

Deep Learning - Major Exam

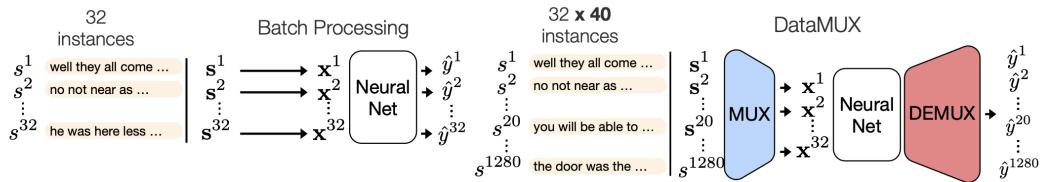


Report

Mukul Shingwani (B20AI023)

Paper link: [DataMUX: Data Multiplexing for Neural Networks](#)

- In this paper, the authors introduced **data multiplexing** (**DataMUX**), a technique that enables deep neural networks to **process multiple inputs simultaneously** using a single compact representation. DataMUX demonstrates that neural networks are capable of generating accurate predictions over *mixture*s of inputs, resulting in increased inference throughput with minimal extra memory requirements.
- The approach uses two key components –
 - a **multiplexing layer** that performs a fixed linear transformation to each input before combining them to create a ‘mixed’ representation of the same size as a single input, which is then processed by the base network, and
 - a **demultiplexing layer** that converts the base network’s output back into independent representations before producing predictions for each input

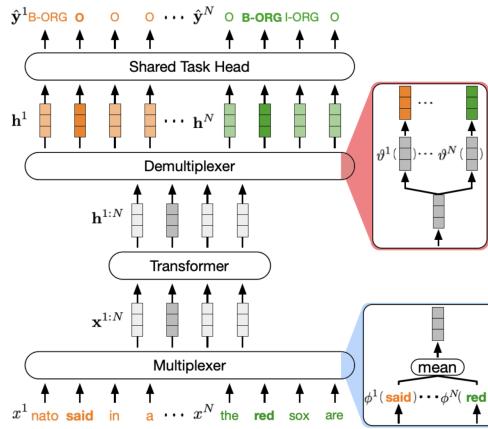


Schematic illustrating DataMUX technique in comparison to traditional mini-batch processing in neural networks

- Here, DataMUX uses a multiplexing layer (MUX), to multiplex 40 instances into a single representation, passing only 32 inputs to the neural network. The demultiplexing layer (DEMUX) uses the output to generate predictions for all 1280 instances.
- Due to the use of a single combined representation, DataMUX can increase network inference throughput with minimal time and memory overhead.
- The **key challenge for DataMUX** lies in **effectively compressing** a mixture of signals so that it can be processed by a neural network without sacrificing task performance. To this end, our procedure involves the addition of a multiplexing layer and a demultiplexing layer to both ends of a vanilla neural network.

Method

- The primary goal of the researchers is to process a set of input instances simultaneously over a shared neural network (multiplexing) with minimal overhead during inference. To achieve this, they designed DataMUX to consist of three components:
 - a **multiplexer module** to combine the multiple input instances into a superposed representation,
 - a **neural network backbone** to process mixed representations, and
 - a **demultiplexing module** to disentangle the processed representations for individual prediction.



The above figure shows an illustration of DataMUX applied to a Transformer model involves the processing of a set of input sentences (x_1, x_2, \dots, x_N), each of length L. The embeddings of each sequence x_i undergo a **transformation** ϕ^i such that the same transformation is applied to every token in x_i . The multiplexing operation then aggregates the sequences by averaging over each position to generate a single combined sequence $x^{1:N} \in R^L \times d$, where d is the embedding size. This combined sequence is then passed on to the central Transformer model for processing. After processing, a demultiplexing operation is performed on the Transformer model's output $h^{1:N} \in R^L \times d$ to generate hidden representations h_1, h_2, \dots, h_N , corresponding to inputs x_1, x_2, \dots, x_N , respectively. Finally, these hidden representations are used to generate predictions for a particular task, such as natural language inference tasks (MNLI) or sentence similarity task (QQP), using a shared task prediction head.

Multiplexing

The multiplexer module is designed to **combine a batch of input instances**, such as images or sentences, into a more compact and order-dependent representation, denoted as $x^{1:N}$. This allows for effective demultiplexing after processing and helps distinguish intra-sequence interactions in the case of sequenced inputs. For each input x_i with index i in the input tuple, the multiplexer module applies a transformation function $\phi^i (R^d \rightarrow R^d)$ to the instance. Finally, all **inputs are averaged** to generate a **single multiplexed representation**.

$$\mathbf{x}^{1:N} = \Phi(\mathbf{x}^1, \dots, \mathbf{x}^N) = \frac{1}{N} \sum_{i=1}^N \phi^i(\mathbf{x}^i).$$

DeMultiplexing

The neural network backbone produces a multiplexed hidden representation $h^{1:N}$ of the input $x^{1:N}$. To generate predictions for each input, the multiplexed representation $h^{1:N}$ must be disentangled into N individual hidden representations, h_1, \dots, h_N . This is achieved by applying a demultiplexing function ϑ^i to obtain each h_i . Specifically,

$$\mathbf{h}^i = \vartheta^i(\mathbf{h}^{1:N}), \forall i \in [1, \dots, N].$$

Multiplexing for Transformers (T-MUX)

- The capabilities and limitations of data multiplexing for the Transformer architecture on various text classification tasks were evaluated by the researchers. A 12-layer Transformer model with a hidden dimension size of 768 and 12 self-attention heads, built on the Huggingface (2019) framework, was used for applying DataMUX, which resulted in a model named T-MUX. The `baseline mode` used for comparison: (B1) a 12-layer vanilla Transformer with a hidden dimension size of 768.
- The models and baselines are evaluated on two types of text classification tasks:
 - `token-level classification`
 - `sentence-level classification`.
- For token-level classification, the CoNLL-2003 `Named Entity Recognition (NER)` task is used. This task is particularly challenging for data-multiplexing models since they must maintain individual token disentanglement while producing representations that can solve the task.
- For sentence-level classification, the models are evaluated on a subset of the `General Language Understanding Evaluation (GLUE) benchmark`, including the natural language inference tasks `MNLI` and sentence similarity task `QQP`.
- <https://huggingface.co/datasets/conll2003> and <https://huggingface.co/datasets/glue> were used to get the datasets, and It got downloaded automatically from the datasets hub. We could also provide our own csv/json/txt file incase we wanted to train and test on that.
- By evaluating on a variety of sentence-level tasks, the capabilities of data multiplexing neural networks on tasks requiring contextual information aggregation are better understood. To learn a task head on top of the demultiplexed [CLS] token representation, a [CLS] token is prepended to all sequences, as in previous works.
- The total loss is a combination of the task loss and retrieval loss (they use alpha = 0.1 in all experiments and so did we):

$$\mathcal{L} = (1 - \alpha)\mathcal{L}_{\text{Task}} + \alpha\mathcal{L}_{\text{Retrieval}}$$

NOTE : I tried running and training the model from scratch but it was too time consuming so I used the `pretrained models` and the provided code in the repo to get the results and do inferencing

Results

NOTE: In the paper the values of N were {2,5,10,20,40} and evaluation of accuracy on tasks such as MNLI, QNLI, SST2, NER, QQP were done and two mixing variants. We, have used N = {2,5,10}, tasks = {MNLI, QQP, NER} and 2 muxing variants {gaussian_hadamard, random_ortho} due to limited time and computational resources (with increasing N it was getting too time costly). These results were calculated in computers on ANVISA lab and not in Colab.

- MNLI**

Model	Baseline	T-MUX(N=2)	T-MUX(N=5)	T-MUX(N=10)
Avg. Test Accuracy (Ortho)	59.8%	61.5%	58.8%	56.3%
Avg. Test Accuracy (Hadamard)	59.8%	63.4%	60.2%	59.7%

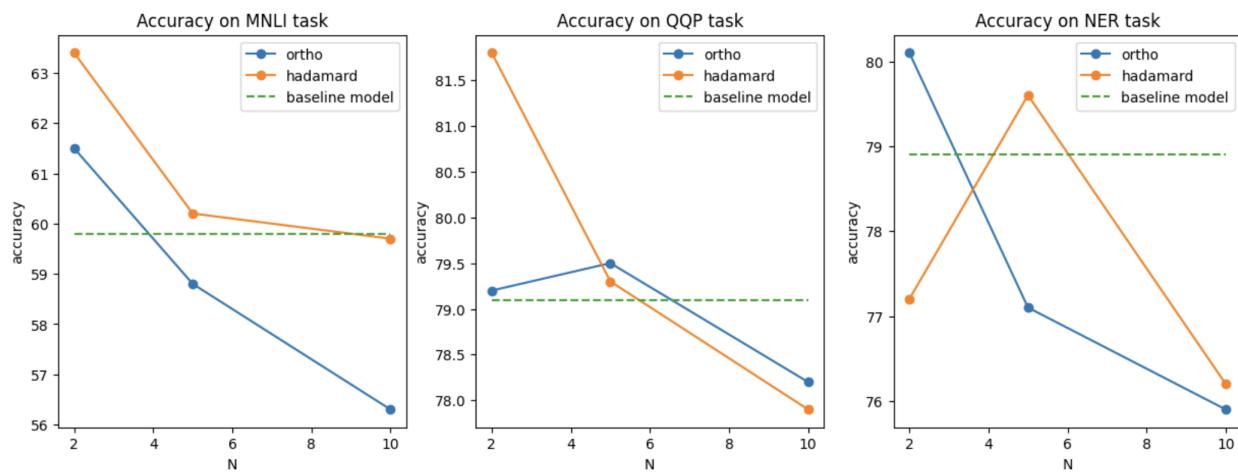
- QQP**

Model	Baseline	T-MUX(N=2)	T-MUX(N=5)	T-MUX(N=10)
Avg. Test Accuracy	78.9%	79.2%	79.5%	78.2%

(Ortho)				
Avg. Test Accuracy (Hadamard)	78.9%	81.8%	79.3%	77.9%

- **NER**

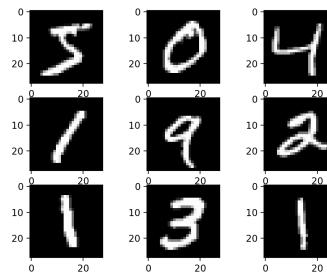
Model	Baseline	T-MUX(N=2)	T-MUX(N=5)	T-MUX(N=10)
Avg. Test Accuracy (Ortho)	78.9%	80.1%	77.1%	75.9%
Avg. Test Accuracy (Hadamard)	78.9%	77.2%	79.6%	76.2%



- In **graph 1**, the gaussian_hadamard transformation works almost as good as the baseline model for this task with max drop being 1%
- In **graph 2**, Both the transformations work almost as good as the baseline model for this task a maximum of 1.75% drop in accuracy
- In **graph 3**, Both the transformations work almost as good as the baseline model for this task too, with about 3% drop in accuracy

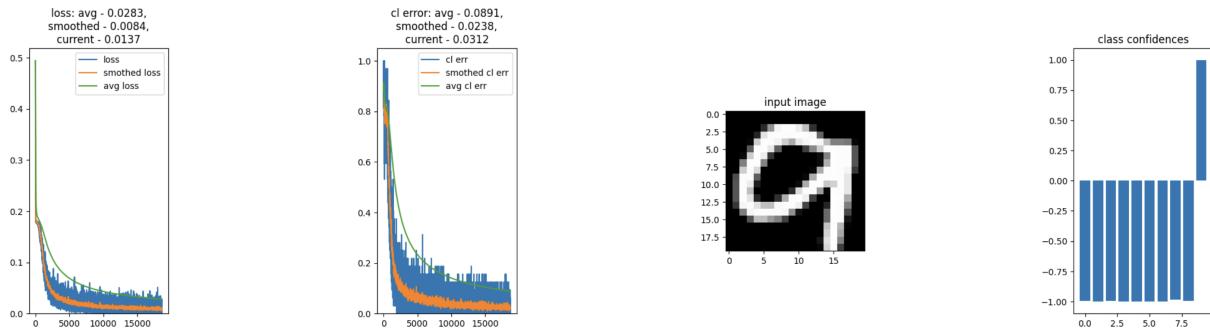
Multiplexing for CNNs

For testing them, authors used the MNIST dataset



Single Instance Architecture

The `LeNet` architecture was used for this and it was trained using SGD optimizer with learning rate of 0.1 for 10 epochs (as per the authors code)



Multiple Instance Architecture (MUX)

In this code, Authors have take a parameter called `mod` this is a parameter of the `MuxLeNet` class constructor and it is used to specify the type of projections that will be used to transform the input data.

The following values can be assigned to `mod` :

- `'Gaussian'` : use Gaussian random matrices for the projections.
- `'graded'` : use identity projections multiplied by a graded scalar factor.
- `'identity'` : use identity projections.
- `'learnable'` : use learnable projections.
- `'SO(N)'` : use special orthogonal matrices as projections.
- `'learnable-W'` : use learnable matrices as projections.
- `'nonlinear'` : use learnable convolutional filters with hyperbolic tangent activations.
- `'nonlinear-expand'` : same as `'nonlinear'` but with an expanded number of output channels.
- `'rotation'` : rotate the input data in fixed steps.
- `'remap'` : remap the input data in fixed steps.

The `mod` parameter is used to initialize the `rop` attribute, which is a list containing the projection matrices or other parameters that will be used for the input data transformation. The `proj` method is then used to apply the projections to the input data.

Results

We have used `three types of projections` for result reproduction task and those are `Rotation`, `Gaussian`, and `non-linear`.

Now we'll see the result of each one them one by one through some plots and accuracy table (N specifies the number of inputs)

- **Using Rotation transforms**

Model	Baseline	MUX(N=1)	MUX(N=2)	MUX(N=4)	MUX(N=8)	MUX(N=16)
Avg. Test Accuracy	97.59%	97.83%	92.85%	58.95%	36.61%	23.21%

- **Using Gaussian transforms**

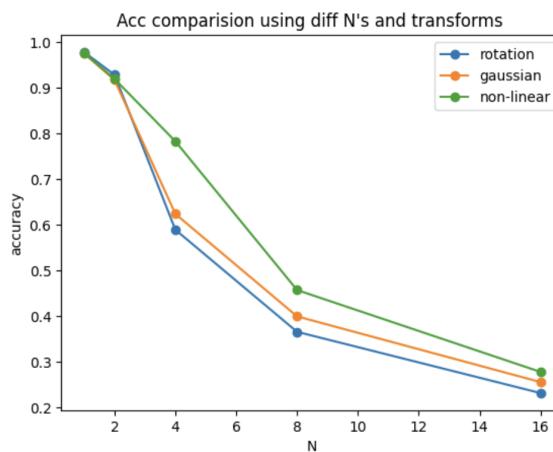
Model	Baseline	MUX(N=1)	MUX(N=2)	MUX(N=4)	MUX(N=8)	MUX(N=16)

Avg. Test Accuracy	97.59%	97.50%	91.74%	62.41%	39.99%	25.61%
--------------------	--------	--------	--------	--------	--------	--------

- **Using Non-linear transforms**

Model	Baseline	MUX(N=1)	MUX(N=2)	MUX(N=4)	MUX(N=8)	MUX(N=16)
Avg. Test Accuracy	97.59%	97.67%	91.96%	78.3%	45.75%	27.84%

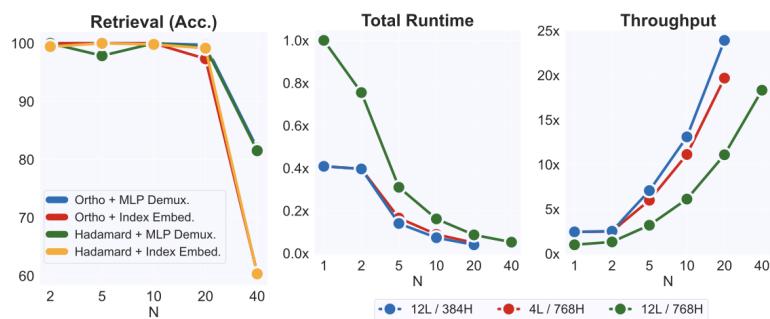
- So, it's quite evident that for `CNNs multiplexing doesn't perform well`, especially after N=2
- It is likely because the transformation destroys the property of spatial locality that CNNs rely on. The researchers therefore also explore N two-layer convolutional networks with a tanh activation as their multiplexing transformations (`(CNN + Nonlinear)`). It is noted that even with this transformation, the dimensionality of the multiplexed representation still remains equal to the dimensionality of a single input.
- It is found that for $N \leq 4$, the performance is above 70%, which is significantly better than CNN+Rotation and CNN+Gaussian, but the performance drops rapidly for $N > 4$. Also, with increase in N, time also goes on increasing.
- Overall, multiplexing for CNNs seems to be more challenging than Transformers, as evidenced by the sharper performance drops with increasing N



Some other results (directly from paper)

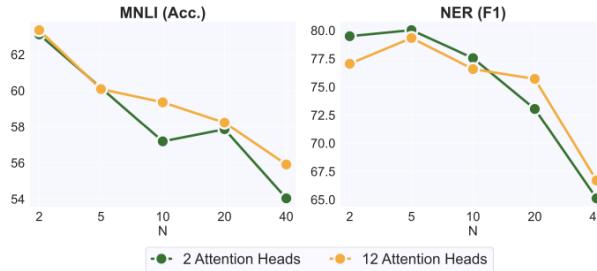
These results are shown for a better understanding of the paper and its concepts.

Due to lack of resources and time I took and inferred it directly

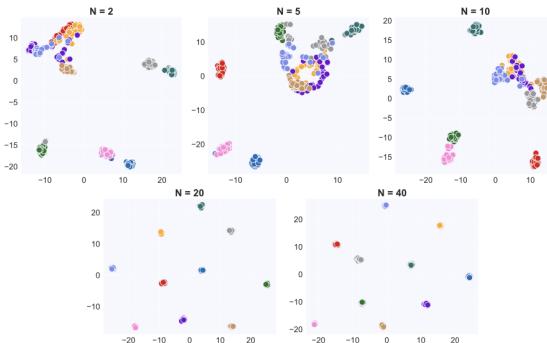


Left graph shows accuracy for the retrieval warm-up task. Surprisingly, models are able to retrieve words with 100% accuracy up to 20 instances across most multiplexing and demultiplexing strategies.

Right graph shows runtime and throughput numbers of T-MUX models on 20K MNLI instances, normalized by a base model's performance without multiplexing. T-MUX (12L/768H) can multiplex up to 40 instances leading to roughly 18x speedup.



The number of attention heads seems invariant to multiplexing: The authors observed this, in order to investigate this further, they trained a variant of T-MUX with two self-attention heads per layer and evaluated its performance on various tasks. Specifically, they used T-MUX with the Hadamard + Index Embedding configuration and found that reducing the number of self-attention heads had minimal impact on retrieval warm-up task performance. In fact, the reduced model achieved a retrieval accuracy of approximately 100% up to $N = 20$. Additionally, the authors observed that on token and sentence-level classification tasks, the performance of the T-MUX model with two self-attention heads was comparable to that of the original T-MUX model with twelve self-attention heads.



The demultiplexed representation of an instance is robust to the other instances it is multiplexed with :

To do this, they randomly selected 10 instances from the MNLI dataset and multiplexed each one with 30 different sets of other instances. This generated 30 different demultiplexed representations for each of the 10 selected samples.

The researchers then reduced the 768 dimensional demultiplexed representation to a 200 dimensional vector with PCA and visualized the resulting points using t-SNE visualization. The clusters for different values of N were compared, and the researchers found that across different values of N , all the points corresponding to the same instance were very close to each other. This suggests that the representation of an instance is not significantly influenced by the set of instances it is multiplexed with, and the demultiplexed representation of an instance is robust to the other instances it is multiplexed with.

New Applications of DataMUX

1) Multi - Label Text Classification

- Multi-label text classification is a type of text classification where a given document can be assigned to one or more categories or labels simultaneously. In other words, instead of assigning a single label to a document, as in single-label classification, multi-label classification assigns multiple labels to it.
- Multi-label text classification is used in various applications, such as document categorization, sentiment analysis, topic classification, and recommendation systems. In such applications, documents can belong to more than one category, and multi-label classification algorithms are used to identify all relevant categories.
- We applied the approach of DataMUX, learned through this paper to achieve this task.

Dataset

- we took the data for this task from kaggle (link mentioned in the references section).
- Content :-
 - The dataset contains 6 different labels(Computer Science, Physics, Mathematics, Statistics, Quantitative Biology, Quantitative Finance) to classify the research papers based on Abstract and Title.
 - The value 1 in label columns represents that label belongs to that paper.
 - Each paper has multiple labels as 1.

Approach

- We used the pretrained model Roberta as given by the authors for achieving this task.
- I am using a tokenizer to convert the titles and abstracts of my training data into tokenized sequences that can be used as input for my multi-label text classification model. I am using the Hugging Face library's `tokenizer` function, which takes in a list of strings as input and returns a dictionary with the tokenized sequences and other information.
- For the titles, I am using the `tokenizer` function on my training data's 'TITLE' column, which contains the titles of my documents.
- Similarly, for the abstracts, I am using the `tokenizer` function on my training data's 'ABSTRACT' column, which contains the abstracts of my documents.
- Then, I created a custom dataset class called ClassificationDataset, which is basically for creating my training and testing data as per the model, using these and Dataloaders of pytorch, I created my train and test data loaders.
- Following this, I created a custom Classifier based on the DataMUX approach
- The `Classifier` takes a parameter `K`, which specifies the number of inputs that the model will take.
- The `MultiplexingLayer` layer transforms the outputs of the `LSTM` layers into a single tensor. This tensor is then passed through a fully connected `base_network` layer to produce the final output. The `DemultiplexingLayer` layer splits the output back into K parts, one for each input.
- Then, I trained and tested my model for 10 epochs, with BCE as the loss function and Adam as the optimizer with lr = 0.001

Results

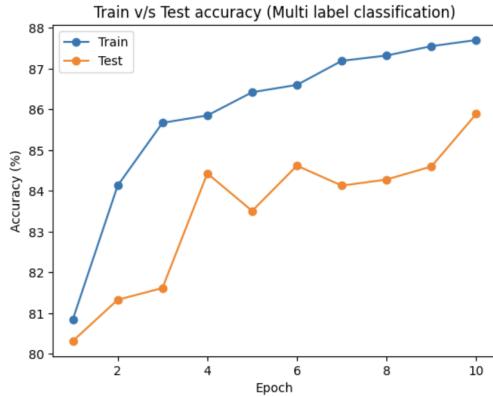
For each of the labels in the dataset loss and accuracy was calculated for both training and testing loops. 1 sample result from epoch 10 is shown below along with plots of the combined accuracy for each train and test epoch.

Testing accuracy achieved was about 85%, this was after taking average of all individual labels, so overall it did a good job.

```
Epoch Train 10: Loss CS: 0.4213 || Loss Phy: 0.3208 || Loss Math: 0.3654 || Loss Stat: 0.4281 || Loss QB: 0.1009 ||
Loss QF: 0.0523 || Loss Total: 1.6888 || Acc CS: 0.8030 || Acc Phy: 0.8737 || Acc Math: 0.8451 || Acc Stat: 0.7805 ||
```

```
Acc QB: 0.9715 || Acc QF: 0.9884 || Acc Total: 0.8770
```

```
Epoch Test 10: Loss CS: 0.4828 || Loss Phy: 0.4469 || Loss Math: 0.4488 || Loss Stat: 0.5265 || Loss QB: 0.0976 ||
Loss QF: 0.0617 || Loss Total: 2.0643 || Acc CS: 0.7675 || Acc Phy: 0.8014 || Acc Math: 0.7959 || Acc Stat: 0.7649 ||
Acc QB: 0.9762 || Acc QF: 0.9873 || Acc Total: 0.8489
```



2) Multiple input data (combining Image + Tabular data)

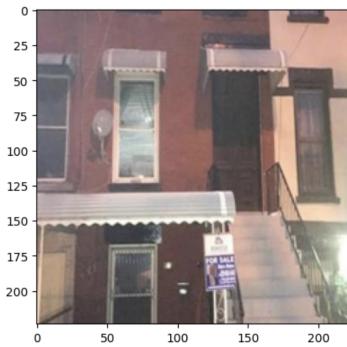
- In this task I thought of predicting the **real estate prices** of houses through matching **image data** and **tabular information**
- compared this with just image only data, to get to know the importance of this combined approach

Dataset and Model

- For the tabular data, I had columns like

```
['zpid', 'price', 'latitude', 'longitude', 'beds', 'baths', 'area']
```

- In this zpid linked us to the corresponding image of the houses



- Next, we concatenated the images with the tabular data and created a new data frame
- So, we got something like this

	zpid	price	latitude	longitude	beds	baths	area
	29777854	430000.0	40.826804	-73.917024	3.0	2.0	1728.0
	30742835	880000.0	40.603546	-73.938332	3.0	3.0	1264.0
	30742959	1160000.0	40.599407	-73.959058	3.0	2.0	1564.0

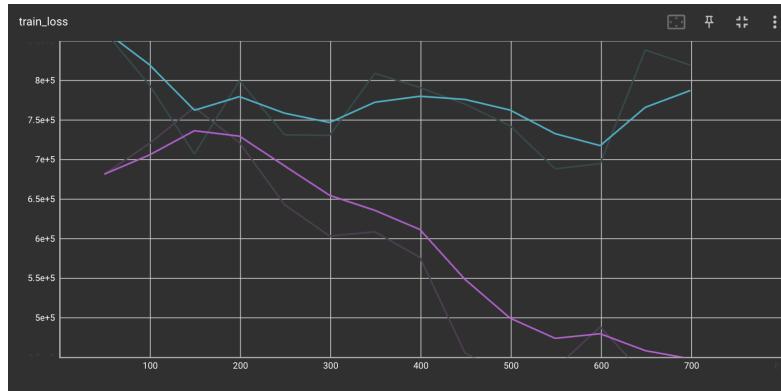
- Following this we created our custom class called `ImageDataset` for giving us the data in the required format
 - In the `__init__()` method, we initialize the required variables and load the tabular data using pandas. The `__len__()` method returns the total size of the dataset, which is equal to the number of rows in the tabular dataframe. In `__getitem__()`, we select a specific row using the index `idx`. We use the index locator of Pandas to achieve this. We also extract the target feature `y` and load the corresponding image using `zpid`. The images in our dataset have four dimensions - RGBA, where the last dimension represents the alpha channel of the image. However, our neural network architecture expects RGB images. So, we remove the last dimension of each image to convert it to RGB format. Finally, we transform both the image and tabular data to torch tensors.
- Next we defined the architecture of our model, to take in image and tabular data, process and combine them together and return a single prediction value. This is what it looks like finally

Name	Type	Params
<hr/>		
0 conv1	Sequential	480
1 conv2	Sequential	4.7 K
2 conv3	Sequential	18.6 K
3 ln1	Linear	692 K
4 relu	ReLU	0
5 batchnorm	BatchNorm1d	32
6 dropout	Dropout2d	0
7 ln2	Linear	85
8 ln4	Linear	60
9 ln5	Linear	110
10 ln6	Linear	55
11 ln7	Linear	11
<hr/>		
716 K	Trainable params	
0	Non-trainable params	
716 K	Total params	
2.866	Total estimated model params size (MB)	

Results

- We trained the model for 10 epochs, with learning rate of 3e-6 and compared the results of `image+tabular data` with `image-only data`
- The error in prediction using `method-1(image+tabular data)` was 4,64,000, but, when only image data was taken then the error was 7,59,000 i.e using `method-2 (image only data)`
- although, they are quite large, but it's due to the fact we had limited data, but still our purpose is answered that when image+tabular data was considered the error in prediction was reduced significantly.

- Shown below is the graph of train loss in tensor board platform, here purple line is for **method-1** and blue line for **method-2**



Other possible applications

1. **Object detection:** You can use this approach for object detection tasks where you need to detect multiple objects in an image. You can use it to process multiple regions of interest (ROIs) simultaneously and generate predictions for each ROI.
2. **Time series forecasting:** You can use this approach for time series forecasting tasks where you need to predict multiple values at once. For example, you can use it for stock price prediction, where you need to predict the prices of multiple stocks at the same time.
3. **Recommender systems:** You can use this approach for recommender systems where you need to predict multiple ratings or preferences at once. For example, you can use it for movie recommendation, where you need to predict how much a user will like multiple movies at the same time.
4. **Language modeling:** You can use this approach for language modeling tasks where you need to predict multiple words or tokens at once. For example, you can use it for machine translation, where you need to predict multiple words in the target language given multiple words in the source language.

Using ChatGPT to solve Application 1

I started off by giving chatGPT abstract of the paper and asked it to remember and understand it, and then asked it to write the code or give logic for my application but ChatGPT wasn't able to solve this, either the code it gave too many errors, or it ran out of context and gave only partial incomplete code. I tried to help it by also giving code used by the authors to see whether it could tweak it as per my requirement but it wasn't able to. But, when I asked for the logic behind the same it gave a rather good response

The screenshot shows a conversation between a user and a large language model (LLM) about implementing a multi-label classification task using DataMUX. The user asks for logic behind the approach, and the LLM provides a sample code snippet. The code includes importing pandas, torch, and transformers, loading a CSV file, defining a DataMUX layer, and creating a PyTorch model class. It also includes a forward pass method, a dataset class, and a training loop.

```

Now using this approach of DatMUX I want to do a multi label classification task, my dataset is in form of a csv file and it contains the following columns [ID, TITLE, ABSTRACT, Computer Science, Physics, Mathematics, Statistics, Quantitative Biology, Quantitative Finance]. can you give a logic behind the same

Sure! Here's some sample code that shows how you can use DataMUX for multi-label classification on a CSV file containing the columns you described:

python
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer, AutoModel
import torch.nn as nn

# Load the CSV file into a pandas DataFrame
df = pd.read_csv("my_dataset.csv")

# Define the tokenizer and model
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")

# Define the DataMUX layer
class DataMUXLayer(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.multiplex = nn.Linear(input_size, output_size)
        self.demultiplex = nn.Linear(output_size, input_size)

    def forward(self, inputs):
        mixed = self.multiplex(inputs)
        mixed_encoded = model(inputs["input_ids"], attention_mask=inputs["attention_mask"])
        mixed_encoded = mixed_encoded.unsqueeze(1) # Add a new dimension
        mixed_encoded = mixed_encoded.repeat(1, mixed.size(1), 1)
        mixed = mixed.unsqueeze(2) # Add a new dimension for the hid
        mixed = mixed.repeat(1, 1, mixed_encoded.size(2)) # Repeat
        mixed = mixed * mixed_encoded # Element-wise multiplication
        mixed = torch.sum(mixed, dim=1) # Sum along the hidden dimension
        demixed = self.demultiplex(mixed)
        return demixed

# Define the dataset and dataloader
class MyDataset(Dataset):
    def __init__(self, df):
        self.df = df

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        inputs = tokenizer(row["TITLE"] + " " + row["ABSTRACT"], padding=True)
        labels = torch.tensor([["Computer Science", "Physics", "Mathematics", "Statistics", "Quantitative Biology", "Quantitative Finance"]])
        return {"inputs": inputs, "labels": labels}

batch_size = 8
dataset = MyDataset(df)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Define the model
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.datmux = DataMUXLayer(768, 256)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(256, 6)

    def forward(self, inputs):
        inputs = self.datmux(inputs)
        inputs = self.dropout(inputs)
        logits = self.classifier(inputs)
        return logits

model = MyModel()

# Define the loss function and optimizer
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)

# Train the model
num_epochs = 10
for epoch in range(num_epochs):
    for batch in dataloader:
        inputs = batch["inputs"]
        labels = batch["labels"]
        logits = model(inputs)
        loss = loss_fn(logits, labels.float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} Loss: {loss.item():.4f}")

```

Using ChatGPT to solve Application 2

In this case also it wasn't able to complete the code.

```

I want to apply this approach of DataMUX that I just taught you to another application where I have a housing data which has column names as [zpid, 'price', 'latitude', 'longitude', 'beds', 'baths', 'area'], the zpid is the corresponding image name, I want to combine the image with this tabular data and send it as a multi input to my model and do a prediction on price, give a code for the same along with logic in pytorch

To use DataMUX to combine image data with tabular data for a multi-input model in PyTorch, you can follow these steps:

1. Load the housing data from the CSV file using Pandas and split it into training and validation sets.
2. Load the corresponding images using PyTorch's torchvision.datasets.ImageFolder class, which assumes that the images are organized into subfolders by class label (in this case, the 'zpid').
3. Use PyTorch's torch.utils.data.Dataset and torch.utils.data.DataLoader classes to create separate data loaders for the tabular data and image data, respectively.
4. Define a custom PyTorch nn.Module for the base network, which will take in the mixed representation generated by the DataMUX layer and output a prediction for the target variable (in this case, 'price').
5. Define a custom PyTorch nn.Module for the DataMUX layer, which will take in the tabular data and the image data and combine them into a mixed representation that can be processed by the base network.
6. Define the loss function and optimizer for training the model.
7. Train the model using the data loaders and loss function, and evaluate its performance on the validation set.

Here is a sample code to implement the above steps:

python
Copy code

import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset

# Load the housing data
housing_df = pd.read_csv('housing_data.csv')
train_df = housing_df.sample(frac=0.8, random_state=1)
val_df = housing_df.drop(train_df.index)

# Define the transforms to be applied to the images
image_transform = transforms.Compose([
    transforms.Resize(256), Regenerate response
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Define the Data class
class Data:
    def __init__(self):
        self.tabular_train_dataset = TabularDataset(train_df)
        self.tabular_val_dataset = TabularDataset(val_df)
        self.tabular_train_loader = DataLoader(self.tabular_train_dataset, batch_size=32, shuffle=True)
        self.tabular_val_loader = DataLoader(self.tabular_val_dataset, batch_size=32, shuffle=False)

        self.image_train_dataset = ImageFolder('images')
        self.image_val_dataset = ImageFolder('images')
        self.image_train_loader = DataLoader(self.image_train_dataset, batch_size=32, shuffle=True)
        self.image_val_loader = DataLoader(self.image_val_dataset, batch_size=32, shuffle=False)

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

    def __getitem__(self, index):
        tabular_data = self.tabular_train_dataset[index]
        image_data = self.image_train_dataset[index]
        return tabular_data, image_data

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

# Define the DataMUX layer
class Data:
    def __init__(self):
        self.tabular_train_dataset = TabularDataset(train_df)
        self.tabular_val_dataset = TabularDataset(val_df)
        self.tabular_train_loader = DataLoader(self.tabular_train_dataset, batch_size=32, shuffle=True)
        self.tabular_val_loader = DataLoader(self.tabular_val_dataset, batch_size=32, shuffle=False)

        self.image_train_dataset = ImageFolder('images')
        self.image_val_dataset = ImageFolder('images')
        self.image_train_loader = DataLoader(self.image_train_dataset, batch_size=32, shuffle=True)
        self.image_val_loader = DataLoader(self.image_val_dataset, batch_size=32, shuffle=False)

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

    def __getitem__(self, index):
        tabular_data = self.tabular_train_dataset[index]
        image_data = self.image_train_dataset[index]
        return tabular_data, image_data

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

# Define the Data class
class Data:
    def __init__(self):
        self.tabular_train_dataset = TabularDataset(train_df)
        self.tabular_val_dataset = TabularDataset(val_df)
        self.tabular_train_loader = DataLoader(self.tabular_train_dataset, batch_size=32, shuffle=True)
        self.tabular_val_loader = DataLoader(self.tabular_val_dataset, batch_size=32, shuffle=False)

        self.image_train_dataset = ImageFolder('images')
        self.image_val_dataset = ImageFolder('images')
        self.image_train_loader = DataLoader(self.image_train_dataset, batch_size=32, shuffle=True)
        self.image_val_loader = DataLoader(self.image_val_dataset, batch_size=32, shuffle=False)

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

    def __getitem__(self, index):
        tabular_data = self.tabular_train_dataset[index]
        image_data = self.image_train_dataset[index]
        return tabular_data, image_data

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

```

References

- <https://arxiv.org/pdf/2202.09318v2.pdf>
- <https://github.com/princeton-nlp/DataMUX/tree/main>
- <https://huggingface.co/princeton-nlp>
- <https://arxiv.org/pdf/2301.12900v2.pdf>
- <https://www.kaggle.com/datasets/shivanandmn/multilabel-classification-dataset>
- <https://rosenfelder.ai/multi-input-neural-network-pytorch/>
- <https://onedrive.live.com/?cid=595cf3061fab0fa5&id=595CF3061FAB0FA5!138981&authkey=!APNEbWW4z4-NzIQ>