

Deep Learning: ECE-7123

Homework 3

Saad Zubairi
shz2020

May 2, 2025

Contents

1 Problem 1: Minimax optimization	3
a) Saddle point	3
b) Gradient descent/ascent update rules	3
c) Convergence of the update rules	4
d) Regular gradient descent on both variables	5
2 Vision-Language models	6
a) Rationale	6
b) Implementation Changes	6
c) Inference Examples	7
i. Prompt: “A cat sitting with a human”	7
ii. Prompt: “A crowd gathered in a park”	7
iii. Prompt: “A dog sitting alone in a park”	8
iv. Prompt: “A man skateboarding in the city”	8
v. Prompt: “All kinds of food on a table”	8
d) Evaluation	9
i. Prompt: “A cat sitting with a human”	9
ii. Prompt: “A crowd gathered in a park”	9
iii. Prompt: “A dog sitting alone in a park”	9
iv. Prompt: “A man skateboarding in the city”	9
v. Prompt: “All kinds of food on a table”	10
e) Final Observations	10
f) Impact of Scaling	10
3 Generative Adversarial Networks (GANs)	12
a) Code Implementation	12
i. config.py	12
ii. dataset.py	12
b) models.py	13
c) utils.py	14
d) train.py	14
e) generate.py	16

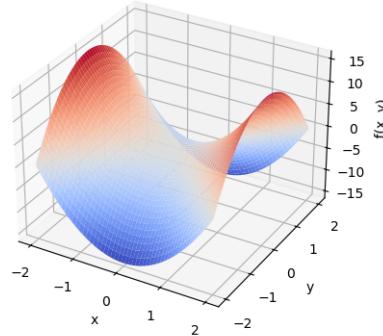
1 Problem 1: Minimax optimization

Given the function:

$$\min_x \max_y f(x, y) = 4x^2 - 4y^2.$$

Which can be plotted as:

$$f(x, y) = 4x^2 - 4y^2$$



a) Saddle point

To find the saddle point, we can take the partial derivatives with respect to x and y :

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) = (8x, -8y)$$

Setting $\nabla f(x, y) = 0$, we solve:

$$8x = 0 \quad \text{and} \quad -8y = 0$$

This gives the saddle point at $(x, y) = (0, 0)$.

b) Gradient descent/ascent update rules

To derive the update rules for gradient descent in x and gradient ascent in y , we start with the function $f(x, y) = 4x^2 - 4y^2$. The gradient of $f(x, y)$ is given by:

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) = (8x, -8y).$$

For gradient descent in x , we update x by moving in the direction opposite to the gradient:

$$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x}.$$

Substituting $\frac{\partial f}{\partial x} = 8x_t$, we get:

$$x_{t+1} = x_t - \eta(8x_t) = x_t(1 - 8\eta).$$

For gradient ascent in y , we update y by moving in the direction of the gradient:

$$y_{t+1} = y_t + \eta \frac{\partial f}{\partial y}.$$

Substituting $\frac{\partial f}{\partial y} = -8y_t$, we get:

$$y_{t+1} = y_t + \eta(-8y_t) = y_t(1 + 8\eta).$$

Thus, the update rules are:

$$x_{t+1} = (1 - 8\eta)x_t, \quad y_{t+1} = (1 + 8\eta)y_t.$$

c) Convergence of the update rules

To ensure convergence, we analyze the update rules:

$$x_{t+1} = (1 - 8\eta)x_t, \quad y_{t+1} = (1 + 8\eta)y_t.$$

For convergence of x_t in gradient descent, the factor $(1 - 8\eta)$ must satisfy:

$$|1 - 8\eta| < 1.$$

Solving this inequality:

$$-1 < 1 - 8\eta < 1.$$

Subtracting 1 from all sides:

$$-2 < -8\eta < 0.$$

Dividing through by -8 (reversing the inequality):

$$0 < \eta < \frac{1}{4}.$$

For y_t in gradient ascent, the factor $(1 + 8\eta)$ must satisfy:

$$|1 + 8\eta| < 1.$$

Solving this inequality:

$$-1 < 1 + 8\eta < 1.$$

Subtracting 1 from all sides:

$$-2 < 8\eta < 0.$$

Dividing through by 8:

$$-\frac{1}{4} < \eta < 0.$$

Since η must be positive for gradient descent and ascent to make sense, we combine the results:

$$0 < \eta < \frac{1}{4}.$$

Thus, the range of allowable step sizes is:

$$0 < \eta < \frac{1}{4}$$

d) Regular gradient descent on both variables

To analyze regular gradient descent on both variables x and y , we update both using the gradient of $f(x, y)$. The gradient is:

$$\nabla f(x, y) = (8x, -8y).$$

The update rules for gradient descent on x and y are:

$$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x}, \quad y_{t+1} = y_t - \eta \frac{\partial f}{\partial y}.$$

Substituting the partial derivatives:

$$\begin{aligned} x_{t+1} &= x_t - \eta(8x_t) = x_t(1 - 8\eta), \\ y_{t+1} &= y_t - \eta(-8y_t) = y_t(1 + 8\eta). \end{aligned}$$

Thus, the update rules are:

$$x_{t+1} = (1 - 8\eta)x_t, \quad y_{t+1} = (1 + 8\eta)y_t.$$

Dynamics of the updates

- For x_t , the factor $(1 - 8\eta)$ determines the convergence. If $0 < \eta < \frac{1}{4}$, then $|1 - 8\eta| < 1$, ensuring that x_t decays to 0.
- For y_t , the factor $(1 + 8\eta)$ determines the behavior. Since $(1 + 8\eta) > 1$ for $\eta > 0$, y_t grows exponentially, leading to divergence.

2 Vision-Language models

Overview

In this experiment, we evaluate three different combinations of image and text encoders for CLIP-style contrastive learning. The goal is to assess how different architectural choices affect the model’s ability to align images and captions.

Model Configurations

Combination 1: ResNet-50 + DistilBERT

- Image Encoder: `resnet50` (output dim: 2048)
- Text Encoder: `distilbert-base-uncased` (output dim: 768)
- Projection Dim: 256

Combination 2: ResNet-18 + BERT

- Image Encoder: `resnet18` (output dim: 512)
- Text Encoder: `bert-base-uncased` (output dim: 768)
- Projection Dim: 256

Combination 3: ResNet-34 + RoBERTa

- Image Encoder: `resnet34` (output dim: 512)
- Text Encoder: `roberta-base` (output dim: 768)
- Projection Dim: 256

a) Rationale

The encoder combinations were selected semi-randomly to reflect varying model scales. The purpose is to observe whether larger encoders (e.g., ResNet-50, RoBERTa) yield visibly better alignment without tuning hyperparameters.

b) Implementation Changes

Config Changes

All modifications are made in the `config` class.

```
model_name = 'resnet50'
image_embedding = 2048
text_encoder_model = 'roberta-base'
text_tokenizer = 'roberta-base'
text_embedding = 768
projection_dim = 512
```

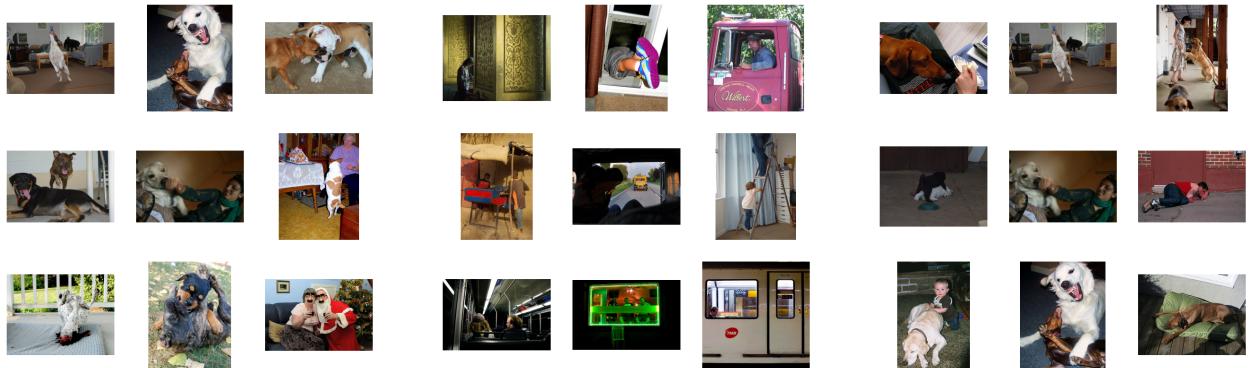
Encoder Changes

In addition to updating the `config` class, the encoders were modified throughout the implementation to support RoBERTa. This included:

- Updating the forward pass to handle RoBERTa's tokenization and embedding outputs.
- Adjusting the projection layers to match the output dimensions of RoBERTa (768) and the specified projection dimension (256).
- Updating the `get_image_embeddings` function to use `AutoTokenizer` for tokenization; compatible with the updated `text_tokenizer`.

c) Inference Examples

i. Prompt: “A cat sitting with a human”



ii. Prompt: “A crowd gathered in a park”



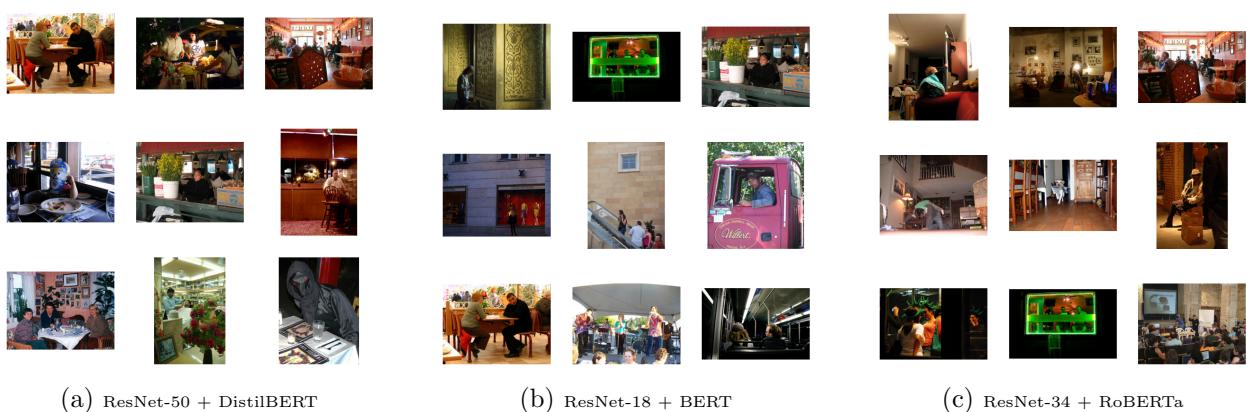
iii. Prompt: “A dog sitting alone in a park”



iv. Prompt: “A man skateboarding in the city”



v. Prompt: “All kinds of food on a table”



d) Evaluation

i. Prompt: “A cat sitting with a human”

This prompt revealed major limitations across all models.

- **Model A:** Returned dog images—incorrect, but at least animal-adjacent.
- **Model B:** Performed the worst: no animals, minimal context alignment, and often returned scenes with no semantic relevance.
- **Model C:** Came up with a range of unrelated content, including vehicles and humans, with no cats in sight.

Observation: The models seem to generalize poorly to cat-related queries, possibly due to underrepresentation in the dataset or visual confusion with dogs.

ii. Prompt: “A crowd gathered in a park”

All models demonstrated some degree of success.

- **Model A:** Retrieved crowd scenes, several of which included greenery and open spaces resembling parks.
- **Model B:** Managed decent crowd representations, albeit with minimal park-like settings.
- **Model C:** Captured the crowd element well but failed to recognize the park context, retrieving urban or event scenes instead.

Observation: Crowd detection appears robust, but fine-grained location understanding (e.g., park vs street) varies by model.

iii. Prompt: “A dog sitting alone in a park”

Performance improved noticeably here.

- **Model A:** The most consistent in retrieving visually relevant scenes.
- **Model B:** Adequate but unremarkable.
- **Model C:** Despite earlier underperformance, correctly surfaced a few images matching the “sitting dog” description.

Observation: The models exhibit strong prior alignment for the concept of ”dog + park,” likely due to frequent co-occurrence in training data.

iv. Prompt: “A man skateboarding in the city”

A clear separation in model quality emerged.

- **Model A:** Provided highly accurate results, with multiple images of urban skateboarding scenes.

- **Model B:** More ambiguous—some city scenes, limited motion context.
- **Model C:** Performed poorly, retrieving unrelated activities (including what appeared to be a tennis match).

Observation: Motion-related prompts require both object and action recognition; only Model A appears capable of handling this dual requirement reliably.

v. Prompt: “All kinds of food on a table”

Mixed results with some surprises.

- **Model A:** Showed multiple dining scenes, often including food, people at tables, or floral arrangements. Semantically close, though not perfect.
- **Model B:** Failed entirely, presenting irrelevant images such as trucks and individuals using phones.
- **Model C:** Returned vague or partial matches—some table elements, few actual food visuals.

Observation: Abstract or multi-object prompts (e.g., “all kinds of food”) seem more challenging for the smaller models. Model A again showed the strongest contextual alignment.

e) Final Observations

- Model A (ResNet-50 + DistilBERT): This model consistently outperformed the others in both scene and object understanding. It handled composite prompts well and recovered semantically aligned images even when fine-grained details were missing. The larger image encoder (ResNet-50) likely contributed to its superior performance, compensating for the relatively smaller text encoder (DistilBERT).
- Model B (ResNet-18 + BERT): This model lagged significantly behind the others. While BERT is more powerful than DistilBERT, the weak image encoder (ResNet-18) likely limited the overall performance. The smaller image encoder struggled to extract sufficient visual features, leading to poor alignment with text embeddings.
- Model C (ResNet-34 + RoBERTa): This model was inconsistent—sometimes retrieving promising matches (especially for prompts involving dogs and parks), but often drifting semantically. The larger text encoder (RoBERTa) provided strong language representations, but the mid-sized image encoder (ResNet-34) may have been insufficient to fully leverage this capability, resulting in unstable grounding between image and text modalities.

f) Impact of Scaling

Scaling plays a critical role in the performance of vision-language models. Larger image encoders, such as ResNet-50, provide richer visual feature representations, which are crucial for aligning with text embeddings. Similarly, more powerful text encoders, such as RoBERTa, offer better semantic understanding of textual prompts. However, the benefits of scaling are not uniform across modalities—imbalances between the capacities of the image and text encoders can lead to suboptimal performance. For example, Model C’s strong text encoder was not fully utilized due to its mid-sized image encoder.

In general, scaling both the image and text encoders proportionally tends to yield the best results, as seen with Model A. However, this comes at the cost of increased computational requirements, which must be balanced against the desired performance and available resources.

3 Generative Adversarial Networks (GANs)

Following is the implementation of a simple GAN using PyTorch. The code is structured to include the generator and discriminator networks, the training loop, and the evaluation of generated images.

a) Code Implementation

All code files can be found [here](#).

i. config.py

This file contains the configuration settings for the GAN, including hyperparameters and model architecture details.

Listing 1: Hyperparameters and Paths

```
DATA_DIR      = "data"
OUTPUT_DIR    = "outputs"
LOG_DIR       = "logs"

IMG_SIZE      = 28
CHANNELS      = 1
LATENT_DIM    = 100

BATCH_SIZE    = 64
LR             = 1e-4
BETAS          = (0.5, 0.999)

EPOCHS        = 50
SAVE_EPOCHS   = {10, 30, 50}

SEED          = 42
```

ii. dataset.py

Listing 2: DataLoader Class

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import config

# Function to create and return a DataLoader for the FashionMNIST dataset
def get_dataloader():
    # Define a series of transformations to preprocess the images
    tf = transforms.Compose([
        transforms.Resize(config.IMG_SIZE),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,)),
    ])

    # Load the FashionMNIST dataset with the specified transformations
    ds = datasets.FashionMNIST(
```

```

        root=config.DATA_DIR,
        train=True,
        download=True,
        transform=tf
    )

    # Create and return a DataLoader for the dataset
    return DataLoader(
        ds,
        batch_size=config.BATCH_SIZE,
        shuffle=True,
        num_workers=4,
        pin_memory=True,
    )
)

```

b) models.py

Listing 3: Discriminator and Generator Models

```

import torch.nn as nn
import config

# Discriminator: A CNN-based model to classify real vs. fake images
class Discriminator(nn.Module):
    def __init__(self, channels=config.CHANNELS, feat=64, slope=0.3, drop=0.3):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(channels, feat, 5, 2, 2, bias=False),  # 1x28x28 -> 64x14x14
            nn.LeakyReLU(slope, inplace=True),
            nn.Dropout(drop),
            nn.Conv2d(feat, feat*2, 5, 2, 2, bias=False),    # 64x14x14 -> 128x7x7
            nn.LeakyReLU(slope, inplace=True),
            nn.Dropout(drop),
        )
        ds = config.IMG_SIZE // 4
        self.fc = nn.Linear(feat*2*ds*ds, 1, bias=False)    # 128x7x7 -> scalar

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        return self.fc(x).view(-1)

# Generator: A CNN-based model to generate images from random noise
class Generator(nn.Module):
    def __init__(self, latent_dim=config.LATENT_DIM, channels=config.CHANNELS, feat=64, slope=0.3):
        super().__init__()
        ds = config.IMG_SIZE // 4
        self.fc = nn.Sequential(
            nn.Linear(latent_dim, feat*4*ds*ds, bias=False),  # 100 -> 256x7x7
            nn.BatchNorm2d(feat*4*ds*ds),
            nn.LeakyReLU(slope, inplace=True),
        )
        self.deconv = nn.Sequential(
            nn.Unflatten(1, (feat*4, ds, ds)),                      # Reshape to 256x7x7
            nn.ConvTranspose2d(feat*4, feat*2, 5, 2, 2, output_padding=1, bias=False),
            nn.BatchNorm2d(feat*2),
            nn.LeakyReLU(slope, inplace=True),
            nn.ConvTranspose2d(feat*2, feat, 5, 2, 2, output_padding=1, bias=False),
            nn.BatchNorm2d(feat),
            nn.LeakyReLU(slope, inplace=True),
            nn.ConvTranspose2d(feat, channels, 5, 1, 2, bias=False),
            nn.Tanh(),
        )
)

```

```

    )

def forward(self, z):
    x = self.fc(z)
    return self.deconv(x)

```

c) utils.py

Listing 4: Utilities

```

import os
import torch
import torch.nn as nn
from torchvision.utils import save_image, make_grid
import config

# Initialize weights for Conv2d, ConvTranspose2d, and Linear layers
def weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.Linear)):
        nn.init.normal_(m.weight, 0.0, 0.02)

# Save generated samples as a grid image
def save_samples(gen, epoch, fixed_noise):
    gen.eval() # Set generator to evaluation mode
    # Disable gradient computation
    with torch.no_grad():
        # Normalize to [0,1]
        imgs = gen(fixed_noise).add(1).div(2)
        # Create grid of images
        grid = make_grid(imgs, nrow=8)
        # Ensure output directory exists
        os.makedirs(config.OUTPUT_DIR, exist_ok=True)
        # Save grid image
        save_image(grid, f"{config.OUTPUT_DIR}/epoch_{epoch}.png")
    # Restore generator to training mode
    gen.train()

```

d) train.py

Listing 5: Training Loop

```

import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm

import config
from dataset import get_dataloader
from models import Generator, Discriminator
from utils import weights_init, save_samples

```

```

def train():
    # Set random seed for reproducibility
    torch.manual_seed(config.SEED)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Load dataset
    dl = get_dataloader()
    # Initialize generator
    gen = Generator().to(device)
    # Initialize discriminator
    disc = Discriminator().to(device)
    # Apply weight initialization to generator
    gen.apply(weights_init)
    # Apply weight initialization to discriminator
    disc.apply(weights_init)

    # Optimizer for generator
    opt_g = optim.Adam(gen.parameters(), lr=config.LR, betas=config.BETAS)
    # Optimizer for discriminator
    opt_d = optim.Adam(disc.parameters(), lr=config.LR, betas=config.BETAS)
    # Loss function
    criterion = nn.BCEWithLogitsLoss()

    # Fixed noise for generating samples
    fixed_noise = torch.randn(64, config.LATENT_DIM, device=device)
    # TensorBoard writer for logging
    writer = SummaryWriter(config.LOG_DIR)

    step = 0
    for ep in range(1, config.EPOCHS+1):
        # Progress bar for each epoch
        loop = tqdm(dl, desc=f"Epoch {ep}/{config.EPOCHS}")
        for real, _ in loop:
            real = real.to(device)
            bs = real.size(0)
            noise = torch.randn(bs, config.LATENT_DIM, device=device)
            fake = gen(noise)

            # Discriminator step

            opt_d.zero_grad()
            d_real = disc(real)
            d_fake = disc(fake.detach())
            loss_d = criterion(d_real, torch.ones_like(d_real)) + criterion(d_fake, torch.zeros_like(d_real))
            loss_d.backward()
            opt_d.step()

            # Generator step

            opt_g.zero_grad()
            d_fake2 = disc(fake)
            loss_g = criterion(d_fake2, torch.ones_like(d_fake2))
            loss_g.backward()
            opt_g.step()

```

```

        # Log discriminator loss
        writer.add_scalar("Loss/Discriminator", loss_d.item(), step)
        # Log generator loss
        writer.add_scalar("Loss/Generator", loss_g.item(), step)
        step += 1
        # Update progress bar with losses
        loop.set_postfix(d_loss=loss_d.item(), g_loss=loss_g.item())

    if ep in config.SAVE_EPOCHS:
        # Save generated samples at specific epochs
        save_samples(gen, ep, fixed_noise)

    # Ensure output directory exists
    os.makedirs(config.OUTPUT_DIR, exist_ok=True)
    # Save generator model
    torch.save(gen.state_dict(), os.path.join(config.OUTPUT_DIR, "generator.pth"))
    # Save discriminator model
    torch.save(disc.state_dict(), os.path.join(config.OUTPUT_DIR, "discriminator.pth"))
    # Close TensorBoard writer
    writer.close()

if __name__ == "__main__":
    # Run training
    train()

```

e) generate.py

Listing 6: Generation

```

import torch
from models import Generator
from utils import save_samples
import config

def generate(model_path=None, n=64):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    G = Generator().to(device)
    mp = model_path or f"{config.OUTPUT_DIR}/generator.pth"
    G.load_state_dict(torch.load(mp, map_location=device))
    # Generate random noise
    noise = torch.randn(n, config.LATENT_DIM, device=device)
    # Generate and save samples
    save_samples(G, "final", noise)

if __name__ == "__main__":
    # Run the generate function if script is executed
    generate()

```