Hello everyone! Today, we're diving back into our `makemore` implementation. So far, we've built up to a Multi-Layer Perceptron (MLP) that looks something like this, which we've been implementing over the last few lectures.

I know everyone's eager to jump into Recurrent Neural Networks (RNNs), LSTMs, and all their cool variants. The diagrams look awesome, the concepts are exciting, and they promise better results. But, unfortunately, we're going to stay with our MLP for one more session.

Why? Well, we've already trained this MLP and are getting a pretty good loss. We have a decent grasp of the architecture. However, there's one line of code that I want to take issue with: `loss.backward()`. We've been relying on PyTorch's `autograd` engine to magically calculate all our gradients. Today, I want to rip that out and write the entire backward pass ourselves, manually, at the tensor level.

I believe this is an incredibly useful exercise for a few key reasons.

Why Manual Backpropagation? The Leaky Abstraction

I've written a whole blog post on this, but the gist is that **backpropagation is a leaky abstraction**. It's not a magic wand that makes neural networks "just work." You can't just stack up arbitrary differentiable functions, call `.backward()`, cross your fingers, and expect greatness.

It's "leaky" because if you don't understand its internals, you can easily shoot yourself in the foot. Your network might fail to train, or train sub-optimally, for reasons that seem mysterious. To debug these issues, you *need* to understand what's happening under the hood. We've already touched on some of these failure modes:

- **Saturated Activations:** The flat tails of functions like `tanh` can cause gradients to become zero, effectively "killing" them.
- **Dead Neurons:** Neurons that get stuck outputting values in these saturated regions stop learning.
- **Exploding or Vanishing Gradients:** A major issue in RNNs (which we're about to cover) where gradients either grow exponentially or shrink to nothing as they propagate through time.

You'll also find subtle but critical bugs in the wild that stem from a misunderstanding of backprop. For instance, I found this snippet in a random codebase where the author was trying to clip the *gradients* by clipping the *loss*.

```
# A subtle but major bug
loss = loss.clamp(max=3) # trying to do gradient clipping
```

What this actually does is set the gradient to zero for any outlier example whose loss exceeds the threshold of 3. This causes the model to completely

ignore those examples during training, which is not what the author intended. Understanding backprop helps you avoid these pitfalls.

So, just because PyTorch offers `autograd`, it doesn't mean we can ignore how it works. We've already built `micrograd`, an autograd engine on the level of individual scalars. Now, it's time to level up and understand backpropagation on tensors.

In short, this exercise will:

- Make you better at debugging neural networks.
- Make everything fully explicit, so you're not nervous about what's hidden away.
- Make you emerge stronger as a practitioner.

A Fun Historical Note

Today, writing your backward pass by hand is not recommended for production code; it's purely an educational exercise. But about 10 years ago, it was the standard, pervasive way to do deep learning. Everyone, including myself, manually wrote their backward passes. We've lost something since everyone started just calling `loss.backward()`.

For example, here's a seminal 2006 paper from Geoff Hinton and Ruslan Salakhutdinov that trained architectures called Restricted Boltzmann Machines. Back then, around 2010, I had my own library for this, written in MATLAB. Python wasn't the king of deep learning yet; MATLAB, a scientific computing package with a convenient tensor class, was what everyone used. The code ran on the CPU, but you got nice plots and a built-in debugger.

The code is somewhat recognizable today: initializing weights, looping through batches, doing a forward pass. But instead of `autograd`, it implemented an algorithm called Contrastive Divergence to estimate the gradient, and then you'd see people meddling with those gradients directly to perform the parameter update.

Here's another example from a 2014 paper of mine, which was an early precursor to models like CLIP. It aligned image regions with text fragments. I dug up the code, written in Python and NumPy. It was standard practice to implement not just the cost function but also the manual backward pass.

```
# Snippet from 2014 code
# ... forward pass to calculate loss ...
loss = ...

# ... manual backward pass ...
d_loss = 1.0
# backpropagate through all the layers by hand...
```

```
# ...
# append regularization gradients...
```

After writing out the backward pass, you would use a **gradient checker** to compare your analytical gradient with a numerical estimate to ensure it was correct. This was the workflow for a long time. Today, autograd engines handle this, but the intuitive understanding that came from doing it manually was invaluable. And that's the level of understanding we want to achieve.

Setting the Stage: Our Neural Network

As a reminder, here's the Jupyter Notebook from our last lecture. We're keeping the architecture the same: a two-layer MLP with a batch normalization layer. The forward pass will be nearly identical. The big change is getting rid of `loss.backward()` and writing our own.

The Starter Code

In our new notebook, "Becoming a Backprop Ninja," the first few cells are unchanged. We do our imports, load the dataset, and build the vocabulary.

I've introduced a new utility function, `cmp`, which we'll use to compare our manually calculated gradients against the ones PyTorch computes. This will be our ground truth to check for correctness.

```
def cmp(s, dt, t):
  # Compares our calculated gradient 'dt' with PyTorch's 't.grad'
  ex = torch.all(dt == t.grad).item()
  ap = torch.allclose(dt, t.grad)
  maxdiff = (dt - t.grad).abs().max().item()
  print(f'{s:15s} | exact: {str(ex):5s} | approx: {str(ap):5s} | maxdiff: {maxdiff}')
```

Next is the parameter initialization. I've made two small but important changes:

1. **I'm initializing biases to small random numbers, not zero.** If variables are exactly zero, the gradient expressions can sometimes simplify in a way that masks an incorrect implementation. Using small random numbers helps unmask these potential bugs.
2. **I'm including the bias `b1` in the first linear layer, even though it's followed by a batch norm layer.** We discussed that this bias is redundant because batch norm re-centers the data anyway. However, I'm keeping it here for the fun of the exercise, so we can calculate its gradient and check that we're doing it correctly, even though the parameter itself is spurious.

Finally, I've expanded the forward pass. It's much longer now, not because the logic has changed, but because I've broken it down into many small, manageable chunks. We have a lot more intermediate tensors.

```
# Original compact forward pass
# h = torch.tanh(X @ W1 + b1) # ... and so on

# New expanded forward pass
emb = C[Xb] # (32, 3, 10)
h_preact = emb.view(emb.shape[0], -1) @ W1 + b1 # (32, 200)
# ... many more intermediate steps ...
loss = -logprobs[torch.arange(n), Yb].mean()
```

The reason for this is that we're about to go backward, from the `loss` all the way to the top. For every intermediate tensor like `logprobs`, we're going to calculate its corresponding gradient, which we'll call `dlogprobs` (for derivative of loss with respect to `logprobs`). We'll prepend a `d` to every variable name as we compute its gradient during backpropagation.

Exercise 1: The Grand Tour of Backpropagation

Okay, let's get to it. I've run the notebook up to the first exercise. Our task is to fill in the blanks and compute the gradient for each intermediate variable, working our way backward from the loss.

A quick pedagogical note: The best way to learn this is to try it yourself! Find the link to the notebook in the video description, open it in Google Colab, and try to derive and implement these gradients. If you get stuck, come back to the video/text and see how I did it.

Step 1: `dlogprobs`

We start at the very end. The loss is calculated from `logprobs` like this:

```
loss = -logprobs[torch.arange(n), Yb].mean()
```

The `logprobs` tensor has a shape of `(32, 27)`. The indexing `[torch.arange(n), Yb]` plucks out the log probability of the *correct* next character for each of the 32 examples in our batch. This gives us a tensor of 32 values. We then take their mean and negate it to get the final scalar `loss`.

So, how does the `loss` depend on the elements of `logprobs`?

Let's use a simpler example. If we had 3 elements, `a`, `b`, and `c`, the loss would be `loss = -(a + b + c) / 3`. If we expand this, we get `loss = -1/3 * a - 1/3 * b - 1/3 * c`. The derivative of the loss with respect to `a` is simply `loss/ a = -1/3`.

Generalizing this, for the 32 elements in our batch that were plucked out and contributed to the loss, their gradient will be `-1/n` (where `n=32`). What about all the *other* elements in the `(32, 27) logprobs` tensor? They had no effect

on the final loss value. If you change them, the loss doesn't change. Therefore, their gradient is **zero**.

Here's how we can implement that:

```
# Exercise 1
# This is the first gradient, dloss/dlogprobs
dlogprobs = torch.zeros_like(logprobs)
dlogprobs[torch.arange(n), Yb] = -1.0/n
```

We create a tensor of zeros with the same shape as `logprobs`, and then we place `-1.0/n` at the specific locations corresponding to the correct characters.

Running our `cmp` function… `dlogprobs | exact: True | approx: True | maxdiff: 0.0` An exact match! We're off to a good start.

Step 2: `dprobs`

The `logprobs` tensor was created by taking an element-wise logarithm of the `probs` tensor: `logprobs = probs.log()`.

This is a simple element-wise operation. From the chain rule, the gradient `dprobs` is the **local gradient** of the `log()` operation multiplied by the **incoming gradient**, `dlogprobs`.

The derivative of `log(x)` with respect to `x` is `1/x`. So, the local gradient is `1 / probs`.

```
# dloss/dprobs
dprobs = (1.0 / probs) * dlogprobs
```

Let's check it. `dprobs | exact: True | approx: True | maxdiff: 0.0` Perfect!

Step 3: `dcount_sum_inv` and Broadcasting

Next up, we backpropagate through this line: `probs = counts * count_sum_inv`.

We have to be careful here because of tensor shapes and **broadcasting**.

- `counts` has shape `(32, 27)`.
- `count_sum_inv` has shape `(32, 1)`.

To perform the element-wise multiplication, PyTorch implicitly **broadcasts** `count_sum_inv`. It replicates the `(32, 1)` column vector 27 times horizontally to create a `(32, 27)` matrix that matches the shape of `counts`.

This looks like a single multiplication, but it's really two sequential operations: **replication**, then **multiplication**. When we backpropagate, we have to undo both.

Let's first backpropagate through the multiplication. If `c = a * b`, then `c/ b = a`. The local derivative with respect to `count_sum_inv` is `counts`. So, the gradient on the *replicated* tensor is `counts * dprobs`.

Now, we need to backpropagate through the replication. When a variable (like an element of `count_sum_inv`) is used multiple times in the forward pass, its gradients from all those uses must be **summed** in the backward pass. Since each element of `count_sum_inv` was replicated across a row, we must sum the gradients horizontally.

```
# dloss/dcount_sum_inv
dcount_sum_inv = (counts * dprobs).sum(1, keepdim=True)
```

We sum along dimension 1 (the columns) and use `keepdim=True` to maintain the `(32, 1)` shape.

Let's check. `dcount_sum_inv | exact: True | approx: True | maxdiff: 0.0` Success!

Step 4: `dcounts` (Part 1)

Now for the other input to the multiplication, `counts`. Here, `c/ a = b`. The local derivative is `count_sum_inv`.

```
# dloss/dcounts (first branch)
dcounts = count_sum_inv * dprobs
```

Here, `count_sum_inv` (`(32, 1)`) is broadcast to match `dprobs` (`(32, 27)`), which is exactly what we need. There's no extra sum required for this part.

However, we can't check this gradient yet! Why? Because `counts` is used in another branch of the computation graph to calculate `count_sum_inv`. We've only calculated the gradient contribution from one branch. We need to calculate the gradient from the other branch and **add** them together.

Step 5: `dcount_sum`

The variable `count_sum_inv` comes from `count_sum_inv = count_sum.pow(-1)`. This is equivalent to `1 / count_sum`.

The derivative of x^{-1} is $-x^{-2}$. So the local derivative is `-count_sum.pow(-2)`.

```
# dloss/dcount_sum
dcount_sum = (-count_sum.pow(-2)) * dcount_sum_inv
```

Let's check it. `dcount_sum | exact: True | approx: True | maxdiff: 0.0` Correct.

Step 6: `dcounts` (Part 2 and Final)

`count_sum` comes from `count_sum = counts.sum(1, keepdim=True)`. This is a sum in the forward pass.

As we've seen, a **sum** in the forward pass becomes a **replication/broadcast** in the backward pass. The gradient from `dcount_sum` (a `(32, 1)` column vector) needs to be "broadcast" back to all the elements of `counts` that contributed to it. In other words, each element in a row of `counts` gets the same gradient from `dcount_sum`.

We can achieve this by creating a tensor of ones and letting PyTorch's broadcasting do the work.

```
# dloss/dcounts (second branch)
# The first branch was: dcounts = count_sum_inv * dprobs
dcounts += torch.ones_like(counts) * dcount_sum
```

Notice the `+=`. We are adding the gradient from this second branch to the gradient we calculated earlier. Now that we've accounted for all paths, we can check the final `dcounts`.

`dcounts | exact: True | approx: True | maxdiff: 0.0` And it's correct! This pattern of summing gradients from multiple branches is fundamental to backpropagation.

Step 7: `dnorm_logits`

The `counts` tensor was created via `counts = norm_logits.exp()`.

The derivative of `e` is famously `e` . The local derivative is `norm_logits.exp()`, which is just the `counts` tensor itself!

```
# dloss/dnorm_logits
dnorm_logits = counts * dcounts # local_grad * incoming_grad
```

Let's check. `dnorm_logits | exact: True | approx: True | maxdiff: 0.0` Looks good.

Step 8: `dlogits_max` and `dlogits` (Part 1)

Now we're at `norm_logits = logits - logits_max`.

This is a subtraction with broadcasting. `logits` is `(32, 27)` and `logits_max` is `(32, 1)`. The `logits_max` column vector is broadcast across the 27 columns.

For `dlogits`, the gradient just flows through. The derivative of `a - b` with respect to `a` is 1. So `dlogits` gets a copy of `dnorm_logits`. For `dlogits_max`, the local derivative is -1. So it gets `-dnorm_logits`. But because `logits_max` was broadcast, we need to **sum** the gradients across the dimension it was broadcast (dimension 1).

```
dlogits = dnorm_logits.clone()
dlogits_max = (-dnorm_logits).sum(1, keepdim=True)
```

The use of `.clone()` here is a subtle but important piece of defensive programming. Without it, `dlogits` would just be a reference to the `dnorm_logits`

tensor. The in-place update `+=` we will perform later would then also modify `dnorm_logits`, corrupting our calculations. `.clone()` creates an independent copy in memory, preventing this issue.

We can check `dlogits_max` now. `dlogits_max | exact: True | approx: True | maxdiff: 0.0` Correct. But wait, `dlogits` isn't finished. It has another gradient path coming from the calculation of `logits_max` itself.

One interesting aside: the purpose of subtracting `logits_max` is purely for numerical stability in the softmax. It doesn't change the final probabilities or the loss. Therefore, we'd expect its gradient, `dlogits_max`, to be zero. And indeed, if you inspect the tensor, it's full of extremely small numbers like `1e-9`. Due to floating-point wonkiness, it's not exactly zero, but it's practically zero, confirming our intuition.

Step 9: `dlogits` (Part 2 and Final)

`logits_max` was calculated using `logits_max = logits.max(1, keepdim=True).values`.

A `max` operation in the forward pass acts like a router in the backward pass. The gradient only flows back to the element that was the maximum. All other elements that were not the max get a gradient of zero from this path.

We need to know the *indices* of the max values to route the `dlogits_max` gradient correctly. PyTorch's `.max()` can return these indices.

```
# Create a one-hot mask to select only the max elements
# F.one_hot creates a mask where the '1' is at the location of the max index
dlogits += F.one_hot(logits.max(1).indices, num_classes=logits.shape[1]) * dlogits_max
```

We use `F.one_hot` to create a `(32, 27)` mask that is 1 at the position of the maximum value in each row and 0 elsewhere. We then multiply this mask by `dlogits_max` (which will be broadcast) to route the gradient to the correct locations. Again, we use `+=`.

Let's check the final `dlogits`. `dlogits | exact: True | approx: True | maxdiff: 0.0` Perfect.

Step 10: Backpropagating Through a Linear Layer

We have now arrived at our second linear layer: `logits = h @ W2 + b2`.

We have `dlogits` and want to find `dh`, `dW2`, and `db2`. You might think we need to dive into a matrix calculus textbook, but we can derive this from first principles. If you write out a tiny 2x2 matrix multiplication, you can see it's just a bunch of adds and multiplies. By applying the chain rule to these simple operations, you can derive the general formulas.

Long story short, the backward pass of a matrix multiplication `A @ B` is another matrix multiplication. The gradients are:

- `dA = dD @ B.T`
- `dB = A.T @ dD`
- `dC = dD.sum(0)` (for a broadcasted bias `C`)

A useful hack is that you don't even need to memorize the transposes. You just need to know the shapes of the gradients you want to produce. For example, `dh` must have the same shape as `h`, which is `(32, 64)`. We have `dlogits (32, 27)` and `W2 (64, 27)`. The only way to matrix-multiply them to get a `(32, 64)` result is `dlogits @ W2.T`. This dimension-matching trick almost always works.

```
# dloss/dh, dloss/dW2, dloss/db2
dh = dlogits @ W2.T
dW2 = h.T @ dlogits
db2 = dlogits.sum(0)
```

Let's check all three. `dh | exact: True | approx: True | maxdiff: 0.0 dW2 | exact: True | approx: True | maxdiff: 0.0 db2 | exact: True | approx: True | maxdiff: 0.0` All correct. We've successfully backpropagated through a linear layer.

Step 11: Backpropagating Through `tanh`

The hidden state `h` was the output of a `tanh` activation: `h = torch.tanh(h_preact)`.

We've done this in `micrograd`. The derivative of `tanh(x)` can be conveniently expressed in terms of its *output*, `y = tanh(x)`. The derivative is `1 - y**2`. In our case, the output is `h`.

```
# dloss/dh_preact
# The local derivative of tanh is 1 - h**2
dh_preact = (1.0 - h**2) * dh
```

Checking it… `dh_preact | exact: True | approx: True | maxdiff: 0.0` Correct again!

The Final Steps: Batch Norm and Embeddings

The rest of the process continues this pattern, navigating the many small steps of our expanded batch norm layer and finally arriving at the first linear layer and the embedding lookup.

The embedding lookup (`emb = C[Xb]`) is an indexing operation. In the backward pass, we need to take the gradients from `demb` and **add** them to the correct rows in `dC`. If a character embedding (a row in `C`) was used multiple times in the batch, its gradients from all those uses will accumulate. The lecturer implemented this with a for-loop for clarity.

```
# dloss/dC
dC = torch.zeros_like(C)
```

```
for k in range(Xb.shape[0]):
  for j in range(Xb.shape[1]):
    ix = Xb[k,j]
    dC[ix] += demb[k,j]
```

While this loop is pedagogically clear, the speaker hints that a more efficient method exists. In practice, you would use a highly optimized, vectorized PyTorch operation like `index_add_` to avoid slow Python loops and achieve the same result much faster.

After going through this entire beast, we have successfully calculated the gradients for all parameters (`dC`, `dW1`, `db1`, `dbn_gain`, `dbn_bias`, `dW2`, `db2`) by hand!

Exercise 2: A More Efficient Cross-Entropy Backward Pass

In Exercise 1, we did way too much work. Breaking down the loss calculation into tiny atomic pieces is great for learning, but it's inefficient. If we consider the entire operation from `logits` to `loss` as a single mathematical function (`F.cross_entropy`), we can derive its gradient analytically. This is much faster.

The function is essentially a **Softmax** followed by a **Negative Log Likelihood**.
$L = -$
$log(p\_y) = -$
log
$left($
$frac{l\_y}{sum\_j e^{l\_j}}$
$right)$ where l_j is the logit for class j, and y is the index of the correct class.

If you do the calculus and find the derivative `L/ l` , the result is beautifully simple:

$fracpartialLpartiall\_i = p\_i -$
$delta\_iy$

where $p\_i =$
$frac{l\_i}{sum\_j e^{l\_j}}$ is the probability from the softmax, and
$delta\_iy$ is 1 if $i = y$ (the correct class) and 0 otherwise.

In other words, the gradient on the logits is just **`probabilities - 1`** at the location of the correct answer, and **`probabilities`** everywhere else.

```
# An efficient implementation for dlogits
probs = F.softmax(logits, 1) # Calculate probabilities

dlogits = probs.clone()
dlogits[torch.arange(n), Yb] -= 1
dlogits /= n # Don't forget to scale by batch size due to the .mean() in the loss
```

When we compare this efficient calculation to what PyTorch computes for the fused `F.cross_entropy` layer, we get an almost perfect match (the max difference is tiny, around `5e-9`, due to floating-point arithmetic).

The Intuition of the Cross-Entropy Gradient

This gradient has a wonderful, intuitive explanation. Think of the gradients as **forces** acting on the logits.

- For all the **incorrect** characters, the gradient is just their probability (`p`). Since we do gradient *descent* (i.e., we move in the *opposite* direction of the gradient), this pushes their logits down. The strength of the push is proportional to how confidently the model predicted them.
- For the **correct** character, the gradient is `p - 1`. This value is negative, so gradient descent pushes its logit *up*.

The sum of all gradients in a row is zero. This means the total "pull-up" force on the correct logit is perfectly balanced by the total "push-down" force on all the incorrect ones. The network is essentially playing a game of tug-of-war, trying to increase the score of the right answer while decreasing the scores of the wrong ones. The more wrong the network is, the stronger the corrective forces. It's a beautiful, self-regulating mechanism.

Exercise 3: Taming the Batch Norm Beast

We're going to apply the same principle to batch normalization. Instead of backpropagating through its many tiny pieces, we'll treat it as a single mathematical function and derive its backward pass analytically. This is a rite of passage for any deep learning practitioner.

The formula for batch norm is: $y\_i =$
$gamma$
$fracx\_i - musqrtsigma^2 + epsilon +$
$beta$

The math to derive its gradient gets quite hairy, involving two full pages in the lecturer's notes. The key challenge is that each input `x` affects the final loss through **three distinct paths**:

1. **Directly:** Through its own normalized value, `x̂` .
2. **Indirectly via the mean:** By contributing to the batch mean, , which affects *every* output.
3. **Indirectly via the variance:** By contributing to the batch variance, ², which also affects *every* output.

To get the total gradient `L/ x` , you must meticulously apply the multivariate chain rule to find the gradient contribution from each of these three paths and then sum them together.

After a lot of algebra, you arrive at a final expression for `L/ x`. It's complex, but it's a closed-form solution that can be implemented efficiently using tensor operations.

```
# Simplified Batch Norm Backward Pass (ignoring gamma/beta for clarity)
# This is a beast to implement correctly with broadcasting!
dh_prebn = (
    (1/n) * bn_gain * bn_var_inv * (n*dh_preact - dh_preact.sum(0) - n/(n-1)*bn_raw * (dh_pr
)
```

Implementing this single line correctly is a major challenge, requiring careful attention to broadcasting to handle all 64 feature dimensions in parallel. But when you get it right, it's far more efficient than the step-by-step approach. And once again, the result matches PyTorch's gradient almost perfectly.

Exercise 4: Putting It All Together

Finally, we replace PyTorch's `loss.backward()` in our training loop with our own, shiny, hand-crafted backward pass. We use our efficient, analytical functions for cross-entropy and batch norm.

```
# The final training loop

# -- FORWARD PASS --
# (as before)

# -- BACKWARD PASS (MANUAL) --
# We use our efficient, analytical backprop functions
dlogits = F.softmax(logits, 1)
dlogits[torch.arange(n), Yb] -= 1
dlogits /= n
# ... backprop through linear layer 2 ...
dh = dlogits @ W2.T
# ... backprop through tanh ...
dh_preact = (1.0 - h**2) * dh
# ... backprop through our efficient batchnorm ...
dh_prebn = (bn_gain * bn_var_inv / n) * (n*dh_preact - dh_preact.sum(0) - n/(n-1)*bn_raw * (
# ... backprop through linear layer 1 and embedding ...
demb_cat = dh_prebn @ W1.T
demb = demb_cat.view(emb.shape)
# ... and so on for all parameters ...

# -- UPDATE --
lr = 0.1 if i < 100000 else 0.01
for p, grad in zip(params, grads):
  p.data += -lr * grad # Manual parameter update
```

We can wrap the whole training step in a `with torch.no_grad():` block to tell PyTorch it doesn't need to build its own gradient graph, which saves memory and computation. We update the parameters using our own gradients: `p.data += -lr * grad`.

And… it works! The loss goes down just as it did before, and the model generates the same kind of name-like gibberish we're used to.

`loss: 2.1288` `samples: jayde, malani, kimora, elianna, jaxtyn, ...`

We've successfully trained our neural network using a completely manual backward pass.

Conclusion

Hopefully, this exercise was interesting and demystified what happens under the hood of `loss.backward()`. We backpropagated through a diverse set of layers: linear, `tanh`, batch norm, cross-entropy, and embedding lookups.

You now have a much deeper, more intuitive sense of how gradients flow backward through a network, from the loss function all the way back to the initial embeddings, pushing and pulling on every parameter along the way. While you probably won't write these by hand in practice, this understanding is crucial for effective debugging and building novel architectures.

If you've followed along and understood the core ideas, you can count yourself as one of the buff doges on the left, not the timid cheems on the right.

In the next lecture, we'll finally move on to **Recurrent Neural Networks, LSTMs**, and their variants. We'll start building more complex architectures and achieving even better results. I'm really looking forward to it, and I'll see you then!