AI Essentials
Final Project Report
Generative cGAN and cVAE models for depicting different classes of animals

Ignat Tyo

AAI-2502M

1. Введение

В проекте исследовались две генеративные модели — **Conditional VAE (CVAE)** и **Conditional GAN (CGAN)** — примененные к датасету Animals, взятого с Kaggle. Цель заключалась в сравнении подходов, сложности их обучения и качества получаемых изображений при условной генерации по классу. Обе модели использовали одинаковое представление классов и были обучены на одном наборе изображений, что позволило более объективно сопоставить результаты.

2. Данные и подготовка

Были использованы изображения животных, которые предварительно масштабировались, нормализовались и приводились к единому размеру. Метки классов кодировались в one-hot представление. Отдельно формировались пары (изображение, класс), необходимые для условных генеративных моделей. Датасет содержит 5 классов животных, по ~2700 изображений для каждого класса: cat, dog, elephant, horse, lion.

```
tf = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.CenterCrop(IMAGE_SIZE),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
```

Примеры изображений

3. Conditional VAE (CVAE)

CVAE строился на стандартной схеме *encoder–decoder*, где на вход энкодера подавались изображение и код класса. В латентном пространстве модель училась представлять распределение объектов каждого класса. Для обучения использовалась комбинация реконструкционного лосса (например, MSE или BCE) и KLD-регуляризации, которая заставляла латентное пространство приближаться к нормальному распределению.

Особенности:

- **Архитектура**: энкодер состоял из последовательности свёрточных слоёв с BatchNorm и ReLU, после чего вычислялись параметры распределения (μ и $\log\sigma^2$). Декодер восстанавливал изображение из латентного вектора, дополненного эмбеддингом класса.

```python
class Encoder(nn.Module):
    def __init__(self, img_channels=3, base=BASE_CHANNELS, embed_dim=EMBED_DIM, n_classes=NUM_CLASSES, latent_dim=LATENT_DIM):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(img_channels, base, 4, 2, 1), nn.BatchNorm2d(base), nn.ReLU(True),
            nn.Conv2d(base, base*2, 4, 2, 1), nn.BatchNorm2d(base*2), nn.ReLU(True),
            nn.Conv2d(base*2, base*4, 4, 2, 1), nn.BatchNorm2d(base*4), nn.ReLU(True),
            nn.Conv2d(base*4, base*8, 4, 2, 1), nn.BatchNorm2d(base*8), nn.ReLU(True)
        )
        feat_dim = base*8*4*4
        self.label_emb = nn.Embedding(n_classes, embed_dim)

        self.fc_mu = nn.Linear(feat_dim + embed_dim, latent_dim)
        self.fc_logvar = nn.Linear(feat_dim + embed_dim, latent_dim)

    def forward(self, x, labels):
        b = x.size(0)
        f = self.conv(x)
        f = f.view(b, -1)
        le = self.label_emb(labels)
        h = torch.cat([f, le], dim=1)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar
```

```python
class Decoder(nn.Module):
    def __init__(self, img_channels=3, base=BASE_CHANNELS, embed_dim=EMBED_DIM, n_classes=NUM_CLASSES, latent_dim=LATENT_DIM)
        super().__init__()
        self.label_emb = nn.Embedding(n_classes, embed_dim)
        self.latent_input_dim = latent_dim + embed_dim
        self.fc = nn.Linear(self.latent_input_dim, base*8*4*4)

        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(base*8, base*4, 4, 2, 1), nn.BatchNorm2d(base*4), nn.ReLU(True),
            nn.ConvTranspose2d(base*4, base*2, 4, 2, 1), nn.BatchNorm2d(base*2), nn.ReLU(True),
            nn.ConvTranspose2d(base*2, base, 4, 2, 1), nn.BatchNorm2d(base), nn.ReLU(True),
            nn.ConvTranspose2d(base, img_channels, 4, 2, 1),
            nn.Tanh()   # outputs in [-1,1]
        )

    def forward(self, z, labels):
        le = self.label_emb(labels)
        z_cond = torch.cat([z, le], dim=1)
        x = self.fc(z_cond)
        x = x.view(-1, BASE_CHANNELS*8, 4, 4)
        x = self.deconv(x)
        return x
```

- **Репараметризация**: использовался трюк $z=\mu+\sigma\cdot\epsilon$, где $\epsilon\sim N(0,I)$.
- **Обучение**: проходило стабильно, лосс плавно уменьшался. Однако визуально выборки менялись медленно: на некоторых эпохах сэмплы почти не различались. Это объясняется сглаженностью латентного пространства и сильной регуляризацией.

```
Epoch 77: recon=0.11403, kld=0.03145, saved: recon_epoch77.png, samples_epoch77.png
CVAE epoch 78: 100%|██████████| 105/105 [00:21<00:00,  4.91it/s, recon=0.115, kld=0.0318]
Epoch 78: recon=0.11369, kld=0.03151, saved: recon_epoch78.png, samples_epoch78.png
CVAE epoch 79: 100%|██████████| 105/105 [00:21<00:00,  4.97it/s, recon=0.115, kld=0.0317]
Epoch 79: recon=0.11398, kld=0.03138, saved: recon_epoch79.png, samples_epoch79.png
CVAE epoch 80: 100%|██████████| 105/105 [00:21<00:00,  4.90it/s, recon=0.117, kld=0.0325]
Epoch 80: recon=0.11358, kld=0.03153, saved: recon_epoch80.png, samples_epoch80.png
```

4. Conditional GAN (CGAN)

CGAN состоял из генератора, которому подавались шум и one-hot класс, и дискриминатора, получавшего изображения (реальные или сгенерированные) вместе с условием класса. В отличие от CVAE, CGAN оптимизируется через состязательное обучение, что делает его более чувствительным к подбору гиперпараметров.

Особенности:

- **Архитектура генератора**: использовались ConvTranspose слои с Conditional BatchNorm, где параметры γ и β вычислялись из эмбеддинга класса. Это позволяло генератору адаптировать нормализацию под конкретный класс.

```python
class CGenerator(nn.Module):
    def __init__(self, z_dim, n_classes, img_channels=3, base=64, embed_dim=128):
        super().__init__()
        self.base = base
        self.label_emb = nn.Embedding(n_classes, embed_dim)

        input_dim = z_dim + embed_dim
        self.fc = nn.Linear(input_dim, base*8*4*4)

        self.deconv1 = nn.ConvTranspose2d(base*8, base*4, 4, 2, 1, bias=False)
        self.cbn1 = ConditionalBatchNorm2d(base*4, n_classes, embed_dim)
        self.deconv2 = nn.ConvTranspose2d(base*4, base*2, 4, 2, 1, bias=False)
        self.cbn2 = ConditionalBatchNorm2d(base*2, n_classes, embed_dim)
        self.deconv3 = nn.ConvTranspose2d(base*2, base, 4, 2, 1, bias=False)
        self.cbn3 = ConditionalBatchNorm2d(base, n_classes, embed_dim)

        self.final = nn.ConvTranspose2d(base, img_channels, 4, 2, 1)

    def forward(self, z, labels):
        l = self.label_emb(labels)
        x = torch.cat([z, l], dim=1)
        x = self.fc(x)
        x = x.view(-1, self.base*8, 4, 4)

        x = F.relu(self.cbn1(self.deconv1(x), labels))
        x = F.relu(self.cbn2(self.deconv2(x), labels))
        x = F.relu(self.cbn3(self.deconv3(x), labels))
        x = torch.tanh(self.final(x))
        return x
```

- **Архитектура дискриминатора**: применялся spectral normalization для стабилизации обучения. Дискриминатор решал две задачи: бинарная классификация «реальное/сгенерированное» и предсказание класса изображения.

```python
class CDiscriminator(nn.Module):
    def __init__(self, n_classes, img_channels=3, base=64):
        super().__init__()
        self.features = nn.Sequential(
            spectral_norm(nn.Conv2d(img_channels, base, 4, 2, 1)),
            nn.LeakyReLU(0.2, inplace=True),
            spectral_norm(nn.Conv2d(base, base*2, 4, 2, 1)),
            nn.LeakyReLU(0.2, inplace=True),
            spectral_norm(nn.Conv2d(base*2, base*4, 4, 2, 1)),
            nn.LeakyReLU(0.2, inplace=True),
            spectral_norm(nn.Conv2d(base*4, base*8, 4, 2, 1)),
            nn.LeakyReLU(0.2, inplace=True),
            nn.AdaptiveAvgPool2d(1)
        )
        self.fc_adv = spectral_norm(nn.Linear(base*8, 1))
        self.fc_cls = spectral_norm(nn.Linear(base*8, n_classes))

    def forward(self, x):
        feat = self.features(x).view(x.size(0), -1)
        adv_out = self.fc_adv(feat).squeeze(1)
        cls_out = self.fc_cls(feat)
        return adv_out, cls_out
```

- **Обучение**: требовало аккуратного выбора learning rate для генератора и дискриминатора, а также проверок на *mode collapse*. При неудачном балансе генератор часто коллапсировал либо игнорировал условие класса.

```
cGAN epoch 97/100: 100%|██████████| 105/105 [00:18<00:00,  5.60it/s, lossD=2.04, lossG=1.23]
cGAN epoch 98/100: 100%|██████████| 105/105 [00:18<00:00,  5.53it/s, lossD=1.85, lossG=1.34]
cGAN epoch 99/100: 100%|██████████| 105/105 [00:18<00:00,  5.55it/s, lossD=2.32, lossG=-0.361]
cGAN epoch 100/100: 100%|██████████| 105/105 [00:18<00:00,  5.53it/s, lossD=2.08, lossG=1.63]
```

5. Эксперименты

```python
# Параметры CGAN
IMAGE_SIZE = 64
BATCH = 128
LATENT_DIM = 256
NUM_EPOCHS_GAN = 100
LR = 2e-4
SAMPLE_PER_CLASS = 4
NUM_CLASSES = 5
```

```python
# Параметры CVAE
IMAGE_SIZE = 64
BATCH = 128
LATENT_DIM = 256
NUM_EPOCHS = 80
LR = 2e-4
EMBED_DIM = 128
BASE_CHANNELS = 64
BETA = 1.0
SAMPLE_PER_CLASS = 6
```

Обе модели обучались в одинаковых условиях: одни и те же батчи, одинаковое one-hot кодирование и сходные ограничения по эпохам. CVAE демонстрировал гораздо более стабильный лосс, тогда как у CGAN наблюдались резкие перепады в зависимости от того, насколько хорошо синхронизированы скорости обучения генератора и дискриминатора. Было проведено несколько запусков CGAN с разными значениями lr; устойчивее работала конфигурация с уменьшенным lr у дискриминатора.

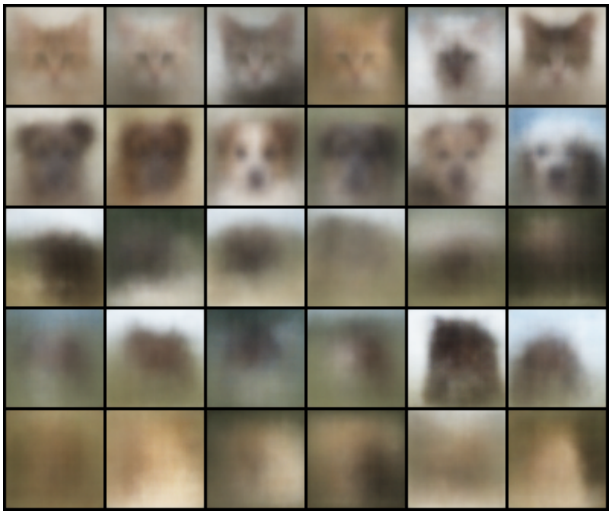
В ходе экспериментов сравнивались: визуальное качество сэмплов, способность различать классы, устойчивость к переобучению, скорость сходимости и чувствительность к гиперпараметрам.

6. Результаты

В ходе экспериментов обе модели показали ограниченное качество генерации изображений.

CVAE:

- Генерируемые изображения имели низкую детализацию: на них было трудно различить животных.
- Исключение составляли кошки и собаки — для этих классов модель восстанавливала узнаваемые формы.
- На разных эпохах выборки почти не отличались друг от друга: сэмплы оставались похожими, что указывает на сильную сглаженность латентного пространства.
- Несмотря на стабильное обучение и плавное уменьшение лосса, визуальное качество не улучшалось.



CGAN:

- Генератор выдавал изображения плохого качества: животные оставались неразличимыми, формы размытыми.
- Начиная примерно с 40-й эпохи, качество стабилизировалось и перестало заметно улучшаться.
- Даже после длительного обучения изображения так и не стали ясными и чёткими.

7. Обсуждение

Причины низкого качества генерации связаны с несколькими факторами:

- **Mode collapse**: несмотря на использование R1-регуляризации, instance noise и балансировки шагов дискриминатора/генератора, генератор склонен к коллапсу мод. Это проявлялось в однотипных изображениях и отсутствии разнообразия.
- **Сложность датасета**: для большинства классов животных модель не смогла выделить устойчивые признаки, кроме кошек и собак, которые более выражены в данных.
- **Архитектурные ограничения**: CVAE страдает от чрезмерной регуляризации латентного пространства, что делает выборки сглаженными. CGAN, напротив, более гибок, но требует тонкой настройки гиперпараметров и может легко деградировать.
- **Недостаточная мощность моделей**: выбранные архитектуры (базовые ConvTranspose/Conv слои) могли быть недостаточны для генерации изображений высокого качества на разрешении 64x64. Более глубокие или современные архитектуры (например, ResNet-блоки, attention-механизмы) могли бы улучшить результаты.

8. Заключение

Эксперименты показали разные пределы подходов. CVAE оказался устойчивым и предсказуемым: передавал базовую форму объектов и учитывал условность, но терял детали из-за регуляризации и простой архитектуры. CGAN, напротив, проявил нестабильность: генератор не удерживал равновесие с дискриминатором и генерировал случайные текстуры, игнорируя классовую информацию. Качество результатов определяется архитектурными свойствами моделей. CVAE стабилен, но ограничен в выразительности; CGAN способен на более реалистичные изображения, но при слабой архитектуре, малом датасете и отсутствии современных методов стабилизации не достигает обученного состояния.