Project Title

Machine Learning using Spark Streaming.

Dataset

CIFAR10 Image Classification Dataset. Contains 60k RGB images of size 32x32 with 10 classes.

Design Details

The whole project is organized into different folders. main.py file is the main driver program that contains all the model instantiations, list of preprocessing to be applied on input image stream. The driver program also instantiates a Trainer object that is responsible for training and validating the model for the user. The model to be trained will be passed as an object to the Trainer object. SparkConfiguration and TrainingConfiguration objects are also passed to the trainer. These configurations are responsible for setting up the Spark Session attributes and Trainer class attributes. This Trainer class is entirely responsible for accepting the input data stream, applying the preprocessing steps and then performing incremental learning on the model passed to the class.

It is in the Trainer Class that the SparkContext, StreamingContext gets created and used for the entire spark session. Embedded in the trainer class is the DataLoader class. The main functionality of DataLoader is to use the current SparkContext and StreamingContext, obtained from the trainer class, and start a SocketTextStream on the specified port. DataLoader reads the input streaming through the socket and applies the map transformations to convert the data from json to the required RDD format. After applying the transformations, the input is preprocessed using the list of preprocessing steps given by the main.py file. The preprocessed stream is then returned back to the Trainer Class.

RDDs are extracted from the preprocessed stream obtained from DataLoader and are passed to the train() function (If trainer is in “train" mode else calls predict() function). The RDD is converted to a spark dataframe and is passed to the model given by the user.

All the models are defined in their own class. These classes follow an API in order to keep things uniform and make it easy for the trainer to train the models. These classes internally hold the “raw model” itself along with some helper attributes that’s necessary for training the model. Before training the model, the trainer class the configure_model() function of the model passed to the trainer class. This sets up all the raw model attributes like batch_size, max_iter, layers, and so on. Now, the model is ready to be trained. DFs are passed to the train() function of the model class which are then converted to Numpy arrays and passed to the model. The model is partially fitted on each batch of RDD and metrics such as accuracy, loss, precision, recall and F1 scores are calculated for the current batch and returned to the trainer class. The trainer class then stores these values in its attributes.

Trainer class also performs model checkpointing and loading based on the configuration options passed to it from the main.py file. It also logs all the training metrics on every model save. This allows us to plot the metrics and visualize the training statistics.

We have also implemented Transfer Learning using Sparkdl. ResNet50 was used to extract features for all the images in both the training and testing sets. Originally, we had planned to extract features and use the features to train the model in a single pipeline. However due to extremely large processing time per batch, we decide to cache all the features to disk and then load the same to train the model. This method is usually done in the field of Computer Vision and hence we thought to implement the caching of features. Doing this allowed us to use the extracted features to train different models.

Surface Level Implementation Details

`transforms/` : Consists of all the preprocessing functions. Each preprocessing function is a class by itself that follows an API to keep uniformity across the board. These functions operate on 3D Numpy arrays only. Finally, the transforms.py is a thin wrapper that calls all the preprocessing functions based on list of transformations passed from the main.py one by one.
This serves as a Preprocessing and Data Augmentation part of the project.

`models/` : Consists of all models. Each model is a class that encapsulates the raw model. Each model class follows an API that makes it easy for the trainer to train and test the model.

`trainer.py & dataloader.py` : Main workhorse of the project. Performed all the activities mentioned in the design details. Also contains classes for initializing the Spark and Training configuration options.

`stream.py` : Data streaming script that streams the data from disk over a TCP connection. It also implements effective queueing to reduce dropping of batches and reduce memory consumption.

`plots/` : Contains a plotting script that will read the training metric logs and generate a graph for visualization

`main.py` : Specifies the spark and training configs. Also specifies the list of transformation steps to be applied as a part of preprocessing. Finally, main.py calls Trainer.train() function with appropriate model to be trained and hands over all the responsibility of model training to Trainer.

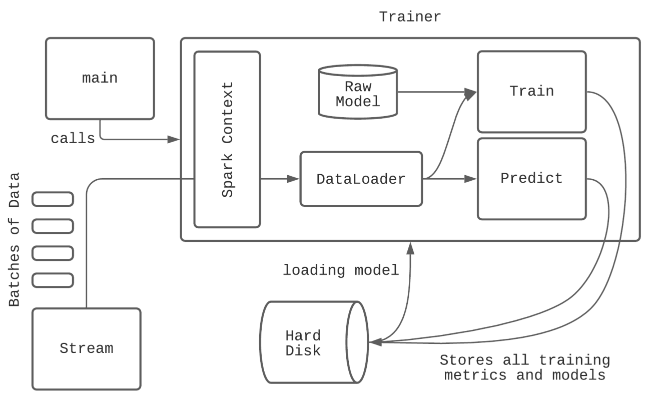


Fig 1: Workflow Diagram

Reason for this Design

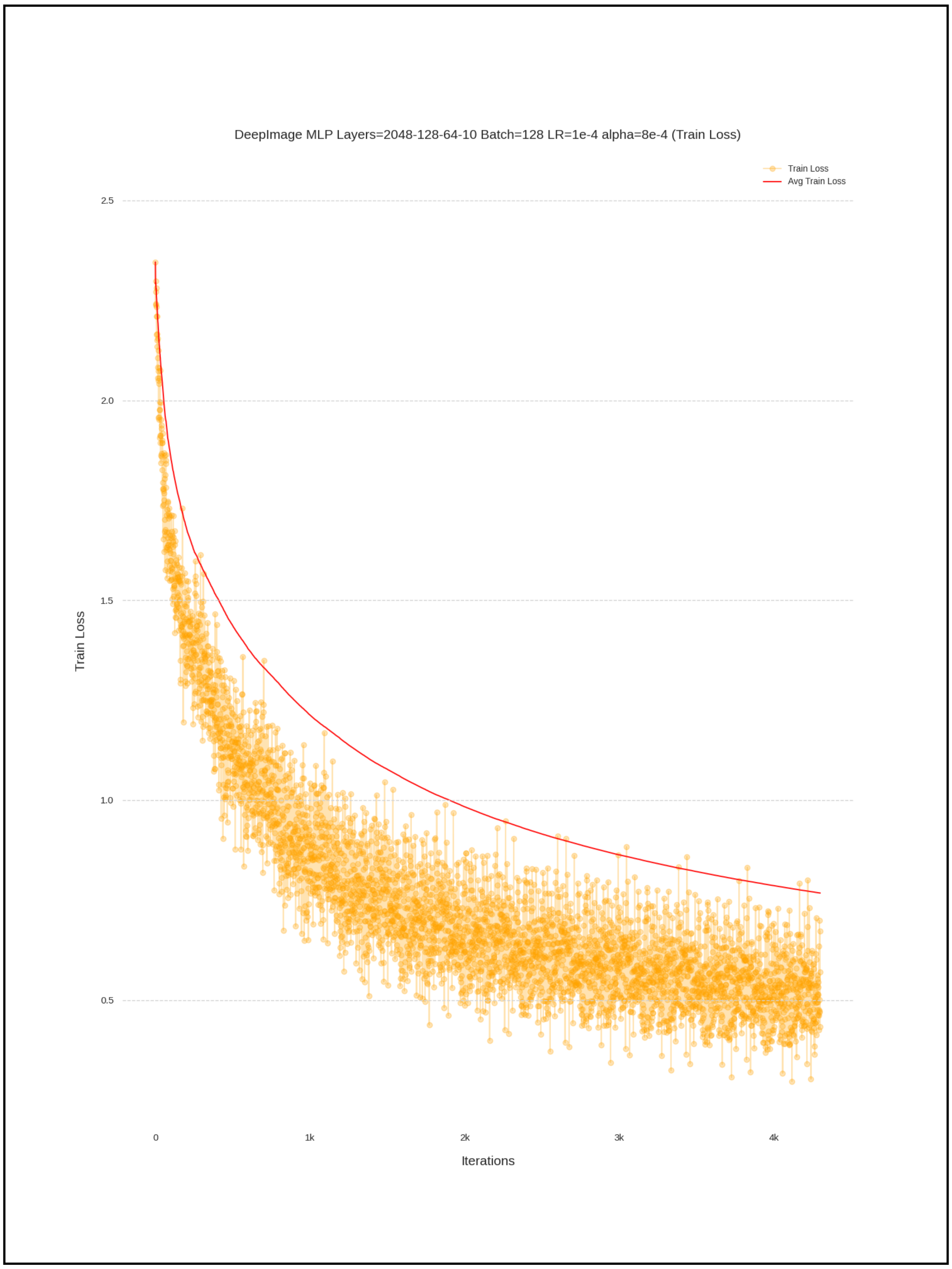
The Trainer and the DataLoader are at the core of this project. They are responsible for 80% of all the things in the project. Using this approach, allowed us to experiment quickly with different models, batch sizes. In our experiments, this design approach drastically reduced the memory consumption and reduced the training time of the model. Automatic logging and checkpointing during training sessions allowed us to train multiple models in parallel (0-3 cores 1st model and 4-7 cores second model). Trainer class also provided a modest level of abstraction to us and helped us to concentrate on the model itself.

Takeaways from Project

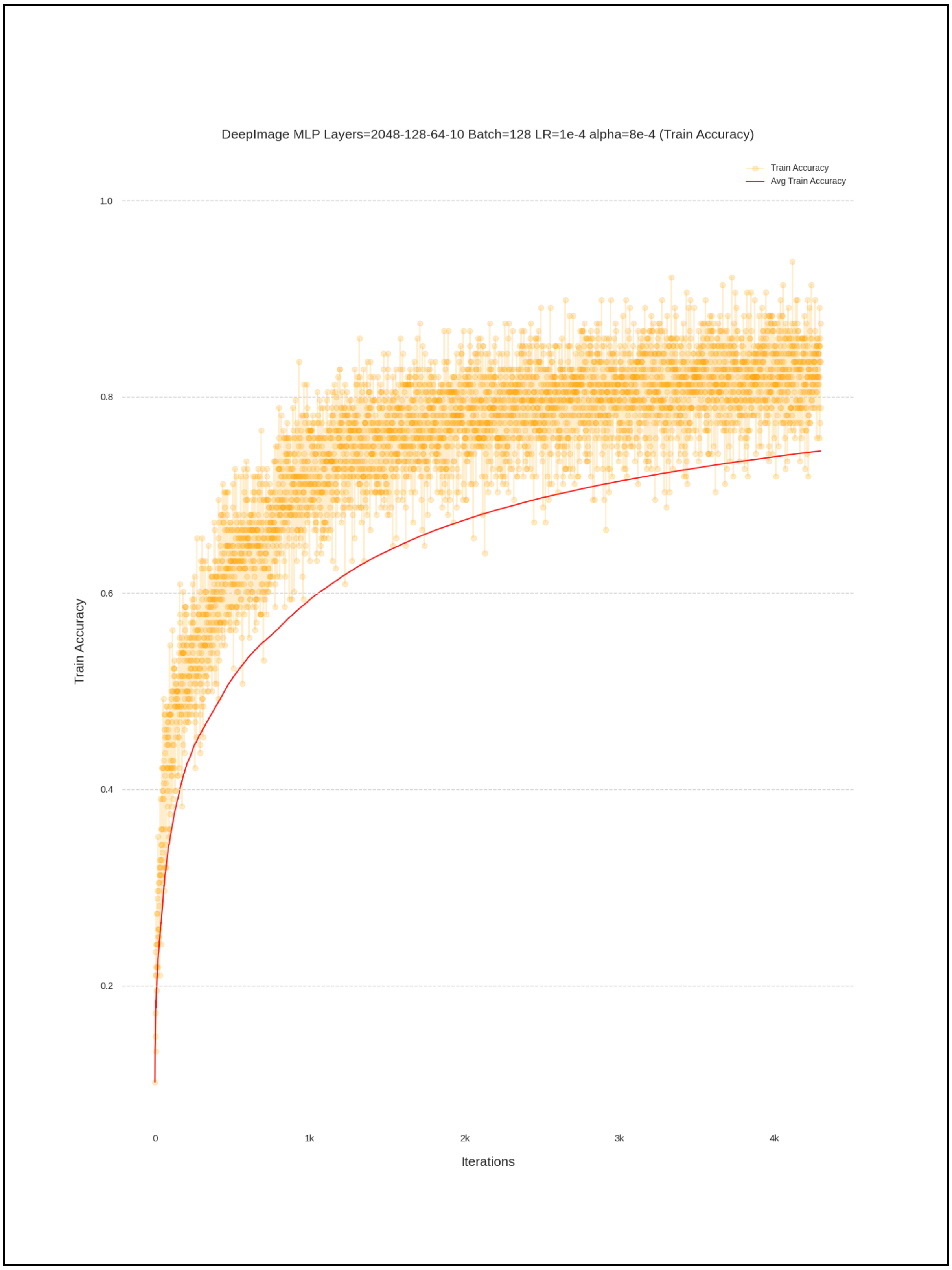
By doing this project, we learnt a lot about Spark and distributed computing in general. The project threw few challenges like streaming and and parsing the dataset, memory management and performance optimizations. We also learnt a lot about project structuring which made things a lot easier for training the model. We also tried to setup a multinode cluster locally, to reduce the training time. However we couldn't connect the nodes properly due to architecture limitations and lack of software compatibility. All in all, it was a great experience.

Graphs

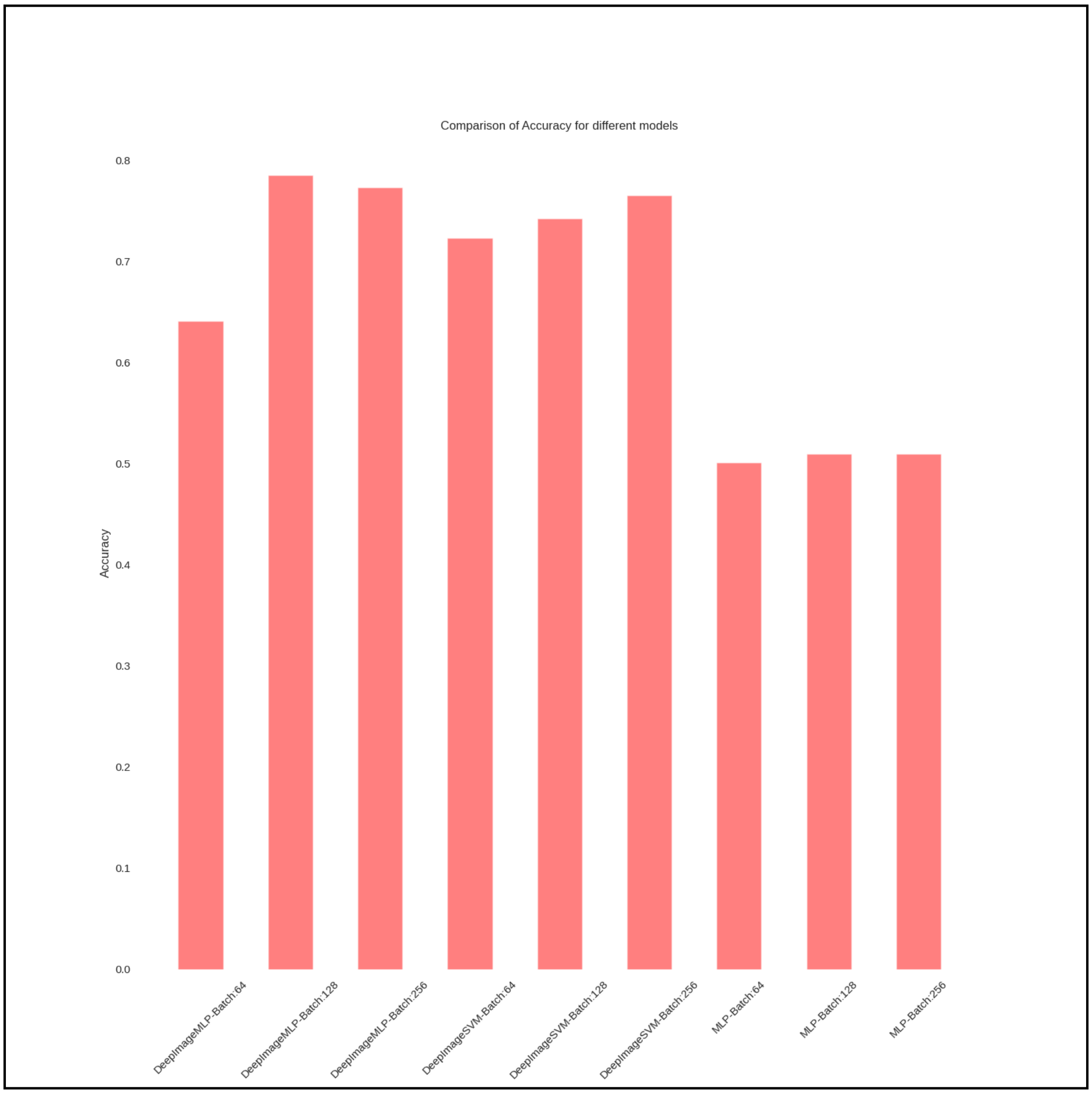
Training Loss Graph for DeepImage MLP



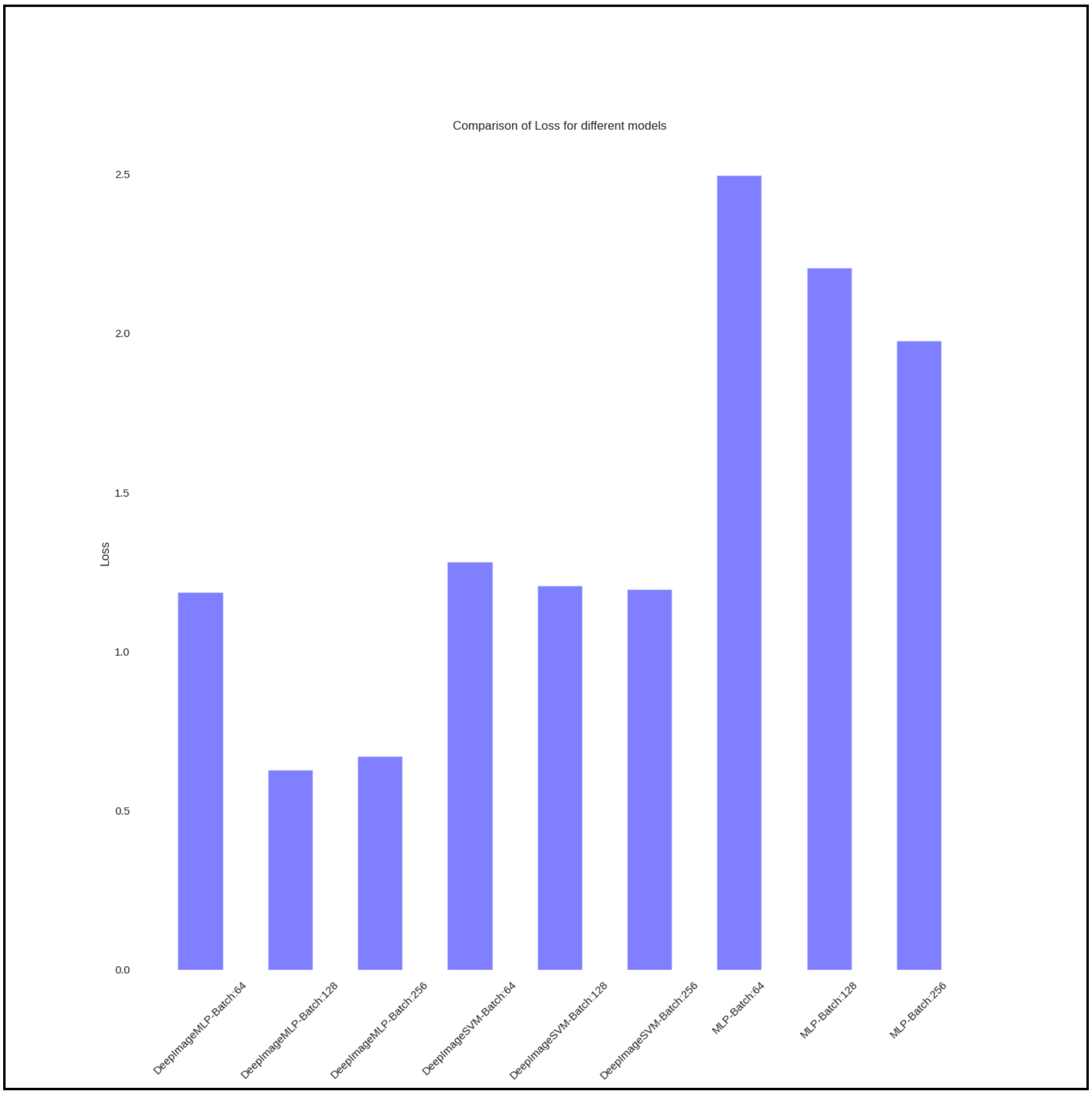
Training Accuracy Graph for DeepImage MLP



Test Accuracy Comparison for all models



Test Loss Comparison for all models


Comparison of Loss for different models
Loss
DeepImageMLP-Batch:64
DeepImageMLP-Batch:128
DeepImageMLP-Batch:256
DeepImageSVM-Batch:64
DeepImageSVM-Batch:128
DeepImageSVM-Batch:256
MLP-Batch:64
MLP-Batch:128
MLP-Batch:256