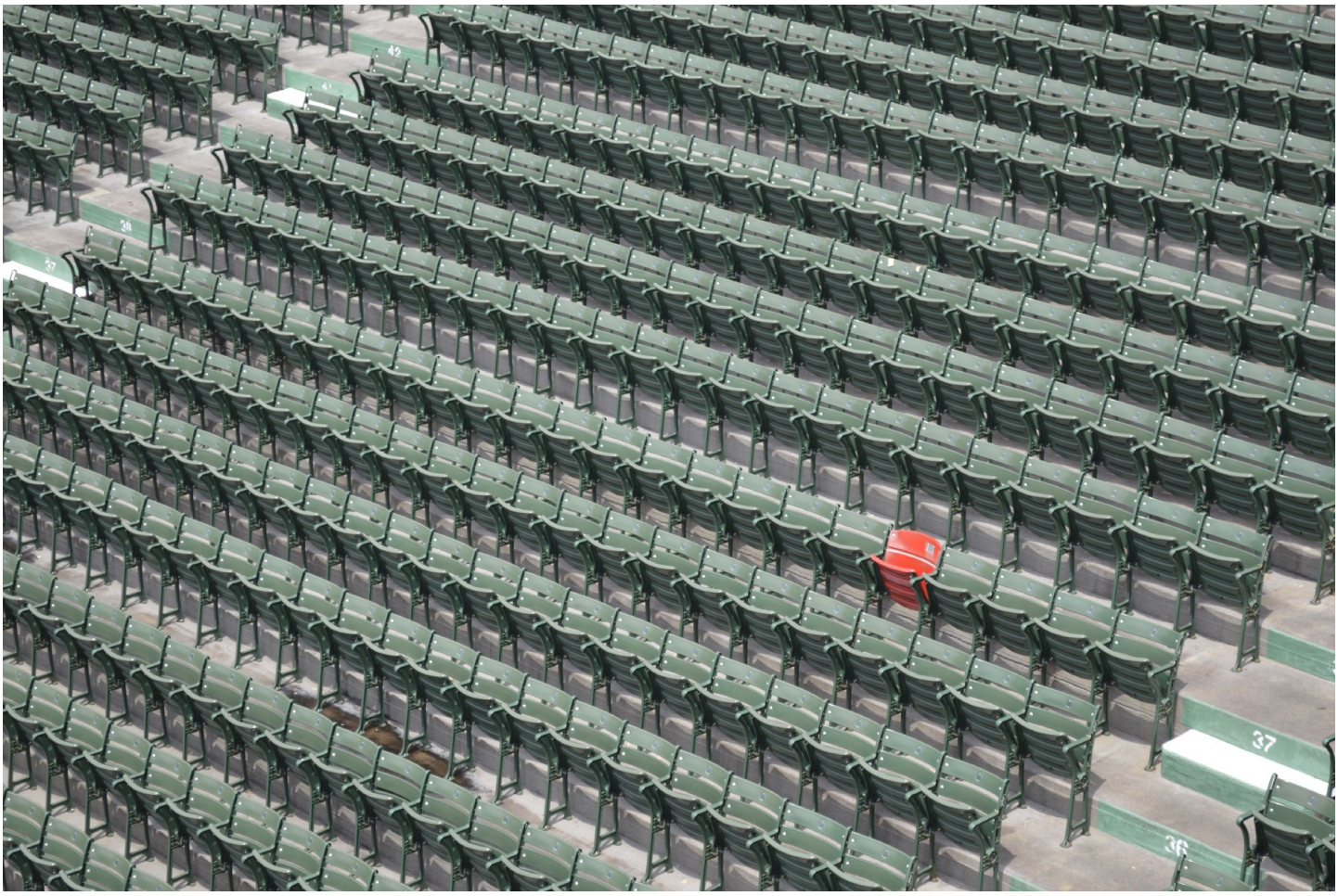


Photo by Veronica Benavides on Unsplash

Building Autoencoders on Sparse, One Hot Encoded Data

A hands-on review of loss functions suitable for embedding sparse one-hot-encoded data in PyTorch

Nick Hespe Sep 28, 2020 · 9 min read ★

past 3 decades. Having been shown to be exceptionally effective in embedding complex data, Autoencoders offer simple means to encode complex non-linear dependencies into trivial vector representations. But while their effectiveness has been proven in many aspects, they often fall short in being able to reproduce sparse data, especially when the columns are correlated like One Hot Encodings.

In this article, I'll briefly discuss One Hot Encoding (OHE) data and general autoencoders. Then I'll cover the use cases that bring about the issues with Autoencoders trained on One Hot Encoded Data. Lastly, I'll discuss the issue of reconstructing sparse OHE data in-depth, then cover 3 loss functions that I found to work well under these conditions:

1. CosineEmbeddingLoss
2. Sorenson-Dice Coefficient Loss
3. Multi-Task Learning Losses of Individual OHE Components

— that solve for the aforementioned challenges, including code to implement them in PyTorch.

One Hot Encoding Data

One hot encoding data is one of the simplest, yet often misunderstood data preprocessing techniques in general machine learning scenarios. The process binarizes categorical data with 'N' distinct categories into N columns of binary 0's and 1's. Where the presence of a 1 in the 'N'th category indicates that the observation belongs to that category. This process is simple in Python using the Scikit-Learn OneHotEncoder module:

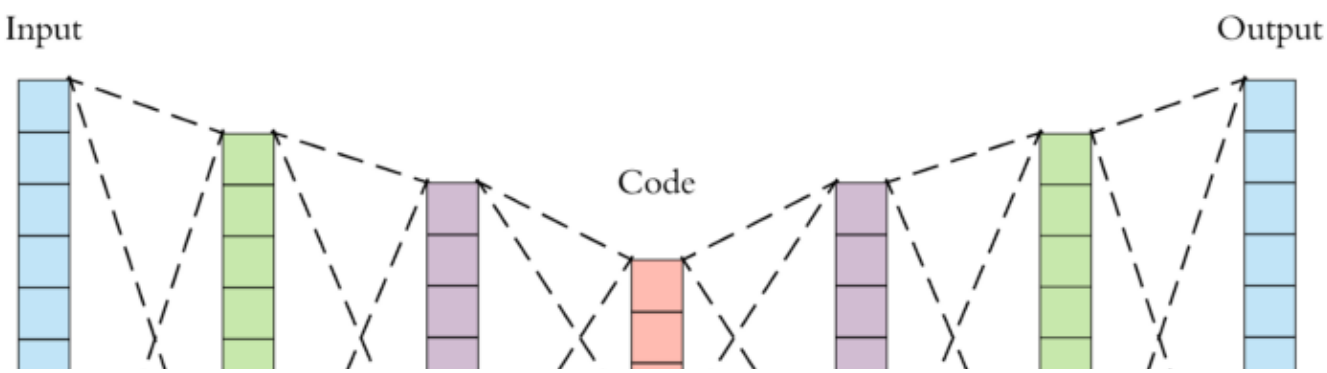
```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Instantiate a column of 10 random integers from 5 classes
x = np.random.randint(5, size=10).reshape(-1,1)

print(x)
>>> [[2][3][2][2][1][1][4][1][0][4]]

# Instantiate OHE() + Fit/Transform the data
ohe_encoder = OneHotEncoder(categories="auto")
```

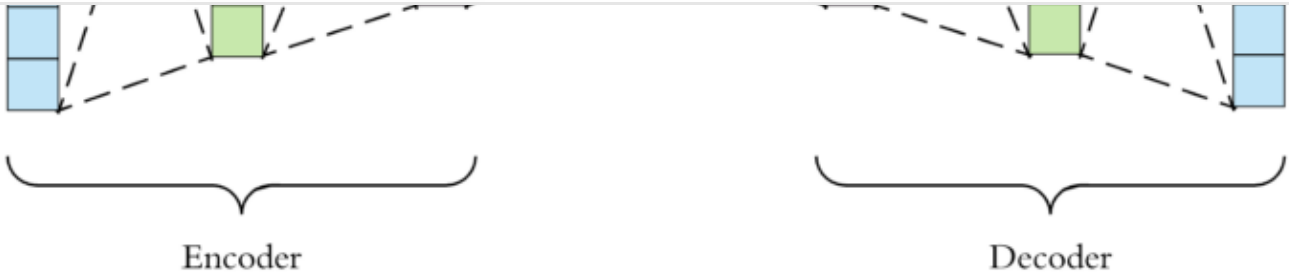
```
>>> matrix([[0., 1., 0., 0., 0.],
            [0., 0., 0., 1., 0.],
            [0., 0., 1., 0., 0.],
            [0., 0., 0., 1., 0.],
            [0., 0., 1., 0., 0.],
            [1., 0., 0., 0., 0.],
            [0., 0., 1., 0., 0.],
            [0., 0., 1., 0., 0.],
            [0., 0., 0., 1., 0.],
            [0., 0., 0., 0., 1.]])

print(list(ohe_encoder.get_feature_names()))
>>> ["x0_0", "x0_1", "x0_2", "x0_3", "x0_4"]
```

But while simple, this technique can sour fast if you are not careful. It can easily add superfluous complexity into your data, as well as change the effectiveness of certain classification methods on your data. For example, columns that are transformed into OHE vectors are now co-dependent, this interaction makes it difficult to represent aspects of the data effectively in certain types of classifiers. For example, if you had a column with 15 different categories, it would take an individual decision tree with a depth of 15 to handle the if-then patterns in that one hot encoded column. A great example of these issues can be found here if you're interested. Similarly, since the columns are co-dependent, if you use a classification strategy with bagging (Bootstrap Aggregating) and perform features sampling, you may miss the one-hot encoded column entirely, or consider only part of its component classes.

Autoencoders

Autoencoders are unsupervised neural networks that work to embed data into an efficient compressed format. It does this by utilizing an encoding and decoding process to encode the data down to a smaller format, then decoding that smaller format back into the original input representation. The model is trained by taking the loss between the model reconstruction (decoding), and the original data.





Courtesy of A. Dertat in his TDS Piece: Applied Deep Learning — Part 3: Autoencoders

Actually representing this network in code is also quite easy to do. We start with two functions: The ***Encoder*** Model, and the ***Decoder*** Model. Both 'models' are wrapped into a class called Network, which will encompass the entire system for our training and evaluation. Lastly, we define a function ***Forward***, which is what PyTorch uses as the entryway into the Network that wraps both the encoding and the decoding of the data.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Network(nn.Module):
   def __init__(self, input_shape: int):
      super().__init__()
      self.encode1 = nn.Linear(input_shape, 500)
      self.encode2 = nn.Linear(500, 250)
      self.encode3 = nn.Linear(250, 50)

      self.decode1 = nn.Linear(50, 250)
      self.decode2 = nn.Linear(250, 500)
      self.decode3 = nn.Linear(500, input_shape)

   def encode(self, x: torch.Tensor):
      x = F.relu(self.encode1(x))
      x = F.relu(self.encode2(x))
      x = F.relu(self.encode3(x))
      return x

   def decode(self, x: torch.Tensor):
      x = F.relu(self.decode1(x))
      x = F.relu(self.decode2(x))
      x = F.relu(self.decode3(x))
      return x

   def forward(self, x: torch.Tensor):
      x = encode(x)
```

```
def train_model(data: pd.DataFrame):
   net = Network()
   optimizer = optim.Adagrad(net.parameters(), lr=1e-3,
weight_decay=1e-4)
   losses = []

   for epoch in range(250):
     for batch in get_batches(data)
        net.zero_grad()

        # Pass batch through
        output = net(batch)

        # Get Loss + Backprop
        loss = loss_fn(output, batch).sum() #
        losses.append(loss)
        loss.backward()
        optimizer.step()
     return net, losses
```

As we can see above, we have an encoding function, which starts at the shape of the input data — then reduces its dimensionality as it propagates down to a shape of 50. From there, the decoding layer takes that embedding, then expands it back out to the original shape. In training, we take the reconstruction from the decoder **and take the loss of the reconstruction vs the original input.**

Problems With Loss Functions

So now we've covered the Autoencoder Structure and the One Hot Encoding Process we can finally talk about the problems associated with using One Hot Encodings in Autoencoders, and how to solve for this issue. When an autoencoder compares the reconstruction to the original input data, there must be some valuation of the distance between the proposed reconstruction and the true value. Typically, in cases where the values output is considered disjoint from one another one would use a cross-entropy loss or MSE loss. But in the case of our One Hot Encodings, there are several issues that make the system more complex:

1. The presence of a one in one column means that there must be a zero in its corresponding OHE columns. ***i.e. columns are not disjoint***

2. Sparsity in the input of the OHE vectors can lead to the system ***choosing to simply return 0's*** for most of the columns to reduce the error

offer a solution to one or both of the issues presented above, and code to implement them in PyTorch:

Cosine Embedding Loss

Cosine Distance is a classic vector distance metric that is used commonly when comparing Bag of Words representations in NLP problems. The distance is calculated by finding the cosine angle between the two vectors calculated as:

$$\alpha_{AB} = \frac{a * b}{|a| * |b|} = \sum \frac{a_i b_i}{\sqrt{\sum a_i^2}\sqrt{\sum b_i^2}}$$

Image by author

This method proves to be good at quantifying the error in the reconstruction of the sparse OHE embeddings because of its ability to evaluate the distance of the two vectors taking into account the deviations of the binary values in the individual columns. This loss is by far the easiest to implement in PyTorch as it has a pre-built solution in Torch.nn.CosineEmbeddingLoss

```
loss_function = torch.nn.CosineEmbeddingLoss(reduction='none')

# . . . Then during training . . .

loss = loss_function(reconstructed, input_data).sum()
loss.backward()
```

Dice Loss

Dice loss is an implementation of the **Sørensen–Dice coefficient** [2], which is very popular in the field of computer vision in segmentation tasks. In simple terms, it is a measure of overlap between two sets, and is related to the Jaccard distance between two vectors. The dice coefficient is highly sensitive to differences in column values in the vectors and is popular in Image Segmentation as it utilizes this sensitivity to effectively differentiate between pixel edges in the image. Dice loss follows the following equation:

$$2 * |X \cap Y|$$

Image by Author

For more information about the Sorensen Dice Coefficient — you can check out this medium post by Shuchen Du

PyTorch does not have an in-house Implementation of Dice Coefficient. But a good implementation can be found on Kaggle in their Loss Function Library — Keras & PyTorch [3]:

```
class DiceLoss(nn.Module):
    def __init__(self, weight=None, size_average=True):
        super(DiceLoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):

        #comment out if your model contains a sigmoid acitvation
        inputs = F.sigmoid(inputs)

        #flatten label and prediction tensors
        inputs = inputs.view(-1)
        targets = targets.view(-1)

        intersection = (inputs * targets).sum()
        dice = (2.*intersection + smooth)/
               (inputs.sum() + targets.sum() + smooth)

        return 1 - dice
```

Individual Loss Functions for Different OHE Columns

Lastly, you can treat each One Hot Encoded column as its own classification problem and take the loss for each of those classifications. This is a use case of a Multi-Task learning problem, where the autoencoder is solving for reconstructing the individual components of the input vector. This works best when you have several / all OHE columns in your input data. For example, if you had an encoded column with 7 categories as the first seven columns: you could treat that as a multi-class classification problem and take the loss as the cross-entropy loss of the sub-problem. You can then combine the losses of the sub-problems together and pass that backward as the loss of the batch as a whole.

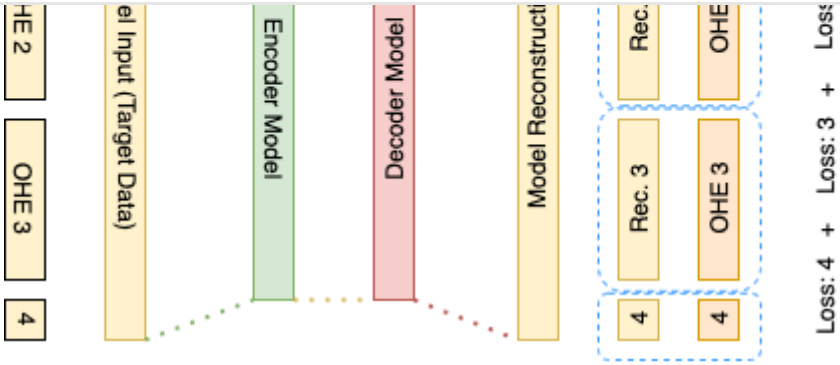


Image by Author

Below you'll see an example of this process with an example of three One Hot Encoded Columns, each with 50 categories.

```
from torch.nn.modules import _Loss
from torch import argmax

class CustomLoss(_Loss):
  def __init__(self):
    super(CustomLoss, self).__init__()

  def forward(self, input, target):
    """ loss function called at runtime """

    # Class 1 - Indices [0:50]
    class_1_loss = F.nll_loss(
        F.log_softmax(input[:, 0:50], dim=1),
        argmax(target[:, 0:50])
    )

    # Class 2 - Indices [50:100]
    class_2_loss = F.nll_loss(
        F.log_softmax(input[:, 50:100], dim=1),
        argmax(target[:, 50:100])
    )

    # Class 3 - Indices [100:150]
    class_3_loss = F.nll_loss(
        F.log_softmax(input[:, 100:150], dim=1),
        argmax(target[:, 100:150])
    )

    return class_1_loss + class_2_loss + class_3_loss
```

In the above code you can see how individual losses are taken on subsets of the reconstructed output, and then combined as a sum at the end. Here we use a ***negative***

Thank You!

In this article, we glanced over the concepts of One Hot Encoding categorical variables and the General Structure and Goal of Autoencoders. We discussed the downsides of One Hot Encoding Vectors, and the main issues when trying to train Autoencoder models on Sparse, One Hot Encoded Data. Lastly, we covered 3 loss functions that tackle the Sparse One Hot Encoding issue. ***There's no better or worse Loss for trying to train these networks, of the functions that I've presented there's no way to tell which one is right for your use case until you try them out!***

Below I've included a bunch of resources that go in-depth into the topics that I've discussed above, as well as some resources for the loss functions that I've presented.

Sources

1. D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation." Parallel Distributed Processing. Vol 1: Foundations. MIT Press, Cambridge, MA, 1986.

2. Sørensen, T. (1948). "A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons". *Kongelige Danske Videnskabernes Selskab*. **5** (4): 1–34. ***AND*** Dice, Lee R. (1945). "Measures of the Amount of Ecologic Association Between Species". *Ecology*. **26** (3): 297–302.

3. Kaggle's Loss Function Library: https://www.kaggle.com/bigironsphere/loss-function-library-keras-pytorch

Other Helpful Resources Mentioned

1. Issues With OHE Data: https://towardsdatascience.com/one-hot-encoding-is-making-your-tree-based-ensembles-worse-heres-why-d64b282b5769

2. Background of Bagging: https://towardsdatascience.com/ensemble-methods-bagging-boosting-and-stacking-c9214a10a205

3. A Great article about Dice Coefficient: https://medium.com/ai-salon/understanding-dice-loss-for-crisp-boundary-detection-bb30c2e5f62b

Autoencoder One Hot Encoding Sparse Data Loss Function Pytorch