MLP basics: Building a Name Generator

Khoo An Xian, 2025

Credits to Andrej Kaparthy the GOAT for his amazing teaching!

Makemore

Given a list of names, generate more names by constantly predicting the next character

What we will do:

Get input & target datasets
xs & ys that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

*Frequencies = input*weight + bias*
All frequencies are initially random and wrong, and as we train the model to improve parameters, frequencies will become correct

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Sample from probability table
to get next char

1: Get training inputs

Dataset xs & ys
that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Optimise model
Improve the probabilities predicted in prev step

Sample from probability table
to get next char

Code

```
# import raw data
words = open('names.txt', 'r').read().splitlines()
```

```
# Create dataset
xs, ys = [], []
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1 = stoi[ch1]
        ix2 = stoi[ch2]
        xs.append(ix1)
        ys.append(ix2)

xs = torch.tensor(xs)
ys = torch.tensor(ys)
```

What's happening

```
words[:3]
>>> ['emma', 'olivia', 'ava']
```

```
.emma.
emma.

. e
e m
m m
m a
a .

xs: [., e, m, m, a]
ys: [e, m, m, a, .]
```

2: Build frequency table

Dataset xs & ys
that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Optimise model
Improve the probabilities predicted in prev step

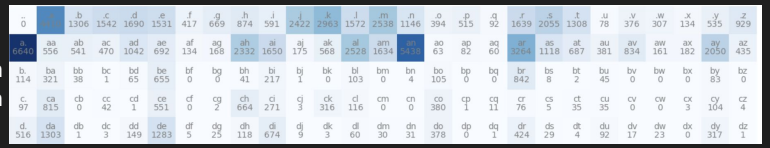
Sample from probability table
to get next char

Code

```
# One hot encoding of xs (5)
import torch.nn.functional as F
xenc = F.one_hot(xs, num_classes=27).float() # (5, 27)


# Initialise 1 neuron with 27 weights for 27 one-hot inputs
W = torch.randn((27,1), generator=g) # (27, 1)
xenc @ W # (5,27)@(27,1) -> (5,1), 5 activations of 1 neuron


# Initialise 27 neurons
W = torch.randn((27,27), generator=g) # (27, 1)
xenc @ W # (5,27)@(27,27) -> (5,27)
```

What's happening

Frequency table



For each of the 5 input chars, we have 27 neuron outputs. This corresponds to the frequency distribution for the 27 possible next chars

- num neurons = vocab size, each neuron outputs probability associated with 1 char
- `(xenc @ W)[3, 13]` *represents activation output of the 13th neuron looking at the 3rd input. (dot pdt btw 3rd row of xenc and 13th col of W)*

We will see that our output has positives and negatives. However we want frequencies that are always positive. Hence we interpret these as `log-counts`.

3: Build probability table

Dataset xs & ys
that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Optimise model
Improve the probabilities predicted in prev step

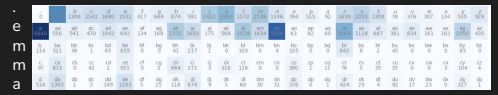
Sample from probability table
to get next char

Code

```
logits = xenc @ W # (5, 27)
counts = logits.exp() # (5, 27)
probs = counts/counts.sum(1, keepdims=True) # (5, 27)
# the last 2 lines are the SOFTMAX FUNCTION which gets
loss = - probs[torch.arange(num), ys].log().mean() #nll
```

What's happening

Frequency table



Log counts → (*exponentiate*) → **Counts** → (*divide by row sum*) → **Probability**

probs (5, 27) is a probability distribution for each of the 5 inputs

To evaluate the model, we are interested in seeing the probability assigned by the model to the correct character. Eg. emma has 5 egs (.e, em, mm, ma, a.)

- Eg 1, interested in 5th char's probability (e) --> probs[0, 5]
- Eg 2, interested in 13th char's probability (m) --> probs[1, 13]
- Eg 3, interested in 13th char's probability (m) --> probs[2, 13]
- Eg 4, interested in 1st char's probability (a) --> probs[3, 1]
- Eg 5. interested in 0th char's probability (.) --> probs[4, 0]

We get this through `probs[torch.arange(5), ys]`. (ys outputs tensor([5, 13, 13, 1, 0]))

Code: `probs[torch.arange(5), ys]`
Output probabilities
>> `[0.0057, 0.0258, 0.0227, 0.0404, 0.0266]`
Aka our model assigned probability of 0.0057 to 'e' for Ex 1 (too low! we want probability 1).

Output negative log likelihood
>> `-probs[torch.arange(5), ys].log().mean()`
We want to maximise the likelihoods for our correct chars. That's equivalent to minimising our neg log likelihood, which we use as our **LOSS**.

4: Putting things together

Dataset xs & ys
that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Optimise model
Improve the probabilities predicted in prev step

Sample from probability table
to get next char

Code

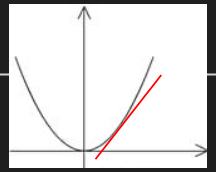
```
# Initialise Weights (27 neurons, each with 27 weights)
W = torch.randn((27,27), generator=g, requires_grad=True)

# Gradient descent
iterations = 50
for k in range(iterations):
    # Forward pass
    xenc = F.one_hot(xs, num_classes=27).float()
    logits = xenc @ W       # log-counts, (5, 27)
    counts = logits.exp()   # equivalent N (5, 27)
    probs = counts/counts.sum(1, keepdim=True) # probability distribution for each input (5, 27)
    loss = - probs[torch.arange(num), ys].log().mean() #nll
    print(loss.item())

    # Backward pass
    W.grad = None
    loss.backward() # back propagation thru operations to calc gradients of weights

    # Update
    W.data += -50 * W.grad
```

What's happening

Back propagation

1. We have the loss L, and we want to minimise it
2. `loss.backward()` – Calculate the derivative of loss wrt all the weights w in our model (dL/dw)
3. `W.data+= -50*W.grad` – Move the weights a little bit in the direction of - gradient

Do this for many `iterations`



5: Sampling from the model

Dataset xs & ys
that store consecutive character pairs that occurred in our names list

Build frequency table
that shows for each char, the number of times it is followed by characters . to z

Build probability table
that shows for each char, the probabilities of char . to z occurring next

Optimise model
Improve the probabilities predicted in prev step

Sample from probability table
to get next char

Code

```
# Sampling from the model
num_words = 5
for i in range(num_words):
    out = []
    ix = 0
    while True:
        # previously: p=P[ix]

        xenc = F.one_hot(torch.tensor([ix]), num_classes=27).float()
        logits = xenc @ W # generate log-count using trained models
        counts = logits.exp()
        p = counts/counts.sum(1, keepdim=True)

        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item() # sample based on p
        out.append(itos[ix])
        if ix == 0:
            break

    print(''.join(out))
```

What's happening

Sample from probability p, take 1 sample at a time and join them together

Samples

```
cexze.
momasurailezityha.
konimittain.
llayn.
ka.
```

Makemore part 2

Building a multi-layer model

Set up: *We are working with a vocab_size of 27 characters, to be embedded into a 2d space.*

- **Input**
  Blocks: If we take a block_size = 3 (context window has 3 chars) , then we have 3 inputs
- **Embedding layer**
  Each character will get a 2 dimensional embedding vector. Embedding table is a 27 by 2 matrix
- **Hidden layer**
  *100 neurons, fully connected to the 6 numbers that make up the 3 words (3*2)*
- **Output layer**
  27 neurons to output 27 logits (one for each possible next char), fully connected to the 100 neurons above.
- **Softmax layer**
  The 27 logits are exponentiated and divided to make probabilities

1: Get training inputs

Input
Indexes of incoming block. If we take a block_size = 3 , then we have 3 inputs

Embedding table
27 by 2 matrix. Each character will get a 2d embedding vector

Hidden layer
100 neurons, fully connected to the 6 numbers that make up the 3 characters

Output layer
27 neurons, each fully connected to the 100 neurons above, output 27 logits

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Code

```
# Build training dataset (inputs X, targets Y)
def build_dataset(words):
    block_size = 3 # num chars in context
    X, Y = [], []
    for w in words[:]:
        #print(w)
        context = [0]* block_size
        for ch in w + '.':
            ix = stoi[ch]
            X.append(context)
            Y.append(ix)
            #print(''.join(itos[i] for i in context), '-->', itos[ix])
            context = context[1:] + [ix] #crop context and put next char in

    X = torch.tensor(X)
    Y = torch.tensor(Y)
    print(X.shape, Y.shape)
    return X, Y
```

Making examples (eg. from emma)

```
... --> e
..e --> m
.em --> m
emm --> a
mma --> .
```

Code

```
# Train Validation Test split
import random
random.shuffle(words)
n1 = int(0.8*len(words))
n2 = int(0.9*len(words))

Xtr, Ytr = build_dataset(words[:n1]) # 80%
Xdev, Ydev = build_dataset(words[n1:n2]) # 10%
Xte, Yte = build_dataset(words[n2:]) # 10%
```

Training set: 182437 examples, 3 chars in input, 1 char in target

```
Xtr([182437, 3]) Ytr([182437])
```

Validation and testing sets:

```
Xdev([22781, 3]) Ydev([22781])
Xte([22928, 3]) Yte([22928])
```

2: Embedding table

Input
Indexes of incoming block. If we take a block_size = 3 , then we have 3 inputs

Embedding table
27 by 2 matrix. Each character will get a 2d embedding vector

Hidden layer
100 neurons, fully connected to the 6 numbers that make up the 3 characters

Output layer
27 neurons, each fully connected to the 100 neurons above, output 27 logits

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Code

```
# Lookup table of embeddings for each char
C = torch.randn((27, 2))

# Make Minibatch of 32 examples from training set
ix = torch.randint(0, Xtr.shape[0], (32,))

# Get embeddings for batch
emb = C[Xtr[ix]]
# Xtr[ix] is the minibatch (32, 3)
# emb is the batch's embeddings (32, 3, 2)
```

What's happening

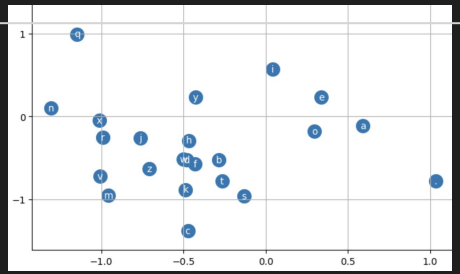
Purpose of embeddings: Help models learn **useful similarities** btw inputs

- Eg. Model sees "A dog is running in a ___." It hasn't encountered that in training set before (out of distribution)
- But it has seen similar sequences in training - "A cat" - can use these to generalise if "dog" and "cat" have similar embeddings

How it works:

- First assign random 2d embedding vectors to each char
- As model trains, improve embeddings by calculating derivative of loss wrt each embedding vector and tweaking embeddings

End up with something like the graph (see that 'a', 'e', 'o' may have similar embeddings)



3: Hidden layer

Input
Indexes of incoming block. If we take a block_size = 3 , then we have 3 inputs

Embedding table
27 by 2 matrix. Each character will get a 2d embedding vector

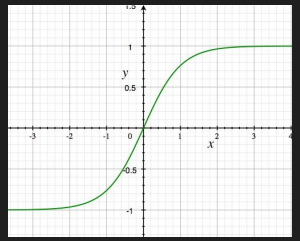
Hidden layer
100 neurons, fully connected to the 6 numbers that make up the 3 characters

Output layer
27 neurons, each fully connected to the 100 neurons above, output 27 logits

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Code

```
# Setting up hidden layer params
W1 = torch.randn((6, 100))
b1 = torch.randn(100)

# Hidden layer
h = torch.tanh(emb.view(-1, 6) @ W1 + b1) #(32,100)
```



What's happening

Hidden layer:

- **# of neurons:** 100
- **Input**: emb – (32, 3, 2)
- **Weights**: 6 per neuron (2*3, each example has 3 chars represented by 2d embeddings, 6 nums in total) – (6, 100)
- **Bias**: 1 per neuron – (100)
- **Output**: 1 activation per neuron per example – (32, 100)
- (32, 6) @ (6, 100) + (100) → (32, 100)

`emb.view(-1, 6) @ W1 + b1`

- Emb is (32, 3, 2) need to cast to (32, 6) to matrix mul with W1 (6, 100)
- This is a **linear transformation**

`torch.tanh(emb.view(-1, 6) @ W1 + b1)`

- This is a **non linear transformation**. Without a non-linearity, all linear layers collapse into **a big linear transformation**, no matter how many layers you stack. → single layer neural net
- **tanh function** squashes input values into the range **(-1, 1)** and helps prevent exploding activations. It is also differentiable and good for gradient-based learning

4: Output layer

Input
Indexes of incoming block. If we take a block_size = 3 , then we have 3 inputs

Embedding table
27 by 2 matrix. Each character will get a 2d embedding vector

Hidden layer
100 neurons, fully connected to the 6 numbers that make up the 3 characters

Output layer
27 neurons, each fully connected to the 100 neurons above, output 27 logits

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Code

```
# Setting up output layer params
W2 = torch.randn((100, 27))
b2 = torch.randn(27)

# Output layer
logits = h @ W2 + b2
```

What's happening

Output layer:
- **# of neurons:** 27
- **Input**: hidden layer activations – (32, 100)
- **Weights**: 100 per neuron – (100, 27)
- **Bias**: 1 per neuron – (27)
- **Output**: 1 activation per neuron per example – (32, 27)
  - **Forms logits for each example**
- (32, 100) @ (100, 27) + (27) → (32, 27)

```
# Softmax layer
counts = logits.exp()
prob = counts/counts.sum(dim=1, keepdim=True)

# Loss
loss = -prob[torch.arange(32), Ytr].log().mean()
```

Softmax layer:
- **Log counts** → (*exponentiate*) → **Counts** → (*divide by row sum*) → **Probability**

Loss:
- For each example, get likelihood for the correct output char
- Get negative log likelihood (nll)
- Take average of nll across all examples → LOSS

The softmax + loss steps can be condensed into

```
loss = F.cross_entropy(logits, Ytr[ix])
```

5: Putting it together

Input
Indexes of incoming block. If we take a block_size = 3 , then we have 3 inputs

Embedding table
27 by 2 matrix. Each character will get a 2d embedding vector

Hidden layer
100 neurons, fully connected to the 6 numbers that make up the 3 characters

Output layer
27 neurons, each fully connected to the 100 neurons above, output 27 logits

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Code: Parameters

```
# Parameters

# Embedding table (2 dim)
C = torch.randn((27, 2), generator=g)

# Params for hidden layer, 100 neurons, 6 weights each
W1 = torch.randn((6, 100), generator=g)
b1 = torch.randn(100, generator=g)

# Params for hidden layer, 27 neurons, 100 weights each
W2 = torch.randn((100, 27), generator=g)
b2 = torch.randn(27, generator=g)

parameters = [C, W1, b1, W2, b2]
for p in parameters:
    p.requires_grad = True
```

Code: Gradient descent

```
# Gradient descent with minibatches
for _ in range(20000):

    # minibatch construct: get random batch of 32 between 0 and 22814
    ix = torch.randint(0, Xtr.shape[0], (32,))

    # forward pass
    emb = C[Xtr[ix]] # (32, 3, 2)
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)  # (32, 100)
    logits = h @ W2 + b2 # (32, 27)
    loss = F.cross_entropy(logits, Ytr[ix])

    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()

    # update
    for p in parameters:
        p.data += -0.1 * p.grad
print(loss.item())
```

- Embedding table
- Hidden layer
- Output layer
- Softmax + Loss

6: Choosing the learning rate

Code

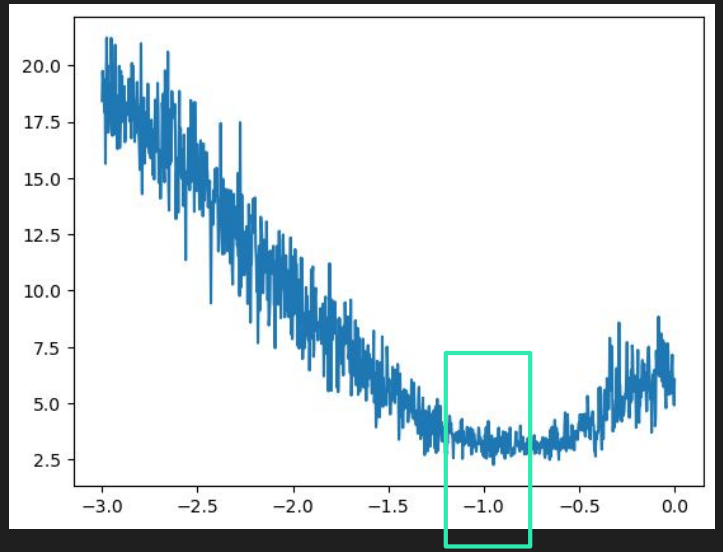
```
# Try 1000 different learning rates from 10^-3 to 10^0 (i.e. 0.001 to 1)
lre = torch.linspace(-3, 0, 1000) # test lr from 10^-3 to 10^0
lrs = 10**lre

# lr trackers
lri = []
lossi = []

for i in range(1000):
    # minibatch construct: random batch of 32
    ix = torch.randint(0, Xtr.shape[0], (32,))

    # forward pass
    emb = C[Xtr[ix]] # (32, 3, 2)
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)  # (32, 100)
    logits = h @ W2 + b2 # (32, 27)
    loss = F.cross_entropy(logits, Y[ix])

    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()

    # update
    lr = lrs[i]
    for p in parameters:
        p.data += -lr * p.grad

    # track
    lri.append(lre[i])
    lossi.append(loss.item())
plt.plot(lri, lossi)
```

What's happening

Finding the best `lr` for updating model

```
# update
for p in parameters:
   p.data += -lr * p.grad
```

Lowest point on graph is interpreted as the point where model is learning fastest without yet diverging



Makemore part 3

Improving our model

- Input
- Embedding table
- Hidden layer - improve initialisation
- BatchNorm layer
- Output layer - improve initialisation
- Softmax layer

1.1: Improving initialisation – fix initial loss

Code

```
# Gradient descent with minibatches
for _ in range(20000):

    # minibatch construct: get random batch of 32 between 0 and 22814
    ix = torch.randint(0, Xtr.shape[0], (32,))

    # forward pass
    emb = C[Xtr[ix]] # (32, 3, 2)
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)  # (32, 100)
    logits = h @ W2 + b2 # (32, 27)
    loss = F.cross_entropy(logits, Ytr[ix])
```

What's happening

Problem:

- **Initial loss is very high** because model initialises parameters randomly.
- In probability distribution, some chars are assigned high prob and others are low prob, and nn is very confidently wrong

What we want: :

- Probability distribution at initialisation should be uniform (because there's no reason to believe any char is more likely than the other)
- We want **logits to be equal** so that after softmax probabilities can be uniform

How to fix: :

- `logits = h @ W2 + b2`
- Make all logits roughly zero for simplicity) by making W2 small (eg. 0.01) and b2 = 0

Outcome:

- Initial logits all closer to 0, more uniform distribution of probabilities, lower initial loss, spend more time doing productive training instead of learning overconfidence and squashing weights down

```
# Parameters
C = torch.randn((vocab_size, n_embed),              generator=g)
W1 = torch.randn((block_size * n_embed, n_hidden),  generator=g)
b1 = torch.randn((n_hidden),                        generator=g)
W2 = torch.randn((n_hidden, vocab_size),            generator=g) * 0.01 # make small to fix initial loss
b2 = torch.randn((vocab_size),                      generator=g) * 0    # make small to fix initial loss
```

1.2: Improving initialisation – fix saturated tanh

Code

```
# Gradient descent with minibatches
for _ in range(20000):

    # minibatch construct: get random batch of 32 between 0 and 22814
    ix = torch.randint(0, Xtr.shape[0], (32,))

    # forward pass
    emb = C[Xtr[ix]] # (32, 3, 2)
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)  # (32, 100)
    logits = h @ W2 + b2 # (32, 27)
    loss = F.cross_entropy(logits, Ytr[ix])
```

```
# Parameters
C = torch.randn((vocab_size, n_embed),              generator=g)
W1 = torch.randn((block_size * n_embed, n_hidden),  generator=g)
b1 = torch.randn((n_hidden),                        generator=g)
W2 = torch.randn((n_hidden, vocab_size),            generator=g) * 0.01
b2 = torch.randn((vocab_size),                      generator=g) * 0
```

What's happening

Problem:

h values are mostly around -1 or 1.

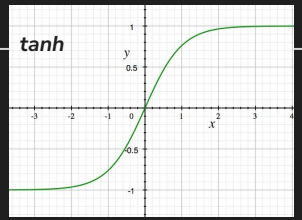
- hpreact takes on quite a large range from -15 to 15, and tanh squeezes it to (-1, 1) so most large values become -1 or 1.

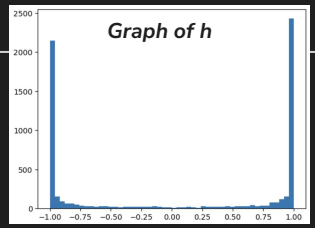
Why is this a problem?

- During back propagation through tanh, self.grad += (1-h**2) * out.grad
- When **t = 1 or -1**, self.grad + =0, killing the gradient (makes sense bc output v close to 1 means its in the tail regions of tanh where changing input won't change output much, hence loss won't change much, hance grad wrt loss is 0.
- **Dead neuron**: If in one tanh neuron, all the activations (h) lie in the tail, it means that no single example ever activates that neuron - no gradient flows through - neuron will never learn and change

How to fix:

- hpreact (activation from hidden layer) is too large (-15, 15)
- Make the activations throughout a nn be **Gaussian-like** and have **0 mean and unit variance** (std close to 1) so we can avoid exploding or vanishing signals





1.2: Improving initialisation – fix saturated tanh

Code

```
# Gradient descent with minibatches
for _ in range(20000):

    # minibatch construct: get random batch of 32 between 0 and 22814
    ix = torch.randint(0, Xtr.shape[0], (32,))

    # forward pass
    emb = C[Xtr[ix]] # (32, 3, 2)
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)  # (32, 100)
    logits = h @ W2 + b2 # (32, 27)
    loss = F.cross_entropy(logits, Ytr[ix])
```

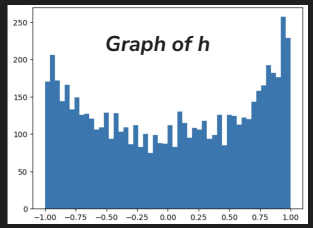
What's happening

GOAL: Make the activations throughout a nn be **Gaussian-like** and **have 0 mean and unit variance** (std close to 1) so that we can avoid exploding or vanishing signals.

SOL 1: **Kaiming Normal Initialization**

- Matrix multiplication increases variance by a factor of fan_in (number of inputs to the layer).
  - input x (mean=0, var_x); weights w (mean=0, var_w),
  - output z has (mean=0, var_z = fan_in * var_x * var_w).
- To preserve variance, we want var_z = var_x, meaning var_w = 1/fan_in. So `std_w = 1/(fan_in**0.5)`
- However, there is a catch. Activations like ReLU, Leaky ReLU, and tanh are contractive — they shrink the variance of their inputs. To compensate for that contraction, we scale up the weights a bit through a **gain**.
  - Diff activation functions have diff gain (tanh gain = 5/3, ReLU gain = 2**0.5, linear gain = 1)
- Final formula: **`std_w = gain/(fan_in**0.5)`** --- **Kaiming Normal Initialization**

```
# Parameters
C = torch.randn((vocab_size, n_embed),                generator=g)
W1 = torch.randn((block_size * n_embed, n_hidden),    generator=g) * (5/3)/((n_embed*block_size)**0.5)
b1 = torch.randn((n_hidden),                          generator=g)
W2 = torch.randn((n_hidden, vocab_size),              generator=g) * 0.01
b2 = torch.randn((vocab_size),                        generator=g) * 0
```



2: Adding BatchNorm layer – fix saturated tanh

Code

```
# How BatchNorm works

# Standardise hpreact to Gaussian across a batch
mean = hpreact.mean(0, keepdim=True).shape  # (1, n_hidden)
std = hpreact.std(0, keepdim=True).shape    # (1, n_hidden)
hpreact = (hpreact - mean)/ std

# Learnable paramters
bngain = torch.ones((1, n_hidden))
bnbias = torch.zeros((0, n_hidden))
hpreact = bngain * (hpreact - mean)/ std + bnbias
```

What's happening

GOAL: Make the activations throughout a nn be **Gaussian-like** and **have 0 mean and unit variance** (std close to 1) so that we can avoid exploding or vanishing signals.

SOL 2: **Batch Normalisation**

- Standardises hpreact to Gaussian across the batch
  - **hpreact = (hpreact - mean)/ std**
- But we only want hpreact to be Gaussian at **initialisation**. During training, we want backpropagation to shape the distribution. So we introduce learnable parameters
  - **hpreact = bngain * (hpreact - mean)/ std + bnbias**
  - At initialisation, bngain=1 and bnbias=0, hence output is exactly unit gaussian. During optimisation, we can back propagate into bngain and bnbias to change them (they are now also params of the nn), which allows hpreact distribution to move around

It is common to add BatchNorm after layers with multiplications (linear/convolutional) to make activation scales consistent and prevent exploding or vanishing gradients

2: Adding BatchNorm layer – fix saturated tanh

Code

```
# minibatch construct
ix = torch.randint(0, Xtr.shape[0], (batch_size,), generator=g)
Xb, Yb = Xtr[ix], Ytr[ix] # Xb (32, 3), Yb (32, 1)

# forward pass
emb = C[Xb] # emb (32, 3, 10)
embcat = emb.view(emb.shape[0], -1) # concatenate to (32, 3*10)
# -- linear layer
hpreact = embcat @ W1 # hidden layer pre-activation
# -- BatchNorm layer
bnmeani = hpreact.mean(0, keepdim=True)
bnstdi = hpreact.std(0, keepdim=True)
hpreact = bngain * (hpreact - bnmeani)/bnstdi + bnbias
with torch.no_grad(): # update running mean and std to be used in test time
    bnmean_running = 0.999 * bnmean_running + 0.001 * bnmeani
    bnstd_running = 0.999 * bnstd_running + 0.001 * bnstdi
# -- nonlinearity (activation)
h = torch.tanh(hpreact) # hidden layer
logits = h @ W2 + b2 # output layer
loss = F.cross_entropy(logits, Yb)

# backward pass
for p in parameters:
    p.grad = None
loss.backward()

# update
lr = 0.1 if i<100000 else 0.01 # step learning rate decay
for p in parameters:
    p.data += -lr * p.grad
```

```
parameters = [C,W1,W2,b2,bngain,bnbias]
```

What's happening

GOAL: Make the activations throughout a nn be **Gaussian-like** and **have 0 mean and unit variance** (std close to 1) so that we can avoid exploding or vanishing signals.

Batch Normalisation standardises hpreact to Gaussian across the batch

hpreact = bngain * (hpreact - mean)/ std + bnbias

However, because BN uses the mean and std across the whole batch, the activations for one example now depend on the others in the same batch. This has 2 effects:

- ✅ **Regularisation** - Varying batch stats make hpreact & logits jitter for each example. This acts like data augmentation and reduces overfitting to any 1 example
- ❌ **Makes inference harder** - Usually want to feed in a single example during inference. But now the model expects a batch as input to compute mean and std

SOL: Use Fixed BatchNorm Stats at inference time so no need batch :)
Two common approaches:

1. **Post-hoc:** After training, compute the true mean and std of all hpreacts and use that for inference
2. **Running estimate****: During training, maintain a running average of batch means and stds which is then used in evaluation

This makes the model work correctly even on a single test example.

Cleaned up model!

Input
Xtr (182580, 3) - 182580 examples of block_size = 3

Embedding table
Each character gets assigned a 10-dim embedding vector. C (27,10)

Hidden layer
100 neurons, fully connected to above layer - Improve initialisation

BatchNorm layer
Make 100 activations of previous layer Gaussian

Output layer
27 neurons to output 27 logits- Improve initialisation

Softmax layer
The 27 logits are exponentiated and divided to make probabilities

Make these into layers to stack in a list easily

Overview

```
n_embed = 10 # dimentionality of embedding vectors for the chars
n_hidden = 100 # num neurons in hidden layer of MLP
g = torch.Generator().manual_seed(2147483647)

C = torch.randn((vocab_size, n_embed), generator=g) # embeddings for each char

layers = [
    Linear(n_embed * block_size, n_hidden, bias = False), BatchNorm1d(n_hidden), Tanh(),
    Linear(            n_hidden, n_hidden, bias = False), BatchNorm1d(n_hidden), Tanh(),
    Linear(            n_hidden, n_hidden, bias = False), BatchNorm1d(n_hidden), Tanh(),
    Linear(            n_hidden, n_hidden, bias = False), BatchNorm1d(n_hidden), Tanh(),
    Linear(            n_hidden, n_hidden, bias = False), BatchNorm1d(n_hidden), Tanh(),
    Linear(            n_hidden, vocab_size, bias = False), BatchNorm1d(vocab_size)
]
```

So what do these layers look like? 🤔

```
with torch.no_grad():
    # for last layer, make less confident (gamma = weight)
    layers[-1].gamma *= 0.1
    # for linear layers, apply gain for kaiming normal initialisation
    gain = 5/3
    for layer in layers[:-1]:
        if isinstance(layer, Linear):
            layer.weight *= gain

parameters = [C] + [p for layer in layers for p in layer.parameters()]
print (sum(p.nelement() for p in parameters)) # total num of params
for p in parameters:
    p.requires_grad = True
```

Cleaned up model!

Layers - Linear, Tanh

```python
class Linear:

    # eg Linear(n_embed * block_size, n_hidden, bias = False)
    def __init__(self, fan_in, fan_out, bias=True):
        self.weight = torch.randn((fan_in, fan_out), generator=g) / fan_in**0.5
        self.bias = torch.zeros(fan_out) if bias else None

    def __call__(self, x):
        self.out = x @ self.weight
        if self.bias is not None:
            self.out += self.bias
        return self.out

    def parameters(self):
        return [self.weight] + ([] if self.bias is None else [self.bias])
```

Kaiming Normal Initialization

```python
class Tanh:
    def __call__(self, x):
        self.out = torch.tanh(x)
        return self.out
    def parameters(self):
        return []
```

Cleaned up model!

Layers - Linear, Tanh, BatchNorm

```
class BatchNorm1d:

    def __init__(self, dim, eps=1e-5, momentum=0.1):
        self.eps = eps
        self.momentum = momentum
        self.training = True # flag for whether BN is called during training or test
        # params
        self.gamma = torch.ones(dim) #bngain
        self.beta = torch.zeros(dim) #bnbias
        #buffers (trained with running 'momentum update')
        self.running_mean = torch.zeros(dim)
        self.running_var = torch.ones(dim)

    # calling BatchNorm
    def __call__(self, x):
        # forward pass
        if self.training:
            xmean = x.mean(0, keepdim=True) # batch mean
            xvar = x.var(0, keepdim=False)  # batch variance
        else:
            xmean = self.running_mean
            xvar = self.running_var
        xhat = (x-xmean)/torch.sqrt(xvar + self.eps) # hpreact = (hpreact - mean)/std
        self.out = self.gamma * xhat + self.beta     # hpreact = bngain * hpreact + bnbias
        # update buffers
        if self.training:
            with torch.no_grad():
                self.running_mean = (1-self.momentum) * self.running_mean + self.momentum * xmean
                self.running_var = (1-self.momentum) * self.running_var + self.momentum * xvar
        return self.out

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

What's happening

`dim` (=number of activations coming in, 100): used to initialise bngain, bnbias, bnmean_running and bnstd_running

`momentum:` used to update running mean and std
(eg. momentum = 0.1 means running_mean = 0.9*running_mean + 0.1 *xmean, value depends on batch size

In training:

1. Calculate batch mean
2. Make activations gaussian using
   `hpreact = bngain * (hpreact - mean)/std + bnbias`
3. Update running mean and std

In inference:

1. Make activations gaussian using running mean and std

Cleaned up model!

```
# Optimisation
max_steps = 200000
batch_size = 32
lossi = []
ud = []

for i in range(max_steps):

    # minibatch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size, ), generator=g) # random indexes
    Xb, Yb = Xtr[ix], Ytr[ix] # batch X, Y

    # forward pass
    emb = C[Xb]                     # embed chars into vectors
    x = emb.view(emb.shape[0], -1)  # concat vectors
    for layer in layers:            # pass thru layers
        x = layer(x)
    loss = F.cross_entropy(x, Yb)   # loss

    # backward pass
    for layer in layers:
        layer.out.retain_grad()     # retain grad of outputs explicitly bc not leaf tensors
    for p in parameters:
        p.grad = None
    loss.backward()

    # update
    lr = 0.1 if i<150000 else 0.01  # lr decay
    for p in parameters:
        p.data += - lr * p.grad

    # track stats
    if i % 10000 == 0:
        print(f'{i:7d}/{max_steps:7d}: {loss.item():.4f}')
    lossi.append(loss.log10().item())
    with torch.no_grad():
        # track ratio of size of each update:data using std
        ud.append([(lr*p.grad.std() / p.data.std()).log10().item() for p in parameters])
```

```
     0/ 200000: 3.3026
 10000/ 200000: 1.8686
 20000/ 200000: 2.2283
 30000/ 200000: 2.0487
 40000/ 200000: 2.6062
 50000/ 200000: 2.2266
 60000/ 200000: 2.2891
 70000/ 200000: 2.2984
 80000/ 200000: 2.2128
 90000/ 200000: 2.2651
100000/ 200000: 2.1352
110000/ 200000: 2.4476
120000/ 200000: 2.0540
130000/ 200000: 1.9266
140000/ 200000: 1.9233
150000/ 200000: 2.2376
160000/ 200000: 1.6648
170000/ 200000: 1.8233
180000/ 200000: 2.4697
190000/ 200000: 1.8458
```

Makemore part 4

Learning positional order through WaveNet

- Input
- Embedding layer
- Flatten Consecutive layer
- Hidden layer
- BatchNorm layer
- Output layer
- Softmax layer

Introducing Wavenet

Assuming a batch of 4 examples with context length of 8, each char a 10 dim vector

```
........ --> x
.......x --> a
......xa --> v
.....xav --> i
```

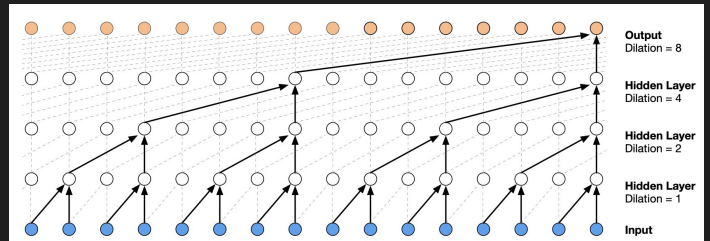


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Previous: MLP with Flattening

For each example, pass all embeddings of 8 characters into the hidden layer at the same time: `(1 2 3 4 5 6 7 8)`

```
Xb # (4,8)
emb = C[Xb] # (4,8,10)
B, T, C = emb.shape
x = emb.view(B, T*C) # (4, 80) Flattening
h = x @ W1 + b1 # (4,80) @ (80,200) + (200) --> (4, 200)
```

Problem:

- All characters (positions 1–8) are flattened into one vector (like reading a whole para as a single long word.) The model can't easily separate what happened earlier vs. later.
- Harder to learn order-based rules/temporal patterns like "if A comes before B, then X" or "the 8th token matters more than the 1st"

New: WaveNet

For each example, group embeddings of 8 characters into groups of 2 to pass into hidden layer: `(1 2) (3 4) (5 6) (7 8)`

```
Xb # (4,8)
emb = C[Xb] # (4,8,10)
B, T, C = emb.shape
x = emb.view(B, T//self.n, C*self.n) # (4, 4, 20) WaveNet
h = x @ W1 + b1 # (4,4, 20) @ (20,200) + (200) --> (4, 4, 200)
```

Solution:

- Each time-step processed distinctly with one hidden vector each (like reading the paragraph word by word, preserving order).
- The model can learn local structure ("qu" or "th").
- Scale with depth and dilated convolutions to capture long-range dependencies. Layer 1 looks at short, local windows (1,2), (3,4)..., Layer 2 dilates or expands the window (1,,4), (4, 8), Higher layers see further (1, 8) (8, 16)

Introducing Wavenet

Layers - Linear, Tanh, BatchNorm, Embedding, Flatten Consecutive

```python
class Embedding: # replace emb = C[Xb]
  def __init__(self, num_embeddings, embedding_dim):
    self.weight = torch.randn((num_embeddings, embedding_dim))

  def __call__(self, IX):
    self.out = self.weight[IX]
    return self.out

  def parameters(self):
    return [self.weight]
```

```python
class FlattenConsecutive: # replace x = emb.view(emb.shape[0], -1)
  def __init__(self, n):
    self.n = n
    # (1 2) (3 4) (5 6) (7 8) --> n=2

  def __call__(self, x):
    B, T, C = x.shape
    x = x.view(B, T//self.n, C*self.n)
    if x.shape[1] == 1: # handling spurious dimension (B, 1, C*n)
      x = x.squeeze(1)

    self.out = x
    return self.out

  def parameters(self):
    return []
```

```python
Xb # (B,T)
emb = C[Xb] # (B,T,C)
B, T, C = emb.shape
x = emb.view(B, T//self.n, C*self.n) # (B, T//n, C*n)
h = x @ W1 + b1
# (B, T//n, C*n) @ (C*n, n_hidden) + (n_hidden) --> (B, T//n,
n_hidden)
```

Eg.

Embedding : (32, 8, 10)
Flatten Consecutive: (32, 4, 20)
Linear : (32, 4, 64)
BatchNorm1d : (32, 4, 64)
Tanh : (32, 4, 64)
Flatten Consecutive: (32, 2, 128)
Linear : (32, 2, 64)
BatchNorm1d : (32, 2, 64)
Tanh : (32, 2, 64)
Flatten Consecutive: (32, 128)
Linear : (32, 64)
BatchNorm1d : (32, 64)
Tanh : (32, 64)
Linear : (32, 27)

Cleaned up model! – *Model and Params*

Layers - Linear, Tanh, BatchNorm, Embedding, Flatten Consecutive, Sequential (combine all!)

```python
class Sequential:

  def __init__(self, layers):
    self.layers = layers

  def __call__(self, x):
    for layer in self.layers:
      x = layer(x)
    self.out = x
    return self.out

  def parameters(self):
    # get params of all layers and stretch them out into 1 list
    return [p for layer in self.layers for p in layer.parameters()]
```

```python
n_embd = 10
n_hidden = 64

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),
])

# parameter init
with torch.no_grad():
    model.layers[-1].weight *= 0.1 # make last layer less confident

parameters = model.parameters()
print(sum(p.nelement() for p in parameters))
for p in parameters:
    p.requires_grad = True
```

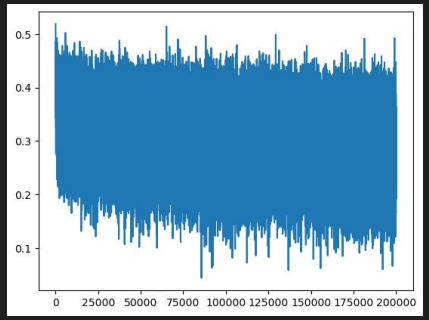
```
Embedding :                (32, 8, 10)
Flatten Consecutive:       (32, 4, 20)
Linear :                   (32, 4, 64)
BatchNorm1d :              (32, 4, 64)
Tanh :                     (32, 4, 64)
Flatten Consecutive: (32, 2, 128)
Linear :                   (32, 2, 64)
BatchNorm1d :              (32, 2, 64)
Tanh :                     (32, 2, 64)
Flatten Consecutive : (32, 128)
Linear :                   (32, 64)
BatchNorm1d :              (32, 64)
Tanh :                     (32, 64)
Linear :                   (32, 27)
```

Cleaned up model! – *Gradient descent*

```python
# same optimisation as part 3
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):

    # minibatch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,))
    Xb, Yb = Xtr[ix], Ytr[ix]

    # forward pass
    logits = model(Xb)
    loss = F.cross_entropy(logits, Yb)

    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()

    # update: simple SGD
    lr = 0.1 if i <150000 else 0.01
    for p in parameters:
        p.data += -lr*p.grad

    # track stats
    if i % 10000 ==0:
        print(f'{i:7d}/{max_steps:7d}: {loss.item():.4f}')
    lossi.append(loss.log10().item())
```

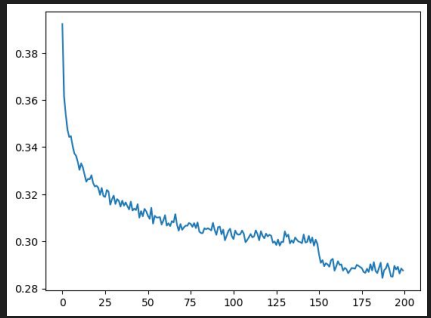
```
Loss
  0    / 200000: 3.3026
 10000/ 200000: 1.8686
 20000/ 200000: 2.2283
 30000/ 200000: 2.0487
 40000/ 200000: 2.6062
 50000/ 200000: 2.2266
 60000/ 200000: 2.2891
 70000/ 200000: 2.2984
 80000/ 200000: 2.2128
 90000/ 200000: 2.2651
100000/ 200000: 2.1352
110000/ 200000: 2.4476
120000/ 200000: 2.0540
130000/ 200000: 1.9266
140000/ 200000: 1.9233
150000/ 200000: 2.2376
160000/ 200000: 1.6648
170000/ 200000: 1.8233
180000/ 200000: 2.4697
190000/ 200000: 1.8458
```

```python
plt.plot(lossi)
```




```python
plt.plot(torch.tensor(lossi).view(-1, 1000).mean(1))
```

lossi array (200,000), make it (200,1000), take mean across 1000 losses to plot 200 means



Cleaned up model! – *Evaluate training and validation loss*

```
# evaluate the loss
@torch.no_grad() # disables gradient tracking
def split_loss(split):
  x,y = {
    'train': (Xtr, Ytr),
    'val': (Xdev, Ydev),
    'test': (Xte, Yte),
  }[split]
  logits = model(x)
  loss = F.cross_entropy(logits, y)
  print(split, loss.item())

split_loss('train')
split_loss('val')
```

```
train 1.920527696609497
val 2.024179220199585
```

Cleaned up model! – *Sample!!*

```python
# sample from the model
for _ in range(20):

    out = []
    context = [0] * block_size # initialize with all ...
    while True:
      # forward pass the neural net
      logits = model(torch.tensor([context]))
      probs = F.softmax(logits, dim=1)
      # sample from the distribution
      ix = torch.multinomial(probs, num_samples=1).item()
      # shift the context window and track the samples
      context = context[1:] + [ix]
      out.append(ix)
      # if we sample the special '.' token, break
      if ix == 0:
        break

    print(''.join(itos[i] for i in out)) # decode and print the generated word
```

Samples now	Samples in Part 1	Training egs
sanay.	cexze.	emma
zuham.	momasurailezityha.	olivia
zaison.	konimittain.	ava
agondre.	llayn.	isabella
karina.	ka.	sophia
kreku.		charlotte
mcklon.		mia
thipion.		amelia
javid.		
sebri.		
came.		
azanni.		
griwai.		
bretzon.		
luishel.		
omasha.		
bodion.		
arzin.		
iyanna.		