Image to Text Model for Kamon

Your task is to create an image-to-text model that converts images of Japanese *kamon* (家紋) into a verbal description of the image. For example

would be converted to: 丸に蔦.

We assume the output text is already tokenized and padded to the length L of the longest description, with an <EOS> tag to the right. Thus if the longest actual description is 5 tokens, L = 5 + 1 = 6, and assuming the above string is tokenized as “丸”, “に”, and “蔦”, then the tokenized and padded input will be:

丸 に 蔦 <EOS> <EOS> <EOS>

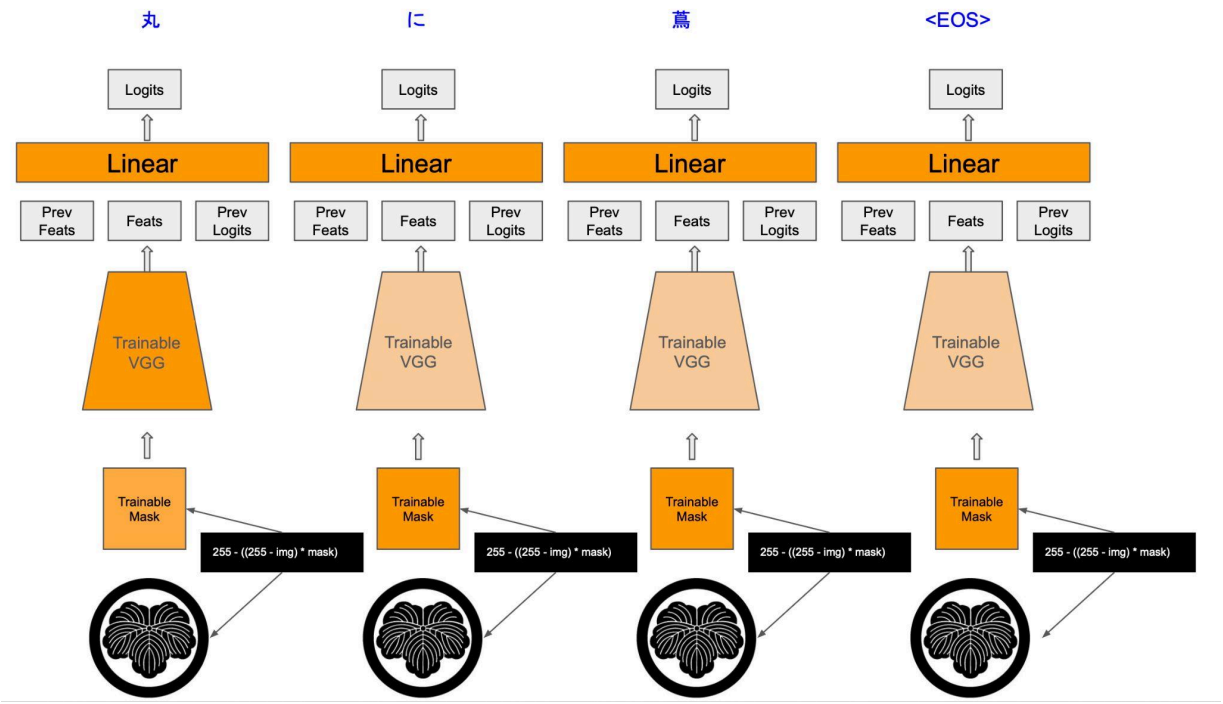
For the input, we assume a sequence of L masks, which are to be trained to mask out various portions of the input image: such masks should be sensitive to the form of the input image: i.e., ideally their trained value would depend on the input image itself so that at a given position for the above image, the outer ring might get masked out, but for another image such as

the outer frame might be masked out.

The output of the masked image at each position is then fed through a trainable VGG16 model to extract features. These features are then combined with the features from the previous position (or fixed dummy features if we are considering the initial position), and the previously predicted logits (or dummy logits if initial), which is then fed into a Linear classifier to produce the logits for the current position.

There should be a single trainable VGG16 model that should be shared across positions.

The system just described can be schematized as in the following diagram:



What was just described is a bigram language model for this task. But your code should be general enough that we can extend the ngram length as desired.

You should assume PyTorch.

For the VGG features, one would typically chop off the last layer in constructing the model, something like the following, which will expose the raw features:

```
def construct_vgg_classifier(self) -> vgg16:
  vgg_model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
  vgg_model.classifier = vgg_model.classifier[:-1]
  params = [p for p in vgg_model.parameters()]
  if not self.also_train_vgg:
    for p in params:
      p.requires_grad = False
  vgg_model.eval()
  self.vgg_nfeatures = params[-1].shape[-1]
  self.vgg = vgg_model
```

Write a class definition for the above described model, and a training script ‘train.py’ that invokes the model. The loss should be standard cross-entropy loss.

Assume a data reader loaded from Kamon.kamon_dataset as follows:

```
import kamon_dataset as kd
```

and invoked as follows:

```
  train_data = kd.KamonDataset(
    division="train",
    image_size=IMAGE_SIZE.value,
    num_augmentations=NUM_TRAIN_AUGMENTATIONS.value,
    one_hot=True,
    omit_edo=True,
  )
  test_data = kd.KamonDataset(
    division="test",
    image_size=IMAGE_SIZE.value,
    num_augmentations=1,
    one_hot=True,
    omit_edo=True,
  )
```

`IMAGE_SIZE` and `NUM_TRAIN_AUGMENTATIONS` are assumed to be values set by flags.DEFINE_integer provided by the py-abseil flags library (they should default to 224 and 9, respectively).

By default the input images are RGB format of dimension 224 x 224, but the KamonDataset __getitem__ function converts them to tensors, along with the one-hot output labels. See kamon_dataset.py's definition of KamonDataset and its __getitem__ function for details.

Needed values for vocabulary size, the actual vocabulary mapping from integer labels to the token strings, the end token value and the sequence length (the length of the longest description + 1 for an <EOS> tag), can be extracted as follows:

```
  v = train_data.vocab_size          # Size of token list
  vocab = train_data.label_to_expr   # Map: int label -> str token
  end_token = train_data.end_token   # int value of <EOS>
  seqlen = train_data.max_len        # seq len = max_descr_len + 1
```

The trainer should of course be runnable on a GPU if the "cuda:0" device is available.

Checkpoints should be saved every N steps, where N defaults to 10_000. Full evaluation on the test set should also be run at that point, and the decoded descriptions output to a jsonlines file for the test examples. Along with the decoded descriptions, the masks for each of the seqlen positions for each decoded data point should be output as 224x224 images in a directory

corresponding to the test example, e.g. test_000/img_000.png, test_000/img_001.png, and so forth.