Autoencodes

- 无监督特征学习

Autoencoders

- 自编码器是能够在无监督的情况下学习输入数据的有效表示（叫做编码）的人工神经网络。这些编码通常具有比输入数据低得多的维度，使得自编码器对降维有用。更重要的是，自编码器可以作为强大的特征检测器，它们可以用于无监督的深度神经网络预训练。最后，他们能够随机生成与训练数据非常相似的新数据；这被称为生成模型。例如，可以在脸部图片数据集上训练自编码器，然后生成新的脸部图片。

Autoencoders

- 自编码器只需学习将输入复制到其输出即可工作。同时也可以用各种方式约束网络让学习过程变得更具挑战性。例如，可以限制内部表示的大小，或者可以向输入添加噪声并训练网络以恢复原始（无噪声）输入。这些约束防止自编码器将输入直接复制到输出，这迫使它学习表示数据的有效方法。简言之，编码是自编码器在某些限制条件下尝试学习恒等函数的副产品。

Autoencoders

- 在本节我们将更深入地解释自编码器如何工作，可以施加什么类型的约束以及如何使用 代码实现它们，无论是用来降维，特征提取，无监督预训练还是作为生成式模型。

有效的数据表示

- William Chase 和 Herbert Simon 在20 世纪 70 年代早期研究过记忆，感知和模式匹配之间的关系，他们观察到，专家棋手用5秒钟观看棋盘就能够记忆所有棋子的位置，这是大多数人不可能完成的任务。 然而，只有当这些棋子被放置在实际棋局位置（来自实际比赛）时才是这种情况，随机放置棋子专家就记不住了。 国际象棋专家其实并没有比平常人更好的记忆，他们只是能注意到国际象棋的模式，这要归功于他们的比赛经验，注意这些模式能帮他们有效地记忆信息。

有效的数据表示

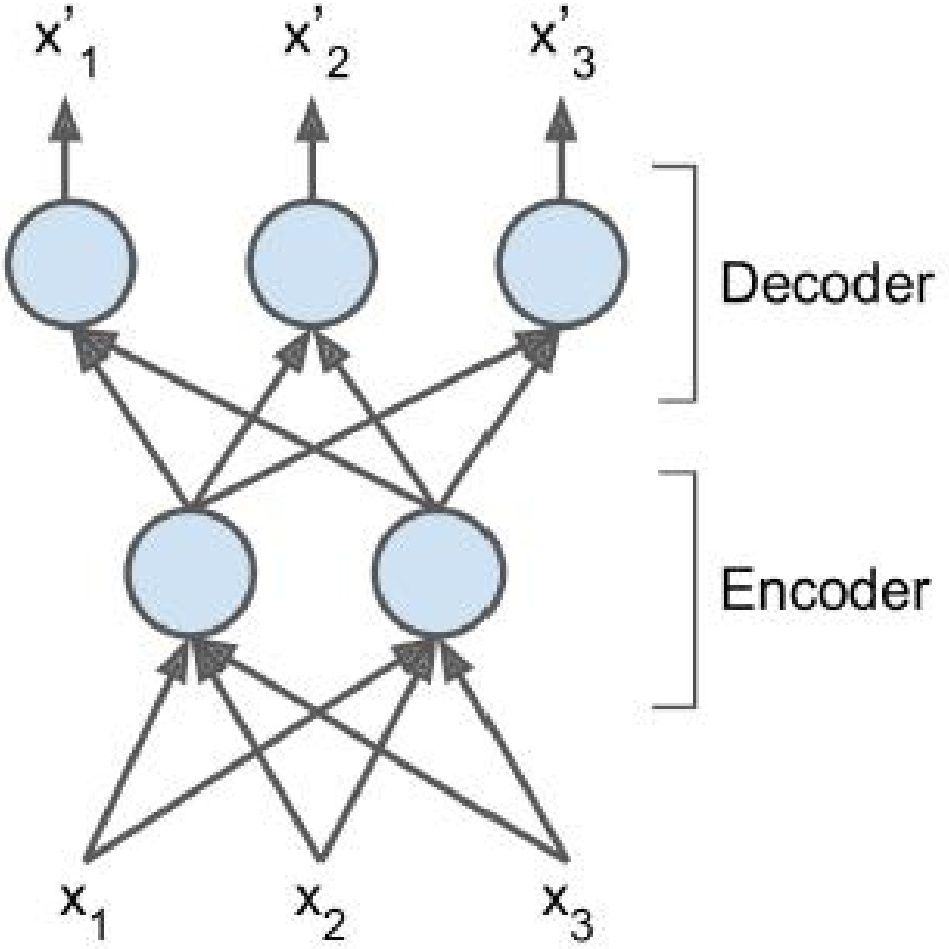
- 就像这个记忆实验中的象棋棋手一样，一个自编码器会查看输入信息，将它们转换为高效的内部表示形式，然后吐出一些（希望）看起来非常接近输入的东西。 自编码器总是由两部分组成：将输入转换为内部表示的编码器（或识别网络），然后是将内部表示转换为输出的解码器（或生成网络）。

有效的数据表示

Outputs
(≈ Inputs)

Internal
representation

Inputs



有效的数据表示

- 由于内部表示具有比输入数据更低的维度（2D 而不是 3D），所以自编码器的中间层是不完整的。 不完整的自编码器不能简单地将其输入复制到编码，但它必须找到一种方法来输出其输入的副本。 它会被迫学习输入数据中最重要的特征（并删除不重要的特征）。

用不完整的线性自编码器执行 PCA

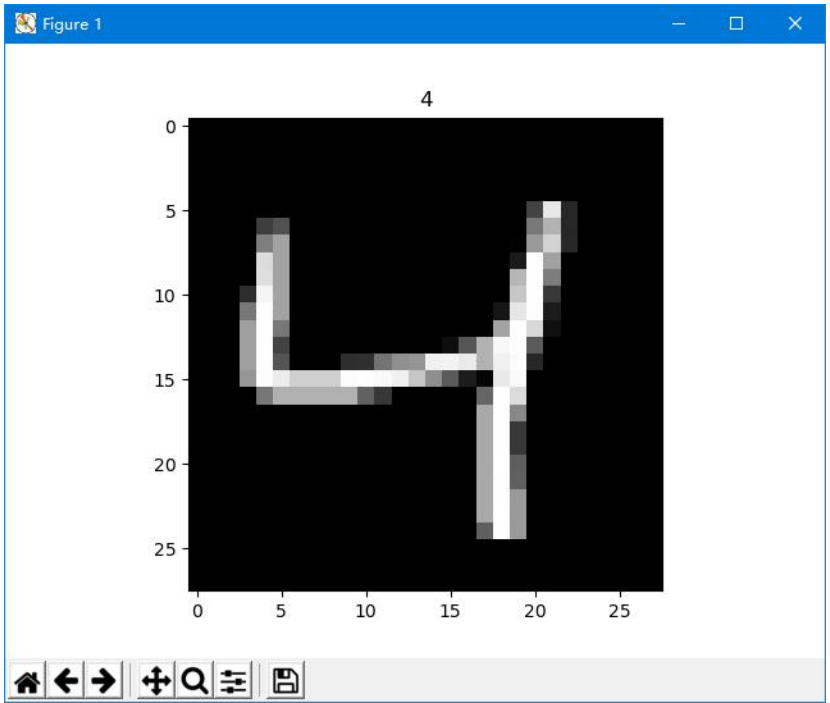
- 如果自编码器仅使用线性激活并且损失函数是均方误差（MSE），它会最终执行主成分分析。

简单自编码器实现

```
EPOCH = 10
BATCH_SIZE = 64
LR = 0.005         # learning rate
DOWNLOAD_MNIST = True
N_TEST_IMG = 5
# Mnist digits dataset
train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,                                 # this is training data
    transform=torchvision.transforms.ToTensor(),
    # Converts a PIL.Image or numpy.ndarray to torch.FloatTensor of shape (C x H x W)
    download=DOWNLOAD_MNIST,                    # download it if you don't have it
)
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE,
shuffle=True)
```

简单自编码器实现

```
# plot one example
print(train_data.train_data.size())     # (60000, 28, 28)
print(train_data.train_labels.size())   # (60000)
plt.imshow(train_data.train_data[2].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[2])
plt.show()
```



```
class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.Tanh(),
            nn.Linear(128, 64),
            nn.Tanh(),
            nn.Linear(64, 12),
            nn.Tanh(),
            nn.Linear(12, 3),   # compress to 3 features which can be visualized in plt
        )
```

```
    self.decoder = nn.Sequential(
        nn.Linear(3, 12),
        nn.Tanh(),
        nn.Linear(12, 64),
        nn.Tanh(),
        nn.Linear(64, 128),
        nn.Tanh(),
        nn.Linear(128, 28*28),
        nn.Sigmoid(),       # compress to a range (0, 1)
    )

def forward(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return encoded, decoded
```

```
autoencoder = AutoEncoder()

optimizer = torch.optim.Adam(autoencoder.parameters(), lr=LR)
loss_func = nn.MSELoss()

# initialize figure
f, a = plt.subplots(2, N_TEST_IMG, figsize=(5, 2))
plt.ion()   # continuously plot

# original data (first row) for viewing
view_data = train_data.train_data[:N_TEST_IMG].view(-1,
28*28).type(torch.FloatTensor)/255.
for i in range(N_TEST_IMG):
    a[0][i].imshow(np.reshape(view_data.data.numpy()[i], (28, 28)), cmap='gray');
    a[0][i].set_xticks(());
    a[0][i].set_yticks(())
```

```
for epoch in range(EPOCH):
    for step, (x, b_label) in enumerate(train_loader):
        b_x = x.view(-1, 28*28)   # batch x, shape (batch, 28*28)
        b_y = x.view(-1, 28*28)   # batch y, shape (batch, 28*28)
        encoded, decoded = autoencoder(b_x)
        loss = loss_func(decoded, b_y)      # mean square error
        optimizer.zero_grad()               # clear gradients for this training step
        loss.backward()                     # backpropagation, compute gradients
        optimizer.step()                    # apply gradients
        if step % 100 == 0:
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data.numpy())
            # plotting decoded image (second row)
            _, decoded_data = autoencoder(view_data)
            for i in range(N_TEST_IMG):
                a[1][i].clear()
                a[1][i].imshow(np.reshape(decoded_data.data.numpy()[i], (28, 28)))
                a[1][i].set_xticks(()); a[1][i].set_yticks(())
            plt.draw(); plt.pause(0.05)
```

简单自编码器实现

Epoch: 0 | train loss: 0.2317

Epoch: 0 | train loss: 0.0671

Epoch: 0 | train loss: 0.0660

……

Epoch: 1 | train loss: 0.0578

Epoch: 1 | train loss: 0.0541

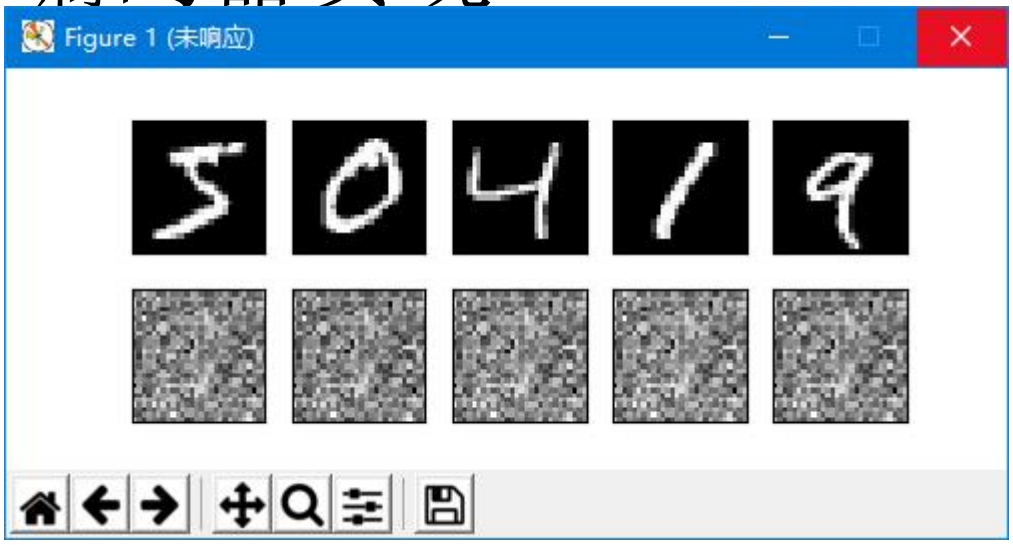
Epoch: 1 | train loss: 0.0563

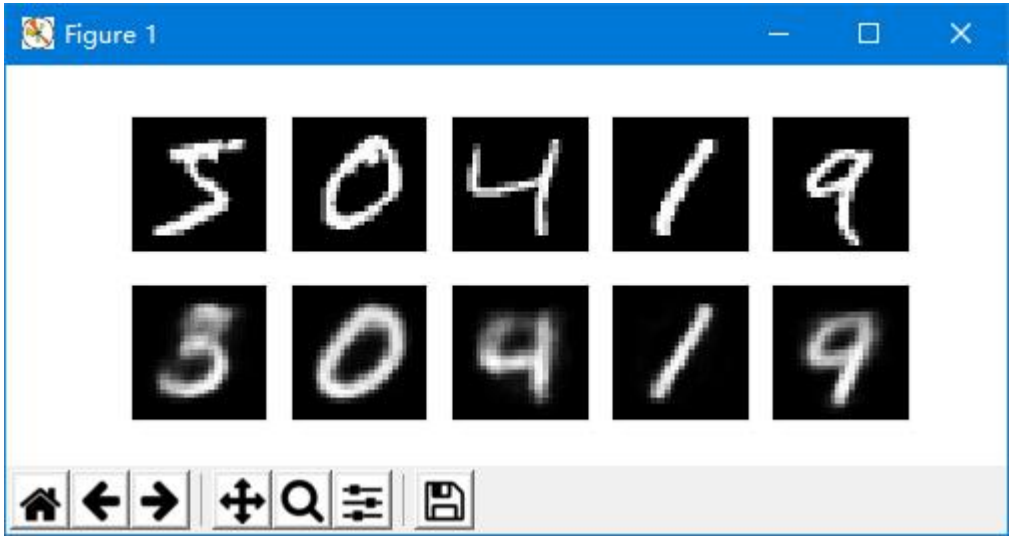
……

Epoch: 9 | train loss: 0.0373

Epoch: 9 | train loss: 0.0429

Epoch: 9 | train loss: 0.0405





简单自编码器实现

```
# visualize in 3D plot
view_data = train_data.train_data[:200].view(-1, 28*28).type(torch.FloatTensor)/255.
encoded_data, _ = autoencoder(view_data)
fig = plt.figure(2); ax = Axes3D(fig)
X, Y, Z = encoded_data.data[:, 0].numpy(), encoded_data.data[:, 1].numpy(), encoded_data.data[:, 2].numpy()
values = train_data.train_labels[:200].numpy()
for x, y, z, s in zip(X, Y, Z, values):
    c = cm.rainbow(int(255*s/9)); ax.text(x, y, z, s, backgroundcolor=c)
ax.set_xlim(X.min(), X.max()); ax.set_ylim(Y.min(), Y.max()); ax.set_zlim(Z.min(), Z.max())
plt.show()
```

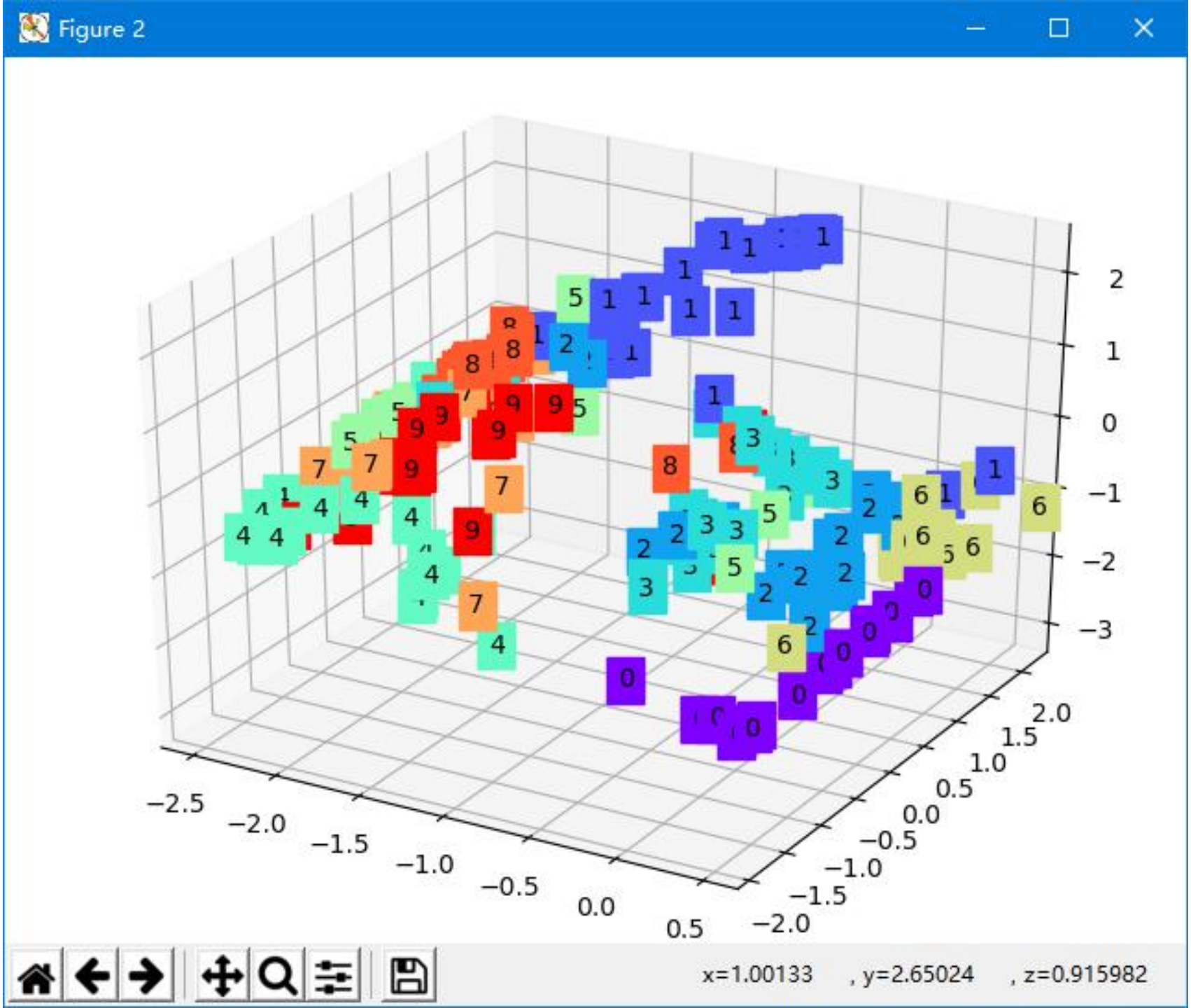


Figure 2
x=1.00133 , y=2.65024 , z=0.915982

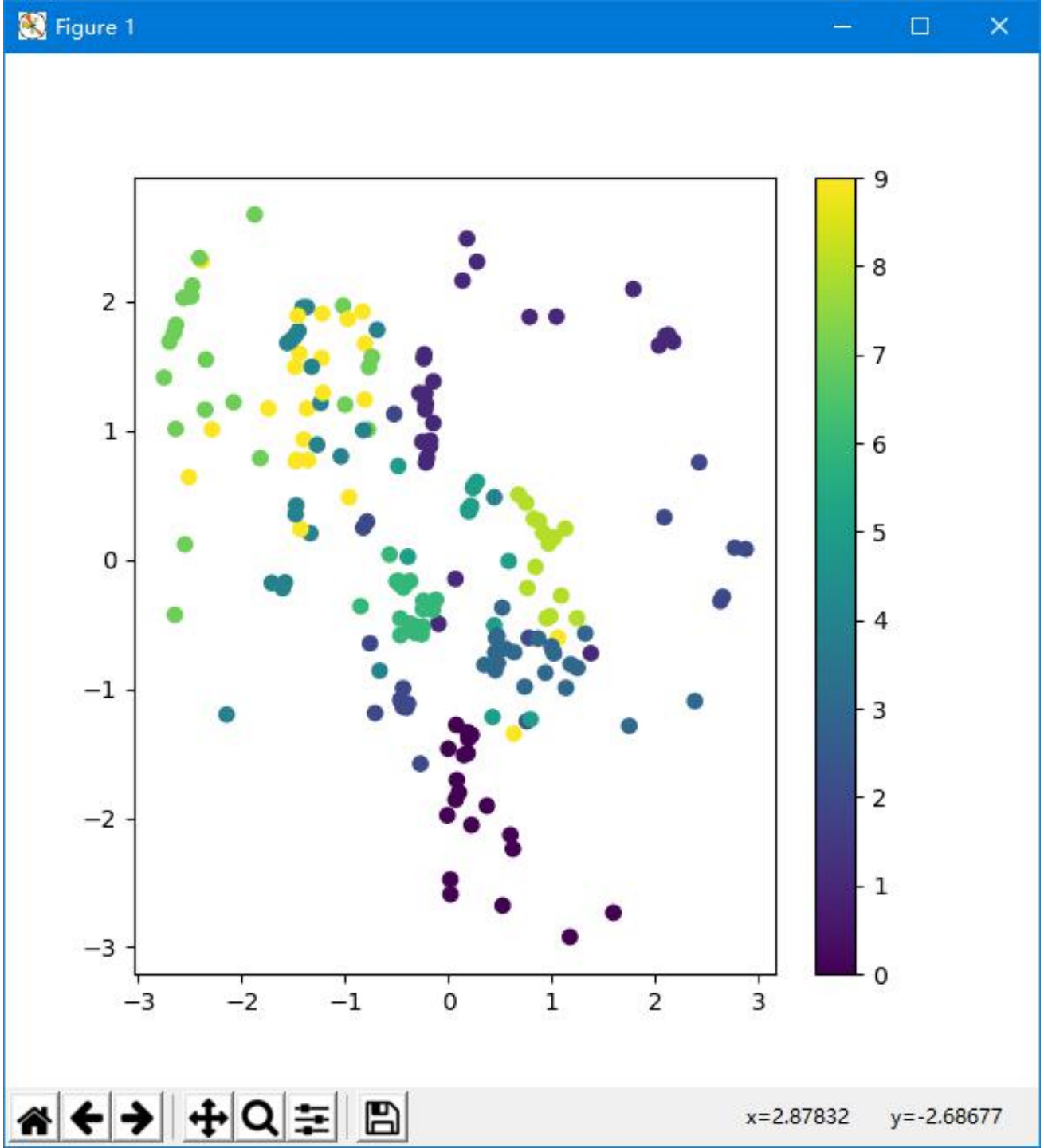


Figure 1
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
9
8
7
6
5
4
3
2
1
0
x=2.87832
y=-2.68677

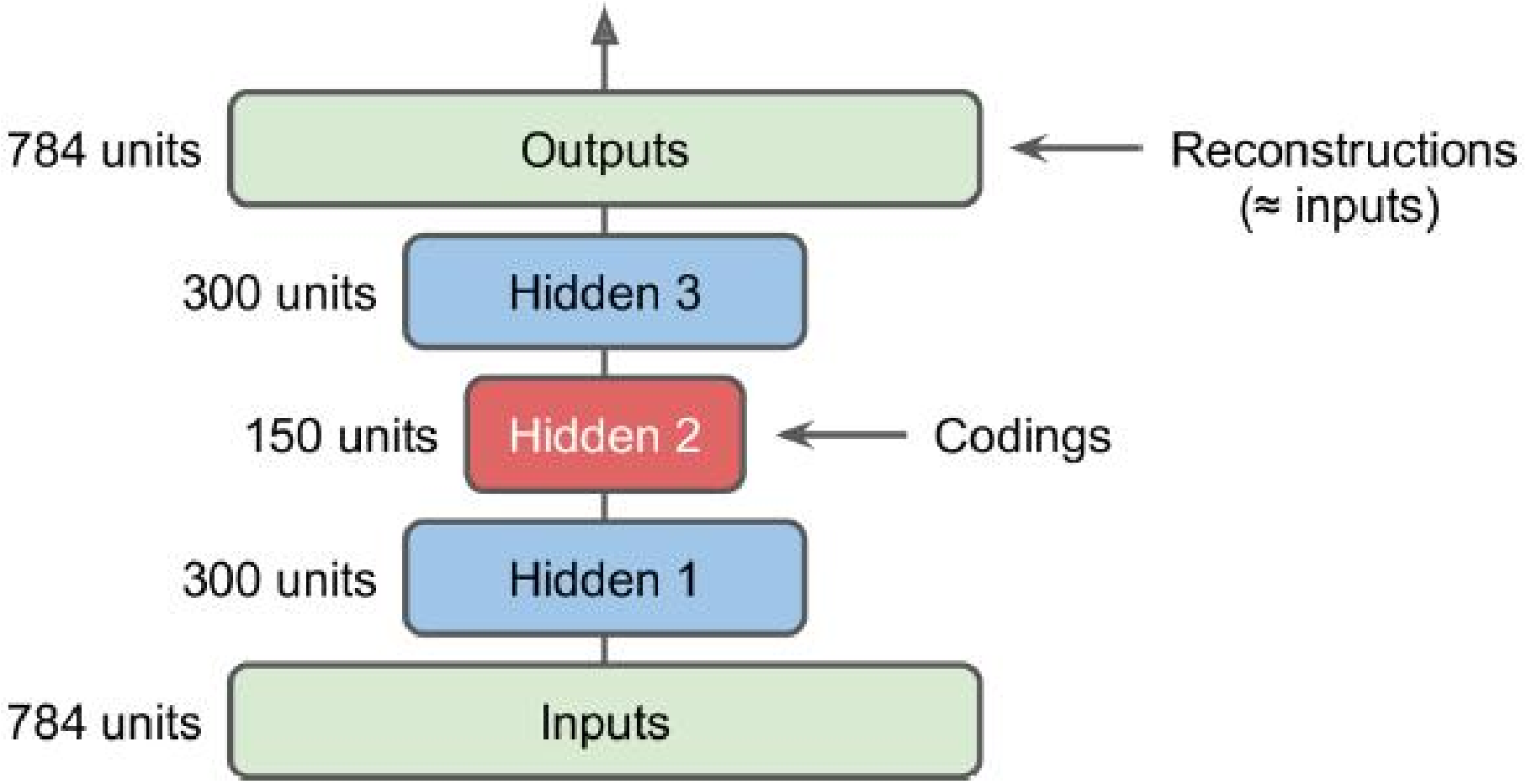
栈式自编码器（SAE）

- 就像我们讨论过的其他神经网络一样，自编码器可以有多个隐藏层。 在这种情况下，它们被称为栈式自编码器（或深度自编码器）。 添加更多层有助于自编码器了解更复杂的编码。 但是，必须注意不要让自编码器功能太强大。 设想一个编码器非常强大，只需学习将每个输入映射到一个任意数字（并且解码器学习反向映射）即可。 很明显，这样的自编码器将完美地重构训练数据，但它不会在过程中学习到任何有用的数据表示（并且它不可能很好地推广到新的实例）。

栈式自编码器（SAE）

- 栈式自编码器的架构关于中央隐藏层（编码层）通常是对称的。 简单来说，它看起来像一个三明治。 例如，一个用于 MNIST 的自编码器可能有 784 个输入，其次是一个隐藏层，有 300 个神经元，然后是一个中央隐藏层，有 150 个神经元，然后是另一个隐藏层，有 300 个神经元，输出层有 784 神经元。

栈式自编码器（SAE）

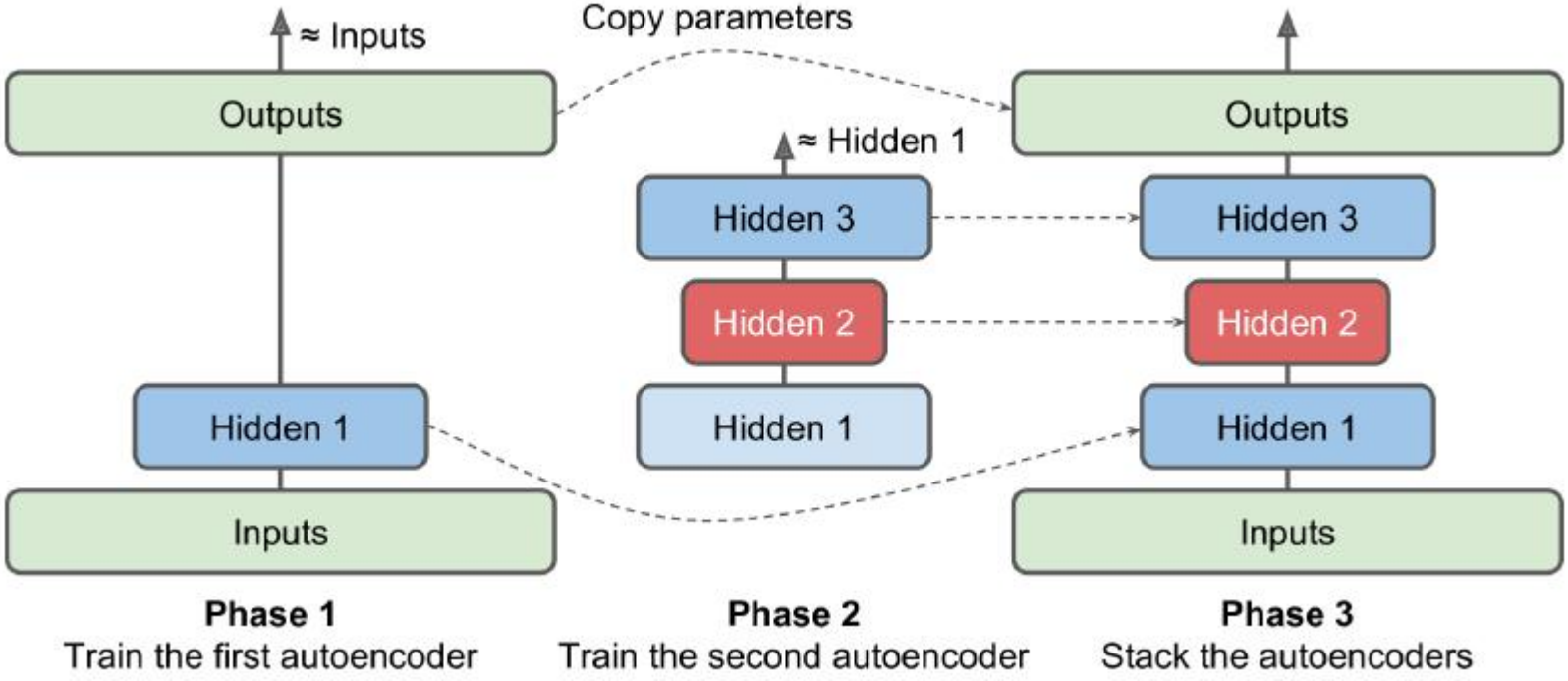

784 units
Outputs
Reconstructions
(≈ inputs)
300 units
Hidden 3
150 units
Hidden 2
Codings
300 units
Hidden 1
784 units
Inputs

关联权重

- 当自编码器整齐地对称时，就像我们刚刚构建的那样，一种常用技术是将解码器层的权重与编码器层的权重相关联。 这样减少了模型中的权重数量，加快了训练速度，并限制了过度拟合的风险。
- 具体来说，如果自编码器总共具有N个层（不计入输入层），并且 W_L表示第L层的连接权重（例如，层 1 是第一隐藏层，则层N / 2是编码层，而层N是输出层），则解码器层权重可以简单地定义为：$W_{N-L+1} = W_L^T$ (with $L = 1, 2, \cdots, N/2$).

拆分训练自编码器

- 我们不是一次完成整个栈式自编码器的训练，而是一次训练一个浅自编码器，然后将所有这些自编码器堆叠到一个栈式自编码器（因此名称）中，通常要快得多，如下图所示。 这对于非常深的自编码器特别有用。



拆分训练自编码器

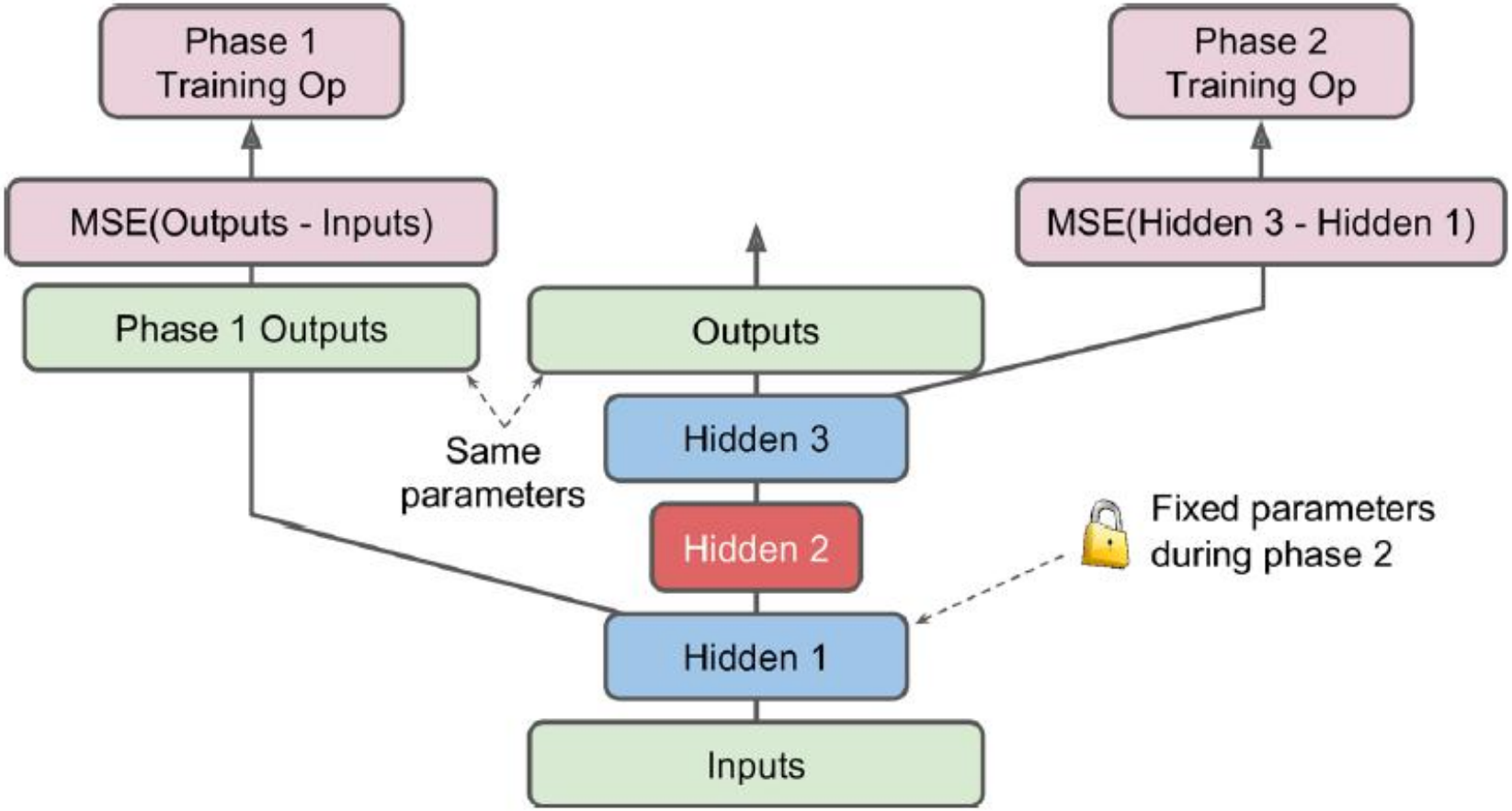
- 在训练的第一阶段，第一个自编码器学习重构输入。在第二阶段，第二个自编码器学习重构第一个自编码器隐藏层的输出。 最后，只需使用所有这些自编码器来构建一个大三明治，即首先堆叠每个自编码器的隐藏层，然后按相反顺序堆叠输出层。 这将形成一个栈式自编码器。可以用这种方式轻松地训练更多的自编码器，构建一个非常深的栈式自编码器。

拆分训练自编码器

- 为了实现这种多阶段训练算法，最简单的方法是训练完一个自编码器后，通过它运行训练集并捕获隐藏层的输出。 这个输出作为下一个自编码器的训练集。 一旦所有自编码器都以这种方式进行了训练,只需复制每个自编码器的权重和偏置，然后使用它们来构建堆叠的自编码器。

拆分训练自编码器

- 另一种方法是使用包含整个栈式自编码器的模块，每个训练阶段执行不同的操作，如下图所示。



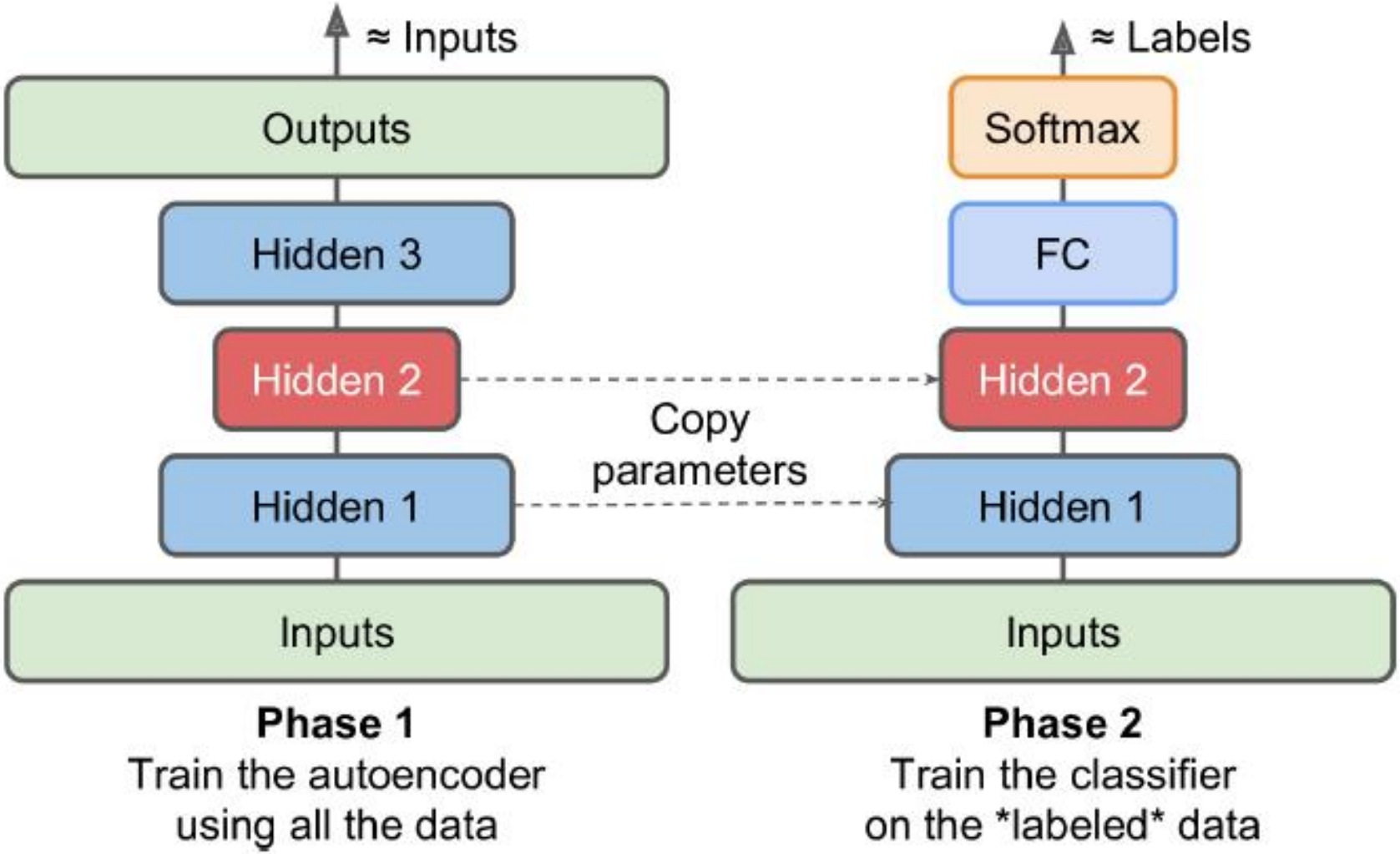
无监督预训练

- 如果正在处理复杂的监督任务，但没有大量标记的训练数据，则一种解决方案是找到执行类似任务的神经网络，然后重新使用其较低层。 这样就可以仅使用很少的训练数据来训练高性能模型，因为神经网络不必学习所有的低级特征；它将重新使用现有网络学习的特征检测器。

无监督预训练使用栈式自编码器

- 同样，如果您有一个大型数据集，但大多数数据集未标记，您可以先使用所有数据训练栈式自编码器，然后重新使用较低层为实际任务创建一个神经网络，并使用标记数据对其进行训练。 例如，后图显示了如何使用栈式自编码器为分类神经网络执行无监督预训练。 正如前面讨论过的，栈式自编码器本身通常每次都会训练一个自编码器。 在训练分类器时，如果您确实没有太多标记的训练数据，则可能需要冻结预训练层（至少是较低层）。

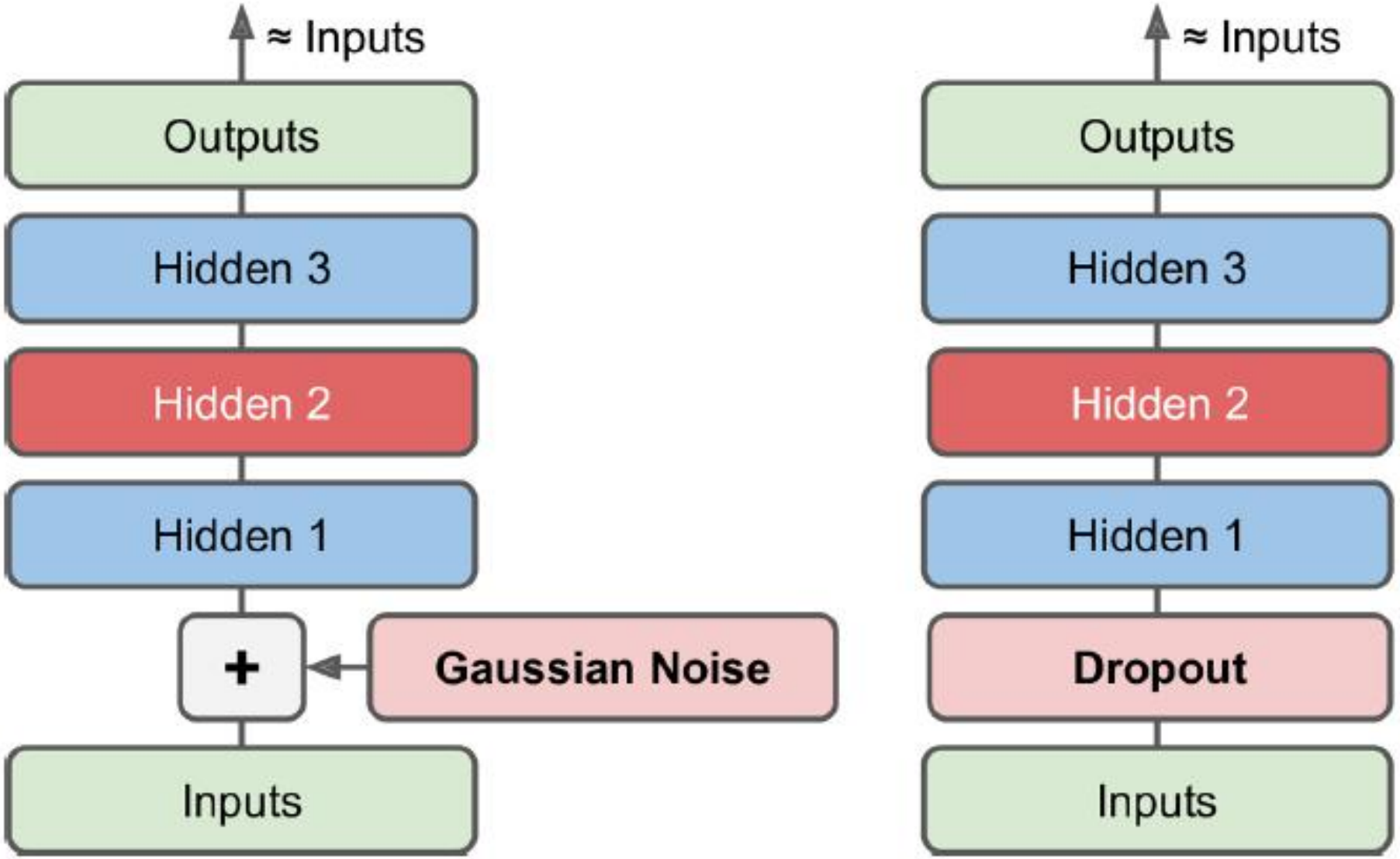
无监督预训练使用栈式自编码器



降噪自编码（DAE）

- 另一种强制自编码器学习有用功能的方法是为其输入添加噪声，对其进行训练以恢复原始的无噪声输入。这可以防止自编码器将其输入复制到其输出，因此最终不得不在数据中查找模式。
- 噪声可以是纯粹的高斯噪声添加到输入，或者它可以随机关闭输入，就像 drop out。

降噪自编码（DAE）


≈ Inputs
Outputs
Hidden 3
Hidden 2
Hidden 1
+
Gaussian Noise
Inputs
≈ Inputs
Outputs
Hidden 3
Hidden 2
Hidden 1
Dropout
Inputs

稀疏自编码器

- 另一种能提取良好特征的约束是稀疏性：通过向损失函数添加适当的项，自编码器被推动以减少编码层中活动神经元的数量。 例如，它可能被推到编码层中平均只有 5% 的显着活跃的神经元。 这迫使自编码器将每个输入表示为少量激活的组合。 因此，编码层中的每个神经元通常都会代表一个有用的特征（如果您每个月只能说几个字，它们很可能值得一听）。

稀疏自编码器

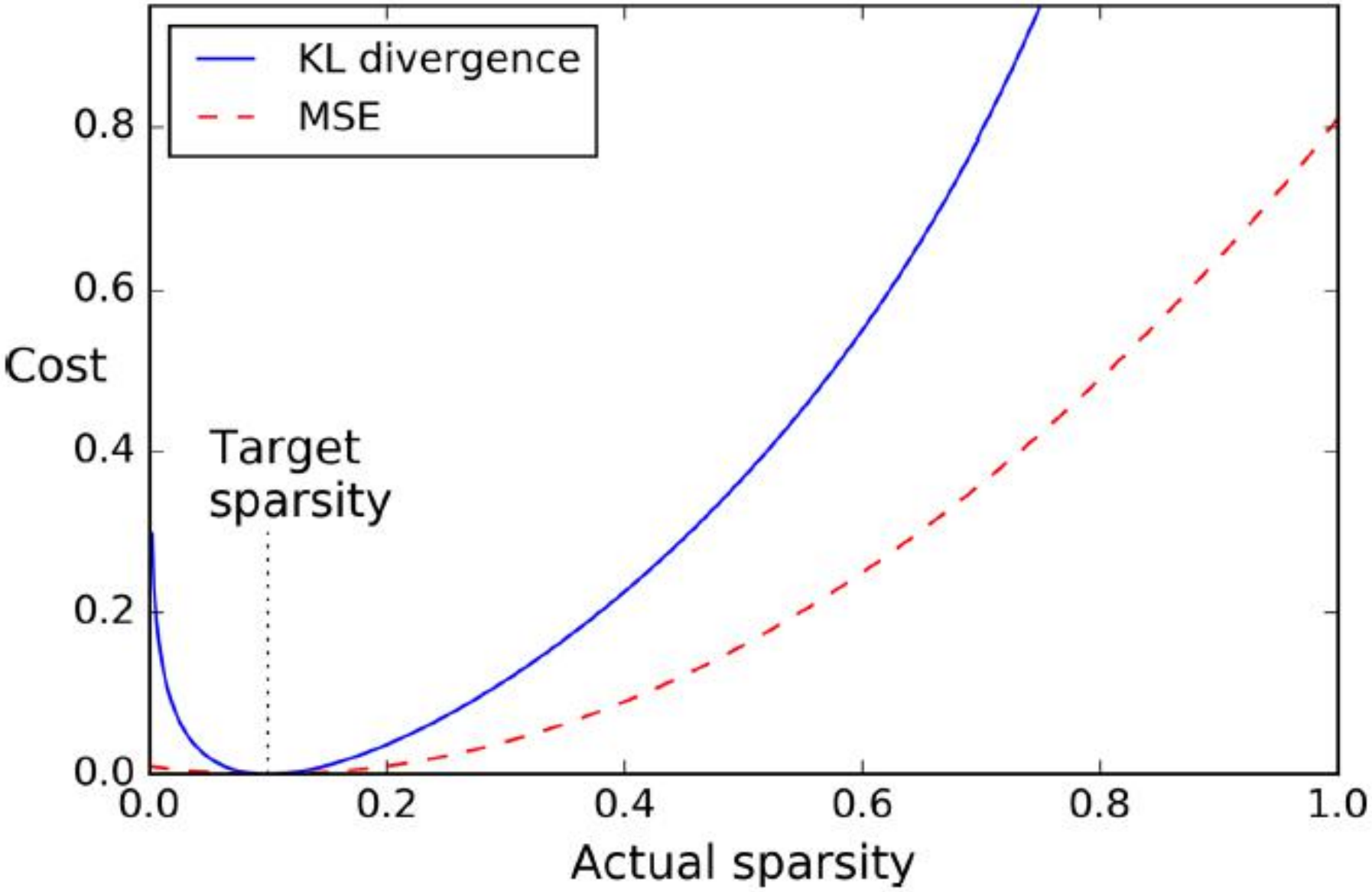
- 为了支持稀疏模型，我们必须首先在每次训练迭代中测量编码层的实际稀疏度。 我们通过计算整个训练批次中编码层中每个神经元的平均激活来实现。 批量大小不能太小，否则平均数不准确。

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)}) \right]$$

稀疏自编码器

- 一旦我们对每个神经元进行平均激活，我们希望通过向损失函数添加稀疏损失来惩罚太活跃的神经元。 例如，如果我们测量一个神经元的平均激活值为 0.3，但目标稀疏度为 0.1，那么它必须受到惩罚才能激活更少。一种方法可以简单地将平方误差$(0.3-0.1)^2$添加到损失函数中，但实际上更好的方法是使用 Kullback-Leibler 散度，其具有比均方误差更强的梯度。

稀疏损失


KL divergence
MSE
Cost
Target
sparsity
0.0
0.2
0.4
0.6
0.8
1.0
Actual sparsity

稀疏损失

- 给定两个离散的概率分布P和Q，这些分布之间的 KL 散度，记为Dkl(P || Q)，可以使用公式 15-1 计算。

Equation 15-1. Kullback–Leibler divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

- 在我们的例子中，我们想要测量编码层中的神经元将激活的目标概率p与实际概率q（即，训练批次上的平均激活）之间的差异。 所以KL散度简化为公式 15-2。

Equation 15-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

变分自编码器（VAE）

- 变分自编码器是另一类重要的自编码器，并迅速成为最受欢迎的自编码器类型之一。
- 它们与我们迄今为止讨论的所有自编码器完全不同，特别是：
  - 它们是概率自编码器，意味着即使在训练之后，它们的输出部分也是随机确定的。
  - 最重要的是，它们是生成自编码器，这意味着它们可以生成看起来像从训练集中采样的新实例。

变分自编码器（VAE）

- 变分自编码器编码数据的分布，它为编码施加约束，使得编码器学习到输入数据的隐变量模型。隐变量模型是连接显变量集和隐变量集的统计模型，隐变量模型的假设是显变量是由隐变量的状态控制的，各个显变量之间条件独立。也就是说，变分编码器不再学习一个任意的函数，而是学习你的数据概率分布的一组参数。通过在这个概率分布中采样，你可以生成新的输入数据。

变分编码器的工作原理

- 首先，编码器网络将输入样本 x 转换为隐空间的两个参数，记作 z_mean 和 z_log_sigma 。然后，我们随机从隐藏的正态分布中采样得到数据点 z ，这个隐藏分布我们假设就是产生输入数据的那个分布。z = z_mean + exp(z_log_sigma)*epsilon，epsilon是一个服从正态分布的张量。最后，使用解码器网络将隐空间映射到显空间，即将 z 转换回原来的输入数据空间。

变分编码器的工作原理

- 参数藉由两个损失函数来训练，一个是重构损失函数，该函数要求解码出来的样本与输入的样本相似（与之前的自编码器相同），第二项损失函数是学习到的隐分布与先验分布的KL距离，作为一个正则，它对学习符合要求的隐空间和防止过拟合有帮助。

变分编码器的实现

```
class Encoder(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(Encoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return F.relu(self.linear2(x))
```

变分编码器的实现

```
class Decoder(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(Decoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return F.relu(self.linear2(x))
```

变分编码器的实现

```
class VAE(torch.nn.Module):
    latent_dim = 8

    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self._enc_mu = torch.nn.Linear(100, 8)
        self._enc_log_sigma = torch.nn.Linear(100, 8)
```

变分编码器的实现

```
def _sample_latent(self, h_enc):
    #Return the latent normal sample z ~ N(mu, sigma^2)
    mu = self._enc_mu(h_enc)
    log_sigma = self._enc_log_sigma(h_enc)
    sigma = torch.exp(log_sigma)
    std_z = torch.from_numpy(np.random.normal(0, 1, size=sigma.size())).float()
    self.z_mean = mu
    self.z_sigma = sigma
    # Reparameterization trick
    return mu + sigma * Variable(std_z, requires_grad=False)

def forward(self, state):
    h_enc = self.encoder(state)
    z = self._sample_latent(h_enc)
    return self.decoder(z)
```

变分编码器的实现

```
def latent_loss(z_mean, z_stddev):
    mean_sq = z_mean * z_mean
    stddev_sq = z_stddev * z_stddev
    return 0.5 * torch.mean(mean_sq + stddev_sq - torch.log(stddev_sq) - 1)

input_dim = 28 * 28
batch_size = 32

encoder = Encoder(input_dim, 100, 100)
decoder = Decoder(8, 100, input_dim)
vae = VAE(encoder, decoder)

criterion = nn.MSELoss()
optimizer = optim.Adam(vae.parameters(), lr=0.0001)
l = None
```

变分编码器的实现

- 假设我们有两个随机变量x1,x2，各自服从一个高斯分布，这两个分布的KL散度：

$$\int p_1(x) log \frac{p_1(x)}{p_2(x)} dx$$

$$= \int p_1(x)(log p_1(x) dx - log p_2(x)) dx$$

$$= \int p_1(x) * (log \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{\frac{(x-\mu_1)^2}{2\sigma_1^2}} - log \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{\frac{(x-\mu_2)^2}{2\sigma_2^2}}) dx$$

$$= \int p_1(x) * (-\frac{1}{2} log 2\pi - log\sigma_1 - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{1}{2} log 2\pi + log\sigma_2 + \frac{(x-\mu_2)^2}{2\sigma_2^2}) dx$$

$$= \int p_1(x)(log \frac{\sigma_2}{\sigma_1} + [\frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2}]) dx$$

$$= \int (log \frac{\sigma_2}{\sigma_1}) p_1(x) dx + \int (\frac{(x-\mu_2)^2}{2\sigma_2^2}) p_1(x) dx - \int (\frac{(x-\mu_1)^2}{2\sigma_1^2}) p_1(x) dx$$

$$= log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \int ((x-\mu_2)^2) p_1(x) dx - \frac{1}{2\sigma_1^2} \int ((x-\mu_1)^2) p_1(x) dx$$

变分编码器的实现

$$= log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\int((x-\mu_2)^2)p_1(x)dx - \frac{1}{2}$$

$$= log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\int((x-\mu_1+\mu_1-\mu_2)^2)p_1(x)dx - \frac{1}{2}$$

$$= log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}[\int(x-\mu_1)^2p_1(x)dx + \int(\mu_1-\mu_2)^2p_1(x)dx + 2\int(x-\mu_1)(\mu_1-\mu_2)]p_1(x)dx - \frac{1}{2}$$

$$= log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}[\int(x-\mu_1)^2p_1(x)dx + (\mu_1-\mu_2)^2] - \frac{1}{2}$$

$$= log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2+(\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

$$KL(\mu_1,\sigma_1) = -log\sigma_1 + \frac{\sigma_1^2+\mu_1^2}{2} - \frac{1}{2}。$$

变分编码器的实现

```
for epoch in range(100):
    for i, data in enumerate(dataloader, 0):
        inputs, classes = data
        inputs, classes = Variable(inputs.resize_(batch_size, input_dim)), Variable(classes)
        optimizer.zero_grad()
        dec = vae(inputs)
        ll = latent_loss(vae.z_mean, vae.z_sigma)
        loss = criterion(dec, inputs) + ll
        loss.backward()
        optimizer.step()
        l = loss.data.item()
    print(epoch, l)
```

变分编码器的实现

Number of samples: 60000

0 0.0682493522763252 3

1 0.0646212324500083 9

2 0.068235352635383 6

3 0.0665578916668891 9

4 0.0665909722447395 3

……

93 0.0647509619593620 3

94 0.0674372538924217 2

95 0.072048120200634

96 0.0674799084663391 1

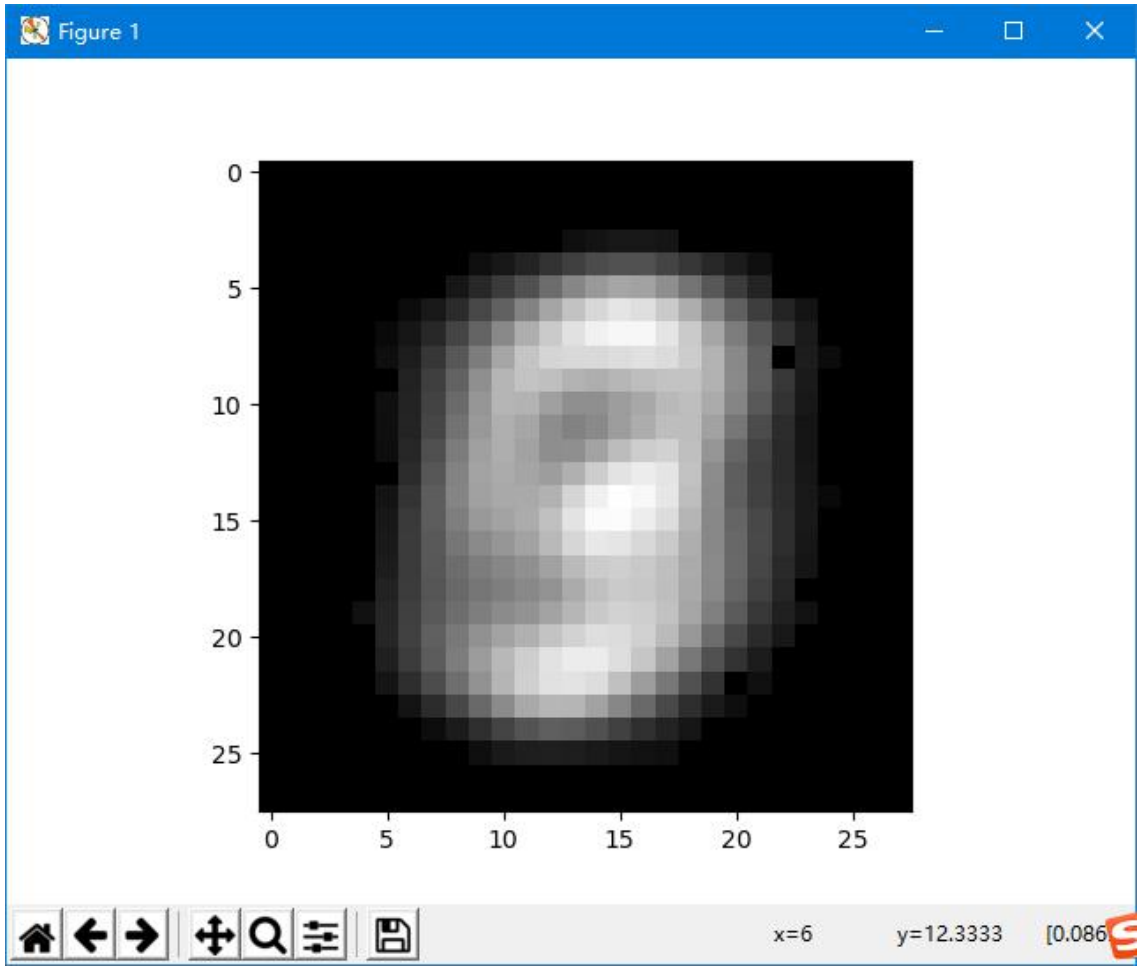
97 0.0648351758718490 6

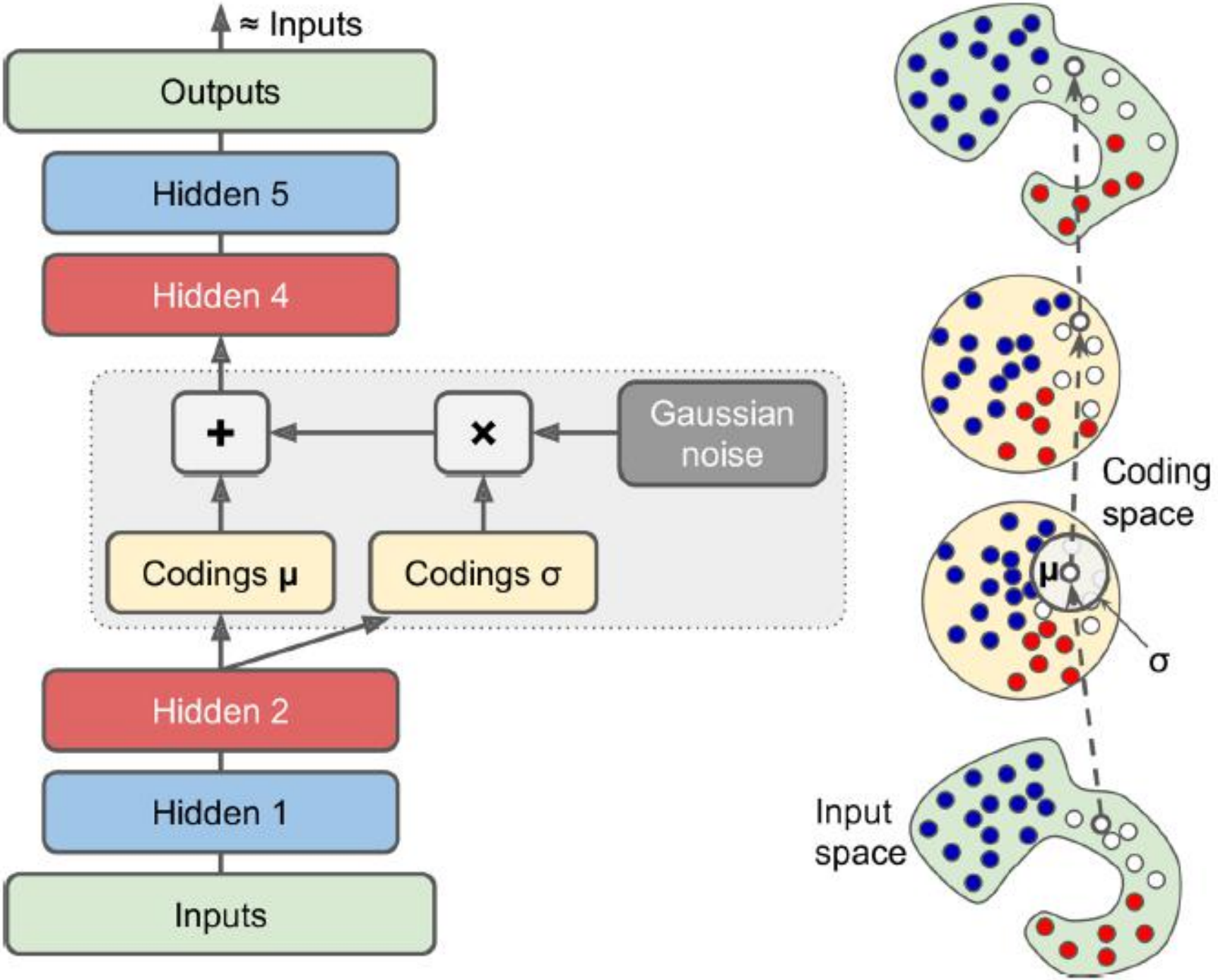
98 0.0625583082437515 3

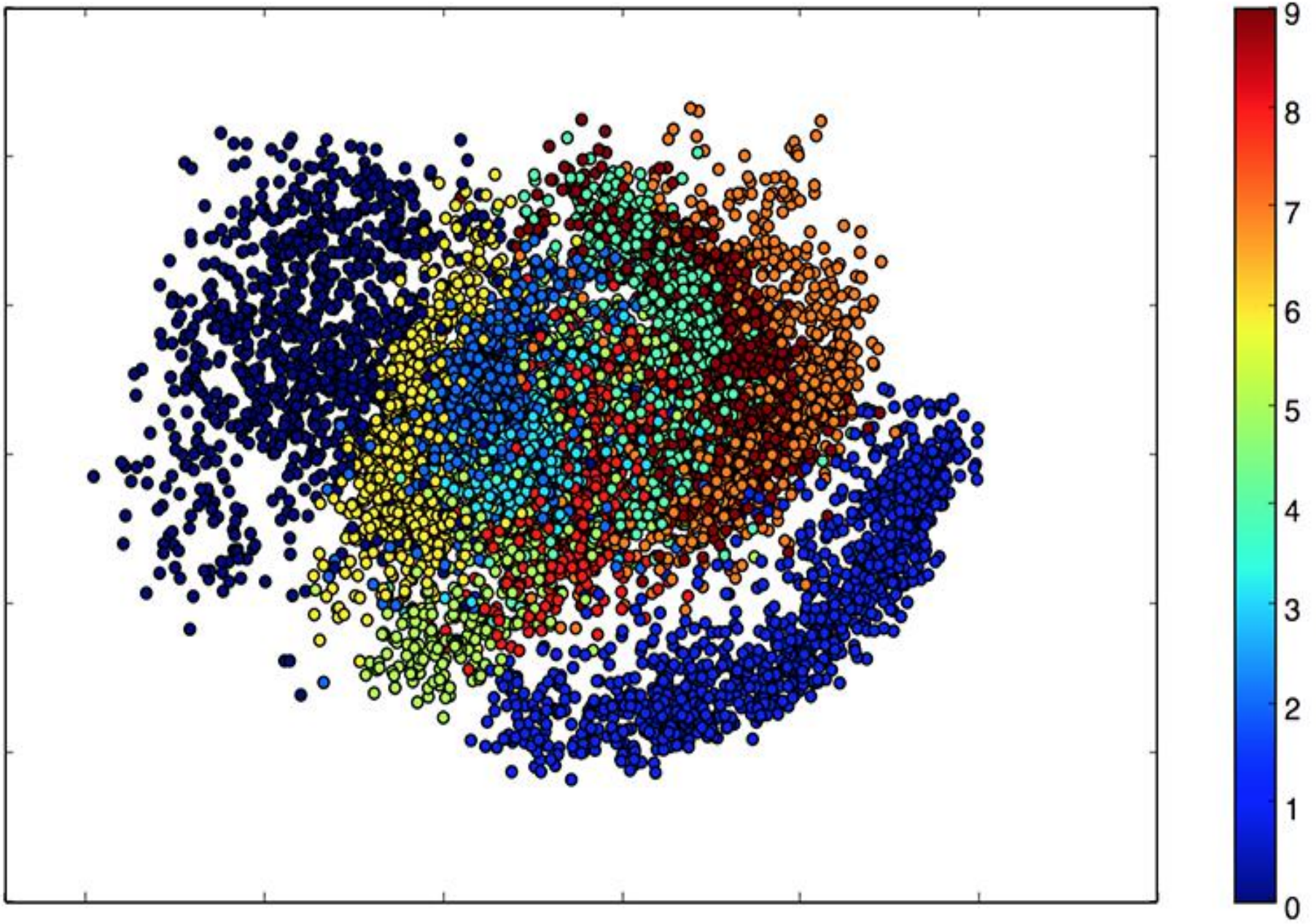
99 0.0680813491344451 9

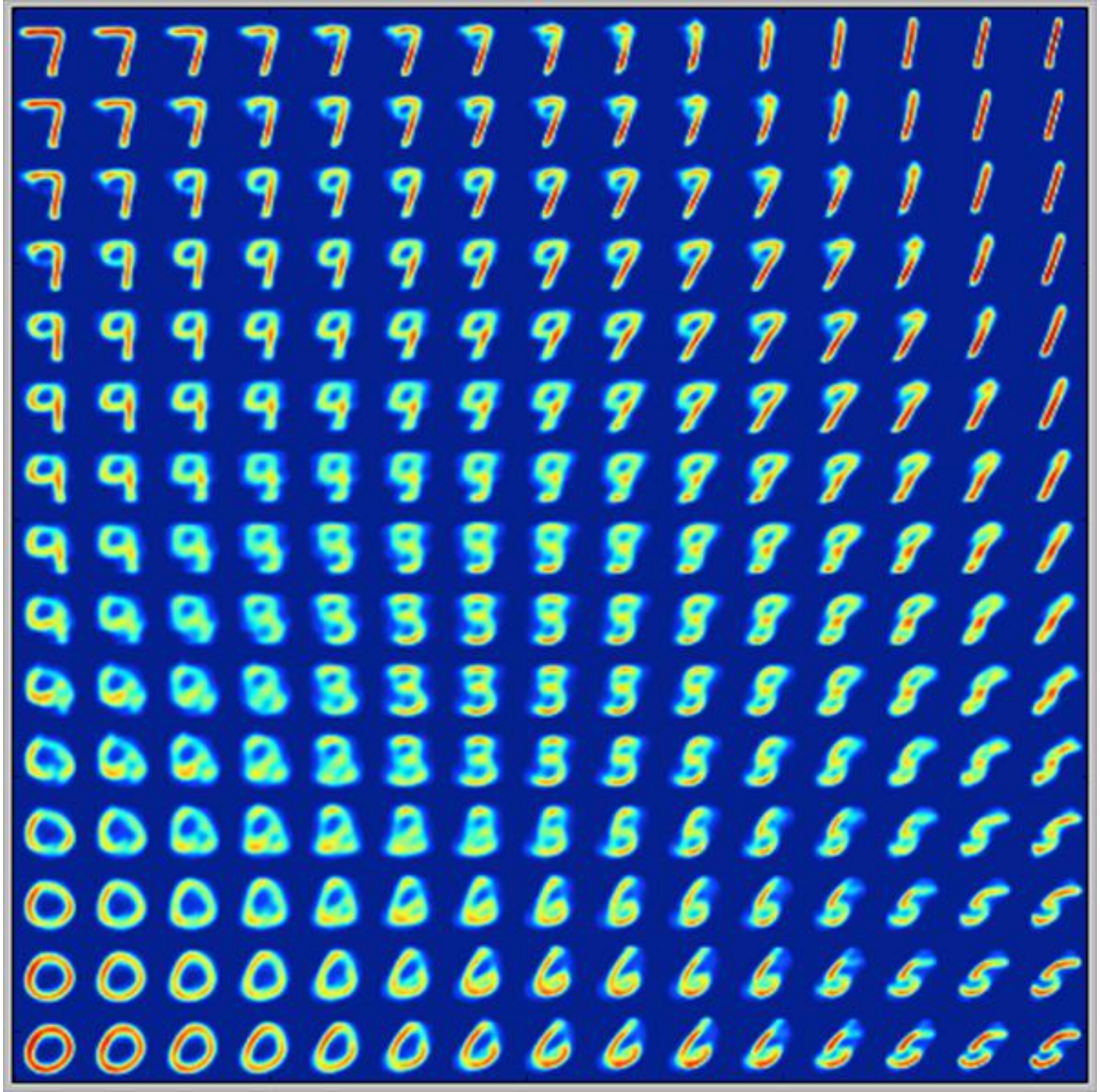
变分编码器的实现

```
plt.imshow(vae(inputs).data[0].numpy().reshape(28, 28), cmap='gray')
plt.show(block=True)
```




≈ Inputs
Outputs
Hidden 5
Hidden 4
+
×
Gaussian noise
Codings μ
Codings σ
Hidden 2
Hidden 1
Inputs
Coding space
μ
σ
Input space


9
8
7
6
5
4
3
2
1
0



其他自编码器

- 监督式学习在图像识别，语音识别，文本翻译等方面取得的惊人成就在某种程度上掩盖了无监督学习的局面，但它实际上正在蓬勃发展。以下是其他几种类型的自编码器的简要说明：

其他自编码器

- 压缩自编码器（CAE,contractive autoencoder）

自编码器在训练过程中受到约束，因此与输入有关的编码的导数很小。 换句话说，两个类似的输入必须具有相似的编码。压缩自编码器在编码 h = f(x) 的基础上添加了显式的正则项，鼓励 f 的导数尽可能小：

$$\mathcal{J}_{\mathrm{AE+wd}}(\theta) = \left(\sum_{x \in D_n} L(x, g(f(x))) \right) + \lambda \sum_{ij} W_{ij}^2$$

$$\mathcal{J}_{\mathrm{CAE}}(\theta) = \sum_{x \in D_n} \left(L(x, g(f(x))) + \lambda \| J_f(x) \|_F^2 \right)$$

其他自编码器

- 压缩自编码器（CAE,contractive autoencoder）

自编码器在训练过程中受到约束，因此与输入有关的编码的导数很小。 换句话说，两个类似的输入必须具有相似的编码。压缩自编码器在编码 h = f(x) 的基础上添加了显式的正则项，鼓励 f 的导数尽可能小：

$$\mathcal{J}_{\mathrm{CAE}}(\theta) = \sum_{x \in D_n} \left(L(x, g(f(x))) + \lambda \| J_f(x) \|_F^2 \right)$$

$$\| J_f(x) \|_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2$$

$$\| J_f(x) \|_F^2 = \sum_{i=1}^{d_h} \left(h_i \left(1 - h_i \right) \right)^2 \sum_{j=1}^{d_x} W_{ij}^2.$$

其他自编码器

- 压缩自编码器（CAE,contractive autoencoder）

自好的特征表示大致有2个衡量标准：

1. 可以很好的重构出输入数据;

2.对输入数据一定程度下的扰动具有不变形。

普通的autoencoder和sparse autoencoder主要是符合第一个标准。而去噪自编码器和压缩自编码器则主要体现在第二个。而作为分类任务来说，第二个标准显得更重要。

其他自编码器

- 压缩降噪自编码器（CDAE,contractive denoising-autoencoder）

将压缩自编码器和降噪自编码器相结合，损失函数定义如下：

$$\frac{1}{N}\sum_{\mathbf{x}\in D}((\mathbf{x}-\mathbf{x}_{rec})^2+\lambda\|J_{\mathbf{h}}(\tilde{\mathbf{x}})\|_F^2).$$

其他自编码器

- 栈式卷积自编码器（SCAE）

学习通过重构通过卷积层处理的图像来提取视觉特征的自编码器。

- 生成随机网络（GSN）

消除自编码器的泛化，增加了生成数据的能力。

其他自编码器

- 赢家通吃（WTA）的自编码

在训练期间，在计算编码层中所有神经元的激活之后，只保留训练批次上每个神经元的前 k% 激活，其余部分设为零。 自然这导致稀疏的编码。 而且，可以使用类似的 WTA 方法来产生稀疏卷积自编码器。

- 对抗自编码器（AAE）

一个网络被训练来重现它的输入，同时另一个网络被训练去找到第一个网络不能正确重建的输入。 这推动了第一个自编码器学习健壮的编码。