كلية الهندسة
College of Engineering
QATAR UNIVERSITY جامعة قطر

Deep Learning Project

CMPS497

Dr. Tamer Elsayed

Cross Prompt Automated Essay Scoring using Neural Models

December 7th, 2024

Students:

Fatima Almohanadi 202002307

Shatha Alhazbi 202108114

Taleela Almuhannadi 202103108

Introduction:

This project explores the development and evaluation of automated essay scoring systems using neural network approaches. We implemented three approaches (Approaches A, B, and C) to address the challenge of cross-prompt essay scoring, each offering unique insights into the capabilities and limitations of neural models in essay scoring task.

Approach A establishes revolves around developing a simple feed-forward neural network with single-score prediction, holistic scoring across different prompts. Approach B extends this approach by implementing a multi-trait prediction system that predicts multiple scores of the essay. Finally, Approach C leverages BERT's pre-trained language model combined with additional features.

Table of Contents:

1. Approach A

1.1. Models and training

Our project workflow, documented on Canva (https://www.canva.com/design/DAGWwUiZwN8/Oc0XGN6T-caOJW_nycFZxQ/view?utm_content=DAGWwUiZwN8&utm_campaign=designshare&utm_medium=link&utm_source=editor), divided the process into 8 sequential steps. Each step, with clear goals, important notes, and defined outputs, guided us from data preparation (Step 1) through model design decisions (Step 2: loss function), a detailed hyperparameter tuning process using grid search with 7-fold cross-validation (Step 3), and progressive refinement like batch size optimization (Step 4). This led to a trained model, followed by final evaluation and deployment. For implementation details, refer to the Google Colab link (https://colab.research.google.com/drive/1xygJtOAgSU8RqBZzWcheTvtcciELE10w?usp=sharing), which showcases the code and results for one target model (prompt 2). The included Read_Me section explains how to configure the neural network model for different prompts during training and evaluation of essay scoring models.

1.1.1. Splitting data for cross-validation

- leave-one-prompt-out cross-validation: 8 prompts = 8 folds
- 7 folds for training, 8th fold for testing (target prompt)
- For each target prompt experiment:
  - Test set: essays of the target prompt
  - Training: essays of other 7 prompts
  - Used 7-fold cross-validation on the training data for hyperparameter tuning
  - We made sure the held out set is not used for training or seen at all by the model.

1.1.2. Training models used as test sets

To ensure proper evaluation, our test set prompt is completely isolated from the training process. Hyperparameter tuning and model training are conducted solely on the training data. Only after these steps, the held-out test set is used to evaluate the model's performance.

The training steps are:

- Used He initialization for model params
- performed the grid search using 7-fold CV to find best hyperparameters out of the given combinations:

```
k_fold = KFold(n_splits=7, shuffle=False) #shuffle = false
because each prompt will have a seperate fold, not random
hidden_units = [8, 16, 32]
num_layers_options = [1, 2, 4, 8]
learning_rates = [0.001, 0.01, 0.1]
batch_size = 4  #its fixed at this stage
```

- Then optimized batch size: [4,8,16,32]

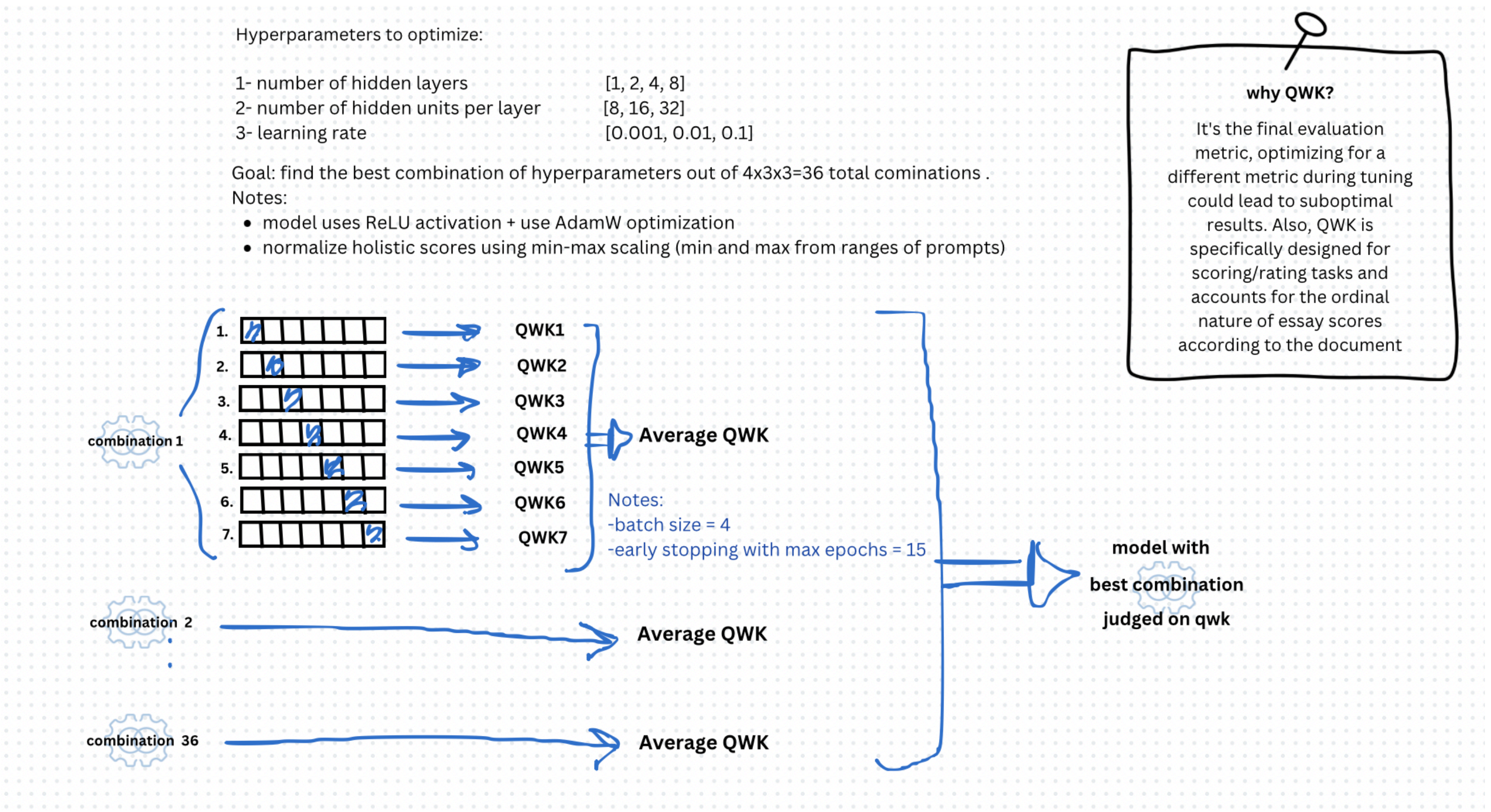
```
#after finding best hyperparameters, we optimizes batch size
train_filter = (data['prompt_ids'] != TEST_PROMPT_ID) &
(data['prompt_ids'] != VALIDATION_PROMPT_ID)
#used ~85% data (6 prompts) for training
#we used the 7th for validation (~15%)
batch_sizes = [4, 8, 16, 32]
```

- Used MSE loss with L2 regularization (λ=0.1)
- Used AdamW optimizer with early stopping (max 15 epochs)
- Selected best configuration based on average QWK scores

```
#initialized model with hyper param combo
model = NeuralNetwork(86, hidden_unit, num_layers)
optimizer = torch.optim.AdamW(model.parameters(), lr=lr,
betas=(0.9, 0.999), weight_decay=0.1)
criterion = nn.MSELoss()
#start training for max epochs 15 with early stopping
```

- Training

```
#trains final model using:
final_train_filter = data['prompt_ids'] != TEST_PROMPT_ID
#uses all data except test prompt
#uses best hyperparameters and batch size from previous steps
```


Hyperparameters to optimize:
1- number of hidden layers [1, 2, 4, 8]
2- number of hidden units per layer [8, 16, 32]
3- learning rate [0.001, 0.01, 0.1]
Goal: find the best combination of hyperparameters out of 4x3x3=36 total cominations .
Notes:
• model uses ReLU activation + use AdamW optimization
• normalize holistic scores using min-max scaling (min and max from ranges of prompts)
why QWK?
It's the final evaluation metric, optimizing for a different metric during tuning could lead to suboptimal results. Also, QWK is specifically designed for scoring/rating tasks and accounts for the ordinal nature of essay scores according to the document
combination 1
1.
2.
3.
4.
5.
6.
7.
QWK1
QWK2
QWK3
QWK4
QWK5
QWK6
QWK7
Average QWK
Notes:
-batch size = 4
-early stopping with max epochs = 15
combination 2
Average QWK
combination 36
Average QWK
model with best combination judged on qwk

1.1.3. Training models for deployment

The deployment model was trained using the best hyperparameters we got from individual prompt-specific models:

```
final_params = {
    'hidden_unit': 32,
    'num_layers': 2,
    'learning_rate': 0.001,
    'batch_size': 32
}
```

- 2 hidden layers: appeared in best performing models and it balances model capacity with training stability.
- with 32 units each: as best performing models had that value. Also, using higher capacity is safer for handling diverse essays; It's better to have extra capacity than too little to be able to capture more complex features if needed.
- Learning rate 0.001: it was consistent across all prompts

- Batch size 32: most common, as most prompts performed well with batch sizes 32. Larger batches provide more stable gradient updates and may lead to better generalization.

We used the same training setup: MSE loss, L2 reg, AdamW, early stopping. However, during training we used all available data across all prompts and we had no testing and evaluation. The model was then saved as a "model-A-deployment.pt"

1.1.4. Designing loss function

Chose MSE loss

```
criterion = nn.MSELoss()
```

As it is a regression problem with continuous numerical scores. the function will deal with normalized values between 0 and 1.

1.1.5. Handling different score ranges

Normalized holistic scores to [0,1] range during all training stages with the help of a helper function and the score_ranges dictionary given in the starter code

```
def normalize_scores(scores, prompt_id):
    score_range = SCORE_RANGES[prompt_id]['holistic']
    return (scores - score_range[0]) / (score_range[1] -
score_range[0]) #scale it between 0 and 1
```

Scaled predictions back to original range during evaluation stages:

```
def denormalize_scores(norm_scores, prompt_id):
    score_range = SCORE_RANGES[prompt_id]['holistic']
    return norm_scores * (score_range[1] - score_range[0]) +
score_range[0] #get the original score back for descaling later
```

Rounded final predictions to nearest valid score in range

1.1.6. Initializing parameters

Used Kaiming/He initialization for weights and zero initialization for biases

```
nn.init.kaiming_normal_(i.weight)  #He initialization for weights
nn.init.zeros_(i.bias)  #Zero initialization for biases
```

For hyperparam initialization, we used AdamW optimizer with L2 regularization, Grid search over 36 combinations (3 hidden units × 4 layer options × 3 learning rates), fixed batch size of

4 during grid search, then optimized later (testing 4, 8, 16, 32) we found the best hyper params based on the best performing combination during grid search, followed by batch size optimization.
Based on those best hyper params we trained the final model of that target prompt, then tested it on the target prompt.

1.2. Experimental Evaluation

1.2.1. Performance per target prompt (validation vs test)

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8
Average QWK for hyper param tuning	0.4693	0.4286	0.4238	0.4316	0.4680	0.4780	0.4621	0.5902
Validation QWK after batch size opt	0.6338	0.5485	0.4367	0.4381	0.4855	0.4267	0.4267	0.6925
Testing	0.5527	0.6176	0.5142	0.5296	0.5232	0.4291	0.4571	0.4653

Main observation on the results

From the results we can see that the QWK scores generally decrease from validation (grid search) to testing, particularly for certain prompts. The potential reasons for this may include different evaluation settings as during evaluation the model might be exposed to essays from other prompts that share similar scoring patterns and in final testing, the model faces a held-out prompt that might have unique characteristics. When testing on a completely held-out prompt, the model struggles to generalize its learned patterns. For prompt 1, we can see that QWK increases in testing than in hyperparam tuning validation, this might be due to the model being exposed to more data, allowing it to generalize better.

Some other observations:

1. Hyperparameter Tuning Performance:

Fairly consistent performance across prompts 0.42-0.59 QWK. But most prompts performed in the 0.42-0.48 range.

2. Validation After Batch Size Optimization:
   Greater variation in scores 0.3855-0.6925. We can see that prompts have better scores after optimizing their batch sizes.

3. Testing Performance:
   Testing scores fell between 0.42-0.62, with our best performing model (prompt 2) scoring 0.6276!. We got lowest performance on Prompts 6 and 7 (0.4291 and 0.4571)

Is there any discrepancy of model performance across prompts. Why?

Yes:

- Prompt 2: 0.6176 (Best performing)
- Prompts 3,4,5: ~0.51-0.53 (Mid-range performance)
- Prompts 6,7,8: ~0.42-0.46 (Lower performance)

This may be due to different score ranges of each prompt. Even though we normalize all holistic scores to [0,1] during training, the original holistic score ranges might affect how well the model can learn to predict scores because prompts with smaller ranges like those with 0-3 have fewer possible score values than those with 0-60 that might require more precision. We can see that our best performing model (prompt 2) had a holistic range of 1-6. while 7 and 8 had 0-30, and 0-60 holistic score ranges respectively. However, prompt 6 had a 0-4 range holistic score and was still amongst the lower performing model. This may be due to other reasons.

Another reason is the imbalance in the number of training examples per prompt or its relevance. If certain prompts have significantly more or fewer essays, it could explain why some prompts perform better (more training data from similar prompts) and others perform worse (less relevant training data), despite using the same model architecture.

Additionally, some prompts might share more similar scoring patterns with each other. So when testing on a particular prompt, if there are more essays from prompts with similar scoring patterns in the training data, the model is likely to perform better on that test prompt.

1.2.2. Best hyperparameter values per target prompt

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8
Hidden units	32	32	16	8	8	8	8	8
Number of layers	4	2	1	4	2	2	4	1
Learning rate	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Batch size	32	16	32	16	16	32	32	32

Hidden Units:

- Prompts 1 and 2 perform best with larger networks (32 units)
- Prompt 3 prefers medium capacity (16 units)
- Prompts 4-8 work better with smaller networks (8 units)

This suggests 1,2,3 prompts need more complex feature processing, while other prompts work better with simpler architectures.

Number of Layers Pattern:

- Varies between 1-4 layers
- Deep networks (4 layers): Prompts 1, 4, 7
- Shallow networks (1-2 layers): Prompts 2, 3, 5, 6, 8

Interesting that network depth isn’t showing a clear correlation with prompt number or performance. Maybe simpler models are just as good?

Learning Rate:

Consistent at 0.001 across all prompts which suggests this is an optimal robust learning rate for the essay scoring task.

Batch Size:

- All preferred larger batches (32) or medium (16)
- No prompts performed best with smaller batch sizes

Indicates larger batch sizes provide more stable training for the task of essay scoring.

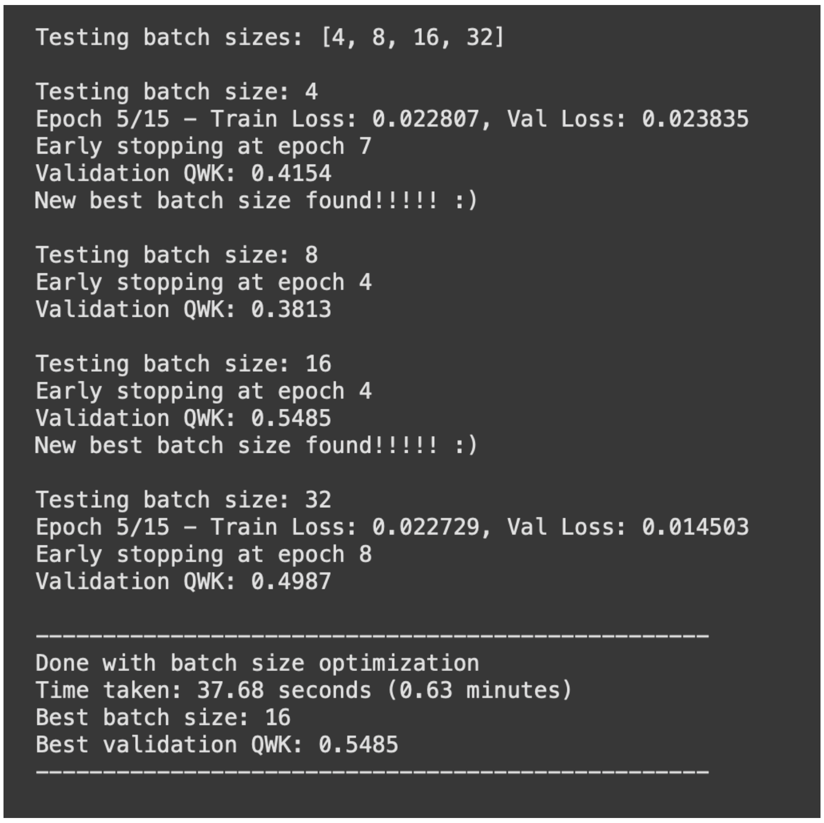
Summary:

While some parameters like learning rate are robust across prompts, the model complexity including units and layers needs to be tuned per prompt for optimal performance.

1.2.3. Effect of changing batch size

The models preferred the larger batch sizes out of the options, this could be because smaller batch sizes might lead to unstable training and a risk of shooting optimal parameters. It also adds more noise. On the contrary, larger batch sizes showed better generalization and stability.

Example, prompt 2 batch size optimization:

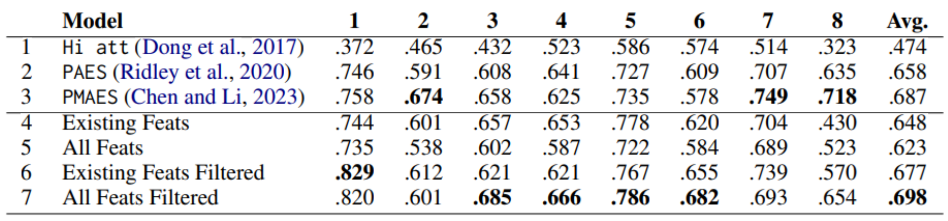


```
Testing batch sizes: [4, 8, 16, 32]

Testing batch size: 4
Epoch 5/15 - Train Loss: 0.022807, Val Loss: 0.023835
Early stopping at epoch 7
Validation QWK: 0.4154
New best batch size found!!!!! :)

Testing batch size: 8
Early stopping at epoch 4
Validation QWK: 0.3813

Testing batch size: 16
Early stopping at epoch 4
Validation QWK: 0.5485
New best batch size found!!!!! :)

Testing batch size: 32
Epoch 5/15 - Train Loss: 0.022729, Val Loss: 0.014503
Early stopping at epoch 8
Validation QWK: 0.4987

--------------------------------------------------

Done with batch size optimization
Time taken: 37.68 seconds (0.63 minutes)
Best batch size: 16
Best validation QWK: 0.5485
--------------------------------------------------
```

From the results we can see that smaller batch sizes showed less stable performance, with batch size 4 achieving a QWK of 0.4154 and batch size 8 showing the poorest performance with a QWK of 0.3813. While batch size 32 achieved OK performance with a QWK of 0.4987, it still didn't match the performance of batch size 16. This pattern suggests that batch size 16 hits a good spot in the trade-off between training stability and model performance. It was large enough to provide stable gradients but small enough to maintain good generalization. The early stopping patterns also shows that batch size

16 achieves better efficiency, it reached its best performance in fewer epochs compared to other configurations.

1.2.4. Performance comparison with state of the art models



	Model	1	2	3	4	5	6	7	8	Avg.
1	Hi att (Dong et al., 2017)	.372	.465	.432	.523	.586	.574	.514	.323	.474
2	PAES (Ridley et al., 2020)	.746	.591	.608	.641	.727	.609	.707	.635	.658
3	PMAES (Chen and Li, 2023)	.758	**.674**	.658	.625	.735	.578	**.749**	**.718**	.687
4	Existing Feats	.744	.601	.657	.653	.778	.620	.704	.430	.648
5	All Feats	.735	.538	.602	.587	.722	.584	.689	.523	.623
6	Existing Feats Filtered	**.829**	.612	.621	.621	.767	.655	.739	.570	.677
7	All Feats Filtered	.820	.601	**.685**	**.666**	**.786**	**.682**	.693	.654	**.698**

Table 1: Holistic scoring results without traits for each prompt. The results for Hi att and PAES are taken verbatim from Chen and Li (2023). The best result in each column is boldfaced.

Our best experimental results (model 2) evaluated on prompt 2 achieved the following:

- **Testing QWK:** 0.6176

Our model (Testing QWK = 0.6176) demonstrates good performance, with a difference of 0.0694 QWK points compared to the leading SOTA model PMAES (QWK = 0.687 for prompt 2 too). It is a great result given the simple architecture of our model.

Feature	**Our Model**	**SOTA Models**
Architecture	Efficient (2 layers, 32 units) Batch size 16, lr = 0.001	More complex (CNNs, LSTMs, attention mechanisms)
Parameters	Smaller	Larger
Training Process	Simpler	More complex
Performance	Competitive at lower computational cost	Better
Computational Power	Less	More

Our results suggest that simpler models can deliver good performance compared to complex, state-of-the-art models. The gap in performance between our model and the best-performing SOTA models is relatively small, and our efficient approach could be beneficial in real-world deployment where computational resources and training time are critical. Furthermore, if we explore the suggested improvements, we can optimize our model more to close the performance gap and enhance its success rate.

2. Approach B

2.1. Models and training

We aimed for a more efficient approach in Approach B. Here, we developed a single codebase that handles training and evaluation for all 9 models.

We utilized a loop to iterate through each prompt within the code. Within the loop, the model configuration adapts based on the current prompt being processed. This allows us to train all eight models (one for each prompt) using the same codebase, eliminating the need for separate implementations. after the 8 prompts, we used the best parameters to then train the deployment model within the codebase. This allowed us to run the code over multiple hours on one device.

2.1.1. Splitting data for cross-validation

Similarly to Approach A. used 7 fold cross validation with one prompt per fold excluding the test prompt. During each fold iteration one prompt becomes validation and the remaining prompts are used for training. As for validation, we also use QWK scores, however the difference was that Approach A validated only on holistic scores while Approach B validated on all available traits for that prompt.

2.1.2. Training models used as test sets

Same as Approach A.

2.1.3. Training models for deployment

After training individual models for each prompt, our code tracks best overall parameters across all prompts. Based on that, it creates a final deployment model using best overall parameters (Same as Approach A). However, in this approach we used masking to handle different trait combinations. (Will explain further in following questions)

2.1.4. Designing loss function

Chose multivariate MSE as the model is predicting multiple numerical scores

$$Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\phi],\sigma^2) = \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_d - \mathrm{f}_d[\mathbf{x},\phi])^2}{2\sigma^2}\right]$$

Dealt with the issue that different essay prompts have different scoring traits through masking any trait that doesn't apply to the specific prompt, so it does not affect the loss.
We created a MiltivariateProbLoss class in our code that implements multivariate MSE. It computes squared differences between predictions and targets, converts these to probabilities, takes the negative log probability, use masks to handle missing scores.

```
class MultivariateLossFn(nn.Module):
   def __init__(self, sigma=1.0):
       super(MultivariateLossFn, self).__init__()
       self.sigma = sigma
       self.const = 1.0 / (math.sqrt(2 * math.pi) * sigma)
   def forward(self, outputs, targets, masks):
       squared_diff = (targets - outputs) ** 2
       exp_term = torch.exp(-squared_diff / (2 * self.sigma ** 2))
       prob_term = self.const * exp_term
       masked_probs = prob_term * masks
       valid_probs = masked_probs + (1 - masks) + 1e-7
       log_prob = torch.log(valid_probs)
       loss = -torch.sum(log_prob * masks, dim=1) / torch.sum(masks, dim=1)
       return loss.mean()
```

2.1.5. Handling different score ranges

handling different score ranges was managed through a combination of normalization, denormalization, and masking. The main challenge was that different prompts had different scoring criteria:

1. Different traits being scored For example prompts 1 & 2 have sentence fluency, word choice, conventions, organization, content.
2. Different score ranges for each trait. For example, some traits had a 0-3 range.

We used a masking system in our original normalization and denormalization helper functions from approach A to handle these differences. First, we normalized all scores to a 0-1 range. But because each prompt had different traits, we needed a way to track which scores were valid. Therefore, we developed a masking system. For each essay, a mask was created that marked which traits exist for that prompt (given 1) and which traits don't exist (given 0). For example, prompt 1 essay mask is [1,1,1,1,1,1,0,0,0] (as it has the first 6 traits). prediction/evaluation, the model's predictions were denormalized back to their original ranges, and the mask was used to only denormalize valid traits. Moreover, The loss function only considers valid traits for each essay when computing the loss. The masking system allowed us to only calculate loss for traits that exist in each prompt.

2.1.6. Initializing parameters

Similarly to Approach A, we used He initialization

```
#initialize weights
for m in self.modules():
   if isinstance(m, nn.Linear):
       nn.init.kaiming_normal_(m.weight)
       nn.init.zeros_(m.bias)
```

And both approaches use the same hyperparameter ranges for grid search and batch size optimization.

2.2. Experimental Evaluation

2.2.1. Performance per target prompt (validation vs test)

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8
Average QWK for hyper param tuning	0.5143	0.5305	0.5342	0.5517	0.5455	0.5560	0.5757	0.5526
Validation QWK after batch size opt	0.4401	0.4791	0.5464	0.5900	0.5385	0.5087	0.6175	0.4083
Testing	0.5963	0.5221	0.6150	0.5959	0.5975	0.4703	0.3796	0.5339

From the results we can see that the QWK scores show different patterns compared to Approach A.

Some observations:

1. Hyperparameter Tuning Performance:

Very stable across prompts (0.51-0.57 QWK), all prompts performed well during this phase showing less variance than Approach A. This shows the multi-trait learning approach provides better initial model performance.

2. Validation After Batch Size Optimization:
   Wider variation in scores (0.40-0.61), with some prompts improving (like Prompt 7 going from 0.5757 to 0.6175) while others decreased (like Prompt 8 dropping from 0.5526 to 0.4083). The decrease might be due to our train-test approach in batch optimization. In grid search we followed cross-validation, while in batch optimization we did a simple 85% training and 15% testing split.
   - Grid search: Each validation fold contains essays from multiple prompts
   - Batch size optimization: Validation only uses essays from one prompt

3. Testing Performance:
   Good overall performance (0.52-0.61 for most prompts) with best results on Prompts 1 and 3 (0.5963 and 0.6150). Only Prompt 7 performed worse (0.3796). It is generally more consistent across prompts than Approach A

2.2.2. Best hyperparameter values per target prompt

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8
Hidden units	32	16	32	32	32	32	32	32
Number of layers	2	4	4	2	1	8	4	2
Learning rate	0.001	0.001	0.001	0.001	0.001	0.001	0.01	0.001
Batch size	4	8	32	32	16	4	32	4

Hidden Units:

- All prompts except prompt 2 preferred larger networks with 32 units. Prompt 2 performed best with 16 units

- More hidden units preference suggests that predicting multiple traits requires more complex feature processing
- Very different from Approach A where at least some prompts preferred smaller networks. 8 was the most common in approach A, while 32 is the most common here!

Number of Layers:

- Varies significantly from 1 to 8 layers
- Most common are 2 layers (Prompts 1, 4, 8) and 4 layers (Prompts 2, 3, 7)
- Prompt 6 preferred 8 layers
- Prompt 5 worked best with just a 1 layer shallow network

No clear pattern, showing that layer depth is highly prompt-specific, same as Approach A!

Learning Rate:

- Consistent at 0.001 across prompts like Approach A except one prompt, prompt 7 preferred a higher rate (0.01)
- Similarly to Approach A, we can conclude the 0.001 learning rate is optimal for essay scoring tasks.

Batch Size:

- High variability (4, 8, 16, 32)
- Smaller batches (4) worked best for prompts 1, 6, 8
- Larger batches (32) preferred by prompts 3, 4, 7

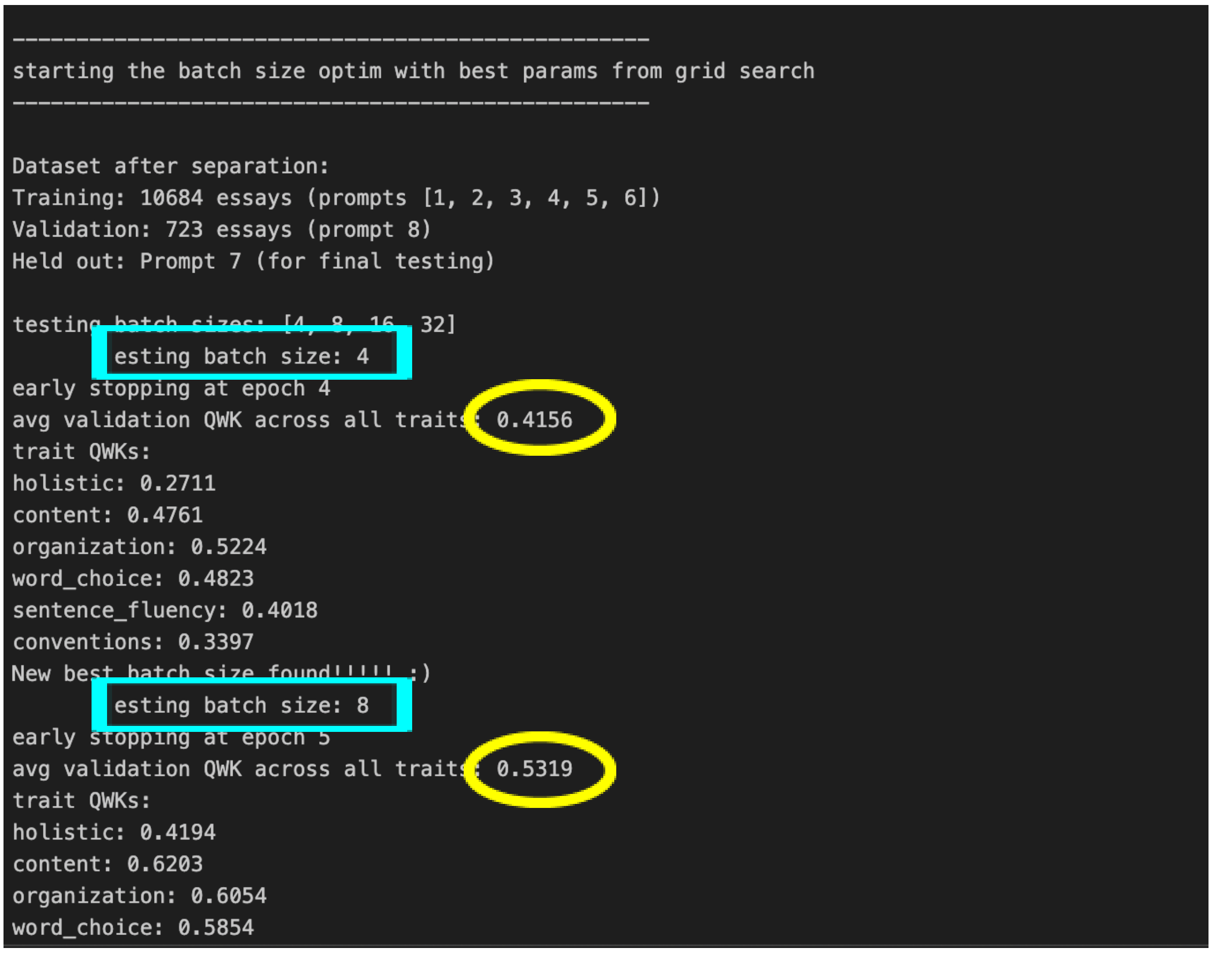
No pattern, batch size is highly dependent on prompt characteristics, unlike Approach A where batch size was consistently large (either 16 or 32).

Comparing between this approach and A, we can see that multi-trait prediction (Approach B) generally requires more model capacity but needs more careful tuning of batch size, likely due to the increased complexity of predicting multiple scores simultaneously.

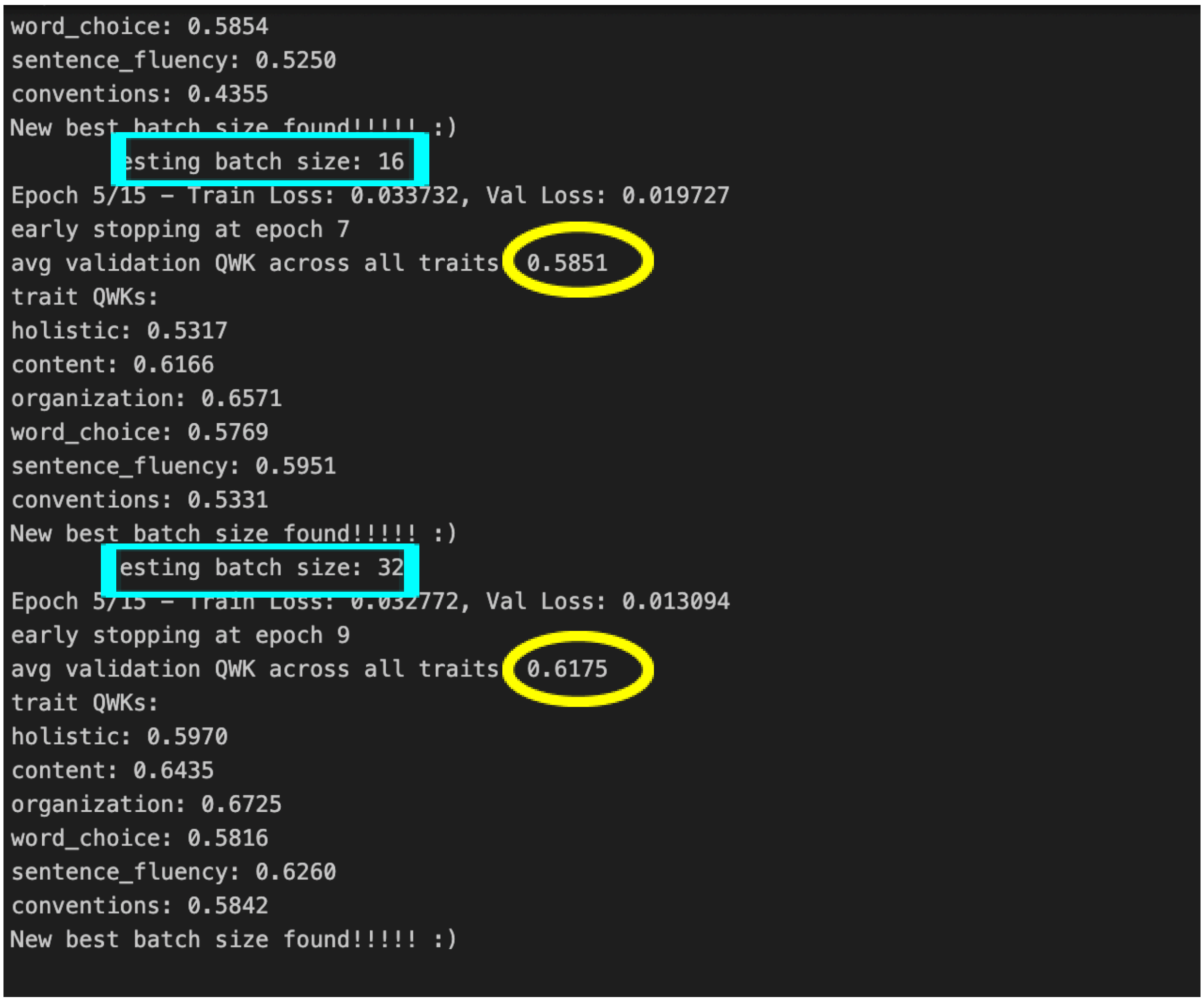
2.2.3. Effect of changing batch size

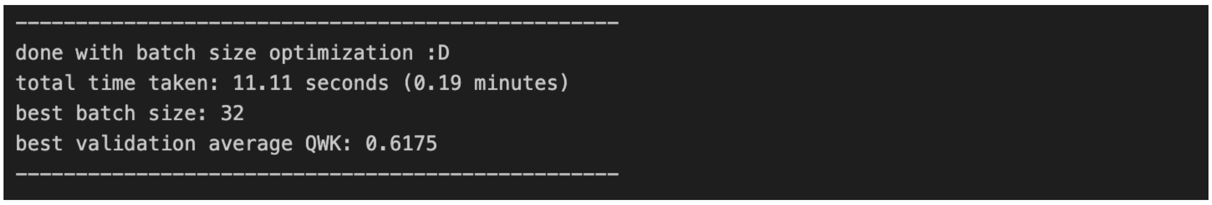
As mentioned earlier, larger batch sizes (16 or 32) performed better for some prompts like 3,4, and 7, while some prompts preferred small batch size (4) like prompt 1,6, and 8. Unlike Approach A which consistently preferred larger batches, Approach B shows more varied optimal batch sizes which suggests that predicting multiple traits makes the model more sensitive to batch size choice.
Some things to note is that larger batches (32) when optimal showed more stable validation losses while smaller batches (4,8) often led to earlier stopping.

As for performance range when changing batch size, we can use prompt 7 result as an example (batch size in cyan rectangle and corresponding score in yellow):



```
--------------------------------------------------
starting the batch size optim with best params from grid search
--------------------------------------------------

Dataset after separation:
Training: 10684 essays (prompts [1, 2, 3, 4, 5, 6])
Validation: 723 essays (prompt 8)
Held out: Prompt 7 (for final testing)

testing batch sizes: [4, 8, 16, 32]
        esting batch size: 4
early stopping at epoch 4
avg validation QWK across all traits: 0.4156
trait QWKs:
holistic: 0.2711
content: 0.4761
organization: 0.5224
word_choice: 0.4823
sentence_fluency: 0.4018
conventions: 0.3397
New best batch size found!!!!! :)
        esting batch size: 8
early stopping at epoch 5
avg validation QWK across all traits: 0.5319
trait QWKs:
holistic: 0.4194
content: 0.6203
organization: 0.6054
word_choice: 0.5854
```


```
word_choice: 0.5854
sentence_fluency: 0.5250
conventions: 0.4355
New best batch size found!!!!! :)
        esting batch size: 16
Epoch 5/15 - Train Loss: 0.033732, Val Loss: 0.019727
early stopping at epoch 7
avg validation QWK across all traits: 0.5851
trait QWKs:
holistic: 0.5317
content: 0.6166
organization: 0.6571
word_choice: 0.5769
sentence_fluency: 0.5951
conventions: 0.5331
New best batch size found!!!!! :)
        esting batch size: 32
Epoch 5/15 - Train Loss: 0.032772, Val Loss: 0.013094
early stopping at epoch 9
avg validation QWK across all traits: 0.6175
trait QWKs:
holistic: 0.5970
content: 0.6435
organization: 0.6725
word_choice: 0.5816
sentence_fluency: 0.6260
conventions: 0.5842
New best batch size found!!!!! :)
```


```
--------------------------------------------------
done with batch size optimization :D
total time taken: 11.11 seconds (0.19 minutes)
best batch size: 32
best validation average QWK: 0.6175
--------------------------------------------------
```

We can see that there is clear progression. The performance consistently improved with larger batch sizes (0.4156 → 0.5319 → 0.5851 → 0.6175) from 4 to 32 batch sizes which leads us to believe that the impact of batch size changes are significant, leading to more stable training and better prediction across all traits.

Holistic Scoring was the mosts dramatic improvement when increasing the batch size

Batch 4: 0.2711

Batch 32: 0.5970

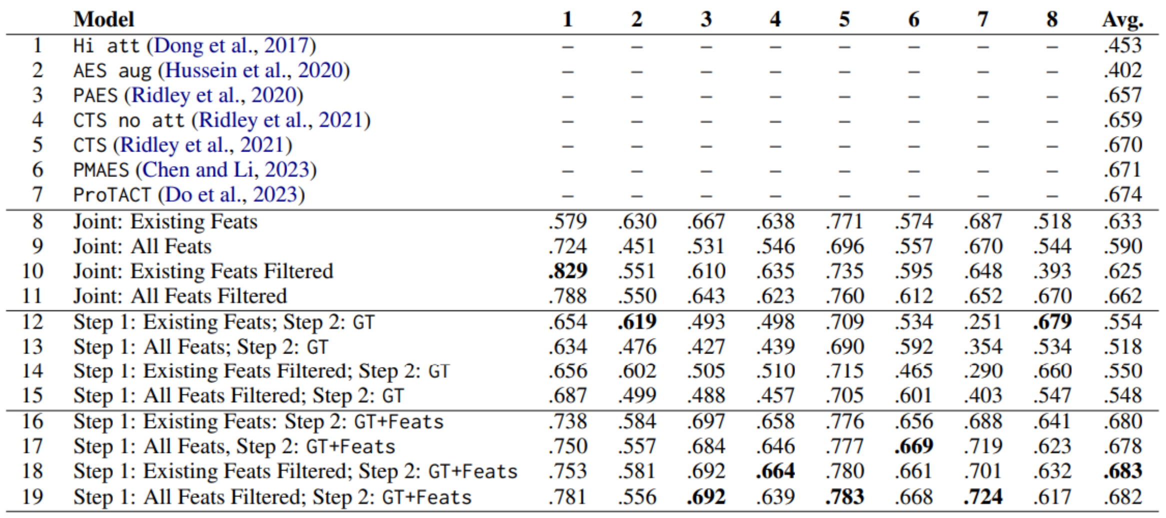
While individual traits varied but all still improved:

Organization: (0.5224 → 0.6725)

Word_choice: Least sensitive (only small improvements)

Conventions: Large improvement (0.3397 → 0.5842)

2.2.4. Performance comparison with state of the art models



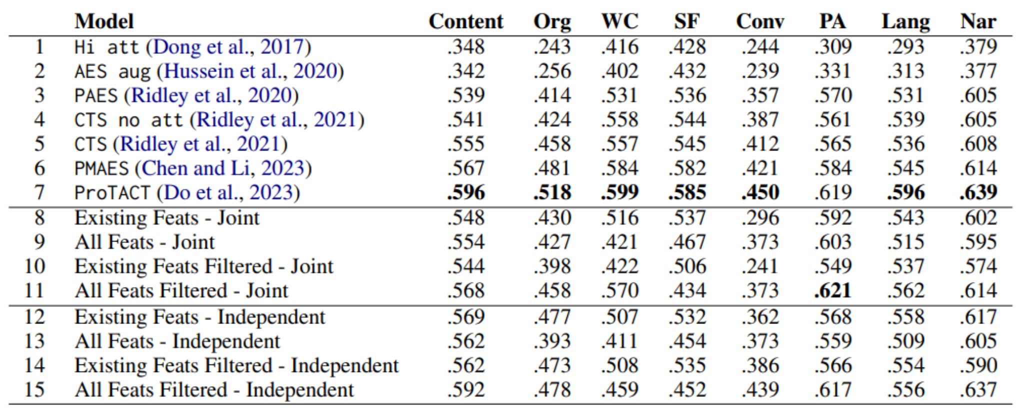
	Model	1	2	3	4	5	6	7	8	Avg.
1	Hi att (Dong et al., 2017)	–	–	–	–	–	–	–	–	.453
2	AES aug (Hussein et al., 2020)	–	–	–	–	–	–	–	–	.402
3	PAES (Ridley et al., 2020)	–	–	–	–	–	–	–	–	.657
4	CTS no att (Ridley et al., 2021)	–	–	–	–	–	–	–	–	.659
5	CTS (Ridley et al., 2021)	–	–	–	–	–	–	–	–	.670
6	PMAES (Chen and Li, 2023)	–	–	–	–	–	–	–	–	.671
7	ProTACT (Do et al., 2023)	–	–	–	–	–	–	–	–	.674
8	Joint: Existing Feats	.579	.630	.667	.638	.771	.574	.687	.518	.633
9	Joint: All Feats	.724	.451	.531	.546	.696	.557	.670	.544	.590
10	Joint: Existing Feats Filtered	**.829**	.551	.610	.635	.735	.595	.648	.393	.625
11	Joint: All Feats Filtered	.788	.550	.643	.623	.760	.612	.652	.670	.662
12	Step 1: Existing Feats; Step 2: GT	.654	**.619**	.493	.498	.709	.534	.251	**.679**	.554
13	Step 1: All Feats; Step 2: GT	.634	.476	.427	.439	.690	.592	.354	.534	.518
14	Step 1: Existing Feats Filtered; Step 2: GT	.656	.602	.505	.510	.715	.465	.290	.660	.550
15	Step 1: All Feats Filtered; Step 2: GT	.687	.499	.488	.457	.705	.601	.403	.547	.548
16	Step 1: Existing Feats: Step 2: GT+Feats	.738	.584	.697	.658	.776	.656	.688	.641	.680
17	Step 1: All Feats, Step 2: GT+Feats	.750	.557	.684	.646	.777	**.669**	.719	.623	.678
18	Step 1: Existing Feats Filtered; Step 2: GT+Feats	.753	.581	.692	**.664**	.780	.661	.701	.632	**.683**
19	Step 1: All Feats Filtered; Step 2: GT+Feats	.781	.556	**.692**	.639	**.783**	.668	**.724**	.617	.682

Table 2: Holistic scoring results with traits. The results for Hi att, AES aug and PAES are taken verbatim from Ridley et al. (2021). The best result in each column is boldfaced.

Holistic scoring with traits

	1	2	3	4	5	6	7	8	avg
Our model	0.7452	0.5437	0.6480	0.6413	0.6996	0.4354	0.6910	0.5029	0.6134
SOTA	0.829	0.619	0.629	0.664	0.783	0.669	0.724	0.679	0.683

For the holistic scoring results per prompt, our model shows different performance compared to SOTA, however, most of our scores are still relatively close. Since both our approach and most of the SOTA here used neural networks, the performance difference might be due to specific architectural choices or training strategies in the SOTA models. While there's room for improvement, achieving around 90% of SOTA performance suggests our model provides a solid foundation for automated essay scoring.



	Model	Content	Org	WC	SF	Conv	PA	Lang	Nar
1	`Hi att` (Dong et al., 2017)	.348	.243	.416	.428	.244	.309	.293	.379
2	`AES aug` (Hussein et al., 2020)	.342	.256	.402	.432	.239	.331	.313	.377
3	`PAES` (Ridley et al., 2020)	.539	.414	.531	.536	.357	.570	.531	.605
4	`CTS no att` (Ridley et al., 2021)	.541	.424	.558	.544	.387	.561	.539	.605
5	`CTS` (Ridley et al., 2021)	.555	.458	.557	.545	.412	.565	.536	.608
6	`PMAES` (Chen and Li, 2023)	.567	.481	.584	.582	.421	.584	.545	.614
7	`ProTACT` (Do et al., 2023)	**.596**	**.518**	**.599**	**.585**	**.450**	.619	**.596**	**.639**
8	Existing Feats - Joint	.548	.430	.516	.537	.296	.592	.543	.602
9	All Feats - Joint	.554	.427	.421	.467	.373	.603	.515	.595
10	Existing Feats Filtered - Joint	.544	.398	.422	.506	.241	.549	.537	.574
11	All Feats Filtered - Joint	.568	.458	.570	.434	.373	**.621**	.562	.614
12	Existing Feats - Independent	.569	.477	.507	.532	.362	.568	.558	.617
13	All Feats - Independent	.562	.393	.411	.454	.373	.559	.509	.605
14	Existing Feats Filtered - Independent	.562	.473	.508	.535	.386	.566	.554	.590
15	All Feats Filtered - Independent	.592	.478	.459	.452	.439	.617	.556	.637

Table 3: Trait scoring results (Org: Organization, WC: Word Choice; SF: Sentence Fluency; Conv: Conventions; PA: Prompt Adherence; Lang: Language; Nar: Narrativity). The results for `Hi att`, `AES aug` and `PAES` are taken verbatim from Ridley et al. (2021). The best result in each column is boldfaced.

Category	SOTA	Our model
Content	0.596	0.5579
Organization	0.518	0.4755
Word choice	0.599	0.5448
Sentence fluency	0.585	0.5539
Conventions	0.450	0.4042
Prompt adherence	0.621	0.5656
Language	0.596	0.5034
Narrativity	0.639	0.5858

Looking at the trait scoring results, our model performs quite competitively compared to SOTA, despite using a simpler approach. For most traits, we achieve within 0.03-0.06 of SOTA performance. What is interesting is that while SOTA results were achieved using complex model architectures, our results come from a simple neural network model. This suggests that simpler architectures can be effective for automated essay scoring when well designed.

3. Approach C

Link to model :
https://drive.google.com/file/d/1Ty8AtODTEdfamf95fI2NkIpDaVAiqUTG/view?usp=sharing

3.1. Splitting the data for cross-validation

We split the data the same way we did in approach A and B for the cross validation, where we have 7 k fold cross validation and each fold represents a prompt, and in each iteration a different prompt will be set as the validation fold

3.2. Training the models used for the test sets

We trained the model here by first performing grid search to tune the design of the regression head layer using cross validation with prompts 2-8 and getting their average performance. We tried 3 approaches:

I. just one output node for predicting the holistic score, with no hidden layers
II. one hidden layer of 4 hidden units before the output node
III. one hidden layer of 8 hidden units before the output node.

Once we found the best regression head design, we used it to train our final model over the whole training set (2-8 prompts) and then tested it on prompt 1.

3.3. Handling the different score ranges

We handled the different score ranges the same way we did in approach A & B which was by normalizing the scores first to get 0-1 then denormalizing when evaluating

3.4. Model and training

3.4.1. CLS and essay features input

```
encoding = tokenizer(
            essay,
            max_length=self.max_length, #512 for BERT
            padding='max_length', #add padding if <512
            truncation=True, #cut if >512
            return_tensors='pt'
```

```
       )
```

We first pass our essay through BERT’s tokenizer to break our essay into tokens, this then returns input_ids that represent each token + attention masks that show what is real text and what is just padding

```
       outputs = self.bert(input_ids=input_ids,
             attention_mask=attention_mask, return_dict=True)
```

the outputs line takes the tokenized numbers of our essay and creates an embedding for each token in our essay text, it contains a representation for every token

```
cls_embedding = outputs.last_hidden_state[:, 0, :]
```

then we take the CLS embedding which is at index 0. BERT puts a special CLS token at this position before reading the essay, and while BERT reads the text, this token learns about all the other words → making it a good summary of the whole essay

```
combined_features = torch.cat([cls_embedding, features], dim=1)
```

and then we combine it with our 86 features, so now combined features = 768 (CLS length) + 86 = 854

```
return self.regression_head(combined_features)
```

finally we pass the combination through the regression head

3.4.2. Parameter initialization

```
self.bert = BertModel.from_pretrained('bert-base-uncased')
```

we first get the pre-trained model and get its parameters

```
for module in self.regression_head.modules():
           if isinstance(module, nn.Linear):
               nn.init.kaiming_normal_(module.weight)
               nn.init.zeros_(module.bias)
```

then whenever we add a new regression head linear layer on top of BERT, we initialize its weights using He initialization and set biases to 0.

3.5. Experimental evaluation

3.5.1. Performance on Prompt 1 (validation vs. test)

QWK score in validation: 0.5428

QWK score in testing: 0.7965

Although this is a bit unusual, the score in testing might be better due to the fact that the model has been trained over all the 2-8 prompts while in validation in each iteration a prompt is not being used in training. This exclusion in the validation phase limits the model's exposure to some patterns, which could explain the lower validation score.

3.5.2. Best configuration of the regression head

The best configuration of the regression head we found was just one output node for predicting the holistic score, with no hidden layers. This could be because as we add a layer with different number of units, it adds complexity and might lead to the model overfitting. Additionally, since we already have many features from both BERT's CLS embedding (size 768) and our essay features (86 features), a simple linear layer was enough to combine these into a final score.

3.5.3. Performance comparison with state-of-the-art models

Our QWK while testing was 0.7965, while the SOTA QWK for prompt 1 was 0.829 which was by the neural network model as we can see in Table 2 above.. The difference between the two results is very minimal and is only at 0.0325. This means our results are quite competitive with SOTA performance as it achieves 96% of its score. Although this suggests our approach of combining BERT's CLS embedding with our features was effective for essay scoring, it shows that we can still achieve similar or even better results with neural networks alone.

Student Contributions

Student	Contribution %
Fatima	33%
Shatha	33%
Taleela	33%

Phase 1

Although tasks were assigned initially, each group member did not stick with the assigned task, The actual implementation was much more collaborative. The team worked together using Google Colab as a shared development environment, so everyone contributed to different parts of the code collectively rather than sticking to strict task assignments, including code debugging.
Fatma: Planning, nn architecture, loss decision, reporting, running models….
Taleelah: implementing plan, batch size optimization work, reporting, running models….
Shatha: grid search work, debugging, models running, training loops..

Phase 2:

The team maintained their collaborative approach while working on the more complex multi-trait prediction system. Any decision was made by the team like the loss function and the deployment model. In addition, all Assisted in debugging normalization issues. Fatma mostly controlled the core logic, Shatha implemented the unified codebase for training all 9 models, and Taleela contributed to performance analysis and debugging.
As the deadline got closer, Taleela worked on fixing any issues with approach A, Fatma focused on Approach B, and Shatha on C. For the main decisions in C, the team also collaborated closely, with Shatha working the most on this approach by setting up the BERT model integration, managing the tokenization process, etc.. and the rest of the team assisted in documentation, analysis and running.

Each team member maintained equal contribution (33%) throughout both phases, with frequent collaboration and support across all tasks. We mostly had meetings where we discussed what each of us had to do at every stop to help. Genuinely, there was no specific structure in working for this project.

Reflections

Approach A

Approach A of this project was a tough journey for our team. While we're not entirely sure about our models, we are sure that we put our hearts into understanding and solving each challenge that came our way.

We discovered firsthand just how tricky it is to build a nn model. Every time we thought we had it figured out, something new would come up whether it was dealing with different score ranges, understanding the problem and QWK scoring, or trying to figure out why our model's performance was the way it was.

Our team spent countless hours debugging together in Google Colab. We wish we had practiced more or given more about pytorch throughout the course so we do not need to have as many issues as we did.

We know we gave it our all and learned a lot in the process. The hands-on experience with deep learning has given us skills we'll definitely use in the future.

Especially the hyper-parameter tuning part. While studying we thought it would be the easiest, but in reality it was the hardest part while defining the model structure was extremely unexpectedly simple.

Approach B & C

Starting approach B we were aware of the computational resources and the time needed to train models. Rather than trying to split the work across multiple devices, we made the decision to optimize our code for sequential execution on a single GPU. This meant our code would run for many hours, training all 9 models at once. This is of course after we made sure one model was running as expected.

The long training times were frustrating, especially when we had to restart due to errors. However, this forced us to be more careful with our code changes. Each run was precious because of how long it took.

The masking system was a big challenge to implement correctly. It took us many attempts to correctly handle different trait combinations across prompts. The concept seemed simple at first, but getting the masks to work properly with our loss function was much harder than we expected, especially since any issue with normalization immediately made the model's QWK 0!

Approach C was both challenging and the most rewarding part of our project. Learning about encoders and fine-tuning deep models felt intimidating at first, but implementing a real life solution that used essay texts combined with the features made transformers a lot more interesting. It was exciting to see how the model could process and analyze text data, and function similarly to how systems work in real life applications. We faced similar challenges to the ones we saw in A and B such as ensuring the scores are within the correct ranges for each prompt but since we finished both approaches before, it was much easier to implement the correct logic and debug any issue we come across.

Despite these challenges, It was very rewarding to see the result and compare it with state-of-the-art-models. The experience taught us valuable lessons about building flexible neural architectures. Looking back, while this approach was more complex than Approach A, it gave us deeper insights into handling heterogeneous data structures in neural networks.