Multimodal Data Preparation

I am learning a vision language model and training it from scratch in pytorch. Can you please explain and teach me the below code to understand how the input image, input text and target text data is getting prepared to forward pass it through model

```
#Adjusting the data loader from makemore for multimodal data
def get_batch(df, batch_size, split='train', img_size=96, val_batch_size=8):
    # Split data into training and validation sets
    n = int(0.9 * len(df))  # first 90% will be train, rest val
    df_train = df.iloc[:n]
    df_val = df.iloc[n:]
    data = df_train if split == 'train' else df_val
    batch_size = batch_size if split == 'train' else val_batch_size
    replace = False if split == 'train' else True
    batch = data.sample(n=batch_size, replace=replace)

    images = torch.cat([base64_to_tensor(img, img_size) for img in batch['b64string_images']], dim=0).to(device)
    text_indices = [torch.tensor(encode(desc), dtype=torch.long) for desc in batch['caption']]
    max_length = max(len(t) for t in text_indices)

    padded_text = torch.full((batch_size, max_length), fill_value=stoi[''], dtype=torch.long).to(device)
    for i, text in enumerate(text_indices):
        padded_text[i, :len(text)] = text

    targets = torch.cat([padded_text[:, 1:], torch.full((batch_size, 1), fill_value=stoi[''], dtype=torch.long, device=device)], dim=1)

    # Truncate or pad targets to match the length of padded_text
    if targets.size(1) > padded_text.size(1):
        targets = targets[:, :padded_text.size(1)]
```

```
    elif targets.size(1) < padded_text.size(1):
        targets = torch.cat([targets, torch.full((batch_size,
padded_text.size(1) - targets.size(1)), fill_value=stoi[''],
dtype=torch.long, device=device)], dim=1)

    return images, padded_text, targets
```

This function, `get_batch`, prepares multimodal input data (images and text) for training a Vision-Language Model (VLM). Let's break it down:

1. Splitting the Dataset

```python
n = int(0.9 * len(df))
df_train = df.iloc[:n]
df_val = df.iloc[n:]
data = df_train if split == 'train' else df_val
```

- The dataset (`df`) is split into **90% training** and **10% validation**.
- Depending on `split`, it selects either `df_train` or `df_val`.

2. Sampling a Batch

```python
batch_size = batch_size if split == 'train' else val_batch_size
replace = False if split == 'train' else True
batch = data.sample(n=batch_size, replace=replace)
```

- Uses different `batch_size` for training (`batch_size`) and validation (`val_batch_size`).
- **Validation batches are sampled with replacement (** `replace=True` **)**, so the same example can appear multiple times.

3. Processing Images

```python
images = torch.cat([base64_to_tensor(img, img_size) for img in
batch['b64string_images']], dim=0).to(device)
```

- Images are stored as base64 strings (`b64string_images`).
- `base64_to_tensor(img, img_size)` decodes and resizes them to `img_size × img_size`.
- `torch.cat(..., dim=0)` stacks them into a single tensor batch.

4. Tokenizing Text Descriptions

```python
text_indices = [torch.tensor(encode(desc), dtype=torch.long) for desc in
batch['caption']]
max_length = max(len(t) for t in text_indices)
```

- Each text caption (`caption`) is tokenized using `encode(desc)`, converting it into a sequence of integer indices.
- `max_length` stores the length of the longest tokenized text in the batch.

5. Padding Text for Uniform Shape

```python
padded_text = torch.full((batch_size, max_length), fill_value=stoi[''],
dtype=torch.long).to(device)
for i, text in enumerate(text_indices):
    padded_text[i, :len(text)] = text
```

- Creates a tensor of shape `(batch_size, max_length)`, filled with padding tokens (`stoi['']`).
- Copies actual text tokens into `padded_text`, keeping the lengths intact.

6. Creating Target Tokens for Supervised Training

```python
targets = torch.cat([padded_text[:, 1:], torch.full((batch_size, 1),
fill_value=stoi[''], dtype=torch.long, device=device)], dim=1)
```

- **Targets are shifted left by one token**, meaning the model learns to predict the next token in the sequence.

- The last column is filled with padding tokens (`stoi['']`).

7. Ensuring `targets` and `padded_text` Match in Length

```python
if targets.size(1) > padded_text.size(1):
    targets = targets[:, :padded_text.size(1)]
elif targets.size(1) < padded_text.size(1):
    targets = torch.cat([targets, torch.full((batch_size, padded_text.size(1) -
targets.size(1)), fill_value=stoi[''], dtype=torch.long, device=device)], dim=1)
```

- Ensures `targets` and `padded_text` have the same shape by truncating or padding if necessary.

Final Output

```python
return images, padded_text, targets
```

- `images` → Tensor of shape `(batch_size, C, H, W)` , where `C=3` (RGB).
- `padded_text` → Tokenized and padded text input.
- `targets` → Shifted version of `padded_text` for training.

This function prepares **(image, text, target)** triplets for a **Vision-Language Model (VLM)** where the model learns to predict the next word in the text sequence while using the corresponding image for context.

Let me know if you want to deep dive into any part! 🚀

Can you please show me an example on this with a deep understanding of input output structure

Let's go through a step-by-step example with dummy data to deeply understand the input and output structure.

🔹 Example Setup

We assume:

1. **Dataset (`df`)** has images stored as **base64 strings** and text descriptions.
2. **Text is tokenized** using `encode()` , and `stoi` is a dictionary mapping tokens to indices.
3. **Images are converted to tensors** using `base64_to_tensor()` .
4. **Batch size = 2** for simplicity.

📌 Dummy Dataset (`df`)

b64string_images	caption
"image1_base64"	"A cat is cute"
"image2_base64"	"Dogs are loyal"

🔹 Step-by-Step Execution

1. Splitting Data

Let's assume we are in the training phase (`split='train'`), so we use **90% of data for training**.

2. Selecting a Batch

Since `batch_size=2` , we randomly select both rows.

---

3. Converting Images to Tensors

Each **base64 string** is converted to a **3-channel image tensor** of size `(3, 96, 96)` .
Assuming:

```python
```

```
base64_to_tensor("image1_base64", 96) → torch.ones(3, 96, 96)
base64_to_tensor("image2_base64", 96) → torch.zeros(3, 96, 96)
```

So,

```python
images.shape = (2, 3, 96, 96)
```

🔹 **Images Tensor Output:**

```ini
images = tensor([
    [[[1, 1, ...], [1, 1, ...], ...],  # Image 1
     [[1, 1, ...], [1, 1, ...], ...],
     [[1, 1, ...], [1, 1, ...], ...]],

    [[[0, 0, ...], [0, 0, ...], ...],  # Image 2
     [[0, 0, ...], [0, 0, ...], ...],
     [[0, 0, ...], [0, 0, ...], ...]]
])
```

---

4. Tokenizing Text

Assume:

```python
stoi = {'': 0, 'A': 1, 'cat': 2, 'is': 3, 'cute': 4,
        'Dogs': 5, 'are': 6, 'loyal': 7}

encode("A cat is cute") → [1, 2, 3, 4]
encode("Dogs are loyal") → [5, 6, 7]
```

🔹 **Tokenized Captions:**

```ini
```

```
text_indices = [
    tensor([1, 2, 3, 4]),  # "A cat is cute"
    tensor([5, 6, 7])      # "Dogs are loyal"
]
```

---

5. Padding Text for Uniform Shape

Since the longest sequence has **4 tokens**, we pad shorter sequences (`Dogs are loyal`) with `stoi[''] = 0`.

🔹 **Padded Text Tensor:**

```csharp
padded_text =
[
    [1, 2, 3, 4],  # "A cat is cute"
    [5, 6, 7, 0]   # "Dogs are loyal" (padded with 0)
]
```

```python
padded_text.shape = (2, 4)
```

---

6. Creating Target Tokens

For **language modeling**, targets are **shifted left**.

Before Shifting (padded_text)

```csharp
[
    [1, 2, 3, 4],  # "A cat is cute"
```

```
    [5, 6, 7, 0]   # "Dogs are loyal"
]
```

After Shifting (targets)

```csharp
[
    [2, 3, 4, 0],  # "cat is cute" + padding
    [6, 7, 0, 0]   # "are loyal" + padding
]
```

🔹 **Final Targets Tensor:**

```python
targets.shape = (2, 4)
```

🔹 Final Output

```python
return images, padded_text, targets
```

📌 Output Structures

1. `images` : (2, 3, 96, 96) → (Batch, Channels, Height, Width)
2. `padded_text` : (2, 4) → Tokenized input with padding
3. `targets` : (2, 4) → Shifted version of padded_text for training

---

🔹 Final Understanding

✅ Images are batch-stacked and ready for vision processing
✅ Text is tokenized, padded, and ready for the transformer model

✅ **Targets are shifted left to train a next-word prediction model**

This **multimodal batch** is then fed into the model for training.

Let me know if you need more clarification! 🚀

But, dont you think from a single Image and caption pair we can generate multiple input output examples?

Yes, absolutely! From a **single image-caption pair**, we can generate **multiple input-output examples** by using different **sub-sequences of the caption** as input while training the model. This approach is commonly used in **causal language modeling** where the model learns to predict the next token given a partial sequence.

🔹 How Multiple Input-Output Examples Are Generated

Instead of using the **entire caption as input**, we generate **multiple (input, target) pairs** by progressively increasing the input length.

📌 Example

Caption: `"A cat is cute"`

Tokenized Sequence: `[1, 2, 3, 4]`

(Let's assume token IDs: `A=1`, `cat=2`, `is=3`, `cute=4`)

Input Sequence (padded_text)	Target (Next Token)
`[1]`	`[2]`
`[1, 2]`	`[2, 3]`
`[1, 2, 3]`	`[2, 3, 4]`

So, from a single caption, **3 training examples** are created.

🔹 Adjusting `get_batch` for This

Instead of having `padded_text` as a full sentence, we generate **progressively growing input-target pairs**.

🔹 Modified Code

python

```python
def get_batch(df, batch_size, split='train', img_size=96, val_batch_size=8):
    # Splitting dataset
    n = int(0.9 * len(df))
    df_train = df.iloc[:n]
    df_val = df.iloc[n:]
    data = df_train if split == 'train' else df_val
    batch_size = batch_size if split == 'train' else val_batch_size
    replace = False if split == 'train' else True
    batch = data.sample(n=batch_size, replace=replace)

    # Convert images
    images = torch.cat([base64_to_tensor(img, img_size) for img in
batch['b64string_images']], dim=0).to(device)

    all_padded_inputs = []
    all_targets = []

    for desc in batch['caption']:
        tokenized = torch.tensor(encode(desc), dtype=torch.long)
        for i in range(1, len(tokenized)):  # Generate multiple (input, target)
pairs
            input_seq = tokenized[:i]  # Input is prefix
            target_seq = tokenized[1:i+1]  # Target is next token

            all_padded_inputs.append(input_seq)
            all_targets.append(target_seq)

    # Padding to max length in batch
    max_len = max(len(seq) for seq in all_padded_inputs)

    padded_inputs = torch.full((len(all_padded_inputs), max_len),
fill_value=stoi[''], dtype=torch.long).to(device)
    padded_targets = torch.full((len(all_targets), max_len), fill_value=stoi[''],
```

```
dtype=torch.long).to(device)

    for i, (inp, tgt) in enumerate(zip(all_padded_inputs, all_targets)):
        padded_inputs[i, :len(inp)] = inp
        padded_targets[i, :len(tgt)] = tgt

    return images.repeat(len(padded_inputs), 1, 1, 1), padded_inputs, padded_targets
```

🔹 Example Output

Input Captions in Batch

```arduino
"A cat is cute"
"Dogs are loyal"
```

Generated Input-Target Pairs

Image	Input (padded_text)	Target (targets)
🖼️ Image 1	`[1]`	`[2]`
🖼️ Image 1	`[1, 2]`	`[2, 3]`
🖼️ Image 1	`[1, 2, 3]`	`[2, 3, 4]`
🖼️ Image 2	`[5]`	`[6]`
🖼️ Image 2	`[5, 6]`	`[6, 7]`

This way, each **image-caption pair** generates multiple training examples. This improves **data efficiency** by using every token position as a learning opportunity.

🔹 Why is This Useful?

✅ **More training examples per image**
✅ **Helps model learn next-token prediction**

✅ **Mimics real-world sequential text generation**

This approach is often used in **transformers like GPT**, where the model **auto-regressively** predicts tokens one at a time.

Would you like me to refine this further based on your dataset? 🚀

In the actual get_batch function, which I have provided to you, is it creating multiple examples from a single Image-Caption pair? Can you confirm?

No, the `get_batch` function **you provided** is **not** generating multiple examples from a single image-caption pair. It treats each image-caption pair as a **single input-output instance** in the batch.

🔹 Why?

1. **Each caption is tokenized once:**

```python
text_indices = [torch.tensor(encode(desc), dtype=torch.long) for desc in batch['caption']]
```

   - This creates **one tensor per caption** instead of generating multiple sub-sequences.

2. **Padding is done at the batch level:**

```python
padded_text = torch.full((batch_size, max_length), fill_value=stoi[''], dtype=torch.long).to(device)
```

   - Each **caption** is padded to match the longest caption in the batch.
   - There is **no progressive increase in input length**.

3. **Targets are just a shifted version of padded text:**

```python
```

```
targets = torch.cat([padded_text[:, 1:], torch.full((batch_size, 1),
fill_value=stoi[''], dtype=torch.long, device=device)], dim=1)
```

- It **shifts the entire sequence** by one step.
- There is **only one training example per caption** (instead of multiple progressive inputs).

🔹 Confirmation

✅ **It processes each image-caption pair as a single unit**
✅ **No sub-sequence generation from captions**
✅ **Only one input-output pair per image in the batch**

If you want to generate **multiple input-output examples per image**, you'll need to modify the function. Do you want me to adjust it for that? 🚀