Technische Universität Bergakademie Freiberg
Institut für Numerische Mathematik und Optimierung
Prof. Dr. O. Rheinbach
Dr. S. Köhler

Wintersemester 2025/26
08.01.2026

Praktikum Wissenschaftliches Rechnen / Scientific Programming Project

Programming Project

Exercise 1

Project Overview

Implement and improve a **Conditional Generative Adversarial Network (cGAN)** for image generation using the Tiny ImageNet dataset. You will work in groups of 2–3 people to develop a system that can generate realistic images conditioned on specific class labels.
The project consists of two main phases:

a) **Baseline Implementation**: Implement a conditional DCGAN (Deep Convolutional GAN) architecture.

b) **Architecture Improvement**: Design and implement your own improved generator and discriminator architectures, followed by systematic hyperparameter tuning.

Dataset

Please, use the **CIFAR-10 dataset** dataset, which consists of:

- Image size: 32×32 pixels (RGB)
- Original dataset: 10 classes with 5 000 training images per class and 1 000 test images per class

The CIFAR-10 dataset is available by torchvision.datasets:

```
from torchvision import datasets, transforms

train_transform = get_train_transform()
test_transform = get_test_transform()

train_dataset = datasets.CIFAR10(
    root=data_root,
    train=True,
    download=True,
    transform=train_transform,
)

test_dataset = datasets.CIFAR10(
    root=data_root,
    train=False,
    download=True,
    transform=test_transform,
)
```

Normalization Parameters

For GAN training with Tanh output activation, you should scale images to the range $[-1, 1]$. This ensures that the generator's output range (Tanh produces values in $[-1, 1]$) matches the range of real images fed to the discriminator. Using matching ranges is crucial for stable training, as it prevents the discriminator from distinguishing real from fake images based solely on their value ranges rather than their visual quality.
Use the following normalization parameters:

```
TIN_MEAN = (0.5, 0.5, 0.5)
TIN_STD = (0.5, 0.5, 0.5)
```

This normalization transforms images from $[0, 1]$ to $[-1, 1]$ using the formula: $\text{normalized} = (x - 0.5)/0.5$.
You can apply normalization in your data preprocessing pipeline using torchvision.transforms.Compose:

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=TIN_MEAN, std=TIN_STD)
])
```

Data Augmentation

Data augmentation is a technique to artificially increase the diversity of your training data by applying random transformations. For GAN training, augmentation is particularly important for the discriminator: it helps prevent overfitting to the limited training set and encourages the discriminator to focus on meaningful image features rather than memorizing specific examples.
Make yourself familiar with different transformations for data augmentation provided by torchvision.transforms, such as:

- RandomHorizontalFlip: Randomly flips images horizontally
- RandomRotation: Applies small random rotations
- ColorJitter: Randomly changes brightness, contrast, saturation, and hue
- RandomResizedCrop: Randomly crops and resizes portions of the image

You can use torchvision.transforms.Compose to chain multiple transformations together. Research which transformations are appropriate for your selected Tiny ImageNet classes—for example, horizontal flipping makes sense for most natural objects but may not be appropriate for images containing text or numbers.
Attention: Only the training dataset should be augmented, *not* the validation/test dataset. The validation and test sets should use only ToTensor() and Normalize() transformations.
Important for GAN training: Data augmentation is typically applied only to the real images fed to the discriminator, not to the images generated by the generator. This asymmetry is intentional—the generator should learn to produce realistic images without artificial augmentation.

Baseline Conditional DCGAN

Implement a conditional DCGAN with the following specifications:

Generator Architecture

The generator should:

- Accept input: latent vector $\mathbf{z} \in \mathbb{R}^{n_z}$ ($n_z = 100$) and class label embedding
- Use transposed convolutions (ConvTranspose2d) for upsampling
- Apply batch normalization after each transposed convolution (except the output layer)
- Use ReLU activations (except Tanh for the output layer)
- Output: $64 \times 64 \times 3$ RGB image with pixel values in $[-1, 1]$

Use the following architecture:

a) Project and reshape: $(n_z + n_{\text{embed}}) \to 512 \times 4 \times 4$

b) ConvTranspose2d: $512 \to 256$ (output: 8×8)

c) ConvTranspose2d: $256 \to 128$ (output: 16×16)

d) ConvTranspose2d: $128 \to 64$ (output: 32×32)

e) ConvTranspose2d: $64 \to 3$ (output: 64×64)

Discriminator Architecture

The discriminator should:

- Accept input: $64 \times 64 \times 3$ RGB image
- Use strided convolutions for downsampling (avoid pooling layers)
- Apply batch normalization after convolutions (except the first layer)
- Use LeakyReLU activations with negative slope 0.2
- Output: single scalar (real/fake probability)

Use the architecture:

a) Conv2d: $3 \to 64$ (output: 32×32, stride=2)

b) Conv2d: $64 \to 128$ (output: 16×16, stride=2)

c) Conv2d: $128 \to 256$ (output: 8×8, stride=2)

d) Conv2d: $256 \to 512$ (output: 4×4, stride=2)

e) Flatten and project to 1 output (+ class embedding)

Conditional Mechanism

Implement conditioning by:

- Creating learnable class embeddings: Embedding($n_{\text{classes}}, n_{\text{embed}}$)
- **Generator**: Concatenate class embedding to latent vector $\mathbf{z}$

Hint: For the embedding, use torch.nn.Embedding.

Training

- **Loss function**: Binary cross-entropy (vanilla GAN loss)
- **Optimizers**: Adam with $\beta_1 = 0.5$, $\beta_2 = 0.999$
- **Learning rates**: Start with 2×10^{-4} for both networks
- **Training ratio**: Train discriminator and generator with 1:1 ratio
- **Batch size**: 64–128 (depending on GPU memory)

Improved Architecture

Design and implement your own improved architectures for both the generator and discriminator. This is the core creative component of the project where you should demonstrate your understanding of deep learning principles and GAN training dynamics.
You are free to explore different approaches:

- **Network architecture**: Experiment with different layer configurations, network depths, and channel dimensions. Consider architectural patterns from recent literature such as residual connections, attention mechanisms, or progressive growing strategies.
- **Activation functions**: While the baseline uses ReLU and LeakyReLU, you might explore alternatives like ELU, GELU, or Swish where appropriate.
- **Normalization techniques**: Beyond batch normalization, consider layer normalization, instance normalization, or spectral normalization for improved training stability.
- **Loss functions**: Explore alternatives to vanilla GAN loss, such as Wasserstein GAN with Gradient Penalty (WGAN-GP) or Least-Squares GAN (LSGAN), which can provide more stable training.

Attention: It is *not* sufficient to simply add more layers to the baseline implementation. Your improvements should be motivated by understanding of the training dynamics and architectural principles. Document your design choices and explain *why* you made them, not just *what* you changed.

Hyperparameter Tuning

Systematically tune your hyperparameters to optimize training stability and output quality. Hyperparameter tuning is not random trial-and-error—it should be a structured process where you vary parameters methodically and document the results.
Consider tuning the following hyperparameters:

- **Learning rates**: Generator and discriminator learning rates can be set independently. Often, using different learning rates helps maintain training balance.
- **Batch size**: Affects training stability and gradient estimates. Larger batches provide more stable gradients but require more memory.
- **Latent dimension size**: The dimensionality of the noise vector $\mathbf{z}$. Higher dimensions provide more variation but may make training harder.
- **Embedding dimension**: The size of the class label embeddings for conditioning.
- **Training ratios**: You can train the discriminator more frequently than the generator (e.g., 2:1 or 3:1 ratio) to prevent generator collapse.
- **Regularization**: Explore dropout rates, weight decay, or gradient penalties to improve generalization and training stability.

Document your tuning process thoroughly:

- Which hyperparameters did you search over, and in what order?
- What ranges did you test? (e.g., learning rates from 10^{-5} to 10^{-3})
- How did you evaluate which configuration was best? What metrics or visual assessments did you use?
- Show comparison plots: loss curves, sample quality evolution, or quantitative metrics across different configurations.
- Did you use grid search, random search, or manual tuning? Justify your approach.

A well-documented tuning process demonstrates scientific rigor and helps others (including your future self) understand your decisions.

Reproducibility Requirements

Your implementation must be fully reproducible, meaning that anyone (including yourself months from now) should be able to run your code and obtain the same results.
You must provide training scripts for both the baseline architecture and your improved architecture. These scripts should function as "black boxes" — a user should be able to run them with minimal setup and reproduce your reported results without needing to understand the internal implementation details.

Code Organization

Organize your code with a clear, modular structure. Use the following directory structure (you may add additional files as needed):

```
project/
|-- README.md                  # Setup and usage instructions
|-- requirements.txt           # Pinned package versions
|-- config/
|   |-- baseline_config.yaml   # Baseline DCGAN hyperparameters
|   +-- improved_config.yaml   # Your improved model config
|-- data/
|   +-- dataloader.py          # Dataset loading and preprocessing
|-- models/
|   |-- generator.py           # Generator architectures
|   |-- discriminator.py       # Discriminator architectures
|   +-- cgan.py                # Complete cGAN implementation
|-- training/
|   |-- train.py               # Training script with CLI
|   +-- losses.py              # Loss function implementations
|-- evaluation/
|   |-- evaluate.py            # Generate samples, metrics
|   +-- visualize.py           # Plotting and visualization
|-- checkpoints/               # Saved model checkpoints
+-- results/                   # Generated images, plots, logs
```

The `README.md` should include clear instructions for:

- Environment setup and dependency installation
- Dataset download and preparation
- How to run training scripts for baseline and improved models

- How to generate samples and visualizations
- Expected training times and hardware requirements

Random Seed Management

Random number generators are used throughout machine learning for weight initialization, data shuffling, dropout, and data augmentation. To ensure reproducible results, you must control all sources of randomness by setting seeds for all libraries you use.
Ensure reproducibility by setting seeds at the start of your training script:

```python
import random
import numpy as np
import torch

def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

Note: Setting cudnn.deterministic = True ensures deterministic behavior but may slightly reduce training speed. Setting cudnn.benchmark = False prevents PyTorch from automatically selecting the fastest algorithms, which can introduce non-determinism.

Configuration Files

Configuration files separate hyperparameters from code, making it easy to run experiments with different settings without modifying the source code. Use YAML format for readability and ease of editing.
Create separate configuration files for the baseline architecture and your improved architecture with all relevant hyperparameters:

```yaml
# Example: baseline_config.yaml
model:
  latent_dim: 100
  embed_dim: 50
  generator_channels: [512, 256, 128, 64]
  discriminator_channels: [64, 128, 256, 512]

training:
  batch_size: 128
  num_epochs: 100
  lr_g: 0.0002
  lr_d: 0.0002
  beta1: 0.5
  beta2: 0.999

data:
  num_classes: 15
  image_size: 64

seed: 42
```

Your training script should load these configuration files and use them to initialize your models and training parameters. This allows others to reproduce your exact experimental setup.

Code Submission

You must submit your complete code repository including:

- All source code (organized according to the structure specified above)
- `README.md` with clear setup and usage instructions
- `requirements.txt` with exact package versions (use `pip freeze`)
- Configuration files for all experiments (baseline and improved)
- Documentation of your hyperparameter search process

Note: You do *not* need to submit trained model checkpoints or generated images—these are too large. However, your code must be able to reproduce these results when run.
To submit your code, send an email to the exercise course leader and request an upload link. Include your group members' names in the email.

Presentation

Prepare a 30–35 minute presentation that demonstrates both your technical understanding and your experimental results. Structure your presentation to cover the following topics:

Theory

- **Introduction to GANs**: Explain the adversarial training principle.
- **Conditional GANs**: How do you incorporate class labels into the GAN framework?

Baseline Results

- **Baseline DCGAN architecture**: Show your architecture diagram or a clear description of the layers
- **Training dynamics**: Present training curves showing generator and discriminator losses over epochs
- **Generated samples**: Display a grid of generated images from each of your selected classes
- **Observed issues or limitations**: Be honest about problems you encountered with the baseline
- **Discriminator accuracy**: Show how well the discriminator distinguishes real vs. fake images over training
- **Latent space exploration**:
  - *Interpolation*: Generate images by smoothly interpolating between two latent vectors with the same class label. This demonstrates whether your latent space is smooth and meaningful.
  - *Class variation*: Fix a latent vector but change the class label. Show how the same "base structure" changes across different classes.

Improved Architecture

- **Architectural innovations**: Clearly explain your design choices. *Why* did you make these changes? What problems were you trying to solve?
- **Hyperparameter tuning**: Show your systematic search process. Present comparison plots or tables showing how different configurations performed.
- **Comparison**: Side-by-side visual comparisons of baseline vs. improved results for the same classes and latent vectors
- **Quantitative evaluation**: Present discriminator accuracy and any other metrics you used

Technical Resources

Key References

- Goodfellow et al., "Generative Adversarial Networks" (2014), https://arxiv.org/abs/1406.2661
- Mirza & Osindero, "Conditional Generative Adversarial Nets" (2014), https://arxiv.org/abs/1411.1784