- The model gets **no learning signal** telling it how to make queries or keys better when the current choice is already the best one.
- It cannot learn smooth, distributed representations or subtle similarities — it's forced into hard, discrete decisions with no gradient-based way to improve.
- In practice, training with argmax attention completely fails to converge to anything useful (people have tried variants like straight-through estimator or REINFORCE, but they're unstable and far worse than softmax).

Softmax fixes this by being **smooth and everywhere differentiable**, giving non-zero gradients to **all** keys (weighted by attention probability), so the model can gradually shift mass toward better keys and away from worse ones.

Answer (b) (short version you can write):

Argmax is not differentiable (its gradient is zero almost everywhere). When the current key is already the most similar, gradients through the attention scores are exactly zero, so the model receives no signal to improve queries or keys. Softmax, by contrast, is smooth and provides non-zero gradients to all keys proportional to their attention weights, enabling effective end-to-end training.

That's it for Problem 2!

Let me know when you're ready for Problem 3 (the PyTorch multi-head attention fill-in-the-blanks) — it's a long one but very mechanical.

Awesome! Problem 3 of our assignment involves filling in some existing code to train a transformer. Let me know if you need help parsing the PDF formatting. Other than that, try filling in all of the blanks in the transformer code, and answer the additional question 3 b) at the end.

Problem 3 – Ordinary Softmax Multihead Attention Implementation

Here are all the correct fills for the code blanks in part (a):

Blank	Correct answer	Explanation
(a)	`d_model`	W_k projects from d_model → d_model (same as W_q)
(b)	`d_model`	W_v projects from d_model → d_model (standard Transformer)
(c)	`d_model`	W_o projects the concatenated heads back to d_model
(d)	`Q`	attn_scores = Q @ K^T → torch.matmul(Q, K.transpose(-2, -1))
(e)	`softmax`	We need torch.softmax to turn scores into probabilities
(f)	`V`	Weighted sum is attn_probs @ V
(g)	`self.num_heads`	We reshape to (batch, seq_len, num_heads, d_k)
(h)	`W_o`	After combining heads we still apply the output linear layer: self.W_o(...)

So the fully completed code looks like this:

Python
```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)          # (a)
        self.W_v = nn.Linear(d_model, d_model)          # (b)
        self.W_o = nn.Linear(d_model, d_model)          # (c)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1))          # (d)
        attn_scores = attn_scores / math.sqrt(self.d_k)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)
        attn_probs = torch.softmax(attn_scores, dim=-1)             # (e)
        output = torch.matmul(attn_probs, V)                        # (f)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)  # (

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).view(batch_size, seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self.W_o(self.combine_heads(attn_output))   # (h)
        return output
```

Problem 3(b)

You change the initialization to:

```Python
self.W_v = nn.Linear(in_features=d_model, out_features=2 * d_model)
```

This makes the value projections twice as wide (2 × d_model).

Question: What other minimal modifications are needed to make the code run again?

Answer:

You must also increase the size of the output projection `W_o` so that it can take the wider concatenated heads and project them back to `d_model`.

Currently:

- Each head produces vectors of size `d_k = d_model // num_heads`
- After concatenation: `num_heads * d_k = d_model`
- `W_o` is `d_model → d_model`

Now:

- Each head produces vectors of size `2 * d_k`
- After concatenation: `num_heads * 2 * d_k = 2 * d_model`
- So `W_o` must now be `2 * d_model → d_model`

Minimal changes required:

1. Change the definition of `W_o`:

   ```Python
   self.W_o = nn.Linear(2 * d_model, d_model)
   ```

2. (Optional but recommended) update the comment/docstring so it's clear the value dimension is no longer `d_model`.

That's it! Everything else (splitting heads, scaling, softmax, etc.) works unchanged because `d_k` is still `d_model // num_heads` — we only made the value vectors deeper, not the key/query dimension.

Why this works with residual connections (the hint):
Residual connections require the attention block output to have the same shape as the input (both `d_model`). After we make `W_o` go from `2*d_model → d_model`, the output shape is again `[batch, seq_len, d_model]`, so `x + attn(x)` is still valid.

Done with Problem 3!

Let me know when you want to tackle Problem 4 (the incremental decoding / Multi-Query Attention blanks) — it's another fun shape-matching exercise!

↳ Explain split_heads reshape

↳ Multi-Query Attention details