Comprehensive Wasserstein GAN (WGAN) Cheat Sheet

AI Assistant

October 21, 2024

1 Key Concepts

- **Wasserstein Distance:**
  - Also known as Earth Mover's Distance
  - Measures the distance between two probability distributions
  - Provides a meaningful and smooth gradient everywhere
- **Critic vs Discriminator:**
  - WGAN uses a critic instead of a discriminator
  - Critic outputs a real number instead of a probability
  - Critic is trained to approximate the Wasserstein distance
- **Lipschitz Continuity:**
  - Critic must satisfy the Lipschitz continuity condition
  - Ensures the Wasserstein distance is well-defined
- **Weight Clipping:**
  - Original method to enforce Lipschitz continuity
  - Clip weights to a fixed range after each gradient update
- **Gradient Penalty (WGAN-GP):**
  - Improved method to enforce Lipschitz continuity
  - Adds a penalty term to the critic's loss

2 Mathematical Formulation

2.1 Wasserstein Distance

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} [\mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_g}[f(x)]] \tag{1}$$

Where:

- P_r is the real data distribution
- P_g is the generated data distribution
- f is a 1-Lipschitz function

2.2 WGAN Objective

$$\min_G \max_C [\mathbb{E}_{x\sim P_r}[C(x)] - \mathbb{E}_{z\sim p(z)}[C(G(z))]] \tag{2}$$

Where:

- G is the generator
- C is the critic
- z is random noise

2.3 Gradient Penalty (WGAN-GP)

$$L = \mathbb{E}_{x\sim P_r}[C(x)] - \mathbb{E}_{x\sim P_g}[C(x)] + \lambda \cdot \mathbb{E}_{x\sim P_x}[(\|\nabla_x C(x)\|_2 - 1)^2] \tag{3}$$

Where:

- λ is the penalty coefficient (typically 10)
- P_x is the distribution of interpolated samples

3 WGAN Algorithm

Algorithm 1 WGAN Training

Require: α, the learning rate
Require: c, the clipping parameter
Require: m, the batch size
Require: n_{critic}, the number of critic iterations per generator iteration

for number of training iterations **do**
    for $t = 1, \ldots, n_{critic}$ **do**
        Sample $\{x^{(i)}\}_{i=1}^m \sim P_r$ a batch from the real data
        Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples
        $\tilde{x}^{(i)} \leftarrow G_\theta(z^{(i)})$ for $i = 1, \ldots, m$
        $L^{(i)} \leftarrow C_w(\tilde{x}^{(i)}) - C_w(x^{(i)})$ for $i = 1, \ldots, m$
        $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, \nabla_w \frac{1}{m}\sum_{i=1}^m L^{(i)})$
        $w \leftarrow \text{clip}(w, -c, c)$
    end for
    Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples
    $\tilde{x}^{(i)} \leftarrow G_\theta(z^{(i)})$ for $i = 1, \ldots, m$
    $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, -\nabla_\theta \frac{1}{m}\sum_{i=1}^m C_w(\tilde{x}^{(i)}))$
end for

4 PyTorch Implementation

4.1 Generator and Critic Classes

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

class Generator(nn.Module):
    def __init__(self, z_dim, img_dim):
        super(Generator, self).__init__()
```

```python
        self.gen = nn.Sequential(
            nn.Linear(z_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, img_dim),
            nn.Tanh()
        )

    def forward(self, z):
        return self.gen(z)

class Critic(nn.Module):
    def __init__(self, img_dim):
        super(Critic, self).__init__()
        self.critic = nn.Sequential(
            nn.Linear(img_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 64),
            nn.LeakyReLU(0.2),
            nn.Linear(64, 1)
        )

    def forward(self, img):
        return self.critic(img)
```

4.2 Training Loop

```python
# Hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
learning_rate = 5e-5
batch_size = 64
img_dim = 28 * 28  # for MNIST
z_dim = 100
num_epochs = 50
critic_iterations = 5
weight_clip = 0.01

# Initialize generator and critic
generator = Generator(z_dim, img_dim).to(device)
critic = Critic(img_dim).to(device)

# Optimizers
opt_gen = optim.RMSprop(generator.parameters(), lr=learning_rate)
opt_critic = optim.RMSprop(critic.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    for batch_idx, (real, _) in enumerate(loader):
        real = real.view(-1, 784).to(device)
        batch_size = real.shape[0]

        # Train Critic
        for _ in range(critic_iterations):
            noise = torch.randn(batch_size, z_dim).to(device)
            fake = generator(noise)
```

```
            critic_real = critic(real).reshape(-1)
            critic_fake = critic(fake).reshape(-1)
            loss_critic = -(torch.mean(critic_real) - torch.mean(critic_fake))
            critic.zero_grad()
            loss_critic.backward(retain_graph=True)
            opt_critic.step()

            # Clip critic weights
            for p in critic.parameters():
                p.data.clamp_(-weight_clip, weight_clip)

        # Train Generator
        output = critic(fake).reshape(-1)
        loss_gen = -torch.mean(output)
        generator.zero_grad()
        loss_gen.backward()
        opt_gen.step()

        if batch_idx % 100 == 0:
            print(
                f"Epoch [{epoch}/{num_epochs}] Batch {batch_idx}/{len(loader)} \
                  Loss D: {loss_critic:.4f}, loss G: {loss_gen:.4f}"
            )
```

5 Key Differences from Standard GAN

- Uses Wasserstein distance instead of Jensen-Shannon divergence
- Critic outputs unbounded real numbers, not probabilities
- No log in the loss function
- Weight clipping or gradient penalty to enforce Lipschitz continuity
- More stable training and meaningful loss

6 Advantages of WGAN

- Improved stability during training
- Meaningful loss metric that correlates with sample quality
- Reduced mode collapse
- Better gradient flow for the generator
- Less sensitive to architecture choices and hyperparameters

7 Practical Tips

- Use RMSProp optimizer instead of Adam for more stable training
- Train the critic to optimality before each generator update
- Monitor the Wasserstein estimate during training
- Use gradient penalty (WGAN-GP) for better performance than weight clipping

- Experiment with different architectures for generator and critic
- Start with a lower learning rate (e.g., 5e-5) and adjust as needed
- Ensure proper normalization of input data (e.g., scale to [-1, 1])
- Use BatchNorm in the generator but not in the critic
- Experiment with different z_{dim} values for the noise input