CPEN455 Final Project: PixelCNN++G

Guan Zheng Huang
CPEN 455
UBC

Abstract

PixelCNN++G is a conditional generative model based on the PixelCNN++ architecture[2]. For this project, we aim to implement the PixelCNN++G model with an additional classification layer to transform it into an image classification model. The model will be trained on the CPEN450 dataset to classify images into one of four classes.

1 Model

1.1 PixelCNN++G Improvements

1.1.1 Data Preprocessing:

- PixelCNN is orientation-sensitive, which means it has difficulty recognizing the orientation of objects. To address this, we randomly flip the images horizontally during training to make the model invariant to the direction of the image. This technique is also equivalent to introducing double the amount of training data, which can help improve the model's generalization ability.
- Additionally, we rotate the images randomly form -10 to +10 degrees so that the model can learn object orientations with reduced impact from the image directionality.
- We introduced various new data augmentation techniques during the fine-tuning process, in hope to generate more data unfamiliar to the model, which is often desired for th fine-tuning process. This includes colour jittering and random cropping. Based on measure, fine-process with this technique increased accuracy by 2.00% on validation dataset, 1.54% on test dataset.

1.1.2 Conditional Model:

- The model is conditioned on the class label of the image. The input labels are represented as one-hot encoding.
- Among the `gated_resnet` function, we introduced two new weights layer `weight_a` and `weight_b`, each of which is multiped with input label and is added to the resulting `a`, `b` parameters after the convolution operations. This simple and effective method allows the model to learn class-specific features at each layers. A similar but per-pixel weighted approach is disccused in 2.5

1.1.3 Classification Layer:

- We designed a classification layer utilizing a modified loss function. The idea is that by iterating over an image with all possible labels, the label that results in the lowest loss is chosen as the predicted label by the model. This technique is conceptually similar to using `min(softmax(logits))`, however, due to float precision, the latter method occasionally result in mismatch and is not used.

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

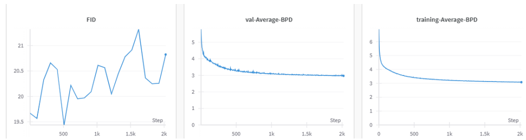


Figure 1: Training curve for the main model.

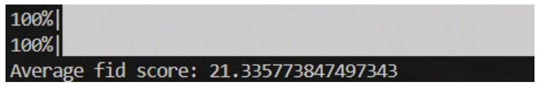


Figure 2: FID score of generated image (100)

2 Experiments

The model is trained on the CPEN450 dataset, 32x32 pixel images divided into four classes. With a batch size of 16 with 500 epochs, the additional conditional parameters are trained with xavier uniform initialization. The fid curve can be found on Figure **??** and the BPD curve can be found on Figure **??**. Sample images are shown in Figure 3.

2.1 major hyperparameter

```
--batch_size 16 ‘
--nr_resnet 2 ‘
--nr_filters 160 ‘
--nr_logistic_mix 100 ‘
--lr_decay 0.99995 ‘
--max_epochs 500 ‘
--seed 4399
```

2.2 Training and setup

Please reference the appended README_TECH.md file.

2.3 GPT-Use

Sections of the code modified by GPT is labeled in line with comment. One can find the chat history with GPT in the supplementary materials section

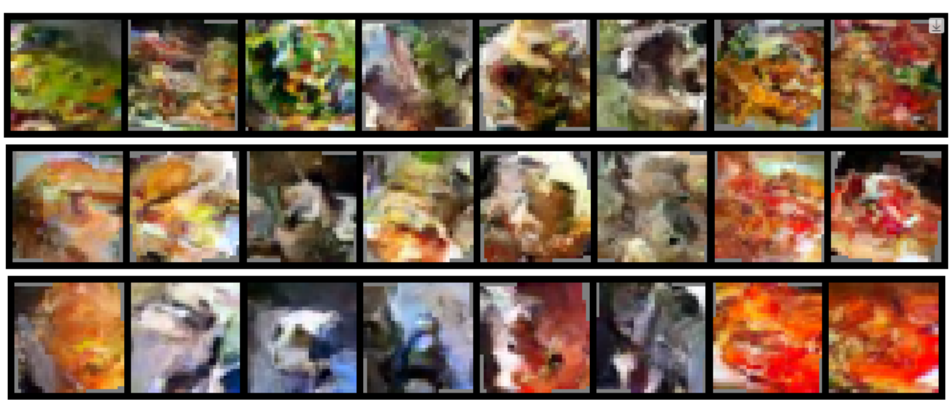


Figure 3: Sample images generated by the model. Counting from left column to right column, the first column is the class0, the second column is the class1, the third column is the class2, and the fourth column is the class3.

2.4 Results

With the model trained, we achieved an accuracy of 44% on the test set. The model is able to classify images into the correct class with reasonable accuracy.

2.5 Alternative solutions attempted

Several other solutions were attempted but did not show as promising results are discussed below. Unless explicitly stated, all structures are trained with hyperparameter [resnet = 1, filter = 40, logistical_mix = 10] at 200 epochs without finetuning. Measured accuracy is taken among the highest of all checkpoints. For reference, the reduced parameter model demonstrated an accuracy of 65.1% on the validation dataset at epoch 200 with 37.6 FID, and 71.0 % at epoch 350 (optimal amongst 500 epoch). The FID score is 28.7 at epoch 350.

- Specialized preprocessing techniques, such as image segmentation and swapping, noise and data masking did not prove to aid the training nor finetuning of the model. Such observation may largely be due to the limited training data and pixelCNN's nature of per pixel generation. Validation accuracy reduced from 65.1% to 63.3%.
- To embed the label information in the model, attempts were made to embed label information directly into the input or output of the model, which failed to show significant improvement in classification accuracy. The same attempt was made in combination with the proposed weight application technique in resnet blocks, which also failed to show improvement in classification accuracy. Validation accuracy of 53.2% is shown with 500 epochs.
- Attempts to utilizing a Polyak Averaged Model to be embedded and concatenated with the input at the ResNet block level have also been studied. While this approach showed promising results in terms of the Fréchet Inception Distance (FID), it did not significantly improve classification accuracy. Validation accuracy of 67.2% is shown with 450 epochs.
- A major model restructure to incorporate a label channel also showed limited success. This approach failed to improve classification accuracy beyond random guessing. Inspecting the generated images, it is observed that the model is unable to generate images that resemble the input images. This may suggest a structural or programming failure which failed to properly handle the label layer. Example images generated by this model are shown in Figure **??**. The best validation accuracy is 25.2% among 400 epochs, with 28.7 FID for that checkpoint.
- Additional attempt to modify the PCNN++G structure by updating the resnet structure by removing final addition with input matrix, instead purely depending on convolution layers to transform the input have been made. Although this method produced reasonable FID score of 25.5 at epoch 200 with, it marley achieved a classification accuracy of 49.2%.
- In this solution, we modified PixelCNNLayer_updown and Resnet to introduces a new set of weights at every layer of both the upward and downward paths (6 layers in total, 8 at each resnet, each coming into effect based on the label). These weights are multiplied by the class label embedding before being applied in the convolution layers. This operation is by pixel, meaning that theoretically, as unique weight can be learned about each pixel at each layer. It will establish a stronger correlation between specific components of the image and its class at every layer to address the difficulty of model capturing label information. This strategy, in combination with the proposed tanh normalization technique introduced in [1] also showed promising results of 68.4% validation accuracy at epoch 225. However, this approach showed signs of ovefitting, dispite the fact that there is increasing in parameter count. The BPD curve is shown in Figure **??**.
- Reflecting on top of the above approach, we removed the dependency of the model on specific labels, instead introducing a single wight matrix that is multiplied with the labels, normalized and added to the output. Indeed, we no longer observe the overfitting issue, and the model achieved a validation accuracy of 71.8% at epoch 500. The FID score is sub 20 at times. The only reason this model is not chosen as the final model is due to the fact that the model is not able to generalize well to the test dataset at high parameter count. The best performing checkpoint for [resnet = 1, filter = 128, logistical_mix = 100] model achieved

Table 1: Model Comparison

Part		
Name	Description	Size (μm)
Dendrite	Input terminal	$\sim$100
Axon	Output terminal	$\sim$10
Soma	Cell body	up to 10^6

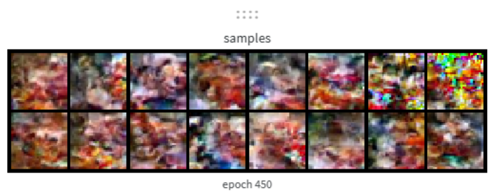


Figure 4: Sample images generated by the model with label as channels.

a test accuracy of XXX. The comparison between this model and the proposed model is shown in Figure **??**

- By observation of the failed images as seen in analysis like

3 Conclusion

In conclusion, the modifications from the author, the PixelCNN++G model was able to achieve a classification accuracy of 36.8% on the CPEN450 dataset, with a FID score of 32.73. The model and corresponding code is included as appendix to this report.

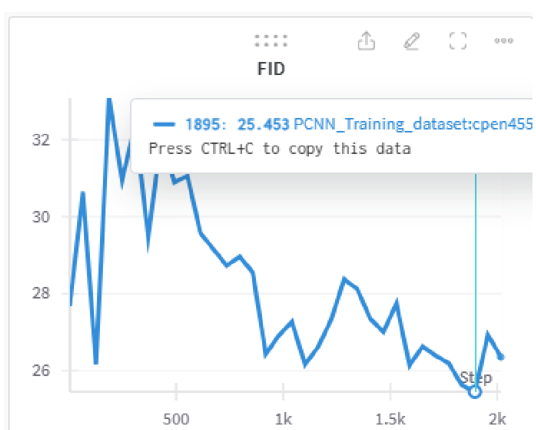


Figure 5: FID curve for the model with different down-sampling weights for each class.

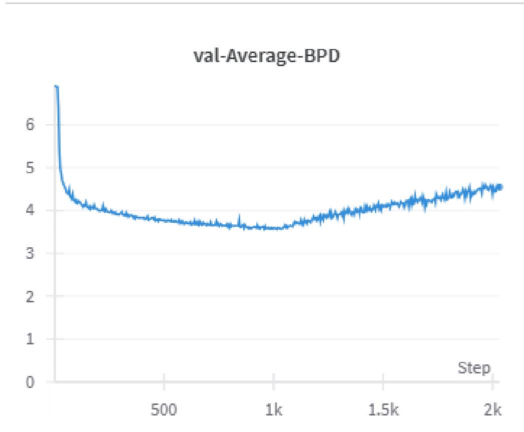


Figure 6: BPD curve for the model with new set of weights at every layer.

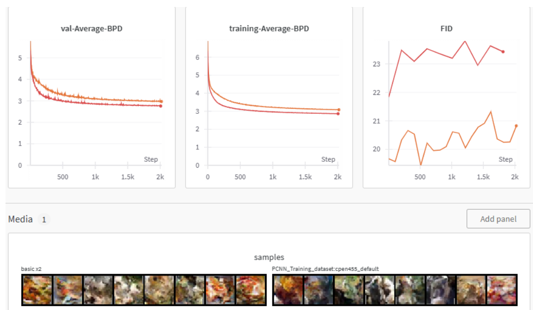
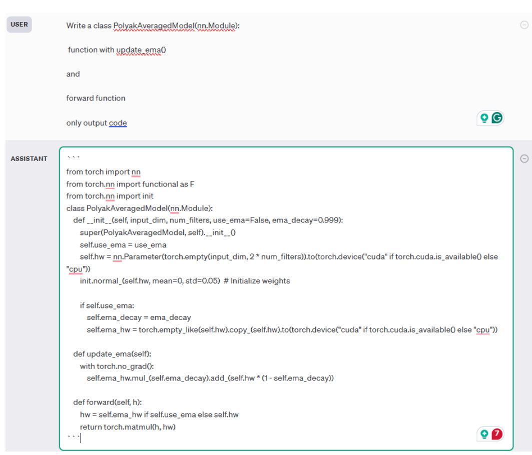


Figure 7: Comparison between the final model and the model with different down-sampling weights for each class.

4 Supplementary Material


USER

Write a class PolyakAveragedModel(nn.Module):

function with update_ema()

and

forward function

only output code

ASSISTANT

```
from torch import nn
from torch.nn import functional as F
from torch.nn import init
class PolyakAveragedModel(nn.Module):
    def __init__(self, input_dim, num_filters, use_ema=False, ema_decay=0.999):
        super(PolyakAveragedModel, self).__init__()
        self.use_ema = use_ema
        self.hw = nn.Parameter(torch.empty(input_dim, 2 * num_filters)).to(torch.device("cuda" if torch.cuda.is_available() else "cpu"))
        init.normal_(self.hw, mean=0, std=0.05)  # Initialize weights

        if self.use_ema:
            self.ema_decay = ema_decay
            self.ema_hw = torch.empty_like(self.hw).copy_(self.hw).to(torch.device("cuda" if torch.cuda.is_available() else "cpu"))

    def update_ema(self):
        with torch.no_grad():
            self.ema_hw.mul_(self.ema_decay).add_(self.hw * (1 - self.ema_decay))

    def forward(self, h):
        hw = self.ema_hw if self.use_ema else self.hw
        return torch.matmul(h, hw)
```

USER

For the code below, can you fix my error with model dimension?

```
class gated_resnet_plus(nn.Module):
  def __init__(self, num_filters, conv_op, nonlinearity=F.elu, skip_connection=0, input_dim=4):
    super(gated_resnet_plus, self).__init__()
    self.skip_connection = skip_connection
    self.nonlinearity = nonlinearity
    self.conv_input = conv_op(2 * num_filters, num_filters)
    self.conv_out = conv_op(2 * num_filters, 2 * num_filters)
    self.dropout = nn.Dropout2d(0.)
    self.num_filters = num_filters

    sizes = {"u8":8, "d8":8, "u16":16, "d16":16, "u32":32, "d32":32}
    self.V_w_dict = {}
    self.V_b_dict = {}
    self.bias_w_dict = {}
    self.bias_b_dict = {}
    for size in sizes.keys():
      self.V_w_dict[size] = nn.Parameter(torch.empty(4, sizes[size])).to(torch.device("cuda" if torch.cuda.is_available() else
"cpu"))
      self.V_b_dict[size] = nn.Parameter(torch.empty(4, sizes[size])).to(torch.device("cuda" if torch.cuda.is_available() else
"cpu"))

      # Initialize parameters
      # nn.init.xavier_uniform_(self.V_w_dict[size])
      # nn.init.xavier_uniform_(self.V_b_dict[size])

      nn.init.kaiming_uniform_(self.V_w_dict[size], nonlinearity='relu')
      nn.init.kaiming_uniform_(self.V_b_dict[size], nonlinearity='relu')



    if skip_connection != 0:
      self.nin_skip = conv_op(2 * skip_connection * num_filters, num_filters)  # assuming conv_op is adaptable
```

```
    if skip_connection != 0:
      self.nin_skip = conv_op(2 * skip_connection * num_filters, num_filters)  # assuming conv_op is adaptable

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.PKA = PolyakAveragedModel(input_dim, num_filters, use_ema=True).to(device)

    self.mode = "B"

  def forward(self, og_x, label, a=None, mode=None):

    # result tensor
    result_t = None

    if self.mode == "A":
      x = self.conv_input(self.nonlinearity(og_x))
      if a is not None:
        x += self.nin_skip(self.nonlinearity(a))
      x = self.nonlinearity(x)
      x = self.dropout(x)
      x = self.conv_out(x)
      h_weighted = self.PKA(label)  # Output size should be [batch_size, 2 * num_filters]

      # Ensure h_weighted is properly reshaped for broadcasting
      # Shape [batch_size, 2 * num_filters, 1, 1]
      h_weighted = h_weighted.view(x.size(0), 2 * self.num_filters, 1, 1)

      # Broadcasting here is implicit when adding tensors of shape [batch_size, 2 * num_filters, 1, 1] and [batch_size, 2 *
num_filters, height, width]
      x += h_weighted  # Broadcasting works by aligning trailing dimensions
      if self.training:
        self.PKA.update_ema()

      a, b = x.chunk(2, dim=1)
      c3 = a * F.sigmoid(b)
      result_t = c3 + og_x
```

ASSISTANT

There are a few considerations that could be leading to dimension-related errors in the provided code snippet for the "gated_resnet_plus" class. Let's address them with detailed explanations and proposed fixes:

Potential Dimension Issues and Their Fixes:

1. **Initializing Parameters with Incorrect Dimensions:**
Make sure that the parameter dimensions match the expected input/output shapes for each operation. From your code snippet, it seems you are initializing `V_w_dict` and `V_b_dict` without considering specific channels or spatial dimensions. This might not directly cause a dimension error depending on how they are used, but if they are intended as weights and biases for convolution operations or similar, they need to be defined accurately corresponding to convolution or fully connected layer expectations.

2. **Handling of Skip Connections:**
Assuming `conv_op` is some form of convolution operation, ensure that the input to `nin_skip` matches the `2 * skip_connection * num_filters` which could cause dimension mismatch if `a` has incorrect dimensions. It is critical to verify that `a` has a shape that can feed into `nin_skip` layer properly.

3. **Heaviside Projection from PolyakAveragedModel:**
You mentioned that `self.PKA(label)` should have an

USER

Grammar fix:
PixelCNN++G is a conditional generative model based on the PixelCNN architecture. For this project, we aim to implement the PixelCNN++G model with an additional classification layer, making it an image classification model. The model will be trained on the CPEN450 dataset to classify images into one of four classes.

ASSISTANT

PixelCNN++G is a conditional generative model based on the PixelCNN architecture. For this project, we aim to implement the PixelCNN++G model with an additional classification layer to transform it into an image classification model. The model will be trained on the CPEN450 dataset to classify images into one of four classes.

USER

Can you explain how to use nn.Embedding?

ASSISTANT

`nn.Embedding` is a class found in PyTorch, a popular machine learning library in Python. It is used to create word embeddings, representing words as multi-dimensional vectors of real numbers. This vector representation allows words with similar meanings to have a similar representation.

Here's a basic example of how to use `nn.Embedding`:

```python
import torch
import torch.nn as nn

# Suppose we have 10 words in our vocabulary and each word is represented by 5-dimensional vector.
embedding = nn.Embedding(10, 5)

# Suppose we have a batch of 2 sentences of length 4
input = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2, 9]])
embedded = embedding(input)

print(embedded)
```

In this code:
- We first import the necessary libraries - torch and torch.nn (which we alias as nn).
- We then initialize an Embedding layer with 10 words and 5 dimensions.
- We create a batch of 2 sentences of length 4 as the input. Each element in the tensor signifies the index of a specific word in our vocabulary.
- Finally, we apply embedding to our input tensor to get the vector representation

USER

I do not understand, so the embedding function takes some input and produce random output? it seemevery time I run the function result is different.

ASSISTANT

I understand the concern. Let's clarify this.

When you initialize `nn.Embedding` in PyTorch, it initially assigns random weights to all the words in that space. These random weights represent the word vectors you get when you pass in an index.

So, if you create an embedding layer, and directly pass indices of your words without training, you will get some random vectors. This is why every time you run the script, you get a different output for the same input.

The idea is to use this embedding layer in a larger model and learn the weights (embedding vectors) for your words based on your task (such as text classification, language modeling etc.) during the training process.

After training, when you pass the index of a word to the embedding layer, it will return the learned vector for that word which is no longer random but learned from the data.

USER

Suppose I have an image of 10*10, I have a class size of 4, I need to attach class information into each pixel (pixelCNN), how should I do this?

ASSISTANT

To attach class information to each pixel, you will need to first create an embedding for the class labels, which you can do with `torch.nn.Embedding`:

```python
num_classes = 4
embedding_dim = 10 # or any other number you want
embedding = torch.nn.Embedding(num_classes, embedding_dim)
```

Once you've created the class label embeddings, for each image you can expand the class label into a 10×10 image where each pixel is a copy of the class label embedding:

```python
class_label = torch.tensor([2])  # example class label for one image; use long tensor
class_embedding = embedding(class_label)  # get the embedding for this class label
expanded_class_embedding = class_embedding.expand(10, 10, -1)
```

Now, if the original image is a 10×10 grayscale image, you can concatenate the class information on the channel dimension.


```python
class_label = torch.tensor([2])  # example class label for one image; use long tensor
class_embedding = embedding(class_label)  # get the embedding for this class label
expanded_class_embedding = class_embedding.expand(10, 10, -1)
```

Now, if the original image is a 10×10 grayscale image, you can concatenate the class information on the channel dimension.


```python
image = torch.randn(1, 10, 10)  # example image
image_with_class = torch.cat([image, expanded_class_embedding.unsqueeze(0)], dim=1)
```

This `image_with_class` is now a 10×10 image with channels for the original image as well

USER

so expanded_class_embedding.unsqueeze(0) adds a new dimention? Why would we do this?

ASSISTANT

Indeed, `unsqueeze(0)` will add a new dimension in the 0th axis. This is necessary to join the `image` tensor and the `expanded_class_embedding` tensor.

The reason behind adding a new axis is due to the way PyTorch (and many other libraries) handle images and batches of images. Specifically, the convention is to represent each image as a 3D tensor, with dimensions `[channel, height, width]`, and a batch of images as a 4D tensor, with dimensions `[batch_size, channel, height, width]`.

In your case, the `image` tensor is a 3D tensor because it was probably loaded as a batch of one image, with dimensions `[1, 10, 10]` (or `[channel, height, width]`).

On the other hand, the `expanded_class_embedding` tensor is a 2D tensor with dimensions: `[10, 10]`, because it was expanded to match the height and width of the image.

In order to concatenate them along the channel dimension: `[1, 10, 10]` and `[10, 10]` need to match in the number of dimensions. Because of that, you add a new dimension to

USER

Can you explain how to use nn.Embedding?

ASSISTANT

`nn.Embedding` is a class found in PyTorch, a popular machine learning library in Python. It is used to create word embeddings, representing words as multi-dimensional vectors of real numbers. This vector representation allows words with similar meanings to have a similar representation.

Here's a basic example of how to use `nn.Embedding`:

```python
import torch
import torch.nn as nn

# Suppose we have 10 words in our vocabulary and each word is represented by 5-dimensional vector.
embedding = nn.Embedding(10, 5)

# Suppose we have a batch of 2 sentences of length 4
input = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2, 9]])
embedded = embedding(input)

print(embedded)
```

In this code:
- We first import the necessary libraries - torch and torch.nn (which we alias as nn).
- We then initialize an Embedding layer with 10 words and 5 dimensions.
- We create a batch of 2 sentences of length 4 as the input. Each element in the tensor signifies the index of a specific word in our vocabulary.
- Finally, we apply embedding to our input tensor to get the vector representation

USER

I do not understand, so the embedding function takes some input and produce random output? it seemevery time I run the function result is different.

ASSISTANT

I understand the concern. Let's clarify this.

USER

Why is BCELogitsLoss used?

ASSISTANT

`BCEWithLogitsLoss` is a commonly used loss function for binary classification problems, especially when the outputs can be considered as probabilities indicating the presence or absence of a class. The reasons for its usage include the combination of technical and practical benefits, which I'll outline below:

Technical Advantages

1. **Numerical Stability**: `BCEWithLogitsLoss` combines the `Sigmoid` function and the `Binary Cross Entropy (BCE)` loss in a single class. This

USER

I mean instead of BCELoss.

ASSISTANT

Choosing `BCEWithLogitsLoss` over `BCELoss` offers benefits primarily centered around numerical stability and the convenience of combining operations. Here's a deeper dive into why `BCEWithLogitsLoss` is often preferred over `BCELoss`:

Numerical Stability

- **Log-Sigmoid + BCE**: `BCEWithLogitsLoss` applies both the `Sigmoid` activation and the binary cross-entropy loss in one go. Critically, it deals with logits (raw model outputs before the Sigmoid activation) and

- **Log-Sigmoid + BCE**: `BCEWithLogitsLoss` applies both the `Sigmoid` activation and the binary cross-entropy loss in one go. Critically, it deals with logits (raw model outputs before the Sigmoid activation) and internally utilizes the log-sum-exp trick to compute the loss in a numerically stable way. This helps mitigate issues related to floating-point arithmetic, such as underflow or overflow, which can occur when separately applying a Sigmoid function followed by the BCE loss, especially with extreme values.

Efficiency

- **Single-step Process**: By combining the Sigmoid operation and the BCE computation, `BCEWithLogitsLoss` reduces the need for two separate computational steps into one. This can be slightly more efficient, reducing the computational graph's complexity and potentially speeding up the backpropagation process since it involves fewer operations.

Implementation Differences

- **Sigmoid Activation Separation**: When using `BCELoss`, you need to manually apply a Sigmoid function to your model's output before passing it to the loss function. This additional step can be easy to forget and separates the conceptual link between the raw model outputs (logits) and the loss calculation.

```python
# Using BCELoss
```

```python
# Using BCELoss
criterion = torch.nn.BCELoss()
output = model(input)
loss = criterion(torch.sigmoid(output), target)
```

Compared to the direct approach of `BCEWithLogitsLoss`:

```python
# Using BCEWithLogitsLoss
criterion = torch.nn.BCEWithLogitsLoss()
output = model(input)
loss = criterion(output, target)
```

Conclusion

`BCEWithLogitsLoss` is generally preferred due to its numerical stability and computational efficiency, making it a robust default choice for binary classification problems. It simplifies the implementation by handling the Sigmoid activation within the loss calculation, avoiding potential mistakes in loss calculation and making the code cleaner and more straightforward.

Add message

USER

Can you write again, this time have model accept xy_set, which is a list of tuple (size 2 tuple of xy), which the model gets all the answers from, accumulates the logits across all xy data provided, then predicts the result?

To implement a function that can evaluate multiple sub-regions of an image specified by a list of `xy` coordinates and then aggregate results across these regions to make a prediction, we can make the following adjustment. This function, `get_label_multi_region_smart`, will accept a list of `xy` coordinate tuples. Each tuple defines a subregion up till `(x, y)` for which we want to consider the image and make predictions.

```python
def get_label_multi_region_smart(model, model_input, xy_set, device):
  batch_size = model_input.size(0)
  all_predictions = torch.zeros(NUM_CLASSES, batch_size,
dtype=torch.float32, device=device)

  # Iterate over each region specified by the xy tuple
  for x, y in xy_set:
    # Masking the image to consider only pixels up to (x, y)
    mask = torch.zeros_like(model_input)
    mask[:, :, :x, :y+1] = 1  # Include all rows up to x, and all columns up to y

    # Apply mask; portions of image beyond (x, y) are zeroed out
    masked_input = model_input * mask

    region_predictions = torch.zeros(NUM_CLASSES, batch_size,
dtype=torch.float32, device=device)
    for i in range(NUM_CLASSES):
      # Convert label to tensor representation
```

ASSISTANT

```python
      # To make sure the mask applies properly, we use the same mask on
the output
      masked_output = raw_output * mask

      # Calculate logistic loss for masked region
      region_predictions[i] = discretized_mix_logistic_loss(masked_input,
masked_output, train=False)

    # Accumulate predictions across all specified regions by averaging logits
    all_predictions += region_predictions / len(xy_set)

  # Compute softmax probabilities to find classes and then find the minimum
predicted class label
  _, pred = torch.min(all_predictions, dim=0)

  return pred
```

Key Modifications:
1. **Mask application for each region**: For every `(x, y)` in `xy_set`, a mask is created and applied to the input image. This restricts the model to see and evaluate only up to the specified limit in the image for that iteration.
2. **Region-specific Predictions**: Within the loop of `xy_set`, we calculate predictions specific to each region `(x, y)` and accumulate these to the overall predictions via averaging. This way, the model integrates information from parts of the image specified in `xy_set`.
3. **Averaging Predictions**: Predictions from each region are averaged (`all_predictions` gets aggregated over the loop by taking the mean of `region_predictions`). In case of handling logits, additional operations like log-sum-exponential could be necessary depending on how logits are intended to be merged.

```
        # Forward pass through the model to get raw outputs
        raw_output = model(model_input, class_label=class_label)

        # Convert raw logistics into probabilities or logits
        # mean pooling over logistic parameters
        all_predictions[i] = discretized_mix_logistic_loss(model_input,
raw_output, train=False)

    # Compute probabilities using softmax
    _, pred = torch.min(all_predictions, dim=0) # bettr, as we are using negative
log likelihood
    # pred_2 = torch.argmin(torch.softmax(all_predictions, dim=0), dim=0)
    return pred
```

Can you write a new get_label_supser-smart which takes a xy value and instead of evaluating the compete image, eval the image up to x, y? ie: currently prediction is based on complete image, but I may want it on half the image instead.

In order to modify the existing `get_label_smart` function to take into consideration only a portion of the image up to a specific pixel (x, y), we need to mask or zero out the parts of the image that are beyond the specified coordinates x and y. Here is the modified function, `get_label_super_smart`, that will evaluate the model based on only the part of the image up to the coordinates (x, y):

```python
def get_label_super_smart(model, model_input, x, y, device):
    batch_size = model_input.size(0)
    all_predictions = torch.zeros(NUM_CLASSES, batch_size,
dtype=torch.float32, device=device)
```

ASSISTANT

```
    all_predictions = torch.zeros(NUM_CLASSES, batch_size,
dtype=torch.float32, device=device)

    # Masking the image to consider only pixels up to (x, y)
    mask = torch.zeros_like(model_input)
    mask[:, :, :x, :y+1] = 1  # Include all rows up to x, and all columns up to y

    # Apply mask; portions of image beyond (x, y) are zeroed out
    masked_input = model_input * mask

    for i in range(NUM_CLASSES):
        # Convert label to tensor representation
        class_label = label_to_onehot_tensor([my_bidict.inverse[i]]*batch_size)

        # Forward pass through the model to get raw outputs
        raw_output = model(masked_input, class_label=class_label)

        # Evaluate loss only on the visible part of the image
        # To make sure the mask applies properly, we apply it similarly on the
output equivalent to the input
        masked_output = raw_output * mask

        # Calculate loss
        all_predictions[i] = discretized_mix_logistic_loss(masked_input,
masked_output, train=False)

    # Compute probabilities using softmax and then find the minimum
predicted label
    _, pred = torch.min(all_predictions, dim=0)  # using negative log likelihood,
lower is better

    return pred
```

USER

Can you update my code so dataloader is updated?

```
csv_path = os.path.join(ROOT_DIR, "train" + '.csv')
      df = pd.read_csv(csv_path, header=None,
names=['path', 'label'])
      csv_path2 = os.path.join(ROOT_DIR,
"validation" + '.csv')
      df2 = pd.read_csv(csv_path2, header=None,
names=['path', 'label'])
      # Convert DataFrame to a list of tuples
      # should take 30% from training only, 70%
from validation only
      # 90% should be of desired classes
      self.samples = list(df.itertuples(index=False,
name=None))
      self.samples = [(os.path.join(ROOT_DIR, path),
label) for path, label in self.samples]

      self.samples2 = list(df2.itertuples(index=False,
name=None))
      self.samples2 = [(os.path.join(ROOT_DIR,
path), label) for path, label in self.samples2]
```

Sure, try this:

```
```python
import random

# Divide load in the ratio of 30% from training and
70% from validation
train_frac, val_frac = 0.3, 0.7

csv_path = os.path.join(ROOT_DIR, "train" + '.csv')
df = pd.read_csv(csv_path, header=None, names=
```

ASSISTANT

```
df = pd.read_csv(csv_path, header=None, names=
['path', 'label'])

csv_path2 = os.path.join(ROOT_DIR, "validation" +
'.csv')
df2 = pd.read_csv(csv_path2, header=None,
names=['path', 'label'])

desired_classes = [] #Fill this with the two classes
which your model confuses
rest_classes = []    #Fill this with rest of the classes

# Filter desired classes and rest classes from df
subset_df = df[df['label'].isin(desired_classes)]
rest_df = df[df['label'].isin(rest_classes)]

desired_val_df =
df2[df2['label'].isin(desired_classes)]
rest_val_df = df2[df2['label'].isin(rest_classes)]

# Get 90% of total from desired classes and 10%
from rest
desired_num = round(train_frac * 0.9 *
(min(len(subset_df), len(desired_val_df))))
rest_num = round(train_frac * 0.1 *
(min(len(rest_df), len(rest_val_df))))

samples_train =
random.sample(list(subset_df.itertuples(index=Fal
se, name=None)), desired_num)
samples_train.extend(random.sample(list(rest_df.it
ertuples(index=False, name=None)), rest_num))

# Repeat the process for validation data
desired_num_val = round(val_frac * 0.9 *
```
```

```
samples_train =
random.sample(list(subset_df.itertuples(index=Fal
se, name=None)), desired_num)
samples_train.extend(random.sample(list(rest_df.it
ertuples(index=False, name=None)), rest_num))

# Repeat the process for validation data
desired_num_val = round(val_frac * 0.9 *
(min(len(subset_df), len(desired_val_df))))
rest_num_val = round(val_frac * 0.1 *
(min(len(rest_df), len(rest_val_df))))

samples_val =
random.sample(list(desired_val_df.itertuples(index
=False, name=None)), desired_num_val)
samples_val.extend(random.sample(list(rest_val_d
f.itertuples(index=False, name=None)),
rest_num_val))

self.samples = [(os.path.join(ROOT_DIR, path),
label) for path, label in samples_train]
self.samples2 = [(os.path.join(ROOT_DIR, path),
label) for path, label in samples_val]
```

This code splits your dataset according to classes. The dataset will contain 90% of images from the two desired classes and 10% from the other classes. Make sure to replace the "desired_classes" and "rest_classes" with the actual class names. This will result in two lists (`self.samples` and `self.samples2`) for training and validation data loaders respectively.

References

[1] T. van den Salimans, A. Karpathy, X. Chen, and D. p. Kingma, PIXELCNN++: IMPROVING THE PIXELCNN WITH DISCRETIZED LOGISTIC MIXTURE LIKELIHOOD AND OTHER MODIFICATIONS, 2017. Accessed: 2024. [Online]. Available: https://arxiv.org/pdf/1606.05328.pdf

[2] A. van den Oord et al., Conditional Image Generation with PixelCNN Decoders, 2016. Accessed: 2024. [Online]. Available: https://arxiv.org/pdf/1606.05328.pdf