Hi everyone! Today we are continuing our implementation of `makemore`, our favorite character-level language model. You might notice the background behind me is different—that's because I'm in a hotel room in Kyoto, and it is awesome!

Over the last few lectures, we've built a Multi-Layer Perceptron (MLP) that takes three previous characters as input to predict the fourth. It's a simple model with a single hidden layer and `tanh` activations. In this lecture, I'd like to make this architecture more complex. Specifically, we want to:

1. **Increase the context size**, taking more than just three characters as input.
2. **Build a deeper model** that doesn't just squash all the input characters into one hidden layer. Instead, we want to fuse this information progressively to make a more informed guess about the next character.

As we make our architecture more complex, we'll arrive at something that looks remarkably like a **WaveNet**. WaveNet was a model published by DeepMind in 2016. While it was designed to predict audio sequences, the underlying modeling setup is identical to ours: it's an autoregressive model that predicts the next element in a sequence. The architecture takes an interesting hierarchical, tree-like approach, which is exactly what we're going to implement today.

Let's get started.

Getting Started: A Recap and Our Code

The starter code for this lecture (Part 5) is very similar to where we ended up in Part 3. (Recall that Part 4 was our manual backpropagation exercise, which was a bit of an aside). I've copy-pasted the relevant chunks, so much of this should look familiar.

We begin by importing libraries, reading our dataset of names, and processing it into examples. The data generation code hasn't changed. We have 182,000 examples where we use three characters to predict a fourth.

In Part 3, we started building our code around layer modules like `class Linear`, thinking of them as Lego bricks we can stack into neural networks. Our custom layers mimic the API and signature of their `torch.nn` counterparts.

We have our `Linear`, `BatchNorm1d`, and `Tanh` layers ready to go.

Peculiarities of Batch Norm

The `BatchNorm1d` layer, as we've discovered, is a bit of a crazy layer. There are a few reasons for this:

- **It has internal state.** It maintains `running_mean` and `running_variance` that are updated outside of backpropagation using an exponential moving

average during the forward pass. State can be a source of complexity and bugs.

- **It has distinct training and evaluation modes.** The layer behaves differently depending on whether it's training (using batch statistics) or evaluating (using the running statistics). Forgetting to switch the model to the correct mode is a common bug.
- **It couples examples within a batch.** We usually think of a batch as a way to compute efficiently in parallel. Batch norm breaks this assumption by using statistics across the entire batch to normalize each example. This is done to control the activation statistics, but it's a departure from how other layers operate.

The rest of the initial setup is standard. I've refactored the random number generator to be initialized globally for simplicity. We define our embedding table `C` and our list of layers, which starts as a simple sequence: a linear layer, a batch norm, a `tanh`, and a final output linear layer. The optimization setup is identical to before.

The initial loss plot, however, is a mess. It's incredibly jagged because our batch size of 32 is too small, leading to very noisy loss estimates. We'll fix that shortly.

After training this initial model, we get a validation loss of `2.10`. This is pretty good, and when we sample from it, we get relatively name-like results that aren't in the training set, like "Yvonne," "Kilo," "Prosper," and "Alaia." They're reasonable, but we can definitely do better.

Refactoring for Elegance and a "PyTorch-ier" Feel

Before we change the architecture, let's clean up our code to be more robust and easier to work with.

Fixing the Jagged Loss Curve

First, let's fix that loss plot. It's daggers in my eyes! The `lossi` variable is a Python list of individual batch losses. To smooth it out, we can group these losses into chunks and average them. A neat PyTorch trick for this is to convert the list to a tensor and then `view` it as a 2D matrix.

```
# We have a 1D tensor of 200,000 losses
losses = torch.tensor(lossi)

# We can view it as a 200x1000 matrix
# Each row contains 1000 consecutive loss values
losses_reshaped = losses.view(-1, 1000)

# Now we can take the mean along the rows (dim=1)
# This gives us 200 averaged points
```

```
averaged_losses = losses_reshaped.mean(1)

# plt.plot(averaged_losses)
```

This gives us a much cleaner plot where we can clearly see the initial steep drop and the effect of the learning rate decay.

From Special Cases to Layers: `Embedding` and `Flatten`

Next, our forward pass is a bit gnarly. The embedding lookup (`C[Xb]`) and the view/concatenation operation are handled as special cases outside our main `layers` list. Let's encapsulate them into their own modules, just like we did for `Linear` and `BatchNorm1d`.

I've written two new modules, `Embedding` and `FlattenConsecutive`, that mimic existing PyTorch layers.

- `Embedding`: Holds the embedding matrix (`C`, now `self.weight`) and performs the indexing lookup in its forward pass.
- `FlattenConsecutive`: Handles the reshaping of the tensor. This is our version of `torch.nn.Flatten`. It effectively performs the concatenation of our character embeddings.

Now, we can add instances of these to our `layers` list and simplify the forward pass code significantly.

Organizing with a `Sequential` Container

We can "PyTorch-ify" our code even further. Instead of managing a raw Python list of layers, we can create a `Sequential` container module. This is a standard pattern in PyTorch (`torch.nn.Sequential`). It's a module that holds an ordered list of other modules and, in its forward pass, simply calls each layer in sequence on the input.

```
class Sequential(Module):
  def __init__(self, layers):
    self.layers = layers

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

  def parameters(self):
    # Returns all parameters from all sub-modules
    return [p for layer in self.layers for p in layer.parameters()]
```

With this, our network definition becomes much cleaner. We create a single `model` object that is a `Sequential` container. The forward pass then becomes

a single, elegant line:

```
logits = model(Xb)
```

A Classic Batch Norm Bug

After refactoring, I tried to sample from the untrained model and ran into an issue. The code produced NaNs (Not a Number). This is a classic bug that beautifully illustrates the weirdness of batch norm.

The problem was that during sampling, we feed the model a context of a **single example** (batch size = 1). The model was still in **training mode**, meaning the batch norm layer tried to calculate the mean and variance of this single-example batch. The variance of a single number is undefined (you can't measure spread with one point), which results in a NaN.

The fix is to properly set the model to evaluation mode before sampling. `for layer in model.layers: layer.training = False` This tells the batch norm layer to use its stored `running_mean` and `running_variance` instead of computing them from the batch, resolving the issue. It's a perfect example of how the statefulness of batch norm can introduce bugs if you're not careful.

Building the Hierarchical Model

Okay, with our code refactored and robust, let's build our new architecture.

The Problem with a Flat Architecture

Our current model takes 8 characters, embeds them, concatenates them into a single long vector (`8 * 10 = 80` dimensions), and feeds this into a hidden layer. This is a flat architecture. The problem is that we are **squashing all of the information from the 8 context characters into a single layer, all at once**. It's not a very productive way to combine information.

The WaveNet Idea: Progressive Fusion

The WaveNet paper proposes a more graceful, hierarchical approach. Instead of combining all 8 characters at once, we first combine characters `1` and `2`, `3` and `4`, `5` and `6`, and `7` and `8`. This gives us four "bigram" representations. Then, in the next layer, we combine these bigram representations. Finally, we combine the two resulting "four-gram" representations to make our final prediction. Information is fused progressively, layer by layer.

A New Baseline

First, let's update our dataset to use a block size of 8 instead of 3. Then, as a simple baseline, I trained the *old flat architecture* but with the larger 8-character

context. Just this simple change improved our validation loss from `2.10` to `2.02`. This shows that more context is helpful, even with a naive architecture. But we can do better by arranging our parameters more intelligently.

The Magic of Batched Matrix Multiplication

Here's the key insight that makes the hierarchical model efficient to implement. Our `Linear` layer performs a matrix multiplication: `x @ W + b`. You might think the input `x` has to be a 2D matrix (`batch_size` x `features`). However, PyTorch's `matmul` operator is more powerful than that.

You can actually pass in a 3D tensor, for example, of shape `(4, 5, 80)`. If you multiply this by a weight matrix of shape `(80, 200)`, the result will be `(4, 5, 200)`. PyTorch treats all preceding dimensions (`4` and `5` in this case) as **batch dimensions**. It performs the `80 -> 200` matrix multiplication in parallel for every single one of the `4 * 5 = 20` vectors.

This is incredibly convenient! We can use this to process our character groups in parallel.

Implementing `FlattenConsecutive`

We need to change our `Flatten` layer. Instead of squashing the `(batch, 8, 10)` tensor of embeddings into `(batch, 80)`, we want to squash it into groups. For the first layer, we want to group pairs of characters. This means we want to reshape `(batch, 8, 10)` into `(batch, 4, 20)`, where we have 4 groups, and each group is the concatenation of two 10-dimensional character embeddings.

We can achieve this with a clever use of `view`. The implementation of our new `FlattenConsecutive(n)` layer takes the number of consecutive elements `n` to flatten.

```
class FlattenConsecutive(Module):
    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) # Remove spurious dimension
        self.out = x
        return self.out
```

Assembling the New Architecture

Now we can build our hierarchical model. We'll have three hidden layers.

1. **Layer 1**: Takes the `(batch, 8, 10)` embeddings. A `FlattenConsecutive(2)` layer turns this into `(batch, 4, 20)`. A `Linear` layer then maps these

20-dimensional bigram vectors to a hidden representation (e.g., 68 dimensions).
2. **Layer 2**: The input is `(batch, 4, 68)`. Another `FlattenConsecutive(2)` turns this into `(batch, 2, 136)`. A `Linear` layer maps these 136-dimensional four-gram vectors to the hidden dimension.
3. **Layer 3**: The input is `(batch, 2, 68)`. A final `FlattenConsecutive(2)` turns this into `(batch, 1, 136)`, which is then squeezed to `(batch, 136)`. This is our final representation of the full 8-character context, which is fed into the last layers to produce the logits.

When I first built this network with the same parameter count as the flat baseline (`~22,000`), the performance was roughly identical (`2.02`). This was suspicious. It suggested something was wrong.

An Even More Subtle Batch Norm Bug

The culprit, once again, was **batch norm**. Our `BatchNorm1d` layer was written assuming a 2D input (`batch_size`, `features`). It was only taking the mean over the first dimension.

When we fed it a 3D input of shape `(32, 4, 68)`, it was still only averaging over the first dimension (the 32). This meant it was treating the `4 * 68` dimensions as independent features. It was calculating separate means and variances for each of the 4 bigram positions, instead of treating the 4 positions as another batch dimension. We want to normalize each of the 68 channels *across all examples and all positions*.

The fix is to tell `mean` and `var` to reduce over a tuple of dimensions: `(0, 1)`.

```
# In BatchNorm1d
if x.ndim == 2:
    dim = 0
elif x.ndim == 3:
    dim = (0, 1) # Reduce over batch and sequence/group dimension

xmean = x.mean(dim, keepdim=True)
xvar = x.var(dim, keepdim=True)
```

After fixing this bug, the validation loss improved slightly, from `2.029` to `2.022`. The fix helps because we are now getting much more stable estimates for our running statistics, as we're averaging over `32 * 4 = 128` numbers for each channel instead of just 32.

Retraining and Scaling Up

With the more general and correct architecture in place, we are now set up to scale the network. I increased the embedding size to 24 and the number of hidden units, bringing the total parameters to ~76,000. After a longer training run, we finally crossed the `2.0` barrier!

Final Validation Loss: 1.993

Concluding Thoughts and Future Directions

We've successfully improved our model's performance from `2.10` down to `1.99`, but the specific number isn't the main takeaway. The process itself is what's important.

The Connection to Convolutions

I want to briefly preview how what we've built relates to the **Convolutional Neural Networks (CNNs)** used in the WaveNet paper. The use of convolutions is strictly for **efficiency**; it doesn't change the model we've implemented conceptually.

Right now, to process a sequence like "DeAndre," we run our model 8 times independently. A convolution allows you to slide our hierarchical model (which acts as a "filter") efficiently across the input sequence. This is faster for two reasons:

1. The sliding loop is implemented in highly optimized C++/CUDA kernels, not slow Python.
2. It allows for the **reuse of intermediate computations**. In our tree structure, many nodes are children of one parent and parents of another. A convolution calculates each node's value only once and reuses it where needed.

So, you can think of what we built as a single application of a **dilated causal convolution**. The "convolution" part is just sliding it efficiently.

Reflections on the Development Process

I hope this lecture gave you a sense of what building deep neural networks is actually like:

- **Reading Documentation:** We spend a lot of time in the PyTorch docs. Unfortunately, they can sometimes be incomplete, unclear, or even wrong. You have to be resilient and do your best.
- **Shape Gymnastics:** A huge amount of time is spent wrestling with the shapes of multi-dimensional tensors, ensuring layers are compatible, and transposing/viewing/squeezing them into the right format.
- **Prototyping Workflow:** I very often prototype layers and logic in a Jupyter notebook to quickly check shapes and functionality. Once I'm satisfied, I copy that code into my main repository (e.g., in VS Code) to run larger experiments.

Where Do We Go From Here?

This lecture unlocks many potential future topics:

- Implementing **actual convolutional layers** for efficiency.
- Exploring **residual and skip connections**, which are part of the full WaveNet architecture.
- Building a proper **experimental harness** to run and track many experiments, which is essential for serious deep learning work.
- Covering other major architectures like **RNNs, LSTMs, GRUs, and Transformers**.

A Challenge for You

I think it's very possible to beat my score of **1.993**. I haven't tried very hard to tune the hyperparameters. You could try:

- Different allocations of hidden units across the layers.
- Different embedding dimensions.
- Going back to the flat network and just making it very large (it would be embarrassing if that beats our hierarchical one!).
- Reading the WaveNet paper and implementing the more complex gated activations or residual connections.
- Tuning the optimization parameters like learning rate.

I'd be curious to see if you can find ways to improve the model. That's all for now, bye!