Aakanksha NS · Follow
Jan 12, 2020 · 7 min read

# Deep Learning for Tabular Data using PyTorch

## On a multiclass classification problem

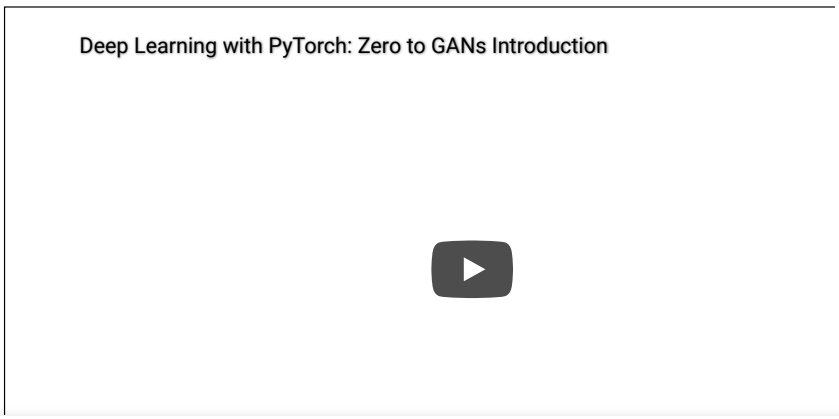

source

Deep learning has proved to be groundbreaking in a lot of domains like Computer Vision, Natural Language Processing, Signal Processing, etc. However, when it comes to more structured, tabular data consisting of categorical or numerical variables, traditional machine learning approaches (such as Random Forests, XGBoost) are believed to perform better. As expected, Neural nets have caught up and in many instances shown to be performing equally well or even better at times.

The easiest way to perform deep learning with tabular data is through the fast-ai library and it gives really good results, but it might be a little too abstracted for someone who's trying to understand what is really going on behind the scenes. Hence, in this article, I've covered how to build a simple deep learning model to deal with tabular data in Pytorch on a multiclass classification problem.

### A little background on Pytorch

Pytorch is a popular open-source machine library. It is as simple to use and learn as Python. A few other advantages of using PyTorch are its multi-GPU support and custom data loaders. If you're unfamiliar with the basics or need a revision, here's a good place to start:



**Animal_Shelter_Outcome**

shared using jovian.ml

## Dataset

I've used the Shelter Animal Outcomes Kaggle competition data:

**Shelter Animal Outcomes**

Help improve outcomes for shelter animals

www.kaggle.com

It's a tabular dataset consisting of about 26k rows and 10 columns in the training set. All columns except `DateTime` are categorical.

| | AnimalID | Name | DateTime | OutcomeType | OutcomeSubtype | AnimalType | SexuponOutcome | AgeuponOutcome | Breed | Color |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A671945 | Hambone | 2014-02-12 18:22:00 | Return_to_owner | NaN | Dog | Neutered Male | 1 year | Shetland Sheepdog Mix | Brown/White |
| 1 | A656520 | Emily | 2013-10-13 12:44:00 | Euthanasia | Suffering | Cat | Spayed Female | 1 year | Domestic Shorthair Mix | Cream Tabby |
| 2 | A686464 | Pearce | 2015-01-31 12:28:00 | Adoption | Foster | Dog | Neutered Male | 2 years | Pit Bull Mix | Blue/White |
| 3 | A683430 | NaN | 2014-07-11 19:09:00 | Transfer | Partner | Cat | Intact Male | 3 weeks | Domestic Shorthair Mix | Blue Cream |

Sample data from the train set

## Problem Statement

Given certain features about a shelter animal (like age, sex, color, breed), predict its outcome.

There are 5 possible outcomes: `Return_to_owner`, `Euthanasia`, `Adoption`, `Transfer`, `Died`. We are expected to find the probability of an animal's outcome belonging to each of the 5 categories.

## Data Preprocessing

Although this step depends largely on the particular data and problem, there are two necessary steps that need to be followed:

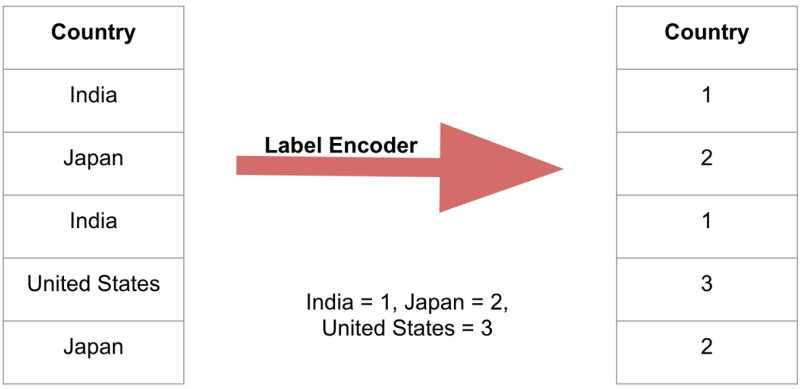
### Getting rid of `Nan` values:

`Nan` (not a number) indicates a missing value in the dataset. The model doesn't accept `Nan` values, hence they must be either deleted or replaced.

For numerical columns, a popular way of dealing with these values is to impute them with 0, mean, median, mode or some other function of the remaining data. Missing values might sometimes indicate an underlying feature in your dataset, so people often create a new binary column corresponding to the column with missing values to record whether the data was missing or not.

**Label encoding all categorical columns:**

Since our model can only take numerical inputs, we convert all our categorical elements to numbers. This means instead of using strings to represent categories, we use numbers. The numbers chosen to represent the categories should be in the range of 0 to the total number of different categories (including `Nan` ) in the column. This is so that when we create categorical embeddings for the column, we want to be able to index into our embedding matrix which would have one entry for each category. Here's a simple example of label encoding:




I've used the `LabelEncoder` class from the scikit-learn library to encode the categorical columns. You could define a custom class to do this and keep track of the category labels because you'd need them to encode test data too.

**Label encoding the target:**

We also need to label encode the target if it has string entries. Also, make sure you maintain a dictionary mapping the encodings to original values because you'll need it to figure out the final output of your model.

**Data Processing particular to the Shelter Outcome problem:**

Along with the above-mentioned steps, I did a little more processing for the example problem.

1. Removed the `AnimalID` column because it's unique and won't help in training.
2. Removed the `OutcomeSubtype` column because it's a part of the target but we're not asked to predict it.
3. Removed `DateTime` column because exact Timestamp of when the record was entered didn't seem like an important feature. In fact, I first tried to split it out into separate month and year columns but later realized that removing the column altogether gave me a better result!
4. Removed `Name` column because it had too many `Nan` values (more than 10k missing). Also, it did not seem like a very important feature in determining an animal's outcome.

**Note**: In my notebook, I stacked the train and test columns and then did the preprocessing to avoid having to do label encoding based on the train set labels on the test set (because it would involve maintaining a dictionary of encoded

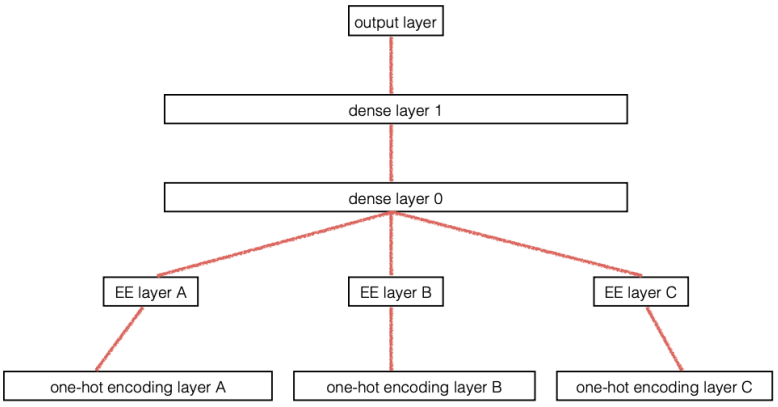
because it may leak some data from the test/validation sets to the training data and lead to an inaccurate evaluation of the model. For example, if you had missing values in a numerical column like `age` and decided to impute it with the average value, the average value should be calculated only on the train set (not stacked train-test-valid set) and this value should be used to impute missing values in validation and test sets too.

### Categorical Embeddings

Categorical embeddings are very similar to word embeddings which are commonly used in NLP. The basic idea is to have a fixed-length vector representation of each category in the column. How this is different from a one-hot encoding is that instead of having a sparse matrix, using embeddings, we get a dense matrix for each category with similar categories having values close to each other in the embedding space. Hence, this process not only saves up memory (as the one-hot encoding for columns having too many categories can really blow up the input matrix, also it is a very sparse matrix) but also reveals intrinsic properties of the categorical variables.

For example, if we had a column of colors and we find embeddings for it, we can expect `red` and `pink` to be closer in the embedding space than `red` and `blue`

Categorical embedding layers are equivalent to extra layers on top of each one-hot encoded input:




source: Entity Embeddings of Categorical Variables research paper

For our shelter outcome problem, we have only categorical columns but I'll be considering columns with less than 3 values as continuous. To decide the length of each column's embedding vector I've taken a simple function from the fast-ai library:

```
#categorical embedding for columns having more than two values
emb_c = {n: len(col.cat.categories) for n,col in X.items() if len(col.cat.categories) > 2}
emb_cols = emb_c.keys() # names of columns chosen for embedding
emb_szs = [(c, min(50, (c+1)//2)) for _,c in emb_c.items()] #embedding sizes for the chosen columns
```

embedding_size_shelter.py hosted with ♥ by GitHub view raw

our dataset while training and for effectively using the `DataLoader` module to manage batches. This involves overwriting the `__len__` and `__getitem__` methods as per our particular dataset.

Since we only need to embed categorical columns, we split our input into two parts: numerical and categorical.

```
class ShelterOutcomeDataset(Dataset):
    def __init__(self, X, Y, emb_cols):
        X = X.copy()
        self.X1 = X.loc[:,emb_cols].copy().values.astype(np.int64) #categorical columns
        self.X2 = X.drop(columns=emb_cols).copy().values.astype(np.float32) #numerical columns
        self.y = Y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X1[idx], self.X2[idx], self.y[idx]
```

shelter_outcome_dataset.py hosted with ❤ by GitHub view raw

We then choose our batch size and feed it along with the dataset to the DataLoader. Deep learning is generally done in batches. `DataLoader` helps us in effectively managing these batches and shuffling the data before training.

```
#creating train and valid datasets
train_ds = ShelterOutcomeDataset(X_train, y_train, emb_cols)
valid_ds = ShelterOutcomeDataset(X_val, y_val, emb_cols)

batch_size = 1000
train_dl = DataLoader(train_ds, batch_size=batch_size,shuffle=True)
valid_dl = DataLoader(valid_ds, batch_size=batch_size,shuffle=True)
```

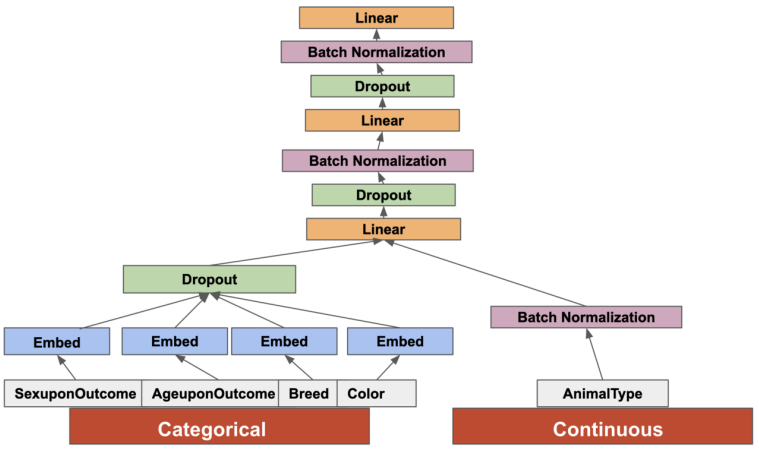
shelter_data_loader.py hosted with ❤ by GitHub view raw

To do a sanity check, you can iterate through the created DataLoaders to look at each batch:

```
i=1
for x1, x2, y in train_dl:
    print('batch_num:',i)
    i+=1
    print(x1, x2, y)
```

```
batch_num: 1
tensor([[   3,   25,  775,  117],
        [   4,   31,  775,   33],
        [   4,    5, 1520,  219],
        ...,
        [   3,   19,  775,  251],
        [   3,   19, 1023,   43],
        [   3,   19,   91,  311]]) tensor([[0.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
```

network. This picture demonstrates the model I've used:




Shelter outcome model

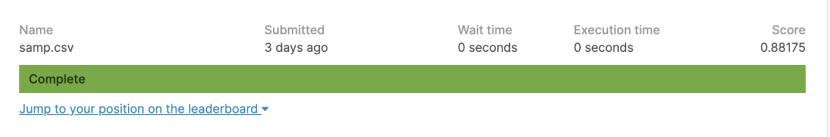
```
class ShelterOutcomeModel(nn.Module):
    def __init__(self, embedding_sizes, n_cont):
        super().__init__()
        self.embeddings = nn.ModuleList([nn.Embedding(categories, size) for categories,size in emb
        n_emb = sum(e.embedding_dim for e in self.embeddings) #length of all embeddings combined
        self.n_emb, self.n_cont = n_emb, n_cont
        self.lin1 = nn.Linear(self.n_emb + self.n_cont, 200)
        self.lin2 = nn.Linear(200, 70)
        self.lin3 = nn.Linear(70, 5)
        self.bn1 = nn.BatchNorm1d(self.n_cont)
        self.bn2 = nn.BatchNorm1d(200)
        self.bn3 = nn.BatchNorm1d(70)
        self.emb_drop = nn.Dropout(0.6)
        self.drops = nn.Dropout(0.3)


    def forward(self, x_cat, x_cont):
        x = [e(x_cat[:,i]) for i,e in enumerate(self.embeddings)]
        x = torch.cat(x, 1)
        x = self.emb_drop(x)
        x2 = self.bn1(x_cont)
        x = torch.cat([x, x2], 1)
        x = F.relu(self.lin1(x))
        x = self.drops(x)
        x = self.bn2(x)
        x = F.relu(self.lin2(x))
        x = self.drops(x)
        x = self.bn3(x)
        x = self.lin3(x)
        return x
```

shelter_outcome_model.py hosted with ♥ by GitHub view raw

## Training

Now we train the model on the training set. I've used the Adam optimizer to optimize the cross-entropy loss. The training is pretty straightforward: iterate through each batch, do a forward pass, compute gradients, do a gradient descent and repeat this process for as many epochs as needed. You can look at my notebook to understand the code.

inputs, we apply a Softmax function over our model output. I also made a Kaggle submission to see how well this model performs:




We've done very less feature engineering and data exploration and used a very basic deep learning architecture, yet our model has done better than about 50% of the solutions. This shows that this approach of modeling tabular data using neural networks is pretty powerful!

**References:**

1. https://www.usfca.edu/data-institute/certificates/fundamentals-deep-learning — Lesson 2
2. https://jovian.ml/aakashns/04-feedforward-nn