INTUITIONS WHILE BUILDING A LANGUAGE MODEL:

USING BIGRAMS

24:10

→ Follow probabilities and sample

→ [. + w + .]

→ We created a 2D tensor (matrix) with rows and columns that have the frequencies of each bigram occuring.

→ 27 x 27 Matrix

→ convert them to probabilities by creating a probability vector

→ Remember to convert the frequencies into floats because we will normalize them.

↳ P/P.sum()

↓

Probability of any single character to character (be the first ~~word~~ of the word.

→ "SAMPLING" is the process of generating new data points (eg. words, sentences or images) based on a learned probability distribution

→ ① Deterministic Sampling

② Random sampling

→ To sample from the distributions we usually use Torch.Multinomial
↳ returns sample from the multinomial probability distribution: you give me probabilities and I will give you integers that are sampled according to this Prob. Distr.

→ Multinomial probability distribution is a distribution that models multiple outcomes, each with its own probability.

→ In a language model, each token in the vocabulary can be an outcome. no. of possible outcomes = "K"
Each outcome i has a fixed probability p_i, where

$$\sum_{i=1}^{K} p_i = 1$$ (probabilities must sum to 1)

→ The Probability Mass Function (PMF) of the multinomial distribution is:

$$P(X_1 = x_1, X_2 = x_2 \cdots X_K = x_K) = \frac{n!}{x_1! x_2! \cdots x_K!} \prod_{i=1}^{K} p_i^{x_i}$$

n = no. of trials
K = No. of outcomes
x_i: count of outcome i
p_i: Probability of outcome i

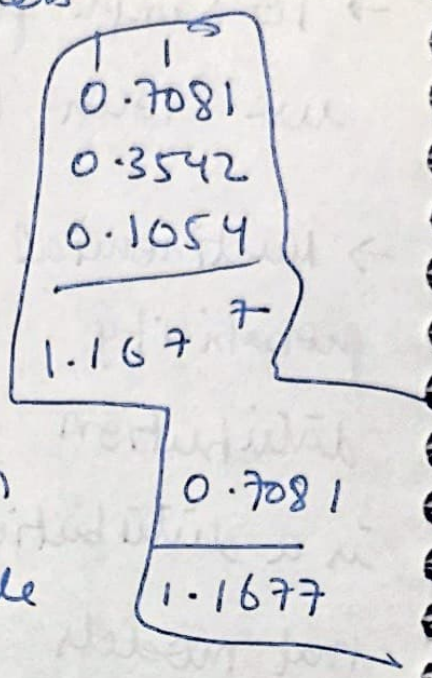
→ Total counts equals no. of trials

→ torch. generator : related randomness

→ replacement = True ? why ?

when we draw an element, we can draw it and then we can put it back into the list of eligible indices to draw again

⇒ (Refer to 29:31 and see the code for more details)



→ for efficiency purpose, prepare a matrix P, that will have the probabilities in it.

→ Same as Frequency Matrix (N) but every single row will have the row of probabilities that is normalized to 1 indicating the probability distribution for the next character given the character before it as defined by which row we are in.

→ keepdim = True ? why ?

. It is used to maintain shape of the tensor for tensor operations in future.

→ Can we divide a tensor of 27, 27 by 27,1
↳ It is fine (Broadcastable)

→ BROADCASTING SEMANTICS

TWO TENSORS CAN BE combined in a Binary operation like division when

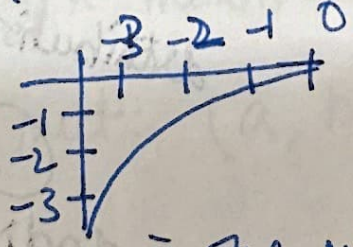
→ Each tensor has atleast 1 dimension
→ when iterating over the dimension sizes, starting at the trailing dimension, the dimension size must either be equal one of them is 1, or one of them does not exist.

(GOTHROUGH BROADCASTING SEMANTICS) **

→ How, quality of model in a single number?
loss?

• Maximum likelihood estimation
↓
Product of all the probabilities
↳ higher, the better

• We use log likelihood, which lowers the probability, lowers the log likelihood.e.



• log likelihood is the sum of all probabilities

$\log(a*b*c) > \log a + \log b + \log c$

→ negative log likelihood = − (log likelihood)
(nll)

→ normalized log likelihood is "nll/n".
(the lower the better)

~~Defining the parameters~~

- GOAL: Maximize likelihood w.r.t the model parameters (statistic modeling)
  - equiv. to maximizing log likelihood (monotonic)
  - equiv. to minimizing the negative log likelihood
  - equiv. to minimizing the average negative log likelihood.

↓

→ NEURAL NETWORKS APPROACH

some weights (W)
↑

- Receives single character as I/P → N.N → O/P next character (guess)

Probability distribution

→ create the training set (x, y)

x = inputs
y = labels

gradient based optimization

→ deffeeeee b/w torch.tensor & torch.Tensor

① ②

→ ① data type is int64 ② data type is float32

→ How all we going to feed in these examples into Neural Network?

~ common way of encoding integers is called one-hot encoding

→ one-hot encoding converts an integer into a long vector of zeros except the integer dimension which is converted into "1". and then that vector will be fed into neural network.

" TORCH.NN.FUNCTIONAL.ONE_HOT" does this

↳ num_classes is how long do you want your vector to be

→ Also make sure the dtype = float32, vector do a .float()

→ @ = matrix multiplication operator

→ matrix mut: (a, b) (b, c)

↳ o/p (a, c) apna tc

→ each neuron does a simple operation wx+b

(x, c @ W) (3, 13) inputs, neuron, column

= By dot product

0.5 → fire rates

creating a probability matrix for N·N

⇒

→ Now we are going to interpret these 27 numbers is that these 27 numbers are giving us log counts.

→ To get counts we will get log counts and exponentiate them.

→ logits = log counts

counts = probability of characters appearing

↳ logits.exp()

```
probs = counts / counts.sum(1, keepdims = True)
probs
```

This is basically Softmax

O/p LAYER — SOFTMAX ACTIVATION FUNCTION — PROB

$$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} \rightarrow \left[\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \right] \rightarrow \begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

→ Now to the last practice is vectorize the losses.

loss = -probs[torch.arange(n), ys].log().mean()

n = no. of bigrams

→ In pytorch the best practice to set initial gradient to zero is by,

W.grad = None [1:40:50]

→ Tells us the influence of that weight on loss function.

→ We look into [W.data += -50 * w.grad]

Note 1:

→ Weights in this are basically the same as count matrix and are log counts

→ W.exp() is count matrix

→ [Go through Note 2:]

BUILDING MAKEMORE PT2 : MLP

→ We are going to be building a multilayer perceptron using the research paper Bengio et al. 2003.

→ We are building a character level language model, in the paper they have a vocabulary of 17,000 words and they built a word level. We are using the same intuition to build a character level.

* What happens in the paper?

→ They take every one of the 17000 words and they are going to associate to each word a 30 dim feature vector

(~~very word~~ is 17,000 words in 30 dimensional vector space)

→ Tune the actual out embeddings using back propagation.

(The vectors will move around in space, the synonyms or matching words might end up near in the space)

→ Use MLP to predict words. using prev words.

→ Maximize the log-likelihood of training data

build dataset

- block size is the content length: how many characters do we take to predict the next one

- X, Y = [], []

  ↓ ↳ labels (The next character to predict)

  i/p

→ refer to 11:44

logic:

stoi = {'.': 0, 'a': 1, 'c': 2, 't': 6}

itos = {0: '.', 1: 'a', 2: 'c', 6: 't'}

initial: [0, 0, 0] (all (.))

character 'c'

ix = stoi[ch] = 2

Addpair: [0,0,0] ---> 2

Update: [0,0,2]

character 'a'

ix = stoi[ch] = 1

Addpair: [0,0,2] → 1

update: [0,2,1]

character 't'

ix = stoi[ch] = 6

Addpair: [0,2,1] ---> 6

update: [2,1,6]

character '.'

ix = stoi[ch] = 0

Add pair: [2,1,6] ---> 0

update: [1,6,0]

result for "cat.":

X: [[0,0,0], [0,0,2], [0,2,1], [2,1,6]]

Y: [2,1,6,0]

GOAL: To predict the next character using the 3 past characters (CONTEXT).

GOAL: Building a ~~lookup table~~ that will take X [[3,2,3]] and pred. Y.
→ We are creating a look matrix called C value Y.
↳ [building the lookup table]
We are going to squeeze in

2-D ~~vector~~ space. (-7, 2)

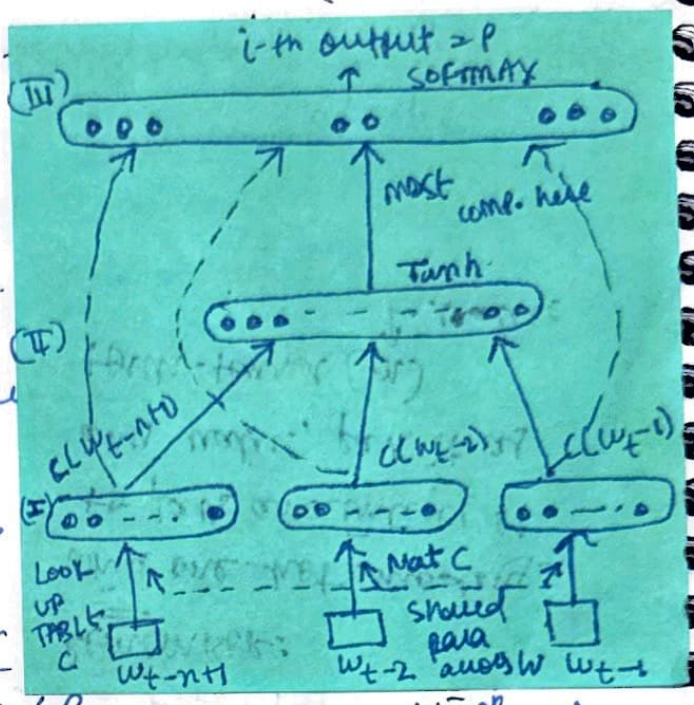
27 ~~characters~~ will have a 2

→ The goal here is to create

a 2D vector.

we have created a look up

embedded matric = C = C



IMPLEMENTING THE HIDDEN LAYER

3, 2-D embeddings as shown (I)

→ Initialize W_1 weights randomly

number of inputs will be 6

number of neurons 100

biases will be 100 of them

→ for neuron, ⊕ in hidden layer the operation should be emb @ W_1 + b_1 → This is not possible because,

emb matrix shape = ([32,3,2])

W_1 shape = ((6,100)) # no matching dims

Then how? convert ([32,3,2]) → [32,6]

to do the Mat mul.

→ use torch.cat ? concentate the 3 embeddings in 'I' (refer to stick note)

" embedded the 3 ; 2D spaces."

⇒ torch.cat([emb[:,0,:], emb[:,1,:], emb[:,2,:]], 1) . shape
↑ over the 1 dimension

⇒ output: torch.size([32,6])

→ But what if Block-size = 5?

TORCH.UNBIND // → sliced output

// → efficient way ↓ a.view (look into this)

↓

in the computer memory any tensor is stored in 1D.

take the matrix that you want to convert and give the dims.

Ex: emb.view(32,6).

→ h = ~~emb~~ torch.tanh(emb.view(emb.shape[0], 6) @ W1 + b1

↑

here we are implementing the II layer with tanh function as activation function. (the dimensions are adjusted now)

shape: [32,6] //.

IMPLEMENTING THE OUTPUT LAYER

→ here we will take 2nd set of weights and bias where W_2 = torch.rand((100, 27)) → one output

↑ input from prev layers

b_2 = torch.rand(27) — same

→ output will be

logits = h @ W_2 + b_2

logits # output

↳ shape = ([32, 27])

IMPLEMENTING THE NEGATIVE LOG LIKELIHOOD LOSS

① exponentiate the logs [counts]

② figure out the probability — across 1 Dimension

= counts / counts.sum(1, keepdims = True)

③ ACTUAL LAYER

we have layer 'y' which we created during dataset preparation which has all the labels.

→ We will move index into the rows of 'prob' and each row we will pluck out the prob assigned to the correct character.

- we will do this by first taking an iterator which

torch.arange(32) → This will initialize 32 numbers which we will use as row values. and then, to pluck out the 'Y' labels which are integers we will do,

* prob [torch.arange(32), Y]

This will pluck out all the 'Y' probabilities.

→ Now, to get the loss (negative log loss likelihood)

(3) Loss = -prob (torch.arange(32), Y).log().mean()
(1)

INTRODUCING F.CROSS_ENTROPY (to counter our loss calculation).

so all of it can be packaged into

*. F.cross_entropy (logits, Y)

WHY (1)? (1) Memefficient forward and backward passes

(2) much more numerically well behaved.

IMPLEMENTING THE TRAINING LOOP, OVERFITTING ONE BATCH:

→ Refu to code in the notebook (grash makemoreptz.ipynb)

there's an ~~error~~ Naiser named (TRAINING LOOP)

→ We are overfitting here because the number of parameters (3481) is higher than the examples (32).

→ ~~The Reason we are not able to achieve zero loss~~

is because

TRAINING ON THE FULL DATASET, MINIBATCHES

→ Perform FWD, BACK and update on minibatches of data.

→ We will randomly select some portion of the dataset and thats a minibatch- We will do equations on that and iterate on batches.

→ TO select the learning rate, the perfect one we use `torch.linspace` to initiate a lr

of learning rates and then exponentiate them.

```
lre = torch.linspace(-3,0,1000)
lrs = 10**lre
```

→ We pass these to the iterator loop and append and store these values in arrays.

```
lri = []
lossi = []
```

→ We then refer to the graph and select a proper learning rate depending on where the loss is low and stable.

→ at the end, we decay the learning rate by exponentiate it (eg: if lr was 0.1, we do it 0.01) to get the best ~~lr~~ and lowest loss.

SPLITTING UP THE DATASET INTO TRAIN/VAL/TEST SPLITS AND WHY

BUILDING MAKEMORE PT3: ACTIVATIONS & GRADIENTS, BATCH NORM:

① FIXING THE INITIAL LOSS:

→ In training of NN, it is almost always the case that you will have a rough idea for what loss to expect at initialization. and that depends on the loss func^n & the problem setup.

→ Usually if the loss is unpredictably high or low or abnormal, its because the logits have extreme values.

② FIXING THE SATURATED FUNCTION (tanh)

→ Tanh is a squashing function, it takes arbitary numbers and squashes b/w −1 & 1.

what happens to tanh during backprop?

→ Remember, if your distribution tanh has extreme values, like more of −1's and 1's, you are locked.

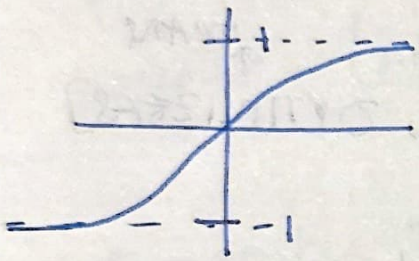
→ During the backpropagation of tanh, it has to go through the following

self.grad += (1 − t**2) * out.grad

when the values are extreme i.e 1 or -1, the gradient will be completely '0', leading to lot issues.

→ if it is the tanh is more of '0', then the gradients directly pass through with their original values.

→ The more you are in the flat tails (-1 or 1) ~~to~~ then the more the gradient gets squashed / destroyed.



→ Same is true for sigmoid, ReLU.

③ CALCULATING THE INIT SCALE: "KAIMING INIT"

→ Until now we have fixed and tweaked the weights and biases to limit the losses and get better results. We have done it manually.

→ Standard deviation is the measure of the spread of gaussian.

→ How do we initialize weights so that these activations take on reasonable values throughout the network? DELVING DEEP INTO RECTIFIERS : KAIMING HE

→ In short, for perfect initialization, the weights and biases are divided by $\sqrt{2/n_l}$.

→ This is implemented in pytorch.

torch.nn.init.kaiming_normal_

↓

most common way of initializing N.N.

[RESIDUAL CONNECTIONS, NORMALIZATIONS, OPTIMIZERS] ← ADAM

↳ Make sure Initializations are done properly

→ std = $\frac{\text{gain}}{\sqrt{\text{fan\_mode}}}$ → we have gain because in tanh and other functions, there is squashing, to maintain the shape we have gain.

gain for tanh = 5/3 (left to does pytorch)

④ BATCH NORMALIZATION (2015)

→ We dont want the pre-activation func^{n}s to be too small because tanh will not do anything and at the same time we dont want it to be too large because tanh will be saturated.

→ We want them to be roughly gaussian (mean = 0, stand. deviation = 1) at initialization.

→ Batch normalization: take hidden states and normalize them to be gaussian.

→ we will take the hidden activations, calculate their mean and standard deviations and divide them per batch.

hpreact = hidden activations.

hpreact = (hpreact − hpreact.mean(0, keepdim = True)) / (hpreact.std(0, keepdim = True)) —(I)

(0 → di)

→ Remember that we would want the hidden activations to be gaussian only at initialization, we dont want them to be forced to be gaussian always. we would like the Neural Net to be move this around to potentially make it more diffuse / sharp and we'd like the backprop to tell us how the distribution should move around.

→ to do that, we will take the normalized i/p and we are scaling them by some gain and offsetting them by bias.

from (I), we will first bngain = torch.ones((1, n-hidden))

bn bias = torch.ones((1, n-hidden))

hpreact = bngain * xi + bnbias (refer to research paper)

→ take Layer and append Batch normalization to control scale of activations / activations at every point of Neural Net.

→ will sprinkle B.N layers allow all the layers.

⇒ LINEAR LAYER
↓
BATCH NORM LAYER
↓
ACTIVATIONS

[torch.nn.LINEAR, BATCHNORM2D]

PYTORCH IFY4

→ CREATE A LINEAR LAYER (TORCH.NN) → LINEAR

↳ FAN IN: NO. OF INPUTS FEATURES
FAN OUT: NO. OF OUTPUTS FEATURES ↓ what is X?

$y = xA^T + b$

why fan-in **0.5 (Kaiming Init)

bias initialized to zeros (initially)

→ BATCH NORM will be done (BATCH NORM1D)

→ $wx+b$ (__called__)

→ $y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$ → Batch Normalization

torch.nn.BatchNorm1D(num_features, eps, momentum, affine=True, track_running_stats=True, device=None, dtype=None)

we must track γ & β after normalization)

* self.training = True, should be used so that while inference is running, it will be torch.no_grad works.

→ ~~γ~~ γ = Gamma (learnable scaling parameter)

β = Beta (learnable shifting parameter)

→ γ = allows the network to scale the normalized output back to a desired range.

β = allows the network to shift the normalized output to a desired mean.

$\gamma > 1$ (amplified)	$\beta > 0$ shifts data to the right (↑ mean)
$0 < \gamma < 1$ (compressed)	$\beta < 0$ shifts data to the left (↓ mean)
$\gamma < 0$ (inverted)	

$x_{scaled} = x \cdot \gamma$ $\quad$ $x_{shifted} = x + \beta$

Batch Normalization, after Normalization the i/p to have mean = 0, variance = 1, we apply shifting & scaling.

$$y = x_{normalized} \cdot \gamma + \beta$$

→ B-N is a technique used to normalize the activations of neurons within a layer during training.

- Improve training speed, stability
- convergence

BUILDING WAVENET (PART 5)

① (embedding
② Flatten)

→ lossi : is a python list of floats (losses)
↳ we

Reimplementing / Refactoring Forward Pass

→ embedding table is outside the layers

→ The emb.view is also outside the layers.

→ We will create these layers and add them to Layers list.

emb = C[Xb] : embedding operation

x = emb.view [emb.shape[0], -1] : flattening operation

write it in the layers.

class embedding:

* This is where embedding happens, remember that the look up table embeddings are also trainable parameters

* If a t e models and they appear in similar contexts. During training, the embedding vectors of 'a' & 'e' might move close to each other in embedding space because model minimizes the loss.

Remember ** there are multiple weights being upgraded,

① Linear layer weights

② Batch Normalization parameters (gamma, beta)

③ weight inactivation layers.

④ embedding weights

How are these weights updated?

⇒

→ F-W pass &
- opened through layers (linear, batchnorm, activations)

→ Loss computation
- o/p is compared to target
- loss function used to adjust weights

→ Backward pass
- gradients of the loss are calculated
- each weight is adjusted

→ SIGNIFICANCE OF EACH Weight

LINEAR LAYER weights:

P.T.O....

① ~~learn to minimize all the~~

→ Embedding weights
  → convert tokens into dense vectors that represent their meaning
→ Series weights
  → Transform these embeddings into richer.
→ Batch Norm
  • Normalize & scale the features to make training more stable & efficient
  • Stablize training by normalize the i/p ~~layer~~ to each layer
  • Prevent vanishing or exploding gradients
→ Output layer weights
combine features to predict the next token in sequence.

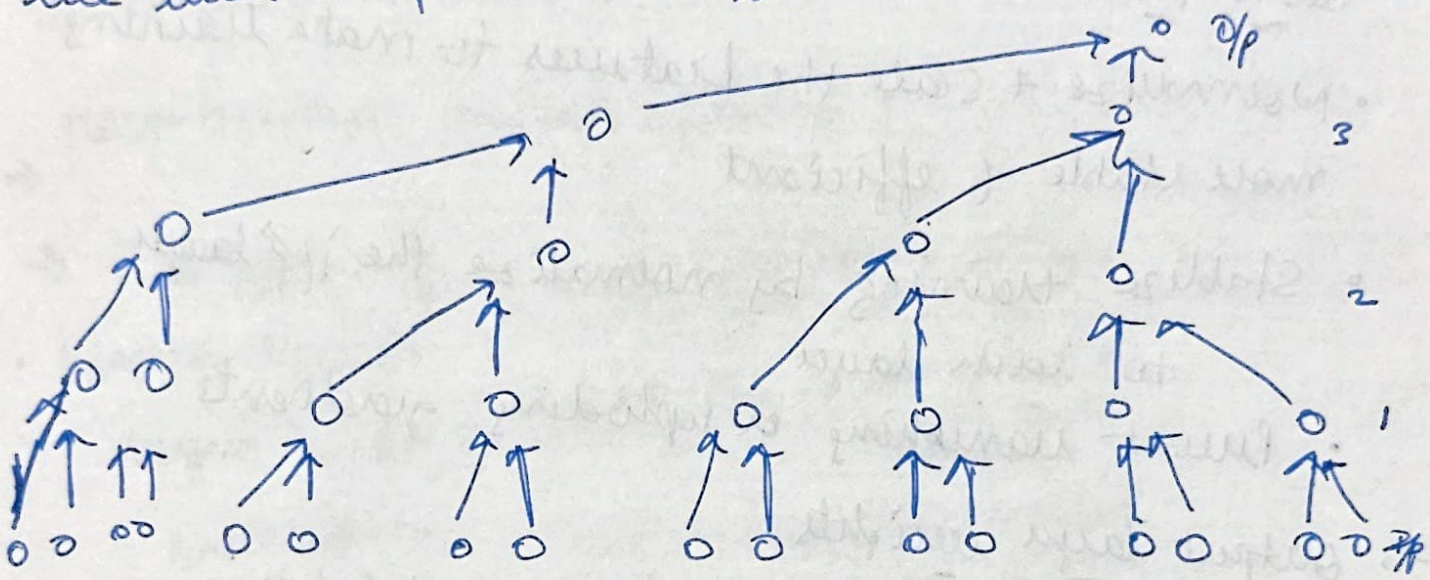
— — —

→ CONTAINERS ARE WAY OF ORGANIZING layers into lists or dicts.

→ SEQUENTIAL maintains a list of layers. in a sequential order.
  • output of one layer is input to next.
  • So Instead of defining & connecting each layer manually we do this.

SCALE UP THE NEURAL NETWORK:

- We are using Bengio-et paper.
- In Bengio-et paper there is a single layer where everything is squashed. Instead of that now we will implement a wavenet like structure.



Bigrams

4-grams

8-grams

Dilated causal convolution layers.

(Progressive fusion)

at

2 consecutive squashing