Variational Autoencoder (VAE) Cheat Sheet

1 Key Concepts

1. **Encoder**: Maps input data x to a distribution over the latent variable z. The encoder produces:
   - $\mu(x)$: Mean of the latent variable z.
   - $\log(\sigma^2(x))$: Log-variance of z.
2. **Decoder**: Maps the latent variable z back to the input space to reconstruct x.
3. **Reparameterization Trick**: Allows differentiable sampling from the latent space:

$$z = \mu(x) + \sigma(x) \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

   Ensures backpropagation through the sampling process by transforming random sampling into a differentiable operation.
4. **KL Divergence**: Measures how close the learned distribution $q(z|x)$ is to the prior $p(z)$ (typically a standard normal distribution):

$$D_{\mathrm{KL}}(q(z|x) \parallel p(z)) = \frac{1}{2} \sum_j \left(1 + \log(\sigma_j^2) - \sigma_j^2 - \mu_j^2\right)$$

5. **Reconstruction Loss**: Measures how well the decoder can reconstruct the original input x. Typically binary cross-entropy for image data:

$$\text{Reconstruction Loss} = \sum_i \text{BCE}(x_i, \hat{x}_i)$$

6. **Evidence Lower Bound (ELBO)**: The objective function maximized in VAEs. It is the sum of:
   - **Reconstruction term**: Measures how well the decoder reconstructs the input.
   - **KL divergence term**: Regularizes the latent space to be close to the prior distribution.

   The ELBO is a **lower bound** on the marginal likelihood of the data:

$$\log p(x) \geq \mathcal{L}(x) = \mathbb{E}_{q(z|x)}[\log p(x|z)] - D_{\mathrm{KL}}(q(z|x) \parallel p(z))$$

2 Detailed Derivations

2.1 ELBO Derivation

The **Evidence Lower Bound (ELBO)** is derived by introducing an approximate posterior $q(z|x)$ and applying **Jensen's Inequality**.

2.1.1 Step-by-Step Derivation

We want to maximize the **marginal likelihood** of the data x:

$$p(x) = \int p(x, z)\, dz = \int p(x|z)p(z)\, dz$$

However, this integral is often intractable, so we introduce an approximate posterior $q(z|x)$, which gives:

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

Now, applying **Jensen's Inequality** to the logarithm of the marginal likelihood:

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

The inequality gives the **ELBO**:

$$\log p(x) \geq \mathbb{E}_{q(z|x)} \left[\log \frac{p(x,z)}{q(z|x)} \right]$$

Breaking down the expression:

$$\mathcal{L}(x) = \mathbb{E}_{q(z|x)}[\log p(x|z)] - D_{\mathrm{KL}}(q(z|x) \parallel p(z))$$

- **First term** $\mathbb{E}_{q(z|x)}[\log p(x|z)]$: Measures how well the decoder reconstructs the input.
- **Second term** $D_{\mathrm{KL}}(q(z|x) \parallel p(z))$: Regularizes the approximate posterior to stay close to the prior.

Thus, the **ELBO** is:

$$\mathcal{L}(x) = \text{Reconstruction Loss} - \text{KL Divergence}$$

2.2 KL Divergence Derivation (Detailed)

The **KL divergence** between two Gaussian distributions $q(z|x) = \mathcal{N}(\mu_q, \Sigma_q)$ and $p(z) = \mathcal{N}(0, I)$ is a critical part of the VAE loss function.

2.2.1 General KL Divergence Formula

The KL divergence between two distributions $q(z)$ and $p(z)$ is:

$$D_{\mathrm{KL}}(q(z) \parallel p(z)) = \int q(z) \log \frac{q(z)}{p(z)} \, dz$$

For two multivariate Gaussian distributions:

- $q(z|x) = \mathcal{N}(\mu_q, \Sigma_q)$
- $p(z) = \mathcal{N}(0, I)$

The KL divergence between these two Gaussians has a **closed-form solution**.

2.2.2 Step-by-Step KL Divergence Derivation

The probability density function for a Gaussian $\mathcal{N}(\mu, \Sigma)$ is:

$$p(z) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2} (z-\mu)^T \Sigma^{-1} (z-\mu) \right)$$

Substitute the densities of $q(z|x)$ and $p(z)$ into the KL divergence formula:

$$D_{\mathrm{KL}}(q(z|x) \parallel p(z)) = \mathbb{E}_{q(z|x)} \left[\log \frac{q(z|x)}{p(z)} \right]$$

Simplifying:

$$D_{\mathrm{KL}}(q(z|x) \parallel p(z)) = \frac{1}{2} \left(\mathrm{tr}(\Sigma_q) + \mu_q^T \mu_q - d - \log |\Sigma_q| \right)$$

In the case where Σ_q is diagonal, i.e., $\Sigma_q = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_d^2)$, the trace and determinant simplify:

$$D_{\mathrm{KL}}(q(z|x) \parallel p(z)) = \frac{1}{2} \sum_j \left(1 + \log(\sigma_j^2) - \sigma_j^2 - \mu_j^2 \right)$$

This is the **closed-form solution** for the KL divergence used in VAEs.

2.3 Reparameterization Trick Derivation

2.3.1 Why Do We Need the Reparameterization Trick?

Sampling from $q(z|x)$ is a **stochastic** process and breaks backpropagation, which requires deterministic operations to compute gradients.

The **reparameterization trick** allows us to sample z in a way that is differentiable:

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Where ϵ is random noise sampled from a standard normal distribution $\mathcal{N}(0, I)$, and μ and σ are the outputs of the encoder.

2.3.2 Derivation

Consider the latent variable z sampled from $q(z|x) = \mathcal{N}(\mu(x), \Sigma(x))$. To ensure the sampling is differentiable, we use the reparameterization trick:

1. Compute $\mu(x)$ and $\log(\sigma^2(x))$ from the encoder.
2. Sample $\epsilon \sim \mathcal{N}(0, I)$.
3. Compute z as:
$$z = \mu(x) + \sigma(x) \cdot \epsilon$$

Where $\sigma(x) = \exp(0.5 \cdot \log(\sigma^2(x)))$.

This formulation ensures that z is differentiable with respect to $\mu(x)$ and $\sigma(x)$, allowing for gradient-based optimization.

3 VAE Loss Function

The loss function for VAEs consists of two parts:

1. **Reconstruction Loss**:
$$\text{Reconstruction Loss} = -\mathbb{E}_{q(z|x)}[\log p(x|z)]$$
For binary data (e.g., images), use binary cross-entropy (BCE).
2. **KL Divergence**:
$$D_{\text{KL}}(q(z|x) \parallel p(z)) = \frac{1}{2} \sum_j \left(1 + \log(\sigma_j^2) - \sigma_j^2 - \mu_j^2\right)$$
3. **Total Loss**:
$$\text{VAE Loss} = \text{Reconstruction Loss} + \text{KL Divergence}$$

4 Complete PyTorch Implementation

Here's a complete PyTorch implementation of a basic VAE:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()

        # Encoder
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)
```

```
        # Decoder
        self.fc2 = nn.Linear(latent_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, input_dim)

    def encode(self, x):
        h = F.relu(self.fc1(x))
        return self.fc_mu(h), self.fc_logvar(h)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h = F.relu(self.fc2(z))
        return torch.sigmoid(self.fc3(h))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

# Loss function
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

# Training function
def train(epoch, model, train_loader, optimizer, device):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)} '
                  f'({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item() / len(data):.6f}
    print(f'====> Epoch: {epoch} Average loss: {train_loss / len(train_loader.dataset):.4f}')

# Main training loop
def main():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # MNIST Dataset
    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTen
    train_loader = DataLoader(dataset=train_dataset, batch_size=128, shuffle=True)

    # Initialize model and optimizer
    model = VAE(input_dim=784, hidden_dim=400, latent_dim=20).to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-3)

    # Train the model
    num_epochs = 10
    for epoch in range(1, num_epochs + 1):
```

```
        train(epoch, model, train_loader, optimizer, device)

if __name__ == "__main__":
    main()
```

This implementation includes:

- A complete `VAE` class with encoder, decoder, and reparameterization trick.
- The loss function combining reconstruction loss (binary cross-entropy) and KL divergence.
- A training function that processes batches of data.
- A main function that sets up the MNIST dataset, initializes the model and optimizer, and runs the training loop.

To use this implementation:

1. Ensure PyTorch and torchvision are installed.
2. Copy the code into a Python file (e.g., `vae_mnist.py`).
3. Run the script. It will download the MNIST dataset and train the VAE for 10 epochs.

This basic implementation can be easily modified for different datasets or architectures. For example, you could replace the fully connected layers with convolutional layers for image data, or adjust the dimensions for different datasets.

```
def vae_loss(recon_x, x, mu, logvar):
    # Reconstruction loss (binary cross-entropy)
    recon_loss = F.binary_cross_entropy(recon_x, x, reduction='sum')

    # KL divergence
    kl_divergence = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return recon_loss + kl_divergence
```

5 Additional Practical Tips

1. **Choosing the latent dimension**:
   - Start with a smaller dimension (e.g., 2-10) for simpler datasets.
   - Increase for more complex data, but beware of overfitting.
   - Use dimensionality reduction techniques like PCA for guidance.
2. **Annealing the KL term**:
   - Gradually increase the weight of the KL term during training.
   - Helps prevent the "posterior collapse" problem.
   - Example: KL weight = min(1, current_epoch / annealing_epochs)
3. **Handling imbalanced reconstruction vs KL loss**:
   - Use β-VAE approach: multiply KL term by β (usually $\beta > 1$).
   - Helps in learning more disentangled representations.
4. **Choosing the prior distribution**:
   - Standard choice: $\mathcal{N}(0, I)$ (standard normal distribution).
   - Consider using more complex priors for specific tasks.
5. **Handling continuous vs discrete data**:
   - Continuous: Use Gaussian distribution for reconstruction.

- Discrete/Binary: Use Bernoulli distribution for reconstruction.

6. **Evaluating VAE performance**:
   - Reconstruction quality (e.g., MSE, SSIM for images).
   - KL divergence value.
   - FID score for generated samples (for image data).
   - Disentanglement metrics (e.g., β-VAE metric).
7. **Debugging tips**:
   - Check if KL divergence is non-zero (avoid posterior collapse).
   - Ensure reconstruction loss is decreasing.
   - Visualize reconstructions and samples regularly during training.
8. **Variants to consider**:
   - β-VAE: For learning disentangled representations.
   - Conditional VAE: When you have labeled data.
   - VQ-VAE: For learning discrete latent representations.
9. **Hyperparameter tuning**:
   - Learning rate: Usually in range 1e-4 to 1e-3.
   - Batch size: Start with 32 or 64, adjust based on GPU memory.
   - Network architecture: Experiment with different encoder/decoder architectures.
10. **Practical applications**:
    - Image generation and manipulation.
    - Anomaly detection.
    - Data compression.
    - Feature learning for downstream tasks.

Remember, VAEs are powerful but can be sensitive to hyperparameters and architecture choices. Experimentation and careful monitoring during training are key to success.

6 Summary

- **ELBO**: Measures how well the model reconstructs data and how close the latent space is to the prior.
- **KL Divergence**: Ensures the learned latent space stays close to the prior distribution.
- **Reparameterization Trick**: Enables backpropagation through the sampling process.
- **Loss Function**: Combines reconstruction loss and KL divergence to form the VAE objective.

This This detailed cheat sheet includes both the **concepts** and **derivations** essential for understanding and implementing VAEs.

7 Advanced Topics

7.1 Conditional VAE (CVAE)

CVAEs incorporate additional information (e.g., labels) into the encoding and decoding processes.

- Encoder: $q(z|x, y)$
- Decoder: $p(x|z, y)$
- ELBO: $\mathcal{L}(x, y) = \mathbb{E}_{q(z|x,y)}[\log p(x|z, y)] - D_{\mathrm{KL}}(q(z|x, y) \parallel p(z|y))$

7.2 β-VAE

Introduces a hyperparameter β to control the trade-off between reconstruction quality and disentanglement of latent representations.

$$\mathcal{L}_\beta = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \beta \cdot D_{\mathrm{KL}}(q(z|x) \parallel p(z))$$

7.3 VQ-VAE (Vector Quantized VAE)

Uses discrete latent variables by incorporating a vector quantization step.

- Replace continuous latent space with a discrete codebook
- Loss function includes a commitment loss to encourage encoder outputs to stay close to codebook entries

7.4 Disentanglement in VAEs

Disentanglement refers to learning latent representations where individual latent units correspond to independent factors of variation in the data.

- Metrics: β-VAE metric, DCI metric, FactorVAE score
- Techniques: β-VAE, FactorVAE, DIP-VAE

8 Challenges and Open Problems

1. **Posterior collapse**: When the latent variables are ignored, and the model behaves like a standard autoencoder.
2. **Blurry reconstructions**: Especially problematic in image generation tasks.
3. **Balancing reconstruction and KL terms**: Finding the right trade-off can be challenging.
4. **Evaluating generative models**: Lack of universally accepted metrics for all scenarios.
5. **Scaling to high-dimensional data**: Computational challenges with very large datasets or high-dimensional inputs.

9 Recent Developments

- **Normalizing Flows**: Enhancing the expressiveness of the approximate posterior.
- **VAE-GAN hybrids**: Combining VAEs with Generative Adversarial Networks for improved sample quality.
- **Hierarchical VAEs**: Building more complex latent variable models with multiple levels of abstraction.
- **Sparse VAEs**: Inducing sparsity in the latent space for improved interpretability.
- **Adversarial Autoencoders**: Using adversarial training to match the aggregated posterior to the prior.

10 Conclusion

Variational Autoencoders represent a powerful framework in generative modeling and unsupervised learning. By combining the strengths of variational inference and deep learning, VAEs offer a principled approach to learning complex data distributions and meaningful latent representations. As the field continues to evolve, VAEs and their variants remain at the forefront of research in machine learning and artificial intelligence.