LTPSpace: A Tool for Machine Learning Hyperparameter Tuning

Kevin L Brockman

Department of Computer Science, University of Wisconsin-Milwaukee

Master's Capstone Project

Advised by Dr. Christine Cheng

Developed for Lab of Dr. Qingsu Cheng

May 2022

Project Overview

Less Than P Space (LTPSpace) is a Python module designed to make machine learning more accessible to a broader range of users, particularly those who may not have a strong background in computer science or access to powerful computing resources. The primary goal of LTPSpace is to simplify the optimization process for classification tasks, enabling users to fine-tune hyperparameters more efficiently, even when faced with large hyperparameter spaces that would be challenging to explore through full parameter space searches.

The complexity of a parameter space is intrinsically tied to the number of parameters involved in a given problem. As more parameters are added, the number of possible combinations increases exponentially, leading to a rapid growth in the complexity of the search space. In the case of a parameter space with n parameters, each having m possible values, the total number of combinations (or search space size) is m^n. The complexity of exploring this space is O(m^n), reflecting the exponential growth in the number of combinations as more parameters are introduced. This exponential increase in complexity can make searching the entire parameter space computationally infeasible, especially for large n or m.

This is where optimization techniques like LTPSpace come into play, enabling efficient exploration and tuning of hyperparameters in high-dimensional parameter spaces. By employing intelligent search strategies one can significantly reduce the computational burden, allowing users to find suitable parameter combinations without exhaustively evaluating every possible option. By employing a more efficient search algorithm, LTPSpace can reduce this complexity to O(nlog(m)) in the majority of cases. This does come with the trade off that we cannot guarantee the search will discover the optimal hyperparameter set, only a local optima, the use of strategies such as random resetting can alleviate this issue however while still vastly out performing full parameter sweeps when it comes to resource requirements.

LTPSpace can be applied to a wide range of classification tasks across various fields, making it a useful tool for researchers and practitioners alike. Its ease of use, efficient algorithm, and the

ability to recover from interruptions through automatic log file analysis, make it practical even for those running on consumer hardware. These features make LTPSpace especially well-suited for users with limited computational resources or those working in environments where uninterrupted computing time is not guaranteed.

By lowering the barriers to entry and simplifying the optimization process, LTPSpace aims to bring the power of machine learning to more users, helping to broaden access to advanced classification techniques and empowering researchers and practitioners to unlock new insights from their data. On a personal level, by undertaking this project I sought to both expand my knowledge of machine learning and its effective use, as well as to practice developing software for wider scale use, beyond my own personal projects, school assignments, or closed source work projects.

For evaluating this tools performance, the PatchCamelyon (PCAM) benchmark dataset was used. The PatchCamelyon is a widely-used resource for evaluating the performance of machine learning models in medical image classification tasks. It comprises approximately 327,000 labeled image patches derived from histopathologic scans of lymph node sections (only a subset of the full dataset was used for this project). Each patch has a resolution of 96x96 pixels and is associated with a binary label indicating the presence (1) or absence (0) of metastatic tissue. The PCam dataset is a balanced representation of both classes, making it an ideal candidate for assessing the efficacy of classification algorithms. Benchmark datasets, such as PatchCamelyon, play a crucial role in the development and evaluation of machine learning tools, as they provide a standardized and objective means for comparing different models and techniques. In particular, the PCam dataset offers valuable insights into the performance of LTPSpace in a real-world medical context, showcasing its potential for assisting researchers and professionals in leveraging machine learning for crucial tasks like cancer diagnosis and prognosis.

As a matter of academic honesty I will also note that, after developing the initial functional version of the tool independently, ChatGPT was then used as a coding assistant for improving and refining the code base including developing helper functions for debugging, and reviewing

code for improved efficiency, effectiveness, and readability. It was also used as a research tool for selecting python modules called used in this tool and learning about their proper use.

User Description

In order to use LTPSpace a user must provide labelled training data, a modelling function, and a parameter set. Training data can be of any format compatible with the tensorflow model generated by the modelling function. The modelling function should take inputs of a parameter value list and an input shape (both of these will be generated by LTPSpace and do not need to be provided directly the user, only defined). The function will construct a tensorflow model using the values at given parameter list indexes as inputs for the hyperparameters that LTPSpace will tune. Finally the user must define each parameter to be used using a helper function to establish the potential range of values, resolution, and other optional characteristics. The tool has a variety of other optional settings which will not be exhaustively detailed here.

```python
# Example model for image classification
def samplemodel(params, input_shape):
    """Model architecture is based on the VGG16 model but has fewer layers and fewer filters per layer to reduce the model complexity.
    The VGG16 architecture is known for its good performance in image classification tasks, and it's a good starting point for many
    classification problems."""
    m = Sequential()

    # First block of convolutional layers
    # Each Conv2D layer performs 2D convolution, which is used for spatial feature extraction.
    # The activation function 'relu' (Rectified Linear Unit) introduces non-linearity and helps to learn complex patterns.
    m.add(Conv2D(params[0], (3, 3), activation='relu', padding='same', input_shape=input_shape))
    # MaxPooling2D layer reduces the spatial dimensions and helps to minimize overfitting by reducing the number of parameters.
    m.add(MaxPooling2D(pool_size=(2, 2)))

    # Flatten layer is used to convert the 2D feature maps into a 1D vector, which can be used as input to the fully connected layers.
    m.add(Flatten())

    # Fully connected (Dense) layer for classification
    # The first Dense layer with 512 units and 'relu' activation function adds more complexity to the model.
    m.add(Dense(params[1], activation='relu'))

    # Dropout layer helps to prevent overfitting by randomly setting a fraction of input units to 0 during training.
    m.add(Dropout(params[2]))

    # The final Dense layer with 1 unit and 'sigmoid' activation function outputs the probability of the image belonging to the positive class.
    m.add(Dense(2, activation='sigmoid'))

    return m
```

Figure 1 – An example modelling function for binary image classification

```
optimizer = Explorer(samplemodel,xdata,ydata,3,debuglevel = logging.DEBUG,log_directory = "Logfiles")
optimizer.modify_Model(kfolds = 10, epochs = 10, batch_size = 32)

# Set up parameters
optimizer.addParam(1, 256, "Conv2DSize")
optimizer.addParam(512, 1024, "Dense1Size")
optimizer.addParam(0.1, 0.9, "Dropout1", False, 0.1)

# Run CNN model test
optparams, score = optimizer.run(randomstart = True, recover = True)
```

Figure 2 – An example of setting up and running the LTPSpace parameter space Explorer

Once initialized, LTPSpace will begin optimizing the model, after each step of the process it will log its current progress to an external file. Should the program be interrupted for any reason, it can be set to recover from a log file to resume from the beginning of the interrupted step. Optimization is performed by rating the results of each parameter set using an evaluation function on the results of testing it. The evaluation function defaults to grading based on accuracy score, but this can be overwritten by a user provided function. Once the parameter space search is complete, a learning curve will be generated for the final model and the results logged along with a graph of the curve. The optimizer will then return the final parameter set and rating.

A typical use of LTPSpace would start with the preparation described previously. To start one can begin with a broad parameter set and an amount of sample data that can complete in a reasonable amount of time (exact sample size will depend on the complexity of each sample, model complexity as a whole, and other factors specific to a given task, more is always better but will take longer to run.) A setting can be used to run the optimization process with random starting conditions, over multiple iterations this can allow the program to locate multiple local optima and select the best among them (possibly even reaching the global optima.)

After a few iterations the data should be reviewed and if possible parameters should be pruned to improve the speed of the speed of future iterations (as noted in the introduction, run time is directly proportional to the number of parameters under consideration.) Pruning can be full (removing a parameter entirely and leaving it as a fixed value in the model) or partial (reducing the range of potential values to search for a given parameter).One example of how to do this is to note if a given parameter generally converges to the same value every time, it can be set

permanently to that value in the model. If a parameter generally ends up within a certain range of values, its range can be limited to that range reducing the resources needed to explored it. Certain parameters may also trend towards extremes, the response to this may vary depending on the model architecture and what the parameter is controlling. A layer being minimized in impact every time to effectively not existing may indicate that the layer should be removed from the model entirely. A parameter tending towards its max or min may indicate that the range needs to be extended as better performance is possible outside its current limitations. Once the model is simplified, the amount of training data in use can be increased to utilize freed computing resources. The process can be repeated until the user is satisfied or a clear stopping point is found where further improvement is unlikely.

Performance can also be improved before beginning by using domain knowledge for the particular problem type and architecture being worked on, a surplus of information is available online which can provide approximate starting points to focus the parameter space search around. The time taken for this research can generally be expected to be far less than the time it will save by pre-pruning the search space.

Functional Description

The LTPSpace algorithm is designed to optimize the hyperparameters of a machine learning model, particularly neural networks, through use of a hill-climbing strategy, the algorithm iteratively adjusts one hyperparameter at a time, seeking to improve model performance as measured by a specific evaluation metric. By completion, the algorithm will locate a local optima within the parameter space being explored.

To achieve this, the algorithm begins with an initial set of hyperparameters, which are defined with a potential range of allowed values and a resolution. By combining these values the algorithm determines the total number of potential states that the parameter may reach. From the base 2 logarithm of this range of potential states, we can then determine the total number of search stages that are required to fully explore the range of this space via a logarithmically decaying learning rate. At the starting stage for a given parameter, it has a learning rate equal to half its total range, this means that during that stage it will test performance at current value

+ ½ range, and current value – ½ range. The next stage will do the same with ¼ the range, this continues until the final stage (0) in which the exploration step is equal to the parameter's resolution, the smallest possible step in the parameter space.

The optimization process is split into two major steps, search and fine-tuning. Search occurs fully in all cases to explore the parameter space working across the full range of each parameter using the learning rates described previously, this step's complexity is O(n*log(m)) where n is the number of parameters and m is the range of values for the parameter, this can also be understood as the number of parameters multiplied by the number of search stages. Certain conditions may allow us to skip exploring particular values meaning performance will sometimes be better but it will never be worse than this limit.

<u>Pseudo code for the search step:</u>

- While stage is greater than or equal to 0
    - While the stage is not completed:
        - For each hyperparameter:
            - Adjust one hyperparameter value based on the current learning rate
            - Evaluate the model performance with the new set of hyperparameters.
            - If the new performance score is better than the current best score:
                - Update the best set of hyperparameters and the associated performance score.
                - Move on to the next hyperparameter.
            - If the new performance score is not better:
                - Move on to the next hyperparameter if we have already tested upward and downward movement for this one, otherwise test the untried direction
        - Decrement the stage by 1

In a theoretical case where the utility of each parameter is fully independent of the others, the search step alone would be sufficient to locate a local optimum. In practice, the parameters will be interacting with one another to determine the overall effectiveness of the model, this means that further exploration may be required to reach our optima.

The fine-tuning step begins at the final stage (stage 0) of the search step. If any improvement is found the parameter set will shift, possibly opening up new improvements from other parameters, in this case when we complete stage 0, instead of finishing the search, we start the stage over again with our new parameter set. This continues until a full iteration is completed without any improvement being found. In most cases (almost all during testing), this will not occur at all or will only happen once or twice, minimally impacting the overall performance of the program. In the worst case, this could result in exploring close to the entire parameter space (the very thing we are trying to avoid with this tool) for a complexity of $O(n^m)$. To account for this, the user can set early stopping conditions, limiting the total number of fine-tuning iterations, or the total run time of the program.

Once fine-tuning is complete, a final evaluation of the model is performed. This is done with a learning curve which provides scoring for 3 different data sets to test overall performance and detect overfitting. Each step of the learning curve is tested with the model training data (each step of the learning curve increases the size of this set), the test data (the remainder from the training data, it is not used to train this particular instance of the model, but was used for parameter tuning), and validation data (a holdout set separated from the other data before the search process began, this data has not been used at all for tuning the parameters providing the most objective evaluation metric.)

After completing the final evaluation, the full learning curve data is logged, and the final parameter set and score are returned to the calling program.

PCAM Trial

Setup:

In order to test the efficacy of LTPSpace, I set out to use it to build a classifier to differentiate lymph node images containing or not containing metastatic tissue. The input is a 96x96 pixel color images and the output is a binary classification indicating the presence or absence of metastatic tissue. The initial model architecture was a simplified version of VGG16 (pictured below), a well-documented image classification model. If the goal of the trial was purely image classification quality a pretrained model could have been used as a base with modification, but the focus on this case was evaluating LTPSpace hyperparameter tuning so models used in the trial were trained from scratch.

Layer (Index)	Layer Type	Filters/Units	Kernel Size	Activation	Padding	Pool Size	Dropout Rate
0	Conv2D	params[0]	(3, 3)	relu	same	N/A	N/A
0	Conv2D	params[0]	(3, 3)	relu	same	N/A	N/A
1	MaxPooling2D	N/A	N/A	N/A	N/A	(2, 2)	N/A
2	Conv2D	params[1]	(3, 3)	relu	same	N/A	N/A
2	Conv2D	params[1]	(3, 3)	relu	same	N/A	N/A
3	MaxPooling2D	N/A	N/A	N/A	N/A	(2, 2)	N/A
4	Flatten	N/A	N/A	N/A	N/A	N/A	N/A
5	Dense	params[2]	N/A	relu	N/A	N/A	N/A
6	Dropout	N/A	N/A	N/A	N/A	N/A	params[3]
7	Dense	2	N/A	sigmoid	N/A	N/A	N/A

Figure 3 – Initial model architecture used in trial, param[x] indicates a value that will be tuned by the LTPSpace tool

Procedure:

Using random restarts, the LTPSpace tool was run repeatedly on a given architecture to observe results, with adjustments being made to the architecture and optimization process based on user analysis. The initial architecture showed reasonable performance (70-80% classification accuracy in most trials), but interpretation was somewhat confounded when plotting learning curves to check for overfitting. Learning curves evaluated performance across a range of training data sample size and were evaluated on 3 data sets: Train (the same data used in the training), Test (the leftover samples not used in this particular training run, but that were used

in the previous parameter optimization of the model) and Validation (a holdout set that has never been used to make decisions on parameter tuning or to train the model). Most runs did not have smooth learning curves instead having very notable drops in performance seemingly randomly at different training sample sizes (generally it would be expected that performance would increase with increased training data until overfitting occurs.) This effect was found to be much reduced at large sample sizes which reduced the random variation in evaluation. Figure 4 compares two learning curves using the same model architecture and parameter values, but with different sample ranges used. With a 100-fold increase in sample size the variation in results is reduced to about a third if that seen in the smaller sample size, at the cost of significantly increased processing time. The smaller sample learning curve took only a few minutes to produce while the larger sample took roughly 10 hours. Based on this, throughout the process preliminary analysis was performed using small sample sizes with top performers repeated in focused high sample size trials for more accurate comparison.

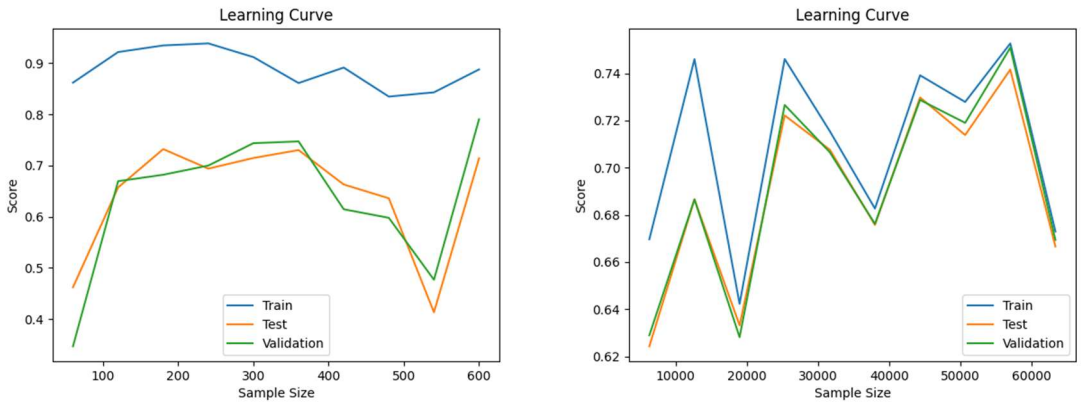

Figure 4 – Comparison of learning curves with 100-fold different in sample sizes

Despite these difficulties, some results were apparent after a few initial trials at low sample sizes. The clearest result was that in almost all cases, the dropout layer would be tuned to a value of 0.7 or 0.8 (in a range of 0.1 to 0.9 with a 0.1 resolution.) Based on these results, this was removed as a parameter and set permanently to 0.7. It was also consistent that the Test and Validation results closely tracked one another's results while the Train data generally outperformed both. This indicates that involvement in earlier parameter tuning did not result is

any notable overfitting so long as the data was not used in training the specific instance of the model being tested.

A less straightforward result was that the first parameter was consistently being tuned to very low values, including down to a single neuron in one trial. This was taken as an indication that the architecture were more complex than was needed and multi trials were taken with reduced total layer count, first removing 1 conv2d layer from each of the convolution and pooling sets, there was no notable loss in accuracy from this change. In a further trial, one of the convolution and pooling sets was removed entirely, and the remaining one had its potential range of values doubled. This not only led to a general decrease in performance, but also resulted in significantly increased search times. While the higher range of values was only selected in one trial, every training attempt to check this range took much longer than other trials due to the following dense layer connecting to every node in the previous layer which results in quadratic complexity scaling. Some improvements were however discovered from this exploration, in no trial was the dense layer ever tuned anywhere lower than the top 75% of its potential values, in order to speed up future parameter turning values lower than this cutoff were removed from consideration.

Four separate model architectures were trialed with the best performer ultimately being the second one (full details of all trials are available in the git repo linked at the end of the document), as previously detailed this model removed one convolutional layer from each set of convolution and pooling layers. The removal of the dropout factor as a reduced the number of search steps for LTPSpace by a factor of 3. The reduction in range of potential dense layer values resulted in a further 20% drop in required search steps. With these performance improvements in place, further trials could be run with a higher sample size for more accurate results.

Layer (Index)	Layer Type	Filters/Units	Kernel Size	Activation	Padding	Pool Size	Dropout Rate
0	Conv2D	params[0]	(3, 3)	relu	same	N/A	N/A
1	MaxPooling2D	N/A	N/A	N/A	N/A	(2, 2)	N/A
2	Conv2D	params[1]	(3, 3)	relu	same	N/A	N/A
3	MaxPooling2D	N/A	N/A	N/A	N/A	(2, 2)	N/A
4	Flatten	N/A	N/A	N/A	N/A	N/A	N/A
5	Dense	params[2]	N/A	relu	N/A	N/A	N/A
6	Dropout	N/A	N/A	N/A	N/A	N/A	0.7
7	Dense	2	N/A	sigmoid	N/A	N/A	N/A

Figure 5 – Final model architecture

Results:

Once the final model architecture was determined, LTPSpace was run to optimize the parameters using a few thousand data samples to complete the full sweep within a reasonable time. Once the final parameter set was determined, the learning curve section was rerun using a much larger data set to get a more robust idea of the final model performance, shown in Figure 6.

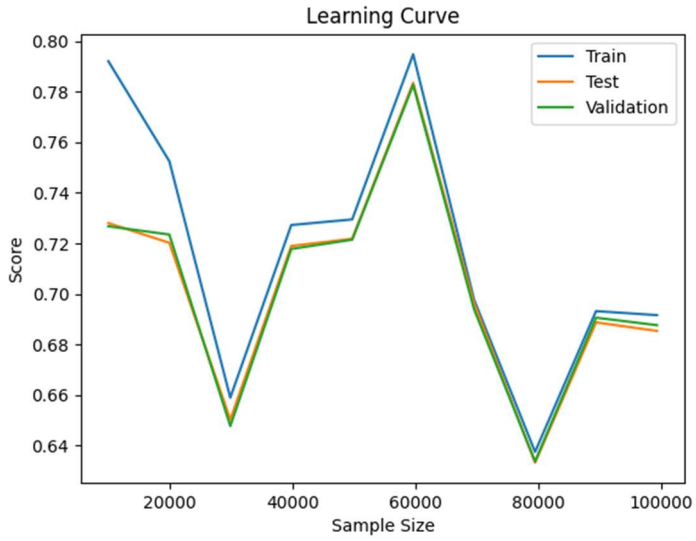

Figure 6 – Learning curve of final model parameters

Comparing the examples in figures 4 and 6, it is apparent that the sample size increase both improves prediction quality and reduces the overall variance in prediction quality across trials. Compared to the first trials, the final trial had an improvement in average accuracy of ~6.5%

with the final model averaging to almost exactly 70% accuracy. This falls short of models developed by researchers, some nearing 100% accuracy, but using significant preprocessing and much more complex network architectures.

In an attempt to better understand the variation between trials, I took a closer look at the data to see if I could find any potential causes. Distribution of samples between true or false classification was consistent across trials, removing that as a possibility, though overall the model had notable better recall than precision. Ultimately I was unable to determine any deciding factor and will work under the assumption that it is a matter of more or less difficult samples occurring in given trials.

Conclusion

In conclusion, the LTPSpace Python module has been developed to enhance the accessibility of machine learning, specifically for users with limited computer science backgrounds and computational resources. The module simplifies the optimization process for classification tasks by efficiently exploring and tuning hyperparameters in high-dimensional parameter spaces, leading to a search complexity reduction to O(nlog(m)) in the majority of cases.

LTPSpace demonstrates versatility in classification tasks and offers ease of use and the ability to recover from interruptions, making it a practical tool for a diverse range of users. The tool's performance has been evaluated using the PatchCamelyon (PCam) benchmark dataset, showcasing its potential for real-world medical applications, such as cancer diagnosis and prognosis. The application of LTPSpace has led to notable performance improvements in image classification tasks, making it a resource for users new to the field or with limited computational resources.

Despite these successes, there is still room for improvement. The program's speed could be significantly enhanced by leveraging GPU capabilities rather than relying solely on CPU computations; however, the complexity of setting up the necessary support posed a challenge during the project's timeline. Additionally, incorporating alternative scoring methods, such as F1 Score or ROC AUC, could provide a more comprehensive evaluation of model quality, guiding training more effectively. Expanding debug information can further assist users in

understanding their models' performance, ultimately contributing to the ongoing development and refinement of the LTPSpace tool.

References

1. Winkelmaier, Garrett, et al. “Quantum Cascade Laser Infrared Microscopy Differentiates Malignant Phenotypes in Breast Histology Sections.” 2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018), 2018, https://doi.org/10.1109/isbi.2018.8363700.
2. Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).
3. Kothari, Ragini, Yuman Fong, and Michael C. Storrie-Lombardi. "Review of laser Raman spectroscopy for surgical breast cancer detection: stochastic backpropagation neural networks." Sensors 20.21 (2020): 6260.
4. Zhang, Zizhao, et al. "Pathologist-level interpretable whole-slide cancer diagnosis with deep learning." Nature Machine Intelligence 1.5 (2019): 236-245.
5. Deep Network designer. VGG-16 convolutional neural network - MATLAB. (n.d.). https://www.mathworks.com/help/deeplearning/ref/vgg16.html
6. PCAM Dataset: https://github.com/basveeling/pcam
7. See Full Project: https://github.com/LBrockmanK/LTPSpace