Here is the documentation for **Make More v1**—a minimal character-level neural language model implemented using PyTorch.

**📄 Make More v1 — Documentation**

**🔍 Overview**

**Make More v1** is a character-level language model inspired by Andrej Karpathy’s "makemore" project. It uses a simple neural network to learn the distribution of characters in names. The model can then generate new, plausible-looking names based on the character distribution it has learned.

This version implements:

* Character bigram probabilities
* Sampling using torch.multinomial
* Name generation based on learned bigram distribution

**📦 Dependencies**

pip install torch

**📘 Key Components**

**1. Bigram Probability Table**

* A 27×27 table N is created, where each entry N[i][j] holds the count of transitions from character i to character j.
* Index 0 corresponds to the start (<S>) or end (<E>) token.

# Example: incrementing bigram counts

for w in words:

chs = ['.'] + list(w) + ['.']

for ch1, ch2 in zip(chs, chs[1:]):

ix1 = stoi[ch1]

ix2 = stoi[ch2]

N[ix1, ix2] += 1

**2. Sampling from Probabilities**

Using torch.multinomial():

g = torch.Generator().manual\_seed(837214089)

ix = torch.multinomial(p, num\_samples=1, replacement=True, generator=g).item()

This picks an index based on the probability distribution p.

**🛠️ Main Name Generation Logic**

for i in range(20):

out = []

ix = 0 # Start with the start token

while True:

p = N[ix].float()

p = p / p.sum() # Normalize to get probabilities

ix = torch.multinomial(p, num\_samples=1, replacement=True, generator=g).item()

out.append(itos[ix])

if ix == 0: # End token

break

print(''.join(out))

This loop generates 20 names by:

* Starting from the start token (ix=0)
* Sampling the next character based on the current row of the bigram probability matrix
* Continuing until the end token is sampled (ix==0)

**🧠 Concepts Used**

| **Concept** | **Explanation** |
| --- | --- |
| Bigram Modeling | Predicting the next character based only on the current one. |
| Probability Sampling | Using multinomial sampling to sample next characters based on learned distributions. |
| Normalization | Ensuring probabilities sum to 1 before sampling. |
| Manual Seeding | Fixing the seed ensures repeatable name generation. |

**📈 Limitations of Version 1**

* Only captures **bigram** (2-char) dependencies.
* Doesn’t use gradient-based learning or neural networks.
* Cannot generalize beyond seen character transitions.

**✅ Output Example**

Generated names might look like:

avon.

carl.

elai.

nace.

tory.

These names are sampled purely from bigram statistics.

**🚀 Next Step**

**Make More v2** introduces a neural network using PyTorch:

* Embeddings for characters
* Multi-layer perceptron
* Training with cross-entropy loss

Let me know if you'd like this documentation as a Markdown file or want a PDF version.

**📘 MakeMore Version 2: Neural Probabilistic Language Model with Character Embeddings**

**Overview:**  
This version builds upon MakeMore V1, transitioning from a statistical bigram model to a neural network-based language model. It implements a simple multi-layer perceptron (MLP) trained to predict the next character in a sequence using character embeddings and a hidden layer. The model is trained on character-indexed name sequences from names.txt.

**🔧 Key Components:**

**1. Embedding Layer (Lookup Table):**

* Each character (27 in total: a–z + .) is represented by a learnable 2D embedding vector.
* Embeddings are extracted via matrix indexing (instead of one-hot and matmul).

**2. Input Representation:**

* Each training example is a block of 3 characters.
* Embeddings for these 3 characters (shape (3, 2)) are concatenated into a single 6D input vector per example.

**3. MLP Architecture:**

* **Layer 1:** Fully connected layer from 6 → 100 dimensions, followed by tanh activation.
* **Layer 2:** Fully connected layer from 100 → 27 dimensions (the vocabulary size), producing raw logits.
* Output logits represent unnormalized scores for the next character.

**4. Loss Function:**

* Two approaches:
  + Manual softmax + log-likelihood loss
  + PyTorch’s efficient F.cross\_entropy (used for training)
    - Handles numerical stability via log-sum-exp trick
    - Efficient forward & backward computation

**5. Training Loop:**

* Trains using mini-batches (32 samples at a time)
* Performs:
  + Forward pass
  + Backward pass using .backward()
  + Parameter updates using manual SGD (p.data -= lr \* p.grad)
* Runs for 100 epochs

**📉 Final Output:**

* After training, the model outputs logits for each character.
* These are converted into probabilities using softmax.
* The trained model's predictions are evaluated by comparing the predicted probabilities against the true next character targets.

**✅ Concepts Covered:**

* Character embeddings
* Tensor reshaping (view)
* PyTorch broadcasting
* Manual vs. built-in loss functions
* Numerical stability in softmax
* Neural network training with PyTorch (no high-level API)

Let me know if you'd like to:

* Add metrics like accuracy or loss plots
* Extend to generate new names with sampling
* Refactor this into a class-based model or training loop

Would you like a markdown-formatted version of this for your README or notebook heading too?

**🧠 Make More v2.1 — Neural Probabilistic Language Model**

**✅ Implementation Overview**

In **version 2.1**, I implemented a **Multi-Layer Perceptron (MLP)** model for predicting the next character in a name based on a 3-character context window (block size = 3). The model architecture includes:

* A learned **character embedding table**
* A **single hidden layer** with non-linear activation (tanh)
* A final **output layer** projecting to vocabulary logits
* **Cross-entropy loss** for training
* **Mini-batch SGD** for parameter updates

This version incorporates:

* Manual forward and backward pass with PyTorch
* Embedding space visualization
* Training/validation loss evaluation
* Minibatch training with torch.randint

**📚 Reference**

This model is inspired by the architecture and methodology proposed in:

**A Neural Probabilistic Language Model**  
Yoshua Bengio, Réjean Ducharme, Pascal Vincent, Christian Jauvin  
Département d’Informatique et Recherche Opérationnelle  
Centre de Recherche Mathématiques  
Université de Montréal, Montréal, Québec, Canada  
*Editors: Jaz Kandola, Thomas Hofmann, Tomaso Poggio, and John Shawe-Taylor*

[A Neural Probabilistic Language Model (Bengio et al., 2003)](https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf).

Absolutely! Here's a clear and well-organized set of documentation texts based on what you described, divided into:

**📘 Make More v2.1 – Final Notes and Next Steps**

**✅ Summary**

* Implemented an **MLP neural language model** using PyTorch.
* Used a **block size of 3** to model 3-character context windows.
* Reached a **minimum dev loss of ~2.17**.
* Introduced:
  + Embedding layer
  + Single hidden layer with tanh
  + Output layer with softmax logits
  + Minibatch training loop
  + Loss tracking and plotting

**🔜 Next Steps for v3**

To improve the model further:

* 🔍 The **initial loss is very high**. We need to address this to accelerate early training and improve final convergence.
* 🛠 We aim to reduce initial loss by:
  + Making **initial logits closer to zero**
  + **Initializing W2 with a small scale** (e.g., \* 0.01)
  + **Setting bias b1 = 0**
* 🔍 Introduce better **weight initialization** using **Kaiming initialization**
* ✅ Add **Batch Normalization** to stabilize hidden layer activations
* 🔧 Prepare the model for **inference-time adjustments** (single example)

**📘 Make More v3 — Deeper MLP with Batch Normalization and Kaiming Init**

**🚨 Initial Loss Problem**

At the start of training, the model produces extremely **high loss values**, due to large initial logits. This happens because:

The output layer's logits are unbounded and initialized with high variance. This leads to large values passed into softmax → extremely confident incorrect predictions → very high cross-entropy loss.

**✅ Solution: Stabilize Initial Logits**

To fix this:

# Reduce scale of W2

W2 = torch.randn((n\_hidden, vocab\_size), generator=g) \* 0.01

# Set b1 to zeros

b1 = torch.zeros(n\_hidden)

**⚙️ Kaiming Initialization**

You used:

W1 = torch.randn((n\_embd\*block\_size, n\_hidden), generator=g) \* (5/3) / ((n\_embd\*block\_size)\*\*0.5)

This is a variant of **Kaiming (He) initialization**, which scales the weights according to the number of input connections (fan\_in) to preserve signal variance through layers.

* The general formula for Kaiming init is:

std=gainfan\_in\text{std} = \frac{\text{gain}}{\sqrt{\text{fan\\_in}}}

* **Gain for tanh** activation:
  + Recommended gain = 5/3 (derived based on preserving variance through tanh nonlinearity)

📚 **Reference Paper**:

**"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"**  
*Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun*  
[arXiv:1502.01852](https://arxiv.org/abs/1502.01852)

**🧪 Batch Normalization**

You implemented batch norm to stabilize the hidden layer before applying non-linearity:

bngain = torch.ones(1, n\_hidden)

bnbias = torch.zeros(1, n\_hidden)

During training, the hidden layer activations hpreat are normalized:

bnmeani = hpreat.mean(0, keepdim=True)

bnstdi = hpreat.std(0, keepdim=True)

hpreat = bngain \* (hpreat - bnmeani) / bnstdi + bnbias

**🎯 Why Track Running Mean/Std?**

BatchNorm behaves differently during **inference** (e.g., generating one name at a time, not in batches). Since there's no batch to compute mean/std from, we track **running statistics** during training:

bnmean\_running = 0.999 \* bnmean\_running + 0.001 \* bnmeani

bnstd\_running = 0.999 \* bnstd\_running + 0.001 \* bnstdi

These are then used at inference time in place of batch stats.

**📦 Summary of v3 Additions**

| **Feature** | **Description** |
| --- | --- |
| 🔧 **W2 scaling** | Set small initial logits to avoid extreme early loss |
| 🔧 **b1 = 0** | Prevents bias from contributing to large activations |
| 📐 **Kaiming Init** | Proper variance scaling for tanh with gain 5/3 |
| 📊 **BatchNorm** | Normalizes activations, improves training stability |
| 📈 **Running mean/std** | Enables correct inference-time normalization |

Let me know if you'd like this exported in **Markdown format** for your README or notebook headings!

**🔁 Backward Pass Implementation (makemore v3.1)**

This version implements a fully manual backward pass for a character-level language model, including a custom Batch Normalization backward step.

**1. Loss:**

logits = ...

logits = F.log\_softmax(logits, dim=1) # logits: (batch, vocab)

loss = F.nll\_loss(logits, Y)

**2. Gradient w.r.t. logits:**

dlogits = F.softmax(logits, dim=1)

dlogits[range(len(Y)), Y] -= 1

dlogits /= Y.shape[0]

**🔄 Backpropagation**

**3. Final Linear Layer (W2, b2):**

dW2 = h.T @ dlogits

db2 = dlogits.sum(0)

dh = dlogits @ W2.T

**4. Batch Normalization Backward (Manual Simplified Formula):**

Let:

* n = batch size
* bnraw = normalized input
* bngain = learnable gain
* bnvar\_inv = 1 / sqrt(variance + ε)

dhprebn = bngain \* bnvar\_inv / n \* (

n \* dhpreact

- dhpreact.sum(0)

- bnraw \* (dhpreact \* bnraw).sum(0) \* (n / (n - 1))

)

**5. BatchNorm Parameters (bngain, bnbias):**

dbngain = (bnraw \* dhpreact).sum(0)

dbnbias = dhpreact.sum(0)

**6. Pre-Activation Gradient (hprebn) through Linear Layer:**

dW1 = x.T @ dhprebn

db1 = dhprebn.sum(0)

dx = dhprebn @ W1.T

**7. Embedding Layer Backward:**

demb = torch.zeros\_like(C)

demb[IX] = dx

**✅ Validation**

At every step, we use the cmp function to compare the manually computed gradients to those from autograd for correctness:

cmp('dlogits', dlogits, logits.grad)

cmp('dW2', dW2, W2.grad)

...

Let me know if you also want the **forward pass documentation** or version notes added.

**📘 MakeMore Version 4.0: Class-based Neural Language Model with Training Dynamics Visualization**

**🔄 Changes & Enhancements from Version 3.0**

Version 4.0 introduces **modular and reusable neural network components using PyTorch nn.Module classes**, marking a significant refactor of the previous function-based model (v3.0). The major improvements in this version include:

1. **Architecture Rewriting in Object-Oriented Style**:
   * All major components of the model (embedding, hidden layers, output layer) are now encapsulated into torch.nn.Module classes, improving clarity, reusability, and scalability for future experiments.
2. **Kaiming Initialization**:
   * Deep layers are initialized using **Kaiming He initialization**, which helps stabilize training, especially for ReLU-based architectures.
   * This was added to avoid activation saturation and vanishing/exploding gradients in deeper networks.
3. **Logit Confidence Adjustment (Output Squashing)**:
   * A with torch.no\_grad() step is added after training to **scale down the last linear layer's weights** (output logits). This reduces overconfidence in softmax outputs and improves generalization by controlling the sharpness of the predicted distribution.
4. **Step-wise Learning Rate Decay**:
   * Instead of a constant learning rate, a step-wise decay schedule is introduced to gradually reduce learning rate during training.
   * This allows fast convergence initially and finer adjustments in later epochs.

**📊 Gradient-to-Weight Ratio Plotting**

To gain insight into how effectively each layer is learning, we implemented two powerful training diagnostics:

**1. Gradient Histogram Plot**

# Sample print output

weight (27, 10) | mean +0.000000 | std 8.88e-03 | grad:data ratio 8.91e-03

...

* For each weight matrix, we compute:
  + mean and std of the gradients
  + Ratio: **grad.std() / param.std()** — tells us how much each parameter is changing per training step.
* We plot the **histogram of gradients** for every weight matrix to visually assess distribution shape (normality, sparsity, outliers, etc.).

**2. Update-to-Data Ratio Tracking Plot**

plt.plot([ud[j][i] for j in range(len(ud))])

* This plot tracks the **log10(update-to-weight ratio)** over training iterations:
  + **X-axis**: Training steps
  + **Y-axis**: log10(grad.std() / weight.std())
  + **Goal**: Keep this value around **-3**, i.e., updates ~0.1% of weight scale
* A dashed horizontal line at **-3** is drawn to indicate the ideal training regime.

**✅ Why It Matters:**

* If the curve stays around **-3**, gradients are healthy.
* If it's too low (e.g., -5), model might be **learning too slowly** → try increasing learning rate.
* If it's too high (e.g., -1), gradients might be **too aggressive** → consider lowering the learning rate.

This visualization helps **diagnose learning issues early**, especially in deep or sensitive models.

**🔍 Summary**

Version 4.0 makes MakeMore more scalable and robust by:

* Refactoring the architecture using object-oriented programming
* Improving training stability through proper weight initialization
* Adding interpretability and monitoring through gradient distribution and update-to-data ratio plots
* Controlling overconfidence in predictions via output squashing

These changes not only improve performance but also provide essential insights into the internal dynamics of the training process.

Thanks for the details! Here's the **cleaned documentation** for **Make More v5.0**, following your style and expanding on your changes clearly:

**📘 Make More v5.0 — WaveNet-Inspired Modular Neural Language Model**

**🔍 Overview**

In version 5.0, the architecture is redesigned to resemble **WaveNet**, a powerful autoregressive model originally developed for audio generation. This version focuses on **composability** and **hierarchical processing** using small, modular components.

**Key Enhancements:**

* 🔧 Introduced **modular building blocks** (Embedding, FlattenConsecutive, Sequential) similar to PyTorch's nn.Module.
* 🧠 Adopted **WaveNet-inspired input batching**: instead of flattening the context window, inputs are processed in structured groups (e.g., bigrams).
* 📊 Improved **loss curve visualization** by averaging over 1000-iteration blocks for clarity.

**🧱 New Modular Components**

**🔤 Embedding**

Custom character embedding layer:

class Embedding:

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]

**🧩 FlattenConsecutive**

Mimics grouped 1D convolutions by flattening adjacent time steps together:

class FlattenConsecutive:

def \_\_init\_\_(self, n):

self.n = n

def \_\_call\_\_(self, x):

B, T, C = x.shape

x = x.view(B, T // self.n, C \* self.n)

if x.shape[1] == 1:

x = x.squeeze(1)

self.out = x

return self.out

def parameters(self):

return []

**🪢 Sequential**

Custom sequential wrapper to compose multiple layers:

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):

return [p for layer in self.layers for p in layer.parameters()]

**🌊 WaveNet-Inspired Architecture**

**A diagram of a stack of dilated causal conversion

AI-generated content may be incorrect.**

Instead of a traditional MLP taking flattened inputs, this version introduces **structured grouping**, simulating 1D convolution over character embeddings:

# Input Shape Example:

e = torch.randn(4, 4, 20) # 4 examples, 4 context tokens, each with 20-dimensional embedding

# Forward transformation example:

out = e @ torch.randn(20, 200) + torch.randn(200)

This simulates a stack of linear layers processing **local patterns** (like a bigram or trigram) before global flattening.

**Motivation:**

* Preserves **temporal order** and **local structure** before combining into global features.
* Mimics **dilated convolutional** behavior of WaveNet.

**📉 Loss Visualization**

Instead of plotting raw loss per step (which is noisy), the training loss is smoothed by averaging over blocks of 1000 steps:

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

**Why this matters:**

* Easier to interpret long-term trends.
* Highlights overfitting, underfitting, or learning plateaus more clearly.

**✅ Summary of Improvements**

| **Feature** | **Description** |
| --- | --- |
| 🧱 Modular Blocks | Custom classes for embedding, sequential logic, and structured reshaping |
| 🌊 WaveNet-style Input | Restructured input as grouped bigrams/trigrams instead of full flattening |
| 🔬 Hierarchical Depth | Supports stacking multiple FlattenConsecutive + Linear layers |
| 📉 Loss Visualization | Improved clarity using averaged loss over intervals |

**📦 Next Steps for v6.0**

* 📈 Add residual connections to emulate full WaveNet
* 🌀 Implement dilated convolutions for larger receptive fields
* 🧠 Incorporate attention or gating for dynamic feature control
* 🧪 Experiment with dropout or layer normalization for regularization

Let me know if you'd like:

* 📄 This as a **Markdown** file for your repo
* 🧪 Help implementing v6.0 ideas
* 🧼 A cleaned version of your full notebook script

Just upload or paste the code, and I’ll take care of the rest!

:  
:::  
Final model sample from make\_more:  
  
names:  
  
carman.

ambrie.

khismi.

xilah.

khalani.

emmahnee.

dellyn.

jarqui.

nermari.

chaiir.

kaleigh.

hamoni.

jaquinn.