Artificial Intelligence Homework 1: Backpropagation in Multilayer Perceptrons

Juan Pablo Ossa Zapata
Ingeniería Matemática
Universidad EAFIT
Medellín, Colombia
jpossaz@eafit.edu.co

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Multilayer perceptrons (MLPs) are a key component of deep learning, which has become one of the most active fields in artificial intelligence (AI) research. The concept of MLPs was introduced in the 1960s, but it wasn't until the development of backpropagation algorithm in the 1980s that they became widely used. MLPs are artificial neural networks that consist of multiple layers of interconnected neurons, each of which applies a non-linear transformation to the input. The output of one layer becomes the input of the next layer, with the final output being produced by the last layer.

Training MLPs is not a trivial task. In order to learn from data, the network needs to adjust the weights and biases of its neurons. This is typically done using the backpropagation algorithm, which propagates the error from the output layer backwards through the network to adjust the weights and biases. However, this algorithm requires a lot of computing power, and the performance of the network depends heavily on the choice of architecture, activation functions, learning rate, and other hyperparameters.

The backpropagation algorithm relies on the availability of gradients, which are computed by taking the partial derivative of the output with respect to each weight and bias in the network. The gradient is used to update the weights and biases during training, in a process called gradient descent. However, if the network is too deep or too wide, the gradients may become very small or vanish altogether, making it difficult for the network to learn. This is known as the vanishing gradient problem.

To avoid the vanishing gradient problem, researchers have developed various techniques such as weight initialization, batch normalization, and skip connections. These techniques are mostly beyond the reach of this work, but they aim to make the gradients more stable and to enable the network to learn more effectively. However, choosing the right architecture and hyperparameters still requires a lot of trial and error.

In this report, we focus on the backpropagation algorithm for MLPs and investigate how different hyperparameters affect the performance of the network. We use a dataset of housing prices in Boston to train and test MLPs with various combinations of architecture, activation functions, and learning rates. We also log the gradients and use them to analyze why some models learn better than others.

The goal of this project is primarily pedagogical, aimed at learning how changes in architecture affect the performance of the backpropagation algorithm in MLPs. By systematically varying the number of layers, number of neurons in each layer, learning rate, and activation functions, we can gain a deeper understanding of the trade-offs between model complexity and generalization ability. This is an important aspect of deep learning research, as it allows us to design more efficient and effective neural networks for a wide range of applications. Additionally, by tracking the learning of the models using various metrics, we can identify the most important factors for training successful neural networks. Overall, the pedagogical goal of this project has important implications for the wider field of AI, helping to advance our understanding of how to build more powerful and robust machine learning models.

The central hypothesis of this project is that the performance of MLPs trained using the backpropagation algorithm will vary depending on the architecture, learning rate, and activation functions used. By systematically exploring these different hyperparameters, we aim to identify the optimal configuration for this particular dataset of housing prices in Boston. We will use the validation metrics to choose the best model and then report its final performance using the test set. This hypothesis is important because it allows us to gain a better understanding of how to train MLPs effectively for a wide range of applications, which has significant implications for the field of AI. By identifying the best hyperparameters for this dataset, we can build more accurate models that generalize well to unseen data, which is essential for all real-world applications.

II. Methods

The central question of this project is which architecture works best with backpropagation algorithm for training MLPs. Backpropagation relies on gradients to update the weights of the network during training, and different architectures can have better or worse gradients depending on their structure. Moreover, different architectures can have varying capacity for learning and generalization, making it important to identify the optimal configuration for a particular dataset. By systematically varying the number of layers, number of neurons in each layer, learning rate, and activation functions, we aim to identify the best architecture for the given dataset of housing prices in Boston. Additionally, we will analyze the norms of the gradients for each neuron in each layer, and the average gradient norm, to better understand why some models learned better than others.

The dataset used in this project is the well-known Boston Housing dataset, which contains information on housing prices in Boston from 1978. For this project, we selected four columns from the dataset - RM, LSTAT, PTRATIO, and MEDV. All of the columns were normalized to a range of -1 to 1 to ensure that the inputs are in the same scale. The first two columns were selected as inputs, and the last two were used as outputs. The specific meaning of the columns is not important for the purposes of this project. Overall, the dataset contains 506 entries, which we split into training, validation, and testing sets with 293, 98, and 98 entries, respectively. By using this dataset, we can test the performance of different architectures and hyperparameters in a realistic and widely studied scenario, which enhances the relevance and impact of our findings.

To evaluate the performance of different architectures and hyperparameters, we used a combination of metrics. First, we calculated the norms of the gradients for each neuron in each layer, as well as the average gradient norm. This analysis helps us identify which architectures have better gradients and are more likely to converge to a good solution. Additionally, we measured the mean squared errors (MSE) for the training, validation, and testing sets. By comparing the MSE for different hyperparameters and architectures, we can identify which configurations perform better overall and which ones are more prone to overfitting. To further assess the potential for overfitting, we used a divergence metric, which is calculated by dividing the validation error by the training error. A higher divergence indicates that the model is overfitting on the training data, and therefore, it is less likely to generalize well to new data. By combining these metrics, we can gain a comprehensive understanding of the performance and generalization capabilities of different MLP architectures and hyperparameters.

In addition to evaluating the overall performance of different architectures and hyperparameters, we also analyzed the importance of individual parameters. To do this, we looked at how changes in specific hyperparameters affected the performance metrics, such as the MSE and divergence. By dividing the target metrics into subsets based on the value of a particular hyperparameter, we were able to visualize how the performance changed as we varied that parameter. This analysis helped us identify which hyperparameters were most important and which ones had a more marginal impact on performance. Furthermore, since the goal of this project is pedagogical, we looked more closely at hyperparameters that introduce higher variance, even if they do not necessarily result in better performance.

One of the metrics that we used to evaluate the performance of different MLP architectures was the divergence metric, which is the ratio of the validation error to the training error. The idea behind this metric is that if the validation error is much higher than the training error, it may be an indication that the model is overfitting to the training data. By monitoring the divergence metric during training, we can identify which models are most likely to suffer from overfitting and adjust their hyperparameters accordingly. However, there are potential downsides to this metric. For example, it may be possible for a model to have a low divergence metric but still perform poorly on unseen data. Therefore, it is important to use other metrics in conjunction with the divergence metric to fully evaluate the performance of the MLP architectures.

To fully evaluate the performance of the MLP architectures, we compare the distribution of the target output with the predicted output for the best performing model. By analyzing the shapes of these distributions, we can identify any inherent biases of the model and what kinds of shapes it is able to reproduce or not. In addition, we use scatter plots to visualize how well the mapping between the input and output data is being learned by the model. Ideally, this mapping should be close to an identity function. By comparing the predicted output with the true output, we can determine if the model is accurately capturing the relationships between the input and output variables.

We trained the model on all the training data without using minibatches or batch sizes. This was done to avoid the noisy gradients and divergence towards infinity observed when using minibatches. However, training on the entire dataset could also lead to a high computational cost and slower convergence, making it less practical for larger datasets. Therefore, we need to carefully balance the trade-off between using minibatches and batch sizes to ensure both efficient training and accurate results.

We used a standard machine learning approach by dividing the dataset into three sets: training, validation, and testing. The training set was used to train the models, the validation set was used to optimize the model parameters, and the test set was used to report the final performance of the models. Specifically, we used 293 entries for training, 98 entries for validation, and another 98 entries for testing. This division ensured that we did not use the same data for training, optimizing parameters, and testing, which could lead to overfitting and biased results. By using this approach, we ensured that the models were trained on a representative subset of the data and that their performance could generalize well to unseen data.

We used distributional analysis to examine the performance of our models by looking at the distribution of error metrics across different activation functions and model architectures. To visualize these distributions, we used a kernel density estimate (KDE) to generate smooth curves that represent the distribution of error values for each combination of parameters.

By comparing the shape and position of these distributions, we were able to identify which models performed best and which parameters had the greatest impact on model performance. It is important to note that distributional analysis is a high-level approach that may overlook important details in the data. For example, it may not capture the nuances of individual data points or identify outliers that could impact the overall results. Additionally, it relies on assumptions about the underlying distribution of the data that may not hold true in all cases.

As mentioned previously, the optimization of the models was done using the backpropagation algorithm. The first step of this algorithm is the fordward pass. The input is propagated through the network, layer by layer, and at each layer, a local field and an activation value are calculated. The local field is the weighted sum of the inputs, with each weight representing the strength of the connection between the input and the neuron. The activation value is the output of the neuron after passing its local field through an activation function, which introduces nonlinearity into the network.

In mathematical terms, the local field of a neuron j in layer i can be expressed as:

$$u_j^{(i)} = \sum_{k=1}^{n^{(i-1)}} w_{jk}^{(i)} a_k^{(i-1)}$$

where $w_{jk}^{(i)}$ is the weight connecting neuron k in layer $i-1$ to neuron j in layer i, $a_k^{(i-1)}$ is the activation value of neuron k in layer $i-1$, and $n^{(i-1)}$ is the number of neurons in layer $i-1$.

The activation value of a neuron j in layer i is then calculated by applying an activation function ϕ to the local field $u_j^{(i)}$:

$$a_j^{(i)} = \phi(u_j^{(i)})$$

Then, the backward pass is performed. This stage corresponds to computing the local gradients of the neurons in each layer of the network, and using them to compute the gradients of the weights. Given the local field u_j of a neuron j in layer l, the local gradient δ_j can be computed as follows:

$$\delta_j = \frac{\partial E}{\partial u_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial u_j} = \frac{\partial E}{\partial y_j}\phi'(u_j)$$

where E is the error function, y_j is the output of neuron j, and $\phi(u_j)$ is the activation function of neuron j.

For all neurons in the network, we allocate the local gradients and set them to zero. We then compute the local gradient of the neurons in the output layer as follows:

$$\delta_j = \frac{\partial E}{\partial y_j}\phi'(u_j) = \phi'(u_j)(t_j - y_j)$$

where t_j is the target output for neuron j. For the neurons in the hidden layers, we compute the local gradient using the local gradients from the next layer:

$$\delta_j = \phi'(u_j)\sum_k w_{j,k}\delta_k$$

where $w_{j,k}$ is the weight between neuron j in layer l and neuron k in layer $l+1$.

Having computed the local gradients for all neurons in the network, we can now compute the gradient of the network with respect to the weights. The gradient of the weight $w_{j,k}$ is given by:

$$\frac{\partial E}{\partial w_{j,k}} = \delta_j x_k$$

where x_k is the input to neuron k in layer $l+1$. This gradient is used to update the weights during training.

When we have the gradients for each weight, we can update them using gradient descent with a learning rate. The update rule for a weight is:

$$w_{i,j} \leftarrow w_{i,j} - \eta\frac{\partial E}{\partial w_{i,j}}$$

where $w_{i,j}$ is the weight connecting neuron i in layer l and neuron j in layer $l+1$, η is the learning rate, and $\frac{\partial E}{\partial w_{i,j}}$ is the partial derivative of the loss function with respect to that weight. We update each weight in the network using this rule, expecting to have a lower error in the next iteration, until we reach convergence.

The implementation for this project was done using the Julia programming language, specifically version 1.8.5. We utilized the Weights and Biases service for logging and tracking the experiments. All the experiments were run on a 32-core node from the Apolo Scientific Computing Center. This allowed us to take advantage of the parallel processing capabilities of Julia and to handle the computational requirements of training and testing multiple models with different combinations of parameters. The use of Weights and Biases also helped keep track of the experiment results and monitor the training progress.

III. Results and discussion

The choice of activation function was the parameter that made the biggest difference. As seen in Figure 1, linear was the best activation function under both metrics, with a validation error of approximately 0.2 and a divergence error of 1.05. Sigmoid had similar validation error distributions, with a wider bell curve centered around 0.5. However, tanh had a slightly higher divergence error with a center at approximately 1.15 and a narrower distribution. Relu, on the other hand, had a thin spike centered around 0.5 for validation error, but a higher divergence error centered at approximately 1.15. These results indicate that the choice of activation function can significantly affect the performance of the model. In particular, linear activation was able to achieve good generalization and avoid overfitting. Note that some of the distributions are wider, implying that they have more room for the other parameters to make a difference.

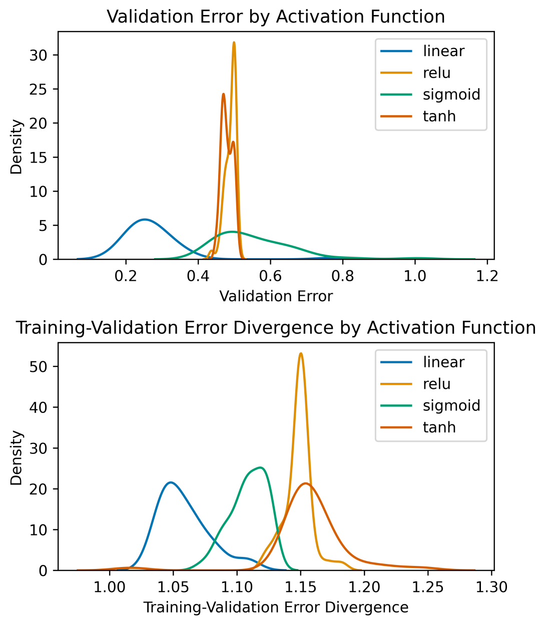

Fig. 1. Validation error and error divergence by activation function.

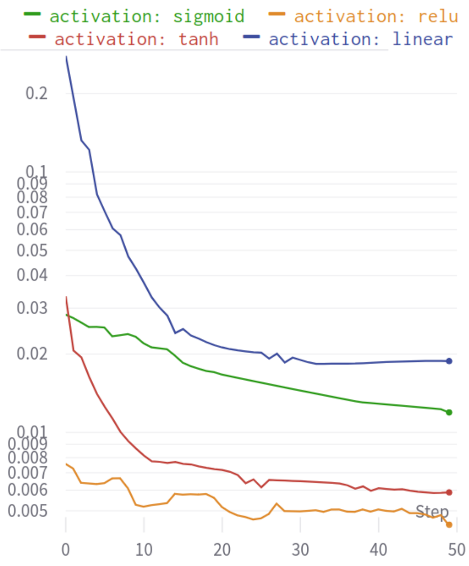

Fig. 2. Average gradient norm by activation function

The plot in Fig. 2 shows the average gradient norm for each activation function across all the 50 epochs. The results indicate that linear has the strongest gradients, followed by sigmoid, tanh, and relu. The strength of the gradients decreases by about one order of magnitude with regards to the last. In the previous analysis, we found that linear performed better than the other activation functions in terms of validation error and error divergence. This is consistent with the observation that linear has the strongest gradients, as it suggests that stronger gradients help to produce better performing models. Overall, the results show that the choice of activation function is a critical parameter to consider when training multilayer perceptrons, and that linear may be a good default choice, at least for simple datasets like the one we used in this study.

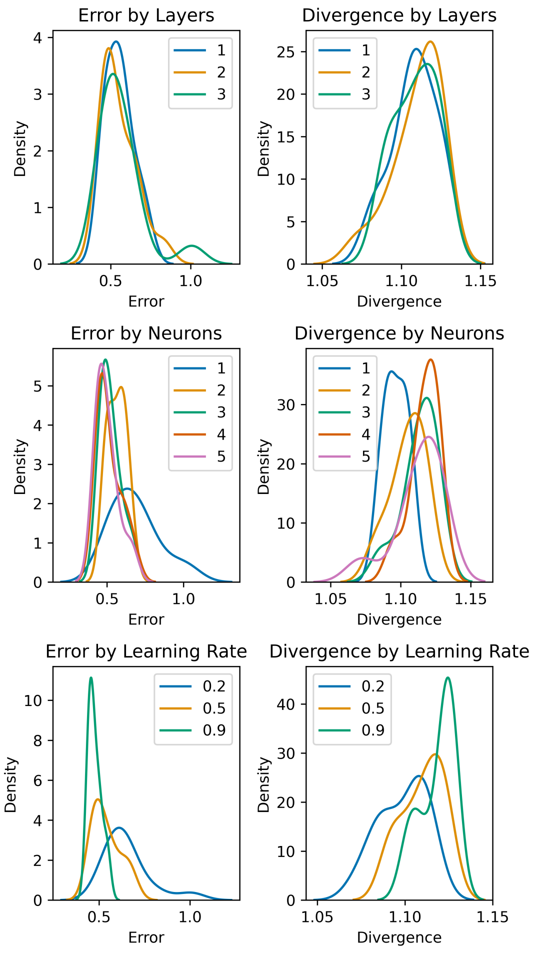

Fig. 3. Validation metrics for various parameters. Only for sigmoid activation. Layers refers to the number of hidden layers, and neurons refers to the number of neurons per layers.

In order to further understand the impact of parameters on the performance of the model, we focused on the sigmoid activation function which had the highest variance in terms of validation error. Figure 3 shows the distributions of error and divergence when changing the number of layers, number of neurons, and learning rate. Interestingly, changing the number of layers does not seem to have a significant impact on the performance of the model, as the distributions are practically the same. However, changing the number of neurons per hidden layer does show more difference, with the distribution for 1 neuron per hidden layer having more variance for error and a bit shifted to the left for divergence. However, all other values don't seem to make much more of a difference.

The most interesting results are observed when changing the learning rate. The distributions show that a high learning rate consistently produces lower error, but higher divergence. On the other hand, a low learning rate has higher variance for error and is shifted to the right, indicating worse performance, but its divergence is also shifted to the left, indicating better generalization. This highlights the fact that the learning rate introduces a tradeoff between performance and overfitting. Therefore, choosing an appropriate learning rate is crucial for achieving optimal performance and preventing overfitting.

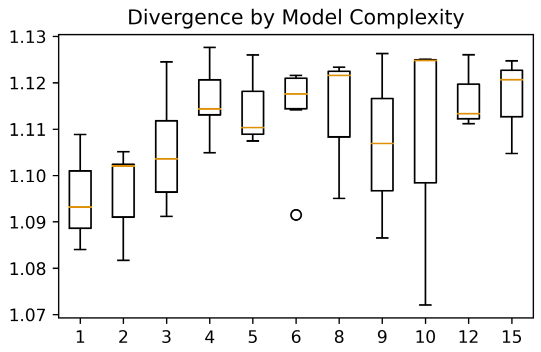

Fig. 4. Boxplot of training-validation error divergence as model complexity increases. Only for sigmoid activation.

The boxplot shown in Figure 4 displays the trend of the training-validation error divergence as the complexity of the model increases. The model complexity is defined as the number of hidden layers multiplied by the number of neurons per hidden layer. As expected, the plot shows a positive correlation between model complexity and overfitting. However, it is interesting to note that some of the more complex models still have the ability to generalize well, as indicated by the whiskers that extend all the way down at complexity 10. These results suggest that model complexity is an important factor in determining the generalization ability of a model, but it is not the only factor. Further analysis is always needed to identify the specific characteristics of the models that lead to their success or failure.

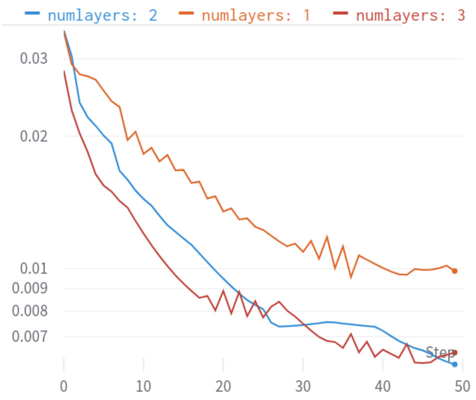

Fig. 5. Average gradient norm by number of hidden layers

We now explore how the number of hidden layers in the neural network architecture affects the strength of gradients. The results, as shown in Figure 5, indicate that using only one hidden layer produces much stronger gradients than using 2 or 3 hidden layers. Interestingly, there is not much difference in gradient strength between using 2 or 3 hidden layers. This finding suggests that increasing the number of hidden layers may not necessarily lead to better gradient strength, which could potentially impact the model's performance and learning.

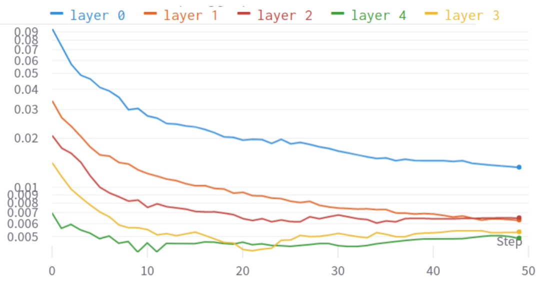

Fig. 6. Average accross experiments of gradient norms for each layer. 0 refers to the last layer, 1 refers to the previous to last layer, and so on.

The results in fig:layergrads clearly show that the gradient is stronger on the last layer and decays as one moves back. This implies that the earlier layers are more vulnerable to getting stuck with their current activations if they don't receive good gradients, which could affect the performance of the entire model. These findings are in line with the previous analysis, which showed that models with fewer hidden layers tended to have stronger gradients, leading to better performance. Therefore, it is important to carefully consider the number of hidden layers in a multilayer perceptron to ensure that all layers receive sufficient gradient information to effectively learn the underlying patterns in the data.

Find best, median, worst model.

Median is 0.48207072084988933 error

TABLE I
SUMMARY OF MODELS

Model	Best	Median	Worst
Activation	Linear	Sigmoid	Sigmoid
Layers	1	2	3
Neurons	2	3	1
Validation Error	0.2238	0.4821	1.007
Divergence	1.046	1.121	1.096
Gradient Average	0.0185	0.0068	0.0096

The summary table I presents the best, median, and worst performing models based on the validation error, divergence, and average gradient norm. The best performing model used a linear activation function, had a single hidden layer with two neurons, and achieved a validation error of 0.2238. This indicates that a simpler architecture with fewer layers and neurons can still perform well on this dataset. The median performing model had two hidden layers with three neurons each and a sigmoid activation function. The worst performing model had three hidden layers with only one neuron each and also used a sigmoid activation function. This suggests that having too many layers or too few neurons per layer can lead to poor performance. The divergence metric was similar across all models, indicating that overfitting was a common issue. The average gradient norm was strongest for the best performing model with a linear activation function and a single hidden layer, which supports the idea that having a simpler architecture can lead to better gradients and better performance.

The fact that linear activation function worked the best out of all the functions that were tested, it might imply that the data used in this experiment was mostly linearly correlated. Linear activation function is a good choice when the input-output relationship is linear. This is because the output of a linear function is proportional to the input. In contrast, nonlinear activation functions like sigmoid, relu, and tanh, introduce nonlinearities to the input-output relationship which might be unnecessary if the data is mostly linear. Therefore, if linear activation function works best, it might indicate that the data has a strong linear correlation.

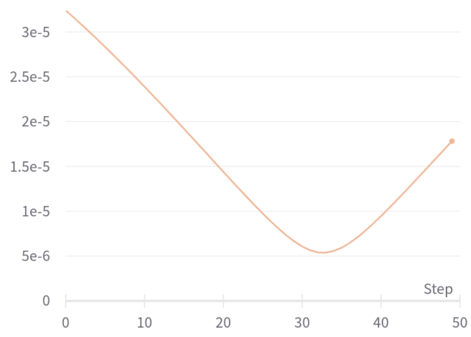

Fig. 7. Gradient norm for the first layer of the worst model.

The results show that the worst performing model has 3 layers with only a single neuron in each one of them. Further analysis shows that the gradients in the first layer of this model were extremely low, which combined with its sigmoid activation, probably caused the data to become numerically compressed and not learn anything at all. This can be seen in figure 7, where the gradient norms for the first layer of the worst model are plotted. It is very important that the gradients are strong enough to propagate through the network to make it learn. However, in this case, the gradients in the first layer were too weak to make any significant impact on the subsequent layers, thus resulting in poor performance.

It is important to note that while the worst model has a good divergence metric, it also has the worst validation error. This may seem contradictory at first, but it can be explained by the fact that the model is performing equally poorly on both the training and validation sets. In other words, it is not overfitting to the training data, but it is also not learning anything useful from the data.

The scatter comparisons in figure 8 for predictions on the test data by the worst, median, and best models are particularly useful to evaluate the actual performance of the models. It's evident that the median and worst models have a very low dynamic range on their outputs, with predictions being mapped to a single value. This implies that these models were unable to capture the complexity of the data and ended up being overly simplistic. The best model has a better dynamic range, but it

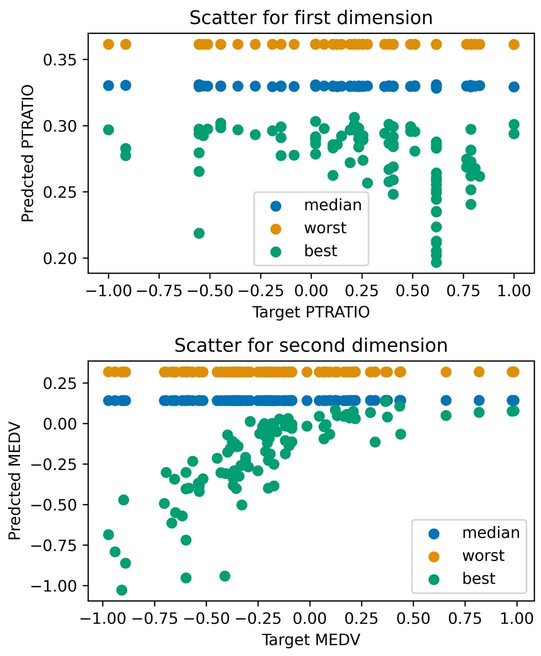

Fig. 8. Scatter comparisons for predictions on test data by the worst, median, and best model.

didn't perform well on the first output, PTRATIO, suggesting that there isn't enough information in the inputs to model it accurately. On the other hand, the second output, MEDV, performed reasonably well for the best model in the range of -1 to 0, where the predictions track accurately. However, in the range of 0 to 1, the predictions worsened, which might suggest that the relationship between the inputs and the output isn't linear in that range. These results highlight the importance of visually inspecting the model predictions to assess their quality and to understand where they're making errors.

The distribution comparison in figure 8, of the target and predicted outputs by the best model provides an insight into how well the model is performing on the given dataset. It shows that the distribution of the predicted output for PTRATIO is significantly different from the real data. This could indicate that the inputs chosen for the model are not sufficient to accurately predict this output variable. On the other hand, the predicted distribution of the MEDV output variable by the best model is better. It is similar in shape to the real data distribution, which is roughly triangular. However, it is slightly tilted. This could imply that the model is capturing some of the underlying patterns in the data but needs to be further improved to precisely capture the triangular shape of the distribution.

IV. Conclusion

Based on the experiments and analysis conducted in this task, we can conclude that we have strong evidence for our hypothesis. Changing the parameters did indeed have a significant effect on the performance of the models, and their generalization ability. The models with different combinations

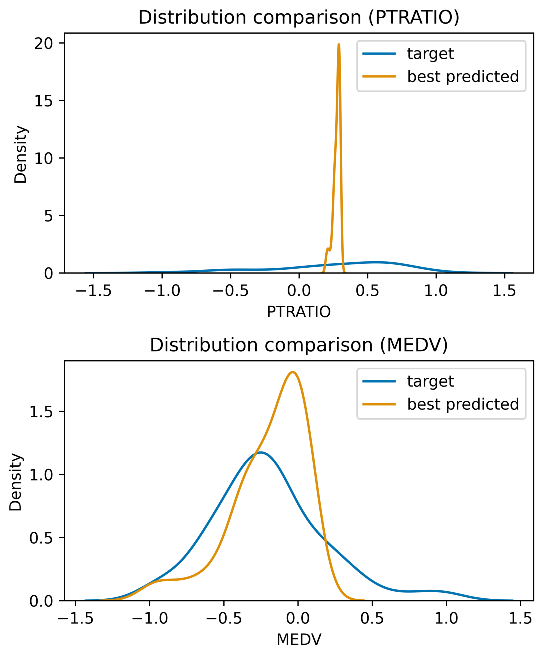

Fig. 9. Distribution comparison on the target and predicted outputs by the best model.

of parameters were trained and evaluated on the same dataset, and the results showed that the choice of parameters can lead to significant differences in model performance.

By varying the number of layers, number of neurons in each layer, learning rates, and activation functions, we were able to see how different architectures and hyperparameters affect the performance of the models. Through the use of metrics such as divergence and gradient norms, we were able to identify overfitting and understand why some models learned better than others.

Overall, the analysis was successful in achieving its pedagogical goal of helping the author learn about the effects of different parameters on model performance. Furthermore, the results provide valuable insights into the design and optimization of neural networks for various tasks.

While our analysis provided valuable insights into the performance of different models trained with different hyperparameters, there are some potential limitations that need to be considered. One limitation is that the distributional analysis we performed is quite broad and may not be able to capture all the nuances of the output distributions. Additionally, our analysis focused only on the specific dataset we used, and it is possible that the results could differ on other datasets.

Another limitation is that our approach for evaluating the models may not be optimal for all scenarios. For example, our use of the validation error divided by the training error as a ”divergence” metric may not be appropriate in situations where the training data is not representative of the population. In such cases, overfitting could occur even if the validation error is lower than the training error.

Furthermore, our decision to use the Julia programming language and the Weights and Biases service may limit the accessibility of our work to those who are not familiar with these tools. While Julia is a powerful language for scientific computing, it is not as widely used as other languages such as Python or R.

In conclusion, while our analysis provides valuable insights into the performance of multilayer perceptron models trained with different hyperparameters, there are some potential limitations that need to be considered. It is important to carefully evaluate the appropriateness of our methods for specific scenarios and datasets, and to consider the accessibility of the tools and languages we used.

There are several potential avenues for future work based on the findings of this study. One possibility would be to explore other neural network architectures, such as convolutional neural networks (CNNs), and apply similar analyses to see how different architectures perform on the Boston Housing dataset. CNNs are commonly used in image and video recognition tasks, but they can also be applied to other types of data, such as time series or text data. It would be interesting to see if a CNN could improve upon the performance of the MLPs on the Boston Housing dataset.

Another potential area of future work could be to explore other datasets to see if the findings of this study hold true across different types of data. The Boston Housing dataset is a well-known and widely used dataset, but there are many other datasets available that could be used to test the performance of MLPs with different parameter settings. It would be interesting to see if the optimal parameter settings found in this study generalize to other datasets, or if different parameter settings are needed for different types of data.

Furthermore, it would be interesting to explore other optimization algorithms besides backpropagation, such as evolutionary algorithms or reinforcement learning, to see how they compare to backpropagation in terms of performance and generalization ability. This would allow for a more comprehensive understanding of the strengths and limitations of different optimization algorithms for training MLPs.

Overall, this study has provided valuable insights into the performance and generalization ability of MLPs with different parameter settings on the Boston Housing dataset. Further exploration of other architectures, datasets, and optimization algorithms could provide even deeper insights into the capabilities of neural networks for regression tasks.

Acknowledgment

The authors acknowledge supercomputing resources made available by the Centro de Computación Científica Apolo at Universidad EAFIT (http://www.eafit.edu.co/apolo) to conduct the research reported in this scientific product.

References