HYPERPAMAMETER OPTIMIZATION WITH GENETIC ALGORITHM FOR LANGUAGE MODELING & CLASSIFYING AND GENERATING NAMES WITH HANDCRAFTED NEURAL MODELS *

Haoran Ye
2027407057
hrye@stu.suda.edu.cn

ABSTRACT

This is the first course project of *Application Practice of Deep Learning*. **For TASK 1**, I[2] train the Transformer encoder for language modeling with TORCHTEXT dataset. The submitted code can be implemented in two ways: (1) training the model with predefined hyperparameters; (2) searching for the best set of hyperparameters that enables the model to achieve the best performance. In particular, I tailor a genetic algorithm (GA) for this Hyperparameter Optimization (HPO) task. **For TASK 2 and 3**, I handcraft neural models, i.e., Base Recurrent Neural Network (RNN), Bidirectional RNN, Multi-layer RNN, Gated Recurrent Unit (GRU), Convolutional Neural Network (CNN) with the most basic modules provided by *Torch.nn* (i.e., nn.Linear, nn.Conv2d). The performance of different models are evaluated and compared on a held-out validation dataset. My code is publicly available[3].

***K*eywords** Hyperparameter Optimization · Genetic algorithm · Language modeling · Name classification · PyTorch

1 Hyperpamameter Optimization with Genetic Algorithm for Language Modeling

This section presents my project for TASK 1.

1.1 Introduction

Language modeling is to predict the upcoming words given a word sequence. It has applications such as sentence generation [1]. More importantly, it has achieved a great success serving as a surrogate task for model pretraining [2, 3].

Transformer [4] architecture has proven to be superior in many areas, e.g. Natural Language Processing (NLP) [4], Computer Vision (CV) [5], Combinatorial Optimization [6]. In particular, the attention mechanism, in place of the recurrent neural design, is more powerful and parallelizable for NLP tasks. Following the tutorial[4], I utilize the Transformer encoder for language modeling.

To maximize the model performance, tuning the hyperparameters for both the model architecture and the training setup is necessary. This process can be done either manually or automatically. Usually, a large number of hyperparameters are involved in a machine learning project. Manually tuning them is tedious in this case, and often leads to suboptimal design. Hyperparameter Optimization (HPO) [7] seeks an optimization method to automate this process. Overall,

*The first course project of *Application Practice of Deep Learning*

[2]This report uses "I" instead of "we" because all work is conducted by myself, even though "we" is universally used in technical writing.

[3]https://github.com/YeHaoran2001/APDL/tree/main/Task1_code

[4]https://pytorch.org/tutorials/beginner/transformer_tutorial.html

existing HPO methods include reinforcement learning (RL) based methods, evolutionary algorithms (EAs) and Bayesian Optimization (BO). Among them, RL-based methods are considered inferior in HPO tasks [8]. Also, the assumption of Gaussian Process made by BO seems invalid for this task setup. My task needs to search for the best among available optimizers, whose indices do not indicate their relevance. For example, Adam optimizer is indexed by 1, stochastic gradient descent (SGD) optimizer by 2, AdamW optimizer by 3. However, it does not necessarily mean that SGD optimizer is more closely related to Adam than AdamW is in any sense. As a result, I resort to EA, a powerful black-box optimization technique, for this HPO problem. Particularly, I design a genetic algorithm which is composed of a crossover operator, a mutation operator, and a selection operator.

My submitted code can be implemented in two ways: (1) training the model with predefined hyperparameters; (2) searching for the set of hyperparameters that enables the model to achieve the best performance under certain condition. By implementing it in the first manner, I manually tune the hyperparameters and obtain their corresponding results. The hyperparameters concerned are number of heads for multi-head attention, dimension of the feed forward layer, dropout rate, optimizer, activation function, and learning rate. The results are compared and discussed. By invoking the designed GA, I find the best set of hyperparameters under a given condition. The result is validated by comparing to the those obtained by manual hyperparameter tuning.

1.2 Approach

This section details the GA designed for HPO, which is presented in Algorithm 1.

Algorithm 1 GA-HPO

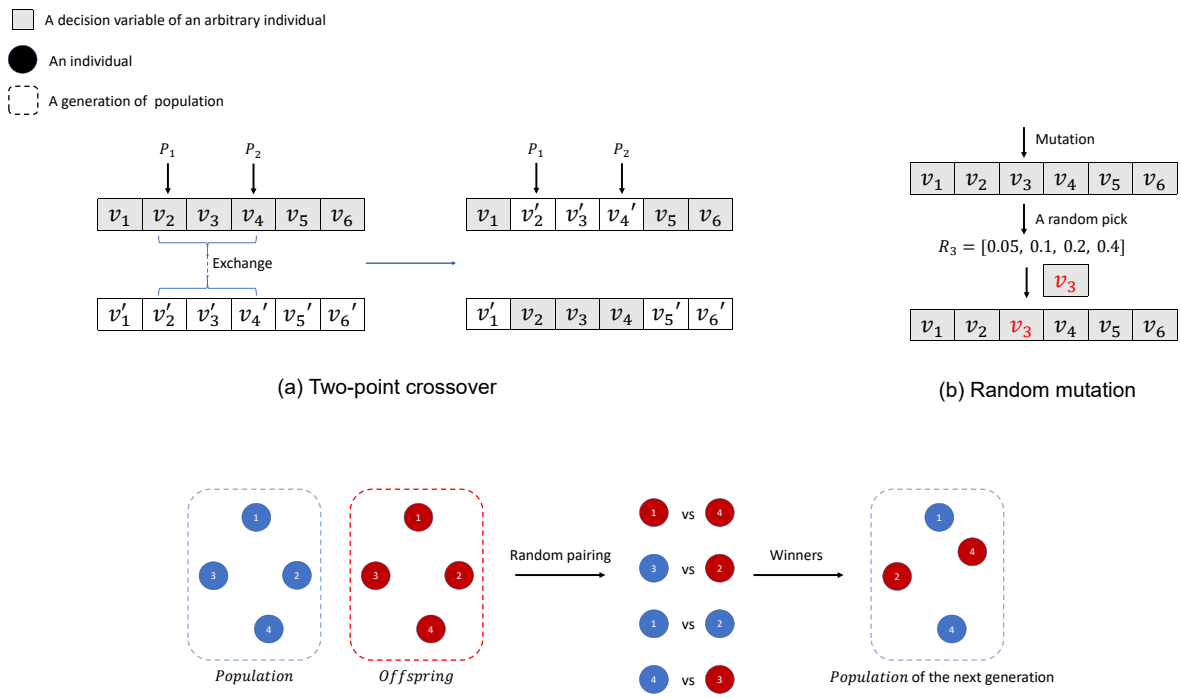
Input: Population size NP; mutation rate PM; the search space R_i for decision variable v_i, $i = 1, 2, \ldots, d$; the maximum number of generations G_m.
Output: The best set of hyperparameters $Elite$.

$\mathbf{Population} \leftarrow Initialization(NP)$
Evaluate $\mathbf{Population}$
for $iter = 1 \rightarrow G_m$ **do**
    $\mathbf{Offspring} \leftarrow Crossover(\mathbf{Population})$
    $\mathbf{Offspring} \leftarrow Mutation(\mathbf{Offspring}, PM)$
    Evaluate $\mathbf{Offspring}$
    $\mathbf{Population} \leftarrow Selection(\mathbf{Population} \cup \mathbf{Offspring})$
end for
$Elite \leftarrow PickBest(\mathbf{Population})$

First, NP individuals are randomly initialized. The encoding of an individual is shown in table 1. Then, GA evaluates all individuals. An evaluation of a individual trains the transformer encoder with the hyperparameters represented by the individual, then returns the inference performance of the well-trained model. I use the two-point crossover in my GA, which exchanges two segments of two chromosomes (the encodings of individuals). The mutation operator randomly regenerates a decision variable, which is simple but reasonable (i.e., referring to the original decision variable is not helpful; the search space is relatively small, so a random regeneration is sufficient). I utilize the tournament selection without replacement as my selection operator. It has two merits: (1) the best individual of the generation is sure to participate in and win a tournament, so that it is preserved; (2) good individuals are picked only once and the relatively poor individuals can win tournaments, so that the population diversity is preserved. The mechanism of the above operators is clearly illustrated in Fig 1. At last, the best individual is returned, representing the best set of hyperparameters for the target task.

Table 1: The encoding and the search space of GA.

Symbol	Description	Search space (R_i)
v_1	Number of heads	$[1, 2, 4, 8, 16]$
v_2	Dimension of the feed forward layer	$[64, 128, 256, 512, 1024]$
v_3	Dropout rate	$[0.05, 0.1, 0.2, 0.4]$
v_4	Activation function	relu, gelu
v_5	Optimizer	Adadelta, Adagrad, Adam, AdamW, Adamax, ASGD, NAdam, RAdam, RMSprop, Rprop, SGD
v_6	Learning rate	$[5., 1., 0.3, 0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001]$



(a) Two-point crossover

(b) Random mutation

(c) Tournament selection (without replacement)

Figure 1: An illustration of the operators in GA. (a) Two-point crossover randomly picks two points, and exchanges the chromosome segments of two mated individuals (parents). (b) Random mutation regenerates a decision variable randomly. (c) Tournament selection randomly pairs the individuals from two generations of population. Each pair competes in terms of the evaluation results (inference performance). The winners form the population of the next generation.

1.3 Code Implementation

I train the Transformer encoder for language modeling with TORCHTEXT dataset. This code can be implemented in two different ways: (1) training the model with predefined hyperparameters; (2) searching for the set of hyperparameters that enables the model to achieve the best performance.

To use the code, you need to generate the data first. To generate training & validation & test data, use:

```
python generate_data.py
```

To train the model with a specified set of hyperparameters and SGD (base model as an example):

```
python run.py --n_epochs 10 --n_head 2 --d_hid 200 --dropout 0.2 --activation relu --
    optimizer 11
```

To search for the best set of hyperparameters:

```
python run.py --search --pop_size 20 --max_iter 10
```

To test the best pretrained model (with the hyperparameters obtained by GA) on the test dataset:

```
python evaluate.py
```

1.4 Experiments

1.4.1 Experimental Setup

The experimental setup follows the tutorial, except that I reset the number of training epochs. I name the model given by the tutorial "Base model". The training loss curve of Base model is presented in Fig 2. A full convergence of the validation loss curve is observed after 10 epochs. Therefore, I set the total training epochs to 10, hoping to validate the experiments. The search space of GA is detailed in table 1. All suggested optimizers [5] are considered except for SparseAdam and LBFGS. That is because SparseAdam does not support dense gradients, and LBFGS needs another hyperparameter "closure".

Experimental environment. All experiments are executed on an AMD EPYC 7402 24-Core Processor and an NVIDIA RTX3090 Graphics Card.

Hyperparameters of GA. The population size is 20; the number of maximum iteration is set to 10; the mutation rate is 0.1. They means a run of GA conducts 200 evaluations, i.e., training the transformer encoder for 10 epochs for 200 times. As a result, one run of GA takes up about 10 hours.

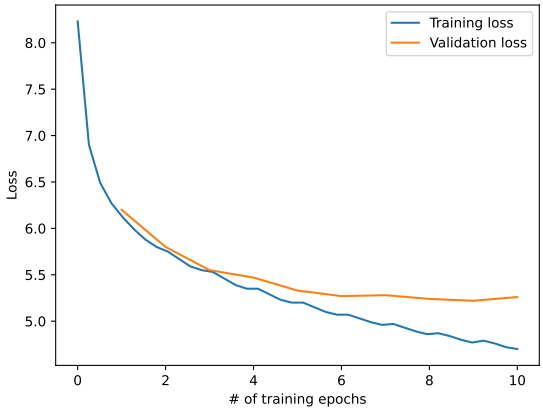


Figure 2: The convergence curves on training dataset and validation dataset: loss w.r.t. the training epochs.

1.4.2 Results and Discussions

The validation perplexities (PPL) w.r.t. different sets of hyperparameters are shown in table 2. The last line reports the best set of hyperparameters obtained by GA.

Table 2: Influence of different hyperpamameters on the model performance. The number of training epochs is fixed for all groups of hyperparameters. PPL is obtained on the validation set. The last line reports the best set of hyperparameters searched by GA.

	d_{ff}	h	P_{drop}	Activation	Optimizer	PPL(dev)
Base	200	2	0.2	relu	SGD	192.33
(A)	100					187.87
	400					176.42
(B)		1				185.34
		4				176.75
(C)			0.1			176.72
			0.4			223.58
(D)				gelu		177.11
(E)					Adam (lr=3×10^{-4})	179.83
Best (searched by GA)	1024	2	0.1	relu	RMSprop (lr=3×10^{-4})	156.24

[5] `https://pytorch.org/docs/master/optim.html`

I have to admit that the above experiments are only for practice purpose, it is far from a fair comparison due to the following reasons. (1) I set the number of training epochs fixed to 10 due to the test on Base model. For the models with more trainable parameters, there is no guarantee that they can be sufficiently trained within ten epochs. (2) The learning rate of SGD optimizer follows that provided by the tutorial, it may not be the most suitable one for ten epochs of training. (3) I only tune a subset of all hyperparamters with other ones fixed. The results maybe inapplicable once other hyperparameters change.

Despite all the above limitations, the results are validated by the following observations. (1) GA successfully delivers a set of hyperparamters that is better than any other obtained by manual tuning. (2) Models with larger capacity (e.g., larger d_{ff}) perform better, validating that the models underfit the dataset. It is reasonable considering that how large the state-of-the-art models are. (3) Models with smaller dropout rates perform better. It is reasonable because such small models should not need much regularization.

2 Classifying and Generating Names with Handcrafted Neural Models

This section presents my project for TASK 2 and 3.

2.1 Introduction

The name classification and generation are two toy tasks of "NLP From Scratch". The task of name classification takes a name as input, and classifies the name into a language from which the name originates. On the contrary, the task of name generation generates names given language labels.

All models in this project are handcrafted, i.e., designed with the most basic modules provided by *Torch.nn* (i.e., nn.Linear, nn.Conv2d). I specially hold out a validation dataset to compare the performance of my models during training, which is not implemented in the original tutorial.

Task 2 and 3, though seem two opposite tasks, are essentially quite similar. The only differences between name generation and classification are that, for generation we need to concatenate a category label to the input, and that we need to compute loss each time step. Therefore, I mainly focus on Task 2. For task 2, I handcraft RNN, bidirectional RNN, multi-layer RNN[6], GRU, and CNN. For task 3, I simply handcraft RNN and GRU as it is requested.

2.2 Neural architecture

The models based on recurrent design are so basic that I decide that there is no need to introduce them. The difficulties of handcrafting them are well handled. This section only presents the proposed CNN for name classification.

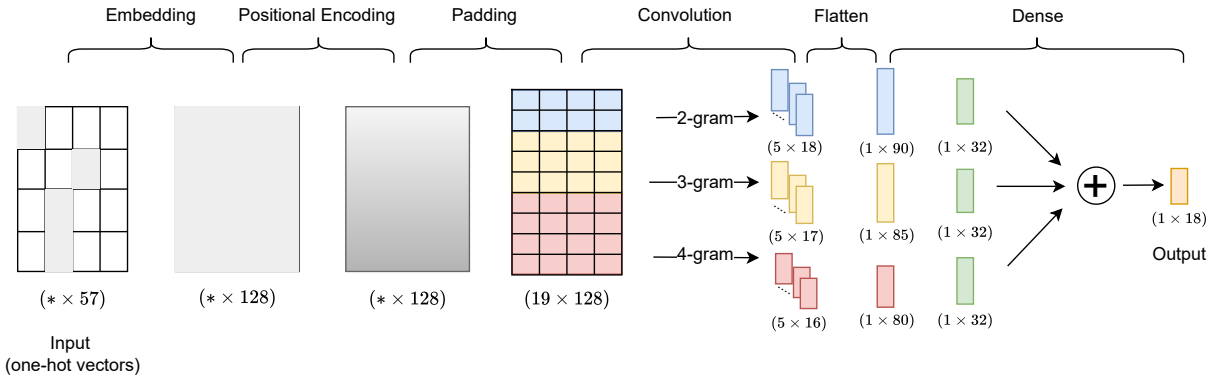


Figure 3: The neural architecture of CNN.

The architecture of the proposed CNN is illustrated in Fig 3. First, the input is passed through a character embedding layer and added a positional embedding [4]. The embedding process is followed with a zero-padding, which makes the first dimension of embedding matrix the maximum name length (19 in this task). The name embedding is then

[6]I separate bidirectional RNN and multi-layer RNN from RNN because handcrafting them are different.

passed through bi-gram, tri-gram and four-gram convolutional layers respectively, as well as dense-connected layers after convolutions. The output is finally obtained after a information fusion of different semantic channels (bi-gram channel, tri-gram channel, etc.).

2.3 Code Implementation

My submitted code can be implemented in the following ways. To train Base RNN[7]:

```
python unitrain.py --model rnn
```

To train an RNN with a different hidden dimension (e.g. 256):

```
python unitrain.py --model rnn --n_hidden 256
```

To train an RNN with multiple hidden layers (e.g. 4):

```
python mltrain.py --n_layers 4
```

To train a Bidirectional Base RNN:

```
python bitrain.py
```

To train the specially designed CNN:

```
python cnntrain.py
```

To train the specially designed CNN with positional encoding:

```
python cnntrain.py --pe
```

To train Base GRU:

```
python unitrain.py --model gru --n_hidden 128
```

2.4 Experiments

2.4.1 Experimental Setup

The experimental setup mostly follows the tutorial. However, the tutorial applies the SGD with batch size equal to one, in which case I observe unstable training. As a result, I set the batch size to 16. Accordingly, I reset the learning rate to 0.03. Also, I conduct a test to see how much training Base RNN needs to converge. The result is shown in Fig 4. I observe a trend of overfitting after training for 50k instances. It maybe unfair to train all models with 50k instances since larger models may need more training. Nevertheless, unless otherwise stated, I fix the training volume to 50k instances due to my limited computational budget.

Experimental environment. All experiments are executed on an AMD EPYC 7402 24-Core Processor and an NVIDIA RTX3090 Graphics Card.

2.4.2 Results and Discussions

The results are shown in table 3.

I discuss three observations. (1) Introducing a sequential inductive bias enhances the model performance greatly. However, the poor performance of CNN can be due to my poor design. (2) Adding a positional encoding to CNN allows it to perform better. (3) The training dataset is too small to come to useful conclusions. The models basically just overfit the training dataset, as is validated by Fig 4.

[7]The base model refers to the one with the hyperparameters suggested by the tutorial.

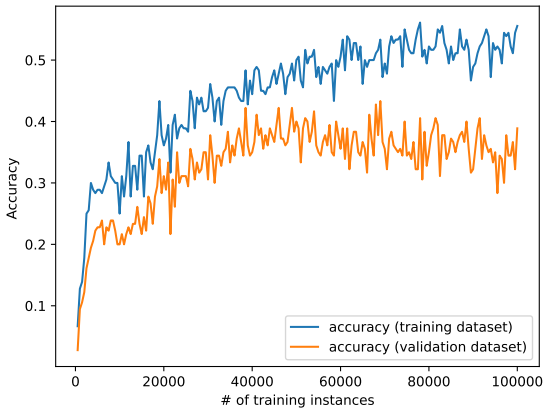


Figure 4: The convergence curves on training dataset and validation dataset: accuracy w.r.t. the number of training instances.

Table 3: The performance of different models. The number of training instances is fixed to 50k except for RNN with 3 layers, which does not fully converge until 100k instances. The accuracy is obtained on the validation set.

	n_{layer}	d_h	Bidirectional	Acc (dev)
Base	1	128	$\times$	40.00%
(A)	2			44.44%
	3			40.56% (100k)
(B)		64		44.44%
		256		45.00%
(C)			✓	42.77%
GRU				42.78%
CNN				16.67%
CNN (pe[8])				19.44%

3 Conclusion

I have sharpen my skill of tuning the models and hyperparameters, as well as English technical writing. This project has successfully fulfilled its mission.

Acknowledgements

I am grateful to Prof. Li and TA Xia for their thoughtful suggestions, helpful guidance, and hopefully, upcoming high scores.

References

[1] Lili Mou, Rui Yan, Ge Li, Lu Zhang, and Zhi Jin. Backward and forward language modeling for constrained sentence generation. *arXiv preprint arXiv:1512.06612*, 2015.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[6] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[7] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated machine learning*, pages 3–33. Springer, Cham, 2019.

[8] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*, pages 3–26. PMLR, 2021.