# Weather Type Classification Using Neural Networks and Logistic Regression
**FYS-STK4155 - Project 3**

Martine Jenssen Pedersen, Sverre Johansen & Kjersti Stangeland
*University of Oslo*
(Dated: December 14, 2025)

Weather forecasting has improved significantly over recent decades due to advances in numerical modeling and increased computational power. As computational resources and runtime increasingly limit further progress, machine learning (ML) has emerged as a complementary approach with the potential to enhance forecasting systems, a problem of clear societal importance. In this study, we investigate how ML models can support meteorologists by providing simple and intuitive weather type classifications. We train logistic regression models and fully connected neural networks on a weather type dataset and evaluate their ability to classify four categories: *Sunny*, *Cloudy*, *Snowy*, and *Rainy*. Numerical experiments were conducted using extensive hyperparameter searches. Our main findings show that optimized logistic regression models achieve an accuracy of 0.860 for the self-implemented configuration, outperforming several of the neural network setups. For the deep neural networks, the highest accuracy of 0.914 was obtained using a PyTorch model with optimized architecture and regularization. This implies that relatively simple ML models can perform competitively on weather type classification tasks, and that increased model complexity does not necessarily guarantee improved performance.

## I. INTRODUCTION

Weather forecasts are approximations of a highly nonlinear and chaotic natural system. These approximations still offer a lot of uncertainty even though they have improved significantly over the years [1]. Forecasting has shown to be important for many areas of society, offering planning possibilities for both industry and private life, societal hazard safety, and optimization of aviation patterns, among plenty of other areas [2]. Classifying weather conditions can therefore be a valuable tool for providing fast and easily understandable forecasts to the general public.

Traditional weather models can be computationally demanding and therefore be inefficient for this task [1]. Trained machine learning models, on the other hand, can provide fast and inexpensive classification predictions from high-dimensional data, thereby providing the general public with sufficient information to plan and optimize. Machine learning models has become widely popular within the climate sciences because of the vast amounts of data collected over the years, as well as the need for computationally less demanding alternatives to the traditional models. Some researchers are hesitant to employ these models as they offer little physical consistency and are difficult to interpret [3]. In comparison, weather classification is strictly a performance task with physical understanding having a lower priority.

In our analysis, we therefore focus on the classification of weather types. By providing forecasts in broader strokes, using four simple weather type classes, we can offer the public fast and reliable insights into the current weather situation. For this purpose, we utilize both custom made and PyTorch-based models for classification through feed-forward neural networks (FFNNs) and logistic regression. These models were chosen because they perform well on independent data. The focus will be on analyzing how changes in hyperparameters and activation functions affect model performance in the classification task. We will use the Adam algorithm together with stochastic (mini-batch) gradient descent (SGD), which is preferred for its efficiency on large datasets such as the one used for this study [4, 5]. Additionally, we evaluate the predictions of the optimized models using heatmaps, confusion matrices, and ROC curves to assess potential occurrences of systematic errors in the models.

The following sections of the report are organized as follows. In Section II, we describe the theoretical framework of neural networks (NNs) and logistic regression, and outline the mathematical foundation of the project. Section III presents the methods and implementation details, while Section IV reports our main findings and discusses their implications. We provide a brief conclusion and outlook in Section V. Additional results are provided in Appendix **??**. All code developed for this report is available in our GitHub repository.[1]

## II. THEORY

### A. Feed-Forward Neural Networks

The central idea in a feed-forward neural network is to transform linear combinations of the input features through given activation functions, thereby modeling the target as a nonlinear function of the features [6]. In this project, we are studying a classification task, where the primary goal is to identify the classes to which unseen samples belong [4]. Since the outputs take on discrete values, we model the data as being generated from an underlying probability distribution over the classes [4].

[1] https://github.com/kjes4pres/Project_3_FYSSTK

The architecture of the network defines the complexity of the model and is designed to find the best model representation for the given data [4]. This architecture is composed of a given number of hidden layers in the network, an arbitrary number of nodes in each layer, and the activation functions for the layers [4].

The unknown parameters in the FFNN model are represented by weights, $W$, and biases, $\boldsymbol{b}$. Each weight is associated with the connection between two nodes [4]. In a fully-connected FFNN, each node in a hidden layer is connected to every node in the foregoing and subsequent layer [4, 5]. Consequently, the number of weights grows rapidly with network size. In addition, a bias term is added to each node in a layer, ensuring that nodes can activate even when the weighted input sum is zero, giving the model more flexibility. Thus, the $l$-th layer has a set of parameters $\boldsymbol{\beta}^{(l)} = (W^{(l)}, \boldsymbol{b}^{(l)})$, where $W^{(l)}$ is a matrix and $\boldsymbol{b}^{(l)}$ is a vector.

Two steps are needed in training of the FFNN: a feed-forward step and a back-propagation step. The latter is described in section II D. The completion of these two steps make up one iteration in which $\boldsymbol{\beta}$ is updated. The feed-forward step goes as follows. The inputs to one layer, $l$, is equal to the outputs from the preceding layer and is denoted $\boldsymbol{a}^{(l-1)}$. These are put together as a linear combination using the weights and bias as $\boldsymbol{z}^{(l)} = \left[W^{(l)}\right]^T \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$, and then fed into the activation function for that layer, denoted by $\sigma^{(l)}$: $\boldsymbol{a}^{(l)} = \sigma^{(l)}(\boldsymbol{z}^{(l)})$. The resulting vector, $\boldsymbol{a}^{(l)}$, is the inputs for the next layer [4, 5].

### B. Cost function

The cost function is the function we want the model to minimize. Since we are studying a classification task, the Categorical Cross-Entropy (CCE) cost function will be used

$$\boldsymbol{C}_{\text{CCE}}(\tilde{\boldsymbol{y}}) = -\frac{1}{n}\sum_{i=1}^{n} \boldsymbol{y}_i^{\text{T}} \log(\tilde{\boldsymbol{y}}_i)\,,$$

where $n$ is the number of data points, $\tilde{\boldsymbol{y}}$ are the model predictions and $\boldsymbol{y}$ are the targets.

The addition of two different regularization terms given by the $L_1$ and $L_2$ norms, which penalizes the parameters in the model, is also explored. This is done by adding one of the two following terms to the cost function: $L_2 = \lambda||W||_2^2$ or $L_1 = \lambda|W|_1$, where $\lambda$ is the regularization parameter and $W$ are the weights.

In this project, this cost function will only be used in combination with the Softmax activation function for the output layer (see section II C), and thus may be referred to as the Softmax Cross-Entropy function [4].

The cost function derivatives are needed for the back-propagation algorithm. The derivative of the CCE has a simple form when combined with the derivative of the Softmax function [7]:

$$\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{z}} = \frac{1}{n}(\tilde{\boldsymbol{y}} - \boldsymbol{y}).$$

The derivatives of the penalty terms are given by:

$$\frac{\partial L_2}{\partial W} = 2\lambda W, \quad \frac{\partial L_1}{\partial W} = \lambda\,\text{sign}(W).$$

### C. Activation functions

The activation function is a crucial part of the neural network's architecture as it introduces nonlinearity into the model. In this report, we consider three commonly used activation functions: Sigmoid, ReLU, and Leaky ReLU (LReLU), defined as follows:

$$\sigma_{\text{Sigmoid}}(z) = \frac{1}{1+e^{-z}},$$

$$\sigma_{\text{ReLU}}(z) = \begin{cases} 0, & z \leq 0 \\ z, & z > 0 \end{cases},$$

$$\sigma_{\text{LReLU}}(z) = \begin{cases} \alpha z, & z \leq 0 \\ z, & z > 0 \end{cases},$$

where $\alpha$ is a small positive constant, set here to $10^{-2}$.

A common challenge in modern machine learning is the vanishing or exploding gradient problem [4]. ReLU alleviates this problem to some extent, but suffers from the "dying ReLU" problem, where some nodes output zero for all inputs and effectively become inactive [4]. The purpose of the LReLU function is to mitigate this problem by avoiding such dead nodes [4]. Again, derivatives are needed for back-propagation and are as follows:

$$\frac{\partial \sigma_{\text{Sigmoid}}(z)}{\partial z} = \sigma_{\text{Sigmoid}}(z)\left[1 - \sigma_{\text{Sigmoid}}(z)\right],$$

$$\frac{\partial \sigma_{\text{ReLU}}(z)}{\partial z} = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases},$$

$$\frac{\partial \sigma_{\text{LReLU}}(z)}{\partial z} = \begin{cases} \alpha, & z \leq 0 \\ 1, & z > 0 \end{cases}.$$

Activation functions can vary between layers. Here, we use one for the hidden layers and one for the output layer. The output activation function allows the network to perform a last transformation of the data. When the network acts as a $K$-class classifier, the output layer has $K$ nodes, one for each class and each representing the probability of the input being in the corresponding class [6]. In this case, the output activation function is the Softmax function given by

$$\sigma_{\text{Softmax}}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}}, \quad i = 0, \dots, K-1,$$

where $i$ is the class index [4, 8]. The outputs are positive and sum to one, consistent with probability theory [4, 8]. As noted before, the derivative of the Softmax function is combined with the derivative of the CCE and can be found in section II B.

### D. Back-propagation

Gradient descent methods represent the generic approach for optimizing the cost functions, where each iterative step brings us closer to a minimum [5]. In the setting of NNs, this approach is referred to as back-propagation and is one of the most central concepts in deep learning [6].

The gradients required for the learning process can readily be derived by the chain rule of differentiation [6], together with the derivatives of cost and activation functions described in sections II C and II B.

The following derivations are based on course lectures for FYS-STK4155 at the University of Oslo [4].

The gradients of the cost function with respect to the model parameters in the output layer are computed first:

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} = \delta_i^{(L)} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}},$$

$$\frac{\partial C}{\partial b_i^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} = \delta_i^{(L)} \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}},$$

where $L$ denotes the output layer and

$$\delta_i^{(L)} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \frac{\partial C}{\partial z_i^{(L)}}$$

represents the output error. Furthermore, since $z_i^{(L)} = \sum_{j=0}^{N^{(L-1)}-1} w_{ij}^{(L)} a_j^{(L-1)} + b_i^{(L)}$, where $N^{(L-1)}$ is the number of nodes in the preceding layer, it follows that $\frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} = a_j^{(L-1)}$ and $\frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} = 1$. Thus, the gradients for the output layer are

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \delta_i^{(L)} a_j^{(L-1)},$$

$$\frac{\partial C}{\partial w_i^{(L)}} = \delta_i^{(L)}.$$

Then, the error is propagated backwards through the hidden layers as follows:

$$\delta_i^{(l)} = \frac{\partial C}{\partial z_i^{(l)}} = \sum_{k=0}^{N^{(l-1)}-1} \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}$$

$$= \sum_{k=0}^{N^{(l-1)}-1} \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = \sum_{k=0}^{N^{(l-1)}-1} \delta_k^{(l+1)} w_{ki}^{(l+1)} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}},$$

where $l$ is any hidden layer and $\frac{\partial a_i^{(l)}}{\partial z_i^{(l)}}$ is the derivative of the activation function in layer $l$.

Finally, the gradient descent updates are given by

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \delta_i^{(l)} a_j^{(l-1)}, \quad b_i^{(l)} \leftarrow b_i^{(l)} - \eta \delta_i^{(l)},$$

where the learning rate, $\eta$, schedule is set by the Adam algorithm described in section II F.

### E. Stochastic gradient descent

The theory behind stochastic gradient descent (SGD) lies in the assumption that the true data set can be understood as a sum of subsets, and that the cost and its derivative follows that same assumption, as shown in equation 1. This means that model only processes batches of the dataset at each iteration, introducing randomness with randomizing the batch chosen. The number of subsets are defined as the number of samples in the data, n, divided by the batch size, M.

$$\nabla_\theta C(\theta) = \sum_{i=0}^{n-1} \nabla_\theta c_i(\boldsymbol{x_i}, \theta) \tag{1}$$

This makes training with SGD computationally more efficient and can prevent memory errors that occur when using the whole dataset each update. It can also prevent getting stuck in local minima as the derivative will be evaluated at different localities in the cost landscape [4].

### F. The Adam optimizer

Adam optimization is a gradient-based algorithm for minimizing the cost function using an adaptive learning rate. The parameters, $\boldsymbol{\beta}$, are updated iteratively using the following update rule:

**Algorithm 1** The Adam algorithm

```
Require: $\eta$, $\rho_1$, $\rho_2$, $\delta$
for each iteration, i, do
    Compute the cost gradient, $\boldsymbol{g}^{(i)}$
    Update time: $t^{(i+1)} = t^{(i)} + 1$
    Calculate 1st moment variables:
        $$\boldsymbol{s}^{(i+1)} = \rho_1 \boldsymbol{s}^{(i)} + (1-\rho_1)\boldsymbol{g}^{(i)}$$
    Calculate 2nd moment variables:
        $$\boldsymbol{r}^{(i+1)} = \rho_2 \boldsymbol{r}^{(i)} + (1-\rho_2)\boldsymbol{g}^{(i)} * \boldsymbol{g}^{(i)}$$
    Correct bias in 1st moment:
        $$\hat{\boldsymbol{s}}^{(i+1)} = \frac{\boldsymbol{s}^{(i+1)}}{1-\rho_1^{t^{(i+1)}}}$$
    Correct bias in 2nd moment:
        $$\hat{\boldsymbol{r}}^{(i+1)} = \frac{\boldsymbol{r}^{(i+1)}}{1-\rho_2^{t^{(i+1)}}}$$
    Calculate effective learning rate:
        $$\eta_e = \frac{\eta \hat{\boldsymbol{s}}^{(i+1)}}{\sqrt{\hat{\boldsymbol{r}}^{(i+1)}} + \delta}$$
    Update parameters: $\boldsymbol{\beta}^{(i+1)} = \boldsymbol{\beta}^{(i)} - \eta_e$
```

where $\boldsymbol{g}^{(i)}$ is the gradient of $\boldsymbol{C}$ with respect to $\boldsymbol{\beta}$ at the $i$-th iteration, $\eta$ is the global learning rate, the different $\rho$ are decay rates, and $\delta$ is a small number used to avoid division by zero, here set to $\delta = 10^{-7}$ [4, 9].

### G. Evaluation metrics

The quality of the classifier is gauged using the accuracy score, a measure of how many of the samples the model correctly puts in its true class and has the mathematical form

$$\text{Accuracy}(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = \frac{1}{n}\sum_{i=0}^{n-1} \mathbb{I}(y_i = \tilde{y}_i),$$

where $\mathbb{I}$ is the indicator function, equal to 1 of the condition $y_i = \tilde{y}_i$ is true and 0 otherwise.

### H. Logistic regression

Logistic regression is a linear classifier [4]. It can be viewed as a special case of a feed-forward neural network with no hidden layers [4, 5]. In this setting, the only activation function is the Softmax function at the output layer, so the model remains linear in the input features.

### I. Confusion Matrix

The confusion matrix visualizes a classification models performance by tabulating the predicted data and the true data together. This gathers together how many of the prediction are right and wrong, and splits it into subsets of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN), for each class. The true positives tell us how many of the predictions are right per class. False positives tell us how many times a predicted class is labeled as something else in the true data. False negatives tell how many times a true label is predicted as something else, and the true negatives are counted as the sum of all samples minus TP, FP, and FN [4].

The true positive rate (TPR) gives us the rate at which the model is able to predict a class:

$$TPR_i = \frac{TP_i}{TP_i + FN_i},$$

where $i$ is the class iteration. The false positive rate (FPR) indicates how often a model predict a class, labeled in the true data as something else:

$$FPR_i = \frac{FP_i}{FP_i + TN_i}.$$

### J. ROC curve

Another way of visualizing classification performance is the Receiver Operating Characteristic (ROC) curve. Formally, an ROC curve plots the FPR on the x-axis against the TPR on the y-axis for a given class label. The curve is generated by varying the decision threshold of the classifier [4]. As the threshold is lowered, more samples are classified as positive, which increases both TPR and FPR. This produces a continuous curve that illustrates the trade-off between correctly detecting positive cases and incorrectly labeling negative cases as positive.

The Area Under the ROC Curve (AUC) provides a single summary metric of the classifier's ability to distinguish between classes. A model that predicts at random will produce a diagonal ROC curve with $AUC = 0.5$, whereas a perfect classifier will have $AUC = 1$ [4].

## III. METHODS

### A. Dataset

In our numerical experiments we use an artificial dataset on weather types retrieved from https://www.kaggle.com/datasets/nikhil7280/weather-type-classification/data [10]. The dataset is artificial, but made to mimic real life weather data, including outliers. The author of the dataset states that its purpose is for students and other practitioners to explore classification algorithm performance and outlier detection, among other data analysis and machine learning objectives [10].

The dataset contains 10 meteorological features and 13 200 samples. Each sample is assigned a weather type label, either *Rainy*, *Snowy*, *Sunny*, or *Cloudy*. The distribution of the four weather can be seen in Figure 1.

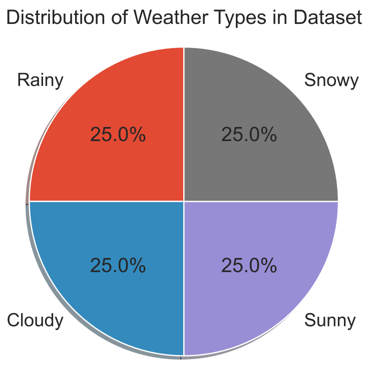


FIG. 1. Distribution of the four weather types in the *kaggle* dataset [10].

The distribution of the features can be seen in Figure 11 in Appendix A. As three of the features (cloud cover, season, location) and the weather class labels are categorical values, we encode the categories to numerical values.

## B. Implementation

In this project, we use two types of neural networks: a feed-forward neural network (FFNN) implemented in Project 2 [11], and a neural network built using PyTorch functionalities [12]. Both frameworks are applied to logistic regression and to full multi-layer networks. For logistic regression, the models consist of a single layer - the output layer - with Softmax activation, and we evaluate their performance across different learning rates, epoch sizes, batch sizes, and $L_1/L_2$ regularization strengths. For the full NN experiments, we additionally vary the network architecture by adjusting the number of hidden layers, the number of nodes per layer, and the choice of activation function in the hidden layers, while the output layer uses Softmax activation in all cases.

All experiments use an 80/20 train–test split using `train_test_split` from Scikit-Learn [13]. The models are trained using stochastic gradient descent with the Adam optimizer. Regularization in the PyTorch models is implemented following the approach of [14].

Scaling is a standard pre-processing step to prevent features with large numerical ranges from dominating those with smaller ranges and to reduce sensitivity to outliers [4]. It is also beneficial for gradient-based optimization, as it increases the likelihood of finding a learning rate that works uniformly well across all parameters [5]. In this project, the input features are normalized to the interval $[0, 1]$ using the `MinMaxScaler`, while the targets are left unscaled because they represent categorical class labels.

### *1. Logistic regression*

The behavior of the logistic regression model is studied first and the search for the optimal hyperparameters is done in two parts: we first test the model with different learning rates, $\eta \in [10^{-4}, 10^{-3}, 0.01, 0.1]$, and regularization parameters, $\lambda \in [10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}]$, for all three regularization types simultaneously, before inheriting the optimal values and testing different batch sizes, $[16, 32, 64, 128, 256]$, and number of epochs, $[20, 50, 100, 200, 500]$. In the first part of training, the batch size is set to 64 and the number of epochs to 50. The same steps are done for both our self-implemented FFNN and for the network implemented using the functionalities of PyTorch. As there are no hidden layers in the logistic regression model, there is no need to consider the model architecture or activation function.

### *2. Neural Networks*

In addition to logistic regression, we experiment with different NNs. We compare our own built FFNN from Project 2, hereafter referred to as *FFNN*, with a NN built with PyTorch functionalities. The comparison spans over three different experiment types. In the first round, we build 27 models with varying number of hidden layers and number of nodes, for Sigmoid, ReLU and LReLU activation functions for both FFNN and Pytorch NN. We test models with 1, 2, 3, and 4 hidden layers and 16, 32, 64, and 128 nodes per layer. In this first round of experiments all models were ran with a learning rate of 0.001, batch size of 32, and 50 epochs.

By assessing the best accuracy score when classifying on the test set, we find the best model architectures in terms of layers and nodes per activation function and per model type. For the second experiment, we therefore inherit the best architectures and train the models with varying learning rate and epoch sizes. We test $\eta \in [10^{-4}, 10^{-3}, 0.01, 0.1, 0.2]$ and epochs $[20, 50, 100, 200, 500]$.

For the third experiment, we again keep the model configurations which yielded the highest accuracy score on the test set, and keep the parameters in terms of learning rate and epochs per activation function. Then, we add regularization and test how $L_1$ and $L_2$ regularization affects model accuracy with varying values of hyperparameter, $\lambda \in [10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}]$.

### C. Software

Our implementation relies heavily on several open-source Python packages. Our FFNN is primarily built using NumPy for efficient vectorized operations [15]. We compare our FFNN to a network built with PyTorch [12]. In both cases we use Pandas [16] for opening the CSV file. Scikit-Learn is used for scaling and splitting the data, and encoding categorical values [13]. Matplotlib [17], and Seaborn [18] are used to generate figures.

All results and figures can be reproduced using the codes supplemented in the GitHub repository. Here, there are also additional figures that were not included in the final report.

### D. Use of AI tools

Artificial Intelligence (AI) tools, namely ChatGPT [19], has been used for parts of this project. The usage has been limited and primarily used for debugging code, formatting of figures, making descriptive documentation strings for functions, and grammatical help when writing the report. All usage have been quality checked by the authors.

## IV. RESULTS AND DISCUSSION

### A. Logistic Regression

The results from the first part of the training is presented in figures 2 and 3. For our own network, the model is nearly indifferent to the regularization parameter, displaying only minor variations in performance. Regularization does not improve the results; the lowest penalty, $\lambda = 10^{-6}$, performs equally well as with no regularization and performance worsens as the penalty increases. In contrast, the learning rate strongly influences the convergence speed and stability of the optimization process [9]. If the learning rate is too high, the model may overshoot the minimum and oscillate, making the results unreliable, whereas a too low learning rate can lead to incomplete convergence [20]. The best performance is obtained with a moderate learning rate of $\eta = 0.01$.

The PyTorch model exhibits some of the same behavior, but with notable differences. As the learning rate increases, the model's accuracy becomes more sensitive to the regularization parameter. Accuracy consistently improves for lower penalties, and the unregularized model continues to outperform the regularized ones. The PyTorch implementation achieves higher accuracy at the lowest learning rate than our own network. Moreover, its best results occur at the highest learning rate, $\eta = 0.1$, inconsistent with the optimal value found for our model. This suggests that the PyTorch network is more robust to both smaller and larger learning rates than our self-implemented network.

The optimal learning rates and regularization parameters identified in the first part of training for both networks are carried over into the second part. The results for the self-implemented network are shown in figure 4. The model performs well overall, with the lowest test accuracy in the heatmap being 0.8265. The best performance occurs at moderate batch sizes and numbers of epochs. Large batches combined with few epochs yield the lowest scores, likely because the model parameters get very few updates during training. Conversely, small batches paired with many epochs also reduce performance, suggesting that the model has undergone too much training and thus overfits. Consistent with the earlier results, the model essentially produces identical results across the regularization types, which is expected given the very small penalty used in this experiment ($\lambda = 10^{-6}$).

The pattern is somewhat different for the PyTorch network, as shown in figure 5. While a general decline in performance is still observed for both few and many training epochs, the PyTorch model achieves its best results with large batch sizes. This could be linked to the higher learning rate used in the PyTorch model, as larger batches produce smoother gradients, preventing the instability that would otherwise occur with noisy gradients at high learning rates.

Based on these results, we conclude that the best-performing model of our self-implemented FFNN used a learning rate of 0.01, batch size of 32, no regularization, and 100 epochs of training, achieving a final test accuracy of 0.8602. The corresponding optimal PyTorch model used a learning rate of 0.1, batch size of 256, no regularization, and 50 training epochs, and obtained an accuracy of 0.8598. Overall, our own network therefore performed marginally better than PyTorch's.

To visualize the behavior of the best-performing models, we plot the confusion matrices and ROC curves using Scikit-Learn. Because the confusion matrices are normalized, each row sums to 1 and represents the fraction of true samples in each class that are predicted as each of the possible classes [4]. Figure 6 shows these plots for the best models obtained with our own network and with the PyTorch implementation, respectively. The models correctly classifies samples to great accuracy, resulting in good performance across all classes, particularly for the *Snowy* class. The underlying connection between the features of this class may be more easily interpreted by the models than the other classes. In contrast, the *Cloudy* class proves most challenging for both models to capture the underlying relationship of. This is potentially due to outliers or complex feature relationships, though verifying the cause would require further inspection of the features themselves. Both confusion matrices show that the models most frequently confuses *Cloudy* with *Rainy*,

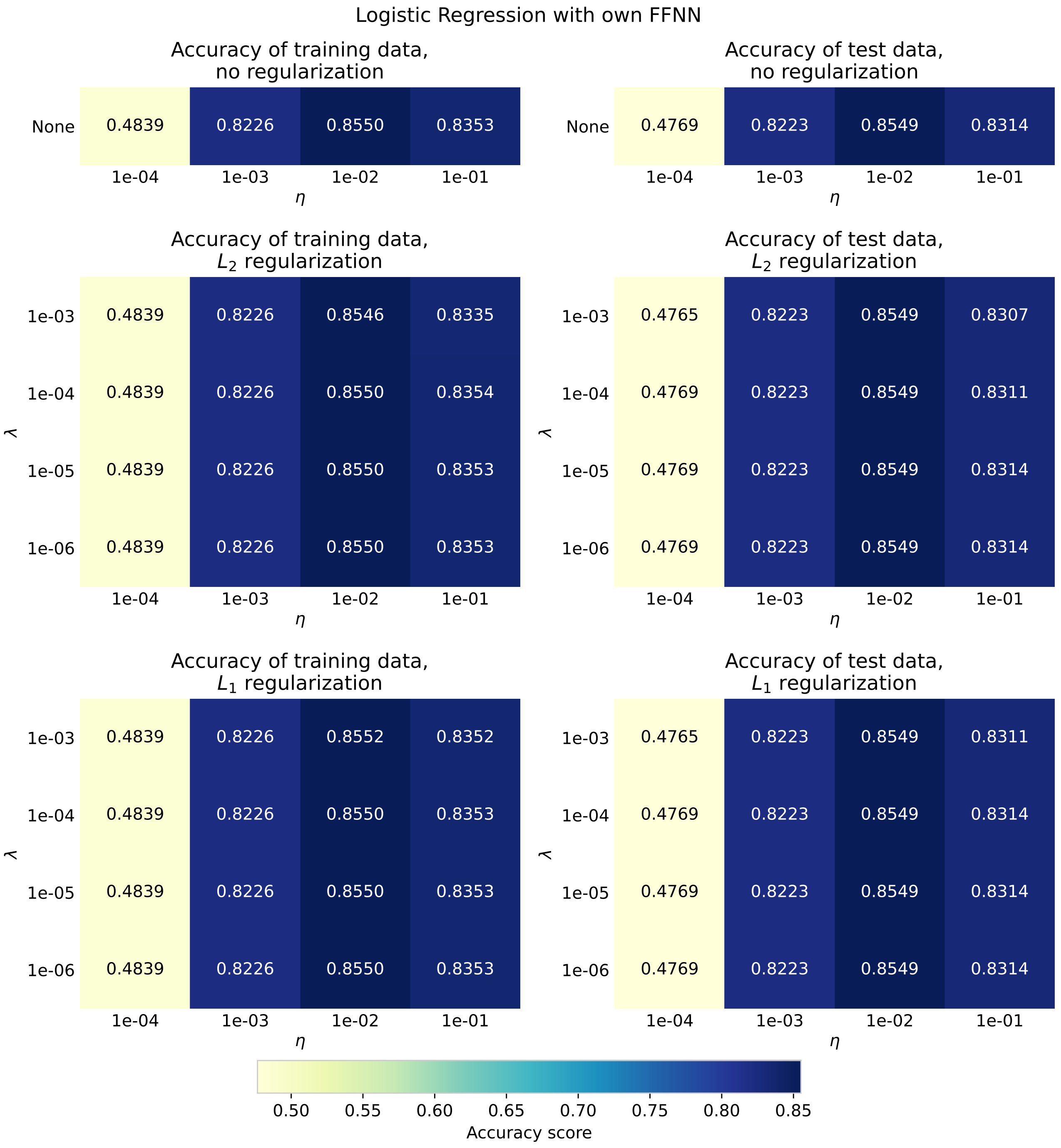


FIG. 2. Logistic regression with different regularization techniques for the self-implemented FFNN. Accuracy score of training (left) and test data (right) with no regularization (top), $L_2$ regularization (middle), and $L_1$ regularization (bottom). Results are shown as functions of learning rate, $\eta$, and regularization parameter, $\lambda$. The best-performing models are found at low regularization and moderate learning rates.

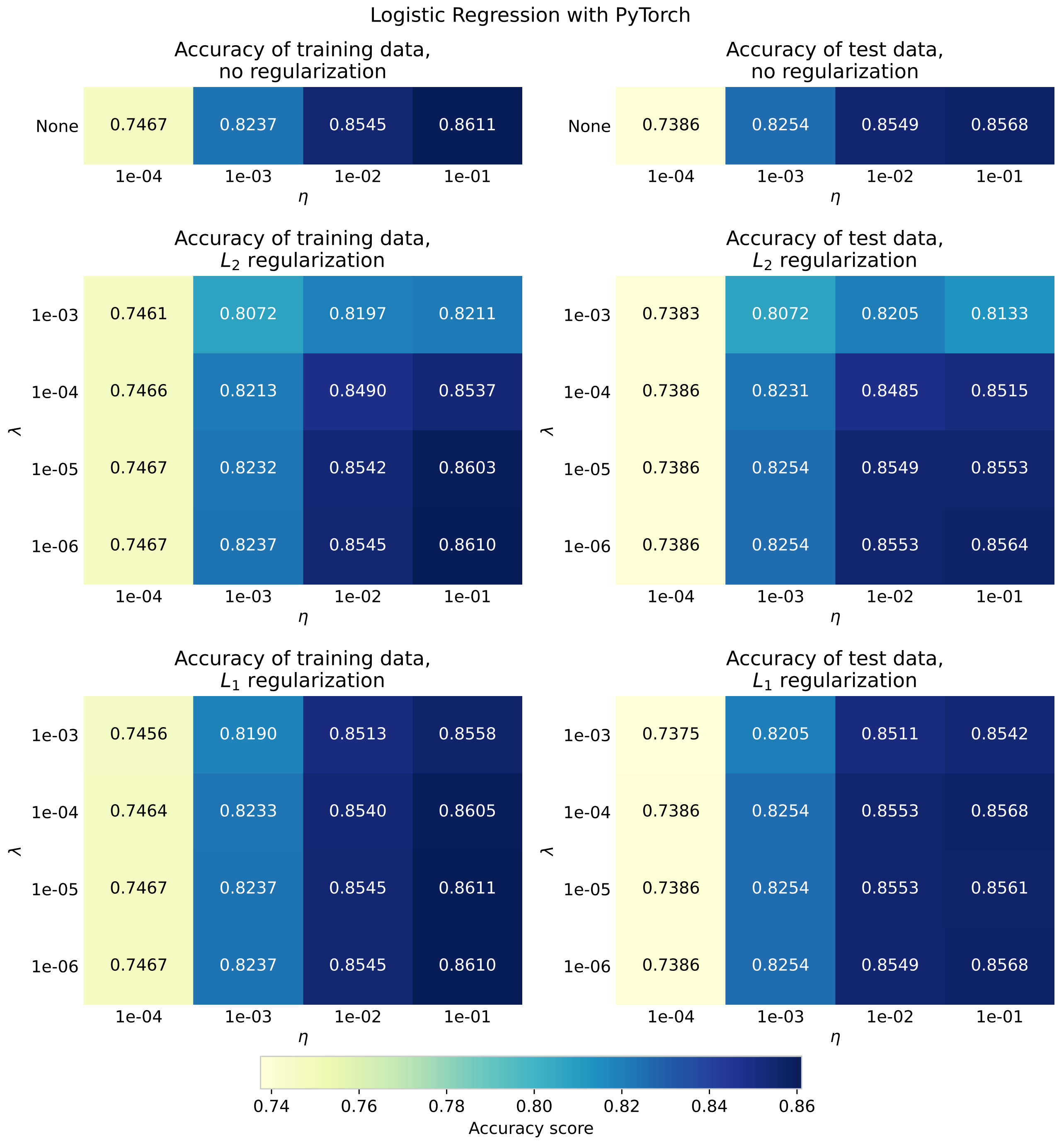


FIG. 3. Logistic regression with different regularization techniques for the FFNN using PyTorch. Accuracy score of training (left) and test data (right) with no regularization (top), $L_2$ regularization (middle), and $L_1$ regularization (bottom). Results are shown as functions of learning rate, $\eta$, and regularization parameter, $\lambda$. The best-performing models are found at low regularization and high learning rates.

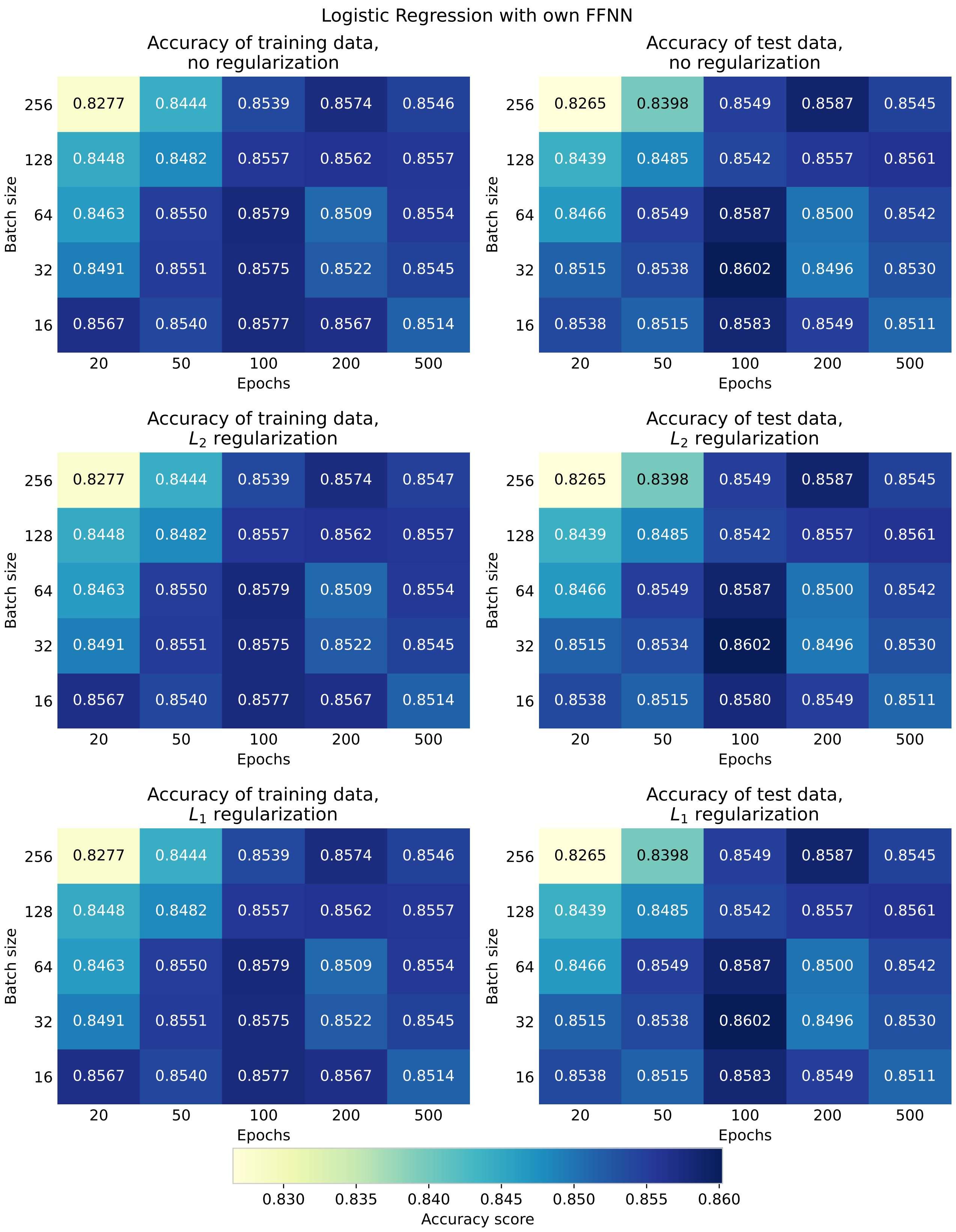


FIG. 4. Logistic regression with different regularization techniques for the self-implemented FFNN. Accuracy score of training (left) and test data (right) with no regularization (top), $L_2$ regularization (middle), and $L_1$ regularization (bottom). Results are shown as functions of number of epochs and batch size. The best-performing models are found at moderate batch sizes and after 100 epochs.

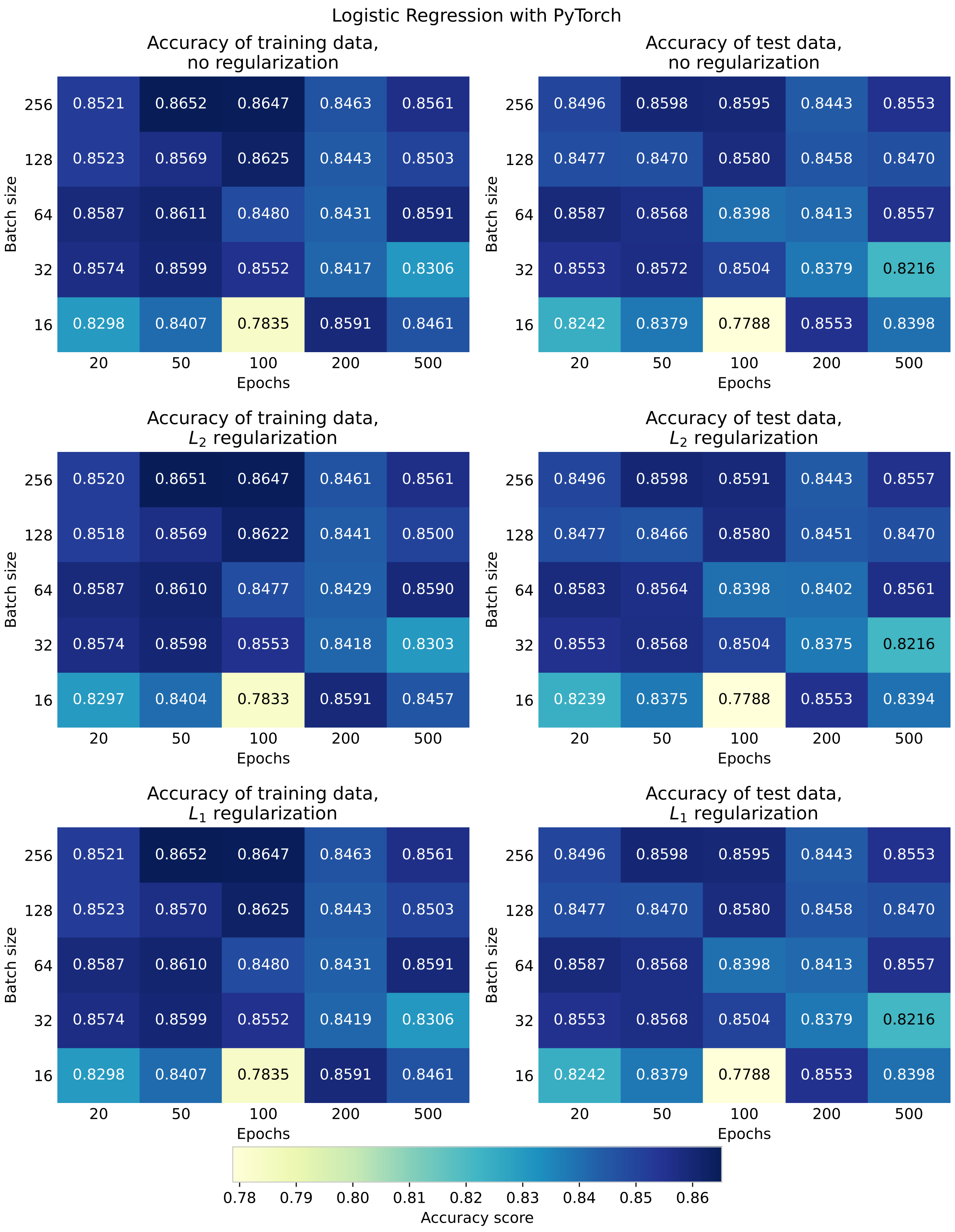


FIG. 5. Logistic regression with different regularization techniques for the FFNN using PyTorch. Accuracy score of training (left) and test data (right) with no regularization (top), $L_2$ regularization (middle), and $L_1$ regularization (bottom). Results are shown as functions of number of epochs and batch size. The best-performing models are found at large batch sizes and after around 50 epochs.

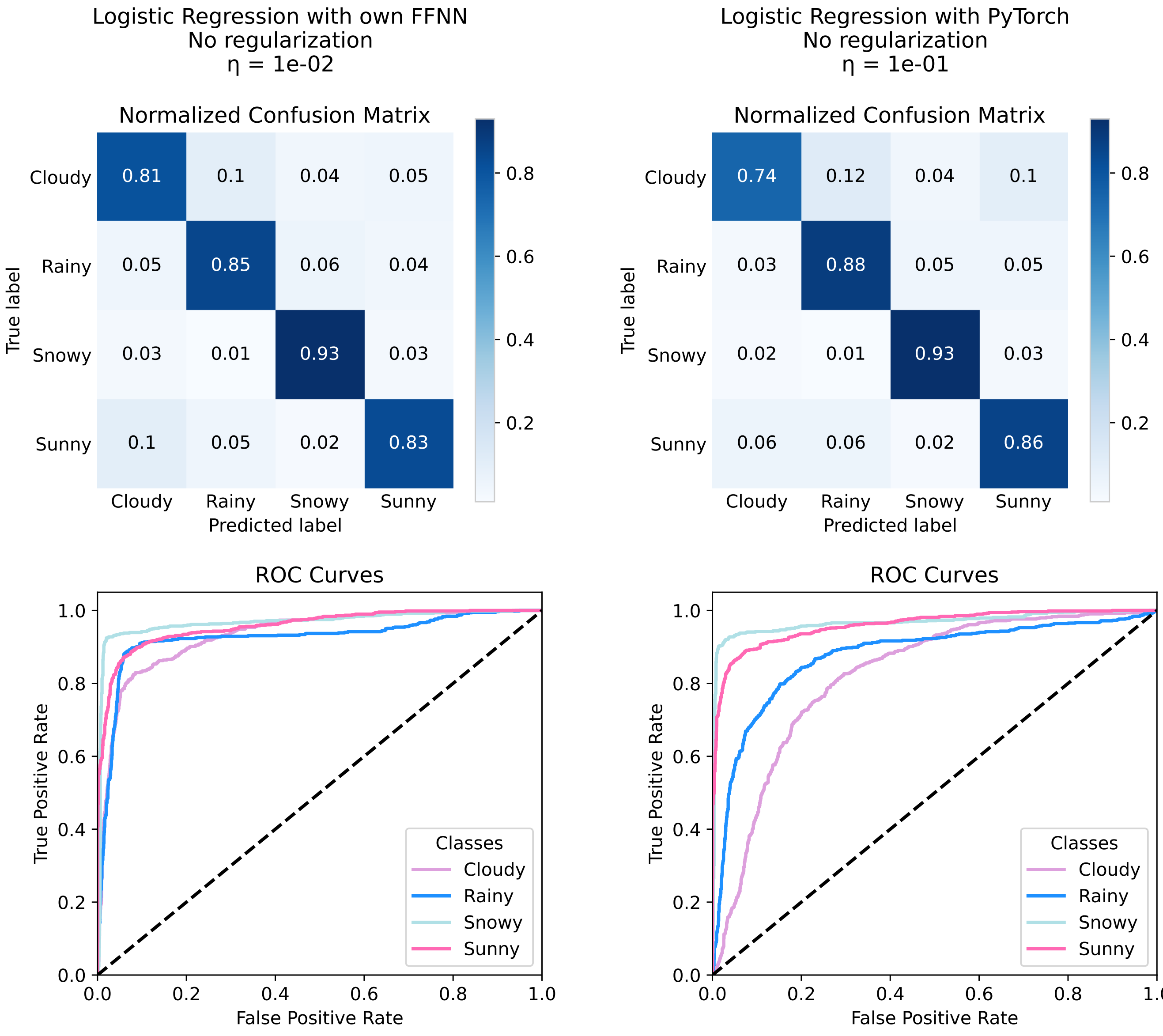


FIG. 6. Comparison of optimal FFNN (left) and PyTorch (right) models for logistic regression. Performance shown through normalized confusion matrices (top) and ROC curves (bottom). Left panels: The best-performing self-implemented logistic regression model with learning rate 0.01, batch size 32, no regularization, and 100 epochs. The model achieves strong diagonal dominance in the confusion matrix and very sharp initial increase for all ROC curves, indicating high classification accuracy across all classes. Total accuracy score = 0.8602. Right panels: the best-performing logistic regression model using PyTorch with learning rate 0.1, batch size 256, no regularization, and 50 epochs. Although this model also exhibits strong diagonal dominance and sharply rising ROC curves, the slope is somewhat less steep than for the FFNN model, resulting in a slightly lower total accuracy score of 0.8598.

which may reflect the fact that these weather types occur under more similar atmospheric conditions than the other classes considered. The same difficulty is also visible in the ROC curves: the curve for the *Cloudy* class is noticeably less steep than the others for both models. A steep curve indicates that the model correctly detect positives without making many false-positive errors early on. This suggests that the models struggle more to separate positive and negative samples for the *Cloudy* class. The PyTorch curves rise more slowly overall, suggesting that the model is somewhat less confident in distinguishing the different classes from each other. Despite these differences in behavior across the different classes, the overall test accuracies of the two best-performing models remain nearly identical.

In this study, hyperparameters were chosen based on their performance on the test data. However, this approach conflicts with the principle that test data is intended to remain untouched throughout the entire training process. Good practice is to divide the dataset into three subsets — training, validation, and test. The validation set should be used for hyperparameter tuning, while the test set is reserved for the final evaluation of the fully trained model. Because this was not done here, the selected parameters are indirectly influenced by the test data. As a consequence, the final test results are likely to exhibit optimistic bias and do not fully represent the model's performance on truly unseen data. This uncertainty factor is also present in the following analysis of deeper neural networks.

## B. Deeper feed-forward neural networks

### 1. *Network architectures*

As logistic regression provided an understanding of how batch size affect the performance, we used the previously identified optimal batch size as a standard for the upcoming analysis. With the introduction of hidden layers, we first examined how variations in depth, width, and activation functions influence the performance of both the PyTorch implementation and our custom-built network. Table I summarizes the optimal architectures, and reveals that the PyTorch model has a clear edge, consistently outperforming the self-implemented model.

Although PyTorch networks exhibited significantly longer runtime, they required substantially less time and effort to implement compared to the custom-built ones. It should also be noted that networks using the Sigmoid activation function, for both the custom FFNN and the PyTorch implementation, favored the largest tested number of nodes per, as shown in figure 7 and table I. This suggests that exploring even wider architectures could give further insights. However, such experiments were not conducted here due to the long runtime of the PyTorch models.

| Model | Activation | Layers | Nodes | Accuracy |
|---|---|---|---|---|
| FFNN | Sigmoid | 3 | 128 | 0.877652 |
| FFNN | ReLU | 3 | 16 | 0.863258 |
| FFNN | LReLU | 3 | 16 | 0.862879 |
| PyTorch | Sigmoid | 2 | 128 | 0.898106 |
| PyTorch | ReLU | 2 | 64 | 0.911364 |
| PyTorch | LReLU | 3 | 64 | 0.910227 |

TABLE I. The optimal configurations of number and layers and nodes per activation function for models built with FFNN and PyTorch. All models were ran with a batch size of 32, 50 epochs and learning rate 0.001. The reported accuracy is for classification on the test set.

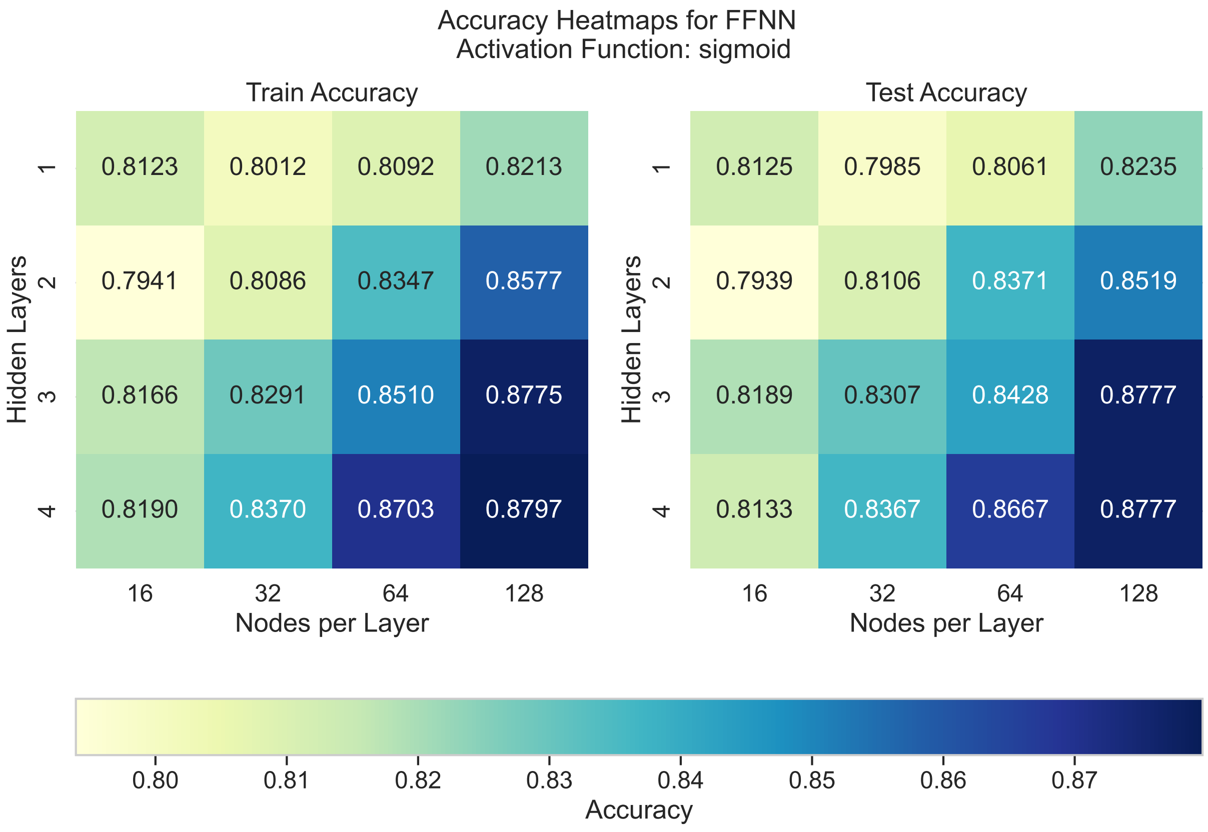
We've seen that the PyTorch model performs better overall, and a notable inconsistency between the two networks is found in their preferred activation functions. The Sigmoid activation yields the best performance in the FFNN, whereas the linear units outperform Sigmoid when using PyTorch. This could be due to the weight initialization of the FFNN not being specialized for any activation function, while the PyTorch network applies a generalized Kaiming scaling when initializing weights [21]. This weight initialization technique draws weight values from a normal distribution with location 0 and scale $2/n_{l-1}$, and [21] found this to improve networks employing linear units.

### 2. *Optimizing learning rate and epochs*

Next we analyze how learning rate and epochs affect the performance. We keep the optimal values from the last analysis. In table II we find the optimal values for $\eta$ and number of epochs for the different activation functions. We can see that the optimal FFNN need more epochs and employ a higher learning rate than the PyTorch setups to reach optimal levels, which may also be a consequence of the weight initialization. The efficiency differences between the PyTorch models and the FFNN shows that the PyTorch models are clearly more refined with respect to training.

| Model | Activation | Epochs | $\eta$ | Accuracy |
|---|---|---|---|---|
| FFNN | Sigmoid | 500 | 0.01 | 0.906061 |
| FFNN | ReLU | 200 | 0.1 | 0.898106 |
| FFNN | LReLU | 500 | 0.2 | 0.895833 |
| PyTorch | Sigmoid | 200 | 0.001 | 0.906061 |
| PyTorch | ReLU | 50 | 0.001 | 0.911364 |
| PyTorch | LReLU | 50 | 0.001 | 0.910227 |

TABLE II. The optimal configurations of number of epochs and learning rate for models built with FFNN and PyTorch. All models were ran with their optimal combination of layers and nodes reported in table I. The reported accuracy is for classification on the test set.



Train Accuracy

| Hidden Layers | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| 1 | 0.8123 | 0.8012 | 0.8092 | 0.8213 |
| 2 | 0.7941 | 0.8086 | 0.8347 | 0.8577 |
| 3 | 0.8166 | 0.8291 | 0.8510 | 0.8775 |
| 4 | 0.8190 | 0.8370 | 0.8703 | 0.8797 |

Test Accuracy

| Hidden Layers | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| 1 | 0.8125 | 0.7985 | 0.8061 | 0.8235 |
| 2 | 0.7939 | 0.8106 | 0.8371 | 0.8519 |
| 3 | 0.8189 | 0.8307 | 0.8428 | 0.8777 |
| 4 | 0.8133 | 0.8367 | 0.8667 | 0.8777 |

FIG. 7. Accuracy scores for FFNN models with varying nodes and layer sizes. All models were ran with a batch size of 32, 50 epochs and learning rate 0.001.

### 3. *Regularization*

As the weather class dataset contains known outliers [10], we applied $L_1$ and $L_2$ regularization to some of our models to see whether this improved accuracy. However, because we only split the dataset into a training set and a test set, without a separate validation set, the models were effectively optimized toward the test set. As mentioned before, this results in the models indirectly being trained on the test data, and thus adding regularization does not necessarily help the models generalize better, as intended. This represents a major pitfall of the analysis and limits the conclusions that can be drawn about the benefits of regularization.

This is seen in table III, where only half of the models see improvement with regularization when compared to the results without in table II. In addition, for the cases where regularization improved accuracy, the optimal regularization strength, $\lambda$, was found to be very small. This suggests that, without a proper validation set to tune hyperparameters, the regularization strength cannot be reliably optimized and thus only very small values appear beneficial.

| Model | Activation | Reg. | $\lambda$ | Accuracy |
|---|---|---|---|---|
| FFNN | Sigmoid | L1 | $10^{-6}$ | 0.906061 |
| FFNN | ReLU | L2 | $10^{-3}$ | 0.895076 |
| FFNN | LReLU | L1 | $10^{-6}$ | 0.896591 |
| PyTorch | Sigmoid | L2 | $10^{-5}$ | 0.909470 |
| PyTorch | ReLU | L1 | $10^{-6}$ | 0.910606 |
| PyTorch | LReLU | L2 | $10^{-5}$ | 0.914015 |

TABLE III. The optimal regularization scheme for different activation functions for NN models built with FFNN and PyTorch. All models were ran with their optimal combination of layers and nodes reported in table I, and the optimal number of epochs and learning rate reported in table II. The reported accuracy is for classification on the test set.

In the earlier experiments using logistic regression, the results differed from those obtained here with deeper NNs. Here, several models show improved accuracy with regularization, whereas none of the logistic regression models did. This difference may arise from the differing number of parameters in the respective models. For deep NNs, the number of model parameters grows rapidly with the total number of nodes. Thus, there are much fewer parameters in logistic regression than in the deeper networks. Regularization puts a penalty on the model parameters, reducing the effective degrees of freedom in the model [5]. With fewer parameters, the $L_1$ and $L_2$ penalties restrict the parameters, resulting in additional bias [4]. Deeper NNs are extremely flexible and may therefore overfit easily [6]. Here, regularization acts to reduce the effect of overfitting as the penalty shrinks the weights, thereby reducing the variance in the model [5, 6]. However, when the regularization parameter, $\lambda$, becomes large, underfitting occurs and model performance is worsened. $L_1$ regularization is very similar to those for $L_2$. Here, some parameters can be shrunk to zero, effectively reducing the number of active nodes in the network and thus simplifying the model [6]. The shrinkage in the two regularization methods behave similarly when $\lambda$ is not large enough to cause strong sparsity in the model parameters for $L_1$ regularization. Therefore, these models perform nearly identically in the present study.

As a supplement to section IV B 1, these results indicate that overly complex architectures often perform better than overly simple ones. A shallow network with few nodes may fail to capture nonlinearities in the data and underfit. On the other hand, in a deeper network, unnecessary parameters can be shrunk by utilizing appropriate regularization to reduce overfitting [6].

Figure 8 and 9 show the performance of the FFNNs and PyTorch NNs with LReLU activation and applied regularization. Both models achieve their highest training accuracy for very small values of $\lambda$, indicating that regularization begins to constrain the model quickly. For the FFNN, this pattern is most consistent for $L_1$ regularization, whereas the PyTorch model exhibits a sharper decline in accuracy for both $L_1$ and $L_2$ as $\lambda$ increases, suggesting that it is more sensitive to the regularization strength. In the FFNN heatmap, figure 8, the peaks in training and test accuracy coincide, implying that regularization affects both datasets similarly. In contrast, the PyTorch heatmap, figure 9, shows that the best test accuracy occurs at a slightly larger $\lambda$ than the best training accuracy, indicating that mild regularization improves generalization even if it reduces the training performance. This behavior suggests that the PyTorch implementation is more prone to overfitting in this configuration and therefore benefits more from small regularization values. Similar behavior was seen in the equivalent heatmaps both for ReLU and Sigmoid activation, where PyTorch models were more sensitive to regularization than FFNN.

### 4. *The best performing neural networks*

In contrast to the results of logistic regression, we find that in the case for full NN models, PyTorch outperform our own FFNN. This is in terms of a slightly higher test accuracy for the final optimal models, as seen in table IV. The optimized models preferred no or little regularization.

We plotted confusion matrices and ROC curves for the optimal models in Figure 10. The figures show the model performance on predicting the four different weather classes, and show that both models classify snowy weather the best, and rainy the worst. As the prediction of the classes is very similar for both models, one could argue that they perform almost equally well on finding the underlying connections between features and labels. As for the ROC curves, a very similar pattern is seen across the models.

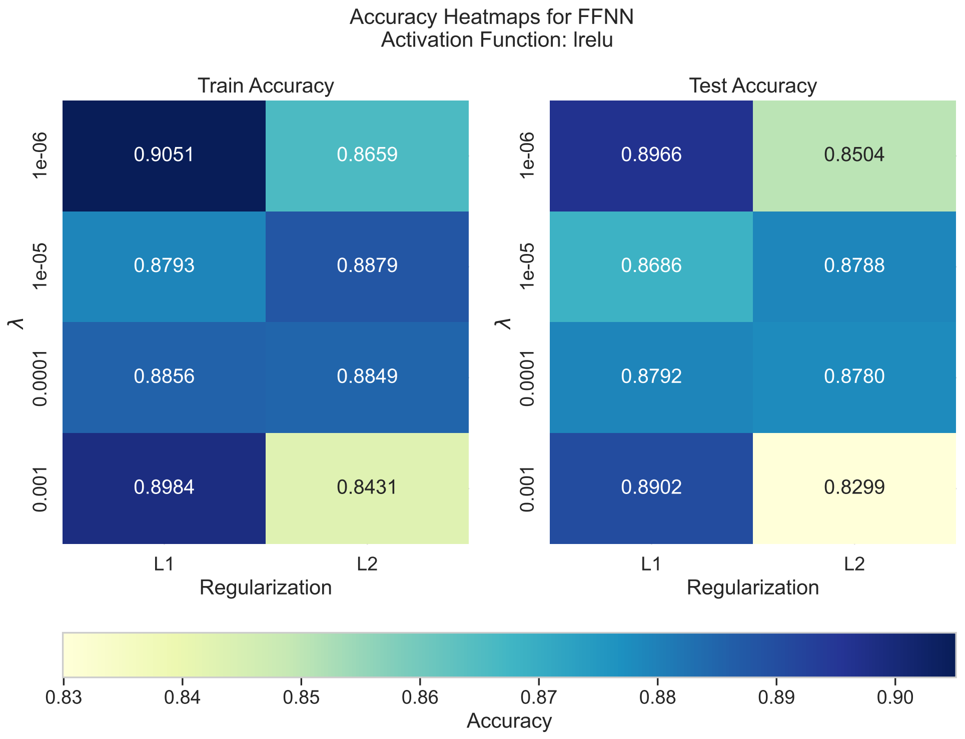


FIG. 8. Accuracy heatmap for FFNN models with LReLU activation, with added regularization. The model architecture, learning rate and number of epochs are the best reported for LReLU with FFNN in tables I and II.

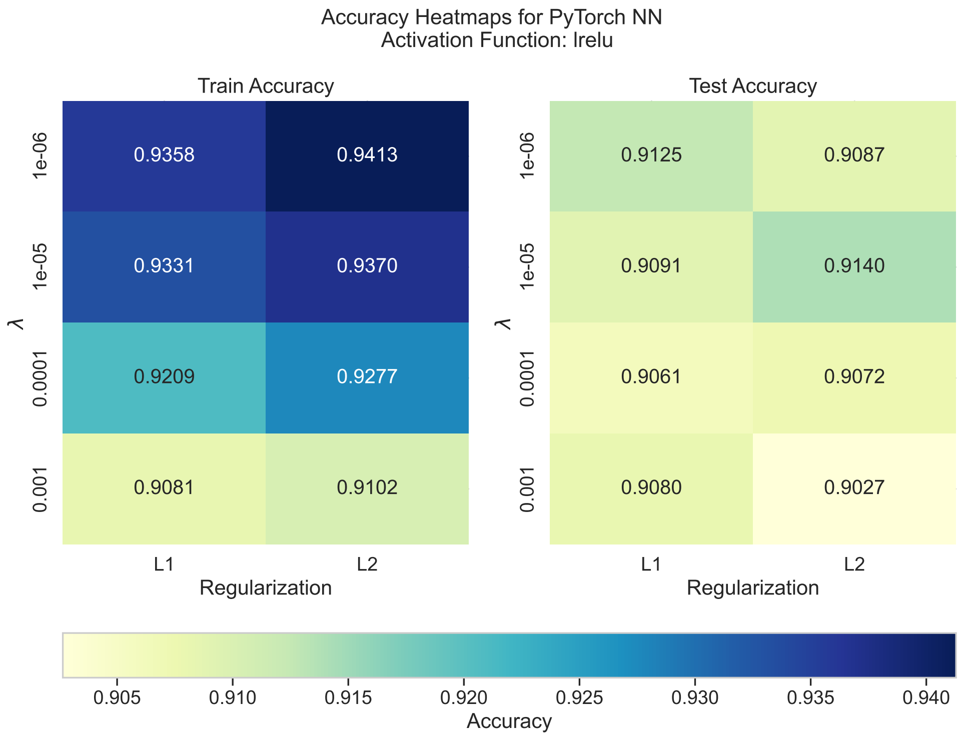


FIG. 9. Accuracy heatmap for PyTorch models with LReLU activation, with added regularization. The model architecture, learning rate and number of epochs are the best reported for LReLU with PyTorch in tables I and II.

| Model | Activation | Layers | Nodes | Learning Rate | Regularization | Best Test Acc. |
|---|---|---|---|---|---|---|
| FFNN | Sigmoid | 3 | 128 | 0.01 | None | **0.906061** |
| FFNN | ReLU | 3 | 16 | 0.1 | None | 0.898106 |
| FFNN | LReLU | 3 | 16 | 0.2 | L1 ($\lambda = 10^{-6}$) | 0.896591 |
| PyTorch | Sigmoid | 2 | 128 | 0.001 | L2 ($\lambda = 10^{-5}$) | 0.909470 |
| PyTorch | ReLU | 2 | 64 | 0.001 | None | 0.911364 |
| PyTorch | LReLU | 3 | 64 | 0.001 | L2 ($\lambda = 10^{-5}$) | **0.914015** |

TABLE IV. Best-performing FFNN and PyTorch models from hyperparameter search.

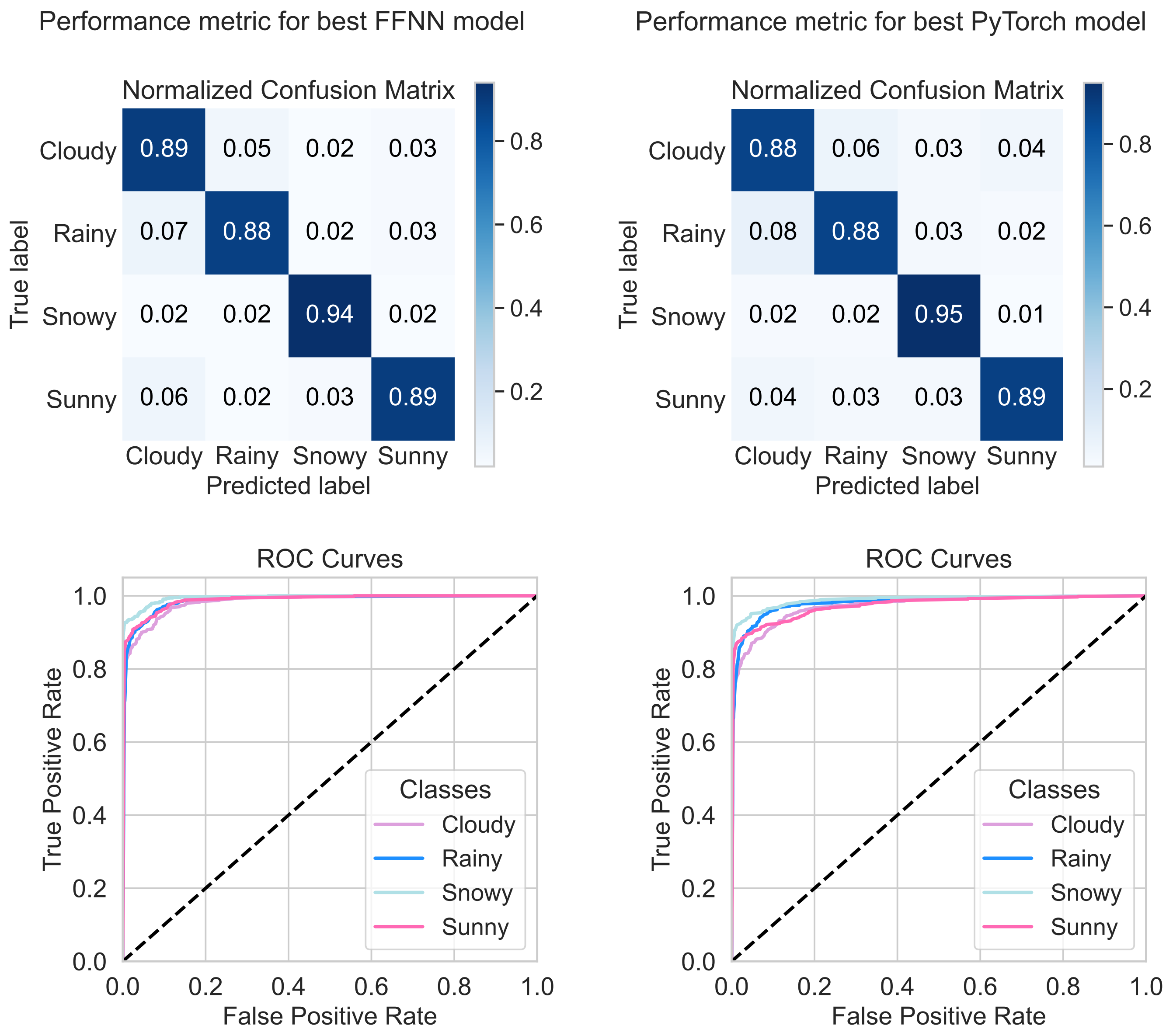


FIG. 10. Comparison of optimal FFNN (left) and PyTorch (right). Performance shown through normalized confusion matrices (top) and ROC curves (bottom).

## V. CONCLUSION

Overall, the behavior of the custom and PyTorch implementations was very similar in the logistic regression case, however, with our own model slightly outperforming PyTorch's. The hyperparameter search for the deeper NNs revealed several systematic differences from the logistic regression results. While the logistic regression models were largely insensitive to regularization and were mainly governed by the learning rate, the deeper networks were far more affected by architectural choices, activation functions, and the interaction between learning rate and number of epochs. The FFNN models tended to prefer wider and deeper architectures when using Sigmoid activation, whereas ReLU and LReLU favored smaller networks, likely reflecting saturation effects in Sigmoid layers and the "dying ReLU" problem for negative inputs to the ReLU units [4]. PyTorch models consistently achieved higher accuracy than our self-implemented networks across activation functions, and were generally more sensitive to regularization, suggesting stronger overfitting but also a larger benefit from mild regularization. These findings show that model complexity plays a central role in neural networks, and that implementation details, such as PyTorch's optimized initialization and training routines, can have a measurable impact on the final predictive performance.

Our study provides an extensive hyperparameter search for both FFNN and PyTorch implementations, but some limitations remain. The runtime of the PyTorch networks restricted the number of tested layer and node configurations, and the lack of a separate validation set may have biased the hyperparameter choices. We therefore recommend that future work include a validation set to obtain unbiased performance estimates, and also investigate the known outliers in the dataset more thoroughly. Despite these limitations, our results demonstrate that relatively simple ML models can classify weather types with satisfactory accuracy, and that deeper networks consistently outperform logistic regression for this task. Moreover, our custom FFNN achieved accuracies comparable to the more mature PyTorch implementation.

Future work could involve exploring even deeper or more advanced NNs or utilizing even more efficient optimizers to improve convergence and performance. While convolutional and recurrent architectures are not well suited to this specific classification problem [4], methods such as batch normalization or adopting more adaptive optimizers could enhance performance. The synthetic dataset used in this study provides a controlled environment for evaluating model behavior and performance, and exploring more diverse weather types with increasingly complex underlying relationships could allow a more detailed representation of atmospheric variability and provide a more challenging test of the models' classification capabilities. Furthermore, extending the analysis to real observational or reanalysis weather data would allow assessment of model generalization and has a practical relevance for real-world weather forecasting. NNs capable of detailed weather type identification could prove valuable for recognizing extreme weather events, which are becoming more frequent under the changing climate.

To conclude, this project demonstrate the flexibility and predictive strength of neural networks for the weather classification task. When equipped with appropriate architecture and hyperparameters, ML models, including simple logistic regression, are capable of classifying broad weather types to great precision, predicting around 90 percent of samples correctly. This suggests that ML models may play a growing part in future weather forecasting.

## VI. REFERENCES

[1] Zied Ben Bouallègue, Mariana C. A. Clare, Linus Magnusson, Estibaliz Gascón, Michael Maier-Gerber, Martin Janoušek, Mark Rodwell, Florian Pinault, Jesper S. Dramsch, Simon T. K. Lang, Baudouin Raoult, Florence Rabier, Matthieu Chevallier, Irina Sandu, Peter Dueben, Matthew Chantry, and Florian Pappenberger. The Rise of Data-Driven Weather Forecasting: A First Statistical Assessment of Machine Learning–Based Weather Forecasts in an Operational-Like Context. *Bulletin of the American Meteorological Society*, 105(6):E864–E883, June 2024.

[2] Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R. Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, Remi Lam, and Matthew Willson. Probabilistic weather forecasting with machine learning. *Nature*, 637(8044):84–90, January 2025.

[3] Massimo Bonavita. On Some Limitations of Current Machine Learning Weather Prediction Models. *Geophysical Research Letters*, 51(12):e2023GL107377, June 2024.

[4] Morten-Hjorth Jensen. Applied Data Analysis and Machine Learning — Applied Data Analysis and Machine Learning.

[5] Sebastian Raschka. *Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python.* Packt Publishing Limited, Birmingham, 1 edition, 2022.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York, New York, NY, 2009.

[7] Thomas Kurbiel. Derivative of the Softmax Function and the Categorical Cross-Entropy Loss, November 2021.

[8] Arnulf Jentzen, Benno Kuckuck, and Philippe von Wurstemberger. Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory, 2023. Version Number: 3.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[10] Nikhil Narayan. Weather Type Classification, November 2025. https://www.kaggle.com/datasets/nikhil7280/weather-type-classification.

[11] Kjersti Stangeland, Sverre Manu Johansen, Jenny Guldvog, and Ingvild Olden Bjerkelund. Developing a Feed-Forward Neural Network from Scratch for Regression and Classification Analysis FYS-STK4155 - Project 2. Technical report, University of Oslo, October 2025.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019. Version Number: 1.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] Sai Teja Anantha. L1/L2 Regularization in PyTorch, July 2025.

[15] Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. Van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[16] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61, Austin, Texas, 2010.

[17] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[18] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. Publisher: The Open Journal.

[19] OpenAI. ChatGPT, 2025.

[20] Martine Pedersen. Implementation and Analysis of Feed-Forward Neural Networks for Regression and Classification. Technical report, University of Oslo, October 2025.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, February 2015. arXiv:1502.01852 [cs].

## Appendix A: Feature distributions

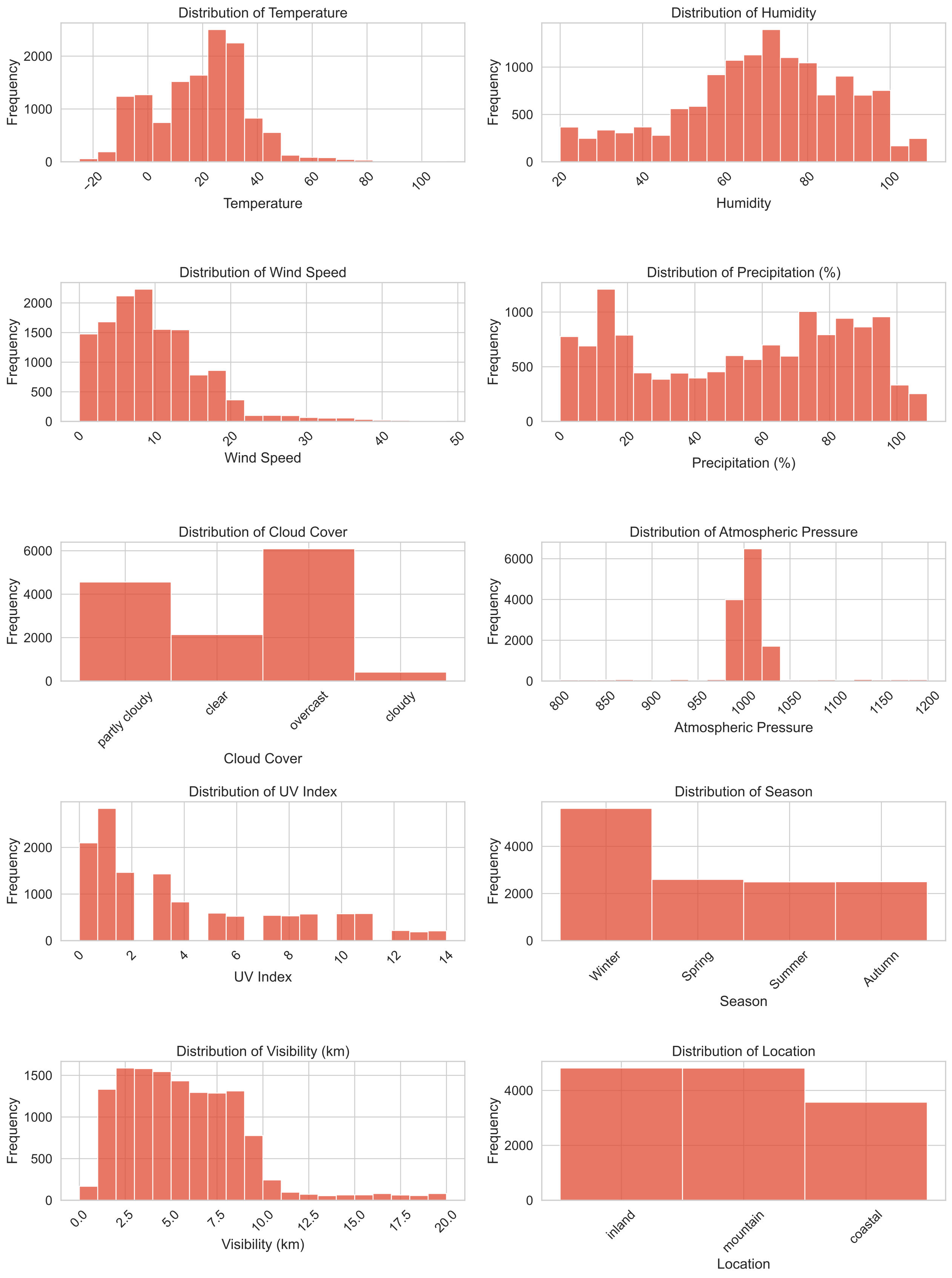


FIG. 11. The feature distribution of the *kaggle* dataset [10].