# E0270: MACHINE LEARNING

## Assignment #2

**Submitted By:**
Rajkamal Ingle(SR No. 22892)
CSA, IISc
April 27, 2024

## Contents

**1 Problem 0** **3**

**2 Problem 1** **3**

2.1 Need of Low Rank Adaptation of Large Language Models . . . . . . . . . . . 3

2.2 Methodology of LoRA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.3 Key Implementation details . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.4 Hyperparameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.5 Accuracy and Loss . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**3 Problem 2** **8**

3.1 Knowledge Distillation - Need and Essence . . . . . . . . . . . . . . . . . . . 8

3.2 Key Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.3 Hyperparameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.4 Accuracy and Loss . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**4 Conclusion** **12**

## 1 Problem 0

Entered Different prompts and saw sometimes meaningful responses were generated, whereas sometimes they didn't make any sense at all. Learned key ideas about the GPT-2 from given reference paper.

## 2 Problem 1

### 2.1 Need of Low Rank Adaptation of Large Language Models

In traditional fine-tuning of existing model for different downstream application, involves updating all the parameters of the original model, which would result in a nearly equivalent model to the original one. Depending on the original model, it can be hard to replicate or update. For example, a model like GPT-3 with 175 Billion parameters become a regular standard, then its computational and storage demands would become tougher to mitigate. Hence, their is a need for low cost adaptation of the model which captures the essence of the model.

LoRA exploits the observation that the change in weights tends to have a low intrinsic rank which forms the basis of its approach. LoRA allows us to train some dense layers in a neural network indirectly by optimizing rank decomposition matrices of the dense layers' change during adaptation instead, while keeping the pre-trained weights frozen. LoRA demonstrates that very low ranks suffice for efficient adaptation when dealing with models as large as GPT-3 and also gives following key advantages:

- Such a pre-trained model obtained from LoRA can be shared and used to build many small LoRA modules for different tasks. We can freeze the shared model and efficiently switch tasks by replacing the decomposition matrices, reducing the storage requirement and task-switching overhead significantly.
- LoRA makes training more efficient and lowers the hardware barrier to entry by up to 3 times when using adaptive optimizers since we do not need to calculate the gradients or maintain the optimizer states for most parameters. Instead, we only optimize the injected, much smaller low-rank matrices.
- The Simple Linear design allows to merge the trainable matrices with the frozen weights when deployed, introducing no inference latency compared to a fully fine-tuned model, by construction.

### 2.2 Methodology of LoRA

1. **Initial Setup and Reparameterization:**
    - Initialize a pre-trained weight matrix $W_0$ of dimension $d \times k$, where $d$ is the input dimension and $k$ is the output dimension.
    - Decompose the update to the weight matrix $\Delta W$ as $W_0 + \Delta W = W_0 + BA$, where $B$ is of dimension $d \times r$, $A$ is of dimension $r \times k$, and $r$ is the rank of the decomposition.

- The rank $r$ is chosen such that $r \leq \min(d, k)$.

2. **Training Procedure:**
   - Freeze $W_0$ during training; it does not receive gradient updates.
   - Trainable parameters are $A$ and $B$, which are updated during backpropagation.

3. **Forward Pass:** Compute the modified forward pass using the reparameterized weights:

$$h = W_0x + \Delta Wx = W_0x + BAx$$

4. **Initialization:**
   - Initialize $A$ with random Gaussian initialization and $B$ with zeros.
   - At the beginning of training, $\Delta W = BA = 0$.
   - Scale $\Delta Wx$ by $\alpha\sqrt{r}$, where $\alpha$ is a constant and r is the rank. Tuning $\alpha$ is comparable to tuning the learning rate in optimization algorithms like Adam. Typically, $\alpha$ is set to a fixed value without tuning after trying a few initial values.

5. **Optimizaiton:**
   - Use Adam optimizer for training.
   - Tune $\alpha$ which scales $\Delta Wx$, which is analogous to tuning the learning rate.
   - Set $\alpha$ to the first $r$ tried and do not further tune it.

6. **Generalization of Full Fine-tuning:**
   - LoRA allows training of a subset of pre-trained parameters without requiring full-rank updates during adaptation.
   - By setting the LoRA rank $r$ to the rank of pre-trained weight matrices, LoRA approximates the expressiveness of full fine-tuning.

7. **Inference and Task Switching:**
   - Compute and store $W = W_0 + BA$ for inference.
   - Recover $W_0$ by subtracting $BA$ and adding a different $B_0A_0$ when switching to another downstream task.
   - Guarantees no additional latency during inference compared to a fine-tuned model.

8. **Application to Transformer:**
   - LoRA can be applied to subsets of weight matrices in a Transformer to reduce trainable parameters.
   - In Transformer, adapt attention weights for downstream tasks while freezing MLP modules.

## 2.3 Key Implementation details

- **Linear LoRA Class implementation:** We implemented a PyTorch module called LoRALinear, which introduces a novel linear layer utilizing the LoRA (Low-Rank Adaptation) technique.
  - Initialization: The LoRALinear module is initialized with parameters for the left and right low-rank matrices, along with an option to specify the LoRA rank.
  - Forward Pass: During the forward pass, the input tensor is multiplied by the left low-rank matrix, followed by the right low-rank matrix, using matrix multiplication operations.
  - Parameter Initialization: The parameters (L_matrix and R_matrix) are initialized using Xavier uniform initialization, ensuring stable training. However, bias initialization is commented out in this implementation.
  - Dropout: Additionally, a dropout layer with a dropout rate of 0.1 is applied to the output tensor for regularization, though it seems there's a typo (should be 0.1 instead of 0,1).
- **Injection of Auxillary LoRA Layers:**
  - Auxiliary LoRA layers are additional low-rank adaptation layers integrated into the existing architecture of a neural network, specifically designed to enhance adaptability and efficiency and they apply low-rank adaptation to specific parts of the model, such as attention and projection operations.
  - By injecting auxiliary LoRA layers, the model gains the ability to fine-tune its behavior more precisely while still benefiting from the computational efficiency of low-rank adaptation.
  - The injection of auxiliary LoRA layers involves adding LoRALinear layers corresponding to the attention and projection embedding layers of the Transformer model.
  - The injection of auxiliary LoRA layers into embedding layers enhances the model's adaptability and efficiency. It allows for more fine-grained adjustment of attention and projection operations during model adaptation to different tasks.
  - Overall, the injection of auxiliary LoRA layers into embedding layers improves the model's ability to handle diverse tasks effectively while maintaining computational benefits.
- **Transformer MLP:**
  - We extend the Transformer architecture with a multi-layer perceptron (MLP) component. It consists of linear layers followed by a GELU activation function and dropout, adhering to the standard Transformer architecture.
  - The MLP's feedforward dimension is set to four times the model dimension, aligning with Transformer conventions.
  - LoRA integration is introduced through auxiliary LoRA layers , enhancing adaptability and efficiency by applying low-rank adaptation.

- Gradients are removed from original linear layers to facilitate training through auxiliary LoRA layers.
- The forward pass is modified to incorporate outputs from auxiliary LoRA layers, ensuring seamless integration into the computation graph for automatic back-propagation.
- Optimization is achieved using the Adam optimizer, a widely used optimization algorithm known for its efficiency and effectiveness in training neural networks.

- **Train and Evaluate functions:** We used a regular implementation by iterating over the train loader and validation loader and used CrossEntropy as loss function and AdamW optimizer.

### 2.4 Hyperparameters

Utilised the gpt2-medium variant of gpt.
Number of Epochs = 10
Default values were used for rest all of the parameters.

### 2.5 Accuracy and Loss

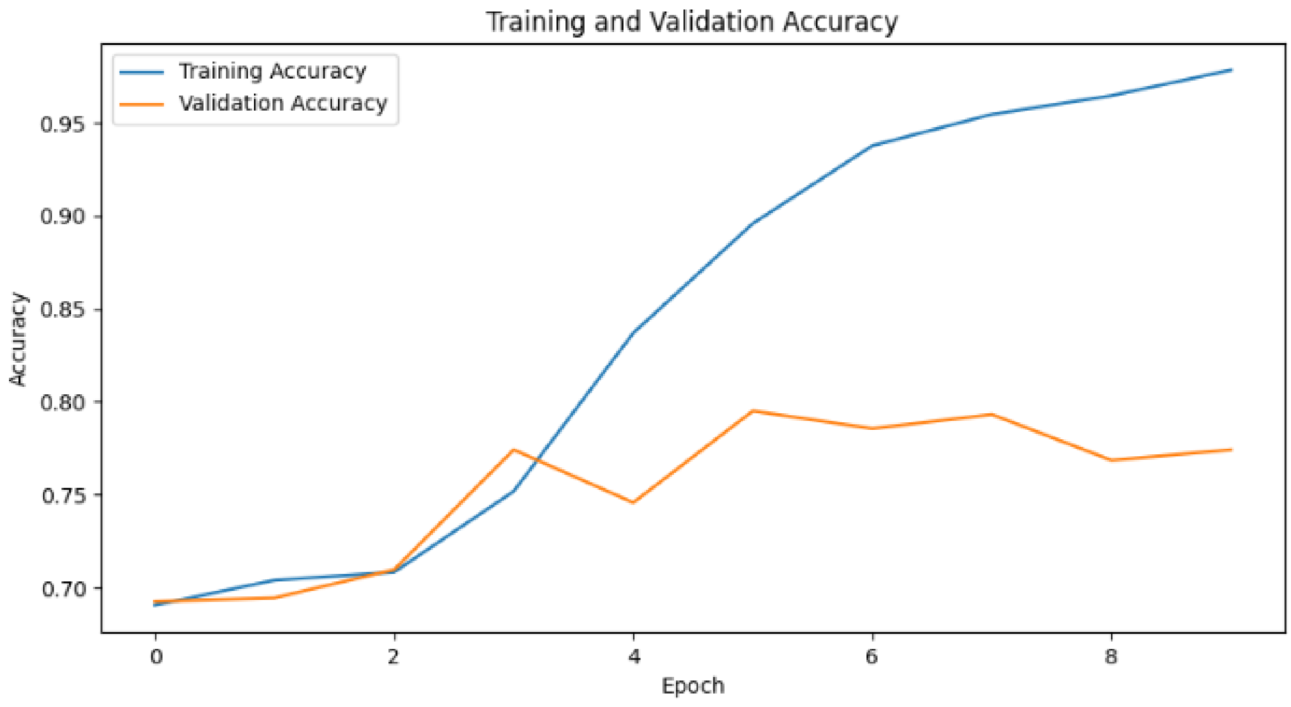

Figure 1: Accuracy for LoRA based training

We can see in the figures 1 and 2 that the training losses as well as accuracies converge till 3-4 epochs(after which there is overfitting) approximately to 0.58 and 80 % respectively also can be seen in table 1.

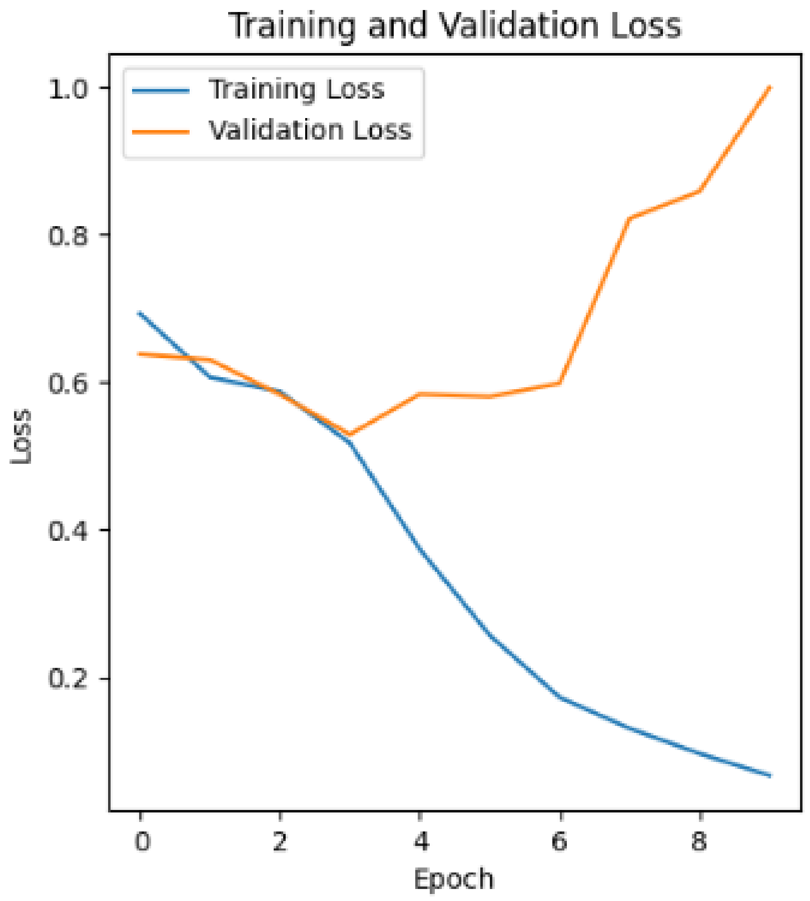

Figure 2: Loss for LoRA based training

Table 1: Loss and Accuracy for LoRA

| | **Train** | **Validation** |
|---|---|---|
| **Loss** | 0.257 | 0.580 |
| **Accuracy** | 0.896 | 0.795 |

## 3 Problem 2

### 3.1 Knowledge Distillation - Need and Essence

In large-scale machine learning, there's a discrepancy between training and deployment requirements. While training involves processing large, redundant datasets with high computational resources, deployment necessitates low latency and minimal computational resources. To bridge this gap, the concept of Knowledge Distillation is introduced. Knowledge distillation is proposed as a method to transfer knowledge from cumbersome to small models [Caruna reference] which involves training a large, cumbersome model, such as an ensemble or a model with strong regularization, followed by distillation of its knowledge to a smaller model suitable for deployment. Essence of the knowledge Distillation can be understood by following points:

- The key insight lies in viewing knowledge not as specific parameter values but as a learned mapping from inputs to outputs. The cumbersome model, trained to discriminate between numerous classes, assigns probabilities to incorrect answers, which reflect its generalization tendencies. Knowledge distillation leverages this by using the cumbersome model's class probabilities, termed "soft targets," to train the small model.
- A significant challenge arises from the low probabilities associated with incorrect answers, which have minimal impact on traditional cost functions. To address this, the given reference paper for Knowledge Distiallation in Assignment - proposes raising the temperature of the final softmax to produce softer targets, enabling effective knowledge transfer. Additionally, it suggests incorporating a small term in the objective function to encourage the small model to predict true targets while matching the soft targets.
- The transfer set for training the small model can include unlabeled data or the original training set, with the addition of an objective term to encourage prediction of true targets. This comprehensive approach enhances the small model's performance by effectively transferring the knowledge from the cumbersome model.

### 3.2 Key Implementation Details

- **DistilRNN:** We were directed to implement a RNN model for this which is inbuilt in PyTorch. The key components of that RNN are as follows:
  - The model starts with an embedding layer, which converts discrete tokens into dense vectors, enabling the network to learn meaningful representations of input sequences. Here, as the vocabulary size is 50257 and embedding dimension is 768, we create a corresponding embedding layer for it.
  - These embeddings are then passed through an RNN layer, which processes the sequences sequentially, generating hidden states that encode the temporal information of the input.The RNN's ability to maintain a hidden state allows it to capture long-range dependencies and temporal dynamics within the sequences.

  - Finally, the model produces predictions by applying a fully connected layer to the hidden state corresponding to the last time step, summarizing the sequence information into a format suitable for classification.

- **Teacher and Student Model:** We also initialized the teacher model to the GPT class in model.py and the student model to the DistilRNN class defined earlier, next we will try to train this student model to capture the behaviour of its teacher model.

- **Training of the student model** We used this PyTorch article for the standard training procedure of Knowledge Distillation. The key idea implemented in it are as follows:

  - Knowledge Transfer: The teacher model provides soft targets (probabilities) derived from its predictions on the training data. The student model learns from both the ground truth labels and the soft targets provided by the teacher, facilitating knowledge transfer from the teacher to the student.
  - Loss Function Combination: The training loss comprises two components: soft target loss and cross-entropy loss. Soft target loss measures the discrepancy between the student's predicted probabilities and the soft targets provided by the teacher. Cross-entropy loss measures the discrepancy between the student's predictions and the ground truth labels.
  - Optimization: The optimizer updates the parameters of the student model based on the combined loss. This process iteratively adjusts the student model's parameters to minimize the discrepancy between its predictions and both the soft targets and ground truth labels.
  - Evaluation Metrics: During training, the function tracks the running loss and calculates the number of correct predictions. These metrics are used to compute the average loss and accuracy over the entire training dataset, providing insights into the model's performance.

### 3.3 Hyperparameters

We used the gpt variant = gpt2-medium for this.
Number of Epochs = 10
Rest all of the parameters were kept to default.

### 3.4 Accuracy and Loss

Table 2: Loss and Accuracy for RNN

| | **Train** | **Validation** |
|---|---|---|
| **Loss** | 0.615 | 0.625 |
| **Accuracy** | 0.70 | 0.6982 |

Here, we can see in figure 4 after several epochs the validation losses and accuracy become stable and the validation accuracy is turning out to be approximately 70 %.

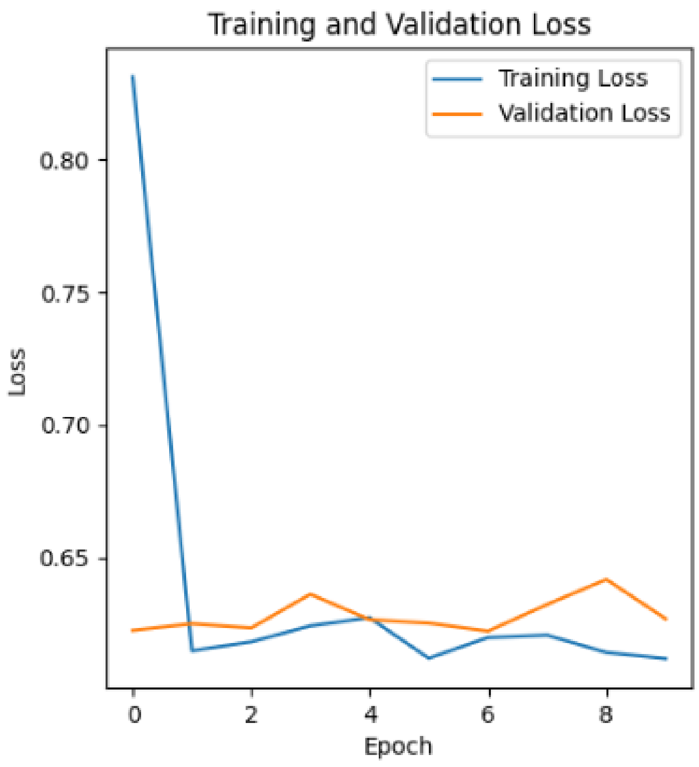

Figure 3: Loss for RNN based training

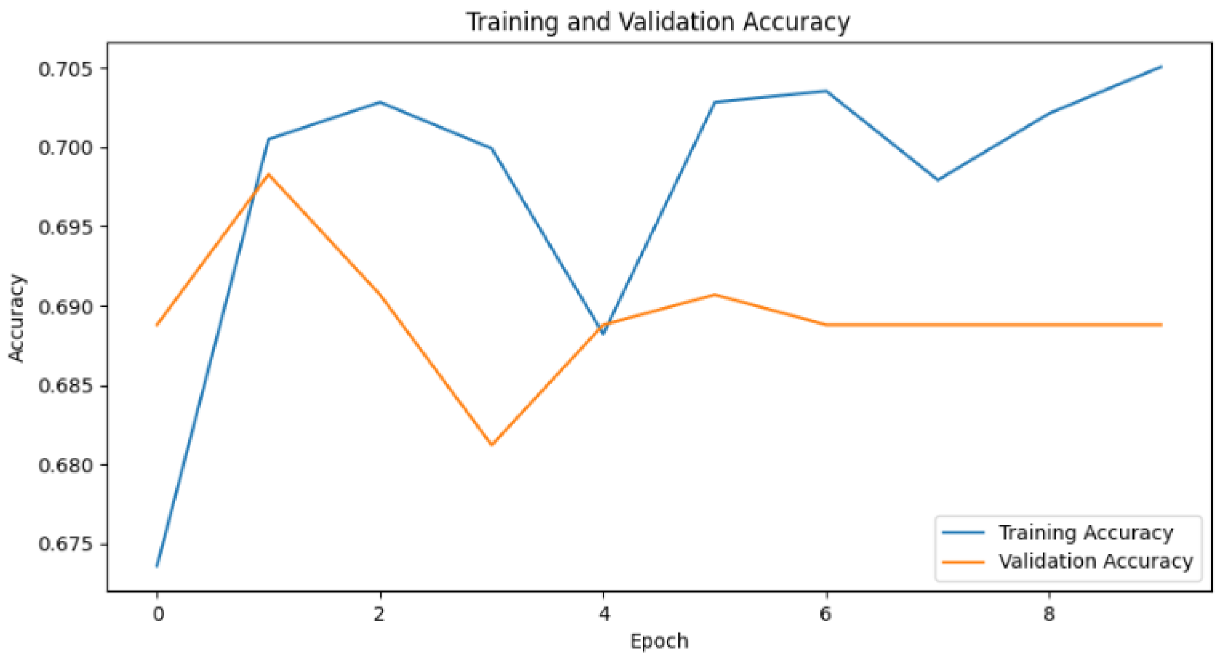

Figure 4: Accuracy for RNN based training

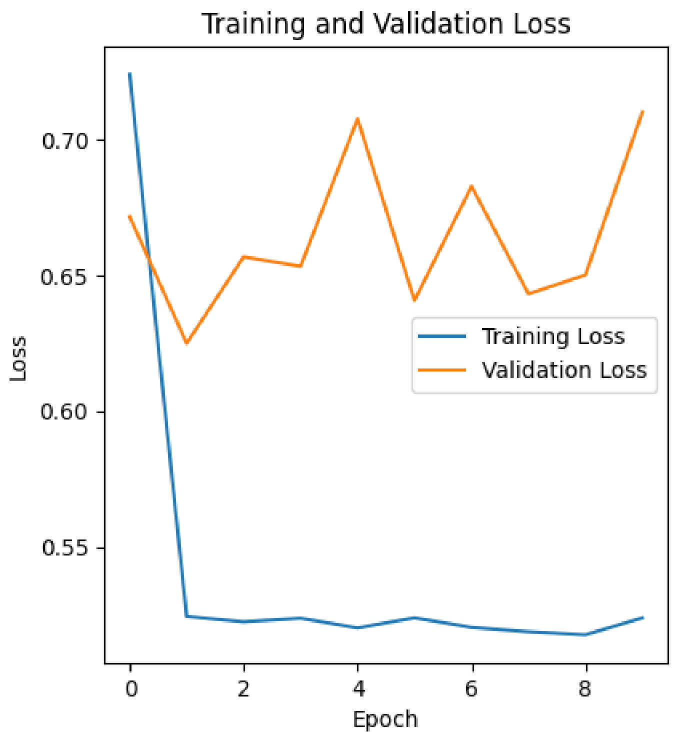

Figure 5: Loss for Distilled Model

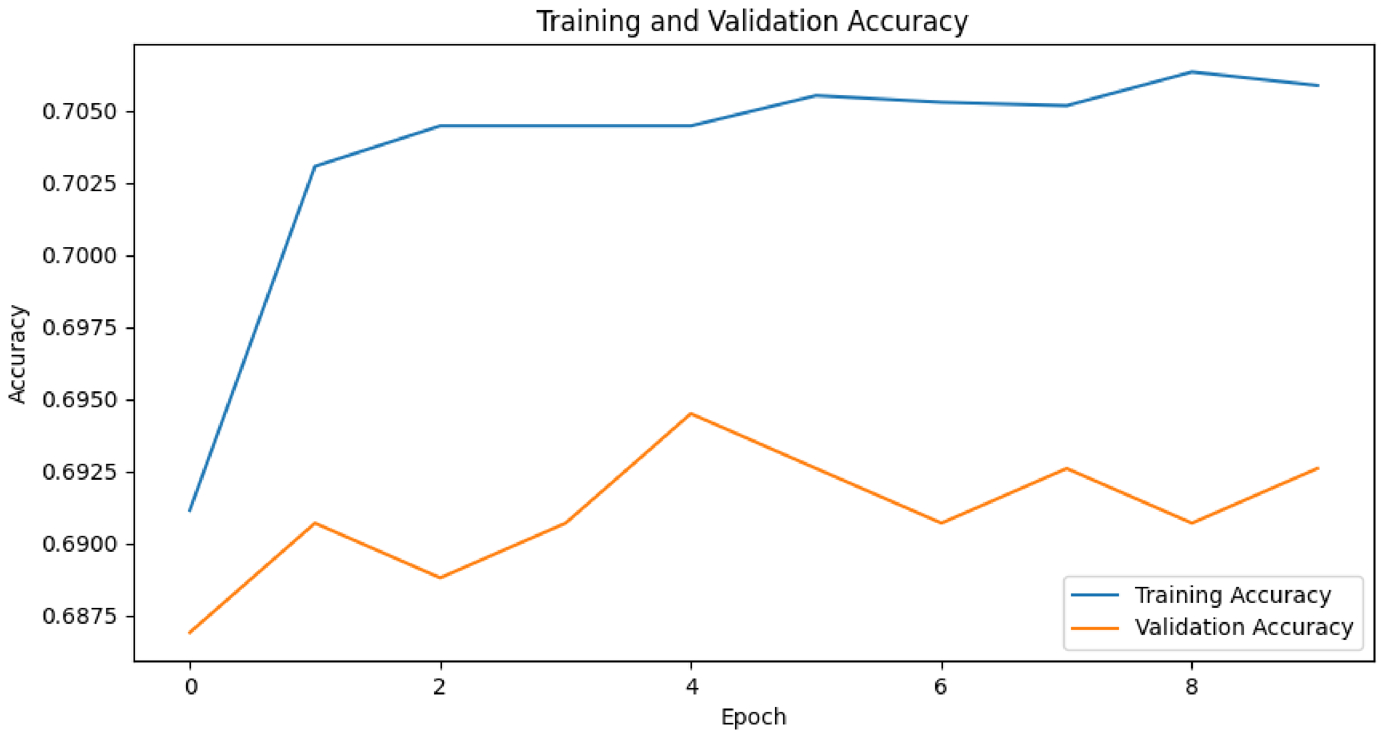

Figure 6: Accuracy for Distilled Model

Table 3: Loss and Accuracy for Distilled Model

| | **Train** | **Validation** |
|---|---|---|
| **Loss** | 0.724 | 0.672 |
| **Accuracy** | 0.70 | 0.6869 |

Here, in figure 6the convergence happens within 1-2 epochs(after which there's just overfitting) and the validation accuracy is turning out to be approximately 68.7 %.

## 4 Conclusion

Final approximate Validation accuracies for LoRA based Training, purely RNN model and the Distilled model using Knowledge Distillation where Teacher model is GPT model and the Student model is DistillRNN - already used in purely RNN are 80 %, 70% and 68.7% respectively.