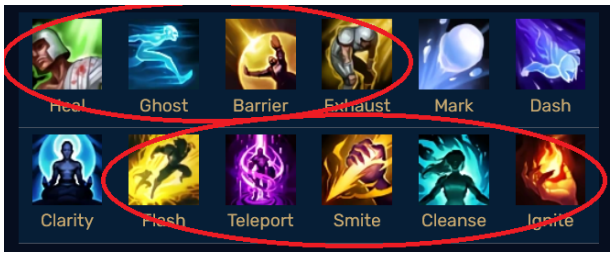
Summoner Spell Recognition and Tracking for League of Legends

EE541: Final project report

Adhithya Bhaskar
adhithya@usc.edu

1) Introduction

The objective of this project is to develop a tool capable of recognizing and tracking summoner spells in League of Legends by analyzing in-game chat. When an opponent uses a summoner spell, any player can "ping" the spell, creating a record in the chat. Our goal is to recognize and keep track of enemy champions and their summoner spells whenever they are pinged in the chat. The image below illustrates the spells of interest, which are the most commonly used and crucial in standard gameplay.



When an opponent uses a summoner spell, any player can "ping" the spell, creating a record in the chat, as depicted in the figure below.

1.1) Timers

Each summoner spell has a predefined cooldown period that significantly impacts the game. Forgetting the Flash timer for even a single opponent can alter the game's outcome. In high-level professional play, a single flash can make the difference between winning or losing a tournament.

The following table lists the cooldown times for each spell:

Spell	Cooldown time (sec)
Heal	240
Ghost	210
Barrier	180
Exhaust	210
Flash	300
Teleport	360
Unleashed teleport	240

Smite	90
Cleanse	210
Ignite	210

2) Dataset

2.1) Synthesis

In order to generate a set of data for training we use the `trdg` library [1] for synthetic text generation in conjunction with Pillow and RandomName generator. Here is an overview of the steps taken:

- Generate synthetic text with `trdg` and augment for the colors, fonts, backgrounds and artifacts that could be observed during a game [2]
- We captured the following details from the official league of legends design page and other tools (color pickers, font recognition, etc.)
  - Text format:
    - `["Timestamp"] "Teammate username" ("Teammate champion"): "Opponent champion" "Summoner Spell"`
    - Example: `[07:04] Curling Captain (Lee Sin): Darius Ghost`
  - Font used in the League of Legends chatbox: "Gill Sans - Regular"
  - Colors:
    - Timestamp: (white) `#d4d7c7`
    - Teammate username and (Teammate champion): (blue) `#4da7d1`
    - Opponent champion: (red) `#d83730`
    - Spell: (yellow) `#e6ab2b`
  - Background generation
    - There are 3 categories of images we need to synthesize in order to cover the full range of potential inputs:
      1. `Opponent Champion - Summoner spell` - This is the category that we are interested in tracking
      2. Empty chat - This would consist of the background image without any text
      3. Random, irrelevant text

Initially, we created backgrounds using sampled images from the game, but the results were not satisfactory. The backgrounds were too simple and solid. We then decided to use screen-recorded gameplay footage and extracted frames from it. We used random crops of the frames as background images for the synthetic dataset.

2.2) Data Split

Since the dataset is composed of "non-matches" and combinations of `Opponent Champion - Summoner spell` (matches), and we want to recognize only the matches, we need to ensure that we are not ignoring the y=-1 labels (which are non-matches). There are 164 champions and 11 relevant spells in the game leading to the number of classes: $\text{Number of champions} * \text{Number of spells} = 1804$ In order to train enough so that the model is able to distinguish each of those class, we would need a very large amount of samples for each class. To make the project more computationally tractable for this class, I have reduced the sample space of champions to a random set of 50 and the summoner spells the 5 most important ones used in the game - "Heal", "Ghost", "Exhaust", "Flash" and "Ignite".

The `show_sample_images` function in `src/prepare_dataset.py` splices together the images and labels into one large image for the corresponding DataLoader. Here are the first 5 images in the training and test sets:

Figure 3: `show_sample_images(train_loader, output_path=output_path, name="train", num_images=5, start_index=0)`

Figure 4: `show_sample_images(test_loader, output_path=output_path, name="test", num_images=5, start_index=0)`

Before fixing the class imbalance and reducing the scope of the dataset, the dataset was composed of 1804 classes and there were only 1-5 images for each of those classes. After reducing the scope of the dataset to 50 champions and 5 spells, and implementing a minimum number of images per class during generation, the dataset became more balanced, with >100 images per class in the training set. The dataset statistics are calculated and displayed as follows:

```
Global seed set to 42
---- Train dataset statistics ----
Number of champions: 17
Number of spells: 5
Number of unique champion-spell combinations (classes): 85
Non-matching cases: 732
Total images: 12601

Images per class (first 5):
Viego-Exhaust: 126
Tryndamere-Ghost: 137
Yasuo-Ghost: 152
K'Sante-Heal: 146
Nami-Ignite: 140

---- Validation dataset statistics ----
Number of champions: 17
Number of spells: 5
Number of unique champion-spell combinations (classes): 85
Non-matching cases: 133
Total images: 2700

Images per class (first 5):
Nautilus-Exhaust: 29
Yasuo-Heal: 35
Vayne-Exhaust: 41
Teemo-Ghost: 36
Warwick-Ghost: 25

---- Test dataset statistics ----
```

```
Number of champions: 17
Number of spells: 5
Number of unique champion-spell combinations (classes): 85
Non-matching cases: 137
Total images: 2701

Images per class (first 5):
Zoe-Exhaust: 31
Vayne-Ignite: 34
Kha'Zix-Exhaust: 31
Zoe-Ghost: 29
Jax-Flash: 36
```

3) Challenges faced with data and training iterations

4) Model architecture and training

We are using PyTorch Lightning, a library that encapsulates a lot of repetitive code we encountered (training loops, metric calculation, etc.) in previous homeworks and provides a more structured and simplified way to write deep learning code. This allows for a more efficient training process and improved reproducibility as we can also integrate logging and checkpointing with ease.

For this image classification problem we use ResNet-50 [3] model. In the initial implementation, the pre-trained weights from the ResNet-50 model are used, and the last fully connected layer is replaced with a new linear layer to match the number of target classes required for the task at hand. The number of classes is calculated based on the champions and spells used in the dataset which can be changed in by calling the training script with arguments: `poetry run python src/train.py --num_champions 50 --target_spells ("Heal", "Ghost") --accelerator 'gpu' --max_epochs 15 --devices 1`. The minimum number of images per class and the number of images can also be changed using the arguments `--min_images_per_class` and `--num_images`.

During the training process, a number of additional techniques are employed to improve the model's performance:

- Early stopping is used to halt the training when the validation loss stops improving, avoiding overfitting and reducing the overall training time.
- A learning rate scheduler is used, specifically the `ReduceLROnPlateau` scheduler, which decreases the learning rate when the validation loss plateaus, allowing the optimizer to fine-tune the model parameters more effectively
- We also use a `ModelCheckpoint` callback to save the best model based on the validation loss
- A dropout layer is added to the model to reduce overfitting
- Color jittering is used to augment the training data and improve the model's generalization ability. We initially tried to transform the image but this resulted in the images being too skewed and the model was not able to learn anything.

We use Adam optimizer with a cross-entropy loss function and track the accuracy, precision, recall, and F1 score. These metrics were key in identifying and diagnosing the class imbalance issue were the code was initially performing suspiciously well: 0.99-1.0 for all the metrics. This was due to the fact that the model was predicting the majority class (non-matches) for all the images. This was fixed by reducing the scope of the dataset and ensuring that there are enough images per class.

The model's training progress is logged using TensorBoard, which allows us to visualize the training and validation loss, accuracy, precision, recall, and F1 score in real time.

5) Results

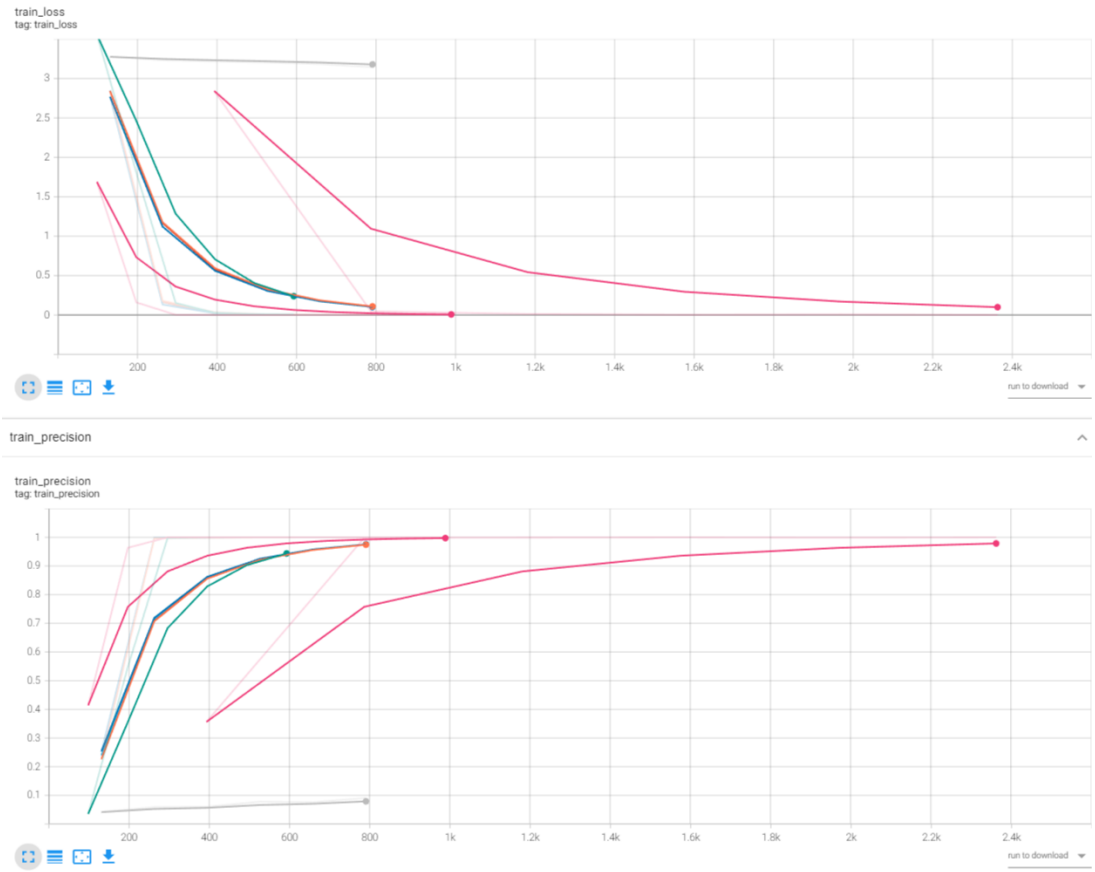
The training code run with the following parameters: `poetry run python src/train.py --accelerator 'gpu' --max_epochs 15 --devices 1`.

```
Training on 1...
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name              | Type               | Params
-------------------------------------------------------
0 | model             | ResNet             | 25.6 M
1 | dropout           | Dropout            | 0
2 | feature_extractor | Sequential         | 23.5 M
3 | classifier        | Linear             | 178 K
4 | loss              | CrossEntropyLoss   | 0
5 | accuracy          | MulticlassAccuracy | 0
6 | precision         | MulticlassPrecision| 0
7 | recall            | MulticlassRecall   | 0
8 | f1_score          | MulticlassF1Score  | 0
-------------------------------------------------------
25.7 M    Trainable params
0         Non-trainable params
25.7 M    Total params
102.941   Total estimated model params size (MB)
Epoch 0: 100%|| 394/394 [01:31<00:00,  4.33it/s, v_num=18, val_loss=0.965, val_acc=0.965, v
aMetric val_loss improved. New best score: 0.965

Epoch 4:  85%|| 336/394 [01:13<00:12,  4.59it/s, v_num=18, val_loss=1.000, val_acc=1.000, v
al_precision=1.000, val_recall=1.000, val_f1=1.000, tEpoch 4: 100%|| 394/394 [01:32<00:00,
 4.26it/s, v_num=18, val_loss=1.000, val_acc=1.000, val_precision=1.000, val_recall=1.000, v
al_f1=1.000, tEpoch 00005: reducing learning rate of group 0 to 1.0000e-05.

Epoch 5: 100%|| 394/394 [01:33<00:00,  4.19it/s, v_num=18, val_loss=1.000, val_acc=1.000, v
al_precision=1.000, val_recall=1.000, val_f1=1.000, tMonitored metric val_loss did not impro
ve in the last 5 records. Best score: 0.965. Signaling Trainer to stop.

Epoch 5: 100%|| 394/394 [01:34<00:00,  4.17it/s, v_num=18, val_loss=1.000, val_acc=1.000, v
al_precision=1.000, val_recall=1.000, val_f1=1.000, t
Testing...
Restoring states from the checkpoint path at tensorboard_logs/summoner-spell-tracker/version
_18/checkpoints/epoch=5-step=2364.ckpt
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Loaded model weights from the checkpoint at tensorboard_logs/summoner-spell-tracker/version_
18/checkpoints/epoch=5-step=2364.ckpt
Testing DataLoader 0: 100%|
                         | 85/85 [00:06<00:00, 12.96it/s]

Test Metrics
```

Test metric	DataLoader 0
test_acc	0.9097
test_precision	0.9163
test_recall	0.9097
test_f1	0.9114

When the number of classes were reduced 20 champions and 5 spells, the model was able to achieve an accuracy of 0.901 on the test and a precision of 0.917. However, when the number of classes were increased to 50 champions and 5 spells, the model's performance dropped significantly, with an accuracy of 0.51 and a precision of 0.48 on the test set. This is likely due to the class imbalance issue, where the model is predicting the majority class (non-matches) for all the images. This also ties into the problem of overfitting when were used the full dataset with 1804 classes. This was fixed by reducing the scope of the dataset and ensuring that there are enough images per class.

To address this issue, further analysis of the dataset's class distribution and the use of techniques such as oversampling, undersampling, or applying class weights during training may be necessary to ensure a fair evaluation of the model's performance. Even though the current loss, accuracy and precision curves are significantly better than the initial implementation, there still seems to be some unusual behaviour causing the validation loss to prematurely plateau. This could be due to the class imbalance issue or the fact that the model is not complex enough to learn the features of the images. We could try other model architectures like VGG [4] and DenseNet [5] in combination with this model to see if it improves the performance.



6) Extensions and future work

In the future, the model architecture could be further improved by exploring more advanced techniques such as attention mechanisms, which can help the model focus on the most relevant parts of the input image. LSTMs and RNNs layers could also be implemented, as they can capture text sequences more effectively than CNNs.

A major area of improvement is to use a OCR or image-to-text model instead of the image classification model like in this project. This would allow us to extract the text from the images and use that to predict the champion and spell. This would be more robust to changes in the UI and would allow us to use the model even if the UI changes.

We have already partially implemented a OCR recognition for the image names to create a labelled list of images from frames extracted from the screenrecorded videos. This is done using the `pytesseract` library. However, the accuracy of the OCR is not high enough to be used in the model. This could be improved by using a more advanced OCR model or by training a custom OCR model on the image names.

Bibliography

[1] E. Belval, "TextRecognitionDataGenerator," 2023.

[2] D. Etter, S. Rawls, C. Carpenter, and G. Sell, "A Synthetic Recipe for OCR," in *2019 Int. Conf. Document Anal. Recognit. (Icdar)*, Sydney, Australia, Sep. 2019, pp. 864–869, doi: 10.1109/ICDAR.2019.00143.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," arXiv, 2015.

[4] K. Simonyan, and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv, 2015.

[5] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," arXiv, 2018.