

SLAM performance prediction using CNNs

Matteo Princisgh

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

To navigate and interact in an environment, an autonomous mobile robot typically needs a reference map. However, creating this map requires the robot to know its position within it. This is a well-known chicken-and-egg problem in robotics known as Simultaneous Localization and Mapping (SLAM). In SLAM, an accurate map is essential for precise localization of the robot, and, in turn, an accurate estimate of the robot's location is necessary to build an accurate map.

1.1 Related work

In the last decades, with the development of many new SLAM algorithms and techniques, there has been a growing need to find a common evaluation metric/framework. Many solutions are based either on manual evaluation [1], or on other external information about the environment and robot operation. For example, a widely used metric that requires the latter is the Absolute Pose Error (APE), proposed by Kümmerle et al. in [2], which is used to measure the displacement between the real trajectory of the robot and its estimated one. The APE is divided into two sub-metrics: the Absolute Translational Error (ATE) and the Absolute Rotational Error (ARE), which, as the name suggests, consider respectively the translational component and the rotational one.

Figure 1 shows how the ATE varies during the exploration of an environment, but for this work the main focus is on the average value calculated after the environment has been fully explored.

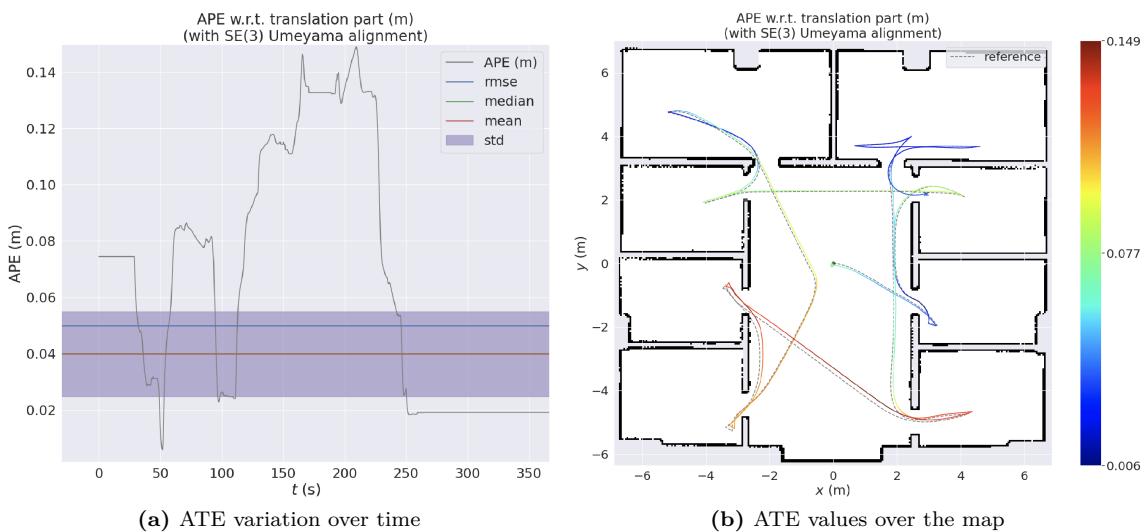


Figure 1: Visualizations of ATE variations during exploration, realized with *evo*¹.

In order to compute ATE and ARE values after a robot has explored an environment, we need to compare all the pairs of key-points sampled from the estimated trajectory of the robot and its ground truth (GT) trajectory. Specifically, during exploration we sample T poses, where each pose is a vector containing the position and orientation of the robot, and we build two sets $x_{1:T}$, $x_{1:T}^*$ that contain, respectively, the estimated and GT poses.

Next, we define $\delta_{i,j} = x_j \ominus x_i$ as the relative transformation² that moves the robot from pose x_i to x_j , and its counterpart $\delta_{i,j}^* = x_j^* \ominus x_i^*$. Then we define the $trans(\cdot)$ and $rot(\cdot)$ functions which extract, respectively, the translational component of a transform (i.e., a vector $z_{ij} \in \mathbb{R}^2$ representing the translation delta needed to move from pose i to j) and the rotational component (i.e., a vector $\theta_{ij} \in \mathbb{R}^2$ that expresses the angular deltas as Euler angles). Finally, to compute the localization error we consider all sorted pairs of transformations $\{\delta_{i,j} : j < i, \forall i \in 2 \dots T\}$:

$$\begin{aligned}\varepsilon(\delta) &= \varepsilon_t(\delta) + \varepsilon_r(\delta) \\ &= \frac{1}{N} \sum_{i,j} trans(\delta_{i,j} \ominus \delta_{i,j}^*) + \frac{1}{N} \sum_{i,j} rot(\delta_{i,j} \ominus \delta_{i,j}^*) \\ &= \frac{1}{N} \sum_{i,j} (||z_{ij} - z_{ij}^*||_2 + ||\theta_{ij} - \theta_{ij}^*||_2) \\ &= \frac{2}{(T-1)^2} \sum_{i=2}^T \sum_{j=1}^{i-1} (||z_{ij} - z_{ij}^*||_2 + ||\theta_{ij} - \theta_{ij}^*||_2)\end{aligned}$$

Note that these metrics are dependent on the availability of the GT data, which is easily obtained in simulated environments but extremely difficult to obtain when deploying a robot in the real world, as it would require an external system to accurately track its real position.

1.2 Task formulation

Calculating the localization error is usually performed after the robot has explored an environment, thus a full re-exploration is required in case the SLAM process yields poor results. Having an *a-priori* prediction of the localization error of a certain environment can be used to preemptively adjust the sensor equipment or the SLAM algorithm used by the robot.

To address these limitations, the work in [3], proposes a method to perform an *a-priori* prediction of the expected ATE and ARE, based on the geometric/topological characteristics of an indoor environment: using a feature extractor, described in [4], a set of structural features is extracted from a floorplan, in order to represent each environment as a vector $v \in \mathbb{R}^k$, where k is the number of distinct features. Successively, each environment is explored by a simulated robot, and, by collecting both GT and estimated trajectories, the corresponding ATE and ARE are computed and stored.

Finally, two models are trained on this dataset:

- **Linear model:** computed using linear regression.
- **Gaussian Process:** computed using an RBF kernel.

The goals of this project are the following:

1. Evaluate how multiple predictors, based on state-of-the-art convolutional neural networks (CNN), perform with respect to the linear and Gaussian models. In particular, with the usage of a CNN, it is possible to delegate the feature extraction stage to the network itself, so that a floorplan's image can be fed directly to the model, without computing a set of hand-crafted features.
2. Automatically handle model selection by delegating both hyperparameter optimization and neural architecture search to an optimization framework.

¹<https://github.com/MichaelGrupp/evo>

²Which is a transformation matrix that describes a rigid transform.

Each environment in our dataset is represented as a floorplan image and has three associated quantities: area (m^2), ATE (m), and ARE (rad). The predictor we want to build takes as input the floorplan image, together with the area, and produces two values in output, corresponding to the predicted ATE and ARE values.

More formally, the model h we want to learn can be expressed as follows:

$$h : X \rightarrow Y, \quad X \in (Mat_{n \times n} \times \mathbb{R}), \quad Y \in \mathbb{R}^2$$

2 Data preprocessing

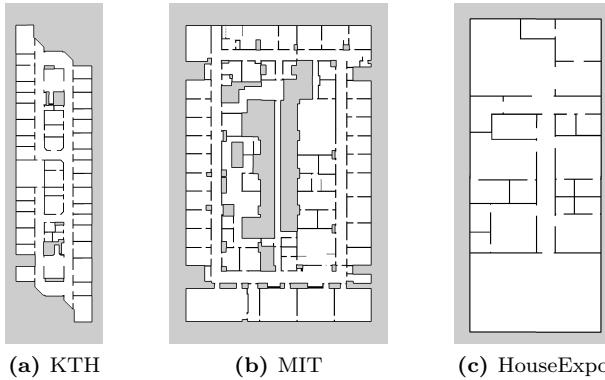


Figure 2: Example of floorplans from the datasets used in this work.

The dataset used in this work consists of floorplans and exploration data from three different sources: MIT campus [5], KTH campus [6], and HouseExpo [7]. The joint dataset consists of 5933 floorplans, and contains both domestic and work, school environments, as shown in **Figure 2**.

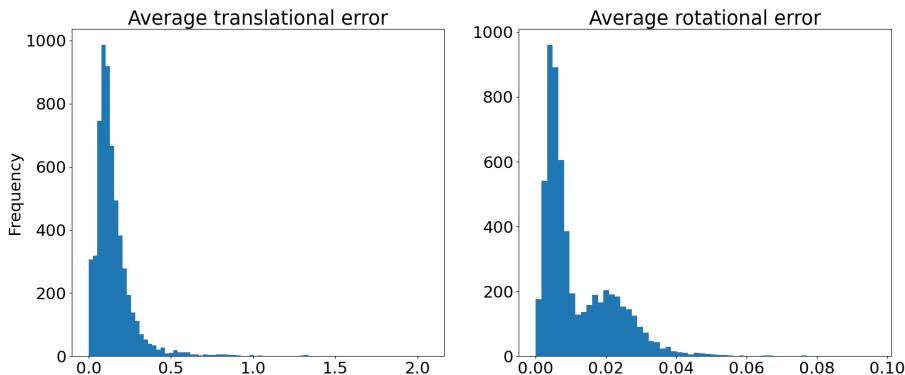


Figure 3: Original localization error distribution

The first step consists in visualizing the distribution of the data in our dataset, as shown in **Figure 3**, where we observe that the ATE and ARE distributions are respectively unimodal and bimodal, both with a short right tail and a small number of outliers.

The two quantities we want to estimate are distributed on distinct ranges and, to ensure that our models learn to predict each quantity proportionately, the data has to be re-scaled so that 99% of the values fall between 0 and 1, as shown in **Figure 4**. Additionally, the area of each floorplan is normalized as well to the $[0, 1]$ interval by dividing each datapoint by the maximum area value in the dataset.

Another preprocessing step consists in fixing images' resolution so that they can be fed to the CNN: for each floorplan we compute the minimum squared bounding box and scale it to a size ³

³This resolution choice is rather generous, given that both CNNs internally re-scale the image to 224x224px. Storing the dataset at a higher resolution leaves some margin for future experiments with different models.

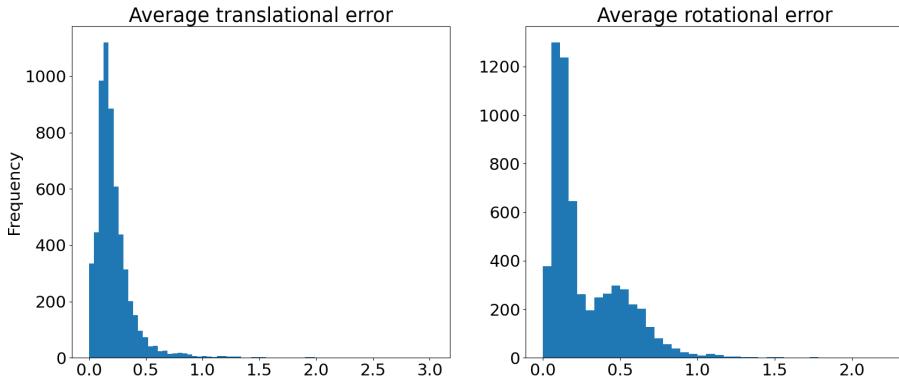


Figure 4: Re-scaled localization errors

of 500x500px. Floorplan images are loaded and processed lazily using TensorFlow’s data API⁴, which automatically handles the allocation and de-allocation policies for the images.

After splitting the floorplans in the usual train/validation/test partitions, respectively with 70/15/15 thresholds, the dataset creation steps consist of:

1. Image loading.
2. Data augmentation, which randomly rotates an image by multiples of 90°.
3. Creation of each datapoint tuple (image, area, label).
4. Shuffling and batching of datapoints into mini-batches of size 64.

Note that the data augmentation step is applied at runtime only on the training set, in order to improve the model generalization capabilities.

The vast majority of environments in the dataset satisfy the *Manhattan World* assumption, which states that all of the walls of an environment are aligned to two main directions, orthogonal to each other. Moreover, the main directions of the floorplan images are aligned with the image borders (i.e., most walls are either horizontal or vertical), thus, the choice of using multiples of 90° for image augmentation, is made to preserve the alignment of the walls.

3 Model architecture

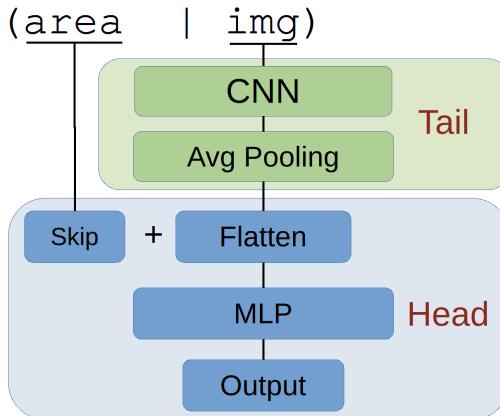


Figure 5: High level overview of the model structure.

The model can be divided in two distinct sections: a *tail* CNN, responsible for feature extraction, and a *head*, which is a multilayer perceptron (MLP), as shown in **Figure 5**.

⁴<https://www.tensorflow.org/guide/data>

The fact that the area input is connected to the MLP is due to the image scaling step described in **Section 2**: the original floorplans are represented in a specific resolution, for which it is known exactly how many pixels in the image represent a meter in the real building, but after rescaling the image to a fixed resolution this information is lost.

A robot exploring two environments of different sizes but identical layouts would yield different localization errors, while the model would produce the same output since the scaled input images are identical. Intuitively, an average localization error of $1m$ assumes a radically different meaning if an environment is very small (worse case) or very big.

Thus, the skip connection for the area parameter, which in practice becomes a bias term, is needed to compensate for the lost information about the environment size.

Moreover, we test and compare the performance of three different tail architectures: ResNet [8], EfficientNetV2 [9] and MobileNetV3 [10]. Considering that a hypothetical deployment of this model to a robotic platform would incur in several hardware limitations, by repeating the experiments on these architectures it is possible to empirically assess the performance/inference time trade-off of using one architecture over the other with respect to this specific task.

3.1 Residual networks

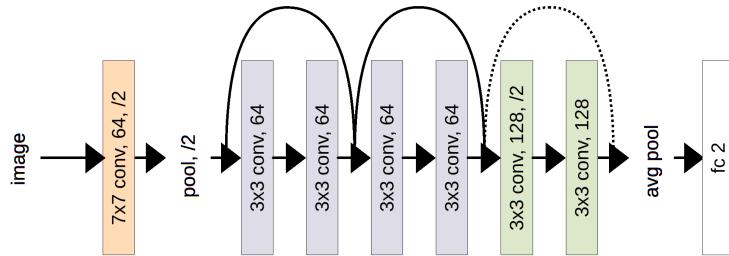


Figure 6: Layer configuration of the custom ResNet.

The ResNet network has been implemented from scratch, and its layers are summarized in **Figure 6**. The main characteristic of this architecture is the usage of shortcut connections, represented by the curved lines, which propagate the input matrix of two (or more) stacked blocks and sums it element-wise to their output (before the activation function). In case the first stacked block reduces the size of its input, the shortcut adjusts the matrix size by performing 1x1 convolutions with a stride of 2 (the dashed shortcut). These skip connections mitigate the training accuracy degradation phenomenon, where deeper networks converge to a training error higher than shallower ones, indicating that, as the depth of a CNN increases over a certain threshold, the optimizers are not capable of finding better solutions (i.e., it becomes easier getting stuck in a local minima).

The stacked blocks (i.e., the colored tiles in **Figure 6**) consist of the following layers:

- **Convolutional layer**, which learns a set of filters (64, 128, or 256) using kernels of size 3x3, except from the first stacked block, which uses a 7x7 kernel. In order to reduce the size of the input tensors, some layers use a stride of two (i.e., the '/2' field in the blocks).
- **Batch Normalization** [11], which normalizes the output of the convolutional layer in order to have zero mean and unit variance, and then applies an additional translation and scaling using two learned parameters γ and β :

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

Where x_i is the output of the convolutional layer, $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are the sample mean and sample variance of the elements of mini-batch \mathcal{B} , and ε is an arbitrarily small constant needed for numerical stability.

- **Activation layer**: which simply computes the non-linear activation function over the output of the previous layers, in this instance the *ReLU* function: $y_i = \max(0, x_i)$.

Additionally, the max pooling layer used before the residual blocks, down-samples the feature maps of the preceding convolutional layer; it does so by taking the maximum activation value in a 2x2 kernel passed over each feature map with a stride of two, whereas the average pooling layers takes the average of the activation values in the kernel.

The ResNet weights are trained from scratch on our dataset, in order to compare its performance versus that of a CNN pre-trained on a generic classification task (i.e., on *Imagenet* [12]). Training a CNN from scratch is a data-hungry process, depending on the network size, but given that the maps used in this work are visually simple and relatively noise-free, a shallow CNN is capable of extracting the features needed for our task. Thus, even with a training set of around 4000 images, it is feasible to train a network capable of competing with more complex architectures.

Instead, the EfficientNet and MobileNet weights are pre-trained on the *Imagenet* dataset and are kept frozen; thus, only the MLP weights are adjusted during training.

Finally, the hidden layers of the MLP heads, as well as the output layer, use ReLU as activation function, and the choice on using any regularization technique, such as Batch Normalization or Dropout, is delegated to the hyperparameter selection phase.

4 Hyperparameter selection

In order to find the *best-performing* model across the three architectures (i.e., ResNet, EfficientNet and MobileNet), we need to perform a hyperparameter (HP) selection phase for training the MLP head. Specifically we want to test, for each tail architecture, a set of common HPs chosen manually, and compare it to an optimized set of HPs, specifically tailored to each tail architecture.

To perform HP optimization two possibilities were considered:

- Perform a random search, over a small HP space, while using k -fold cross-validation on each combination of values.
- Perform black-box optimization over a larger HP search space, but test each combination only on the validation set (due to time constraints).

Ultimately, the choice fell on the latter option and the Optuna [13] framework was used to perform black-box optimization. Given the size of the dataset and the training step time of the models, each training run required a time ranging from 5 up to 20 minutes (on a machine with an RTX2080). This meant that choosing the second HP optimization approach allows for testing a much larger set of HP values for a fixed time budget. The trade-off of the second approach is an increase of uncertainty in the estimation of the model’s performance of each HP configuration.

The hyperparameters that were chosen for optimization are the following:

- Learning rate, in the interval [1e-2, 1e-5]
- Number of layers in the MLP head, in the interval [1, 6]
- Number of units in each layer, powers of two in the interval [64, 2048]
- Batch normalization (use / don’t use)
- Dropout (use / don’t use)

4.1 Sampling and pruning algorithms

The two main components used in the optimization phase are Tree-structured Parzen Estimators (TPE) [14], which belongs to the family of Bayesian optimization algorithms for generating new HPs, and HyperBand [15] for pruning trials with poorly-performing HPs.

The TPE sampler works as follows:

1. Take in input the hyperparameter search space \mathcal{H} .

2. Perform n startup trials (in our case $n = 10$) by randomly sampling sets of parameters $p \in \mathcal{H}$, and score each trial according to the validation loss produced by the model built with p .
3. Divide the trials in two partitions, using a quantile γ for thresholding, and define two distributions; the ‘good’ distribution $\ell(t)$ and the ‘bad’ distribution $g(t)$, created using a Gaussian Mixture Estimator over the two partitions.
4. To find the next best set p^* , we draw k samples distributed according to $\ell(t)$, and build the candidate set $S := \{p_i \sim \ell(t), \forall i = 1 \dots k\}$. Then we extract p^* as follows:

$$p^* = \arg \max_{p \in S} \frac{\mathbb{P}(p \in \ell(t))}{\mathbb{P}(p \in g(t))}$$

The idea is that we want to extract a p^* that maximizes the likelihood of belonging to the ‘good’ distribution, while minimizing the probability of belonging to the ‘bad’ one.

Steps 3 and 4 are repeated each time a new set of HPs is drawn and evaluated, so that the distribution $\ell(t)$ is updated in order to converge to a region that minimizes the validation loss of the models built by extracting HPs from it.

The HyperBand algorithm is needed to detect and prune trials with poor HP configurations, thus avoiding wasting time and resources on training models with poor hyperparameters. It works by modeling the budget allocation problem as a Multi-Armed Bandit problem: each HP configuration is a bandit, and the algorithm tries to find the optimal exploration/exploitation trade-off, where the explorative approach consists in testing many configurations for a short number of iterations, versus the exploitative approach which tests fewer configurations but with a greater allocation of resources.

Given a finite number of trials T , which is the maximum global number of training runs to be performed, HyperBand allocates the trials across a set of brackets and assigns a budget β to each bracket. The budget can be either a time limit or a resource limit; the default budget metric for the Optuna implementation is the number of training epochs.

Table 1 shows an example of how the algorithm works: there are $k = 3$ brackets, in which the leftmost is the explorative one and the rightmost is the exploitative one; the n_i field indicates the number of HP configurations that are to be tested during the i -th iteration, and b_i is the budget allocated to each configuration, where $b_i = \frac{\beta}{n_i}$.

Brackets						
i	n_i	b_i	n_i	b_i	n_i	b_i
0	16	1	4	4	2	8
1	8	2	2	8	1	16
2	4	4	1	16	-	-
3	2	8	-	-	-	-
4	1	16	-	-	-	-

Table 1: Example of budget allocation of HyperBand.

At the end of each iteration i , the worst performing configurations are discarded and the better ones are trained in the subsequent iteration with additional budget. In the example, the number of configurations kept for the successive iteration is $\frac{n_i}{2}$, but a different fractional coefficient η can be chosen to reduce the number of iterations. Finally, each bracket requires $\lceil \log_\eta(n_0) \rceil$ trials to be completed, which means that the example above would require 41 trials to test 22 different HP combinations, spending a total budget of 160 epochs.

The stopping criterion for each HP optimization step, other than HyperBand pruning, is EarlyStopping with a patience of 7 epochs with respect to the validation loss. Finally, the limit of optimization runs (i.e., the *Trial* parameter in **Algorithm 1**) is set to 50, the minimum resource allocated to each trial is three epochs and the maximum is 20.

The optimization process can be summarized as follows:

Algorithm 1: Optimize Hyperparameters

```

1 sampler  $\leftarrow$  TPE();
2 p  $\leftarrow$  baseline parameters;
3 v*  $\leftarrow \infty$ ;
4 for i  $\in$  1 ... Trials do
5   m  $\leftarrow$  train_model(p);           // stop training with pruning or EarlyStopping
6   v  $\leftarrow$  evaluate_model(m);       // compute the validation loss
7   if v  $<$  v* then
8     | v*  $\leftarrow$  v;
9     | p*  $\leftarrow$  p;
10  end
11  p  $\leftarrow$  sampler.Sample();
12 end
13 return p*;                  // return HPs that minimize val loss

```

4.2 Optimized architectures

After the optimization stage, the HP values obtained are shown in **Table 2**, where we observe how the optimization process converged to some peculiar combinations of layers for the MLP heads: for example, the EfficientNet head consists of five layers with [256, 256, 1024, 512, 512] units, which is a fairly unusual succession of layers. Moreover, for all architectures apart from ResNet, neither batch normalization nor dropout were used, which leads to models that converge very quickly to good validation losses, but tend to overfit for longer training runs, as described in the next section.

	Baseline	ResNet	EfficientNet	MobileNet
Learning Rate	1e-4	2e-4	5e-4	1.3e-3
Batch Norm.	Yes	No	No	No
Dropout	No	Yes	No	No
Dense layers	3	1	5	5
Units per layer	[1024, 512, 128]	[1024]	[256, 256, 1024, 512, 512]	[512, 64, 64, 64, 128]

Table 2: Manually selected HP (baseline) vs optimized HP.

With the data collected during the optimization process, it is possible to visualize the relative importance of each HP, as well as the contour plot of the objective function value with respect to the variation of parameter pairs, as shown in **Figure 7**. These visualizations give some insight on how the various combinations of HP affect the model performance and may be useful in case of an eventual ablation study. In particular, we observed how, across all of the models, the three hyperparameters with the higher relative importance were: learning rate, batch normalization, and number of units in the first dense layer. Indicating that carefully tuning these hyperparameters has a great influence on the resulting model performance.

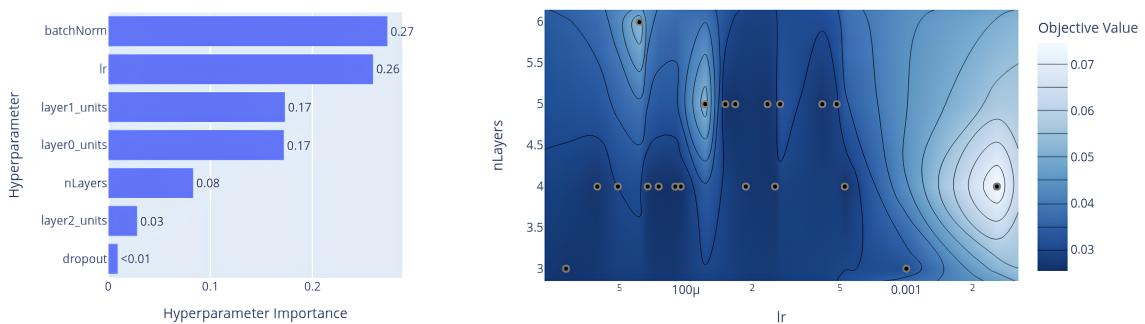


Figure 7: Relative importance and contour plot of the validation loss variation with respect to learning rate and number of MLP layers.

5 Training the models

The training phase is divided in two parts, where two different optimizers were used:

- ResNet tail training using SGD, which is the only architecture that requires to be trained from scratch, whereas EfficientNet and MobileNet use pre-trained weights.
- Training the heads of the other configurations with Adam, with both the baseline HP heads and the optimized ones.

In both phases, the MSE loss function is used, which, despite being more sensitive to outliers, converges quicker than the MAE loss. Moreover, both loss functions were tested and MSE consistently yielded better-performing models in a smaller number of epochs.

In order to train our custom ResNet from scratch, the SGD optimizer is used with mostly the same HPs as in [8], minus the learning rate, which is two orders of magnitude smaller due to the fact that our network is shallower than the original one. Additionally, there are no head layers attached during this phase, thus the two output units are connected directly to the flattened output of the CNN. The SGD HPs used for training are:

- **Learning rate:** $\eta = 1 \times 10^{-3}$, with a plateau decaying factor of 0.2, meaning that if during training the validation loss becomes stationary, the LR is reduced to allow for finer gradient steps necessary for converging to a local minima.
- **Momentum:** $\beta = 0.9$, which applies gradient smoothing using a weighted moving average on the succession of gradients. Formally:

$$\begin{aligned} m_t &= \beta m_{t-1} + \eta_t \nabla L(W_t) \\ W_{t+1} &= W_t - m_t \end{aligned}$$

Where m_t is the gradient computed at time t , W_t are the weights of the network, ∇L is the loss gradient (computed wrt. a random extraction of one datapoint), η_t is the learning rate, and β is the momentum.

- **Weight Decay:** $\lambda = 1 \times 10^{-4}$, is a regularization technique which consists in adding to the loss a penalty term proportional to the L2 norm of the weights in the network.

Putting it all together, the penalty term and final weight update become:

$$\begin{aligned} L_\lambda(W_t) &= L(W_t) + \frac{\lambda}{2} \|W_t\|^2 \\ W_{t+1} &= W_t - m_t - \eta_t \lambda W_t \end{aligned}$$

During the head training phase, the Adam optimizer is used, which combines the momentum technique described above with *RMSProp*, which is a similar technique but considers the weighted moving average of the squared gradients:

$$V_t = \gamma V_{t-1} + \eta_t (\nabla L(W_t))^2$$

Where γ is the discounting factor, similar to the β factor of momentum. Then, the squared root of this moving average is used to adaptively reduce the learning rate, and the update rule for the weights becomes:

$$W_{t+1} = W_t - \frac{m_t}{\sqrt{V_t + \varepsilon}}$$

This ensures that each weight of the network is adjusted adaptively, with momentum smoothing and scaled updates in case of large gradients⁵. The stopping criterion for each run is EarlyStopping, with a patience of 7 epochs, and, after stopping the training process, the best weights (i.e., those that minimize validation loss) are restored and saved to disk.

⁵The additional bias correction steps for the m_t and V_t terms were omitted for brevity.

5.1 Training results

The HyperBand parameters chosen for the HP optimization phase, allow us to test 17 different HP configurations, with a maximum trial amount of 50. The time needed to optimize each head depends heavily on the total number of parameters of the model, thus optimizing ResNet requires approximately 30 minutes, whereas optimizing EfficientNet requires up to five hours.

The number of parameters and the inference time of the optimized models are summarized in **Table 3**, where the GPU used for these tests is an RTX2080 and the CPU an i7-9700K. The times reported are relative to a single datapoint; calculated by dividing the inference time of a batch by its cardinality.

Model	Tail params	Head params	t-GPU(ms)	t-CPU(ms)
ResNet	389'760	135'170	0.6	4.2
EfficientNet	12'930'622	1'511'170	15.2	562
MobileNet	2'996'352	542'274	4.5	250

Table 3: Parameters count and inference time of each optimized model.

After having optimized each model, a training run is performed with the best HPs and k -fold cross validation is used to obtain robust estimates for the model performances. The results are summarized in **Table 4**, where we can observe how the ResNet optimized model outperforms the others, with respect to both the test and cross validation losses. While EfficientNet follows closely the performance of ResNet, MobileNet performs well on the test set but has a really high cross validation loss, which indicates that it may have a tendency to overfit on the training data, preventing it from generalize as well as the other models.

Model	Train Loss	Val Loss	Test Loss	5-fold CV
ResNet				
Baseline	0.0368	0.0365	0.0354	0.0416
Optimized	0.0366	0.0358	0.0352	0.0367
EfficientNet				
Baseline	0.0355	0.0370	0.0364	0.0382
Optimized	0.0351	0.0361	0.0363	0.0376
MobileNet				
Baseline	0.0261	0.0417	0.0452	0.0471
Optimized	0.0359	0.0362	0.0356	0.0519

Table 4: Baseline vs optimized performance of the three CNNs.

In summary, a shallow ResNet trained from scratch on a small dataset outperforms more complex pre-trained architectures. This indicates that our learning task is sufficiently easy, likely due to the visual simplicity of the images, combined with a good correlation between the structural characteristics of an environment and the resulting ATE/ARE values.

Figure 8 shows the training and validation loss curves observed for both baseline and optimized heads, where we can observe how the optimized architectures converge very quickly to a lower validation loss than their baseline counterparts. In fact, the baseline architectures require at least 10 to 15 epochs for the validation losses to become stationary, whereas the optimized ones require around 5 epochs. As an example of a poor training result, **Figure 8e** shows how the MobileNet head with baseline parameters overfits, which is deducible by the validation loss increase after the 20 epoch mark, while the training loss keeps decreasing.

Moreover, as shown in **Figure 8e** and **Figure 8c**, the validation losses are fairly noisy before convergence, whereas their optimized counterparts appear to converge in a smoother fashion.

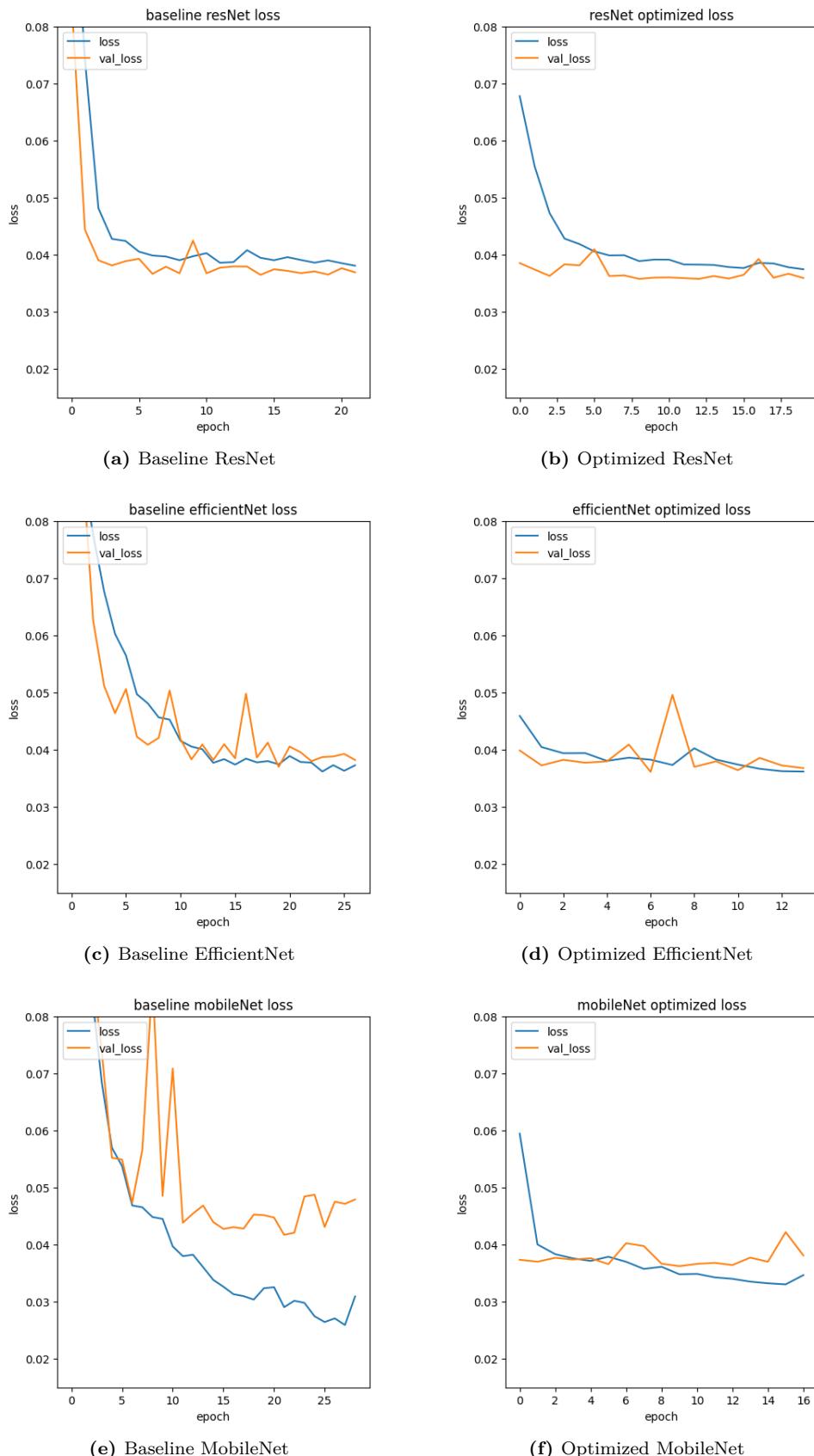


Figure 8: Training curves of the various models

6 Experimental results

Finally, once the models have been optimized and trained, it is possible to assess their performance relative to the two models described in **Section 1.1**, using the same metrics of [3] (i.e., R2 score, RMSE, and NRMSE).

As shown in **Table 5**, our custom ResNet performs slightly better than the linear model to predict both translational and rotational error.

ATE	R2-score	RMSE	NRMSE
ResNet	0.45	0.101	0.049
EfficientNet	0.40	0.105	0.051
MobileNet	0.39	0.106	0.052
Linear Model	0.46	0.104	0.050
Gaussian Process	0.27	0.104	0.050
ARE	R2-score	RMSE	NRMSE
ResNet	0.050	0.0095	0.131
EfficientNet	0.040	0.0096	0.132
MobileNet	0.043	0.0096	0.132
Linear Model	0.050	0.0099	0.136
Gaussian Process	-9.47	0.0327	0.449

Table 5: Performance comparison of CNNs vs traditional models.

The fact that the ResNet model performed better than its pre-trained counterparts is likely due to the fact that the CNN was trained specifically on the kind of images that are present in our dataset. This is also confirmed by the fact that, after the optimization process, the pre-trained models have deeper, more complex heads, whereas the ResNet model only needs a single layer with 1024 units. Moreover, the ResNet model achieved the best score during k -fold cross validation, indicating that it is likely capable of generalizing well.

The deployment of the ResNet model on a robotic platform would also benefit from having a very small CPU inference time of around 4ms, compared to heavier models such as EfficientNet, which has an inference time of more than 500ms.

7 Conclusions

The model selection phase yielded the expected results (i.e., finding good HPs in a limited time frame), and was able to produce a CNN-based model that performs better than the linear one. Despite the lack of interpretability of the ResNet model, its low inference times allows for real-time usage on a robot, and its tail CNN weights can be used to perform transfer learning for future prediction tasks.

Further improvements of this work may include:

- Extend the hyperparameter selection phase to perform neural architecture search and optimization of the ResNet CNN layers
- Fine-tuning of the pre-trained models
- Usage of other tail architectures, like attention-based models; although the usage of transformers would require a much bigger labeled dataset which is, to the best of our knowledge, not publicly available.

References

- [1] Benjamin Balaguer, Stefano Carpin, and Stephen Balakirsky. Towards quantitative comparisons of robot algorithms: Experiences with slam in simulation and real world systems. In *Proc. IROS*, 2007.
- [2] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On measuring the accuracy of slam algorithms. *Autonomous Robots*, 2009.
- [3] Matteo Luperto, Valerio Castelli, Fabio Bonsignorio, Francesco Amigoni, et al. Predicting performance of slam algorithms. In *Proc. AIXIA*, 2021.
- [4] Matteo Luperto and Francesco Amigoni. Extracting structure of buildings using layout reconstruction. In *Proc. IAS*, 2019.
- [5] Emily J Whiting. *Geometric, topological & semantic analysis of multi-building floor plan data*. PhD thesis, Massachusetts Institute of Technology, 2006.
- [6] Patric Jensfelt Alper Aydemir and John Folkesson. What can we learn from 38,000 rooms? reasoning about unexplored space in indoor environments. In *Proc. IROS*, 2012.
- [7] Li Tingguang, Ho Danny, Li Chenming, Zhu Delong, Wang Chaoqun, and Max Q.-H. Meng. Houseexpo: A large-scale 2d indoor layout dataset for learning-based algorithms on mobile robots. *arXiv preprint arXiv:1903.09845*, 2019.
- [8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. CVPR*, 2016.
- [9] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *Proc. ICML*, 2021.
- [10] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proc. CVPR*, 2019.
- [11] Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. CVPR*, 2009.
- [13] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proc. SIGKDD*, 2019.
- [14] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyperparameter optimization. *Proc. NIPS*, 2011.
- [15] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.