582 Coding Project

February 29, 2024

Justin Restivo
justin.restivo@yale.edu

Problem Selection

I chose to pursue the "Build a recommender system - predict new songs/playlists for listeners" option suggestion from the assignment suggestions. I chose to implement a naive version of Word2Vec, and my target implementation was in Rust, using the Burn crate and built with Nix. In this project, I had two primary goals:

- Understand Word2Vec well enough to implement it from scratch (e.g. without relying explicitly on tensorflow).
- Explore the maturity of the machine learning ecosystem in Rust.

Building the project

Download the MPD dataset into a directory, unzip it, then set the absolute path to the data in `DATA_DIR` before building.

If there are issues with following step(s), please reach out to me.

With nix

To build the executable to run the project on the dataset, first install nix using the determinate systems nix installer (or, if nix is already installed, enable flakes):

```
curl --proto '=https' --tlsv1.2 -sSf -L https://install.determinate.systems/nix | sh -
s -- install
```

Finally in the root directory of the project, run the executable:

```
nix develop -c cargo run --release
```

Without nix

Install cargo, then run `cargo run --release`

Design Decisions

At a high level, the problem I wanted to address was "how can we perform good playlist generation?"

Dataset Choice

I settled on looked through the Million Playlist Dataset (MPD) from spotify. This was appealing due to the sheer amount of data. MPD contained 1000000 playlists and 66346428 unique tracks, as well as an evaluation metrics for quality of the predictions. With the target of playlist generation, I wanted to find a reasonablely efficient algorithm to fully utilize this dataset. Word2vec is a quite natural choice here, since with an optimized version of the skip-gram model, theoretically I'd be able to, with a reasonable training time, generate a model and evaluate it.

Implementation Language

I chose to implement my project in Rust. The primary reason for this was to learn the Burn framework .

Burn is an alternative to tensorflow written in Rust that provides (at first glance) a similar API. The appealing part of Burn is the seamless integration over multiple backends including WGPU, which abstracts over a bunch of different graphics APIs, including Vulkan (applicable to my GPUs). WGPU also allows compilation to webassembly, and thereby ease of creation for performant webapps that do machine learning. See mnist demo.

I wanted to use Burn for the final project, since the goal for the final project is webapp.

Several other perks about Rust make it appealing for expressing algorithms in:

- A memory safe systems language, so fast and flexible when processing with data
- Expressive type system
- Fantastic tooling
- Static and strongly typed which makes runtime errors are rarer. I had expected that most runtime errors would be pushed back to compile-time errors that can be easily solved.

One caveat is that the Rust ecosystem is not as mature as Python. This seemed like an especially good opportunity to "implement state-of-the-art algorithms" as the instructions request. In other words, I would be able to learn more than I would have otherwise by relying on library functions.

Background

Let's discuss the literature for solving this problem using a word2vec style approach.

Word2Vec

Word2Vec is typically used for predicting future words based on a vocabulary and corpus of words. That is, Word2Vec is able to, given a context, predict future words (the Continuous Bag of Words model). And, alternatively, given a word, predict a context (the Skip-Gram model). This fits nicely in with music generation, where, given a context, we would like to generate a playlist.

The Original paper is the original paper that proposes these two architectures. However, we are interested in the Skip-Gram model specifically, since it is aimed at generating context words. This is precisely what we are interested in when generating playlists.

The skip-gram model begins with defining a vocabulary of all possible words. Then, it encodes inputs into "one-hot" vectors. That is, vectors where the ith word makes the ith index of the input vector 1. It then multiplies the words by an "embedding" layer that just embeds each word. This may be thought of as a matrix where each row. Notice that this is just a fully connected layer of a neural network with no bias. This step looks like (assuming vocabulary set V and embedding size $|S|$):

$$[\leftarrow \text{one hot vec, length } |V| \rightarrow] \begin{pmatrix} w_{00} & \cdots & w_{0|E|} \\ \vdots & \vdots & \vdots \\ w_{|V|0} & \cdots & w_{|V|\,|E|} \end{pmatrix}$$

Then, the output of the embedding layer is multiplied by another “projection” layer, which projects the embeddings onto an context window of words, then heirarchical softmaxed to determine the distribution of words. This is primarily for performance reasons, as previous models that included hidden layers and normal softmax tended to be quite slow. This project layer is a matrix of size $|S|\ x\ |nV|$ where nV is the size of the layer.

Optimizations

The original paper’s design was iterated on many times. This blogpost goes into a in-depth discussion of possible optimizations. Two interesting optimizations include:

- Avoiding blowup in the output vecs by simply batching terms, applying a projection layer, then having the output be a distribution over terms.
- Employ non-contrastive estimation to train the output as a binary classifier that “good” when hitting the output words, and “bad” otherwise. This employs negative samples with each batch (all pairs in a context window are passed through in each batch, the error is computed use MLE (maximum liklihood estimate), then back-propped on a per-batch basis).

Other approaches

TODO fill out with transformer and knearest neighbors

Implementation

I managed to only implement a very naive approach. The network architecture consisted of three layers. The context window was 1 on both sides, and no batching was done.

Loading the dataset

This was tricky, and indicative of performance issues I would encounter later. Loading the entire dataset was large (37GB). There was a large amount of extraneous data, so I would go through each json file in the dataset, parse it into memory, then parse the list of playlists into a reduced structure containng only the playlist songs. Each song was reduced down to a tuple of author and song name, which served as a unique identifier. If this data wasn’t stripped, the dataset (especially post processing) ended up overflowing my 128GB of ram.

At this point I could form the windows, which are the `MyDataItem` struct. These were batched in with only with one input/output tuple per batch.

There was 2180152 songs, which I thought wouldn’t be a problem given my hardware. But, the stack would overflow constructing an array of integers of size 2180152 on the stack. The solution here was to move to the heap.

Constructing the model

I opted for Adam over plain SGD because then the learning rate wouldn’t matter as much. My learning rate was (relatively arbitrarily) chosen to be 0.1.

Burn allows the user to define a `step` function for training data that maps the input data to a loss and output classification (in my case, a vector with the probability of observing the ith word fit in the ith index). Like Tensorflow, layers in the model are tracked, and, given a loss function, burn can automatically derive the backprop step to adjust weights. I chose cross entropy as my loss function. In classification problems (which this is), cross entropy loss compares the output probability density function to the target (trivially defined by the output label). This made it seem like a good choice.

The network architecture itself consisted of a embedding layer, denoted with a linear layer with no bias, and a hidden layer with bias.

I split the data up such that 1/32nd of it could be used as validation data and the remainder was training data.

I wanted to get something small to work before adding in batching over a larger window size and a more complex loss function. Ultimately, I ran out of time before implementing this.

Training the model

I prototyped on just one of the `json` files and used by CPU to train. This allowed me to debug through some sizing issues and get the network to train. However, this on its own was too sparse of a dataset, and the training set didn't help with the validation data. This makes sense because the playlists are essentially disjoint (due to too many songs). If I want this to work, I would need a smaller "toy" dataset. I ran out of time to find one of these.

It's worth noting that the model I constructed **does** function, even when trained on such a small set of data. It's able to, once trained, make predictions. However, those predictions aren't great (the songs aren't related). Nevertheless, I wrote the machinery for mapping back and printing out the data. This part extends straight forwardly into playlist generation, since we can pass in an input word and argmax over (or sample from) the output distribution to obtain the output word. We can then feed that word as input to the model and repeat the process.

Results / Discussion / Conclusion

I don't have much to show for results. I was at the point of being ready to train on the spotify MPD dataset, but wasn't able to get my GPUs (a 5700xt and a 560rx) to be recognized as usable devices. This I believe is due to some internal issue of compatibility between burn and AMD GPUs. I didn't have a smaller backup dataset, and ended up dumping my remaining time into trying to get this to work. I would expect the WGPU framework to use vulkan as a backend and notice my two GPUs, but it doesn't. I'm working with the burn team to figure out why.

The dataset is too large to train on its entirety on a CPU, and too sparse to use to train only a subset of it. I didn't realize this in the beginning but I'm not clear on what I could swap this out to be, either. I do think the network architecture would work on a smaller dataset.