

Meta-Learning: ARC

Group 14
Tanmay Singh
2021569
CSAI
Class of '25



INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI



My Approach to the ARC Challenge



*I used the approach discussed in one of the initial papers covered in the class, [Model-Agnostic Meta-Learning](#), wherein I used a custom **Encoder-Decoder Model with Attention Layers** to take the input & generate an encoded representation along with the use of **Attention layers**. I later generate back the original output representation & calculated the loss using a custom loss metric.*

The training pipeline included the traditional inner & outer loops of MAML, wherein the meta-update takes place in the outer loop using the gradients computed in the inner loop.

The data was also processed in a manner that mimicked few-shot training & validation of the meta-updated meta-model on the unseen tasks from the validation set.

Notebook: [Link](#)

Data Processing



```
random.shuffle(train_files)
train_files, val_files = train_test_split(train_files, test_size=val_split, random_state=random_seed)

def process_files(file_list, key='train'):
    dataset = []
    for file_path in file_list:
        data = load_json(file_path)
        for item in data[key]:
            dataset.append({
                'input': item['input'],
                'output': item['output']
            })
    return dataset

train_dataset = process_files(train_files, key='train')
val_dataset = process_files(val_files, key='train')
eval_dataset = process_files(evaluation_files, key='test')
```

Splitting into Training & Validation Sets, with an 80:20 ratio

Data Processing (Padding with a non-encountered pad value)



```
def pad_dataset_uniform(dataset, target_size=30, pad_value=10):
    padded_dataset = []
    for task in dataset:
        padded_task = {
            "input": pad_grid_uniform(task["input"], target_size, pad_value),
            "output": pad_grid_uniform(task["output"], target_size, pad_value)
        }
        padded_dataset.append(padded_task)
    return padded_dataset
```

```
padded_train_dataset = pad_dataset_uniform(train_dataset, target_size=30, pad_value=10)
padded_val_dataset = pad_dataset_uniform(val_dataset, target_size=30, pad_value=10)
padded_eval_dataset = pad_dataset_uniform(eval_dataset, target_size=30, pad_value=10)
```

Data Processing (Separate Input & Output Loaders for each set)



```
class FlattenedARCDataset(Dataset):
    def __init__(self, data, is_input=True):
        self.data = data
        self.is_input = is_input

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

    def __getitem__(self, idx):
        if self.is_input:
            grid = self.data[idx]["input"]
        else:
            grid = self.data[idx]["output"]
        return torch.tensor(grid, dtype=torch.float32)

train_inputs = FlattenedARCDataset(flattened_train_dataset, is_input=True)
train_outputs = FlattenedARCDataset(flattened_train_dataset, is_input=False)

val_inputs = FlattenedARCDataset(flattened_val_dataset, is_input=True)
val_outputs = FlattenedARCDataset(flattened_val_dataset, is_input=False)

eval_inputs = FlattenedARCDataset(flattened_eval_dataset, is_input=True)
eval_outputs = FlattenedARCDataset(flattened_eval_dataset, is_input=False)

train_input_loader = DataLoader(train_inputs, batch_size=BATCH_SIZE, shuffle=True)
train_output_loader = DataLoader(train_outputs, batch_size=BATCH_SIZE, shuffle=True)

val_input_loader = DataLoader(val_inputs, batch_size=BATCH_SIZE, shuffle=False)
val_output_loader = DataLoader(val_outputs, batch_size=BATCH_SIZE, shuffle=False)

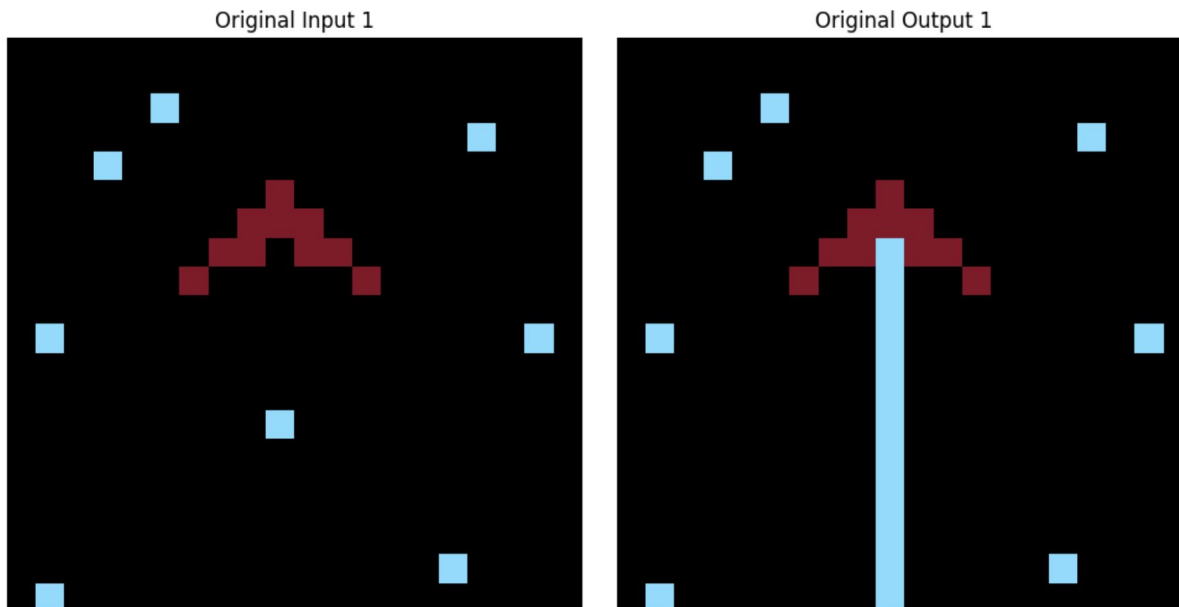
eval_input_loader = DataLoader(eval_inputs, batch_size=BATCH_SIZE, shuffle=False)
eval_output_loader = DataLoader(eval_outputs, batch_size=BATCH_SIZE, shuffle=False)
```

Novelties in Work

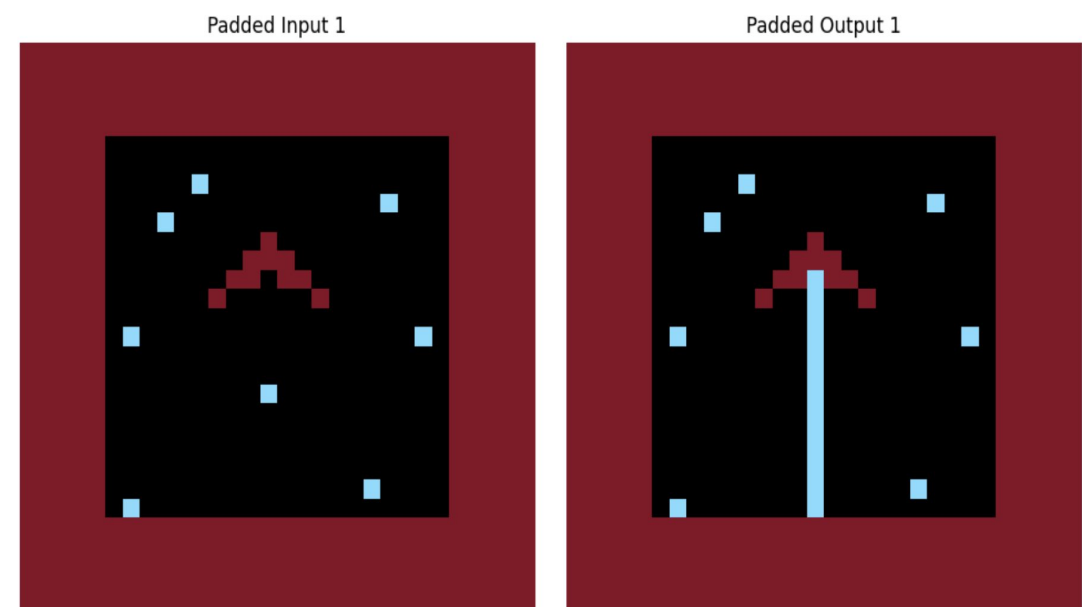


There are no significant novelties in my approach. The only difference was the way I prepared the data for training. I padded the input & output grids using a value not in either of the grids to a size of 30x30 in a uniform manner, which I later flattened into a size of (900, 1) for training & validation by creating separate input & output data loaders & datasets for all three sets (training, validation & evaluation).

ORIGINAL (IP/OP)



PADDED (IP/OP)



Approach Discussed



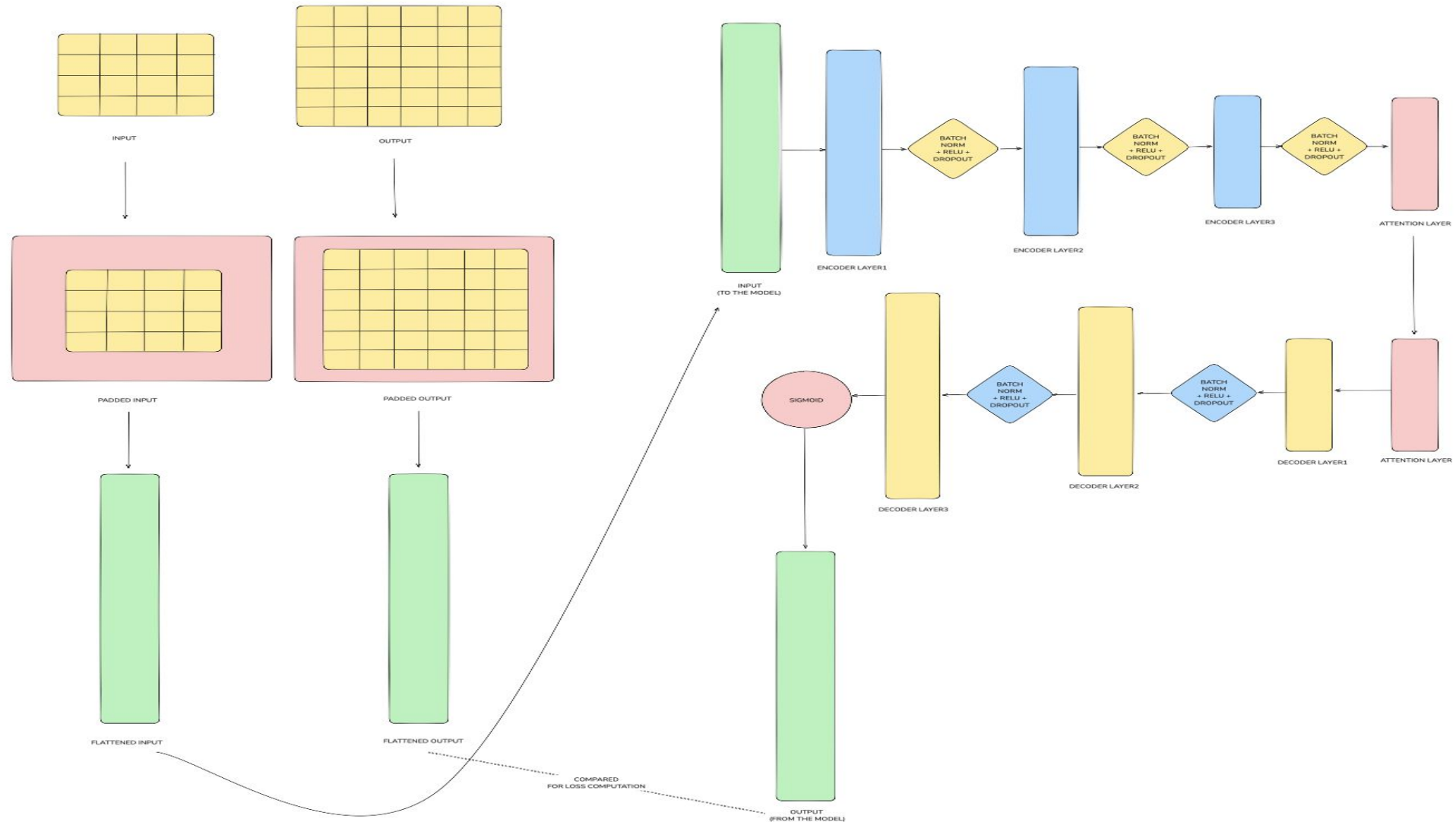
Data Processing:

1. The data from the training folder (within the arc_data folder, there are two folders, training & evaluation) was split in the ratio of 80:20, where the 80% of data was used for few-shot training of the meta-model & validation one was used to test the performance of the meta-updated meta-model.
2. The evaluation data was prepared from the evaluation folder within the arc_data folder following the same processing steps as above.
3. The data was padded with a value not in either grids to max allowed size (30x30) which was then flattened to be used as an input to the custom Encoder-Decoder model having attention layers.

Model & Evaluation:

1. The model used which I used for the ARC challenge was a custom Encoder-Decoder model with attention layers, that took the input (900, 1), and generated an encoded representation of it, with the use of attention layers & finally generated/decoded back the output grid.
2. The Training & Validation was done using a model pipeline that had the tradition outer & inner loops used in MAML, with a custom loss function where each value between the predicted & the actual output is compared (absolute difference), and then averaged out to return the average loss.
3. The outer loop does the meta-update using the gradients computed in the inner-loop during few-shot training.
4. The training loss was logged per epoch, while the validation one was logged per batch, since the meta-model was first trained on the training set & once trained enough, it was used to generate predictions on the unseen task (samples) from the validation set.

Model Architecture (Encoder-Decoder)



Loss Function



A relatively simple one, where the difference between the predicted & actual values was computed & later averaged to give the average loss for that particular sample.

```
def compute_elementwise_loss(self, predicted, actual):  
    differences = torch.abs(predicted - actual)  
    total_loss = differences.sum()  
    avg_loss = total_loss / predicted.numel()  
    return avg_loss
```

Why this Approach? Did it worked?



Although I tried a lot of approaches, but none of them solved any tasks on the *publicly available evaluation set (0/419)*. It did solve 1 task in the Validation set, though.

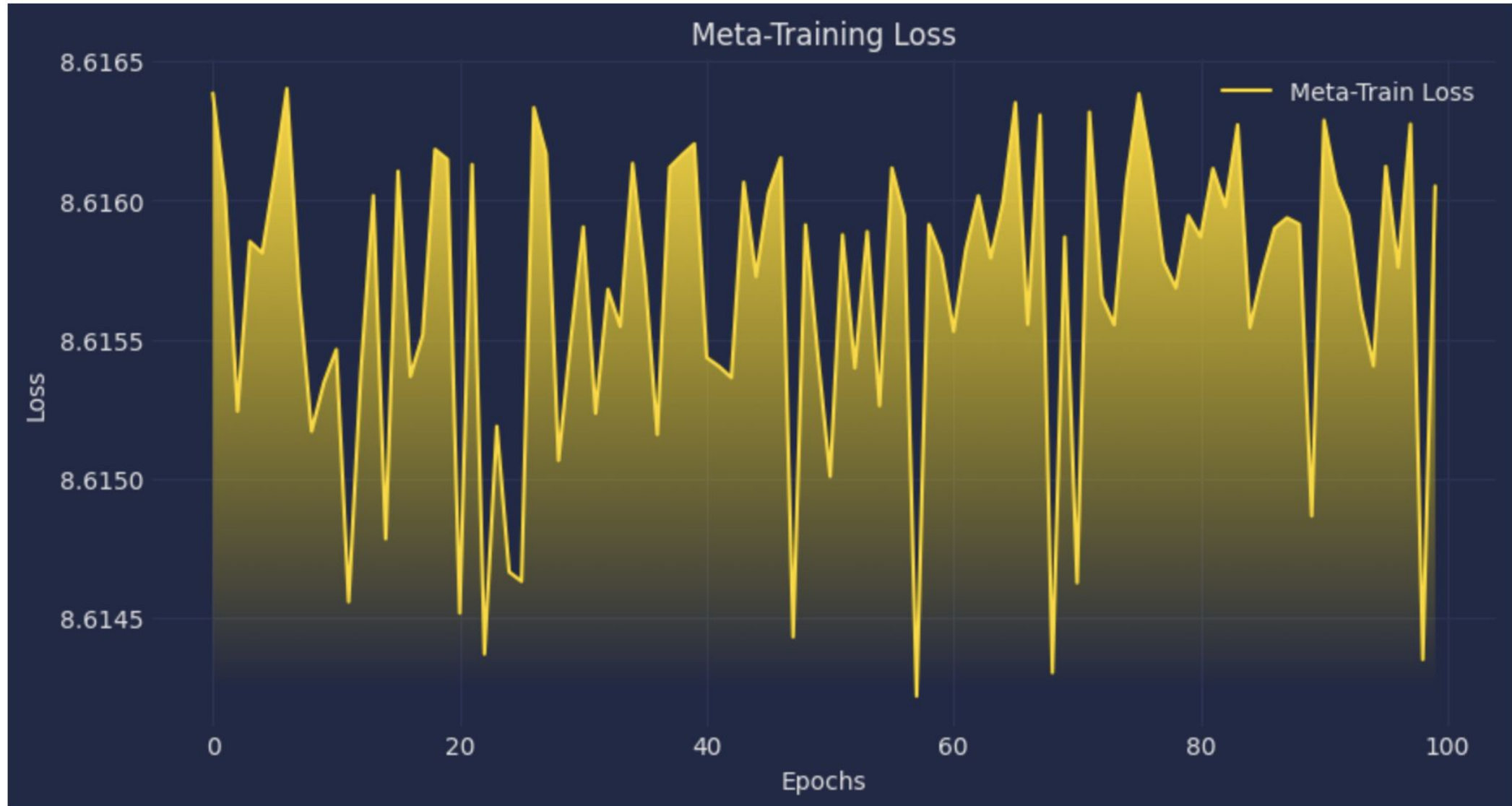
Other approaches that I tried were:

1. Using CodeIt's DSL integrated with Michael Hodel's Starter notebook that generated programs exhaustively & searched for them using search algorithms (both optimized & non-optimized).
2. Tried Fine-tuning a LLM like LLAMA & T5 for seq2seq task.
3. Tried using various attention based & deep neural network models, but none of them translated into good results.

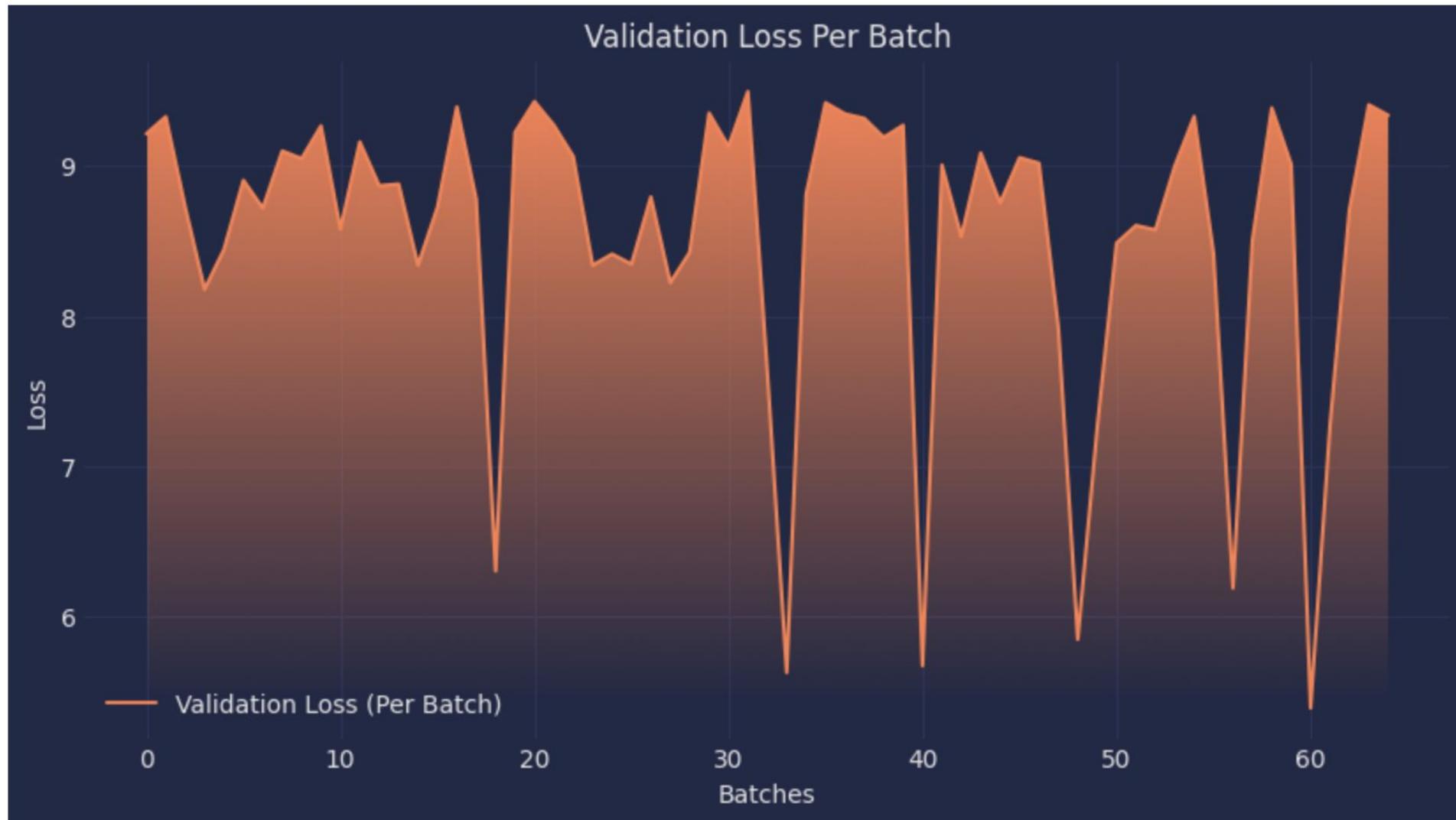
Although none of the approaches solved any tasks on the evaluation set, I used my current approach to address the ARC challenge as it is traditional in nature (in the domain of meta-learning). Thus, I found MAML to be a good base to tackle the ARC challenge even though it did not translated into desired results on the evaluation set. On the contrary, the main idea behind sticking to this approach was that the model was showing signs of training as the training & validation losses seemed to change every time, although not by much.

Results: [LINK](#)

Evaluation



Evaluation



Meta-Training completed. Final Validation Loss: 8.5500


Did it generalise?



Sadly, it did not! Even though it suggested that the model was learning, but it failed to generalise on the evaluation set. It did solved 1 task in the validation set, but its not a concrete measure of the generalisation/performance of the model.

```
strict_match_accuracy = strict_match_evaluation(model, val_input_loader, val_output_loader, DEVICE)
```

✓ 0.1s

Strict Match Evaluation:  65/? [00:00<00:00, 383.77it/s]

Strict Match Accuracy: 0.39% (1/259)

```
strict_match_accuracy = strict_match_evaluation(model, eval_input_loader, eval_output_loader, DEVICE)
```

Strict Match Evaluation: 0it [00:00, ?it/s]

Strict Match Accuracy: 0.00% (0/419)

Learnings from the Project



- It's a tough challenge, and a lot of approaches failed.
- Fine-tuning an LLM or incorporating a DSL with Program Synthesis suggested a good base for generalisation but did not yield expected results, and were time & resource consuming in nature.
- Traditional Meta-Learning models did not give desired results in my case, although they looked promising.
- Lastly, we're still a long way from creating models that generalise well to unseen tasks & it's a food for thought for many of the enthusiastic engineers, the world over!



THANK YOU

