network, it will too closely reproduce the data that it was fed instead of producing useful “new” data. Underfitting is the opposite, and takes place when the neural network hasn’t learned enough from the data. To mitigate and measure these phenomenons, we split our dataset into a **training set** (about 80% of the data), a **development set** (about 10% of the data), and a **test set** (the last 10% of the data). The training data is the only set that should be used to train the model. The development set should then be used to evaluate the network’s loss, since this data hasn’t been seen yet by the network. If the loss on the training data is much lower than the loss on the development data, the network is likely overfitted; it is performing much better on data it is familiar with. If the losses are approximately the same, the model is likely underfitted; it hasn’t learned enough, and the developer should likely tweak the hyperparameters (train longer, increase the size of the network, tweak the learning rate, etc.).

Improving and Stabilizing the Multilayer Perceptron

The model described above is functional and leaves us with relatively low loss, but there is some wasted effort inherent to its architecture, and other potential weaknesses which should be addressed. Since all of our weights are randomly initialized, the output probabilities will initially predict highly incorrect outcomes. The loss calculated on these initial values will be very high, much higher than the loss would be with evenly distributed probabilities (i.e. where each next character is equally likely to be predicted). This occurs when the weights are initialized with large random numbers, outputting very confident predictions, despite not having been trained to output confidently *correct* predictions. This can be mitigated by initializing the weights matrix for the output layer with values very close to 0, so that each logit in the output is near-zero, producing output probabilities that are near-equal. This simple step leaves us with a significantly lower initial loss, which then reduces total computation. Instead of wasting computation correcting the initial predictions, the model will approach accurate weights much faster. As the model learns, these near-zero weights will be adjusted to approximate their correct values.

Another improvement to the model is facilitated by **batch normalization**. This is another layer that is placed between hidden layers to scale inputs between layers, so that anomalies don’t take place. We’ve already discussed an embeddings layer which normalizes the inputs, but activations sent as inputs from one layer to the next are left raw. In batch normalization, these intermediate values are scaled and shifted using their standard deviation and mean. This addresses the issue of **saturated neurons**, where an activation function for a layer fails to output values evenly distributed along its range (i.e. evenly distributed between 0 and 1), but instead tends toward the extremes, creating hyperactive neurons and dead neurons that have no learning power at all (lots of 0s and lots of 1s). To normalize a batch, we center the pre-activation values, or the values that feed into the activation function, using the mean and the standard deviation of each batch to scale and shift the activations from the previous layer. The goal is to create a roughly Gaussian shape for the distribution of data inputted to the next layer.

Even with these stabilizing measures, the multi-layer perceptron described is not very deep, and thus has limited learning capabilities. One factor in the current implementation that makes it difficult to create deeper networks is our method for processing inputs; currently, we are squashing all of our input into the neural network at the input layer simultaneously. This mutes the data and doesn’t allow for each piece of input to influence the outcome as much as it can. Another improvement to the model is the concept of the WaveNet, where smaller chunks of input data are concatenated, and these chunks are combined piece-by-piece.



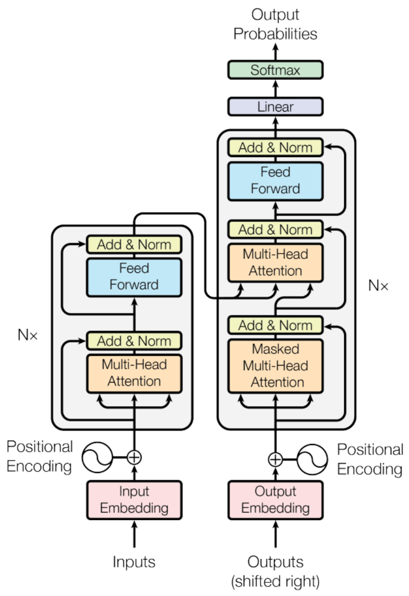
From the WaveNet paper

Such a model enables an even deeper neural network, where each input element can have a greater impact. These improvements represent only a subset of optimizations to neural networks; this is its own field of research and requires many more dedicated posts to fully discuss. I will focus the rest of this post on lecture 7 (the most viewed lecture in the series) and the transformer architecture introduced there, which has revolutionized neural network implementation.

The Transformer

The transformer architecture was first introduced in the 2017 paper Attention Is All You Need, and forms the foundation for famous natural language models like ChatGPT (GPT = generative pre-trained transformer), as well as a long list of models beyond natural language processing. The basic idea is that training data should be able to communicate with other training data and decide which data to value more. “Attention” is the idea that the tokens are aware of each other’s existence and can focus on the more important ones. In this way, information can be gathered in a data-dependent way, where some pieces of training data are more valuable than others.

With the transformer architecture, small chunks, or batches, of the dataset are separated and used for training piece-by-piece, with a fixed **batch size**. The training data in language models is commonly referred to as a **token**, which is typically a chunk of characters smaller than an average word (a token in the name-prediction example would be a single character). Previously, all of our context information would inform the following prediction with equal weight or value: any character in the series of three characters in our context blocks was equally important. Now, with attention, a token can express affinities for other tokens, where a certain past token might factor in more heavily than others ~~in~~ when calculating a prediction. This replicates real-world sentence formulation more accurately. For example, when choosing the correct verb in a sentence, the subject of the sentence should be a more important factor than the article the sentence started with. An **attention head** is where the the weighted sums representing relationships between tokens are calculated. Usually, multiple attention heads are computed in parallel, allowing the model to encode different types of relationships between tokens. It is important to note, however, that tokens within a batch can only communicate with themselves; there is no communication between batches of data.



Architecture from Transformer Paper

The transformer paper suggests an architecture that stacks “attention layers” with multilayer perceptrons. In each attention layer, attention vectors are used to represent contextual relationships between tokens in a batch. Multiple attention vectors are used in multi-head attention layers, which are then combined using a weighted average. This setup creates a deeper neural network, which is capable of fitting our data much more closely, effectively learning more and incorporating context in a more powerful way.

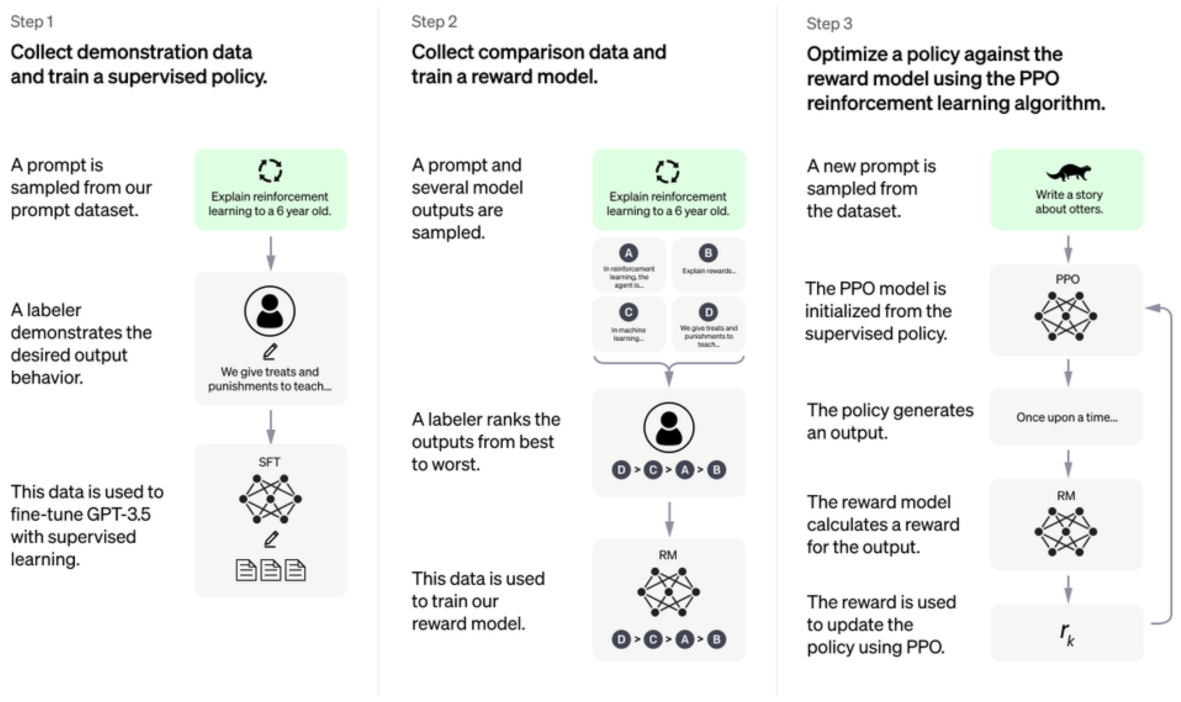
Training Neural Networks

By the end of lecture 7, the viewer has coded alongside Andrej to build a simplified model of a GPT which is capable of generating a specified number of tokens in a manner similar to the data it was trained on (i.e. generating Shakespeare-like content given Shakespeare’s work to train on). The hyperparameters are entirely configurable, so the developer is able to choose the learning rate, training iterations, number of embeddings, number of layers, the number of heads, head size, the batch size, the block size, and the dropout rate. When choosing these hyperparameters, I was tempted to push the size of the network ever higher – the bigger the better, right? Near the end of the video, Andrej mentions that the example network he trained took fifteen minutes to train on his high-powered GPU, and warns that anyone unequipped with a dedicated GPU shouldn’t attempt to train that network. I tried it anyway, but gave up when my MacBook grew alarmingly warm after six hours of CPU time and the training was still only halfway done.

This was quite interesting to me, as I had heard of a GPU before, but never thought to look into their use or the difference between a GPU and a CPU. Graphics processing units were originally developed to assist in rendering image or video, or performing other operations which require highly parallel mathematical/graph operations. GPUs have thousand of cores, which make them especially adept at performing lots of small operations in parallel, which is exactly what training a neural network requires. CPUs (core processing units) have between two and sixty-four cores, and are far more generalist than GPUs, primarily performing operations serially. For this reason, training a neural network using a CPU is exceptionally slow and prohibitively expensive depending on the size of the network. Considering the massive datasets and network sizes used to train production-ready neural networks such as ChatGPT, it’s easy to imagine how massive computation costs are for participating companies, and how much attention is given to training techniques and choosing well-suited hardware.

Pre-training vs. Fine-tuning

Lastly, an important note: training a very deep neural network with lots of language data will not produce a highly-capable AI assistant like ChatGPT without more work. The process described above and discussed throughout the lecture series is known as “pre-training”, where a model is fed massive amounts of language data and trained to generate more, similar data. However, a model that spits out random generated data that approximates its training data isn't very useful. The model must be "fine-tuned" to match its intended purpose. A chat-model must be able to answer questions and converse, a code generator must be able to generate code based on a prompt, and so on. OpenAI released the diagram below explaining their fine-tuning process in this blog post introducing ChatGPT. The fine-tuning process requires far more human intervention, including ranking outputs, creating a reward model which rewards high-quality responses, and the utilization of reinforcement learning. This process is nuanced, and is not nearly as open-source as the pre-training stage is. This is where the neural network is converted from a raw neural network to a highly commercial product that can be sold and monetized, so logically these techniques are kept secret.



Step 1
Collect demonstration data and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain reinforcement learning to a 6 year old.
A labeler demonstrates the desired output behavior.
We give treats and punishments to teach...
This data is used to fine-tune GPT-3.5 with supervised learning.
SFT
Step 2
Collect comparison data and train a reward model.
A prompt and several model outputs are sampled.
Explain reinforcement learning to a 6 year old.
A labeler ranks the outputs from best to worst.
D > C > A > B
This data is used to train our reward model.
RM
Step 3
Optimize a policy against the reward model using the PPO reinforcement learning algorithm.
A new prompt is sampled from the dataset.
Write a story about otters.
The PPO model is initialized from the supervised policy.
PPO
The policy generates an output.
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Wrapping Up

We've come a long way from backpropagation to a full-fledged natural language model with configurable hyperparameters, layer normalization, skip connections, self-attention, and multi-layer perceptrons. If this introduction piqued your interest, following the lecture series is the best way to expand on more difficult concepts and create your own implementation of a neural network. I will likely continue to write about neural networks, and am taking a class on the topic this coming semester, so stay tuned for more posts on the topic! Thanks for taking the time to read.