گزارش پیاده سازی

EBGAN

پروژه نهایی

توسط : کتایون کبرائی – مهرداد برادران

استاد مربوطه : استاد خردپیشه

در این پروژه تلاش بر پیاده سازی Generative aderserial network بر پایه فانکشن های Energy-based میباشد. در اصل به جای discreaminator از تابع انرژی استفاده میشود. در سال‌های اخیر، مدل‌های مبتنی بر انرژی به دلیل بهبود روش‌های ترینینگ، توجه روزافزونی را به خود جلب کرده‌اند. دیتایی که در این پروژه روی ان کار میشود و از ان برای تولید تصاویر جدید استفاده میشد دیتاستMNIST است.

برای اینکه بتوانیم این مدل را پیاده سازی کنیم ابتدا نیاز است با نحوه استفاده از تابع انرژی در این مدل اشنا شویم. هدف این مدل این است با توجه به مجموعه داده ای با عناصر زیاد، توزیع احتمال را در کل فضای داده را تخمین بزنیم. یعنی میخواهیم یک توزیع احتمال بر روی تمام تصاویر ممکن داشته باشیم در جایی که این تصاویر دارای likelihood بالایی هستند و واقعی به نظر می رسند. مسئله دیگر این است متد های ساده ای مثل درونیابی هم روی این مسائل کار نمیکنند چون این مسائل مخصوصا در تصاویر اچ دی دیمنشن های بالایی دارند.

با این حال، چگونه با استفاده از یک شبکه عصبی ساده، توزیع احتمال p(x) را بر روی این همه ابعاد پیش بینی کنیم؟ این توضیح واضحا باید دو ویژگی داشته باشد:

1- توزیع احتمال باید هر مقدار احتمالی x را یک مقدار غیر منفی نسبت دهد.

$$p(x) \geq 0$$

2- چگالی احتمال باید در تمام ورودی های ممکن به 1 جمع/انتگرال گرفته شود.

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

موارد زیادی وجود دارد که شامل این دو مورد شود و تابع انرژی هم یکی از انهاست. ایده اساسی مدل های مبتنی بر انرژی این است که می توان هر تابعی را که مقادیر بزرگتر از صفر را پیش بینی می کند، با تقسیم بر حجم آن به توزیع احتمال تبدیل کرد. فرض کنید یک شبکه عصبی با خروجی با یم نورون داریم مانند رگرسیون. می‌توانیم این شبکه را $E_\theta(x)$ نامیم، جایی که θ پارامترهای شبکه ما هستند و X داده‌های ورودی (مثلاً یک تصویر). خروجی E_θ یک مقدار اسکالر بین منفی بینهایت و مثبت بینهایت خواهد شد. اکنون می‌توانیم از نظریه احتمال اولیه برای نرمال‌سازی همه ورودی‌های ممکن استفاده کنیم:

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

تابع اکسپونشسال تضمین می کند که الزاما احتمالی بزرگتر از صفر را به هر ورودی ممکن نسبت می دهیم. ما جلوی E از علامت منفی استفاده می کنیم زیرا E_θرا میخواهیم تابع انرژی می نامیم: نقاط داده با احتمال زیاد انرژی کم دارند، در حالی که نقاط داده با احتمال کم انرژی بالایی دارندZ_θ عبارت‌های نرمال‌سازی ما است که تضمین می‌کند که انتگرال چگالی 1 میشود. ما می‌توانیم این را با ادغام روی$q_\theta(x)$ نشان دهیم:

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

حال $q_\theta(x)$ همان توزیع احتمال اموخته شده توسط مدل ماست و اموزش داده شده تا در حد امکان به توزیع ناشناخته p(x) نزدیک باشد. مزیت اصلی این فرمول انعطاف پذیری زیاد ان خواهد شد زیرا که ما E_θ را میتوانیم هرچه خواستیم قرار دهیم.

با این وجود، وقتی به معادله بالا نگاه می کنیم، می توانیم یک مسئله اساسی را ببینیم: چگونه Z_θ را محاسبه کنیم؟ هیچ شانسی وجود ندارد که بتوانیم Z_θ را به صورت تحلیلی برای ورودی های با بعد های بالا و یا شبکه های عصبی بزرگتر محاسبه کنیم، در حالی که فرایند مستلزم دانستن Z_θاست. اگرچه نمی‌توانیمlikelihood دقیق یک نقطه را تعیین کنیم، روش‌هایی وجود دارد که با آن‌ها می‌توانیم مدل‌های مبتنی بر انرژی را آموزش دهیم. بنابراین، در ادامه بهContrastive Divergence برای آموزش مدل نگاه خواهیم کرد.

وقتی مدلی را در مورد مدل سازی Generative ترین میکنیم، معمولاً با تخمین حداکثر likelihood انجام می شود. به عبارت دیگر، سعی می کنیم likelihood نمونه های موجود در ترین ست را به حداکثر برسانیم. از آنجایی که به دلیل ثابت عادی سازی ناشناخته Z_θ نمی توان likelihood دقیق یک نقطه را تعیین کرد، باید مدل های مبتنی بر انرژی را کمی متفاوت آموزش دهیم.

ما نمی‌توانیم فقط $\exp(-E\theta(xtrain))$ un-normalized probability را به حداکثر برسانیم، زیرا هیچ تضمینی وجود ندارد که Z_θ ثابت بماند یا اینکه Xtrain نسبت به بقیه محتمل‌تر شود. با این حال، اگر آموزش خود را بر اساس مقایسه likelihood امتیازها قرار دهیم، می‌توانیم یک هدف پایدار ایجاد کنیم. یعنی، می‌توانیم هدف حداکثر likelihood خود را در جایی که احتمال Xtrain را در مقایسه با یک نقطه داده نمونه‌گیری تصادفی مدل خود به حداکثر می‌رسانیم، دوباره بنویسیم:

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

توجه داشته باشید که همچنان به حداقل رساندن لاس هدف ما خواهد بود. بنابراین، ما سعی می کنیم انرژی را برای نقاط داده از مجموعه داده به حداقل برسانیم، در حالی که انرژی را برای نقاط داده نمونه برداری تصادفی از مدل خود به حداکثر برسانیم . اگرچه این هدف شهودی به نظر می رسد، اما چگونه از توزیع اصلی ما مشتق شده است؟ ترفند این است که Z_θرا با یک نمونه مونت کارلو تقریب می کنیم. این دقیقاً هدف مان را به ما می دهد.

برای نمونه‌برداری از یک مدل مبتنی بر انرژی، می‌توانیم زنجیره مارکوف مونت کارلو را با استفاده از Langevin Dynamics اعمال کنیم. ایده الگوریتم این است که از یک نقطه تصادفی شروع شود و با استفاده از گرادیان های E_θ به آرامی به سمت جهت بالاتر حرکت کند. با این وجود، این برای به دست آوردن کامل توزیع احتمال کافی نیست. باید نویز ω را در هر مرحله گرادیان به نمونه فعلی اضافه کنیم. تحت شرایط خاصی مانند اینکه ما مراحل گرادیان را بی نهایت بار انجام می دهیم، می توانیم یک نمونه دقیق از توزیع مدل شده خود ایجاد کنیم. با این حال، از آنجایی که این عملاً امکان پذیر نیست، ما معمولاً زنجیره را k مراحل محدود می کنیم . به طور کلی، روش نمونه گیری را می توان در الگوریتم زیر خلاصه کرد:

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: $\tilde{\boldsymbol{x}}^k \leftarrow \tilde{\boldsymbol{x}}^{k-1} - \eta \nabla_{\boldsymbol{x}} E_\theta(\tilde{\boldsymbol{x}}^{k-1}) + \omega$, where $\omega \sim \mathcal{N}(0, \sigma)$
4: **end for**
5: $\boldsymbol{x}_{\text{sample}} \leftarrow \tilde{\boldsymbol{x}}^K$

حال که با مسئله Energy-based اشنا شدیم، به توضیح کد و فرایند شکل گیری ان میپردازیم تا در ادامه ان مثال هایی از استفاده این مدل برای این دیتاست ببینیم.

پس از ایمپورت کردن کتابخانه های مورد نیاز کد، دیتا مورد نیاز را از کتابخانه های پیش فرض پایتورچ اضافه میکنیم و سپس ان را به دیتا لودر میدهیم تا دسته بندی ها و تنظیمات مورد نظر روی ان انجام شود. در این حین دیتای مورد نظرر را بین 1- و 1 نرمالایز میکنیم تا پیاده سازی این کد اسان تر شود و با دیتا راحت تر بتوانیم کار کنیم. بچ سایز را مشخص میکنیم تا هر بار تعداد معنی از دیتای کلی ما برداشته شود. شافل را میگذاریم تا دیتا به صورت رندوم انتخاب شود:

```
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)
```

بعد از اماده سازی کلاس معماری مدل مان را میسازیم. از انجایی که شکل تصاویر این دیتاست 28 * 28 میباشد نیاز به پیاده سازی مدلی عمیق نداریم و تنها از چند لایه کانولوشنی با گام های 2 تایی استفاده میکنیم. استفاده از یک تابع فعال سازی صاف مانند سوییش به جای رلو در مدل انرژی، تمرین خوبی است. این به این دلیل است که ما با توجه به تصویر ورودی، که نباید پراکنده باشد، به گرادیان هایی که برمی گردیم تکیه می کنیم.

```
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
```

در قسمت بعدی به آموزش با المان های نمونه می پردازیم. برای استفاده از اهداف Contrastive Divergence باید در طول آموزش نمونه هایی تولید کنیم. کارهای قبلی نشان داده است که به دلیل ابعاد بالای تصاویر، برای به دست آوردن نمونه های معقول، به تکرارهای زیادی در داخل نمونه برداری MCMC نیاز دارد. با این حال، یک ترفند آموزشی وجود دارد که لاس نمونه برداری را به میزان قابل توجهی کاهش می دهد: استفاده از بافر نمونه برداری. ایده این است که ما نمونه‌های دو دسته آخر را در یک بافر ذخیره می‌کنیم و دوباره از آن‌ها به عنوان نقطه شروع الگوریتم MCMCبرای دسته‌های بعدی استفاده می‌کنیم. این لاس نمونه برداری را کاهش می دهد زیرا مدل به تعداد مراحل بسیار کمتری برای همگرایی به نمونه های معقول نیاز دارد. با این حال، برای اینکه صرفاً به نمونه‌های قبلی تکیه نکنیم و نمونه‌های جدید را نیز مجاز بدانیم، 5 درصد از نمونه‌های خود را مجدداً از ابتدا شروع می‌کنیم (نویز تصادفی بین -1 و 1).
برای این کار کلاس سمپلر خود را میسازیم. برای تعریف ان مدل مورد نظر، سایز تصاویر ورودی، بچ سایز و ماکسیمم تعداد دیتاپوینتی که باید در بافر ذخیره شوند را به مدل میدهیم.

```
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)]
```

سپس در ان فانکشنی برای گرفتن بچ از تصاویر فیک پیاده سازی میکنیم که ورودی های ان یکی تعداد ایتریشن هاییست که روی الگوریتم MCMC زده میشود و بعدس لرنینگ ریت در الگوریتم بالا خواهد شد:

```
  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
```

```python
    @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)
```

```python
            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
```

ایده بافر در الگوریتم زیر کمی واضح تر می شود.

با آماده بودن بافر نمونه برداری، می توانیم الگوریتم ترینینگ خود را تکمیل کنیم. در زیر خلاصه ای از الگوریتم ترین کامل یک مدل انرژی در مدل سازی تصویر نشان داده شده است.

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

چند عبارت اول در هر تکرار ترینینگ مربوط به نمونه برداری از داده های واقعی و جعلی است، همانطور که در بالا با بافر نمونه دیدیم. سپس، هدف Contrastive Divergence را با استفاده از مدل انرژی بیسد محاسبه می کنیم. با این حال، یک ترفند ترینینگ اضافی که به آن نیاز داریم اضافه کردن یک افت منظم در خروجی ای تتا است. از آنجایی که خروجی شبکه محدود نیست و افزودن یک بایاس یا عدم وجود یک بایاس بزرگ به خروجی، regularization loss را تغییر نمی‌دهد، باید به نحوی دیگر اطمینان حاصل کنیم که مقادیر خروجی در محدوده معقولی قرار دارند. بدون regularization loss ، مقادیر خروجی در محدوده بسیار زیادی نوسان خواهند داشت. با این کار، اطمینان می‌دهیم که مقادیر داده‌های واقعی در حدود 0 است و داده‌های جعلی احتمالاً کمی پایین‌تر هستند. از آنجایی که از regularization loss اهمیت کمتری نسبت به Contrastive Divergence دارد، ما یک ضریب وزنی الفا داریم که معمولاً کمی کوچکتر از 1 است. در نهایت، ما یک مرحله به روز رسانی را با یک بهینه ساز در تلفات ترکیبی انجام می دهیم و نمونه های جدید را به بافر اضافه می کنیم.

```python
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()
```

ما حداقل نویز را به تصاویر اصلی اضافه می کنیم تا از تمرکز مدل بر روی ورودی های کاملاً "کلین" جلوگیری کنیم. برای اعتبارسنجی،باید Contrastive Divergenceتصاویر کاملا تصادفی و نمونه‌های دیده نشده را محاسبه کنیم.

```
        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())
```

ما یک مرحله تست را اجرا نمی کنیم زیرا مدل های جنریتور مبتنی بر انرژی معمولاً در یک تست ست ارزیابی نمی شوند.

پس از اتمام کار شروع به ترین مدل میکنیم:

```
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)
```

حال شروع به ساختن تصاویر میکنیم:

برای شناسایی عملکرد مدل در طول آموزش، از فریم وورک کال بک پایتورچ لایتنینگ به طور گسترده استفاده خواهیم کرد. به یاد داشته باشید که کال بک ها می توانند برای اجرای عملکردهای کوچک در هر نقطه از آموزش استفاده شوند، به عنوان مثال پس از اتمام یک دوره. در اینجا، از برای جنریتور استفاده خواهیم کرد. این کال بک جنریتور برای افزودن تولید تصاویر جدید به مدل در طول ترین استفاده می شود. پس از هر ان دوره ، دسته کوچکی از تصاویر تصادفی می‌گیریم و تکرارهای ام سی ام سی زیادی را انجام می‌دهیم تا زمانی که نسل مدل همگرا شود:

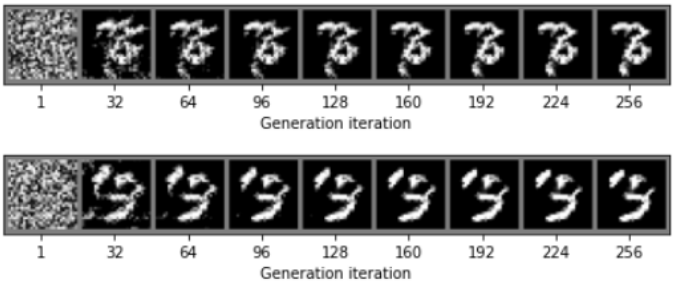
```
class GenerateCallback(pl.Callback):
    def __init__(self, batch_size=8, vis_steps=8, num_steps=256, every_n_epochs=5):
        super().__init__()
        self.batch_size = batch_size
        self.vis_steps = vis_steps
        self.num_steps = num_steps
        self.every_n_epochs = every_n_epochs

    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
```

حال میتوانیم نمونه هارا چک کنیم.

اگر تعداد ایپاک های ترین را روی 3 ایپاک بیاوریم مدل درست ترین نشده هنوز و تصویر خوبی تولید نمیشوند:

```
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)
```

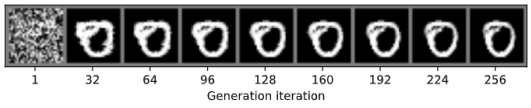


اما اگر اجازه بدهیم مدل به درستی و در زمان طولانی ترین شود یعنی مثلا تعدا ایپاک هارا 70 بگذاریم میبینیم که تصاویر بسیار نزدیک تولید شده و میتوان کیفیت بالایی دریافت کرد.

```
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. "
```