
CPEN455 Final Project: PixelCNN++G

Guan Zheng Huang
CPEN 455
UBC

Abstract

This work introduces PixelCNN++G, an image generation model derived from the PixelCNN++ architecture [1]. PixelCNN++G, a conditional generation model, can also be used for image classification. We demonstrate the capabilities of this modification by training the model on the CPEN450 dataset, categorizing images into four classes.

1 Model

1.1 PixelCNN++G Improvements

1.1.1 Data Preprocessing:

- PixelCNN++ is sensitive to image orientation, which can hinder its ability to accurately recognize object orientations. To mitigate this, we horizontally flip the images randomly during training, which not only helps the model become invariant to direction but also effectively doubles the training dataset, enhancing the model's generalization capabilities.
- We also rotate the images randomly within a range of -10 to +10 degrees. This helps the model to better learn object orientations with minimal influence from image directionality.
- Further, we employ new data augmentation techniques during the fine-tuning stage, such as color jittering and random cropping, to expose the model to more varied data, which is crucial in the fine-tuning process.

1.1.2 Conditional Model:

- The model is conditioned on image class labels, which is represented as one-hot encoding.
- Within the `gated_resnet` function, we introduced two additional layers (`weight_a` and `weight_b`), each multiplicatively interacting with the input label before adding to the resultant parameters (`a`, `b`) post-convolution. This approach enables the model to learn class-specific features at each layer effectively. To illustrate this, a simplified diagram of PixelCNN++G is presented in Figure 1. It is important to note that this diagram omits detailed features such as shifting and masking; it focuses on demonstrating the influence of the weighting mechanism.

1.1.3 Classification Layer:

- We implemented a classification layer using a modified per-image loss function. By evaluating each image across all potential labels and choosing the label that yields the lowest loss, the model predicts the image's label. This approach, conceptually similar to `argmin(softmax(logits))`, avoids the pitfalls of floating-point precision errors found in the softmax approach, which occasionally lead to incorrect predictions.

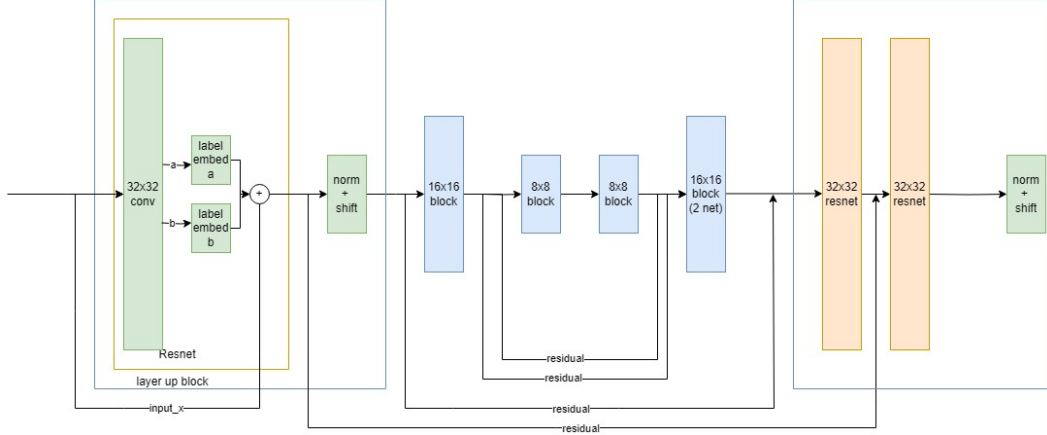


Figure 1: Training curve for the main model.

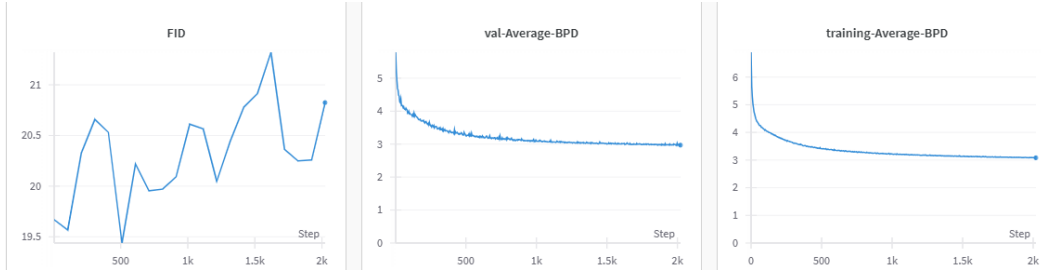


Figure 2: Training curve for the main model.

1.1.4 Finetuning Pipeline:

- A new fine-tuning pipeline has been developed to enhance the model's performance on specific classes or across all classes with a reduced learning rate. This pipeline can incorporate a validation set for additional data analysis; however, subsequent validation on this portion might skew results. This process was not used to produce the final PixelCNN++G checkpoint as the base model performed best, demonstrating efficacy in addressing class imbalance issues observed in other models. A detailed review of this process and the result comparison is available in the appendix under section 2.2.

2 Experiments

The model is trained on the CPEN450 dataset, 32x32 pixel images divided into four classes. The training curve is shown in Figure 2. Sample images are shown in Figure 3.

2.1 major hyperparameter

```
--batch_size 32 '
--nr_resnet 1 '
--nr_filters 128 '
--nr_logistic_mix 100 '
--lr_decay 0.99995 ' # until batch 501, changed to 0.997 afterward
--lr 0.0002 ' # until batch 501, changed to 0.0001 afterward
--max_epochs 801 ' # final batch used is 775, however, validation accuracy reached 86+ since
--seed 4399
```

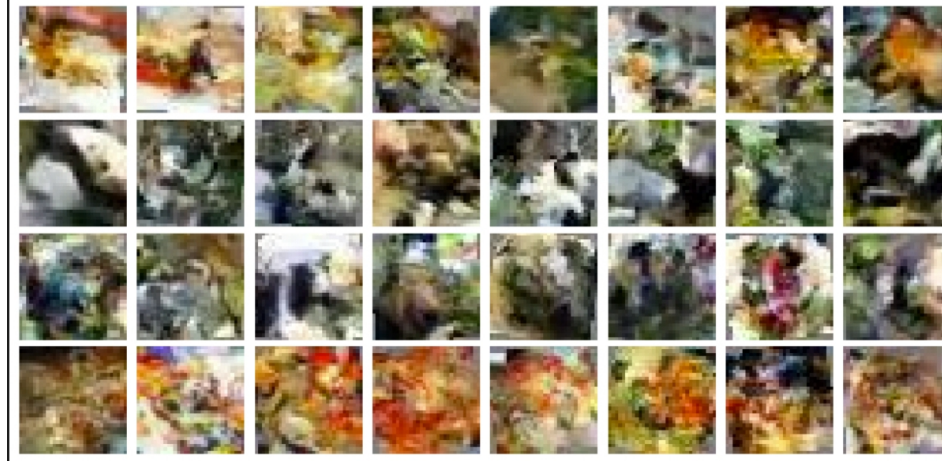


Figure 3: Sample images generated by the model, each row represents a class.

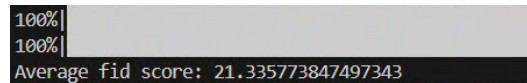


Figure 4: FID score of generated image (100)

53 2.2 Training and setup

54 Please reference the appended CPEN455HW-2023W2\README_TECH.md file.

55 2.3 Results

56 With the model trained, we achieved an accuracy of 0.89% and f1 of 0.89% on the test set with a fid
57 score of 21.3, as shown in Figure 4.

58 2.4 Alternative solutions attempted

59 We explored several interesting alternative solutions, detailed below, which did not yield as promis-
60 ing results as our chosen approach. Unless stated otherwise, all models were trained with the hyper-
61 parameters [resnet = 1, filter = 40, logistical_mix = 10] for 200 epochs without fine-tuning. Measured
62 accuracies were taken from the highest performing checkpoints. For context, our reduced parameter
63 model achieved an accuracy of 74.1% on the validation dataset at epoch 200 with a Fréchet Inception
64 Distance (FID) of 37.6, increasing to 81.2% accuracy and a FID of 28.7 at epoch 350 (the optimal
65 point within a 500 epoch span).

- 66 • Specialized preprocessing techniques, such as image segmentation, swapping, noise, and
67 data masking, did not enhance the training or fine-tuning performance of the model. These
68 methods may not have been effective due to the limited training data and PixelCNN's pixel-
69 wise generation characteristic. Validation accuracy decreased from 74.1% to 73.3
- 70 • Employing a Polyak Averaged Model, integrated at the ResNet block level, showed po-
71 tential in improving FID scores but failed to significantly boost classification accuracy.
72 Validation accuracy peaked at 77.2% after 450 epochs.
- 73 • A complete structural overhaul to include a label channel was unsuccessful in elevating
74 classification accuracy beyond mere random chance. Generated images from this model
75 failed to resemble input images, indicating possible structural or implementation errors.
76 Optimal validation accuracy was a mere 25.2% over 400 epochs, with 28.7 FID at the same
77 checkpoint.
- 78 • Modifying the PCNN++G by eliminating the final addition with the input matrix in favor
79 of purely relying on convolution layers resulted in a reasonable FID of 25.5 at epoch 200,

Table 1: Model Performance Comparison

Model Variant	Epoch	Validation Accuracy (%)	FID
PIXELCNN++G	350	74.1	28.7
PIXELCNN++G	350	81.2	28.7
Polyak Averaged Model	200	77.2	-
Label Channel Model	400	25.2	-
Pure Convolution	200	69.2	25.5
Modified PixelCNNLayer	300	84.9	-
Single Weight Matrix	500	71.8	20

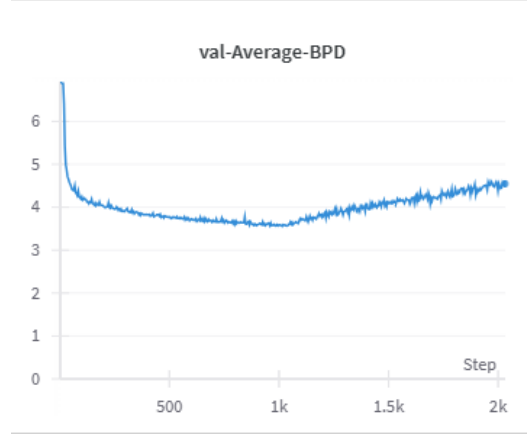


Figure 5: BPD curve for the model with new set of weights at every layer.

but a disappointing classification accuracy of only 69.2%. This is the approach utilized by vann den Oord et al. [2] in third approach to conditional PIXELCNN model. However, the highlighted approach utilized additional conditional AVE decoder which was not implemented by this solution.

- Revising PixelCNNLayer_{up/down} and ResNet to introduce a new set of weights at each layer, which are multiplied by the class label embedding prior to convolution, demonstrated a stronger correlation between image components and their classification at every layer. This approach reached an accuracy of 84.9% at epoch 300 but exhibited signs of increasing BPD due to the handling of gradients. The BPD curve for this model variation is shown in Figure 5.
- Eliminating specific label dependencies in favor of a single weight matrix interacting with the labels resulted in no overfitting, achieving a validation accuracy of 71.8% at epoch 500, with occasional sub-20 FID scores. However, this model struggled to generalize effectively to the test dataset at high parameter settings, achieving a test accuracy of 86.1% by epoch 475 (increasing to 87.6% post-fine-tuning). A comparison with our proposed model is illustrated in Figure 6.
- For further details on classification, fine-tuning, data analysis, and training processes, please refer to the technical documentation detailed in 2.2.

3 Conclusion

In conclusion, with the modifications from the author, the PixelCNN++G model was able to achieve a classification accuracy of 88.9% on the tes dataset, with a FID score of 21.3. The model and corresponding code is included as appendix to this report.

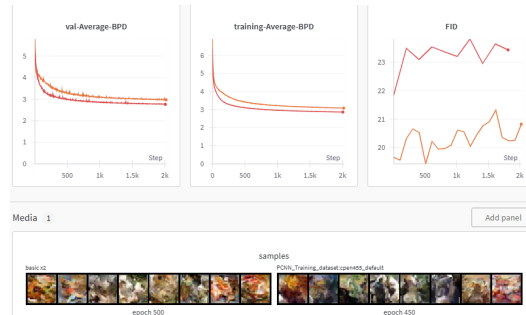


Figure 6: Comparison between the final model and the model with different down-sampling weights for each class.

102 4 Supplementary Material

103 4.1 GPT-Use

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

```

USER: Write a class PolyakAveragedModel(torch.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(torch.nn.Module):
    def __init__(self, input_dim, num_filters, use_ema=False, ema_decay=0.9999):
        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(torch.nn.Module):
    def __init__(self, num_filters, conv_op, nonlinearity=F.relu, skip_connection=0, input_dim=6):
        super(gated_resnet_plus, self).__init__()
        self.skip_connection = skip_connection
        self.nonlinearity = nonlinearity
        self.conv_in = conv_op(2 * num_filters, num_filters, num_filters)
        self.conv_out = conv_op(2 * num_filters, 2 * num_filters, 2 * num_filters)
        self.dropout = nn.Dropout2d(0.5)
        self.num_filters = num_filters

        sizes = ("x0", 16, "x1", 16, "x2", 16, "x3", 16, "x4", 32, "x5", 32, "x6", 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.res_skip = conv_op(2 * skip_connection * num_filters, num_filters, num_filters) # assuming conv_op is adaptable

```

```

if skip_connection != 0:
    self.in_skip = conv_op2 * skip_connection * num_filters, num_filters) # assuming conv_op is adaptable

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

self.mode = 'D'

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

    # result tensor
    result, t = None

    if self.mode == 'A':
        x = self.conv_input(self.nonlinearify(og, x))
        if a is not None:
            x = self.conv_skip(self.nonlinearify(a))
        x = self.nonlinearify(x)
        x = self.dropout(x)
        x = self.conv_output(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(-1, 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=0)
        c3 = a * Faigmod(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 "in_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 "in_skip" layer properly.
3. **Heavy-Duty Projection from PolyAveragedModel:**
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([[2, 4, 0], [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 seem every 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.rand(1, 10, 10) # example image
image_with_class = torch.cat([image, expanded_class_embedding.unsqueeze(0)], dim=0)
'''
```

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 dimension? Why would we do that?

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([[2, 4, 15, 14], [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 seem every 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 # 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

```

```

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[] = 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.

```

Here is a sampling function for a `pixelcnn generative` network:

```

def sample(model, sample_batch_size, obs, sample_op,
class_label=torch.tensor):
 """ sampling function
 Args:
 model: model to sample from
 sample_batch_size: number of samples to generate
 obs: tuple of dimensions for the data
 sample_op: function to sample from the model output
 """
 model.train(False)
 with torch.no_grad():
 data = torch.zeros(sample_batch_size, obs[0], obs[1], obs[2])
 data = data.to(next(model.parameters()).device)
 for i in range(obs[1]):
 for j in range(obs[2]):
 data_v = data
 out = model(data_v, sample=True, class_label=class_label)
 out_sample = sample_op(out)
 data[:, :, i, j] = out_sample.data[:, :, i, j]
 return data

```

Here is a classification based on the loss of a `image`:

```

def get_label_smart(model, model_input, device):
 batch_size = model_input.size(0)
 all_predictions = torch.zeros(NUM_CLASSES, batch_size,
dtype=torch.float32, device=device)
 for i in range(NUM_CLASSES):
 # Convert label to tensor representation
 class_label = label_to_onehot_tensor(lmvs_bidict.inverse[i]).batch_size

```

```

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

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

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

```

Can you write a new `get_label_super_smart` which takes a `xy` value and instead of evaluating the complete 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)

```

```

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(torch.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

```

ASSISTANT

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]

```

USER

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=

```

```

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=False, name=None)), desired_num)
samples_train.extend(random.sample(list(rest_df.itertuples(index=False, name=None)), rest_num))

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

```

ASSISTANT

```

samples_train =
random.sample(list(subset_df.itertuples(index=False, name=None)), desired_num)
samples_train.extend(random.sample(list(rest_df.itertuples(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_df.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.

```

## 106 References

- 107 [1] T. van den Salimans, A. Karpathy, X. Chen, and D. p. Kingma, PIXELCNN++:  
108 IMPROVING THE PIXELCNN WITH DISCRETIZED LOGISTIC MIXTURE LIKELI-  
109 HOOD AND OTHER MODIFICATIONS, 2017. Accessed: 2024. [Online]. Available:  
110 <https://arxiv.org/pdf/1606.05328.pdf>
- 111 [2] A. van den Oord et al., Conditional Image Generation with PixelCNN Decoders, 2016. Accessed:  
112 2024. [Online]. Available: <https://arxiv.org/pdf/1606.05328.pdf>