به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

مدل‌های مولد عمیق

تمرین شماره ۲

نام و نام خانوادگی	علیرضا حسینی
شماره دانشجویی	۸۱۰۱۰۱۱۴۲
تاریخ ارسال گزارش	۱۴۰۲/۰۸/۲

فهرست گزارش سوالات

سوال 1 – سوالات اثباتی ELBO و VAE

الف) فاصله KL بین $q(z|x)$ و $p(z)$

با توجه به توزیع های داده شده داریم :

$$q(z|x) = \frac{1}{(2\pi)^{\frac{k}{2}}|\ diag(\sigma(x)^2)|} \exp\left[-\frac{1}{2}\big(z-\mu(x)\big)^T diag(\sigma(x)^2)^{-1}(z-\mu(x))\right]$$

$$p(z) = \frac{1}{(2\pi)^{\frac{k}{2}}} \exp\left[-\frac{1}{2}z^T z\right]$$

همچنین میدانیم :

$$|\ diag(\sigma(x)^2)| = \prod_{i=1}^{k} \sigma_i^2(x)$$

حال طبق تعریف KL داریم :

$$D_{KL}\big(q(z|x)\ ||\ p(z)\big) = \int q(z|x) \log\left(\frac{q(z|x)}{p(z)}\right) dz$$

با توجه به روابط $q(z|x)$ و $p(z)$ عبارات 2π خط خورده و log تمامی exp ها را ساده میکند و بنابراین داریم :

$$D_{KL}\big(q(z|x)\ ||\ p(z)\big) = \int q(z|x)\ \left[-\frac{1}{2}\log\prod_{i=1}^{k}\sigma_i^2(x)\ -\frac{1}{2}\big(z-\mu(x)\big)^T diag(\sigma(x)^2)^{-1}\big(z-\mu(x)\big) + \frac{1}{2}z^T z\ \right]\mathrm{dz}$$

$$= \frac{1}{2} log \prod_{i=1}^{k}\sigma_i^2(x)\ \int q(z|x)\ dz\ + \frac{1}{2}\int q(z|x)\ \sum_{i=1}^{k}\left(\frac{\big(z_i-\mu_i(x)\big)^2}{\sigma_i^2(x)} - z_i^2\right) dz$$

همچنین داریم :

$$\begin{cases} E_{q(z|x)}[(z_i-\mu_i)^2] = \sigma_i^2(x) \\ E_{q(z|x)}[z_i^2] = \sigma_i^2(x) + \mu_i^2(x) \end{cases}$$

با توجه به روابط فوق داریم :

$$D_{KL}\big(q(z|x)\ |\quad |p(z)\big) = -\frac{1}{2}\sum_{i=1}^{k}\log\sigma_i^2(x)\ + \frac{1}{2}\sum_{i=}^{k}(1+\ \mu_i^2(x) - \ \sigma_i^2(x)\ -\ z_i^2\ )$$

$$D_{KL}\big(q(z|x)\ ||\ p(z)\big) = \frac{1}{2}\sum_{i=1}^{k}(\sigma_i^2(x) + \mu_i^2(x) - 1 - log\ \sigma_i^2(x))$$

حکم ثابت شد.

ب) IWAE

بنابر نامساوی Jenson داریم :

$$\begin{cases} E(f(x) \geq f\big(E(x)\big) \ ; if \ f: convex \\ E(f(x) \leq f\big(E(x)\big) \ ; if \ f: concave \end{cases}$$

با توجه به اینکه log یک تابع concave میباشد داریم :

$$\log(E\left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right) \geq E(\log\left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right))$$

برای طرف چپ نامساوی داریم :

$$\log\left(E\left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right)\right) \cong \frac{1}{m}\sum_{i=1}^{k}\frac{p_\theta\big(x,z^{(i)}\big)}{q_\varphi\big(z^{(i)}|x\big)} = \frac{1}{m}\sum_{i=1}^{k}\frac{p_\theta\big(z^{(i)}|x\big)p(x)}{q_\varphi\big(z^{(i)}|x\big)} = \frac{1}{m}\,m\,p_\theta(x) = \ p_\theta(x)$$

برای طرف راست نیز داریم :

$$E(\log\left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right)) \ = E(\log\frac{1}{m}\sum_{i=1}^{k}\frac{p_\theta(x,z)}{q_\varphi(z|x)})$$

بنابراین میتوان گفت که :

$$\mathrm{p}_\theta(x) \geq L_m(x\,;\,\theta\,,\varphi)$$

و یک کران معتبر برای log likelihood میباشد.

ج) Posterior Collapse در VAE

Posterior Collapse ، پدیده ای است که می تواند در VAE رخ دهد. Posterior Collapse به طور خاص به وضعیتی اشاره دارد که در آن مدل متغیرهای پنهان را نادیده می گیرد و برای تولید داده ها صرفاً به decoder متکی است و به طور مؤثرencoder و فضای پنهان (latent) را نادیده می گیرد.

دلیل Posterior Collapse را می توان به فرآیند بهینه سازی در حین آموزش نسبت داد. در یک VAE، دو جزء اصلی وجود دارد: رمزگذار، که داده های ورودی را به یک توزیع در فضای پنهان نگاشت می کند، و رمزگشا که داده ها را از نمونه های گرفته شده از این توزیع تولید می کند. این مدل با به حداکثر رساندن ELBO که شامل یک ترم بازسازی و یک ترم منظم‌سازی است، آموزش داده می‌شود.

اگر رمزگشا نسبت به رمزگذار بیش از حد قدرتمند شود، مدل ممکن است با نادیده گرفتن متغیرهای پنهان و تکیه بر رمزگشا، بازسازی داده های ورودی را آسان تر کند. این یک راه حل غیربهینه است زیرا به طور موثر ساختار زیربنایی داده ها را در فضای پنهان ثبت نمی کند.

برای مقابله با Posterior Collapse روش های متعددی وجود دارد:

- KL Annealing: به تدریج وزن عبارت واگرایی KL را در تابع loss در طول آموزش افزایش داده میشود. این به مدل اجازه می دهد تا در مراحل بعدی آموزش بیشتر بر روی متغیرهای پنهان تمرکز کند.
- Warm Up : میتوان با وزن کم در ترم واگرایی KL شروع کرد و با پیشرفت آموزش به تدریج آن را افزایش داد. این به مدل کمک می کند تا ابتدا بر بازسازی تمرکز کند و سپس به سمت استفاده از متغیرهای پنهان سوق یابد.
- حداکثر سازی اطلاعات: تابع هدف اصلاح میشود تا مدل را تشویق کند تا از اطلاعات موجود در متغیرهای پنهان استفاده کند. این می تواند شامل افزودن اصطلاحات اضافی به تابع لاس باشد که مدل را به دلیل استفاده نکردن از فضای پنهان جریمه می کند.
- تغییرات معماری: برای مثال، می‌توان ظرفیت رمزگذار را افزایش داد ، لایه‌های بیشتری اضافه کرد یا معماری را تغییر دهید تا نادیده گرفتن متغیرهای پنهان برای مدل دشوارتر شود.
- استفاده از تکنیک‌های منظم‌سازی: سایر تکنیک‌های منظم‌سازی مانند drop out یا batch normalization را برای جلوگیری از اتکای بیش از حد مدل به پارامترهای خاص میتوان اضافه کرد.
- Augmentation : تغییراتی را در داده‌های ورودی در طول آموزش وارد میکند تا مدل را مجبور به یادگیری نمایش قوی‌تری در فضای پنهان کند.

اغلب لازم است که ترکیبی از این تکنیک ها را برای یافتن بهترین رویکرد برای یک مشکل خاص آزمایش کرد. انتخاب هایپرپارامترها و جزئیات خاص معماری نیز می تواند نقش مهمی در کاهش Posterior Collapse داشته باشد.

مطالب بر اساس مقاله زیر بیان شده است.

ALIAS PARTH GOYAL, Anirudh Goyal, et al. "Z-forcing: Training stochastic recurrent networks." *Advances in neural information processing systems* 30 (2017).

https://arxiv.org/abs/1711.05411

د) اثبات ELBO به عنوان حد پایین likelihood

$$\log p_\theta(x) = \log \int_z p_\theta(x,z)dz = \log \int_z p_\theta(x,z) \left(\frac{q_\varphi(z|x)}{q_\varphi(z|x)}\right) dz$$

$$log\ p_\theta(x) = \log \int_z q_\varphi(z|x) \left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right) dz$$

با توجه به نامساوی Jenson و اینکه log یک تابع concave میباشد داریم :

$$log\ p_\theta(x) \geq \int_z q_\varphi(z|x)\ log \left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right) dz$$

برای سمت راست نامساوی داریم :

$$p_\theta(x,z) = p(x|z)p(z)$$

$$\int_z q_\varphi(z|x)\ log \left(\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right) dz = \int_z q_\varphi(z|x) \log\ [p(x|z) + \log\,(p(z) - \log q_\varphi(z|x)]\, dz$$

$$= \int_z q_\varphi(z|x) \log p(x|z)\, dz \ + \int_z q_\varphi(z|x) \log \left(\frac{p(z)}{q_\varphi(z|x)}\right) dz$$

$$= E_{q_\varphi(Z|X)}[\log(p_\theta(x|z)] - D_{KL}\left(q_\varphi(z|x)||\, p(z)\right)$$

بنابراین ثابت شد که :

$$log\ p_\theta(x) \geq E_{q_\varphi(Z|X)}[\log(p_\theta(x|z)] - D_{KL}\left(q_\varphi(z|x)||\, p(z)\right)$$

سوال ۲ – پیاده سازی VAE

الف) کد مربوط به Reparameterization Trick

با توجه به توضیحات گفته شده و اینکه این تابع باید یک میانگین و واریانس بگیرد و یک سمپل را برگرداند میتوان به صورت زیر آن را تعریف کرد.

```
z = torch.distributions.normal.Normal(m,torch.sqrt(v)).rsample()
```

در عبارت فوق m و v را میگیرد و z را به عنوان یک sample برمیگرداند. که برای اینکار از توزیع نرمال موجود در تورچ استفاده شده است.

یک راه حل دیگر برای کد این قسمت استفاده از کد زیر میباشد که این هم درست است و در ضمن سرعت محاسباتی بیشتری هم دارد.

```
epsilon = torch.randn_like(v)
z = m + torch.sqrt(v) * epsilon
```

ب) کد مربوط به Negative ELBO Bound

برای کد این بخش از کد زیر استفاده شده است که توضیحات آن در ادامه آمده است.

```
def negative_elbo_bound(self, x):
    """
    Computes the Evidence Lower Bound, KL and, Reconstruction costs

    Args:
        x: tensor: (batch, dim): Observations

    Returns:
        nelbo: tensor: (): Negative evidence lower bound
        kl: tensor: (): ELBO KL divergence to prior
        rec: tensor: (): ELBO Reconstruction term
    """

    qm, qv = self.enc.encode(x)

    # Sample z from the approximate posterior
    z = sample_gaussian(qm, qv)

    # Decode z to get the reconstruction
    x_recon = self.dec.decode(z)
```

```
    # Compute the reconstruction loss
    rec = -1 * log_bernoulli_with_logits(x, x_recon)

    # Compute the KL divergence
    kl = kl_normal(qm, qv, self.z_prior_m, self.z_prior_v)

    # Compute the negative ELBO
    nelbo = kl + rec

    return nelbo.mean(), kl.mean(), rec.mean()
```

با توجه به صورت مساله و توجه به کد مراحل به صورت زیر میباشد.

ابتدا میانگین و واریانس از ENCODER گرفته میشود و در ادامه یک سمپل از آن ساخته شده و آن سمپل وارد بخش decoder میشود تا x_recon درست شود.

در ادامه loss ها تعریف میشود که یکی KL بین گوسی ساخته شده توسط encoder و sample تولیدی است و یکی هم مربوط به reconstruction

لازم به ذکر است که log_bernoulli_with_logits احتمال log منفی بازسازی را در مقایسه با ورودی اصلی محاسبه می کند. و برای واگرایی KL بین توزیع تقریبی پسین (qm و qv) و توزیع قبلی متغیر پنهان (self.z_prior_m و self.z_prior_v) تابع kl_normal واگرایی KL را برای دو توزیع گاوسی محاسبه می کند.

در نهایت nelbo نیز به صورت جمع ۲ معیار قبل تعریف شده و میانگین هر ۳ توسط تابع برگردانده میشود.

ج) فرایند یادگیری مدل

ابتدا با توجه به اینکه در کد VAE داده شده در dataloader دیتای تست لود نشده بود بخش load Data به صورت زیر تغییر داده شده است.

```
def get_mnist_data(device, batch_size, validation_split=0.1):
    preprocess = transforms.ToTensor()

    # Load the entire MNIST dataset
    full_dataset = datasets.MNIST('data', train=True, download=True,
transform=preprocess)
```

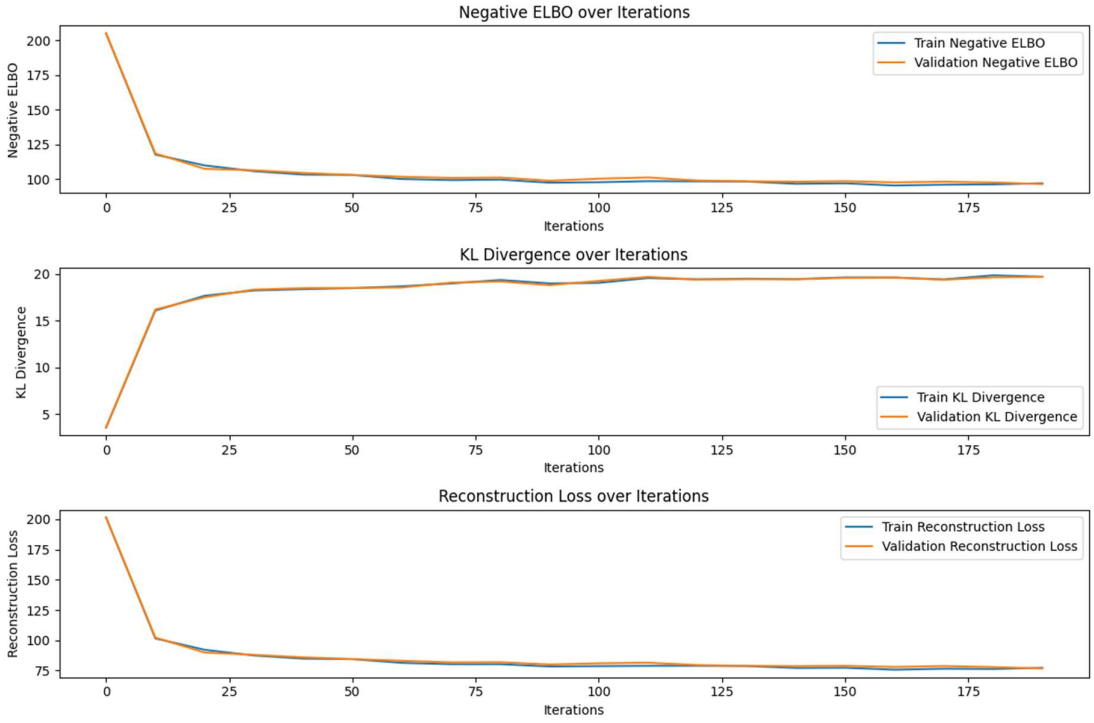
```
    # Calculate the number of validation samples based on the split
    num_train = len(full_dataset)
    num_val = int(validation_split * num_train)
    num_train = num_train - num_val

    # Split the dataset into training and validation sets
    train_dataset, val_dataset = torch.utils.data.random_split(
        full_dataset, [num_train, num_val])

    # Create data loaders for training and validation
    train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True)

    val_loader = torch.utils.data.DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=True)

    # Create pre-processed training and validation sets
    X_train =
train_loader.dataset.dataset.train_data.to(device).reshape(-1,
784).float() / 255
    y_train = train_loader.dataset.dataset.train_labels.to(device)
    X_val =
val_loader.dataset.dataset.train_data.to(device).reshape(-1,
784).float() / 255
    y_val = val_loader.dataset.dataset.train_labels.to(device)

    return train_loader, val_loader, (X_val, y_val)

train_loader, val_loader, _ = get_mnist_data(device, batch_size,
validation_split=0.1)
```

در ادامه همانطور که در کد زیر مشخص است ابتدا مدل و optimizer تعریف شده است و درادامه لیست هایی برای ذخیره سازی مقادیر loss ساخته شده و فرایند آموزش انجام میشود که کد loop Train در ادامه آمده است.

```
vae = VAE(z_dim=z).to(device)
optimizer = optim.Adam(vae.parameters(), lr=learning_rate)

# Lists to store the loss terms during training and validation
```

```
train_losses = []
train_kls = []
train_recs = []
val_losses = []
val_kls = []
val_recs = []

for i in tqdm(range(iter_max)):
    for batch_idx, (xu, yu) in enumerate(train_loader):

        optimizer.zero_grad()

        xu = torch.bernoulli(xu.to(device).reshape(xu.size(0), -1))
        yu = yu.new(np.eye(10)[yu]).to(device).float()
        loss, summaries = vae.loss(xu)

        loss.backward()
        optimizer.step()

    #print(summaries['train/loss'])

    # Report the loss terms across time for train and validation
datasets
    if i % 10 == 0:
        # Training dataset
        train_loss, train_kl, train_rec = vae.negative_elbo_bound(xu)
        train_losses.append(train_loss.item())
        train_kls.append(train_kl.item())
        train_recs.append(train_rec.item())

        # Validation dataset (assuming you have a validation loader
named 'val_loader')
        for batch_idx, (xv, yv) in enumerate(val_loader):
            xv = torch.bernoulli(xv.to(device).reshape(xv.size(0), -
1))
            yv = yv.new(np.eye(10)[yv]).to(device).float()
            val_loss, val_kl, val_rec = vae.negative_elbo_bound(xv)

        val_losses.append(val_loss.item())
        val_kls.append(val_kl.item())
        val_recs.append(val_rec.item())

print("Training Completed!")
```

منحنی تغییرات KL و NELBO وReconstruction Loss را در نهایت پلات کرده که خروجی به صورت زیر میباشد.



شکل 1 منحنی **Loss** برای داده های آموزش و ارزیابی در فرایند آموزش **VAE**

مشاهده میشود که مقادیرکاهشی یا افزایشی به خوبی انجام شده است و مقدار KL در حدود ۱۹ و Rec loss به حدود ۷۵ و NELBO به حدود۱۰۰ میرسد.

منحنی های loss در هر بار ران کردن ممکن است کمی متفاوت بشود ولی در نهایت به همین حدود اعداد بعد از ۲۰ ایپاک میرسند.

فرایند آموزش حدود ۲۰ دقیقه طول کشید.

د) پیاده سازی کد مربوط به IWAE

با توجه به محاسبات و فرمول داده شده در سوال ۱ بخش ب میتوان کد این بخش را به صورت زیر زد.

```
def negative_iwae_bound(self, x, iw):
    """
```

```
    Computes the Importance Weighted Autoencoder Bound
    Additionally, we also compute the ELBO KL and reconstruction
terms

    Args:
        x: tensor: (batch, dim): Observations
        iw: int: (): Number of importance weighted samples

    Returns:
        niwae: tensor: (): Negative IWAE bound
        kl: tensor: (): ELBO KL divergence to prior
        rec: tensor: (): ELBO Reconstruction term
    """


    qm, qv = self.enc.encode(x)
    qm = duplicate(qm, iw)
    qv = duplicate(qv, iw)
    x = duplicate(x, iw)

    z = sample_gaussian(qm,qv)

    probs = self.dec.decode(z)

    rec = -1 * log_bernoulli_with_logits( x,probs)
    kl = log_normal(z,qm,qv) -  log_normal(z, self.z_prior_m[0],
self.z_prior_v[0])
    nelbo = kl + rec

    niwae = -1 * log_mean_exp(-nelbo.reshape(iw,-1), dim=0)


  return niwae.mean(), kl.mean(), rec.mean()
```

محاسبات تقریبا مثل قبل است فقط در اینجا از log_mean_exp برای محاسبه niwae استفاده شده است.

ه) ارزیابی پیاده سازی IWAE

با توجه به توضیحات داده شده در صورت سوال کد را به صورت زیر میزنیم.

```
# Load the trained VAE model
vae = torch.load('vae_model.pth').to(device)

# Values of m to evaluate
```

```
m_values = [5, 50, 150]

# Store the results
results = {'m': [], 'niwae_avg': [], 'elbo_avg': []}

for m in m_values:
    niwae_vals = []
    elbo_vals = []

    for _ in tqdm(range(50)):  # 50 repetitions
        for xv, _ in val_loader:
            xv = torch.bernoulli(xv.to(device).reshape(xv.size(0), -
1))

            # Compute negative IWAE
            niwae, _, _ = vae.negative_iwae_bound(xv, m)
            niwae_vals.append(niwae.item())

            # Compute negative ELBO
            elbo, _, _ = vae.negative_elbo_bound(xv)
            elbo_vals.append(elbo.item())

    # Average the results
    results['m'].append(m)
    results['niwae_avg'].append(sum(niwae_vals) / len(niwae_vals))
    results['elbo_avg'].append(sum(elbo_vals) / len(elbo_vals))

# Report the results
print("Results:")
for m, niwae_avg, elbo_avg in zip(results['m'], results['niwae_avg'],
results['elbo_avg']):
    print(f"m = {m}: Average Negative IWAE = {niwae_avg}, Average
Negative ELBO = {elbo_avg}")
```

نتیجه نهایی به صورت زیر میباشد:

m = 5: Average Negative IWAE = 95.54060094197591, Average Negative ELBO = 97.33104011535644

m = 50: Average Negative IWAE = 94.50123512268067, Average Negative ELBO = 97.35990272521973

m = 150: Average Negative IWAE = 94.220611038208, Average Negative ELBO = 97.33820798238118

با توجه به نتایج فوق میتوان گفت که پیاده سازی درست بوده است و همچنین مشاهده شد که IWAE یک کران معتبر برای likelihood میباشد.

- برای بررسی کد ها و اطمینان از نتایج آموزش و مقادیر به دست آمده ، نتایج با گیت هاب زیر بررسی شده است.

https://github.com/kolchinski/cs236

سوال ۳ – مدل های Flow-Based

الف) توزیع احتمال متغیر X

با توجه به صورت سوال و روابط داده شده داریم :

$$X = 1.5\,Z + 3$$

$$X = f(Z) => Z = f^{-1}(X) = \frac{X-3}{1.5}$$

$$\frac{d\,f^{-1}(X)}{dx} = \frac{2}{3}$$

$$p(X) = P_Z\left(\frac{X-3}{1.5}\right)\frac{2}{3}$$

$$p(X) = N\left(\frac{X-3}{1.5} \mid 0,1\right)$$

$$p(X) = \frac{1}{\sqrt{2\pi}}\exp\left[\frac{4}{9}(X^2 + 9 - 6X)\right]$$

ب) آشنایی با مقاله RealNVP

به طور خلاصه برای مقاله داده شده داریم :

مقاله با عنوان «تخمین چگالی با استفاده از RealNVP » در ICLR 2017 منتشر شد و بر یادگیری بدون نظارت مدل‌های احتمالی، تمرکز دارد. نویسندگان، Laurent Dinh، Jascha Sohl-Dickstein، و Samy Bengio، تبدیل‌های (Real NVP) را معرفی می‌کنند که قدرتمند، پایدار، معکوس‌پذیر و قابل یادگیری هستند. این تبدیل‌ها امکان یک الگوریتم یادگیری بدون نظارت را با محاسبه دقیق log-liability، نمونه‌برداری کارآمد، استنتاج کارآمد متغیرهای پنهان و یک فضای پنهان قابل تفسیر را فراهم می‌کنند. توانایی مدل برای مدیریت تصاویر طبیعی در چهار مجموعه داده نشان داده شده است.

مدل Real NVP به چالش مدل‌سازی داده‌های ساختاریافته با ابعاد بالا می‌پردازد. این استنتاج، نمونه برداری و تخمین log-density نقاط داده کارآمد و دقیق را انجام می دهد. معماری بازسازی دقیق و کارآمد تصاویر ورودی از ویژگی های سلسله مراتبی استخراج شده توسط مدل را امکان پذیر می کند.

جنبه های کلیدی NVP واقعی عبارتند از:

- تغییر فرمول متغیر[1]: برای مدل‌سازی توزیع‌ها بر روی داده‌های پیوسته استفاده می‌شود، جایی که ژاکوبین تبدیل مثلثی است و تعیین‌کننده آن را به‌طور کارآمد قابل محاسبه می‌کند.
- لایه‌های کوپلینگ : این لایه‌ها برای ایجاد یک تابع bijective انعطاف‌پذیر و قابل انعطاف با کنار هم قرار دادن دنباله‌ای از bijective ساده استفاده می‌شوند. بخشی از بردار ورودی با استفاده از یک تابع به روز می شود که به روشی convolution به باقیمانده بردار ورودی بستگی دارد. این رویکرد به طور موثر تعیین کننده ژاکوبین تبدیل را محاسبه می کند.
- معماری Multiscale: این مدل یک معماری Multiscale را با استفاده از یک عملیات فشرده سازی پیاده سازی می کند که Spatial Size را با تعداد کانال ها مبادله می کند. این رویکرد شامل ترکیبی از لایه‌های کوپلینگ ، checkerboard masks و channel-wise masking است.
- Batch Normalization و Residual Networks : این شبکه ها در شبکه های عصبی کانولوشن عمیق برای توابع s (مقیاس) و t (translation) استفاده می شوند. Batch Normalization اصلاح شده است تا برای آموزش با مینی بچ های کوچک قوی تر باشد.

نویسندگان اثربخشی مدل را از طریق آزمایش بر روی مجموعه داده های تصویر طبیعی مانند CIFAR-10، Imagenet، LSUN و CelebA نشان می دهند. این مدل از نظر کیفیت نمونه و احتمال ورود به سیستم به عملکرد رقابتی دست می یابد. آزمایش‌ها توانایی مدل را برای تولید نمونه‌های متنوع، با فرآیند نمونه‌گیری کارآمد و موازی نشان می‌دهند. مدل Real NVP همچنین فضای پنهان را به طور معناداری سازماندهی می کند، و پتانسیل را برای وظایف یادگیری نیمه نظارت شده نشان می دهد.

در نتیجه، این مقاله Real NVP را به عنوان یک ابزار قدرتمند برای یادگیری بدون نظارت، پر کردن شکاف بین رویکردهای مدل‌سازی مختلف مانند مدل‌های رگرسیون خودکار، رمزگذارهای خودکار متغیر و شبکه‌های متخاصم ارائه می‌کند. معماری این مدل امکان ارزیابی، استنتاج و نمونه‌برداری دقیق و قابل اجرا را فراهم می‌کند و آن را به ابزاری همه کاره برای کاربردهای مختلف در یادگیری ماشین تبدیل می‌کند.

[1] Change of Variable Formula

ج) دترمینان ماتریس ژاکوبین لایه کوپلینگ

با توجه به روابط لایه داده شده داریم :

$$J = \begin{bmatrix} \frac{\partial y_a}{\partial x_a} & \frac{\partial y_a}{\partial x_b} \\ \frac{\partial y_b}{\partial x_a} & \frac{\partial y_b}{\partial x_b} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \exp(.) + \frac{\partial t_{x_a}}{\partial x_a} & \exp\left(s(x_a)\right) \end{bmatrix}$$

با توجه به پایین مثلثی بودن برای دترمینان نیاز به محاسبه $\frac{\partial y_b}{\partial x_a}$ نمیباشد. و دترمینان حاصل ضرب قطر اصلی میباشد که در اینجا میشود $\exp\left(s(x_a)\right)$ که خود $s(x_a)$ یک اسکالر میباشد.

د) مدل Real NVP روی مجموعه داده MNIST

جهت آموزش بر روی مجموعه داده MNIST به صورت زیر عمل میشود.

لازم به ذکر است تمامی کد ها و مدل ها بر اساس 2 گیت هاب زیر نوشته شده است و صرفا تغییرات عوض کردن dataloader حذف کردن بخش Mask Convolution و ذخیره کردن مقادیر loss و پلات کردن آن ها و گرفتن خروجی از مدل آموزش داده شده به آن ها اعمال شده است.

https://github.com/chrischute/real-nvp

https://github.com/bjlkeng/sandbox

ابتدا ماژول های مورد نیاز نوشته میشود

بخش residual block توجه به گیت هاب resnet[1] نوشته شده است که برای جلوگیری از نوشتن های زیاد میتوانید به کد آن که همراه گزارش ارسال شده است مراجعه فرمایید.

این مدل چندین مؤلفه کلیدی، از جمله شبکه‌های s و t (توابع مقیاس‌بندی وtranslation)، لایه‌های نرمال‌سازی و پارامترهای مقیاس‌پذیری قابل یادگیری برای s و t را مقداردهی اولیه می‌کند. همچنین همانطور که گفته شد از یک معماری Multi Scale پیروی می کند، که در آن ورودی به تدریج فشرده می شود و به ابعاد فضایی کوچکتر اما با افزایش تعداد کانال ها تقسیم می شود. این معماری برای مدیریت داده های با ابعاد بالا مانند تصاویر بسیار مهم است.

[1] https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

در ادامه به کلاس ادیت شده real NVP پرداخته میشود که در این مدل تمامی mask conv ها برداشته شده است و بخش هایی از آن ادیت شده است..

- شکل ورودی مجموعه داده MNIST (1 کانال، 28 در 28) را تنظیم میشود.
- لیست هایی (nn.ModuleList و nn.ParameterList) برای توابع مقیاس بندی (s) و translation (t) و مقیاس ها و بایاس های مربوط به آنها ایجاد می کند.
- این مدل به تدریج شکل ورودی را از طریق لایه ها تغییر می دهد و به طور متناوب بین تغییر تعداد کانال ها و کاهش ابعاد فضایی تغییر می کند.
- لایه های کوپلینگ نهایی پس از تبدیل چند مقیاسی اضافه می شوند.
- این مدل از لایه‌های کوپلینگ Affine استفاده می‌کند، جایی که بخشی از ورودی با استفاده از توابع ساده و معکوس بر اساس بقیه ورودی به‌روزرسانی می‌شود. این لایه ها از توابع مقیاس بندی (s) و transaltion (t) تشکیل شده اند.
- EPSILON = 1e-5 - یک مقدار ثابت کوچک را برای ثبات عددی تنظیم می کند.
- self.shape = (1، 28، 28) - شکل اولیه ورودی را تنظیم می کند (برای تصاویر MNIST).
- self.planes = planes - تعداد صفحات (کانال) را تنظیم می کند.

اجزای مدل:

- self.s = nn.ModuleList() - یک لیست برای ذخیره ماژول های تابع مقیاس بندی ایجاد می کند.
- self.t = nn.ModuleList() - فهرستی برای ذخیره ماژول های تابع transaltion ایجاد می کند.
- self.s_scale = nn.ParameterList() و غیره - لیست هایی را برای پارامترهای مقیاس پذیری قابل یادگیری و بایاس ها برای s و t راه اندازی می کند.
- shape = self.shape - یک متغیر را برای پیگیری شکل از طریق لایه ها راه اندازی می کند.
- ساخت مدل:

حلقه for برای ایجاد لایه های شبکه، اضافه کردن توابع s و t و پارامترهای آنها تکرار می شود. شکل در فواصل زمانی خاص تغییر می کند و تعداد کانال ها یا ابعاد فضایی را تغییر می دهد.

- برای i در محدوده (num_final_coupling): - لایه های کوپلینگ نهایی را اضافه می کند.
- self.validation = False - پرچمی برای نشان دادن اینکه آیا مدل در حالت اعتبارسنجی است یا خیر.

روش های تنظیم حالت:

- def validate(self): و def train(self, mode=True): - روش هایی برای جابجایی بین حالت های اعتبار سنجی و آموزش.

Forward مدل :

- بخش اول (در صورت self.training یا self.validation:) تبدیل رو به جلو، اعمال لایه های کوپلینگ ، نرمال سازی و reshape را انجام می دهد.
- بخش دوم (در زیر else:) تبدیل معکوس را انجام می دهد و ورودی را از حالت تبدیل شده بازسازی می کند.

مراحل پردازش در Forward Pass:

- این کد عملکردهای مقیاس‌بندی و translation را اعمال می‌کند، Batch Normalization را انجام می‌دهد، داده‌ها را با استفاده از عملیات‌هایی مانند Un Shuffle Pixel و pixel shuffle تغییر شکل می‌دهد، و خروجی تبدیل شده و مقادیر میانی را ردیابی می‌کند.

کد نهایی مدل به صورت زیر میباشد:

```
class RealNVP(nn.Module):
    EPSILON = 1e-5

    def __init__(self, num_coupling=6, num_final_coupling=4,
planes=64):
        super(RealNVP, self).__init__()
        self.num_coupling = num_coupling
        self.num_final_coupling = num_final_coupling
        self.shape = (1, 28, 28)

        self.planes = planes
        self.s = nn.ModuleList()
        self.t = nn.ModuleList()
        self.norms = nn.ModuleList()

        # Learnable scalar scaling parameters for outputs of S and T
        self.s_scale = nn.ParameterList()
        self.t_scale = nn.ParameterList()
        self.t_bias = nn.ParameterList()
        self.shapes = []

        shape = self.shape
        for i in range(num_coupling):
            self.s.append(bottleneck_backbone(shape[0], planes))
            self.t.append(bottleneck_backbone(shape[0], planes))
```

```
            self.s_scale.append(torch.nn.Parameter(torch.zeros(shape)
, requires_grad=True))
            self.t_scale.append(torch.nn.Parameter(torch.zeros(shape)
, requires_grad=True))
            self.t_bias.append(torch.nn.Parameter(torch.zeros(shape),
requires_grad=True))

            self.norms.append(MyBatchNorm2d(shape[0]))

            self.shapes.append(shape)

            if i % 6 == 2:
                shape = (4 * shape[0], shape[1] // 2, shape[2] // 2)

            if i % 6 == 5:
                # Factoring out half the channels
                shape = (shape[0] // 2, shape[1], shape[2])
                planes = 2 * planes

        # Final coupling layers
        for i in range(num_final_coupling):
            self.s.append(bottleneck_backbone(shape[0], planes))
            self.t.append(bottleneck_backbone(shape[0], planes))

            self.s_scale.append(torch.nn.Parameter(torch.zeros(shape)
, requires_grad=True))
            self.t_scale.append(torch.nn.Parameter(torch.zeros(shape)
, requires_grad=True))
            self.t_bias.append(torch.nn.Parameter(torch.zeros(shape),
requires_grad=True))

            self.norms.append(MyBatchNorm2d(shape[0]))

            self.shapes.append(shape)

        self.validation = False

    def validate(self):
        self.eval()
        self.validation = True

    def train(self, mode=True):
        nn.Module.train(self, mode)
        self.validation = False

    def forward(self, x):
        if self.training or self.validation:
```

```
            s_vals = []
            norm_vals = []
            y_vals = []

            for i in range(self.num_coupling):
                shape = self.shapes[i]

                t = self.t_scale[i] * self.t[i](x) + self.t_bias[i]
                s = self.s_scale[i] * torch.tanh(self.s[i](x))
                y = x * torch.exp(s) + t
                s_vals.append(torch.flatten(s))

                if self.norms[i] is not None:
                    y, norm_loss = self.norms[i](y,
validation=self.validation)
                    norm_vals.append(norm_loss)

                if i % 6 == 2:
                    y = torch.nn.functional.pixel_unshuffle(y, 2)

                if i % 6 == 5:
                    factor_channels = y.shape[1] // 2
                    y_vals.append(torch.flatten(y[:,
factor_channels:, :, :], 1))
                    y = y[:, :factor_channels, :, :]

                x = y

            # Final checkboard coupling
            for i in range(self.num_coupling, self.num_coupling +
self.num_final_coupling):
                shape = self.shapes[i]

                # Updated calculations without mask
                t = self.t_scale[i] * self.t[i](x) + self.t_bias[i]
                s = self.s_scale[i] * torch.tanh(self.s[i](x))
                y = x * torch.exp(s) + t
                s_vals.append(torch.flatten(s))

                if self.norms[i] is not None:
                    y, norm_loss = self.norms[i](y,
validation=self.validation)
                    norm_vals.append(norm_loss)

                x = y

            y_vals.append(torch.flatten(y, 1))
```

```
            return (torch.flatten(torch.cat(y_vals, 1), 1),
                    torch.cat(s_vals),
                    torch.cat([torch.flatten(v) for v in norm_vals])
if len(norm_vals) > 0 else torch.zeros(1),
                    torch.cat([torch.flatten(s) for s in
self.s_scale]))
        else:
            y = x
            y_remaining = y

            layer_vars = np.prod(self.shapes[-1])
            y = torch.reshape(y_remaining[:, -layer_vars:], (-1,) +
self.shapes[-1])
            y_remaining = y_remaining[:, :-layer_vars]

            # Reversed final checkboard coupling
            for i in reversed(range(self.num_coupling,
self.num_coupling + self.num_final_coupling)):
                shape = self.shapes[i]

                if self.norms[i] is not None:
                    y, _ = self.norms[i](y)

                t = self.t_scale[i] * self.t[i](y) + self.t_bias[i]
                s = self.s_scale[i] * torch.tanh(self.s[i](y))
                x = (y - t) * torch.exp(-s)

                y = x

            layer_vars = np.prod(shape)
            y = torch.cat((y, torch.reshape(y_remaining[:, -
layer_vars:], (-1,) + shape)), 1)
            y_remaining = y_remaining[:, :-layer_vars]

            # Multi-scale coupling layers
            for i in reversed(range(self.num_coupling)):
                shape = self.shapes[i]

                if self.norms[i] is not None:
                    y, _ = self.norms[i](y)

                t = self.t_scale[i] * self.t[i](y) + self.t_bias[i]
                s = self.s_scale[i] * torch.tanh(self.s[i](y))
                x = (y - t) * torch.exp(-s)

                if i % 6 == 3:
                    x = torch.nn.functional.pixel_shuffle(x, 2)
```

```
                y = x

                if i > 0 and i % 6 == 0:
                    layer_vars = np.prod(shape)
                    y = torch.cat((y, torch.reshape(y_remaining[:, -
layer_vars:], (-1,) + shape)), 1)
                    y_remaining = y_remaining[:, :-layer_vars]

            assert np.prod(y_remaining.shape) == 0

            return x
```

در ادامه تابع لاس به صورت زیر تعریف میشود که این تابع برای آموزش یک مدل RealNVP، با در نظر گرفتن جنبه های منحصر به فرد Normalizing Flow از جمله ژاکوبین تبدیل و توزیع داده های تبدیل شده، طراحی شده است. گنجاندن یک ترم منظم سازی و نرمال سازی بر اساس اندازه دسته، شیوه های استاندارد برای بهبود ثبات و تعمیم در ترین هستند.

```
PI = torch.tensor(np.pi).to(device)
def loss_fn(y, s, norms, scale, batch_size):
    logpx = -torch.sum(0.5 * torch.log(2 * PI) + 0.5 * y**2)
    det = torch.sum(s)
    norms = torch.sum(norms)
    reg = 5e-5 * torch.sum(scale ** 2)
    loss = -(logpx + det + norms) + reg
    return torch.div(loss, batch_size), (-logpx, -det, -norms, reg)
```

به کمک کد زیر دیتاست ها MNIST دانلود میشود.

```
train_dataset = datasets.MNIST('data', train=True, download=True,
                               transform=transforms.Compose([
                                   transforms.ToTensor(),
                               ]))


test_dataset = datasets.MNIST('data', train=False, download=True,
                              transform=transforms.Compose([
                                  transforms.ToTensor(),
                              ]))
```

لوپ ترین و تست هم به صورت زیر تعریف میشود همچنین مقادیر لاس نیز برگردانده میشوند.

```
def train_loop(dataloader, model, loss_fn, optimizer, batch_size,
report_iters=10, num_pixels=28*28):
```

```
    size = len(dataloader)
    prev = []
    loss_train =[]
    for batch, (X, _) in enumerate(dataloader):
        # Transfer to GPU
        X = pre_process(X)
        X = X.to(device)

        # Compute prediction and loss
        y, s, norms, scale = model(X)
        loss, comps = loss_fn(y, s, norms, scale, batch_size)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()

        prev = [(name, x, x.grad) for name, x in
model.named_parameters(recurse=True)]
        optimizer.step()
        if batch % report_iters == 0:
            loss, current = loss.item(), batch
            loss += num_pixels * np.log(255)
            loss_train.append(loss)
    print("Train_loss :", mean(loss_train))
    return  mean(loss_train)

def test_loop(dataloader, model, loss_fn, num_pixels=28*28):
    size = len(dataloader)
    num_batches = len(dataloader)
    test_loss = 0
    loss_val =[]
    with torch.no_grad():
        model.validate()
        for X, _ in dataloader:
            X = pre_process(X)
            X = X.to(device)
            y, s, norms, scale = model(X)
            loss, _ = loss_fn(y, s, norms, scale, batch_size)
            test_loss += loss

        model.train()


    test_loss /= num_batches
    test_loss += num_pixels * np.log(255)
    print("Val Loss :" , test_loss.item())
    return test_loss.item()
```

یک سری توابع جهت اضافه کردن نویز و برگرداندن آن به حالت بین صفر و یک و همچنین عملیات برعکس آن نیز با عناوین post process و preprocess تعریف میشود.

```
def pre_process(x):
    x = x * 255.
    x = x + torch.rand(x.shape)
    return x / 255

def post_process(x):
    return torch.clip(x, min=0, max=1)
```

مدل با پارامتر های زیر و scheduler زیر در نهایت تعریف میشود.

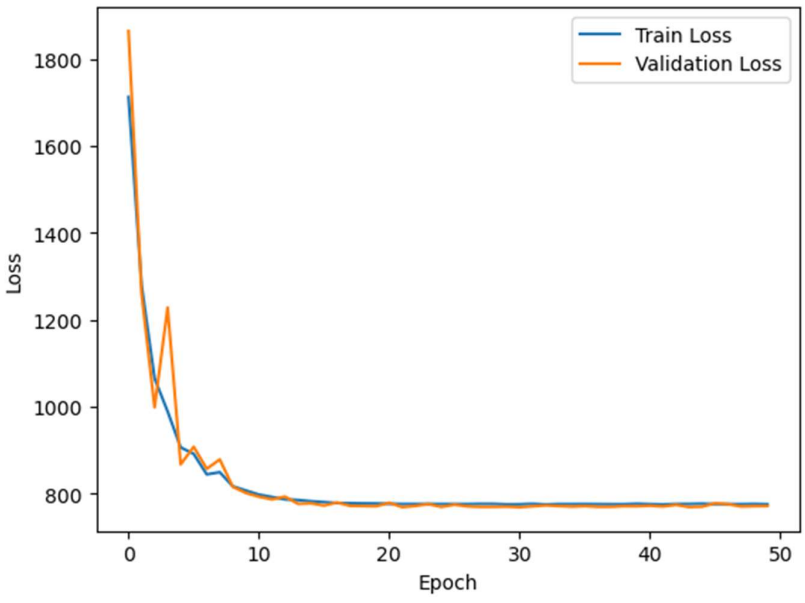
```
learning_rate = 0.0005
batch_size = 128
epochs = 10

model = RealNVP(num_coupling=12, num_final_coupling=4,
planes=64).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5,
gamma=0.2)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=True)
```

در پایان نیز به کمک کد زیر فرایند آموزش انجام میشود. (در نهایت بهترین مدل بر اساس val نیز ذخیره میشود)

```
model.train()


PATH = 'checkpoints/'
os.makedirs(PATH, exist_ok=True)

# Lists to store training and validation losses
train_losses = []
validation_losses = []

best_validation = None

for t in range(epochs):
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
    print(f"Epoch {t+1}\n-------------------------------")
```

```
    train_loss = train_loop(train_loader, model, loss_fn, optimizer,
batch_size)
    validation_loss = test_loop(test_loader, model, loss_fn)

    train_losses.append(train_loss)
    validation_losses.append(validation_loss)

    torch.save({
        'epoch': t,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': validation_loss,
    }, os.path.join(PATH, f'mnist-{t}.model'))

    if best_validation is None or validation_loss < best_validation:
        best_validation = validation_loss
        best_path = os.path.join(PATH, f'mnist-{t}.model')

    scheduler.step()

print("Done - ", best_path)
```

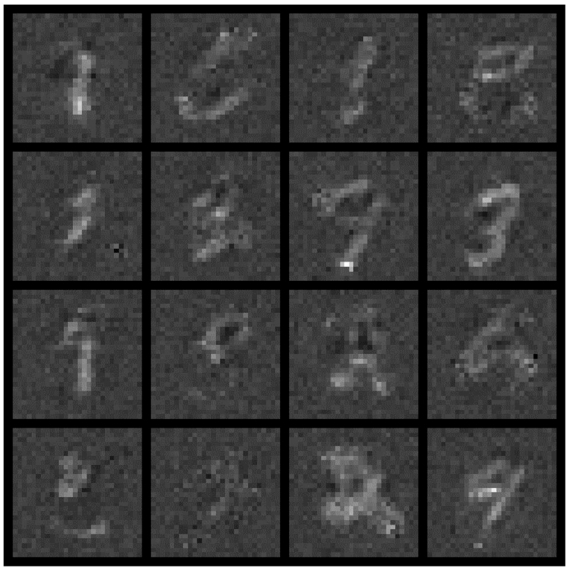
منحنی لاس برای داده های آموزش و ارزیابی به صورت زیر میشود:



شکل 2 : منحنی لاس برای داده های آموزش و ارزیابی برای **Real NVP** بدون **mask CONV**

برای گرفتن خروجی از مدل نیز میتوان به صورت زیر مدل را لود کرده و یک توزیع نرمال را به مدل داده و خروجی را مشاهده کنیم که این کار برای 16 بار انجام شده است و نتایج به صورت زیر میباشد.

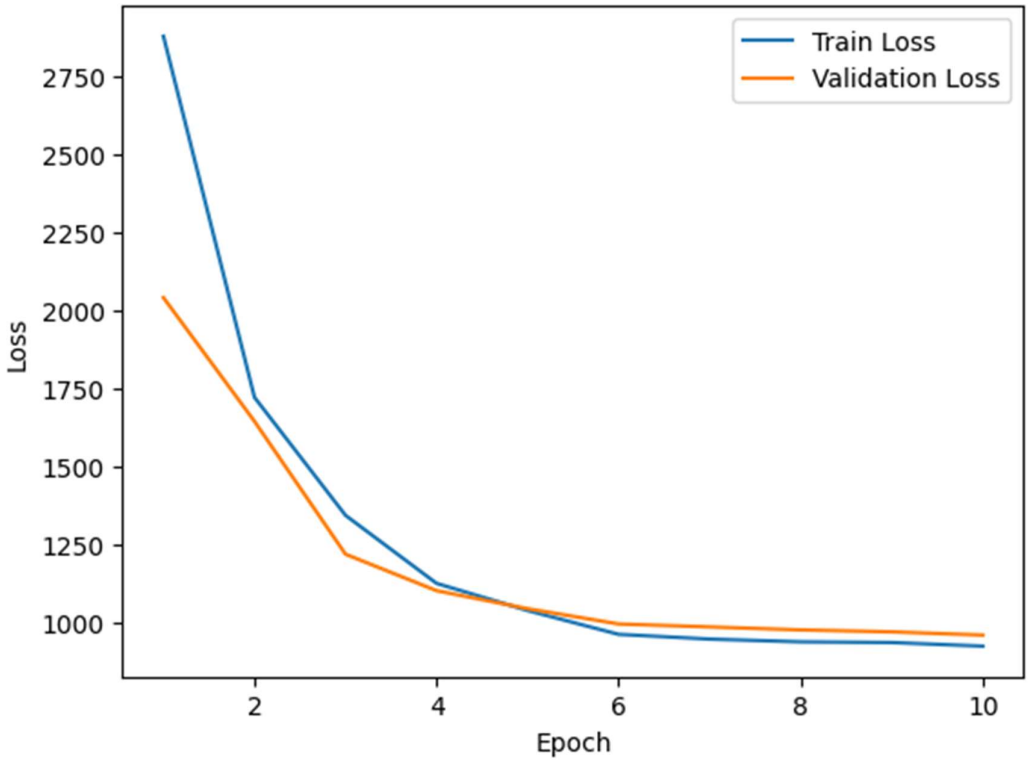
```
model = RealNVP(num_coupling=12, num_final_coupling=4,
planes=64).to(device)

#checkpoint = torch.load('checkpoints/mnist-1.model')
checkpoint = torch.load(best_path)
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()

cols, rows = 4, 4
with torch.no_grad():
    X = torch.Tensor(torch.normal(torch.zeros(cols * rows, 28 * 28 *
1),
                                  torch.ones(cols * rows, 28 * 28 *
1))).to(device)
    Y = model(X)
    samples = post_process(Y).cpu().numpy()

figure = plt.figure(figsize=(9, 9))
for i in range(1, cols * rows + 1):
    img = samples[i - 1]
    figure.add_subplot(rows, cols, i)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



شکل 3 : 16 نمونه تصاویر تولیدی مدل **real NVP**

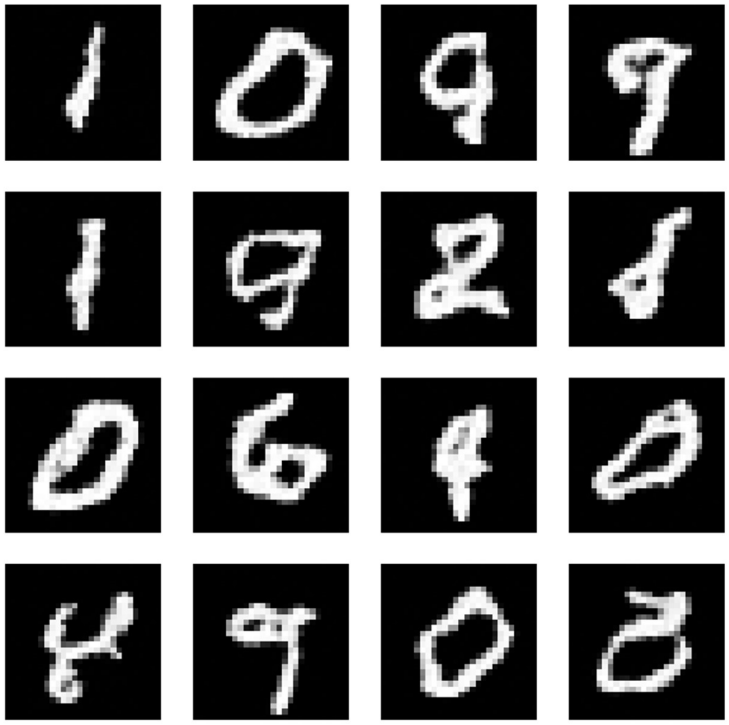
همانطور که مشاهده میشود کیفیت تصاویر تولیدی خروجی مناسب نمیباشد به همین دلیل الگوریتم را اینبار به MASK CONV و Batch normalization تکرار میکنیم. کد این بخش نیز همراه با کد ارسالی فرستاده شده است.

منحنی لاس در این حالت به صورت زیر میشود. (مشاهده میشود که با وجود مقادیر بالاتر loss اما نوسان در val دیگر وجود ندارد.)



شکل 4 : منحنی لاس برای داده های آموزش و ارزیابی برای **Real NVP** با **mask CONV**

خروجی های تولید شده در این حالت به شرح زیر میباشد که نشان از دقت و عملکرد بالاتر مدل real NVP کامل با تمام ماژول هایش دارد.



شکل 5 : 16 نمونه تصاویر تولیدی مدل **real NVP** با **MASK CONV**