Gruppe 4

Introduktion til
intelligente systemer
02461

2024
19/01

Authors:
Glymov, Jonas Lolk : s234845
Lorentzen, Alexander Baatz:
s234815

Lektor:
Mikkel N. Schmidt

Optimization of hyperparameters

On a custom TinyVGG model using random- and grid search

DTU

Contents

Section accountable identification

s234845 = J, s234815 = A

11982 out of 12000 characters.

All code and assets used is publically available here.

Abstract

This study delves into hyperparameter tuning of a custom TinyVGG model, featuring batch normalization and dropout. To find optimal hyperparameters, grid- and random search with K-fold cross-validation was used. Results showed that combining random search with data augmentation produced the highest accuracy of 85%, and exposed significant differences between the accuracy of grid- and random search, due to limited trainable parameters.

1 Introduction (J)

Contrary to MLPs, Convolutional Neural Networks (CNNs) are exceptionally good at learning patterns in images. Using convolutional layers, CNN processes characteristic patterns within pictures while simultaneously compressing them into smaller parts.[1]

1.1 Research Question

The effectiveness of a neural network mainly depends on the model architecture, data and hyperparameters. The latter being the main objective of this study. Hyperparameters, once set, typically remain unchanged after a model has entered training mode. The performance metrics are heavily influenced by these parameters[2]. Due to the infinite number of potential combinations, algorithms are commonly used to determine the most optimal parameters for a given model[2].

This study aims to show how hyperparameter optimization can be accomplished through search algorithms, with the objective of improving the accuracy of a modified TinyVGG architecture.

2 Methods

2.1 Dataset (A)

The choice hinged on a few main components, demanding adequate size and quality to prevent a potential confounder later in the study. Animals-10[3], addressed these criteria, with more than 26,000 images across 10 classes.

However, time constraints and lack of computational resources forced us to reevaluate. By the use of a confusion matrix, we gauged class performance by misclassification rates. Furthermore, every second class was systematically removed, yielding a truncated dataset of 13,656 images. By random statistical fairness, each class was halved, resulting in a final dataset of 6,827 images.

This ensures that the reduced dataset maintains a degree of representativeness comparable to the original dataset.

2.2 Model Architectures (A)

Influenced by TinyVGG's minimalist design[1], our architecture mirrors its appearance, prioritizing computational efficiency and constraining trainable

Figure 1: Image samples of classes in the dataset

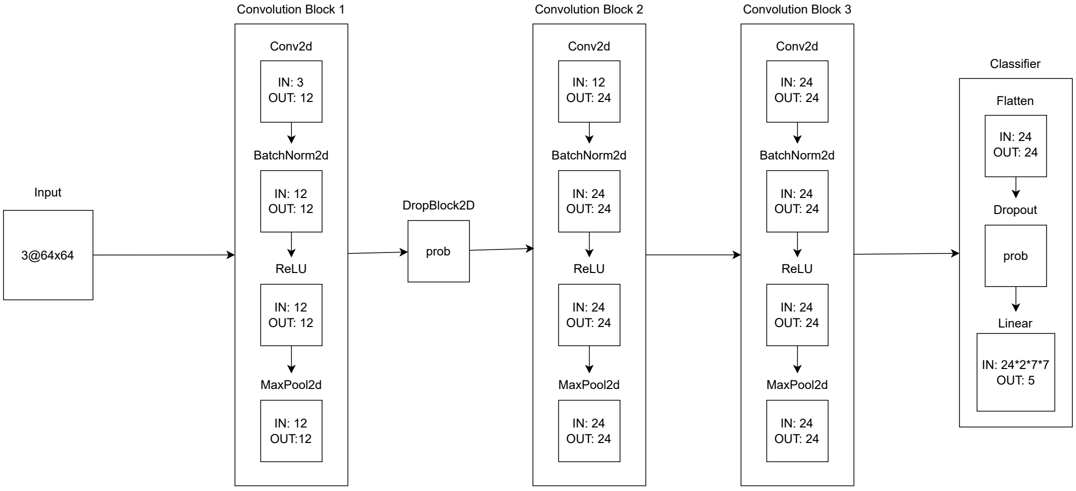

Figure 2: The architecture of the slightly modified TinyVGG model

parameters to under 15,000 for 5 classes. We expanded to 3 convolutional blocks, each with a single convolutional layer, addressing overfitting concerns by integrating batch normalization layers after each convolutional layer[4]. This improves regularization and allows a wider range of learning rates[4]. To enhance regularization further, we introduced two dropout techs. Conventional dropout[1] is applied before the fully connected layer in the classifier. Unscheduled DropBlock, inspired by AmoebaNet[5], is implemented between the initial convolutional blocks. For multi-class classification, our model uses the Cross-Entropy loss function, akin to the default TinyVGG model.

2.3 Model Selection (A)

To establish a few baseline hyperparameter-values, we drew inspiration from [2], and conducted a few experiments of our own. The following parameters were chosen:

- Batch size: 32

[1] Deactivates neurons randomly with probability $p \in [0.0, 0.1]$[2]

- Optimizer: RMSprop
- Learning rate: 0.001
- Dropout rate: 0.3

Afterwards, the two models were trained for 50 epochs. To better evaluate their performance, we utilized `seaborn`[6] to visualize their variability over 5 distinct seeds, as depicted in Figure 3.

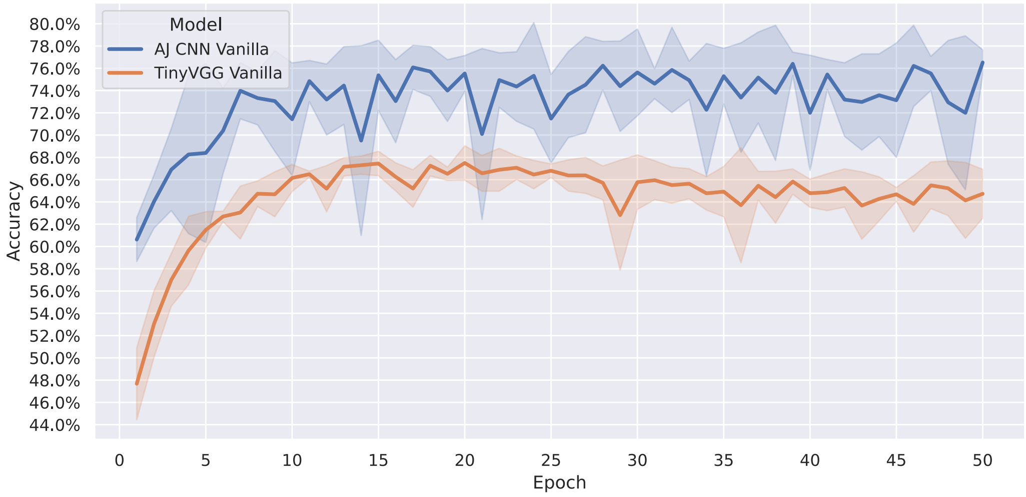

Figure 3: TinyVGG vs. our model, shaded areas being SD

It's important to note that although our model has fewer convolutional blocks overall, it includes 12 hidden units in the first block and twice that number in subsequent convolutional blocks. This diverges from TinyVGG, with 10 hidden units in both the first and second blocks. However, our model maintains superior computational efficiency at 5.47 MACs (MB) compared to TinyVGG's 5.95 MACs (MB).[2]

2.4 Preprocessing: Data Augmentation (A)

Hoping to minimize overfitting, we utilize data augmentation. Given that the dataset is mostly comprised of "everyday" images of animals, we strove to use simple transform compositions to better simulate human error during photography. Consequently, we chose the following augmentations:

[2] MAC = multiply-accumulate operations, a metric for measuring model complexity and efficiency.

- `RandomCrop`, with reflection padding on all four borders.
- `RandomHorizontalFlip`, with a 50% probability.

Each augmentation process initiates with a resize (64x64) and finalizes with a normalization transform, moderating the bias towards specific color channels. This allows for artificial diversification and extension of the dataset during the training phase. Putting it to the test, it's clear to see just how effective data augmentation is at overfit-mitigation, as evident in Figure 4.

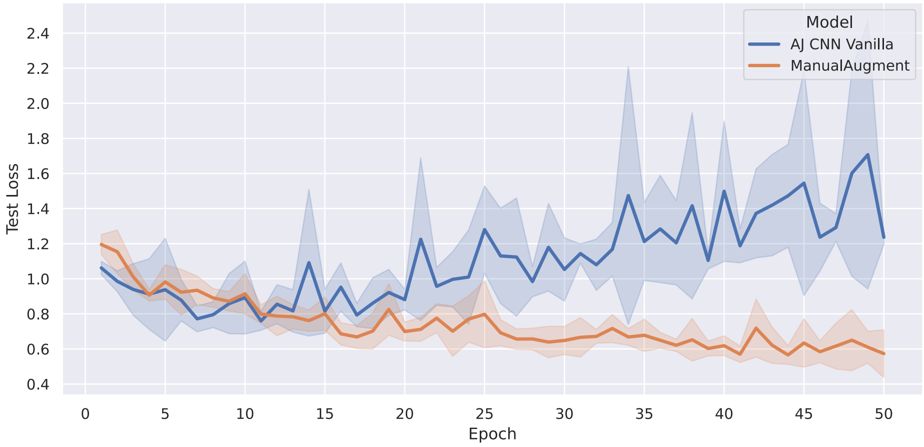

Figure 4: Mean test loss over 50 epochs, across 5 seeds

The reduction in test loss leads to much better generalization potential, which in turn improves test accuracy - refer to Figure 5.

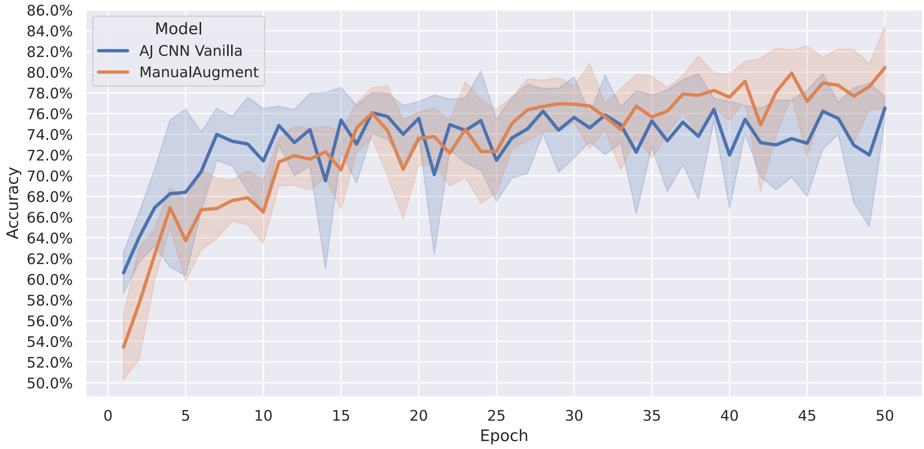

Figure 5: Mean test accuracy over 50 epochs, across 5 seeds

2.5 K-fold Cross Validation (J)

Instead of performing an 80/20 split between training and validation sets, we adopt a technique known as cross-validation, k-folds. It involves dividing the data into k folds, each fold representing a subset of the data. The training process iterates over $k - 1$ folds, where the last k fold is used for validation. In total, this is done k times, with the folds being shifted each time. The final model estimate is then the mean of all k-fold accuracies. By shuffling the data, this method further lessens the risk of overfitting, preventing the model from learning unwanted patterns.[7] An illustration of our implementation:

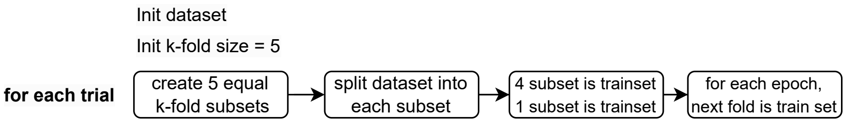
Init dataset
Init k-fold size = 5
for each trial
create 5 equal k-fold subsets
split dataset into each subset
4 subset is trainset 1 subset is trainset
for each epoch, next fold is train set

Figure 6: The implementation of the k-fold algorithm [7]

2.6 Hyperparameter Optimization using Grid- and Random Search (J)

The random search algorithm works by defining a parameter space of potential hyperparameters, the so-called search-space. It then randomly samples within the specified range and records the accuracy[8]. Conversely, the grid search algorithm defines a grid of constant values, iterating over all possible combinations parameter grid to determine the accuracy of each combination.[8] Grid and random search are implemented without the use of preexisting Python modules. To achieve this, we design an objective function with the purpose of creating a CNN model and conducting training and testing with the proposed hyperparameters. Finally, we sort for the best test accuracy and its corresponding hyperparameters.

2.7 Experimental Setup (J)

The experimental phase is divided into 2 segments. First, the search for optimal hyperparameters using grid- and random search, with and without data augmentation. We then proceed to train and validate each model to assess the effectiveness of the different parameters.

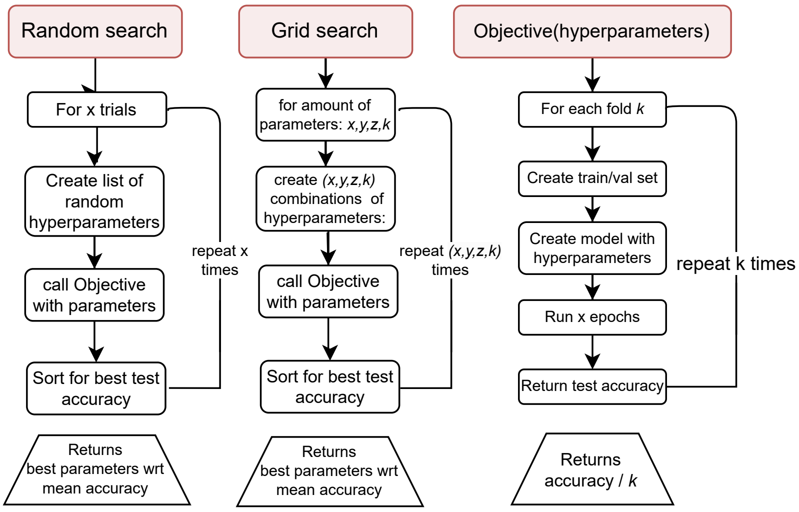
Random search
For x trials
Create list of random hyperparameters
call Objective with parameters
Sort for best test accuracy
repeat x times
Returns best parameters wrt mean accuracy
Grid search
for amount of parameters: x,y,z,k
create (x,y,z,k) combinations of hyperparameters:
call Objective with parameters
Sort for best test accuracy
repeat (x,y,z,k) times
Returns best parameters wrt mean accuracy
Objective(hyperparameters)
For each fold k
Create train/val set
Create model with hyperparameters
Run x epochs
Return test accuracy
repeat k times
Returns accuracy / k

Figure 7: The flow tree of the search algorithms [7]

1. **Optimization:**

With random search, we anticipate to optimize the following hyperparameters within specified ranges, over 100 trials:

- Learning rate[3]: $[0.1; 0.00001]$
- Batch size: $\{16, 32, 64\}$
- Dropout rate: $[0.1; 0.5]$
- Epochs: $[15; 30]$
- Optimizer: Adam, SGD, RMSprop

With grid search, on the other hand, we expect to optimize the following hyperparameters:

- Learning rate: $\{0.01, 0.001, 0.001\}$
- Batch size: $\{16, 32, 64\}$
- Dropout rate: $\{0.3, 0.4, 0.5\}$
- Epochs: 20
- Optimizer: Adam, SGD, RMSprop

[3] A logarithmic distribution is applied to the range of learning rate values to favor lower values[9]; the sci-kit stats package is used for the reciprocal distribution function

This entails iterating through $3 \cdot 3 \cdot 3 \cdot 1 \cdot 3 = 84$ combinations.

Computational costs: Using k-fold, each iteration is performed $k = 5$ times. Thus, random search involves:

$$100_{\text{trials}} \cdot 5_{\text{folds}} \cdot (15 \leq \text{epochs} \leq 30) = 500 \cdot (15 \leq x \leq 30)$$

computations, while grid search requires:

$$81_{\text{combinations}} \cdot 5_{\text{folds}} \cdot 20_{\text{epochs}} = 8100.$$

To optimize computational resources, a median early stop algorithm is implemented. It computes the average test loss of the last 5 epochs of each trial and compares it against a set threshold. If the average is above said threshold the trial is pruned.

2. Training and Validation:

Following the search for optimal hyperparameters, we compare the performance of the following setups using the most effective hyperparameters from the optimization phase. The setups are evaluated based on test accuracy using the Animals-10 dataset, and a regular 80/20 train/validation split. Each setup is run over 5 different seeds, and the mean accuracy is computed for comparison.

3 Results and Statistical Analysis (J)

3.1 Optimization results

Algorithm	**LR**	**BS**	**Epochs**	**Optim.**	**Drop.**	$\mu_{\text{test accuracy}}$
Grid Search	0.001	16	20	Adam	0.4	74%
Grid Search + Aug	0.001	16	20	Adam	0.4	76%
Random Search	0.00245	32	28	Adam	0.5	73%
Random Search + Aug	0.00034	16	30	Adam	0.1	79%

Table 1: Test results

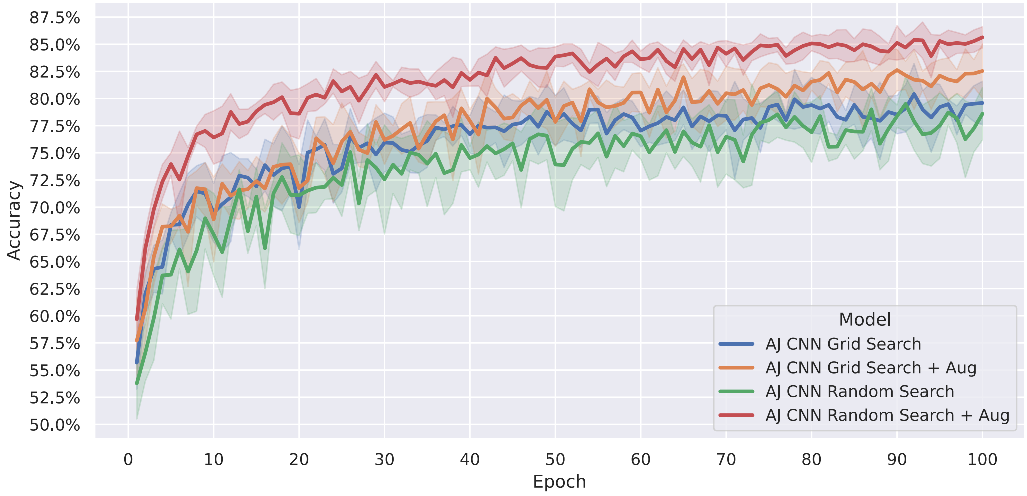

Figure 8: Mean test accuracy over 100 epochs, shaded areas being mean SD

Model and algorithm	**Mean test accuracy, CI**
Grid Search	$0.79,\ [0.77; 0.81]$
Grid Search + Aug	$0.82,\ [0.80; 0.84]$
Random Search	$0.78,\ [0.76; 0.80]$
Random Search + Aug	$0.85,\ [0.83; 0.87]$

Table 2: 95% confidence interval for a proportion: $\hat{p} \pm 1.96 \cdot \sqrt{\frac{\hat{p}(1-\hat{p})}{n_{\text{test data}}}}$

As the confidence intervals for the 2 grid search results overlap, there is insufficient evidence to conclude that they are significantly different. Despite the different augmentations applied to them, they share the same hyperparameters; therefore a p-test would show they're significantly different at a 5% significance level. With a p-value of 0.0498, we reject the null hypothesis that they are equal, with moderate evidence against the hypothesis, suggesting a small difference.[4]

The p-value comparing grid search to random search is 0.5234; therefore, at a 5% significance level, they are not significantly different. The p-value comparing grid search with augmentation to random search is 0.0094; therefore, at a 5% significance level, they are significantly different. Since the random search with augmentation confidence intervals do not overlap with the others, it is significantly different.

Referring to Figure 9, it is evident that the best-performing algorithm outperforms the stock model by about 5%.

[4] The method used is a 2 sample proportion z-test.

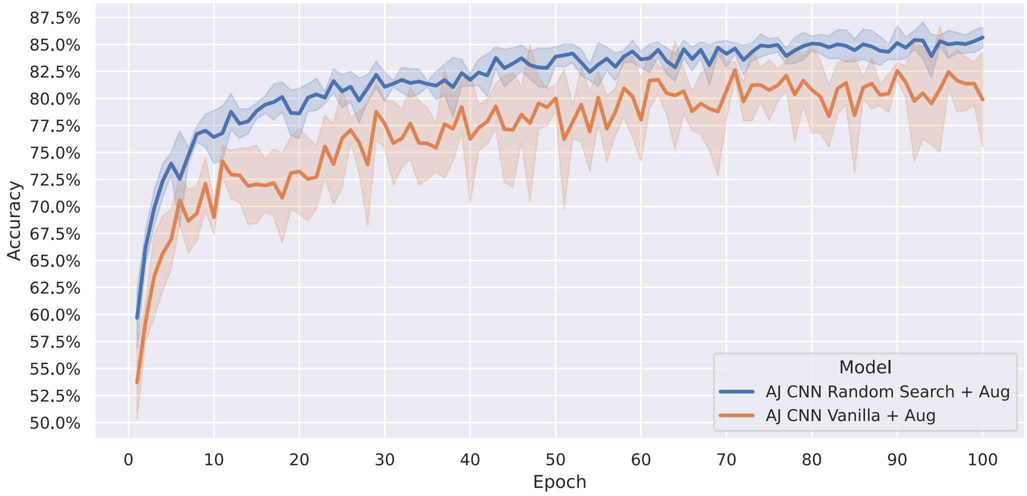

Figure 9: Mean test accuracy over 100 epochs, shaded areas being mean SD

4 Discussion

4.1 Results (A)

The notable disparity in test accuracy between random- and grid search can be attributed to the fixed set of constant values in grid search. While grid search relies on predefined values, random search explores a broader range by randomly selecting values within specified intervals. This implies that with an increased number of iterations, random search is more likely to approach the global minimum, albeit at the cost of greater computational resources. Within 100 trials, we were fortunate to obtain a set of values that outperformed the stock values.

The variability in results between the 2 runs of random search raises questions about whether the disparities are due to hyperparameters or data augmentation. Further testing with the respective parameters is needed. Figure 8 indicates that the new hyperparameters demonstrate more consistent performance throughout the entire duration of the run, displaying notably reduced deviation from the mean.

4.2 Methods (J)

In retrospect, a better implementation of the early stopping algorithm could have significantly reduced computational size. Out of 362 computations, only 44 trials (38 from random and 6 from grid) were pruned, being only 12% of the trials. The saved computational power could have been utilized to explore a broader range of hyperparameter values for each of the search algorithms, hence upping the chances for better results.

Additionally, the use of transfer learning with feature extraction and pre-trained parameters would have allowed for a better starting point. The time saved through this approach could have been allocated to the use of the complete dataset rather than a condensed version. However, this contradicts the intention of maintaining the compactness and simplicity of the easily trainable TinyVGG.

4.3 Broader View (J)

A potential bias in our current state that may occur while doing random /grid optimization is if we just choose to compute a large amount of time and conclude it is the perfect method. This is extremely resource-intensive and may not necessarily yield a better result than the one we got. Instead, one could adopt a lighter algorithm such as Bayesian optimization.[10]

4.4 Conclusion (A)

In conclusion, hyperparamter optimization effectively improved the test accuracy of a custom TinyVGG CNN. The model achieved a 5% increase in accuracy using data augmentation, k-fold, and random search, whereas the grid search algorithm faced challenges surpassing the accuracy of the stock model due to the limited hyperparameters it was provided.

References

[1] Poloclub, “Cnn explainer.” [Online]. Available: https://poloclub.github.io/cnn-explainer/

[2] T. Yu and H. Zhu, *Hyper-Parameter Optimization: A Review of Algorithms and Applications.* Inspur Electronic Information Industry Co., Ltd, 2020. [Online]. Available: https://arxiv.org/abs/2003.05689

[3] C. Alessio, “Animals-10.” [Online]. Available: https://www.kaggle.com/datasets/alessiocorrado99/animals10

[4] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” [Online]. Available: http://arxiv.org/abs/1502.03167

[5] G. Ghiasi, T.-Y. Lin, and Q. V. Le, “Dropblock: A regularization method for convolutional networks.” [Online]. Available: http://arxiv.org/abs/1810.12890

[6] M. Waskom, “Statistical estimation and error bars.” [Online]. Available: https://seaborn.pydata.org/tutorial/error_bars.html

[7] scikit-learn developers, *Cross-validation: evaluating estimator performance*. [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html

[8] J. Brownlee, “Hyperparameter optimization with random search and grid search.” [Online]. Available: https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/

[9] Y. Wang, “Why do we sample from log space when optimizing learning rate.” [Online]. Available: https://stats.stackexchange.com/questions/291552/why-do-we-sample-from-log-space-when-optimizing-learning-rate-regularization-p

[10] L. Gupta, “Comparison of hyperparameter tuning algorithms: Grid search, random search, bayesian optimization.” [Online]. Available: https://medium.com/analytics-vidhya/comparison-of-hyperparameter-tuning-algorithms-grid-search-random-search-bayesian-optimiz