E0270: Machine Learning
Assignment-2

SR No.: 06-18-01-10-22-23-1-23198
Name: Lochan Surya Teja Neeli

Problem 1

LoRA Implementation for GPT-2 Architecture

1. **Introduction:** The LoRA (Low-Rank Approximation) implementation for GPT-2 architecture offers a method to approximate large weight matrices with lower-rank matrices, reducing computational complexity while retaining crucial information.

   LoRALinear Class: The LoRALinear class extends the `nn.Module` in PyTorch for GPT-2 architecture. It initializes with parameters for input and output features dimensions and rank of low-rank approximation. The U and V matrices, representing the low-rank decomposition, are initialized using Kaiming uniform method. The `forward` method performs low-rank approximation by multiplying input tensor with U and V matrices.

   Usage: Applying LoRA to complex and large neural networks offers a method to reduce the number of trainable parameters, particularly focusing on attention weights in the Transformer architecture. By treating attention weight matrices as single matrices and freezing certain modules like MLP layers, parameter efficiency is achieved without sacrificing performance in downstream tasks.

   Conclusion: Practical benefits of LoRA include significant reductions in memory and storage usage, particularly noticeable when the rank of the low-rank approximation is smaller than the model dimension. For instance, in a large Transformer model like GPT-3 175B, VRAM consumption during training can be substantially reduced, allowing for training with fewer GPUs and alleviating I/O bottlenecks. Moreover, LoRA enables task switching during deployment at a lower cost by swapping only the LoRA weights, facilitating the creation of customized models on-the-fly. Additionally, training speedups have been observed, as gradient calculations are not required for the majority of parameters. It is suitable for tasks where efficiency is critical.

2. All the `TODO` sections in the code are filled in. The causal attention linear layers and the transformer's weights and biases are frozen by setting the attribute `requires_grad =False`

3.
   - Optimizer: Adam
   - Loss Function: Cross Entropy
   - The training strategy is implied in the algorithm provided in the end of this document

   Results

Parameter	Value
Number of parameters	125.03M
Number of trainable parameters	0.63M
Reduction	99.50%

- Maximum accuracy on the COLA validation set: 80.05% (for 4 epochs and rank=4), and 81.02% for(10 epochs and rank=8)for the GPT2 model.

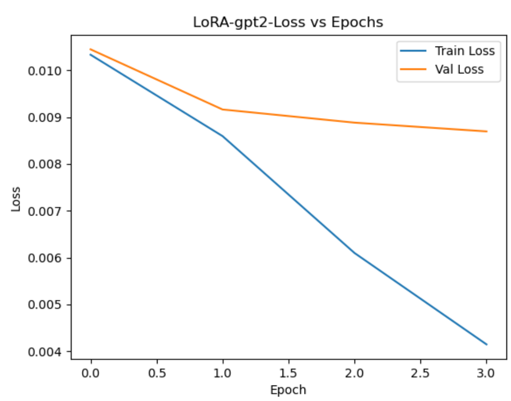
Parameter	Value
Mode	LoRA
Sr. No.	23198
GPU ID	0
GPT Variant	GPT-2
Max New Tokens	100
Model Path	models/LoRA.pth
Learning Rate (LR)	0.001
Batch Size	64
Epochs	3
LoRA Rank	4
Device	CUDA

(a) LoRA Model Configuration for GPT2

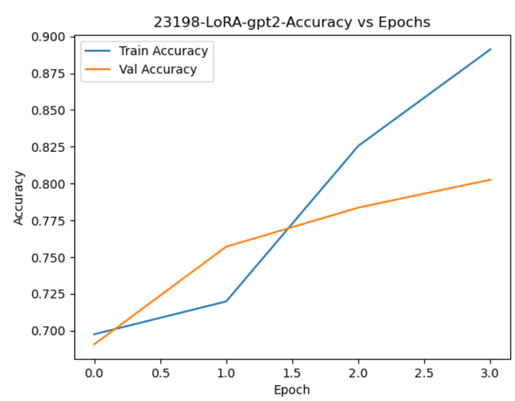
Parameter	Value
Mode	LoRA
Sr. No.	23198
GPU ID	0
GPT Variant	GPT2-Medium
Max New Tokens	100
Model Path	models/LoRA.pth
Learning Rate (LR)	0.001
Batch Size	64
Epochs	3
LoRA Rank	4
Device	CUDA

(b) LoRA Model Configuration for GPT2-medium

Table 1: LoRA Model Configuration



(a) Loss



(b) Metrics

Figure 1: LoRA GPT2 Loss and Metrics

Problem 2

Knowledge Distillation with GPT-2 Architecture

" The cumbersome model could be an ensemble of separately trained models or a single very large model trained with a very strong regularizer such as dropout [9]. Once the cumbersome model has been trained, we can then use a different kind of training, which we call "distillation" to transfer the knowledge from the cumbersome model to a small model that is more suitable for deployment.

we tend to identify the knowledge in a trained model with the learned parameter values and this makes it hard to see how we can change the form of the model but keep the same knowledge.

Our more general solution, called "distillation", is to raise the temperature of the final softmax until the cumbersome model produces a suitably soft set of targets. We then use the same high temperature when training the small model to match these soft targets. We show later that matching the logits of the cumbersome model is actually a special case of distillation.

In the simplest form of distillation, knowledge is transferred to the distilled model by training it on a transfer set and using a soft target distribution for each case in the transfer set that is produced by using the cumbersome model with a high **temperature** in its softmax. The same high temperature is used when training the distilled model, but after it has been trained it uses a temperature of 1. When the correct labels are known for all or some of the transfer set, this method can be significantly improved by also training the distilled model to produce the correct labels. One way to do this is to use the correct labels to modify the soft targets, but we found that a better way is to simply use a weighted average of two different objective functions. The **first objective function** is the cross entropy with the soft targets and this cross entropy is computed using the same high temperature in the softmax of the distilled model as was used for generating the soft targets from the cumbersome model. The **second objective function** is the cross entropy with the correct labels. This is computed using exactly the same logits in softmax of the distilled model but at a temperature of 1. We found that the **best results were generally obtained by using a considerably lower weight on the second objective function**. Since the magnitudes of the gradients produced by the soft targets scale as $\frac{1}{T^2}$ it is important to multiply them by T^2 when using both hard and soft targets. This ensures that the relative contributions of the hard and soft targets remain roughly unchanged if the temperature used for distillation is changed while experimenting with meta-parameters.

So in the high temperature limit, distillation is equivalent to minimizing $\frac{(z_i - v_i)^2}{2}$, provided the **logits are zero-meaned separately for each transfer case**. At lower temperatures, distillation pays much less attention to matching logits that are much more negative than the average. This is potentially advantageous because these logits are almost completely unconstrained by the cost function used for training the cumbersome model so they could be very noisy. On the other hand, the very negative logits may convey useful information about the knowledge acquired by the cumbersome model. Which of these effects dominates is an empirical question. We show that when the distilled model is much too small to capture all of the knowledge in the cumbersome model, intermediate temperatures work best which strongly suggests that ignoring the large negative logits can be helpful.

Dropout can be viewed as a way of training an exponentially large ensemble of models that share weight.

training the baseline model with hard targets leads to severe overfitting (we did early stopping, as the accuracy drops sharply after reaching 44.5%), whereas the same model trained with soft targets is able to recover almost all the information in the full training set (about2% shy). It is even more remarkable to note that we did not have to do early stopping: the system with soft targets simply "converged" to 57%. This shows that soft targets are a very effective way of communicating the regularities discovered by a model trained on all of the data to another model.

For really big neural networks, it can be infeasible even to train a full ensemble, but we have shown that the performance of a single really big net that has been trained for a very long time can be significantly improved by learning a large number of specialist nets, each of which learns to discriminate between the classes in a highly confusable cluster." [1]

1. Filled in the DistilRNN class in the `model.py` file.
   DistilRNN Architecture:
   (a) **Embedding Layer**:
      - Converts input tokens to dense vectors.
      - Vocabulary size: 50,257.
   (b) **RNN Layer**:
      - Vanilla RNN with specified layers and hidden size.
      - Processes input sequences, maintaining hidden states.
   (c) **Fully Connected (FC) Layer**:
      - Maps RNN output to output space.
      - Output size: Number of classes.
   (d) **Forward Method**:
      - Embeds input sequence.
      - Initializes hidden state.
      - Passes through RNN.
      - Selects last hidden state.
      - Applies FC layer.
2. **Loss Function for the training:**
   Soft Targets Calculation:

$$\text{soft\_targets} = \text{softmax}\left(\frac{\text{teacher\_logits}}{T}\right)$$

Soft Probability Calculation:

$$\text{soft\_prob} = \text{log\_softmax}\left(\frac{\text{student\_logits}}{T}\right)$$

Soft Targets Loss Calculation:

$$\text{soft\_targets\_loss} = \frac{1}{N}\sum_{i=1}^{N}(\text{soft\_targets}_i \times (\log(\text{soft\_targets}_i) - \text{soft\_prob}_i)) \times \left(T^2\right)$$

Label Loss Calculation:

$$\text{label\_loss} = \text{loss\_fn}(\text{student\_logits}, y)$$

Overall Loss Calculation:

$$\text{loss} = \text{stl\_weight} \times \text{soft\_targets\_loss} + (1 - \text{stl\_weight}) \times \text{label\_loss}$$

- N= Batch size
- T= Temperature
- stl_weight= weight for the soft target loss (≥ 0.5)

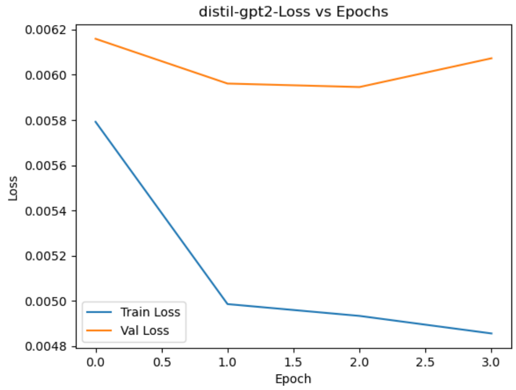
Distil Mode

Parameter	**Value**
mode	distil
sr_no	23198
gpu_id	0
gpt_variant	gpt2
max_new_tokens	100
model_path	models/LoRA.pth
lr	0.001
batch_size	128
epochs	3
LoRA_rank	4
T	2.0
stl_weight	0.75
device	cuda

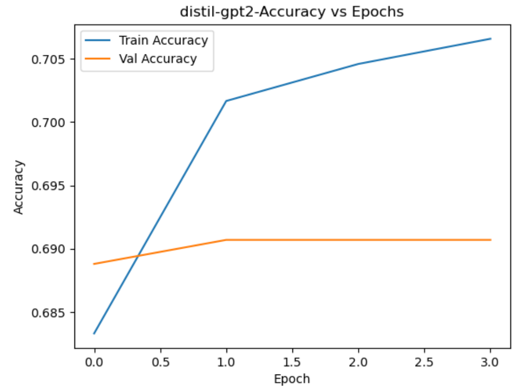
RNN Mode

Parameter	**Value**
mode	rnn
sr_no	23198
gpu_id	0
gpt_variant	gpt2
max_new_tokens	100
model_path	models/LoRA.pth
lr	0.001
batch_size	128
epochs	3
LoRA_rank	4
T	2.0
stl_weight	0.75
device	cuda

Table 2: Model Configurations



(a) Loss

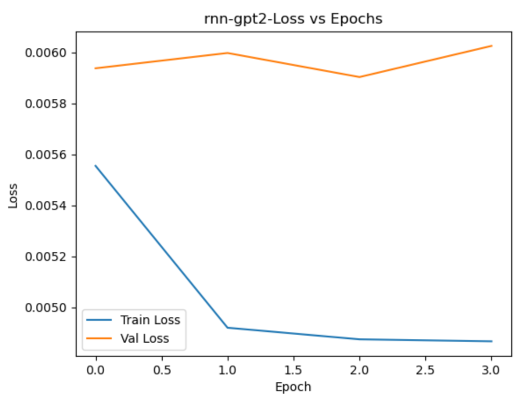


(b) Metrics

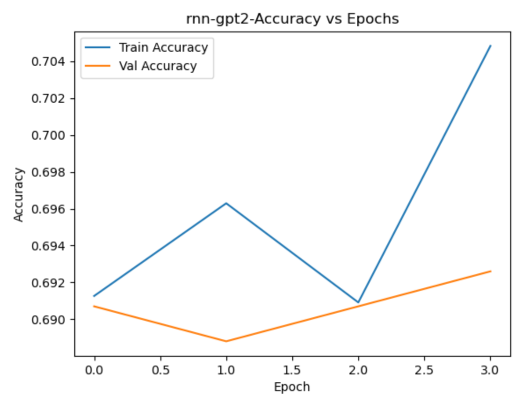
Figure 2: Distil RNN(1 layer) Loss and Metrics

* **DistilRNN model total number of parameters**:39780098
DistilRNN achieved a highest accuracy of 70.01% on the COLA validation dataset.

4. Comparison



(a) Loss



(b) Metrics

Figure 3: Student RNN(1 layer) Loss and Metrics

The highest accuracy achieved by the student RNN is 69%. Whereas, the DistilRNN achieved a highest accuracy of 70.01% on the COLA validation dataset. That is a 1% increase in the performance in the KD version over the non-KD version.

References

[1] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Algorithm 1 Training and Evaluation

```
function TRAIN(teacher_model, student_model, train_loader, args)
    Initialize loss_fn and optimizer as specified
    Initialize train_loss and train_acc to 0
    Set model_mode based on args.mode
    if teacher_model is in training mode then
        Set teacher_model to evaluation mode
    end if
    Set student_model to train mode
    for each batch (X, mask, y) in train_loader do
        Move data to device
        Reset optimizer gradients
        Forward pass through teacher_model and student_model if needed
        Calculate loss and backpropagate
        Update train_loss and train_acc
    end for
    Calculate average train_loss and train_acc over the dataset
    return train_loss, train_acc
end function
function EVALUATE(model, val_loader, args)
    Initialize val_loss and val_acc to 0
    Set model to evaluation mode
    for each batch (X, mask, y) in val_loader do
        Move data to device
        Forward pass through model
        Calculate loss
        Update val_loss and val_acc
    end for
    Calculate average val_loss and val_acc over the dataset
    return val_loss, val_acc
end function
```