# JAX or PyTorch: Financial News Sentiment Analysis

Kerem Haktan Kurt

September 2024

## 1 Introduction

This task aims to develop a model of financial sentiment analysis to categorize financial news. The development is done open-ended (easier to adapt and develop for further needs).

We were needed to develop the model using two separate frameworks PyTorch and JAX. And we were free to use a model of our choice (LSTM, BERT, LLM etc.).

And at first I tried to develop the model on BERT but I was unable to do so because of the extreme overfitting the model faced. And I was unable to solve the problem even though I spent a lot of time. And my final conclusion was the model was too small to understand Turkish literature in its sense, Let alone the financial market situations. I tried every possible scenario to overcome overfitting:

- Data augmentation with lowering the learning rate.
- Playing with batch and epoch sizes as many as I can.
- Initializing the weights according to their size in the training set. (approx. 1.1 for neg and 0.9 for pos)
- Stopping to overcome overfitting

But none of them worked. The model was only predicting positive for all test sentences even though getting an accuracy of around 80% on the training set.

Then I tried using XLM-R (Roberta) both base and large models. My selection of this model is because it has knowledge on Turkish literature, meaning that I needed to only train for the financial knowledge the model lacks. However, the base model was unable to understand connections and made biased predictions towards the positive (20 neg guesses while over 250 positive guesses). Then I tried training it for longer epochs with bigger batches but didn't work. After some time I run into overfitting issues, resulting in the model completely guessing. Even though the large model was a bit better, it is impossible to train these models onto a level where it can be used, the accuracies were around 55% while the recall is 100%.

After these attempts, I thought I needed an LLM model so that the model could understand the language easier and all that I needed to do was teach it financial knowledge and, boom. However, things went south quickly because of the size of the model I am using (8b parameter Llama3.1). The model went overfitting quickly and due to long times of training the effort was tremendous. And also I needed to quantize the model to load onto the GPU because 16GB VRAM was not enough. And after messing with it a couple of days I had enough and I changed my plans.

The new plan was to use BERT due to the ease of use and the small parameter size resulting in the short time training it. So I searched for something that can understand some sort of Turkish, which I can build the financial fundamentals on, and I found it in: `https://huggingface.co/dbmdz/bert-base-turkish-cased`. Which is a BERT model trained on a cased Turkish dataset. Which I definitely need the most. And built the model on it with ease.

## 2 Data Description

Our data was in txt format under two folders one named negative and the other one named positive. The texts are in Turkish and vary greatly in length while most of them did not reach 256 tokens for the ease of training.

The sources of our data come from news websites like Haberler, and sometimes from banks and investment capital firms like Yapıkredi, Garanti, Gedik, etc.

## 3 Data Preprocessing

First we need to load the data from the directories and label them with "positive" and "negative". After we did these steps I shuffled the dataset so that I didn't have an ordered list with first negatives and then positives because that would mess with the training process and would be an obstacle for teaching the model accurately predicting the sentiment.

Then I split the dataset into 80-20 train and test split. After splitting the datasets into train and test, I tokenized each of them fully before any training and evaluation part instead of dynamic tokenizing, because I didn't need such a thing. After that, I added a padding token so that all the inputs are of the same token size.

## 4 The Model

- BERTurk (dbmdz/bert-base-turkish-cased)
- Aim: BERT model with Turkish literature capacities
- History: BERTurk is a community-driven cased BERT model for Turkish.

## 5 Training

- **Epochs**: 2 - 3
- **Learning Rate**: 2e-5
- **Batch Size**: 2 - 16 (experimented a lot)
- **Optimizer**: AdamW
- **Gradient Clipping**: 1.0

### 5.1 PyTorch Training

The training has 2 epoch and after each epoch, I check for classic criteria to check if the model is going in a good direction (training loss, validation loss, accuracy, precision, recall, f1). Both epochs helped the model further specialize in financial news classification reaching an astonishing 90% accuracy after 2 epochs and I thought that a 3rd epoch was not needed, or may be used with a slightly lower learning rate. But 90% itself is perfect accuracy for many models especially for such a BERT model with 110m parameters, compared with billions of parameters on LLMs.

I have used PyTorch's built-in trainer, because why not? They built it and it does the same thing. I have tried forward pass, and gradient calculation while backpropagation, but they do the same thing. The interface is just nicer, you enter your hyperparameters, and ta da.

The training journey of my failed versions has a lot of history, I have tried gradient clipping, data augmentation, and weight balancing because some models were favoring positive over and over, etc...

Wandb of the run(link)

### 5.2 JAX Training

I tried training for 3 epochs using JAX framework and increasingly get better results. The training system is more complicated than that of a classical PyTorch training sequence. You have to handle states carefully in a functional programming manner.

## 6 Rewriting in JAX

The training system on JAX framework is a bit different than that of the PyTorch, because in JAX it lets you define individual training steps and, then an overall training function for each batch to call this function.

The gradient is calculated using a more functional programming approach meaning on the go. I needed to handle data myself to feed the model, instead of the PyTorch approach of DataLoader. Because the model itself does not have mutable states in it. The parameters and states are explicitly passed between functions while training. This requires careful handling of state updates.

Normally jax does not have a built-in optimizer however the optax library helped with that. Also, I needed to define my own loss function because the loss functions are just functional code pieces and need to differentiate using jax.grad(), which overall is more flexible, but more complicated to implement.

Also after the training is done, evaluation is done on the updated states, because of the functional programming style this is more forward.

## 7 Evaluation

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

$$F1 = 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$$

- **Overall Accuracy**: Measures the proportion of correct predictions.
- **Precision**: Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.
- **Recall**: Recall (also called Sensitivity or True Positive Rate) is the ratio of correctly predicted positive observations to all observations in the actual clas.
- **F1-Score**: F1-Score is the harmonic mean of Precision and Recall. It is used to provide a balance between Precision and Recall.

## 8 Results

### 8.1 PyTorch Framework

**Times Before & After:**

- **Evaluation Loss**: 0.7299 $\rightarrow$ 0.4569
- **Model Preparation Time**: 0.0068 s $\rightarrow$ 0.0061s
- **Steps per Second**: 0.304 $\rightarrow$ 0.305

**Start $\rightarrow$ Epoch 1 & Epoch 2:**

- **Accuracy**: 42.69% $\rightarrow$ 86.84% $\rightarrow$ 90.35%
- **Precision**: 0.7563 $\rightarrow$ 0.8730 $\rightarrow$ 0.9039
- **Recall**: 42.69% $\rightarrow$ 86.84% $\rightarrow$ 90.35%
- **F1 Score**: 0.2588 $\rightarrow$ 0.8691 $\rightarrow$ 0.9036

- **Training Loss**: 0.6945 → 0.5170
- **Validation Loss**: 0.4634 → 0.4569

The model showed some big improvements after fine-tuning. Accuracy jumped from 42.69% to 90.35% over two epochs, and both precision and recall improved a lot, hitting over 90%. This means the model made way fewer mistakes in its predictions. The F1 score also had a huge boost, going from 0.2588 to 0.9036, which shows that precision and recall are working well together. On top of that, both training and evaluation losses dropped, meaning the model got better at minimizing errors and making accurate predictions. Even small things like training speed and setup time saw a slight boost.

### 8.2 JAX Framework

**Start → Epoch 1 → Epoch 2 → Epoch 3:**

- **Accuracy**: 45.54% → 68.45% → 77.08% → 86.31%
- **Precision**: 0.7500 → 0.8990 → 0.9355 → 0.8840
- **Recall**: 0.0162 → 0.4811 → 0.6270 → 0.8649
- **Confusion Matrix**:

$$\begin{pmatrix} 150 & 1 \\ 182 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 141 & 10 \\ 96 & 89 \end{pmatrix} \rightarrow \begin{pmatrix} 143 & 8 \\ 69 & 116 \end{pmatrix} \rightarrow \begin{pmatrix} 130 & 21 \\ 25 & 160 \end{pmatrix}$$

At the beginning of the training, our model had a native bias toward the negative side, We can see this from the confusion matrix, and even after the first epoch the model started to correctly identify positive cases even though, it is still biased toward the negative. However, after the third epoch, the model had no bias towards no edges and correctly identified over 86% of our test case which, for a text classification model, for just a BERT model. Incredible. With only 1365 data, this achievement is really nice.

## 9 PyTorch vs JAX

- **Core**: PyTorch focuses on dynamic computation graphs while JAX is around static computation graphs, making Pytorch intuitive and flexible while JAX may be faster with its Accelerated Linear Algebra.

  JAX framework handles the operations in a more functional programming style, meaning that the functions get something and give something, generally do not change any variable and state in them. Like when you need to calculate and apply the loss with backpropagation you need to do the loss function and gradient function separately in JAX

  PyTorch is perfect for deep learning and is mostly used in that area however JAX extends beyond deep learning and is used in scientific research and computing.

- **Performance**: When it comes to performance overall JAX is way better, especially with TPUs. Native PyTorch implementation is very fast however JAX compilation can be faster with XLA (Accelerated Linear Algebra). Resulting in a let's say niche approach in JAX

- **Documentation & Environment**: The documentation of pytorch is superior and more organized, easier to find literally anything about deep learning than JAX. Also the environment of the Torch Is way more mature than the JAX environment. However, these are normal things because the JAX framework is fairly new.

- **Conclusion**: We can say that for large-scale applications including TPUs, working on researching, in need of flexible differentiation operations using JAX is superior to Pytorch. However, for a daily user, and a user wanting to deploy model here and there Pytorch is the go-to due to its ease of use and community support.

## 10 References

- `https://sjmielke.com/jax-purify.htm`
- `https://pytorch.org/torchtune/stable/tutorials/llama3.html`
- `https://jax.readthedocs.io/en/latest/quickstart.html`
- Hyper-Parameter Optimization: A Review of Algorithms and Applications
- Algorithms for Hyper-Parameter Optimization
- `https://huggingface.co/docs/transformers/en/model_doc/llama#transformers.FlaxLlamaModel`
- `https://huggingface.co/docs/transformers/en/tasks/sequence_classification`