Entropy-Regularized Autoencoder

Hassan El Bouz

July 14, 2025

Abstract

This report describes the implementation and analysis of an entropy-regularized autoencoder trained using PyTorch. The model aims to figure out the latent space by itself by incorporating both reconstruction loss and a custom entropic loss. We discuss the data pipeline, architecture, loss functions, training procedure, and experimental results, including interpretability of learned weights and latent codes.

1 Introduction

Autoencoders are unsupervised neural architectures designed to learn compact representations by reconstructing input data. While standard autoencoders minimize reconstruction error, their latent space is generally fixed with play over the structure of the latent space or interpretability. In this work, we enhance the autoencoder's capacity to learn to "compactify" the latent space by introducing an entropic regularization term.

2 Theory and Background

2.1 Autoencoders

Autoencoders are series of neural networks that encode input data into a lower-dimensional latent space and then decode it back to the original space. The architecture typically consists of an encoder, a bottleneck (latent space), and a decoder. The latent space has less neurons than the input, forcing the model to reduce the degrees of freedom (or better, dimension), thus learning a compressed representation of the input data. The output has the same shape as the input, and the model is trained by minimizing the difference between the input and output, usually using Mean Squared Error (MSE).

2.2 Shannon Entropy

Entropy, or Shannon Entropy, is a measure of uncertainty or randomness in a system. A generally good interpretation of entropy is the expected number of bits needed to encode the output of a random variable in the most efficient way, or an asymptotic limit. For example, if my random variable takes 4 values with equal probability, say, (A, B, C, D), the entropy is 2 bits, because I can encode each value with 2 bits (00 for A, 01 for B, 10 for C, and 11 for D). However, if A happens more frequently than the others, say with probability 0.50, the entropy will be lower, as I can encode A with 1 bit (0) and the others with 2 bits (11 for B, 100 for C, and 101 for D), and my expected number of bits will be lower than 2 bits.

Entropy is defined mathematically as:

$$H(X) = -\sum_{i=1}^{n} p_i \log(p_i)$$

where p_i is the probability of the i-th outcome of a random variable X. When we use logarithm base 2, it has units of bits, but any other logarithm differs by a constant multiple. It is not difficult to prove that the maximum entropy is achieved when all outcomes are equally likely, i.e., $p_i = \frac{1}{n}$ for n outcomes, and minimized when one outcome has probability 1 and all others have probability 0. In other words, the more my distribution is clustered, the lower the entropy. For uniform distributions, the entropy will simply be $\log(n)$, where n is the number of outcomes.

2.3 Renyi Entropy

Renyi entropy is a generalization of Shannon entropy, defined as:

$$H_\alpha(X) = \frac{\alpha}{1-\alpha} \log\left(||p||_\alpha\right)$$

where $|| \cdot ||_\alpha$ is $\alpha-$norm.

A few special cases of Renyi entropy are:

- For $\alpha = 1$, it reduces to Shannon entropy.
- For $\alpha = 0$, it reduces to the logarithm of the number of nonzero probabilities.
- For $\alpha \to \infty$, it approaches the min-entropy, which is the negative logarithm of the maximum probability of any single outcome.

The intuition of Renyi entropy is that it generalizes the "logarithm of number of outcomes". Hence, the higher the α, the more it focuses on the most probable outcomes, while lower α values consider all outcomes more equally.

2.4 Entropic Compactifier Regularization

In this subsection, we introduce how entropy can be used to create bottlenecks throughout the autoencoder architecture. The idea is to penalize the model for having more neurons in the bottleneck. But how do we do that? We can start with a fixed latent space size, say 20 neurons, and then assign to each neuron a weight that determines how "active" it is. There are two ways to do this: either sum the magnitude of the contributions of the incoming connections from the previous layer and add the biases, or sum the contributions of the outgoing connections to the next layer and add the biases. The first has the intuition of

"Ah, this node is important! let's contrubute to it more", while the second has the intuition of "Ah, this node is important! let's use it more".

After evalutating the "activeness" of each neuron, we would like them to be more localized in the latent space. We can treat the weight distribution as a probability distribution after normalizing it, and then compute its entropy. We would like to minimize it, and the parameter α controls how much we want to penalize the model for having more neurons in the bottleneck.

2.5 Path Entropic Loss

Another approach with dealing with bottlenecks is to consider all possible paths from the first layer to the last. Each path can be represented as a sequence of weights and biases, and we can assign a "cummulative" weight to each path by multiplying the weights and biases along it. Similar to the previous approach, the more weights the path has, the more "active" it is. A bottleneck can be created by penalizing the model for having more active paths, i.e. we want to localize the weights and biases into fewer paths. We hope that this localization will be "nice" and fall into one layer, creating a bottleneck. To do so we can do the same approach and calculate the entropy of the path distributions. Unfortunately, this approach is computationally expensive and scales up quickly, but as an advantage, it figures out the bottleneck location and size by itself.

3 Implementation

3.1 Entropic Compactifier Loss Function

The code for the Entropic Compactifier Loss Function is implemented in Pytorch. In short, we define 3 types of functions:

1. **Node Value Functions:** These compute the "activeness" of each node in the latent space by summing the absolute values of incoming or outgoing weights and adding the biases.

2. **Entropy:** These compute the entropy of the normalized node values, either for left or right nodes.

3. **Entropic Loss Function:** This computes the entropy of the system from both left and right nodes by joining the weights, or adding both entropies

The code is given below:

Listing 1: Entropic Compactifier Loss Function

```python
def node_right_value(W, B):
    return torch.sum(torch.abs(W), dim=1) + torch.abs(B) # Weights for the incoming connections (node to the right)

def node_left_value(W, B):
    return torch.sum(torch.abs(W), dim=0) + torch.abs(B) # Weights for the outgoing connections (node to the left)

def entropic_loss_right(W, B, alpha=1.0, beta = 1.0):
    right_values = node_right_value(W, B)
    normalized_values = right_values / torch.sum(right_values)
    if beta == 1.0:
        loss = -torch.sum(normalized_values * torch.log(normalized_values + 1e-10))
    else:
        loss = beta/(1-beta) * torch.log(torch.norm(normalized_values, p=beta) + 1e-10)
    return alpha * loss

def entropic_loss_left(W, B, alpha=1.0, beta = 1.0):
    left_values = node_left_value(W, B)
    normalized_values = left_values / torch.sum(left_values)
    if beta == 1.0:
        loss = -torch.sum(normalized_values * torch.log(normalized_values + 1e-10))
    else:
        loss = beta/(1-beta) * torch.log(torch.norm(normalized_values, p=beta) + 1e-10)
    return alpha * loss

def entropic_loss(W_l, W_r, B, alpha=1.0, beta=1.0): # Difference between this and cumm_entropic_loss is that this one computes the loss for both left and right nodes separately
    loss_l = entropic_loss_left(W_r, B, alpha, beta)
    loss_r = entropic_loss_right(W_l, B, alpha, beta)
    return loss_l + loss_r

def cumm_entropic_loss(W_l, W_r, B, alpha=1.0): # Difference between this and entropic_loss is that this one assigns the node weight from the left and right before computing the loss
    values = node_right_value(W_l, B) + node_left_value(W_r, B)
    normalized_values = values / torch.sum(values)
    loss = -torch.sum(normalized_values * torch.log(normalized_values + 1e-10)) # Add small constant to avoid log(0)
    return loss * alpha
```

3.2 Path Entropic Loss Function

The path entropic loss function is more complex and requires iterating through all possible paths in the network. To ensure it is optimized, we first define an array that contains all possible paths we want to work with. Since this does not change, we do it once and then use it as the input in the loss function.

Later on, by using the optimized torch slicing, we can cleverly compute the products of the sum of weights and biases along each path. Finally, we compute the entropy of the path distribution.

Listing 2: Path Entropic Loss Function

```
def path_entropy_parallel(first_node, last_node, weight_list, bias_list=None, a=1, paths
    = None):
    device = weight_list[0].device

    if paths is None:
        sizes = [w.size(1) for w in weight_list] + [weight_list[-1].size(0)]
        grids = [torch.arange(s, device=device) for s in sizes]
        all_paths = cartesian_product_gpu(grids)

        # Only keep paths from first_node to last_node
        valid_mask = (all_paths[:, 0] == first_node) & (all_paths[:, -1] == last_node)
        selected_paths = all_paths[valid_mask]
    else:
        selected_paths = paths

    if selected_paths.size(0) == 0:
        return torch.tensor(0.0, device=device, requires_grad=True)

    num_paths = selected_paths.size(0)
    num_layers = len(weight_list)

    weights = torch.ones(num_paths, device=device)

    for i in range(num_layers):
        src_idx = selected_paths[:, i]
        tgt_idx = selected_paths[:, i + 1]

        # Apply weight
        W = weight_list[i]

        # Apply bias at current source node
        if bias_list is not None and bias_list[i] is not None:
            weights *= torch.abs(W[tgt_idx, src_idx]) + torch.abs(bias_list[i][tgt_idx])
        else:
            weights *= torch.abs(W[src_idx, tgt_idx])

    # Normalize and compute entropy
    abs_weights = weights.abs()
    total = abs_weights.sum()
    probs = abs_weights / (total + 1e-9)

    if a == 1:
        entropy = -torch.sum(probs * torch.log2(probs + 1e-9))
    else:
        entropy = a / (1 - a) * torch.log2(torch.norm(probs, p=a))

    return entropy, selected_paths
```

4 Methodology

4.1 Data Loading

We use the MNIST dataset of 28×28 grayscale handwritten digits to try to create a program that can generate number figures. PyTorch's `torchvision.datasets` provides an efficient pipeline. Data is batched for training.

4.2 Model Architecture

The autoencoder architecture is as follows:

- Input: Flattened 28×28 image
- Encoder: Linear layers with ReLU activations, compressing to a latent dimension
- Bottleneck: Several layers with the same latent size, enabling information bottlenecking
- Decoder: Mirrors the encoder, reconstructing back to image shape
- Output: Sigmoid activation for pixel values in $[0, 1]$

In this implementation, we do not seperate the network explicitly into an encoder and decoder. Remember that the goal is to let the network figure out the bottleneck by itself, so we do not need to specify it. However, we can still guide the network into encoder and decoder to make training more efficient. We expect the bottleneck to happen somewhere in the ”extended latent space” where it is just a safe area for the network to place the bottleneck appropriately.

Listing 3: Autoencoder Definition

```
class Autoencoder(nn.Module):
    def __init__(self, extended_latent_dim=20):
        super().__init__()
        self.autoencoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 64), nn.ReLU(),
            nn.Linear(64, extended_latent_dim), nn.ReLU(),
            nn.Linear(extended_latent_dim, extended_latent_dim), nn.ReLU(),
            nn.Linear(extended_latent_dim, extended_latent_dim), nn.ReLU(),
            nn.Linear(extended_latent_dim, 64), nn.ReLU(),
            nn.Linear(64, 28*28), nn.Sigmoid(),
            nn.Unflatten(1, (1, 28, 28))
        )
```

4.3 Loss Function

The total loss combines:

- **Reconstruction Loss:** Mean Squared Error (MSE) between input and output.

- **Entropic Loss:** Either of the two entropic losses defined earlier.

$$\text{Total Loss} = \lambda_1 \cdot \text{MSE} + \lambda_2 \cdot \text{EntropicLoss}$$

In the first experiment, we work with the compactifier with $\lambda_1 = 0.7$ and $\lambda_2 = 0.3$ in our experiments.

Listing 4: Loss Computation

```
recon = model(batch)
criterion_loss = criterion(recon, batch)
eloss = entropic_loss(W_l, W_r, B, alpha=1.0, beta=0.04)
loss = criterion_term * criterion_loss + entropy_term * eloss
```

Above, we specify based on preference the Renyi-Entropy of parameter 0.04 (called β, while α is the logarithm base), since we wish to capitalize over the smallest weights as well.

4.4 Training Procedure

We train for 20 epochs using Adam optimizer. At each batch:

1. Forward pass
2. Compute losses
3. Backpropagation and parameter update
4. Optionally, print weight statistics for monitoring

5 Experiments and Results

In this setup, we were able to reduce the latent space size to 10 neurons.

5.1 Reconstruction Quality (Compactifier)

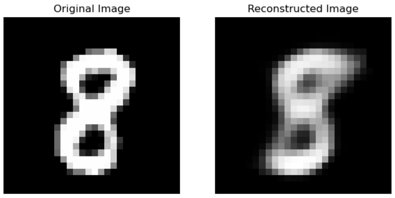

Figure 1: Sample reconstructed digits after training.

The encoding of the digits is:

```
[9.5884e-03, 0.0000e+00, 3.6991e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,
 3.0069e+01, 6.4067e+00, 4.2131e+01, 1.4036e-02, 2.1916e-02, 0.0000e+00,
 0.0000e+00, 1.2810e+02, 0.0000e+00, 5.8852e+00]
```

Which, even though is of size 16, we use less than 10 neurons to encode it. The machine ”deactivates” the neurons, but not remove them completely. Of course, we can rewrite the entire network with only these 10 neurons.

5.2 Latent Space Analysis (Compactifier)

To analyze what each neuron in the latent space does, we send to the decoder a tensor full of zeros except at a single neuron where we vary the value from -3 to 3. The decoder will then reconstruct the image, and we can see how each neuron affects the output.

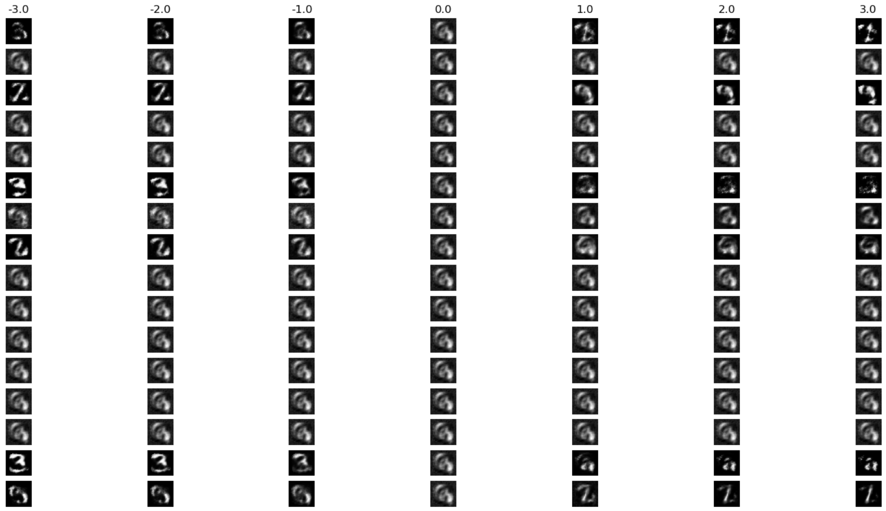

Figure 2: Sample reconstructed latent space after training.

We further see that some neurons do not play a role

5.3 Reconstruction Quality (Path Entropic Loss)

After training the autoencoder, we were able to achieve this latent structure:

```
Layer 0: 784 effective nodes (28*28 = 784)
Layer 1: 64 effective nodes
Layer 2: 16 effective nodes
Layer 3: 4 effective nodes
Layer 4: 12 effective nodes
Layer 5: 64 effective nodes
Layer 6: 784 effective nodes
```

In this case, we were too extreme by reducing it to 4 neurons. Nevertheless, let us do the analysis of it.

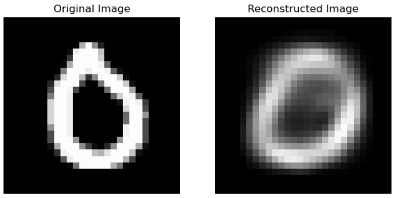

Figure 3: Sample reconstructed digits after path entropy training.

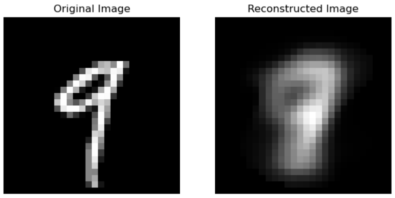

Figure 4: Failed reconstructed digits after path entropy training.

Things like 0, are encoded very well, but things like, 9, are encoded poorly.

To do the same analysis of the latent space as before, we send to the decoder a tensor full of zeros except at a single neuron where we vary the value from -3 to 3. The decoder will then reconstruct the image, and we can see how each neuron affects the output.

6 References

- PyTorch documentation

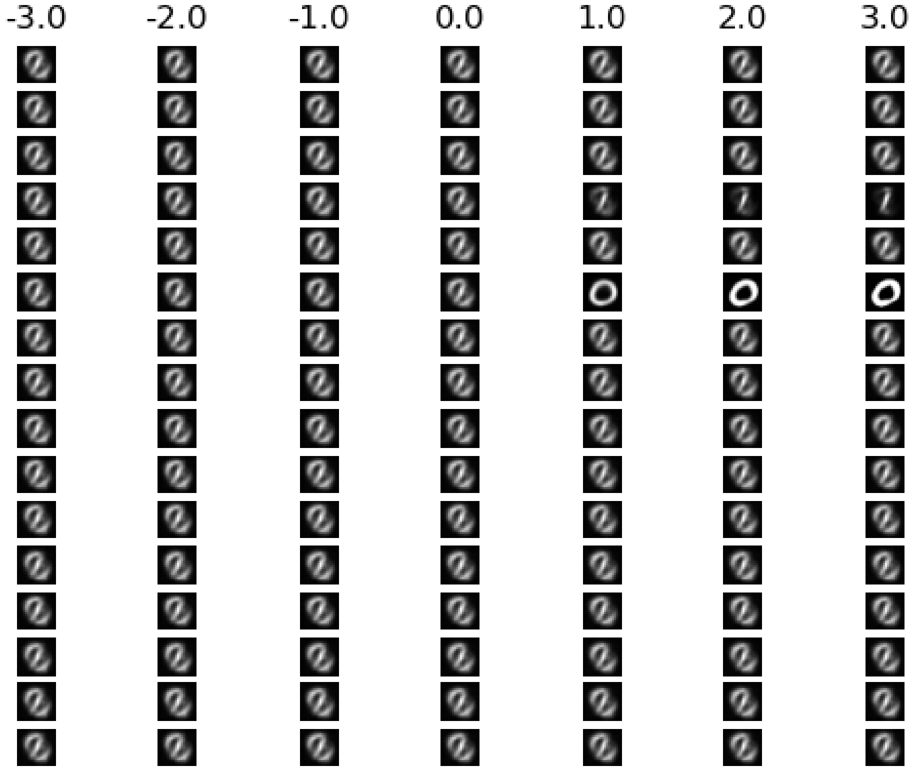

Figure 5: Sample reconstructed latent space after path entropy training.