Final Technical Report

Team: Four

Vsevolod Klyushev - v.klyushev@innopolis.university

Dmitry Beresnev - d.beresnev@innopolis.university

Ivan Inchin - i.inchin@innopolis.university

Github: https://github.com/Kadaverciant/RL_for_keyboard_layout

Topic: Generation of the optimal keyboard layout for programmers using RL methods.

Problem description

Our task is to find more optimal keyboard layout for code writing. Existing keyboard layouts were created for writing literature text. However, the code-writing process has its features. One of the most significant changes involves the frequent incorporation of special characters, as well as the increased usage of specific words, phrases, and expressions.

A unique keyboard layout might be generated by $\frac{134!}{(4!\cdot 4!\cdot 4!\cdot 8!\cdot 14!\cdot 2\cdot 2\cdot 2)} \approx 5\cdot 10^{207}$ ways. The computer performs about 10^8 simple operations over 1 second. It could be assumed that the metric calculation for an arbitrary keyboard layout requires just one simple operation, then it would take more than $1.5\cdot 10^{192}$ years to check every possible combination. Therefore random generation is not good way to solve such problem.

Dataset

We decided to use dataset from kaggle with C/C++ code snippets. It contains almost 120000 different programs with mean length 485 symbols. We cleaned all data, connected lines, encoded each symbol in it (except unknown one, for example, Chinese characters in comments) and padded with spaces.

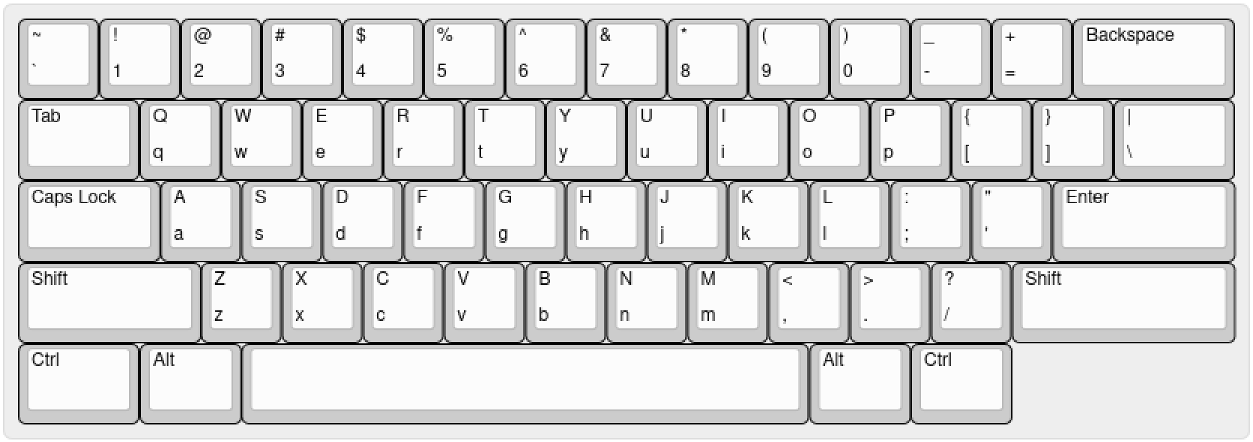
Environment description

Let’s check standard QWERTY layout representation

```
High layout:
~       !       @       #       $       %       ^       &       *       (       )       _       +       <back>
<tab>   Q       W       E       R       T       Y       U       I       O       P       {       }       |
<caps>  A       S       D       F       G       H       J       K       L       :       "       <enter> <enter>
<shift> <shift> Z       X       C       V       B       N       M       <       >       ?       <shift> <shift>
<ctrl>  <alt>   <space> <space> <space> <space> <space> <space> <space> <alt>   <ctrl>


Low layout:
`       1       2       3       4       5       6       7       8       9       0       -       =       <back>
```

```
<tab>   q       w       e       r       t       y       u       i       o       p       [       ]       \
<caps>  a       s       d       f       g       h       j       k       l       ;       '       <enter> <enter> D
<shift> <shift> z       x       c       v       b       n       m       ,       .       /       <shift> <shift>
<ctrl>  <alt>   <space> <space> <space> <space> <space> <space> <space> <alt>   <ctrl>
```


~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + = Backspace
Tab Q q W w E e R r T t Y y U u I i O o P p { [}] | \
Caps Lock A a S s D d F f G g H h J j K k L l : ; " ' Enter
Shift Z z X x C c V v B b N n M m < , > . ? / Shift
Ctrl Alt Alt Ctrl

Since actual keyboard layout doesn’t have rectangular shape we made several simplifications.

There are 14 keys in first 4 rows and 11 in last one.

STherefore we have 102 unique symbols/keys and 134 cells/positions.

We can use these 134 cells/positions as state representation of our environment.

The base of our environment is 2 dictionaries and 2 two-dimensional arrays (one per each layout). In dictionaries we store list of cells/positions that are dedicated for special symbol. Arrays represents cells/positions. In each cell we store one symbol/key.

We create two classes that are necessary for our task: `Finger` and `KeyboardLayout`.

`Finger` class:

- Tracks position of the single finger
- Returns to default position after several ticks (keys typed)
- Change position of the single finger
- Returns distance that finger has to overcome during movement process

`KeyboardLayout` class:

- Contains lowercase and uppercase layouts as well as dictionaries with coordinates for each symbol
- Contains 10 fingers and operates with them
- Accumulates total distance that all fingers passed
- Has the ability to reach symbol from uppercase layout via combination `shift+key`
- Can swap different symbols in both lowercase and uppercase layout

Now let’s talk more about distance/value function:

$$\lambda_{row}|x_1 - x_2|^2 + \lambda_{col} \cdot |y_1 - y_2|^2 + P$$

where

- x_1, y_1 - coordinates of current finger position
- x_2, y_2 - coordinates of target cell/key
- $\begin{cases} P = 1, \text{ if } |y_1 - y_2| > 3 \\ P = 0 \text{ otherwise} \end{cases}$
- λ_{row} - penalty multiplier for moving between rows
- λ_{col} - penalty multiplier for moving between columns

We decided to take $\lambda_{row} = 1$ and $\lambda_{col} = 1.2$ because moving between rows is easier in terms of fingers displacement than moving between columns. We choose such P because moving further than 2 rows usualy requires significant displacement of the wrist.

Solution architecture

We decided to use two neural networks, one for state estimation and one for decision making. Such approach is called Actor-Critic. However, in our case reward depends not only on the current state (keyboard layout), but also on input text. Therefore, we had to make some adjustments to default Actor-Critic idea. We use one State value NN that is responsible for estimating state value (total score) separately on some dataset. Later we trained Policy NN for action decisions.

To make the State value NN for score estimation we took several linear and ReLU functions with input size 134 and output size 1 with MSE loss.

Now let's focus on Policy NN. We have 67 keys in each layout. There would be 8911 unique ways to take 2 cells and swap them:

- $67 \cdot 66/2 = 2211$ possible ways to take 2 cells from low layout (order does matter)
- $67 \cdot 66/2 = 2211$ possible ways to take 2 cells from high layout (order does matter)
- $67^2 = 4489$ possible ways to take 1 cell from low layout and 1 from high layout.

So, action space size is quite big. We decided to use 2 linear layers (134x128 and 128x8911) with ReLU as activation function for first layer and SoftMax for final one. In order to decide which action to take on current state we performed sampling using the distribution based on Policy network output.

Train process

Everything was trained on Windows 10, Python 3.10 and CUDA 11.8 (6gb GPU).

We decided to train our Policy NN on first 10 programs from our dataset. The reasons for that are time and computational resources limitations. We initialized Policy NN with SGD optimizer with `lr=1e-3`.

We used QWERTY layout as starting state. Then we conducted several episodes of training, in each episode we made 100 swaps maximum. Stopping criteria for each episode were either repetition of state within episode or exceeding the QWERTY score by more than 1000. As a reward we use difference between current score and QWERTY score on our dataset. Loss was multiplication of reward and probability of selected action.

Results

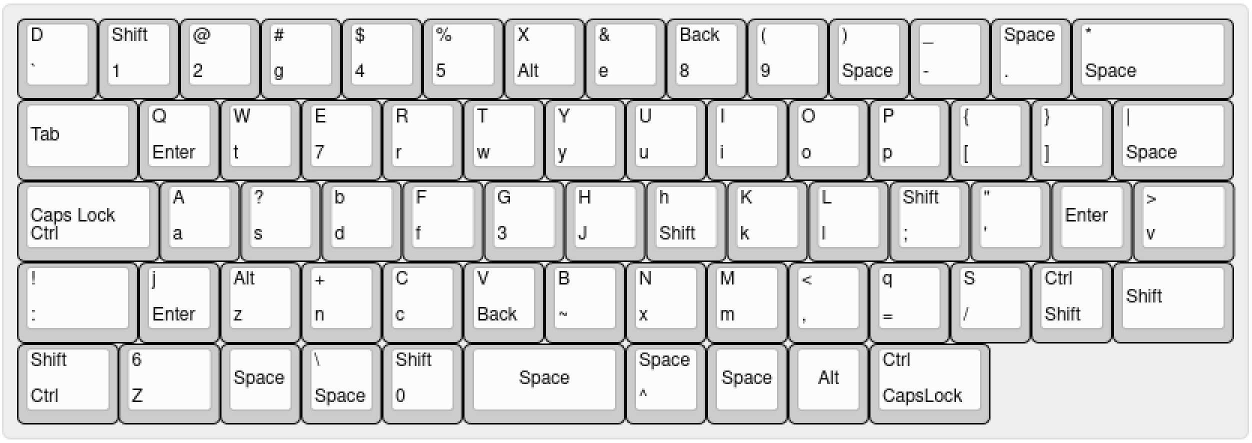
Best found layout from Policy NN

QWERTY score on first 10 programs from our dataset was **10272.6**, while our optiSmized layout which was found after 500 episodes has score equal to **7800**. It is more than **24%** improvement. It took approximately 1 hour to train our Policy NN.

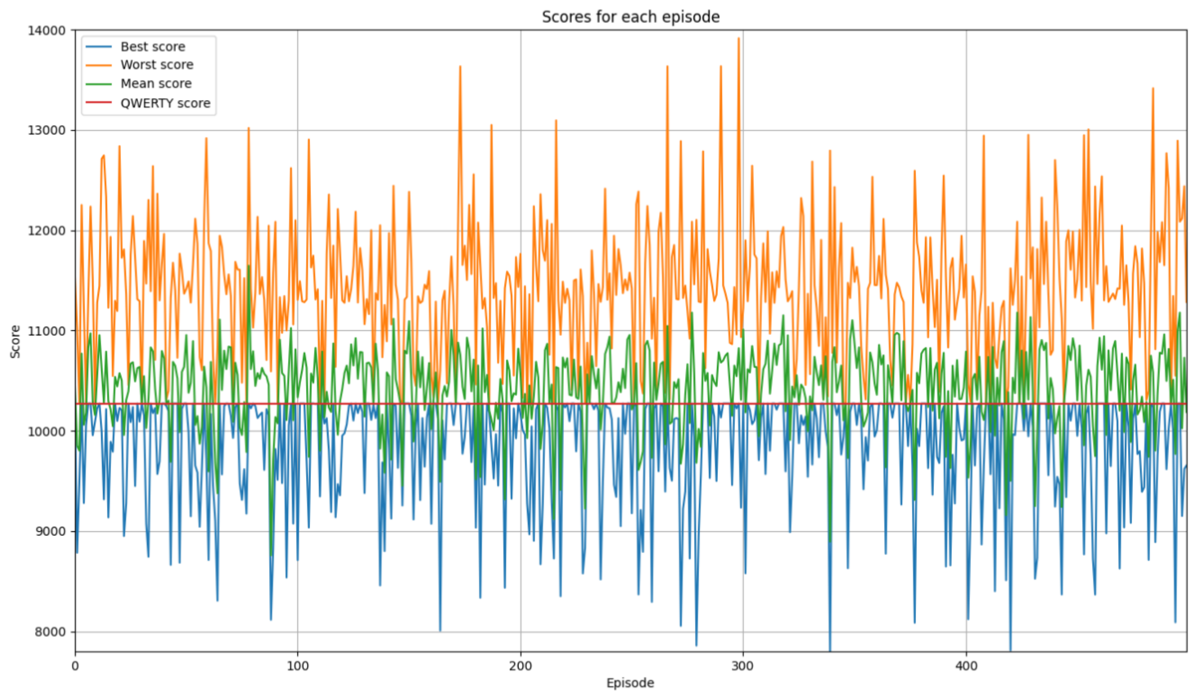
Optimized keyboard:

```
High layout:
D       <shift> @       #       $       %       X       &       <back>  (       )       _       <space> *
<tab>   Q       W       E       R       T       Y       U       I       O       P       {       }       |
<caps>  A       ?       b       F       G       H       h       K       L       <shift> "       <enter> >
!       j       <alt>   +       C       V       B       N       M       <       q       S       <ctrl>  <shift>
<shift> 6       <space> \       <shift> <space> <space> <space> <space> <alt>   <ctrl>


Low layout:
`       1       2       g       4       5       <alt>   e       8       9       <space> -       .       <space>
<tab>   <enter> t       7       r       w       y       u       i       o       p       [       ]       <space>
<ctrl>  a       s       d       f       3       J       <shift> k       l       ;       '       <enter> v
:       <enter> z       n       c       <back>  ~       x       m       ,       =       /       <shift> <shift>
<ctrl>  Z       <space> <space> 0       <space> <space> ^       <space> <alt>   <caps>
```



The following figure shows the scores of keyboard layout during training:



As you can see, after several steps best score for each layout almost always lower than QWERTY score.

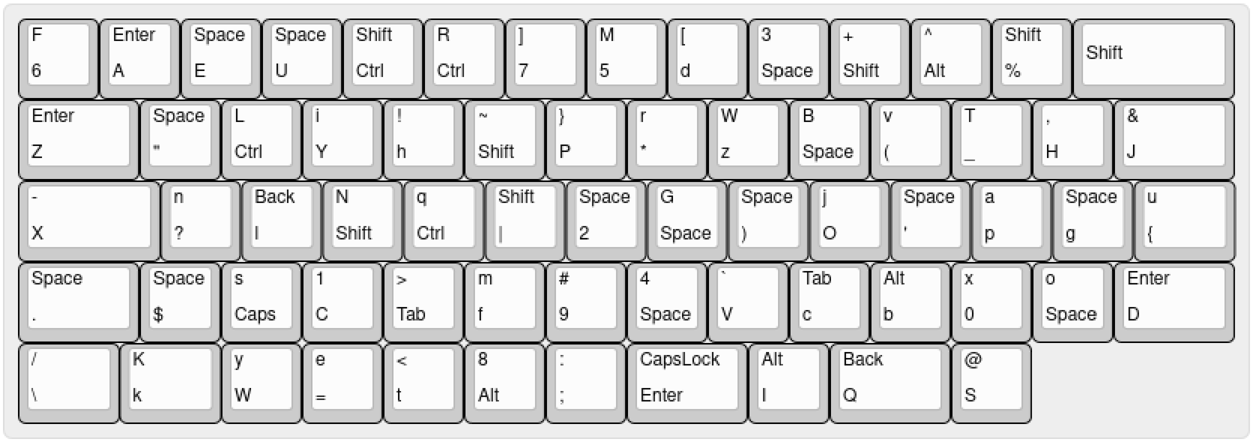
Baseline explorations

Let us now explore **baseline** solution - random generation:

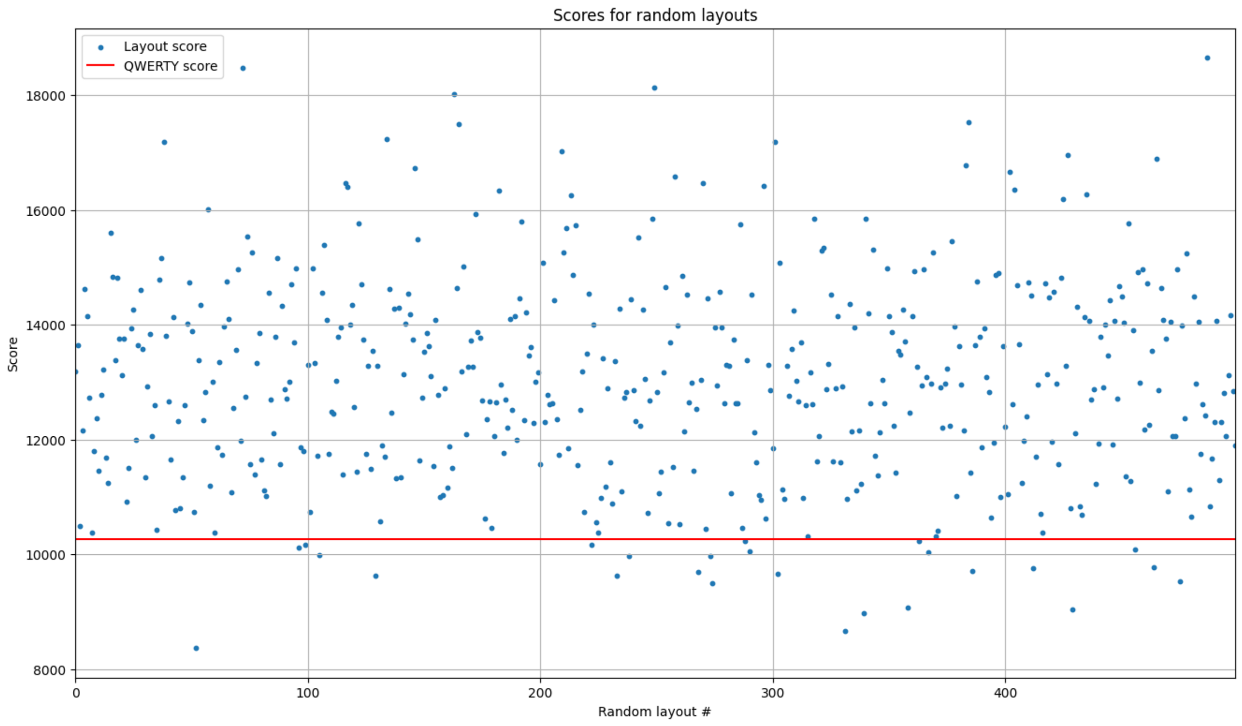
We generated 500 random layouts, best one achieved score 8369. Here it is:

```
High layout:
F       <enter> <space> <space> <shift> R       ]       M       [       3       +       ^       <shift> <shift>
<enter> <space> L       i       !       ~       }       r       w       B       v       T       ,       &
-       n       <back>  N       q       <shift> <space> G       <space> j       <space> a       <space> u
<space> <space> s       1       >       m       #       4       `       <tab>   <alt>   x       o       <enter>
/       K       y       e       <       8       :       <caps>  <alt>   <back>  @


Low layout:
6       A       E       U       <ctrl>  <ctrl>  7       5       d       <space> <shift> <alt>   %       <shift>
Z       "       <ctrl>  Y       h       <shift> P       *       z       <space> (       _       H       J
X       ?       l       <shift> <ctrl>  |       2       <space> )       O       '       p       g       {
.       $       <caps>  C       <tab>   f       9       <space> V       c       b       0       <space> D
\       k       W       =       t       <alt>   ;       <enter> I       Q       S
```



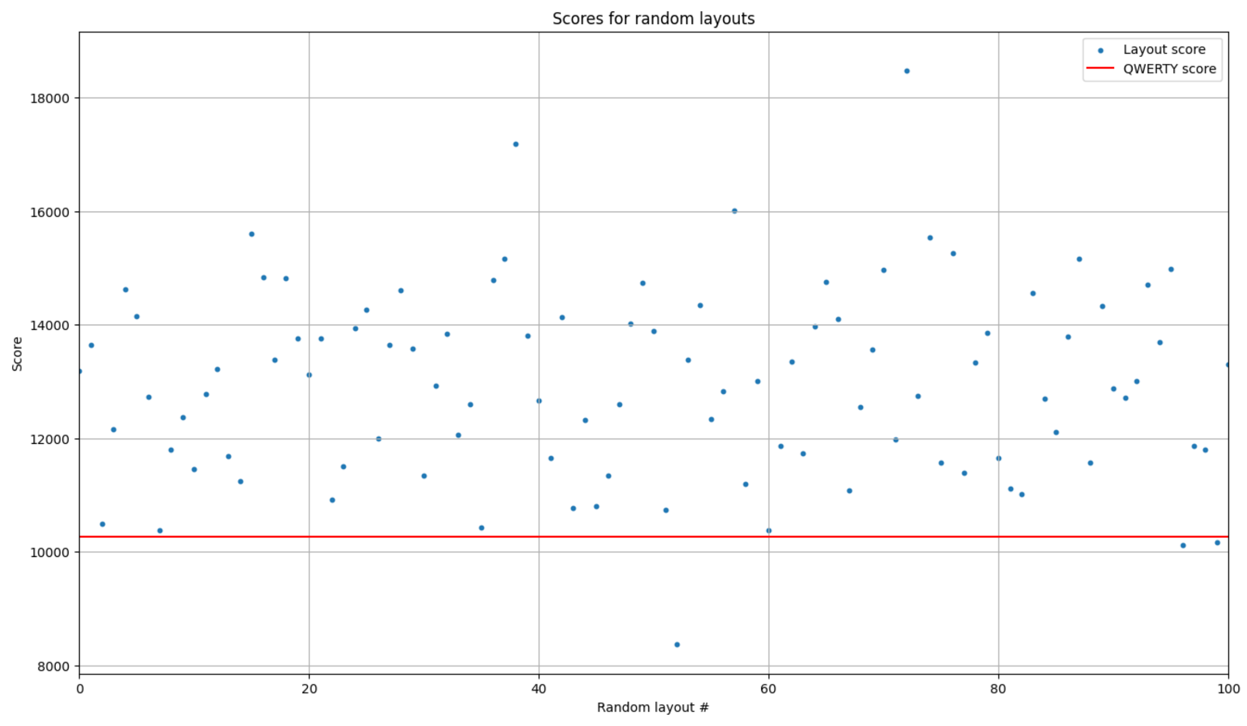
Sounds good, however, let’s take a look on results for each generated random layout:



As we can see in average score of random layout is much higher than QWERTY score, at the same time average scores from optimized keyboards were close to QWERTY.

Actually, we used different seeds for each generation for reproducibility (seed = sequence number of generation, for example, 0 for first generated layout, 1 for second, etc.). So we were lucky to face best keyboard within first 100 layouts.

Let’s look closely on first 100 randomly generated layouts:



As you can see, among first 100 layouts only 3 outperformed QWERTY.

Therefore, we can claim that our approach is significantly better than random layout generation.

Insights

- Both keyboards tends to place enter, space and shift near one of fingers positions
- For real usage both keyboards won't be useful, since numbers are located in strange order
- For different datasets optimal keyboards would be also different

Limitations

Initially we thought to pass encoded text as input for both Policy and State value NNs, however due to resource limitations we abandoned that idea.

We also wanted to train our model on whole dataset and for different programming languages, however we also couldn't afford it with our resources.

We haven't considered interactions with `CAPS LOCK`, `ctrl+c` and `ctrl+v`, `backspace` and `delete`.

Work distribution

We all worked together to solve this interesting problem. However, we can also highlight signature task of each member:

- Vsevolod Klyushev - worked on model and training process
- Dmitry Beresnev - worked with data encoding, repository structure and code quality

- Ivan Inchin - worked on result presentation and baseline construction (random generation of keyboard layouts)

References

- A. Goldie and A. Mirhoseini, “Placement Optimization with Deep Reinforcement Learning,” *arXiv.org*, Mar. 18, 2020. https://arxiv.org/abs/2003.08445
- A. Mirhoseini *et al.*, “Chip Placement with Deep Reinforcement Learning,” *arXiv:2004.10746 [cs]*, Apr. 2020, Available: https://arxiv.org/abs/2004.10746
- D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: https://doi.org/10.1038/nature16961.
- Y. Matsuo *et al.*, “Deep learning, reinforcement learning, and world models,” *Neural Networks*, Apr. 2022, doi: https://doi.org/10.1016/j.neunet.2022.03.037.
- Actor-Critic examples: https://github.com/chengxi600/RLStuff/tree/master/Actor-Critic
- Dataset: https://www.kaggle.com/datasets/joshuwamiller/software-code-dataset-cc
- Keyboard image generator http://www.keyboard-layout-editor.com/#/