Here's a detailed **comparison between a Wasserstein GAN (WGAN)** and a **standard GAN**, particularly from the perspective of **performance in PyTorch**, including **training stability, loss behavior, and generated sample quality**.

🔁 1. **Objective Function**

Feature	**Standard GAN**	**Wasserstein GAN (WGAN)**
Loss Function	Binary Cross-Entropy (BCE)	Wasserstein Loss (Earth Mover's Distance)
Discriminator Output	Probability (0–1) via sigmoid	Real-valued (no sigmoid)
Goal	Minimize JS Divergence	Minimize Wasserstein (EM) distance

---

🧠 2. **Training Stability**

Aspect	**Standard GAN**	**WGAN**
Mode Collapse	Frequent, especially with high learning rates	Rare, due to smoother gradients
Gradient Vanishing	Common when discriminator gets too good	Solved by removing sigmoid + using continuous output space
Stability	Sensitive to hyperparameters and architecture	More stable and robust across a range of hyperparameters

---

📊 3. **Loss Behavior**

Loss Curve Interpretation	**Standard GAN**	**WGAN**
Generator Loss	Often unstable, hard to interpret	Linearly correlated with sample quality
Discriminator Loss	May saturate (0 or 1 probabilities)	Meaningful during the entire training
Monitoring Training	Difficult; loss does not correlate well	Easier to debug and monitor

---

🧪 4. **PyTorch Implementation Differences**

Generator – Both are similar.

Discriminator Differences:

- **Standard GAN** uses `nn.Sigmoid` and `BCELoss`.

- **WGAN** removes `Sigmoid`, avoids `BCELoss`, and uses **mean output** as the critic score.

```
# WGAN Discriminator example
class Critic(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1)  # No Sigmoid here!
        )

    def forward(self, x):
        return self.model(x)
```

Optimizer:

- **Standard GAN**: Adam with β1=0.5, β2=0.999.
- **WGAN**: RMSprop or Adam with smaller β1 (e.g., β1=0) recommended.

Weight Clipping:

```
# Required in original WGAN
for p in critic.parameters():
    p.data.clamp_(-0.01, 0.01)
```

(Optional) WGAN-GP: Gradient Penalty instead of clipping — improves further stability.

PROF

🎯 5. **Sample Quality & Convergence**

Factor	Standard GAN	WGAN
Sample Diversity	Lower, more prone to mode collapse	Higher, less collapse
Visual Quality	Acceptable, but can fluctuate	More consistent and better with fewer epochs
Training Time	Faster per epoch	Slightly slower due to multiple critic steps
Epochs to Converge	Varies, often needs tuning	Usually converges faster with consistent loss

---

📌 Summary Table

Feature	Standard GAN	Wasserstein GAN (WGAN)
Stability	❌ Unstable	✅ Stable
Loss Meaning	❌ Not meaningful	✅ Correlated with performance
Gradient Issues	❌ Exploding/vanishing	✅ Smoother gradients
Implementation Ease	✅ Simple	⚠️ Requires clipping / GP
Mode Collapse	❌ Common	✅ Rare
Sample Quality	⚠️ Moderate	✅ High (especially with GP)

✅ When to Use WGAN over GAN

Scenario	Use WGAN if:
You face mode collapse in vanilla GANs	
Your generator loss doesn't correlate with sample quality	
You're training on complex, high-dimensional data	
You want more interpretable training behavior	

🧪 Bonus: Code Comparison Repository

You can explore real PyTorch examples:

- **WGAN**: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/wgan
- **Standard GAN**: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/gan

---

Here's a **minimal GPT-2-like Transformer model** implemented in **PyTorch** for **text generation**, with the following core components:

- Token embedding
- Positional encoding
- Multi-head self-attention
- Transformer blocks
- Language modeling head (causal LM)

🧠 GPT-2-like Transformer in PyTorch

✅ Prerequisites

```
pip install torch torchvision torchaudio
```

Full Code Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

# Config
class GPT2Config:
    vocab_size = 50257       # same as GPT-2
    block_size = 128         # context window
    n_layer = 4
    n_head = 4
    n_embd = 256             # embedding size
    dropout = 0.1

# Causal Self-Attention
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        self.n_head = config.n_head
        self.head_dim = config.n_embd // config.n_head

        self.qkv = nn.Linear(config.n_embd, 3 * config.n_embd)
        self.out_proj = nn.Linear(config.n_embd, config.n_embd)
        self.dropout = nn.Dropout(config.dropout)

        # causal mask
        self.register_buffer("mask",
torch.tril(torch.ones(config.block_size,
config.block_size)).unsqueeze(0).unsqueeze(0))

    def forward(self, x):
        B, T, C = x.size()
        qkv = self.qkv(x)
        q, k, v = qkv.chunk(3, dim=-1)

        q = q.view(B, T, self.n_head, self.head_dim).transpose(1, 2)
        k = k.view(B, T, self.n_head, self.head_dim).transpose(1, 2)
        v = v.view(B, T, self.n_head, self.head_dim).transpose(1, 2)

        att = (q @ k.transpose(-2, -1)) / math.sqrt(self.head_dim)
        att = att.masked_fill(self.mask[:, :, :T, :T] == 0, float('-
inf'))
        att = F.softmax(att, dim=-1)
        att = self.dropout(att)

        out = (att @ v).transpose(1, 2).contiguous().view(B, T, C)
        return self.out_proj(out)
```

PROF

```
# Transformer Block
class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.Sequential(
            nn.Linear(config.n_embd, 4 * config.n_embd),
            nn.GELU(),
            nn.Linear(4 * config.n_embd, config.n_embd),
            nn.Dropout(config.dropout),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

# GPT-2 Model
class GPT2(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embed = nn.Embedding(config.vocab_size,
config.n_embd)
        self.pos_embed = nn.Parameter(torch.zeros(1, config.block_size,
config.n_embd))
        self.blocks = nn.Sequential(*[TransformerBlock(config) for _ in
range(config.n_layer)])
        self.ln_f = nn.LayerNorm(config.n_embd)
        self.head = nn.Linear(config.n_embd, config.vocab_size,
bias=False)

        self.block_size = config.block_size
        self.apply(self._init_weights)

    def forward(self, idx, targets=None):
        B, T = idx.size()
        assert T <= self.block_size

        tok_emb = self.token_embed(idx)                # (B, T, C)
        pos_emb = self.pos_embed[:, :T, :]             # (1, T, C)
        x = tok_emb + pos_emb                          # (B, T, C)
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.head(x)

        if targets is None:
            return logits, None
        else:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.view(-1))
```

```
            return logits, loss

    def _init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding)):
            nn.init.normal_(module.weight, mean=0.0, std=0.02)

    @torch.no_grad()
    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -self.block_size:]
            logits, _ = self(idx_cond)
            next_token = torch.argmax(logits[:, -1, :], dim=-1,
keepdim=True)
            idx = torch.cat([idx, next_token], dim=1)
        return idx
```

🔧 Example Usage (with Dummy Data)

```
config = GPT2Config()
model = GPT2(config)

# Simulated dummy input (batch size 1, seq length 8)
x = torch.randint(0, config.vocab_size, (1, 8))
logits, loss = model(x, x)
print("Loss:", loss.item())

# Generate tokens from prompt
generated = model.generate(x, max_new_tokens=20)
print("Generated token IDs:", generated)
```

PROF

🧠 Notes

- This model is **causal**, meaning each token only attends to previous tokens — exactly like GPT-2.
- For production, replace token embedding with a tokenizer like Hugging Face's `GPT2Tokenizer`.
- You can scale layers or use `nn.TransformerDecoderLayer` if you want a shortcut.
- The full GPT-2 uses:
    - 12 layers
    - 768 hidden units
    - 12 heads
    - Positional embeddings, LayerNorm, GELU, etc.
