Implementation Report

EBGAN (Energy-Based Generative Adversarial Network)

Final Project

Katayoun Kebraei – Mehrdad Baradaran
Professor Kheradpisheh

In this project, the goal is to implement an Generative Adversarial Network based on energy-based functions. Instead of using a discriminator, an energy function is employed. In recent years, energy-based models have gained increasing attention due to improvements in training methods. The dataset used in this project for generating new images is the MNIST dataset.

To implement this model, we first need to understand how to use the energy function within the model. The aim is to estimate the probability distribution over the entire data space, given a large dataset. This means we aim to model a probability distribution over all possible images where the images generated have high likelihood and look realistic. A challenge with simple methods like interpolation is that they often fail due to the high dimensionality, especially in high-definition images.

However, how can we predict the probability distribution p(x) over such a large number of dimensions using a simple neural network? This distribution must have two key properties:

1. The probability distribution should assign a non-negative value to any possible input:

   $$p(x) \geq 0$$

2. The probability density must sum/integrate to 1 over all possible inputs:

   $$\int_x p(x)dx = 1$$

Many functions can satisfy these conditions, and the energy function is one of them. The core idea behind energy-based models is that any function predicting non-negative values can be transformed into a probability distribution by normalizing it over the entire volume. Suppose we have a neural network with a single output neuron, similar to a regression model. We can call this network f_θ, where θ represents the network parameters and x is the input data (for example, an image). The output $f_\theta(x)$ will be a scalar value between negative and positive infinity. We can now normalize all possible inputs using basic probability theory:
:

$$q_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta} \quad \text{where} \quad Z_\theta = \begin{cases} \int_{\mathbf{x}} \exp(-E_\theta(\mathbf{x}))d\mathbf{x} & \text{if } x \text{ is continuous} \\ \sum_{\mathbf{x}} \exp(-E_\theta(\mathbf{x})) & \text{if } x \text{ is discrete} \end{cases}$$

The exponential function ensures that a probability greater than zero is assigned to every possible input. We use a negative sign in front of E because we refer to E_θ as the energy function: data points with high probability have low energy, while data points with low probability have high energy. Z_θ is our normalization term that ensures the integral of the density sums to 1. We can demonstrate this by integrating over $q_\theta(x)$:

$$\int_{\mathbf{x}} q_\theta(\mathbf{x})d\mathbf{x} = \int_{\mathbf{x}} \frac{\exp(-E_\theta(\mathbf{x}))}{\int_{\tilde{\mathbf{x}}} \exp(-E_\theta(\tilde{\mathbf{x}}))d\tilde{\mathbf{x}}} d\mathbf{x} = \frac{\int_{\mathbf{x}} \exp(-E_\theta(\mathbf{x}))d\mathbf{x}}{\int_{\tilde{\mathbf{x}}} \exp(-E_\theta(\tilde{\mathbf{x}}))d\tilde{\mathbf{x}}} = 1$$

Now, $q_\theta(x)$ represents the probability distribution learned by our model, which has been trained to approximate the unknown distribution p(x) as closely as possible. The main advantage of this formulation is its flexibility, as we can define E_θ it in any way we choose.

However, when we look at the above equation, we can see a fundamental issue: how do we compute Z_θ ? There's no way we can analytically compute Z_θ for high-dimensional inputs or larger neural networks, as the process requires knowing Z_θ. Although we cannot determine the exact likelihood for a given point, some methods allow us to train energy-based models. Therefore, we will now look at contrastive divergence for training the model.

When we train a model for generative modeling, we usually do so by estimating the maximum likelihood. In other words, we aim to maximize the likelihood of the samples in the training set. Since the unknown normalization constant Z_θ prevents us from determining the exact likelihood of a point, we need to train energy-based models somewhat differently.

We cannot simply maximize the normalized probability, as there's no guarantee that Z_θ will remain constant or that XTrain will become more likely compared to others. However, if we base our training on the comparison of scores, we can create a stable objective. That is, we can rewrite our maximum comparison objective to maximize the likelihood of XTrain in relation to a randomly sampled point from our model:

$$\begin{aligned}\nabla_\theta \mathcal{L}_{\text{MLE}}(\theta; p) &= -\mathbb{E}_{p(\mathbf{x})}\left[\nabla_\theta \log q_\theta(\mathbf{x})\right] \\ &= \mathbb{E}_{p(\mathbf{x})}\left[\nabla_\theta E_\theta(\mathbf{x})\right] - \mathbb{E}_{q_\theta(\mathbf{x})}\left[\nabla_\theta E_\theta(\mathbf{x})\right]\end{aligned}$$

Note that minimizing the loss will still be our objective. Therefore, we aim to minimize the energy for data points from the dataset while maximizing the energy for randomly sampled data points from our model. Although this goal seems intuitive, how is it derived from our original distribution? The trick is to approximate Z_θ with a Monte Carlo sample. This gives us exactly our objective.

To sample from an energy-based model, we can apply the Markov Chain Monte Carlo using Langevin dynamics. The idea of the algorithm is to start from a random point and gently move towards higher probability directions using the gradients of E_θ. However, this alone is not sufficient to fully capture the probability distribution. We need to add noise ω to the current sample at each gradient step. Under certain conditions, such as performing gradient steps infinitely many times, we can create an accurate

sample from our modeled distribution. However, since this is practically infeasible, we usually limit the chain to k steps. In general, the sampling method can be summarized in the following algorithm:

Algorithm 1 Sampling from an energy-based model

1: Sample $\tilde{\boldsymbol{x}}^0$ from a Gaussian or uniform distribution;
2: **for** sample step $k = 1$ to K **do** ▷ Generate sample via Langevin dynamics
3: $\quad \tilde{\boldsymbol{x}}^k \leftarrow \tilde{\boldsymbol{x}}^{k-1} - \eta \nabla_{\boldsymbol{x}} E_\theta(\tilde{\boldsymbol{x}}^{k-1}) + \omega$, where $\omega \sim \mathcal{N}(0, \sigma)$
4: **end for**
5: $\boldsymbol{x}_{\text{sample}} \leftarrow \tilde{\boldsymbol{x}}^K$

Now that we are familiar with the energy-based problem, we will explain the code and the process of its formation, followed by examples of using this model for the dataset.

After importing the necessary libraries, we add the required data from the default PyTorch libraries and then pass it to the data loader to perform the desired classifications and settings. Meanwhile, we normalize the data to be between -1 and 1 to make the implementation of this code easier and to work more comfortably with the data. We specify the batch size so that a meaningful number of data points is taken from our entire dataset each time. We enable shuffling to ensure that the data is selected randomly.

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

train_data = MNIST(root=DATASET_PATH, train=True, transform=transform, download=True)

test_data = MNIST(root=DATASET_PATH, train=False, transform=transform, download=True)

train_dataloader = data.DataLoader(train_data, batch_size=64, shuffle=True, num_workers=4)
test_dataloader = data.DataLoader(test_data, batch_size=64, shuffle=False, num_workers=4)
```

After preparing the class, we build the architecture of our model. Since the dataset images are 28x28 pixels, there's no need to implement a deep model. We use a few convolutional layers with a stride of 2. Using a smooth activation function like Swish instead of ReLU is good practice for energy-based models. This is because we rely on gradients with respect to the input image, which should not be sparse.

```python
class Model(nn.Module):
    def __init__(self, hidden_features=32, out_dim=1, **kwargs):
        super().__init__()
        hidden_features1 = hidden_features // 2
        hidden_features2 = hidden_features
        hidden_features3 = hidden_features * 2

        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, hidden_features1, kernel_size=5, stride=2, padding=4),
            nn.SiLU(),
            nn.Conv2d(hidden_features1, hidden_features2, kernel_size=3, stride=2, padding=1),
            nn.SiLU(),
            nn.Conv2d(hidden_features2, hidden_features3, kernel_size=3, stride=2, padding=1),
            nn.SiLU(),
            nn.Conv2d(hidden_features3, hidden_features3, kernel_size=3, stride=2, padding=1),
            nn.SiLU(),
            nn.Flatten(),
            nn.Linear(hidden_features3 * 4, hidden_features3),
            nn.SiLU(),
            nn.Linear(hidden_features3, out_dim),
        )

    def forward(self, x):
        x = self.cnn_layers(x).squeeze(dim=-1)
        return x
```

In the next section, we will focus on training with sample elements. To utilize contrastive divergence objectives, we need to generate samples during training. Previous work has shown that due to the high dimensions of images, obtaining reasonable samples requires many iterations within the MCMC sampling. However, there is a training trick that significantly reduces sampling loss: using a sampling buffer. The idea is to store the last two batches of samples in a buffer and use them again as a starting point for the MCMC algorithm for the next batches. This reduces sampling loss because the model requires far fewer steps to converge to reasonable samples. However, to avoid relying solely on previous samples and to allow for new samples, we restart 5% of our samples from scratch (random noise between -1 and 1).

To do this, we create our sampler class. To define it, we provide the model with the desired input image size, batch size, and maximum number of data points to be stored in the buffer.

```python
class Sampler:
    def __init__(self, model, img_shape, sample_size, max_len=8192):
        super().__init__()
        self.model = model
        self.img_shape = img_shape
        self.sample_size = sample_size
        self.max_len = max_len
        self.examples = [(torch.rand((1,) + img_shape) * 2 - 1) for _ in range(self.sample_size)]
```

Here's the translation:

Next, we implement a function to obtain batches of fake images, where the inputs include the number of iterations to be applied in the MCMC algorithm, followed by the learning rate in the above algorithm.

```
def sample_new_exmps(self, steps=60, step_size=10):
    # new batch of "fake" images
    n_new = np.random.binomial(self.sample_size, 0.05)
    rand_imgs = torch.rand((n_new,) + self.img_shape) * 2 - 1
    old_imgs = torch.cat(random.choices(self.examples, k=self.sample_size - n_new), dim=0)
    inp_imgs = torch.cat([rand_imgs, old_imgs], dim=0).detach()

    # MCMC sampling
    inp_imgs = Sampler.generate_samples(self.model, inp_imgs, steps=steps, step_size=step_size)

    # Add new images to the buffer and remove old ones if needed
    self.examples = list(inp_imgs.chunk(self.sample_size, dim=0)) + self.examples
    self.examples = self.examples[: self.max_len]
    return inp_imgs
```

```
    @staticmethod
    def generate_samples(model, inp_imgs, steps=60, step_size=10, return_img_per_step=False):
        is_training = model.training
        model.eval()
        for p in model.parameters():
            p.requires_grad = False
        inp_imgs.requires_grad = True


        had_gradients_enabled = torch.is_grad_enabled()
        torch.set_grad_enabled(True)

        noise = torch.randn(inp_imgs.shape)

        imgs_per_step = []

        for _ in range(steps):
            noise.normal_(0, 0.005)
            inp_imgs.data.add_(noise.data)
            inp_imgs.data.clamp_(min=-1.0, max=1.0)

            out_imgs = -model(inp_imgs)
            out_imgs.sum().backward()
            inp_imgs.grad.data.clamp_(-0.03, 0.03)
            inp_imgs.data.add_(-step_size * inp_imgs.grad.data)
            inp_imgs.grad.detach_()
            inp_imgs.grad.zero_()
            inp_imgs.data.clamp_(min=-1.0, max=1.0)
```

```
            if return_img_per_step:
                imgs_per_step.append(inp_imgs.clone().detach())


        for p in model.parameters():
            p.requires_grad = True
        model.train(is_training)

        torch.set_grad_enabled(had_gradients_enabled)

        if return_img_per_step:
            return torch.stack(imgs_per_step, dim=0)
        else:
            return inp_imgs
```

The buffer idea becomes a bit clearer in the following algorithm.

With the sampling buffer ready, we can complete our training algorithm. Below is a summary of the full training algorithm for an energy-based model in image modeling.

Algorithm 2 Training an energy-based model for generative image modeling

Initialize empty buffer $B \leftarrow \emptyset$
while not converged **do**
  Sample data from dataset: $\boldsymbol{x}_i^+ \sim p_{\mathcal{D}}$
  Sample initial fake data: $\boldsymbol{x}_i^0 \sim B$ with 95% probability, else $\mathcal{U}(-1, 1)$
  for sample step $k = 1$ to K **do** ▷ Generate sample via Langevin dynamics
    $\tilde{\boldsymbol{x}}^k \leftarrow \tilde{\boldsymbol{x}}^{k-1} - \eta \nabla_{\boldsymbol{x}} E_\theta(\tilde{\boldsymbol{x}}^{k-1}) + \omega$, where $\omega \sim \mathcal{N}(0, \sigma)$
  end for
  $\boldsymbol{x}^- \leftarrow \Omega(\tilde{\boldsymbol{x}}^K)$ ▷ Ω: Stop gradients operator
  Contrastive divergence: $\mathcal{L}_{CD} = 1/N \sum_i E_\theta(\boldsymbol{x}_i^+) - E_\theta(\boldsymbol{x}_i^-)$
  Regularization loss: $\mathcal{L}_{RG} = 1/N \sum_i E_\theta(\boldsymbol{x}_i^+)^2 + E_\theta(\boldsymbol{x}_i^-)^2$
  Perform SGD/Adam on $\nabla_\theta(\mathcal{L}_{CD} + \alpha \mathcal{L}_{RG})$
  Add samples to buffer: $B \leftarrow B \cup \boldsymbol{x}^-$
end while

The first steps in each training iteration involve sampling from real and fake data, as we discussed earlier with the sampling buffer. Then, we calculate the **contrastive divergence** objective using the energy-based model. However, an additional training trick we need is to add a regularization drop to the output E_θ. Since the network output is not bounded, and adding or omitting a large bias to the output doesn't change the **regularization loss**, we need to ensure that the output values remain within a reasonable range in another way. Without regularization, the values of **regularization loss** will fluctuate over a very wide range. By doing this, we ensure that the values of real data hover around 0, while fake data values are likely to be slightly lower.

Since the **regularization loss** is less important compared to **contrastive divergence**, we use a weighting factor α, which is typically slightly less than 1. Finally, we perform an update step with an optimizer on the combined losses and add new samples to the buffer.

```
class DeepEnergyModel(pl.LightningModule):
    def __init__(self, img_shape, batch_size, alpha=0.1, lr=1e-4, beta1=0.0, **CNN_args):
        super().__init__()
        self.save_hyperparameters()
        self.cnn = Model(**CNN_args)
        self.sampler = Sampler(self.cnn, img_shape=img_shape, sample_size=batch_size)
        self.example_input_array = torch.zeros(1, *img_shape)

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

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr, betas=(self.hparams.beta1, 0.999))
        scheduler = optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.97)
        return [optimizer], [scheduler]


    def training_step(self, batch, batch_idx):
        real_imgs, _ = batch
        small_noise = torch.randn_like(real_imgs) * 0.005
        real_imgs.add_(small_noise).clamp_(min=-1.0, max=1.0)

        fake_imgs = self.sampler.sample_new_exmps(steps=60, step_size=10)

        inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
        real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)

        reg_loss = self.hparams.alpha * (real_out ** 2 + fake_out ** 2).mean()
        cdiv_loss = fake_out.mean() - real_out.mean()
```

We add minimal noise to the original images to prevent the model from focusing exclusively on completely "clean" inputs. For validation, we need to compute the contrastive divergence for completely random and unseen samples.

```
        cdiv_loss = fake_out.mean() - real_out.mean()
        loss = reg_loss + cdiv_loss

        # Logging
        self.log("loss", loss)
        self.log("loss_regularization", reg_loss)
        self.log("loss_contrastive_divergence", cdiv_loss)
        self.log("metrics_avg_real", real_out.mean())
        self.log("metrics_avg_fake", fake_out.mean())
        return loss

    def validation_step(self, batch, batch_idx):
        real_imgs, _ = batch
        fake_imgs = torch.rand_like(real_imgs) * 2 - 1

        inp_imgs = torch.cat([real_imgs, fake_imgs], dim=0)
        real_out, fake_out = self.cnn(inp_imgs).chunk(2, dim=0)

        cdiv = fake_out.mean() - real_out.mean()
        self.log("val_contrastive_divergence", cdiv)
        self.log("val_fake_out", fake_out.mean())
        self.log("val_real_out", real_out.mean())
```

We do not perform a test phase, as energy-based generative models are generally not evaluated on a test set.

Once the process is complete, we start training the model.

```
def train_model(**kwargs):
    trainer = pl.Trainer(
        default_root_dir=os.path.join(CHECKPOINT_PATH, "MNIST"),
        max_epochs=3,
        gradient_clip_val=0.1
    )

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "MNIST.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = DeepEnergyModel.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)
        model = DeepEnergyModel(**kwargs)
        trainer.fit(model, train_dataloader, test_dataloader)
        model = DeepEnergyModel.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)
    return model
```

```
model = train_model(img_shape=(1, 28, 28), batch_size=train_dataloader.batch_size, lr=1e-4, beta1=0.0)
```

Now, we begin generating images:

To monitor the model's performance during training, we will extensively use PyTorch Lightning's callback framework. Remember that callbacks can be utilized to execute small tasks at any point during training, for example, after completing an epoch. Here, we will use a callback for the generator. This generator callback is designed to add the generation of new images to the model during training. After each epoch, we take a small batch of random images and perform several MCMC (Markov Chain Monte Carlo) iterations until the model's generation converges.

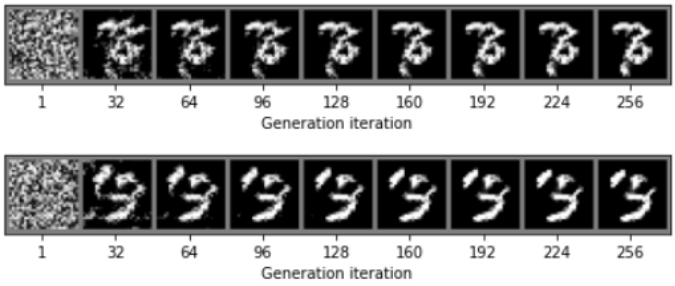
```
class GenerateCallback(pl.Callback):
    def __init__(self, batch_size=8, vis_steps=8, num_steps=256, every_n_epochs=5):
        super().__init__()
        self.batch_size = batch_size
        self.vis_steps = vis_steps
        self.num_steps = num_steps
        self.every_n_epochs = every_n_epochs

    def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch % self.every_n_epochs == 0:
            imgs_per_step = self.generate_imgs(pl_module)
            for i in range(imgs_per_step.shape[1]):
                step_size = self.num_steps // self.vis_steps
                imgs_to_plot = imgs_per_step[step_size - 1 :: step_size, i]
                grid = torchvision.utils.make_grid(
                    imgs_to_plot, nrow=imgs_to_plot.shape[0], normalize=True, range=(-1, 1)
                )
                trainer.logger.experiment.add_image("generation_%i" % i, grid, global_step=trainer.current_epoch)

    def generate_imgs(self, pl_module):
        pl_module.eval()
        start_imgs = torch.rand((self.batch_size,) + pl_module.hparams["img_shape"]).to(pl_module.device)
        start_imgs = start_imgs * 2 - 1
        imgs_per_step = Sampler.generate_samples(
            pl_module.cnn, start_imgs, steps=self.num_steps, step_size=10, return_img_per_step=True
        )
        pl_module.train()
        return imgs_per_step
```

Now we can check the samples.

If we reduce the number of training epochs to 3, the model will not have been properly trained yet, and the generated images won't be of good quality.

```
def train_model(**kwargs):
    trainer = pl.Trainer(
        default_root_dir=os.path.join(CHECKPOINT_PATH, "MNIST"),
        max_epochs=3,
        gradient_clip_val=0.1
    )
```

```
generate_images(imgs_per_step, callback)
```

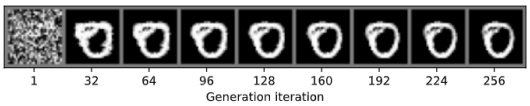


However, if we allow the model to train properly over a longer period, such as setting the number of epochs to 70, we will see that the generated images are much closer to the target, and high-quality results can be achieved.

```
def train_model(**kwargs):
    trainer = pl.Trainer(
        default_root_dir=os.path.join(CHECKPOINT_PATH, "MNIST"),
        gpus=1 if str(device).startswith("cuda") else 0,
        max_epochs=60,
        gradient_clip_val=0.1
    )
```

[92]:
```
generate_images(imgs_per_step, callback)
```

```
/opt/conda/lib/python3.7/site-packages/torchvision/utils.py:64: UserWarning: The parameter 'range' is deprecated since 0.12 and will be removed in 0.14. Please use 'value_range' instead.
  "The parameter 'range' is deprecated since 0.12 and will be removed in 0.14. "
```




```
/opt/conda/lib/python3.7/site-packages/torchvision/utils.py:64: UserWarning: The parameter 'range' is deprecated since 0.12 and will be removed in 0.14. Please use 'value_range' instead.
  "The parameter 'range' is deprecated since 0.12 and will be removed in 0.14. "
```