به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

مدل‌های مولد عمیق

تمرین شماره دو

نام و نام خانوادگی	کیانا هوشانفر
شماره دانشجویی	۸۱۰۱۰۱۳۶۱
تاریخ ارسال گزارش	۱۴۰۲/۰۹/۰۲

فهرست گزارش سوالات

سوال ۱

الف)

$$q(z|x) = \mathcal{N}(z; \mu(x), \text{diag}(\sigma^2(x)))$$
$$p(z) = \mathcal{N}(z; 0, I)$$

$$D(q(z|x)\|p(z)) = \int q(z|x) \log \frac{q(z|x)}{p(z)} dz$$

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

$$= \int q(z|x) \left[-\frac{1}{2}\log \prod_{i=1}^{n} \sigma_i^2(x) - \frac{1}{2}(z-\mu(x))^T \text{diag}(\sigma^2(x))^{-1}(z-\mu(x)) + \frac{1}{2} z^T z\right] dz$$

$$= -\frac{1}{2}\log \prod_i \sigma_i^2(x) \int q(z|x)\, dz + \frac{1}{2}\int q(z|x) \sum_i \left(\frac{(z-\mu(x))^2}{\sigma_i^2(x)} - z_i^2\right) dz$$

$$= -\frac{1}{2}\sum \log \sigma_i^2(x) + \frac{1}{2}\sum_i \left(\left(1 + \mu_i^2(x) - \sigma_i^2(x) - z_i^2\right)\right)$$

$$E[(z_i-\mu_i)^2] = \sigma_i^2 \quad E[z_i^2] = \mu_i^2 + \sigma_i^2$$

$$\Rightarrow D(q(z|x)\|p(z)) = \frac{1}{2}\sum_{i=1} \left(\sigma_i^2(x) + \mu_i^2(x) - \log(\sigma_i^2(x)) - 1\right)$$

$D(q(z|x)\|p(z))$ = Cross entropy − entropy

cross entropy $= -\int q(z) \log p(z)\, dz = -\int q(z) \log\left((2\pi)^{-\frac{1}{2}} \exp(-\frac{z^2}{2})\right) dz$

$$= \frac{1}{2}\log 2\pi \int q(z)\, dz + \frac{1}{2}\int z^2 q(z)\, dz = \frac{1}{2}\left[\log 2\pi + \mu^2 + \sigma^2\right]$$

$\int z^2 q(z)\, dz \to \mu^2 + \sigma^2$

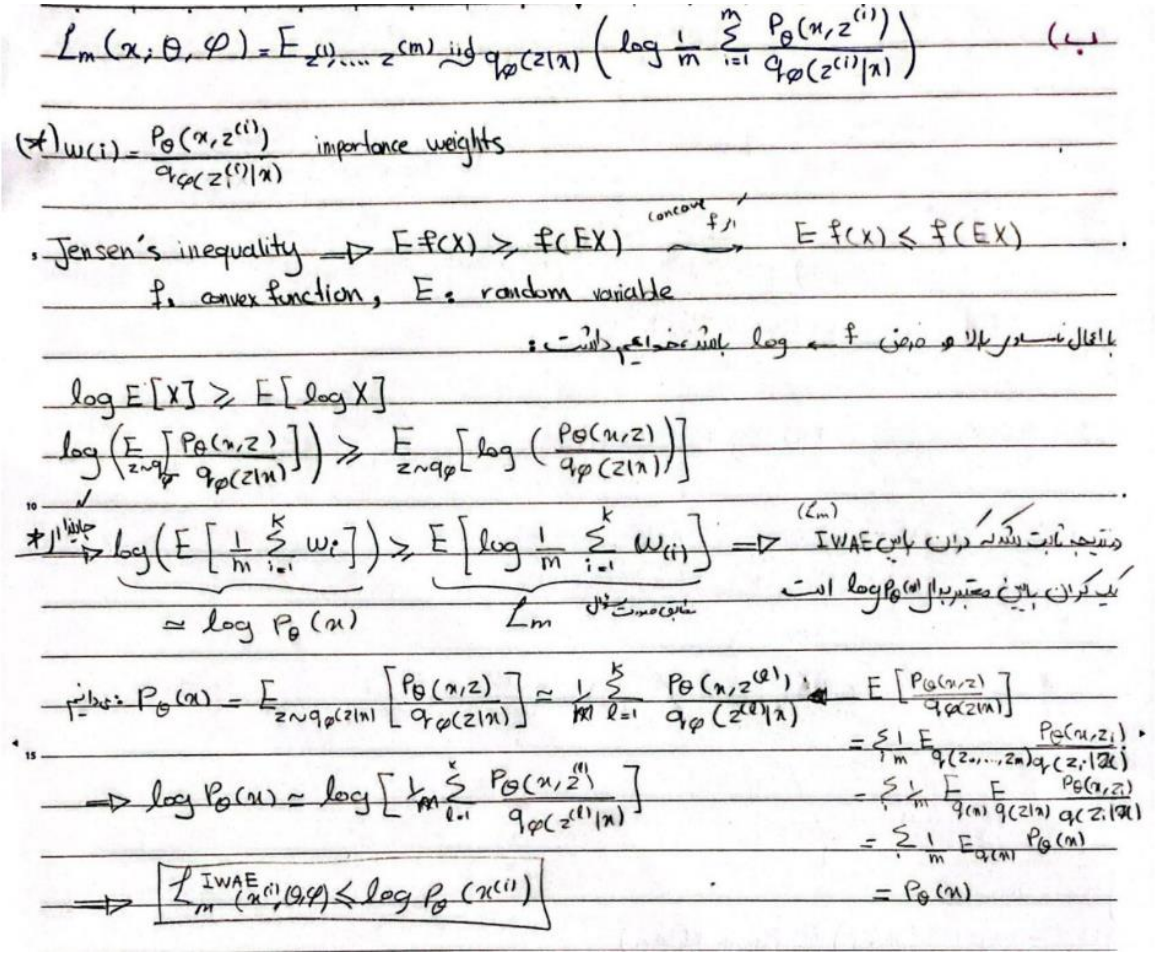
entropy $= -\int q(z) \log q(z)\, dz = -\int q(z) \log\left[(2\pi\sigma^2)^{-\frac{1}{2}} \exp(z-\mu)^2\right] dz$

$$= \frac{1}{2}\log 2\pi \int q(z)\, dz + \frac{1}{2}\int \frac{(z-\mu)^2}{\sigma^2} q(z)\, dz = \frac{1}{2}\left[\log 2\pi + \log\sigma^2 + 1\right]$$

$$\Rightarrow D(q(z|x)\|p(z)) = \frac{1}{2}\left[\log 2\pi + \mu^2 + \sigma^2 - \log 2\pi - \log\sigma^2 - 1\right] = \frac{1}{2}\left[\mu^2 + \sigma^2 - \log\sigma^2 - 1\right]$$

بردار باشد $\Rightarrow$ $D_{KL} = \sum_{i=1}^{K} \frac{1}{2}\left[\sigma_i^2 + \mu_i^2 - \log\sigma_i^2 - 1\right]$

ب)



ج)

Posterior collapse در رمزگذارهای خودکار متغیر VAEs زمانی به وجود می آید که توزیع variational posterior به طور نزدیک با prior برای زیرمجموعه ای از متغیرهای پنهان مطابقت داشته باشد. Posterior collapse به وضعیتی اشاره دارد که در آن متغیرهای latent آموخته شده در مدل تولیدی به طور موثر مورد استفاده قرار نمی گیرند و در عوض توسط مدل در طول فرآیند یادگیری نادیده گرفته می شوند. این معمولاً منجر به این می‌شود که رمزگشا بتواند نمونه‌هایی با ظاهر واقعی تولید کند، اما متغیرهای پنهان (که قرار است اطلاعات معنی‌داری در مورد داده‌ها جمع‌آوری کنند) بی‌اطلاع می‌شوند یا به درستی استفاده نمی‌شوند. (زمانی که سیگنال از ورودی x به پارامترهای پسین بسیار ضعیف یا پر نویز باشد، می گوییم یک Posterior collapse است، و در نتیجه، رمزگشا شروع به نادیده گرفتن نمونه های z گرفته شده از قسمت posterior می کند.)

دلیل Posterior collapse اغلب در فرآیند بهینه سازی در طول تمرین نهفته است. در VAEها، مدل آموزش داده می‌شود تا حد پایین‌تری را در احتمال داده‌ها به حداکثر برساند و در عین حال واگرایی بین توزیع تقریبی پسین (رمزگذار) و توزیع prior متغیرهای پنهان را نیز به حداقل برساند. چالش زمانی به وجود می‌آید که مدل کاهش عبارت واگرایی را به فروپاشی تقریبی توزیع پسین به یک نقطه ترجیح می دهد و آن را ساده تر می یابد، و به طور مؤثر متغیرهای پنهان را نادیده می‌گیرد و آنها را بی‌اطلاع می‌کند.

جلوگیری یا کاهش Posterior collapse در VAE وجود دارد:

۱. از یک تابع هدف متعادل استفاده کنیم: عبارت بازسازی (احتمال داده) و عبارت منظم سازی (واگرایی KL را در تابع هدف متعادل کنید. اگر یک اصطلاح بر دیگری تسلط داشته باشد، مدل ممکن است یک جنبه را بر دیگری اولویت دهد و منجر به مسائلی مانند فروپاشی پسین شود.

۲. به تدریج وزن ترم واگرایی KL را در طول تمرین افزایش دهیم. این به مدل اجازه می‌دهد در ابتدا بر بازسازی تمرکز کند و سپس اصطلاح منظم‌سازی را بعداً معرفی کند، که می‌تواند به تثبیت روند آموزش کمک کند.

۳. ظرفیت فضای latent: ظرفیت فضای latent را در نظر بگیریم. اگر خیلی کوچک باشد، مدل ممکن است ظرفیت کافی برای نشان دادن تغییرپذیری در داده ها را نداشته باشد و منجر به فروپاشی پسین شود. با فضای پنهان بزرگتر آزمایش کنید.

۴VAE. های شرطی: در آن متغیرهای latent به اطلاعات اضافی مشروط هستند. این می تواند به گرفتن وابستگی های پیچیده تر در داده ها کمک کند.

۵. معماری های مختلف: تغییرات در معماری، مانند استفاده از نوع دیگری از شبکه عصبی یا افزایش پیچیدگی مدل، می‌تواند بر وقوع این پدیده تأثیر بگذارد.

۶. تکنیک‌های منظم‌سازی: سایر تکنیک‌های منظم‌سازی، مانند اضافه کردن نویز به داده‌های ورودی یا استفاده از حذف در فضای پنهان را بررسی کنیم تا مدل را تشویق کند تا از کل فضای latent استفاده کند.

(بطور خلاصه: Posterior collapse به بهترین وجه به عنوان نمونه ای از non-identifiability شناخته می شود. اگر چگالی posterior متغیرهای پنهان برابر با prior خود باشد، به این معنی است که اطلاعات کافی در نمونه برای تخمین مدل انتخابی وجود ندارد.)

د)

اثبات این: $\log p_\theta(x) \geq ELBO(x;\theta,\varphi) = E_{q_\varphi}[\log p_\theta(x|z)] - KL(q_\varphi(z|x) \| p(z))$ (د

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

$$= \int q_\varphi(z|x) \log p_\theta(x,z)\, dz - \int q_\varphi(z|x) \log q_\varphi(z|x)\, dz$$

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

$$= \int q_\varphi(z|x) \log p_\theta(x|z)\, dz + \int q_\varphi(z|x) \log \frac{p_\theta(z)}{q_\varphi(z|x)} dz$$

$$= E_{z\sim q_\varphi(z|x)}[\log p_\theta(x|z)] - KL(q_\varphi(z|x) \| p_\theta(z))$$

$$\Rightarrow E[\log p_\theta(x|z)] = ELBO + KL(q_\varphi(z|x) \| p_\theta(z))$$

$\oplus$ ÷ & ← KL divergence

$$\Rightarrow ELBO \ll E[\log p_\theta(x|z)] = \log p(x) \Rightarrow \boxed{\log p_\theta(x) \geq ELBO}$$

Subject: Date:

$$p_\theta(x^{(i)}) = E_{z\sim p_\theta(z)}[p_\theta(z|x^{(i)})] = \int p_\theta(z)\, p_\theta(z|x^{(i)})\, dz$$

posterior بسنده

$$= \int \frac{p_\theta(x^{(i)},z)}{q_\varphi(z|x^{(i)})} q_\varphi(z|x^{(i)})\, dz = E_{z\sim q_\varphi(z|x^{(i)})}\left[\frac{p_\theta(x^{(i)},z)}{q_\varphi(z|x^{(i)})}\right]$$

Jensen's inequality بااستفاده (با تقعر)

$$\log p_\theta(x^{(i)}) = \log E_{z\sim q_\varphi(z|x)}\left[\frac{p_\theta(x,z)}{q_\varphi(z|x)}\right] \geq E\left[\log \frac{p_\theta(x,z)}{q_\varphi(z|x)}\right]$$

elbo

$E[f(x)] \geq f(Ex)$

$$\Rightarrow \boxed{\log p_\theta(x) \geq elbo}$$

$$\log p(x) = \int q_\varphi(z|x) \log p_\theta(x|z)\, dz = E_{q_\varphi(z|x)}[\log p_\theta(x|z)]$$

سوال ۲

الف)

```
# Generate random samples from a standard normal distribution
epsilon = torch.randn_like(v)

# Reparameterization trick
z = m + torch.sqrt(v) * epsilon
```

شکل ۱ – کد کامل شده بخش **sample_gaussian**

این تابع دو پارامتر ورودی m میانگین و v واریانس را می گیرد که هر دو به صورت تنسور نمایش داده می شوند. این تابع یک تانسور z را برمی‌گرداند که نمونه‌هایی را نشان می‌دهد که از توزیع گاوسی گرفته شده‌اند.

torch.randn_like(v) نمونه های تصادفی را از یک توزیع نرمال استاندارد با شکلی مشابه با تنسور ورودی v تولید می کند. این به صورت اپسیلون نشان داده می شود و بردار اعداد تصادفی است که از توزیع نرمال با میانگین ۰ و واریانس ۱ نمونه برداری شده است.

ترفند پارامترسازی مجدد تکنیکی است که در مدل‌های احتمالی، به‌ویژه در رمزگذارهای خودکار متغیر VAE استفاده می‌شود.

z = m + torch.sqrt(v) * epsilon ترفند پارامترسازی مجدد را اعمال می کند. در اینجا m میانگین، v واریانس و اپسیلون نمونه تصادفی از توزیع نرمال استاندارد است.

این رابطه نمونه ها را از یک توزیع نرمال استاندارد (اپسیلون) به نمونه هایی از یک توزیع گاوسی با میانگین m و واریانس v مشخص شده تبدیل می کند.

ب)

```
# Encode input
mean, variance = self.enc.encode(x)

# Sample from the encoded distribution
z = sample_gaussian(mean, variance)

# Decode the sampled representation
logits = self.dec.decode(z)

# Calculate KL divergence, reconstruction error, and negative ELBO
kl = kl_normal(mean, variance, self.z_prior[0], self.z_prior[1])
rec = -log_bernoulli_with_logits(x, logits)
nelbo = kl + rec
```

شکل ۲ – تابع **negative_elbo_bound**

این تابع کران پایین شواهد منفی NELBO را محاسبه می کند، کمیتی که در طول فرآیند آموزش به حداقل می رسد.

مرحله ۱: رمزگذاری ورودی

داده ورودی x از طریق رمزگذار self.enc ارسال می شود. انتظار می رود رمزگذار دو پارامتر کلیدی، میانگین و واریانس را ارائه دهد، که نشان دهنده پارامترهای یک توزیع گاوسی متناسب با داده های ورودی است.

مرحله ۲: نمونه برداری از توزیع رمزگذاری شده

یک نمونه z از توزیع گاوسی گرفته می شود که با میانگین و واریانس به دست آمده از رمزگذار مشخص می شود.

مرحله ۳: رمزگشایی

سپس نمایش z latent نمونه برداری شده با استفاده از رمزگشا self.dec رمزگشایی می شود تا logit هایی به دست آید که می تواند به عنوان ورودی بازسازی شده تفسیر شود.

مرحله ۴: محاسبه KL Divergence

KL Divergence بین توزیع کدگذاری شده و توزیع قبلی مشخص شده self.z_prior برای فضای latent محاسبه می شود. این متغیر مقدار اطلاعاتی را که هنگام استفاده از توزیع رمزگذاری شده برای تقریب prior از دست می‌رود، نشان می‌دهد.

مرحله ۵: محاسبه خطای Reconstruction

احتمال negative log-likelihood، x با توجه به logits های بازسازی شده محاسبه می شود. این عبارت خطای بازسازی را نشان می‌دهد و نشان می‌دهد که مدل چقدر قادر به بازسازی داده‌های ورودی است.

مرحله ۶: محاسبه ELBO منفی

KL Divergence و خطای بازسازی برای تشکیل Negative Evidence Lower Bound (NELBO) ترکیب می شوند. به حداقل رساندن NELBO در طول آموزش، مدل را تشویق می کند تا تعادل مؤثری بین بازسازی و پایبندی به توزیع prior مشخص شده پیدا کند.

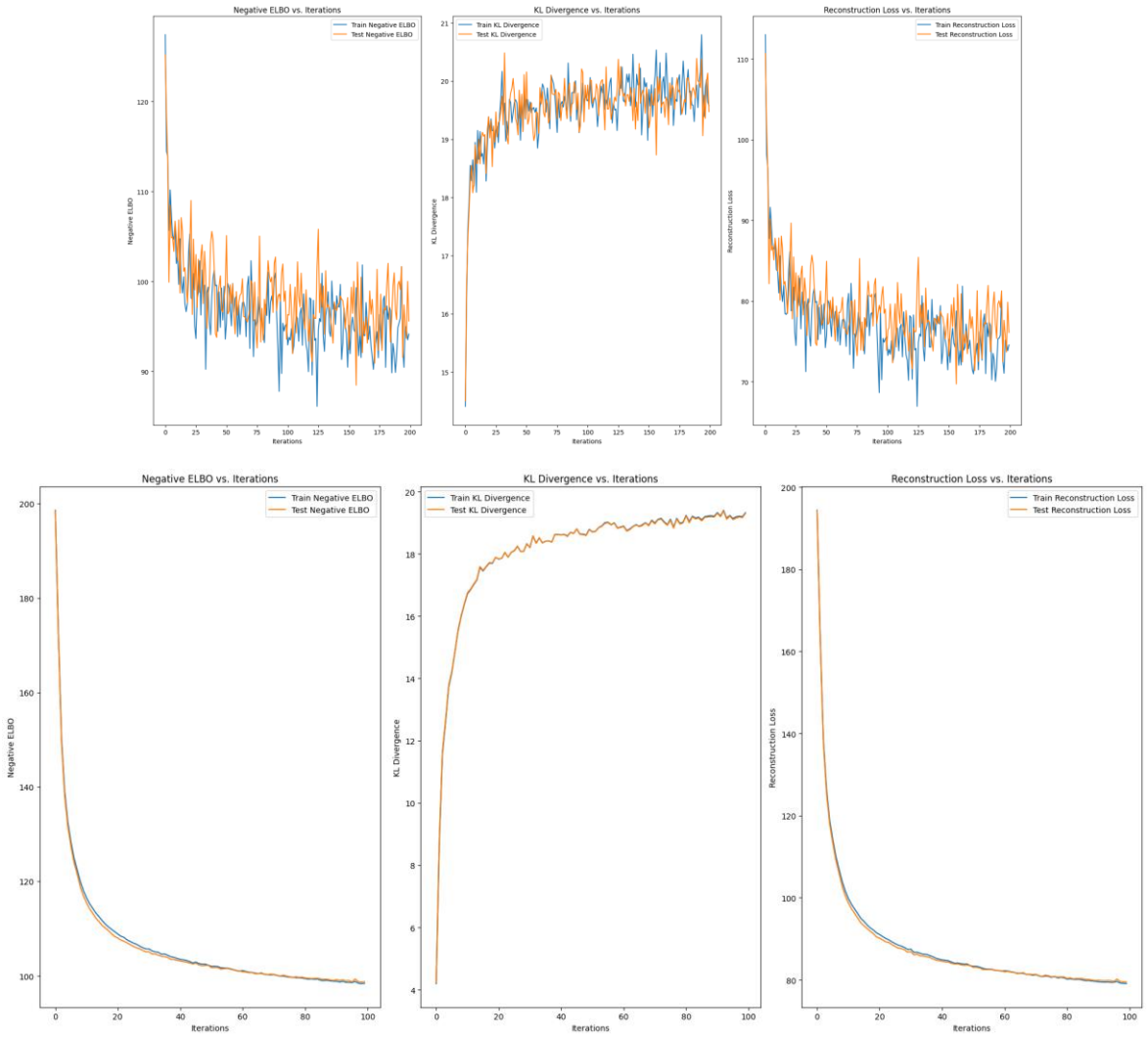
ج)

مدل vae را با train loop پایین آموزش می دهیم:

```python
import torch
import torch.optim as optim
import numpy as np
from tqdm import tqdm

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

# Lists to store the loss terms during training and test
train_losses = []
train_kls = []
train_recs = []
test_losses = []
test_kls = []
test_recs = []

# Training loop
for epoch in tqdm(range(iter_max)):
    vae.train()  # Set the model to training mode

    for batch_idx, (xu, yu) in enumerate(train_loader):
        optimizer.zero_grad()

        # Convert input to binary and one-hot encode labels
        xu = torch.bernoulli(xu.to(device).reshape(xu.size(0), -1))
        yu = yu.new(np.eye(10)[yu]).to(device).float()

        # Calculate loss and backpropagate
        loss, summaries = vae.loss(xu)
        loss.backward()
        optimizer.step()

    if epoch % 1 == 0:
        # Training dataset
        vae.eval()  # Set the model to evaluation mode

        # Lists to store individual batch losses during an epoch
        epoch_train_losses = []
        epoch_train_kls = []
        epoch_train_recs = []
```

```python
        for batch_idx, (xu, yu) in enumerate(train_loader):
            xu = torch.bernoulli(xu.to(device).reshape(xu.size(0), -1))
            yu = yu.new(np.eye(10)[yu]).to(device).float()
            train_loss, train_kl, train_rec = vae.negative_elbo_bound(xu)

            epoch_train_losses.append(train_loss.item())
            epoch_train_kls.append(train_kl.item())
            epoch_train_recs.append(train_rec.item())

        # Calculate the mean of losses for the epoch
        mean_train_loss = np.mean(epoch_train_losses)
        mean_train_kl = np.mean(epoch_train_kls)
        mean_train_rec = np.mean(epoch_train_recs)

        train_losses.append(mean_train_loss)
        train_kls.append(mean_train_kl)
        train_recs.append(mean_train_rec)

        print(f"Epoch {epoch + 1}/{iter_max} - Training Negative ELBO: {mean_train_loss}, KL Divergence: {mean_train_kl}, Reconstruction Loss: {mean_train_rec}")

        # Validation dataset
        epoch_test_losses = []
        epoch_test_kls = []
        epoch_test_recs = []

        for batch_idx, (xv, yv) in enumerate(test_loader):
            xv = torch.bernoulli(xv.to(device).reshape(xv.size(0), -1))
            yv = yv.new(np.eye(10)[yv]).to(device).float()
            test_loss, test_kl, test_rec = vae.negative_elbo_bound(xv)

            epoch_test_losses.append(test_loss.item())
            epoch_test_kls.append(test_kl.item())
            epoch_test_recs.append(test_rec.item())

        # Calculate the mean of losses for the epoch
        mean_test_loss = np.mean(epoch_test_losses)
        mean_test_kl = np.mean(epoch_test_kls)
        mean_test_rec = np.mean(epoch_test_recs)

        test_losses.append(mean_test_loss)
        test_kls.append(mean_test_kl)
        test_recs.append(mean_test_rec)

        print(f"Epoch {epoch + 1}/{iter_max} - Validation Negative ELBO: {mean_test_loss}, KL Divergence: {mean_test_kl}, Reconstruction Loss: {mean_test_rec}")
```

شکل ۳ – کد **train** مدل



شکل ۴ – نمودارها ی خواسته شده (در ۲ شیوه نمایش مختلف)

نمودار اولی، فقط دارد برای batch آخر مقادیر را ذخیره میکند، که هر epoch عوض میشود. درنتیجه شاهد نوسان هستیم. اما در نمودار پایینی آمدیم مقادیر خواسته شده را در هر batch_idx بدست آورده و در انتها از آن میانگین گرفته و مقدار را ذخیره کردیم و نمایش دادیم. مشاهده میکنیم که نمودار با شیب نرمی افزایش و کاهش پیدا کرده است و نوسان نداریم.

از نمودارهای بالا همانطور که انتظار داشتیم، متوجه می شویم که reconstruction error کاهش پیدا کرده است. KL Divergence افزایش پیدا کرده و بر روی مقدار ۲۰ ثابت شده هست. Negative elbo هم loss را در اینجا نشان می دهد که روند کاهش در آن می بینیم. الگوریتم‌های بهینه‌سازی معمولاً یک تابع loss را به حداقل می‌رسانند و ELBO باید حداکثر شود پس انتظار داریم که این نمودار کاهشی باشد که منهای آن حداکثر شود.

د)

```
# Encoding
m, v = self.enc.encode(x)

# Duplicate samples
m, v, x = map(lambda t: t.repeat(iw, 1), (m, v, x))


# Sample from the Gaussian distribution
z = sample_gaussian(m, v)

# Decoding
logits = self.dec.decode(z)

# Compute KL divergence
kl = log_normal(z, m, v) - log_normal(z, self.z_prior[0], self.z_prior[1])

# Compute reconstruction term
rec = -log_bernoulli_with_logits(x, logits)

# Compute NELBO (Negative ELBO)
nelbo = kl + rec

# Compute negative importance-weighted ELBO
niwae = -log_mean_exp(-nelbo.reshape(iw, -1), dim=0)

# Take mean over importance-weighted samples
niwae, kl, rec = niwae.mean(), kl.mean(), rec.mean()
```

شکل ۵ – تابع **negative_iwae_bound**

هدف این کد آموزش VAE و ارزیابی کیفیت نمایش متغیر latent آموخته شده با محاسبه importance-weighted evidence lower bound است. این معیار اغلب در استنتاج متغیر برای ارزیابی اینکه چگونه یک مدل تعادل بین برازش داده ها و نزدیک ماندن به توزیع قبلی انتخاب شده را متعادل می کند، استفاده می شود. علامت منفی استفاده می‌شود زیرا الگوریتم‌های بهینه‌سازی معمولاً یک تابع loss را به حداقل می‌رسانند و ELBO باید حداکثر شود.

این روش کران پایینی niwae را برای یک داده ورودی داده شده x و یک پارامتر وزن اهمیت iw محاسبه می‌کند.

رمزگذاری: داده های ورودی x از طریق یک رمزگذار self.enc.encode(x) برای به دست آوردن میانگین m و واریانس v یک توزیع گاوسی که نشان دهنده متغیر latent است، منتقل می شود.

نمونه های تکراری: میانگین m ، واریانس v و داده های ورودی x سپس iw بار تکرار می شوند. این کار برای ایجاد چندین نمونه با مقادیر متغیر latent مختلف برای برآورد اهمیت وزن انجام می شود.

تکثیر نمونه ها و استفاده از وزن های اهمیت، راهبردی برای به دست آوردن تخمین دقیق تر و پایدارتر از انتظارات در variational inference است. این به کاهش واریانس تخمین کمک می کند و به پایداری آموزشی مدل کمک می کند.

نمونه از توزیع گاوسی: نمونه های z از توزیع گاوسی که با میانگین m و واریانس v تعریف شده است، گرفته می شود. تابع sample_gaussian یک تابع است که نمونه هایی را از یک توزیع گاوسی تولید می کند.

رمزگشایی: نمونه‌های z از یک رمزگشا self.dec.decode(z) عبور داده می‌شوند تا logitهایی که نشان‌دهنده داده‌های بازسازی‌شده هستند، به دست آید.

محاسبه واگرایی KL : واگرایی KL بین توزیع نمونه های متغیر z latent و توزیع prior ، self.z_prior محاسبه می شود. این اندازه گیری می کند که چه مقدار از اطلاعات در هنگام استفاده از posterior تقریبی m و v به جای prior از دست می رود.

عبارت بازسازی محاسبه: احتمال ورود منفی داده های بازسازی شده log_bernoulli_with_logits(x-)، ((logits محاسبه می شود. این عبارت نشان می دهد که مدل چقدر داده های ورودی را بازسازی می کند.

محاسبه NELBO: به عنوان مجموع واگرایی KL و احتمال negative log likelihood بازسازی محاسبه می شود.

محاسبهniwae : با در نظر گرفتن میانگین log منفی نمایی از NELBO محاسبه می شود که به ردیف های iw تغییر شکل داده شده است.

نمونه‌های دارای وزن میانگین را در نظر بگیرید: در نهایت، میانگین شرایط niwae، واگرایی KL و بازسازی بر روی نمونه‌های دارای وزن اهمیت iw محاسبه می‌شوند.

۵)

```python
if __name__ == "__main__":

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

    # Values of m to evaluate
    m_values = [5, 50, 150]

    # Evaluate the model
    results = evaluate_vae_model(vae, test_loader, m_values)

    # Report the results
    report_results(results)
```

شکل ۶ – کد ارزیابی پیاده سازی

در کد ارائه شده، m_values فهرستی از مقادیر را برای تعداد نمونه های اهمیت مورد استفاده در محاسبه رمزگذار خودکار وزنی با IWAE و ELBO در طول ارزیابی مدل VAE نشان می دهد.

m به تعداد نمونه‌هایی اشاره دارد که از توزیع تغییرات هنگام محاسبه تخمین‌های مونت کارلو از expectations خاص، مانند مواردی که در محاسبه مرزهای IWAE و ELBO دخیل هستند، گرفته می‌شود. **افزایش مقدار m به طور کلی منجر به تخمین های دقیق تر می شود**.

```
Results:
m = 5: Negative IWAE = 94.35448744812011, Negative ELBO = 96.62817235870361
m = 50: Negative IWAE = 92.95993561553955, Negative ELBO = 96.62349592590331
m = 150: Negative IWAE = 92.61229615783691, Negative ELBO = 96.63181770019531
```

شکل ۷ – نتایج حاصل شده از این قسمت

Negative IWAE: مقادیر کمتر بهتر است، که نشان دهنده تناسب بهتر مدل با داده ها است. در نتایج ، با افزایش m، Negative IWAE کاهش می‌یابد، که نشان می‌دهد استفاده از نمونه‌های مهم‌تر و بیشتر، کیفیت تخمین log-likelihood را بهبود می‌بخشد.

Negative ELBO: مشابه Negative IWAE، مقادیر کمتر بهتر است. در نتایج ، روند آنقدر واضح نیست، اما با افزایش m ELBO منفی اندکی کاهش می یابد.

مقادیر negative log-likelihood نشان می‌دهد که مدل مولد کار بهتری را برای گرفتن الگوهای اساسی در داده‌ها و تولید نمونه‌هایی که شباهت زیادی به توزیع داده‌های واقعی دارند انجام می‌دهد. بنابراین، در زمینه مدل‌های مولد و معیارهای ارزیابی مبتنی بر احتمال، معمولاً مقادیر کمتر بهتر در نظر گرفته می‌شوند. این موضوع با نتایجی که بدست آوردیم مطابقت دارد.

سوال ۳

الف)

$$x = 1.5z + 3, p(z) = N(Z, 0, 1) \rightarrow p(x) = ?$$

$$p(x) = p(z = f^{-1}(x)) \left| \frac{\partial f^{-1}(x)}{\partial x} \right|$$

$$x = f(z) = 1.5z + 3 \rightarrow z = f^{-1}(x) = \frac{x - 3}{1.5} \Rightarrow \frac{\partial f^{-1}(x)}{\partial x} = \frac{1}{1.5} = \frac{2}{3}$$

$$p(x) = p_z\left(\frac{x - 3}{1.5}\right) \cdot \frac{2}{3} \Rightarrow \boxed{p(x) = N\left(\frac{x - 3}{1.5} \mid 0, 1\right) \cdot \frac{2}{3}}$$

ج)

$$y_a = x_a$$
$$y_b = \exp\left(s(x_a)\right) \odot x_b + t(x_a)$$
$$y = concat(y_a, y_b)$$

شکل ۸ - خروجی y

ماتریس ژاکوبین برای محاسبه log-likelihood استفاده می شود. در زمینه normalizing flows، log-likelihood را می‌توان به‌عنوان لگاریتم تعیین‌کننده ماتریس ژاکوبین تبدیل بیان کرد. ماتریس ژاکوبین توضیح می دهد که چگونه تغییر در هر عنصر ورودی بر تغییر در هر عنصر خروجی تبدیل تأثیر می گذارد.

هنگامی که یک تابع چگالی احتمال PDF تعریف شده در فضای اصلی دارید، و یک تبدیل برای به دست آوردن نمونه در یک فضای جدید اعمال می کنید، باید نحوه تغییر چگالی احتمال به دلیل تبدیل را در نظر بگیرید. دترمینان ماتریس ژاکوبین این تغییر در چگالی را محاسبه می کند و با گرفتن لگاریتم آن،

حاصلضرب دترمینان ها (به دلیل قانون زنجیره) را به جمع تبدیل می کند که از نظر محاسباتی راحت تر است.

در RealNVP، از آنجایی که تبدیل به صورت معکوس و از element-wise اعمال می شود، ماتریس ژاکوبین شکل ساده ای دارد که محاسبه آن را نسبتاً آسان می کند. این امر هم در مرحله آموزش و هم در مرحله تولید ضروری است.

به شکل زیر دترمینان این ماتریس را بدست می آوریم:

$$J=\begin{bmatrix} \frac{\partial y_a}{\partial x_a} & \frac{\partial y_a}{\partial x_b} \\ \frac{\partial y_b}{\partial x_a} & \frac{\partial y_b}{\partial x_b} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \exp(S(x_a)) \times \frac{\partial S(x_a)}{\partial x_a} \times x_b + \frac{\partial t(x_a)}{\partial x_a} & \exp(S(x_a)) \end{bmatrix}$$

$$\det J = \frac{\partial y_a}{\partial x_a} \times \frac{\partial y_b}{\partial x_b} - \frac{\partial y_a}{\partial x_b} \times \frac{\partial y_b}{\partial x_a} = \exp(S(x_a)) = \exp(\mathrm{tr}(S(x_a)))$$
$$= \exp(\sum_j S(x_a)_j)$$

د)

- تغییر متغیر:

فرمول Change of Variable در forward pass مدل RealNVP، اعمال می شود. فرمول تغییر متغیر در normalizing flows برای تبدیل یک توزیع احتمال معین به یک توزیع پیچیده تر استفاده می شود.

Given an observed data variable $x \in X$, a simple prior probability distribution p_Z on a latent variable $z \in Z$, and a bijection $f : X \to Z$ (with $g = f^{-1}$), the change of variable formula defines a model distribution on X by

$$p_X(x) = p_Z\big(f(x)\big) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \tag{2}$$

$$\log\big(p_X(x)\big) = \log\Big(p_Z\big(f(x)\big)\Big) + \log\left(\left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \right), \tag{3}$$

where $\frac{\partial f(x)}{\partial x^T}$ is the Jacobian of f at x.

شکل ۹ – بخش تغییر متغیر در مقاله

در کد به شکل زیر پیاده سازی می کنیم:

```
1. AffineCheckerboardTransform Layers (Checkerboard Coupling Layers):

for transform in self.transforms_checkered_1:
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian
```

در اینجا، AffineCheckerboardTransform یک نوع لایه کوپلینگ است. در forward pass هر لایه، ورودی z بر اساس پارامترهای لایه، تحت یک تبدیل affine قرار می‌گیرد. تعیین کننده log ژاکوبین این تبدیل در log_det_jacobian_total محاسبه و انباشته می شود. این تعیین کننده گزارش یک جزء کلیدی از فرمول تغییر متغیر است.

```
2. AffineChannelwiseTransform Layers (Channel-wise Coupling Layers):

for transform in self.transforms_channelwise:
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian
```

مشابه حالت قبل، لایه‌های channel-wise coupling یک تبدیل affine به ورودی z اعمال می‌کنند. تعیین کننده log ژاکوبین محاسبه شده و به log_det_jacobian_total اضافه می شود.

```
for transform in self.transforms_checkered_2:
```

```
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian
```

در نهایت، لایه‌های checkerboard coupling اضافی اعمال می‌شوند و لگاریتم دترمینان آن‌ها در log_det_jacobian_total ذخیره می‌شوند.

متغیر log_det_jacobian_total مجموع عوامل تعیین‌کننده log ژاکوبین‌ها را در تمام تبدیل‌ها نشان می‌دهد. در normalizing flows، تابع چگالی احتمال نهایی (pdf متغیر تبدیل‌شده با استفاده از فرمول تغییر متغیر به‌دست می‌آید)، که در آن تعیین‌کننده ژاکوبین تأثیر تبدیل بر چگالی‌های احتمال را محاسبه می‌کند. (روش معکوس) در طول آموزش برای بازسازی داده‌های ورودی استفاده می‌شود، اما فرمول تغییر متغیر عمدتاً در طول عبور به جلو برای مدل‌سازی توزیع احتمال اعمال می‌شود.

- Coupling layers

Affine coupling layer

We will begin the development of the RealNVP architecture with the core bijector that is called the *affine coupling layer*. This bijector can be described as follows: suppose that x is a D-dimensional input, and let $d < D$. Then the output y of the affine coupling layer is given by the following equations:

$$y_{1:d} = x_{1:d} \tag{1}$$
$$y_{d+1:D} = x_{d+1:D} \odot \exp(s(x_{1:d})) + t(x_{1:d}), \tag{2}$$

where s and t are functions from $\mathbb{R}^d \to \mathbb{R}^{D-d}$, and define the log-scale and shift operations on the vector $x_{d+1:D}$ respectively.

The log of the Jacobian determinant for this layer is given by $\sum_j s(x_{1:d})_j$.

The inverse operation can be easily computed as

$$x_{1:d} = y_{1:d} \tag{3}$$
$$x_{d+1:D} = (y_{d+1:D} - t(y_{1:d})) \odot \exp(-s(y_{1:d})), \tag{4}$$

In practice, we will implement equations (1) and (2) using a binary mask b:

$$\text{Forward pass:} \quad y = b \odot x + (1 - b) \odot (x \odot \exp(s(b \odot x)) + t(b \odot x)), \tag{5}$$
$$\text{Inverse pass:} \quad x = b \odot y + (1 - b) \odot (y - t(b \odot x)) \odot \exp(-s(b \odot x)). \tag{6}$$

شکل ۱۰ – **coupling layers**

لایه‌های AffineCheckerboardTransform و AffineChannelwiseTransform نقش‌های متمایزی را در گرفتن وابستگی‌ها در داده‌ها ایفا می‌کنند، که اولی از یک الگوی شطرنجی استفاده می‌کند و دومی از رویکرد کانالی استفاده می‌کند. بلوک‌های ResNet ظرفیت مدل‌سازی این لایه‌های جفت را با ارائه یک معماری عمیق افزایش می‌دهند. این مؤلفه‌ها با هم به انعطاف‌پذیری و بیان مدل RealNVP در گرفتن توزیع‌های پیچیده داده کمک می‌کنند.

۱. AffineCheckerboard Transform:

```python
class AffineCheckerboardTransform(nn.Module):
    def __init__(self, height, width, top_left_zero=False):
        super(AffineCheckerboardTransform, self).__init__()
        self.mask = self.create_mask(height, width, top_left_zero)  # (1,1,height,width)
        self.scale_scale = nn.Parameter(torch.zeros(1), requires_grad=True)
        self.shift_scale = nn.Parameter(torch.zeros(1), requires_grad=True)
        self.net = ResNet()

    def create_mask(self, height, width, top_left_zero):
        mask = (torch.arange(height).view(-1, 1) + torch.arange(width))
        if not top_left_zero:
            mask += 1
        return (mask % 2).unsqueeze(0).unsqueeze(0)

    def forward(self, x, reverse=False):
        self.mask = self.mask.to(x.device)
        # x has size (batch_size, 1, height, width)
        x_masked = x * self.mask
        # log_scale and shift have size (batch_size, 1, height, width)
        log_scale, shift = self.net(x_masked).chunk(2, dim=1)
        log_scale = log_scale.tanh() * self.scale_scale + self.shift_scale

        log_scale = log_scale * (1 - self.mask)
        shift = shift * (1 - self.mask)
        if reverse:
            x = (x - shift) * torch.exp(-log_scale)
            return x
        else:
            x = x * log_scale.exp() + shift
            return x, log_scale
```

AffineCheckerboard Transform - شکل ۱۱

لایه اتصال شطرنجی برای تبدیل زیرمجموعه ای از داده های ورودی بر اساس الگوی شطرنجی طراحی شده است. ایده کلیدی در اینجا این است که زیر مجموعه های مختلف داده های ورودی در یک الگوی متناوب تبدیل می شوند. این به مدل کمک می‌کند تا وابستگی‌ها و همبستگی‌ها را در داده‌ها به‌طور مؤثرتری دریافت کند.

الگوی شطرنجی از طریق یک ماسک باینری که بر اساس ابعاد داده های ورودی ایجاد می شود اعمال می شود. این لایه با استفاده از خروجی ResNet (self.net) یک تبدیل affine را به قسمت پوشانده شده ورودی اعمال می کند. ResNet به مدل سازی روابط پیچیده در داده های پوشانده شده کمک می کند. تعیین کننده ورود به سیستم ژاکوبین تبدیل محاسبه و ذخیره می شود، که برای محاسبه چگالی احتمال متغیر تبدیل شده بسیار مهم است.

نمونه هایی از AffineCheckerboardTransform در لیست های transforms_checkered_1 و transforms_checkered_2 در مدل RealNVP استفاده می شود. این لایه ها به صورت متوالی در forward pass اعمال می شوند.

۲. AffineChannelwiseTransform:

```python
class AffineChannelwiseTransform(nn.Module):
    def __init__(self, top_half_as_input):
        super(AffineChannelwiseTransform, self).__init__()
        self.top_half_as_input = top_half_as_input
        self.scale_scale = nn.Parameter(torch.zeros(1), requires_grad=True)
        self.shift_scale = nn.Parameter(torch.zeros(1), requires_grad=True)
        self.net = ResNet(in_channel=2, out_channel=4)

    def forward(self, x, reverse=False):
        # x.size() is (batch_size, 4, H//2, W//2)
        # fixed, not_fixed have size (batch_size, 2, H//2, W//2)
        if self.top_half_as_input:
            fixed, not_fixed = x.chunk(2, dim=1)
        else:
            not_fixed, fixed = x.chunk(2, dim=1)
        # log_scale and shift have size (batch_size, 2, H//2, W//2)
        log_scale, shift = self.net(fixed).chunk(2, dim=1)
        log_scale = log_scale.tanh() * self.scale_scale + self.shift_scale

        if reverse:
            not_fixed = (not_fixed - shift) * torch.exp(-log_scale)
        else:
            not_fixed = not_fixed * log_scale.exp() + shift

        if self.top_half_as_input:
            x_modified = torch.cat([fixed, not_fixed], dim=1)
            log_scale = torch.cat([log_scale, torch.zeros_like(log_scale)], dim=1)
        else:
            x_modified = torch.cat([not_fixed, fixed], dim=1)
            log_scale = torch.cat([torch.zeros_like(log_scale), log_scale], dim=1)

        if reverse:
            return x_modified
        return x_modified, log_scale
```

شکل ۱۲ - **AffineChannelwiseTransform**

این لایه با هدف تبدیل کانال های مختلف داده های ورودی به طور مستقل است. این به مدل اجازه می‌دهد تا وابستگی‌ها را در هر کانال به طور جداگانه ثبت کند و قابلیت مدل‌سازی انعطاف‌پذیرتری را ارائه دهد.

داده‌های ورودی به دو نیمه تقسیم می‌شوند (بسته به پارامتر top_half_as_input، نصف بالا یا نیمه پایین). یک تبدیل وابسته به نیمی از داده ها بر اساس خروجی ResNet (self.net) اعمال می شود. لگاریتم تعیین کننده ژاکوبین تبدیل محاسبه و ذخیره می شود.

نمونه هایی از AffineChannelwiseTransform در لیست transforms_channelwise در مدل RealNVP استفاده می شود. این لایه ها به صورت متوالی در forward pass اعمال می شوند.

۳. بلوک های ResNet:

```python
class ResNetBlock(nn.Module):
    def __init__(self, dim):
        super(ResNetBlock, self).__init__()
        self.block = nn.Sequential(
            WeightNormConv2d(dim, dim, 1, stride=1, padding=0),
            nn.ReLU(),
            WeightNormConv2d(dim, dim, 3, stride=1, padding=1),
            nn.ReLU(),
            WeightNormConv2d(dim, dim, 1, stride=1, padding=0),
        )

    def forward(self, x):
        return x + self.block(x)


class ResNet(nn.Module):
    def __init__(self, in_channel=1, out_channel=2, intermediate_channel=128, num_blocks=4):
        super(ResNet, self).__init__()
        layers = [
            WeightNormConv2d(in_channel, intermediate_channel, 3, stride=1, padding=1),
            nn.ReLU(),
        ]
        for _ in range(num_blocks):
            layers.append(ResNetBlock(intermediate_channel))
        layers.append(nn.ReLU())
        layers.append(WeightNormConv2d(intermediate_channel, out_channel, 3, stride=1, padding=1))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

شکل ۱۳ - بلوک های **ResNet**

بلوک های ResNet به عنوان بلوک های سازنده لایه های جفت AffineCheckerboardTransform و AffineChannelwiseTransform استفاده می شوند. آنها یک مدل عمیق و انعطاف پذیر برای ثبت وابستگی های پیچیده در داده ها ارائه می دهند.

هر بلوک ResNet از سه لایه کانولوشن با فعال سازی ReLU تشکیل شده است. این لایه ها به مدل کمک می کنند تا تبدیل های پیچیده داده های ورودی را یاد بگیرد. بلوک های ResNet در کلاس های AffineCheckerboardTransform و AffineChannelwiseTransform استفاده می شوند.

- convolutional layers with normalized weights

```python
class WeightNormConv2d(nn.Module):
    def __init__(self, in_channel, out_channel, kernel_size, stride=1, padding=0, bias=True):
        super(WeightNormConv2d, self).__init__()
        self.conv = nn.utils.weight_norm(
            nn.Conv2d(in_channel, out_channel, kernel_size, stride=stride, padding=padding, bias=bias)
        )

    def forward(self, x):
        return self.conv(x)
```

convolutional layers with normalized weights - شکل ۱۴

در این کلاس، ماژول nn.Conv2d با nn.utils.weight_norm استفاده شده است تا یک لایه کانولوشنال با وزن نرمال سازی شود. نرمال سازی وزن در طول تمرین روی وزنه های لایه کانولوشن اعمال می شود

- Combining coupling layers:

1. Checkerboard Pattern Coupling Layers (Section 3.2):

```
self.transforms_checkered_1 = nn.ModuleList([
    AffineCheckerboardTransform(height, width, False),
    ActNorm(1),
    AffineCheckerboardTransform(height, width, True),
    ActNorm(1),
    AffineCheckerboardTransform(height, width, False),
    ActNorm(1),
    AffineCheckerboardTransform(height, width, True)
])
```

در اینجا، nn.ModuleList شامل دنباله ای از لایه های جفت شده با الگوهای شطرنجی است. هر لایه به طور متناوب بین یک لایه AffineCheckerboardTransform و یک لایه ActNorm قرار می گیرد. الگوی شطرنجی وابستگی های ساختار یافته را معرفی می کند.

2. Channel-wise Coupling Layers (Section 3.3):

```
self.transforms_channelwise = nn.ModuleList([
    AffineChannelwiseTransform(True),
    ActNorm(4),
    AffineChannelwiseTransform(False),
    ActNorm(4),
    AffineChannelwiseTransform(True),
])
```

مشابه حالت قبل، این لیست ماژول حاوی لایه های coupling است، اما این بار آنها channel-wise. هستند. AffineChannelwiseTransform با لایه های ActNorm جایگزین می شود. این نوع پوشش به ویژه زمانی موثر است که وابستگی های channel-specific در داده ها وجود داشته باشد.

3. Additional Checkerboard Pattern Coupling Layers (Section 3.2):

```
self.transforms_checkered_2 = nn.ModuleList([
    AffineCheckerboardTransform(height, width, False),
    ActNorm(1),
    AffineCheckerboardTransform(height, width, True),
    ActNorm(1),
    AffineCheckerboardTransform(height, width, False)
])
```

مجموعه نهایی لایه های جفت با الگوهای شطرنجی. این لایه ها ظرفیت تبدیل اضافی را فراهم می کنند.

- ترکیب عملیات در Forward Pass:

در روش فورورد مدل RealNVP، این لایه های کوپلینگ به صورت متوالی اعمال می شوند. بعد از هر مجموعه از لایه ها، ابعاد فضایی ورودی با استفاده از عملیات فشردن و فشار دادن تنظیم می شود.

```
# Section 3.2: Coupling layers with checkerboard pattern
for transform in self.transforms_checkered_1:
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian

z = self.squeeze(z)
log_det_jacobian_total = self.squeeze(log_det_jacobian_total)

# Section 3.3: Coupling layers with channel-wise masking
for transform in self.transforms_channelwise:
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian

z = self.unsqueeze(z)
log_det_jacobian_total = self.unsqueeze(log_det_jacobian_total)

# Section 3.2: More coupling layers with checkerboard pattern
for transform in self.transforms_checkered_2:
    z, log_det_jacobian = transform(z)
    log_det_jacobian_total += log_det_jacobian
```

این عملیات تضمین می‌کند که مدل وابستگی‌ها را در داده‌ها از طریق ترکیبی از الگوی شطرنجی و تبدیل‌های کانال دریافت می‌کند. متغیر log_det_jacobian_total تعیین کننده log ژاکوبین را از هر تبدیل جمع می کند، که برای برگشت پذیری مدل بسیار مهم است.

- Inverse Pass:

روش معکوس تبدیل ها را به ترتیب معکوس معکوس می کند، از آخرین مجموعه لایه های جفت الگوی شطرنجی شروع می شود، سپس لایه های کانالی، و در نهایت مراحل اولیه پیش پردازش

```
# Section 3.2: More coupling layers with checkerboard pattern
(inverse)
for transform in self.transforms_checkered_2[::-1]:
    z = transform(z, reverse=True)

z = self.squeeze(z)

# Section 3.3: Coupling layers with channel-wise masking (inverse)
for transform in self.transforms_channelwise[::-1]:
```

```
    z = transform(z, reverse=True)

z = self.unsqueeze(z)

# Section 3.2: Coupling layers with checkerboard pattern (inverse)
for transform in self.transforms_checkered_1[::-1]:
    z = transform(z, reverse=True)

# Section 1: Preprocess (inverse)
z = self.preprocess(z, reverse=True)
```

این فرآیند وارونگی به مدل اجازه می دهد تا داده های اصلی را از نمایش تبدیل شده بازسازی کند. ترکیب این لایه‌های جفت، همراه با اجزای دیگر مانند normalization و بلوک‌های ResNet، RealNVP را قادر می‌سازد تا توزیع داده‌های پیچیده را به طور موثر مدل‌سازی کند.

- عملیات Squeeze and Unsqueeze:

این عملیات برای تغییر شکل تانسور ورودی استفاده می شود. فشردن ابعاد فضایی را ترکیب می کند و فشردن آن این عملیات را معکوس می کند. این عملیات به ویژه در هنگام برخورد با داده های تصویر، که در آن ساختار فضایی نیاز به دستکاری دارد، مرتبط هستند.

```
def squeeze(self, x):
    '''converts a (batch_size,1,4,4) tensor into a (batch_size,4,2,2) tensor'''
    batch_size, num_channels, height, width = x.size()
    x = x.reshape(batch_size, num_channels, height // 2, 2, width // 2, 2)
    x = x.permute(0, 1, 3, 5, 2, 4)
    x = x.reshape(batch_size, num_channels * 4, height // 2, width // 2)
    return x

def unsqueeze(self, x):
    '''converts a (batch_size,4,2,2) tensor into a (batch_size,1,4,4) tensor'''
    batch_size, num_channels, height, width = x.size()
    x = x.reshape(batch_size, num_channels // 4, 2, 2, height, width)
    x = x.permute(0, 1, 4, 2, 5, 3)
    x = x.reshape(batch_size, num_channels // 4, height * 2, width * 2)
    return x
```

شکل ۱۵ – کد **Squeeze and Unsqueeze**

- Loss function:

```
def loss_function(target_distribution, z, log_det_jacobian):
    log_likelihood = target_distribution.log_prob(z) +
log_det_jacobian
    return -log_likelihood.mean()
```

تابع loss برای آموزش یک normalizing flow، مانند RealNVP، معمولا بر اساس احتمال داده های مشاهده شده تحت توزیع تبدیل شده است. در normalizing flow، لگاریتم احتمال منفی NLL یک انتخاب رایج به عنوان تابع هزینه است.

کد ترین مدل به شرح زیر است:

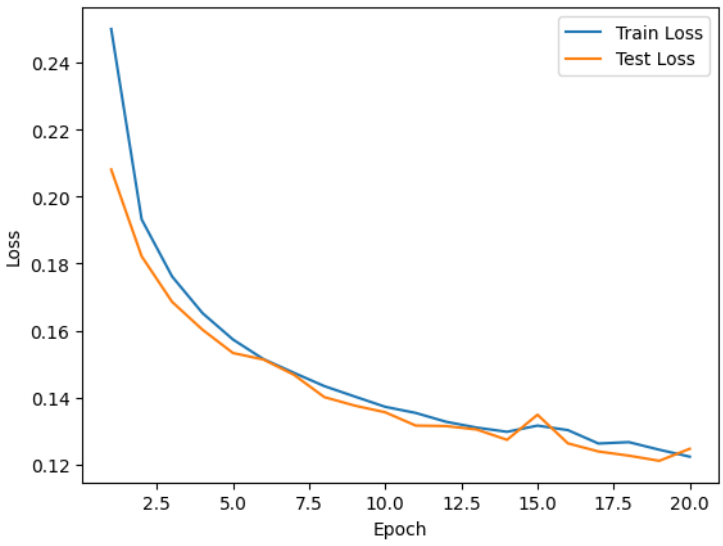
در داخل حلقه آموزشی، مدل روی حالت آموزش (realnvp_model.train()) تنظیم می شود و بهینه ساز صفر می شود. forward pass برای به دست آوردن پیش بینی z و تعیین کننده ورود به سیستم کل ماتریس ژاکوبین انجام می شود. یک توزیع نرمال به عنوان توزیع هدف ایجاد می شود.loss با استفاده از تابع هزینه محاسبه می شود و پس انتشار انجام می شود. مراحل مشابهی برای ارزیابی روی مجموعه آزمایشی انجام می شود، اما بدون انتشار مجدد.

```
# Define the dataset and data loaders
transform = transforms.Compose([
    transforms.ToTensor(),
])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=128, shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=128, shuffle=False, num_workers=2)

# Initialize the RealNVP model and move it to GPU
height, width = 28, 28  # MNIST image dimensions
realnvp_model = RealNVP(height, width).to("cuda")

# Define the optimizer
optimizer = optim.Adam(realnvp_model.parameters(), lr=1e-4)

# Custom loss function
def loss_function(target_distribution, z, log_det_jacobian):
```

```
    log_likelihood = target_distribution.log_prob(z) +
log_det_jacobian
    return -log_likelihood.mean()

# Training loop with tqdm
num_epochs = 20

train_losses = []
test_losses = []

for epoch in range(num_epochs):
    realnvp_model.train()
    train_loss_epoch = 0.0

    with tqdm(total=len(train_loader), desc=f'Epoch {epoch +
1}/{num_epochs}', unit='batch') as pbar:
        for batch_idx, (data, _) in enumerate(train_loader):
            optimizer.zero_grad()

            # Move data to GPU
            data = data.to("cuda")

            # data = data[:, 0, :, :].unsqueeze(1)  # Use only one
channel (grayscale)
            z, log_det_jacobian_total = realnvp_model(data)

            # Create a normal distribution for the target
            target_distribution = Normal(torch.zeros_like(z),
torch.ones_like(z))

            loss = loss_function(target_distribution, z,
log_det_jacobian_total)
            loss.backward()
            optimizer.step()

            train_loss_epoch += loss.item()

            pbar.update(1)
            pbar.set_postfix({'Loss': train_loss_epoch / (batch_idx +
1)})

    train_loss_epoch /= len(train_loader)
    train_losses.append(train_loss_epoch)

    # Evaluate on the test set
    realnvp_model.eval()
    test_loss_epoch = 0.0
```

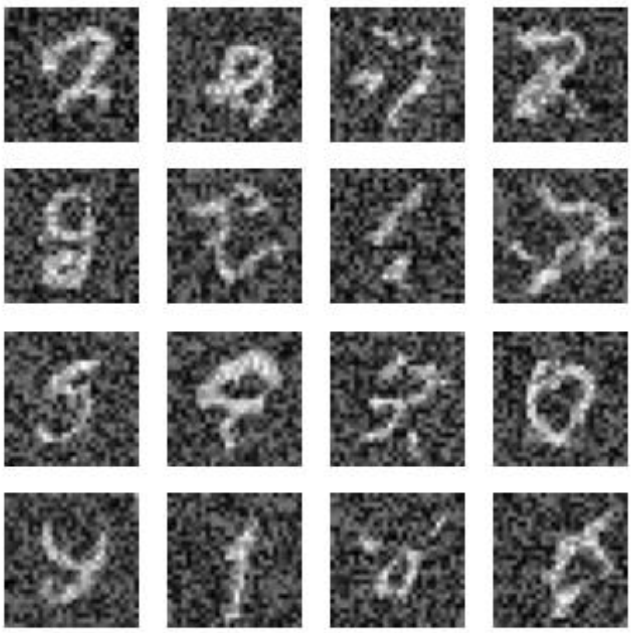
```
    with torch.no_grad(), tqdm(total=len(test_loader), desc=f'Test
Loss', unit='batch') as pbar:
        for data, _ in test_loader:
            # Move data to GPU
            data = data.to("cuda")

            z, log_det_jacobian_total = realnvp_model(data)

            # Create a normal distribution for the target
            target_distribution = Normal(torch.zeros_like(z),
torch.ones_like(z))

            loss = loss_function(target_distribution, z,
log_det_jacobian_total)
            test_loss_epoch += loss.item()

            pbar.update(1)

    test_loss_epoch /= len(test_loader)
    test_losses.append(test_loss_epoch)

    print(f'Epoch {epoch + 1}/{num_epochs}, Train Loss:
{train_loss_epoch:.4f}, Test Loss: {test_loss_epoch:.4f}')
```

نتایج:



شکل ۱۶ – **train-val loss**

مشاهده میکنیم که تابع خطا آموزش و تست روند نزولی داشته است.



شکل ۱۷ - خروجی مدل

خروجی مدل را در بالا مشاهده میکنیم. در خروجی بعضی از اعدا همانند ۲-۰-۹-۵-۱ به خوبی نمایان هستند.

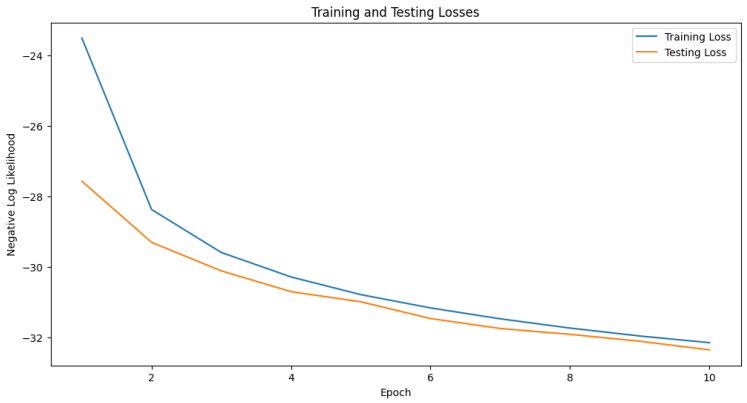
مدلی که در بالا پیاده سازی کردیم، مدل به نسبت پیچیده ای بود، در ادامه مدل ساده تری را پیاده سازی میکنیم:

```
class CouplingLayer(nn.Module):
    def __init__(self, input_dim, output_dim, hid_dim, mask):
        super().__init__()
        self.s_fc1 = nn.Linear(input_dim, hid_dim)
        self.s_fc2 = nn.Linear(hid_dim, hid_dim)
        self.s_fc3 = nn.Linear(hid_dim, output_dim)
        self.t_fc1 = nn.Linear(input_dim, hid_dim)
        self.t_fc2 = nn.Linear(hid_dim, hid_dim)
        self.t_fc3 = nn.Linear(hid_dim, output_dim)
        self.mask = mask

    def forward(self, x):
        x_m = x * self.mask
        s_out =
torch.tanh(self.s_fc3(F.relu(self.s_fc2(F.relu(self.s_fc1(x_m))))))
        t_out =
self.t_fc3(F.relu(self.t_fc2(F.relu(self.t_fc1(x_m)))))
        y = x_m + (1-self.mask)*(x*torch.exp(s_out)+t_out)
        log_det_jacobian = s_out.sum(dim=1)
```

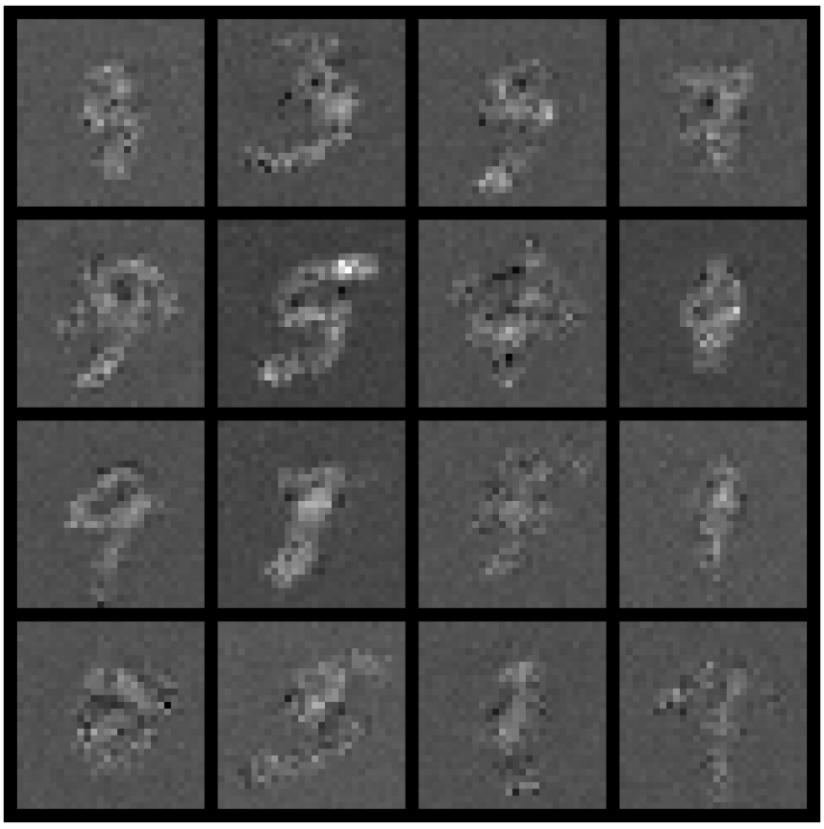
```
        return y, log_det_jacobian

    def backward(self, y):
        y_m = y * self.mask
        s_out =
torch.tanh(self.s_fc3(F.relu(self.s_fc2(F.relu(self.s_fc1(y_m))))))
        t_out =
self.t_fc3(F.relu(self.t_fc2(F.relu(self.t_fc1(y_m)))))
        x = y_m + (1-self.mask)*(y-t_out)*torch.exp(-s_out)
        return x


class RealNVP(nn.Module):
    def __init__(self, input_dim, output_dim, hid_dim, mask, n_layers
= 6):
        super().__init__()
        assert n_layers >= 2, 'num of coupling layers should be
greater or equal to 2'

        self.modules = []
        self.modules.append(CouplingLayer(input_dim, output_dim,
hid_dim, mask))
        for _ in range(n_layers-2):
            mask = 1 - mask
            self.modules.append(CouplingLayer(input_dim, output_dim,
hid_dim, mask))
        self.modules.append(CouplingLayer(input_dim, output_dim,
hid_dim, 1 - mask))
        self.module_list = nn.ModuleList(self.modules)

    def forward(self, x):
        ldj_sum = 0 # sum of log determinant of jacobian
        for module in self.module_list:
            x, ldj= module(x)
            ldj_sum += ldj
        return x, ldj_sum

    def backward(self, z):
        for module in reversed(self.module_list):
            z = module.backward(z)
        return z

mask = torch.from_numpy(np.zeros(INPUT_DIM).astype(np.float32))
mask[:INPUT_DIM//2] = 1.0
model = RealNVP(INPUT_DIM, OUTPUT_DIM, HIDDEN_DIM, mask,
N_COUPLE_LAYERS)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
```

```
prior_z = distributions.MultivariateNormal(torch.zeros(INPUT_DIM),
torch.eye(INPUT_DIM))
```

خروجی در این حالت:



شکل ۱۸- **train – test loss**



شکل ۱۹ – خروجی مدل

در این حالت با توجه به اینکه مقدار loss قابل قبولی نداریم ولی خروجی خوبی میگیریم.