CSCI 3485 Lab 2

Rafael Almeida, Ryan Novitski

September 19, 2024

1 Introduction

Neural networks are powerful machine learning tools that can be used to solve complex data classification problems using flexible and adaptive architectures. However, their performance heavily relies on a plethora of complex parameters that directly impact its effectiveness, such as the number of layers, the number of units per layer, the learning rate, and the optimization algorithm. Understanding how these parameters effect the overall design is crucial for its success. Our goal for this lab is to explore these parameters and provide a comprehensive analysis on how these parameters affect the model's performance for the purpose of better understanding when choosing these values in future models.

2 Methodology

The first experiment is on 2D floating point values. Two algorithms were used to classify the points into centered blobs, and anisotropicly distributed points. The experimental variables for the neural networks were the number of layers and units per layer. Additionally, the networks were trained on data with differing number of classes. This experiment is relevant because it will tell us the affects of increasing layers and units on the classification accuracy, and how this affects the classification score with differing data distributions and number of classes. The hypothesis is that increasing the complexity of the network architecture should improve its classification accuracy. Additionally, the number of classes should decrease the accuracy of the networks, but this may vary in degrees given the network architecture. Moreover, it is hard to predict the effects different data distributions will have on the classification accuracy of the networks.

The second experiment uses the same type of data and clustering algorithm. It tests the effect of learning rates and learning rate strategies on the effectiveness of the classification model. Using the same datasets as the tests above, the neural network is trained using varying learning rates from 0.001 to 0.991, with each of the three different learning strategies available. The lower the learning rate is, the less it converges each layer. Because of this, various strategies regarding the learning rate were formed to attempt to balance under/overfitting data by jumping too little or too far. The hypothesis is that the more dynamic the learning rate strategy is, the better it will perform. Additionally, lower learning rates will generally be more accurate than higher rates.

The third experiment tests the effect different solvers have on the effectiveness of the classification model. Using the same datasets and learning rates as the tests above, the neural network is tested using both the SGD and ADAM solvers. While SGD is a simple and effective approach, ADAM is more dynamic and better able to prevent under/overfitting of the data. The hypothesis is that the ADAM solver will outperform the SGD solver in all categories.

3 Results

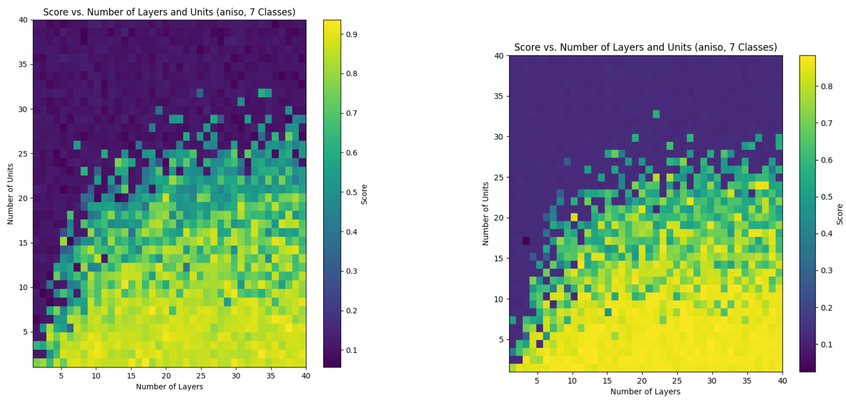
3.1 Layers and Units

The parameters for this experiment are:

- Number of data points: 500
- Number of features: 2
- Random state: 1
- Distribution algorithms: blob and anisotropic
- Range of number of classes 2-6
- Range of number of layers: 1-40
- Range of number of units per layer: 1-40

The number of data points is set to 500 because larger values would take much longer to train and would not change the general direction of the results besides the classification accuracy magnitude, as seen in Figure 1. The random state was set to 1 because experimentation demonstrated that it would create more complex data to classify linearly. Therefore, a value of 1 might result in more interesting data sets to attempt to classify. A range of 40 for the number of units per layer was chosen because, around this value, the neural networks would begin to perform horribly. Any larger values would yield only a few interesting results, as seen in Figure 2.

3.1.1 The Effects of Data Size



(a) Data size of 500

(b) Data size of 5000

Figure 1: Notice the data size does not affect the trend

3.1.2 Going Above 40 Units

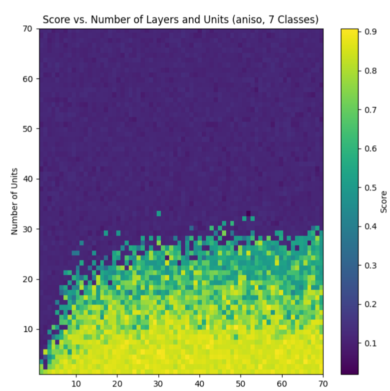
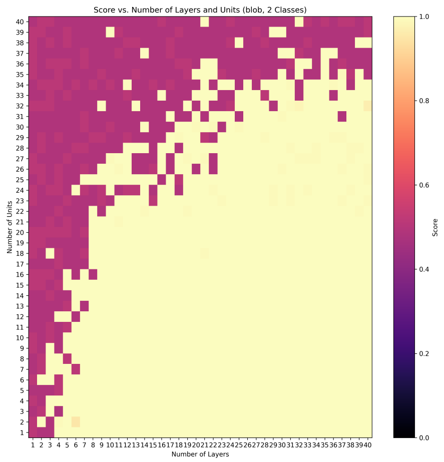
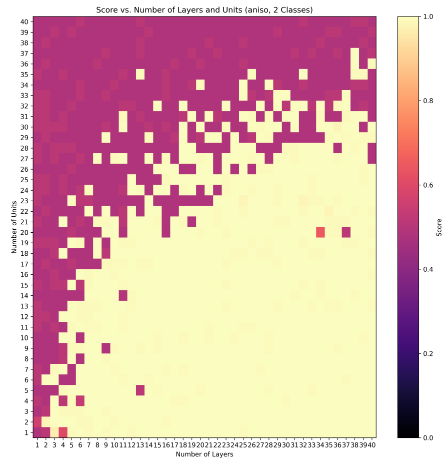
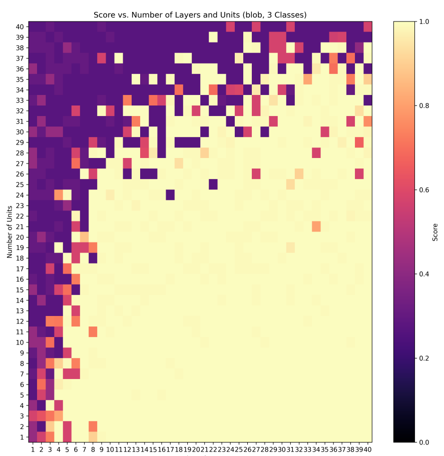


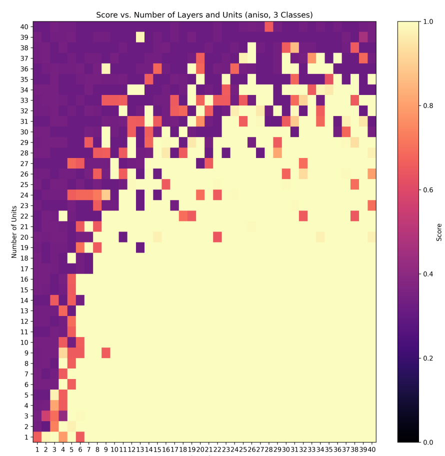
Figure 2: Notice the classification accuracy is low for any value greater than 40 units per layer

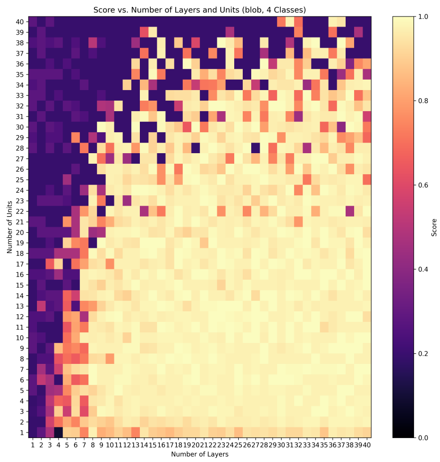
3.1.3 Score vs. Number of Layers and Units For Varying Data Distributions and Number of Classes

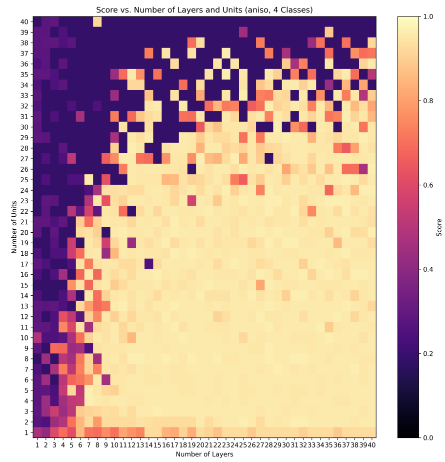


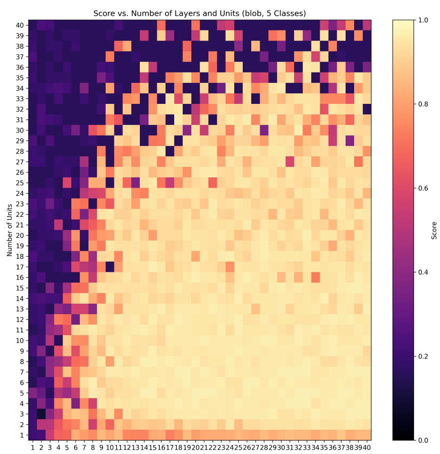


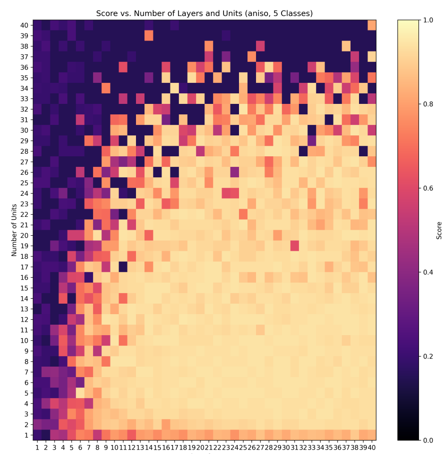


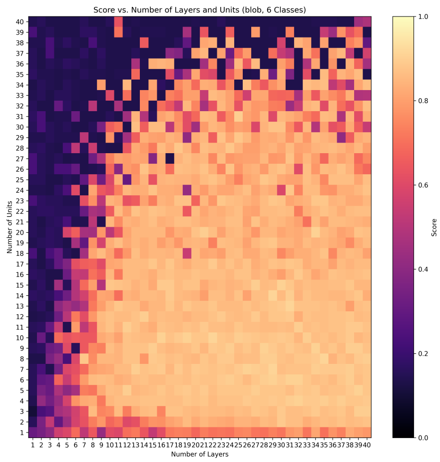


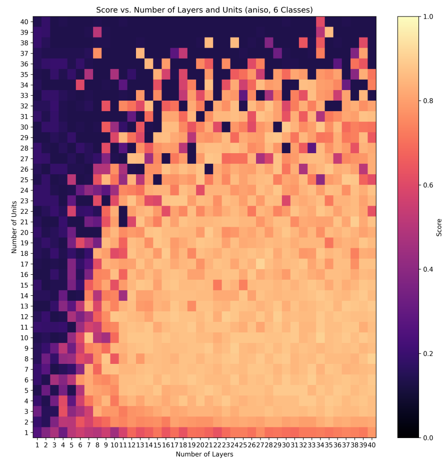

Score vs. Number of Layers and Units (blob, 4 Classes)
Number of Units
Number of Layers
Score


Score vs. Number of Layers and Units (aniso, 4 Classes)
Number of Units
Number of Layers
Score


Score vs. Number of Layers and Units (blob, 5 Classes)
Number of Units
Number of Layers
Score


Score vs. Number of Layers and Units (aniso, 5 Classes)
Number of Units
Number of Layers
Score





3.2 Solver Choice and Learning Rate

The parameters for this experiment are:

- Number of data points: 1000
- Number of features: 2
- Random state: 1
- Distribution algorithms: blob and anisotropic
- Range of number of classes: 2-3
- Number of layers: 10
- Number of units per layer: 5
- Range of initial learning rates: 0.001-0.991 (0.01 step size)
- Learning rate types: Constant, invscaling, adaptive
- Solvers used: SGD, ADAM

The number of datapoints is arbitrarily fixed at 1000. To maintain consistency with previous tests, the number of features, random state, and plot distributions remain the same. The range of number of classes was lowered to 2-3 to provide better visualizations for the data using different solvers. Based on the results of the previous tests, the highest efficiency was found with 10+ layers and 1-5 units per layer; to account for this, this experiment sets the layers and units at 10 and 5 respectively. All initial learning rates between .001 and .991 are tested at .01 increments using both solvers and all three variations of the learning rate.

3.2.1 Effects of Solver Choice: SGD

Figure 8: SGD performs best with low constant/adaptive

3.2.2 Effects of Solver Choice: ADAM

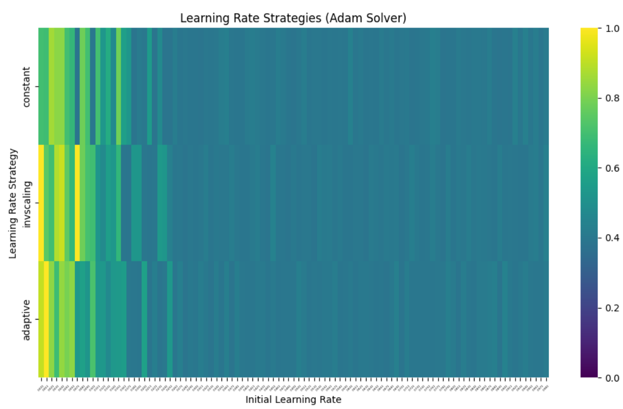


Figure 9: ADAM performs best with low invscaling/adaptive

4 Discussion

4.1 Layers and Units

4.1.1 Units Per Layer

As seen in section 3.1.3, increasing units per layer negatively affects the neural network score unless the number of layers is also increased, and only up to a certain point. From Figure 2, you can see that any number of units per layer over forty drastically affects the network score, regardless of the number of layers. Additionally, notice that at least four units are needed in the heat maps with higher classes for the network to perform well. This can be because more units are required in the output layer, so it would intuitively insinuate that a network with a shape similar to "¡" is required. The network will perform better if the number of units in the hidden layers is closer to the number of units in the output layer. Thus, a network with a shape similar to "¿" is not performant because, at the final layers, there are too few units feeding values into the output layer, spreading thin the information accumulated through the network.

4.2 Number of Layers

Looking at the number of layers, we notice a similar pattern: increasing numbers of layers with a good choice of units per layer does not have any meaningful effects (in the case of units, an ever-increasing number has adverse effects). However, notice that as the number of classes increases, a minimum number of layers is required for the network to perform decently, similar to the number of units. Putting this together, after a good choice of units per layer, the number of layers has diminishing returns. This can be explained by imagining that, given a good number of units per layer, adding more and more layers, will only generate weights with very slightly better values between each layer. Although the accuracy may improve, it will only be by a little.

4.3 Heatmap Jaggedness

To explain the jaggedness of the heatmaps, consider that not all networks had their optimizers converge. When training the networks, a maximum of 200 iterations was used. This means that certain combinations of the number of layers and units per layer may have required more iterations to have the optimizer converge or reach a good enough optimization. On the other hand, some networks may have reached an optimal value in the few iterations by chance. This explains how some networks with a few layers and many units per layer (predicted to be a bad network) could have performed well and vice-versa.

4.4 Solver Choice and Learning Rates

Based on the results of the tests conducted using various learning rates, learning rate algorithms, and solvers, various conclusions can be supported and improved with further testing. The results provide evidence that the choice regarding these parameters has a direct and strong effect on the effectiveness of the neural network.

4.4.1 Initial Learning Rate

Each test was run using 100 different learning rates spanning from 0.001 and 0.991. Generally speaking, under the specified parameters, it seems clear that a lower initial learning rate provides more accuracy to the model than a higher one across both solvers and all learning rate strategies. A higher learning rate allows for faster convergence, but also creates a higher potential for overfitting the data. A lower learning rate creates a slower descent towards the correct prediction, which these results support to be a more effective approach. The only exception to this trend came from the SGD solver using the invscaling learning strategy; instead of having a higher accuracy with lower initial learning rates, the accuracy was scattered randomly across all values.

4.4.2 Learning Rate Strategies: Constant

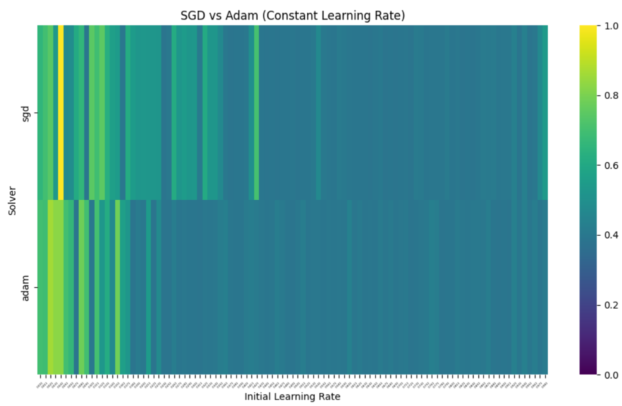


Figure 10: SGD generally performs better with these conditions

The constant learning rate uses a fixed learning rate for the duration of testing. For both solvers, there was a clear drop in accuracy as the initial learning rate increased. This intuitively makes sense, as an immutable high learning rate creates a big risk for overfitting. Although ADAM performed slightly more consistently across the lower learning rates, the SGD solver had a higher maximum accuracy rating.

4.4.3 Learning Rate Strategies: Invscaling

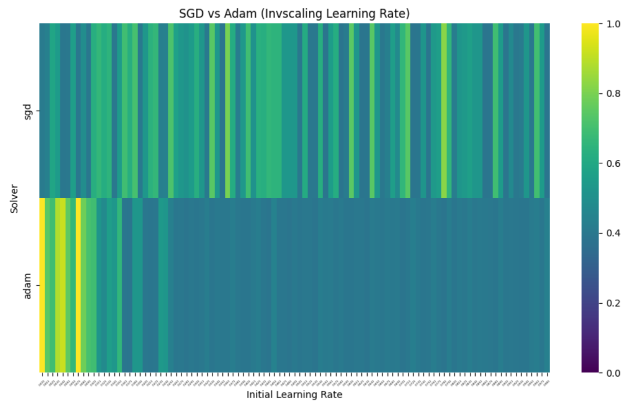


Figure 11: Adam has higher predictability

Invscaling uses a descending learning rate to allow the model to both move quickly towards convergence and reduce the risk of overfitting. The invscaling learning rate is defined as:

$$\text{learning rate}_t = \frac{\text{initial learning rate}}{(1 + \text{power\_t} \times t)^{\text{exponent}}} \tag{1}$$

where:

- **initial learning rate** is the starting learning rate.
- **t** is the current iteration or epoch number.
- **power_t** is a constant parameter that scales the rate of decrease.
- **exponent** controls the decay rate, often set to 0.5.

Using this equation, the ADAM solver proved to be more efficient than the results shown using the constant learning rate; it was significantly more effective at lower learning rates, but was still equally inconsistent with higher learning rates. However, the SGD solver created a very randomized spread across all learning rates. This is and interesting observation and demonstrates the power invscaling can have to combat overfitting data.

4.4.4 Learning Rate Strategies: Adaptive

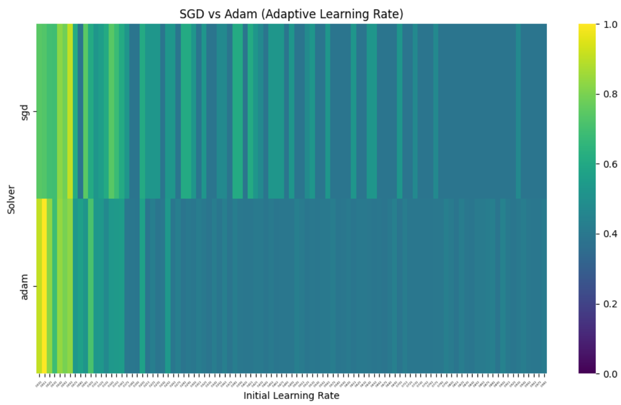


Figure 12: Adam has higher consistency across some rates

The adaptive learning rate strategy allows for a more customized approach to the changing learning rate using these properties:

- The **Learning Rate Adjustment** changes based on the magnitudes calculated during training. This allows the learning rate to be both increased or decreased based on the previous step's success rate.
- **Gradient Accumulation:** The adaptive learning rate keeps track of past gradients to help scale the learning rates based on how frequently that node has been updated.
- **Per-Parameter Learning Rate:** The adaptive learning rate uses a seperate learning rate for each parameter in the model, allowing for maximum flexibility and potential for success.

Using this strategy, it seems there is a better level of consistency across the board for all learning rates. Although ADAM performed slightly better under certain learning rates using the invscaling method, there is a slight improvement across higher learning rates.

5 Conclusion

Based on the conclusions drawn, the process of choosing an optimal number of layers and units per layer for a network and a given dataset can be outlined. One can start with at least as many units as classes, and progressively increase the number of layers until a balance between improved accuracy and training time is achieved. Although the training time for each network was tested, the data was not included due to the difficulty of representing the four dimensions. However, this information would be valuable in making a more informed decision.

Overall, it seems clear a lower initial learning rate proves to be more consistent than a higher value across both solvers and all descent strategies. The ADAM solver generally has higher maximum accuracy than the SGD solver, but its accuracy drops off much faster with increasing rates. Using these parameters, the data suggests the most effective combination of solvers and learning rate strategies is the ADAM solver with an initial learning rate below 0.1 and either the invscaling or adaptive strategy. Intuitively, the adaptive strategy should work more consistently across randomized data and testing due to its greater flexibility.

6 Code

GitHub Repository

6.1 ryan.py

```

import warnings
from time import time

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from tqdm import tqdm

from data_generators import aniso, blob
from read_and_write import write_to_file

# Ignoring convergence warnings to not mess up the progress bar
warnings.filterwarnings("ignore", category=ConvergenceWarning)

# Default values
DATA_SIZE = 1000
NUMBER_OF_FEATURES = 2
RANDOM_STATE = 1

# Dataset ranges
DATA_TYPES = [blob, aniso]
MIN_NUMBER_OF_CLASSES = 2
MAX_NUMBER_OF_CLASSES = 3

# Network architecture fixed values
```

```python
NUMBER_OF_LAYERS = 10    # 10+ layers based on results of network architecture tests
NUMBER_OF_UNITS = 5     # Lower units(1-5) based on results of network architecture tests

# Learning rate ranges and solvers
MIN_LEARNING_RATE_INIT = 0.001
MAX_LEARNING_RATE_INIT = 0.991
LEARNING_RATE_STEP_SIZE = 0.01
SOLVERS = ['sgd', 'adam']
LEARNING_RATE_STRATEGIES = ['constant', 'invscaling', 'adaptive']

# Operations counts
total_datasets = len(DATA_TYPES) * (
    MAX_NUMBER_OF_CLASSES - MIN_NUMBER_OF_CLASSES + 1
)
total_networks = len(SOLVERS) * len(LEARNING_RATE_STRATEGIES) * (
    int((MAX_LEARNING_RATE_INIT - MIN_LEARNING_RATE_INIT) / LEARNING_RATE_STEP_SIZE) + 1
)

# Initializing timer
master_start_time = time()

# Building datasets
data_pbar = tqdm(total=total_datasets, desc="Building Datasets")

datasets = []
for data_type in DATA_TYPES:
    for number_of_classes in range(MIN_NUMBER_OF_CLASSES, MAX_NUMBER_OF_CLASSES + 1):
        X, y = data_type(DATA_SIZE, number_of_classes, NUMBER_OF_FEATURES, RANDOM_STATE)
        X_train, X_test, y_train, y_test = train_test_split(X, y)
        datasets.append((data_type.__name__, number_of_classes, X_train, X_test, y_train, y_test))
        data_pbar.update(1)

data_pbar.close()

# Running experiments
experiment_pbar = tqdm(total=total_networks * total_datasets, desc="Running Experiments")

results = []
for solver in SOLVERS:
    for learning_rate_strategy in LEARNING_RATE_STRATEGIES:
        for learning_rate_init in np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE):
            network = MLPClassifier(
```

```
                hidden_layer_sizes=(NUMBER_OF_UNITS,) * NUMBER_OF_LAYERS,
                solver=solver,
                learning_rate=learning_rate_strategy,
                learning_rate_init=learning_rate_init
            )

            for dataset in datasets:
                start_time = time()

                X_train, X_test, y_train, y_test = dataset[2], dataset[3], dataset[4], dataset[5]
                network.fit(X_train, y_train)
                score = network.score(X_test, y_test)

                results.append((
                    dataset[0], dataset[1], solver, learning_rate_strategy,
                    learning_rate_init, score, time() - start_time
                ))

                experiment_pbar.update(1)

experiment_pbar.close()
print(f"Total time: {time() - master_start_time}")

# Writing results to file in case of somehow losing the data
write_to_file(results, "solver_learning_rate_results.txt")

# Converting results to a numpy array for easier processing
results = np.array(results, dtype=[
    ("type_of_data", "U50"),
    ("classes", int),
    ("solver", "U10"),
    ("learning_rate_strategy", "U10"),
    ("learning_rate_init", float),
    ("score", float),
    ("total_time", float),
])

# Aggregate results for blobs and aniso
def aggregate_results(results):
    aggregated_results = []
    for solver in SOLVERS:
        for lr_strategy in LEARNING_RATE_STRATEGIES:
            for learning_rate_init in np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE):
                mask = (results["solver"] == solver) & (results["learning_rate_strategy"] == lr_strategy)
                scores_for_lr_init = results[mask & (results["learning_rate_init"] == learning_rate_init)]["score"]
```

```python
                mean_score = np.mean(scores_for_lr_init)
                aggregated_results.append((solver, lr_strategy, learning_rate_init, mean_score))
    return np.array(aggregated_results, dtype=[("solver", "U10"), ("learning_rate_strategy", "U10"), ("learning_rate_init", float), ("score", float)])

aggregated_results = aggregate_results(results)

# Heatmap function
def plot_heatmap(data, row_labels, col_labels, title, xlabel, ylabel):
    plt.figure(figsize=(10, 6))
    sns.heatmap(data, xticklabels=col_labels, yticklabels=row_labels, cmap="viridis", cbar=True, vmin=0, vmax=1)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xticks(rotation=45, fontsize=3)
    plt.tight_layout()
    plt.show()

# Heatmap for sgd vs adam for each learning rate strategy
def plot_solver_comparison(aggregated_results, lr_strategy):
    heatmap_data = np.zeros((2, len(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE))))
    solvers = ['sgd', 'adam']

    for i, solver in enumerate(solvers):
        for j, learning_rate_init in enumerate(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE)):
            mask = (aggregated_results["solver"] == solver) & (aggregated_results["learning_rate_strategy"] == lr_strategy)
            score = aggregated_results[mask & (aggregated_results["learning_rate_init"] == learning_rate_init)]["score"]
            heatmap_data[i, j] = np.mean(score)

    plot_heatmap(heatmap_data, solvers, np.round(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE), 3),
                 f"SGD vs Adam ({lr_strategy.capitalize()} Learning Rate)", "Initial Learning Rate", "Solver")

# Heatmap for comparing learning rate strategies for each solver
def plot_lr_strategy_comparison(aggregated_results, solver):
    heatmap_data = np.zeros((3, len(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE))))
    lr_strategies = ['constant', 'invscaling', 'adaptive']

```

```python
    for i, lr_strategy in enumerate(lr_strategies):
        for j, learning_rate_init in enumerate(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE)):
            mask = (aggregated_results["solver"] == solver) & (aggregated_results["learning_rate_strategy"] == lr_strategy)
            score = aggregated_results[mask & (aggregated_results["learning_rate_init"] == learning_rate_init)]["score"]
            heatmap_data[i, j] = np.mean(score)

    plot_heatmap(heatmap_data, lr_strategies, np.round(np.arange(MIN_LEARNING_RATE_INIT, MAX_LEARNING_RATE_INIT, LEARNING_RATE_STEP_SIZE), 3),
                 f"Learning Rate Strategies ({solver.capitalize()} Solver)", "Initial Learning Rate", "Learning Rate Strategy")

# Plot visualizations
plot_solver_comparison(aggregated_results, 'constant')
plot_solver_comparison(aggregated_results, 'invscaling')
plot_solver_comparison(aggregated_results, 'adaptive')

plot_lr_strategy_comparison(aggregated_results, 'sgd')
plot_lr_strategy_comparison(aggregated_results, 'adam')
```

6.2 rafael.py

```python
"""
Executes experiments to analyze the effects of different number of layers and units per layer on the score of a network.
"""

from os import makedirs, path
from time import time
from warnings import filterwarnings

import matplotlib.pyplot as plt
from numpy import array as nparray
from numpy import zeros as npzeros
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from tqdm import tqdm

from data_generators import aniso, blob
from read_and_write import write_to_file

# ignoring convergence warnings to not mess up the progress bar
```

```
filterwarnings("ignore", category=ConvergenceWarning)

# default values
FOLDER_NAME = "layers_units"
DATA_SIZE = 500  # barely any difference between 500-5000 data points
NUMBER_OF_FEATURES = 2
RANDOM_STATE = 1

print("Press enter for default values")

# dataset ranges
DATA_TYPES = [blob, aniso]
MIN_NUMBER_OF_CLASSES = int(input("MIN_NUMBER_OF_CLASSES or 2: ") or 2)
MAX_NUMBER_OF_CLASSES = int(input("MAX_NUMBER_OF_CLASSES or 7: ") or 7)

# network architecture ranges
MIN_NUMBER_OF_LAYERS = int(input("MIN_NUMBER_OF_LAYERS or 1: ") or 1)
MAX_NUMBER_OF_LAYERS = int(input("MAX_NUMBER_OF_LAYERS or 40: ") or 40)
MIN_NUMBER_OF_UNITS = int(input("MIN_NUMBER_OF_UNITS or 1: ") or 1)
MAX_NUMBER_OF_UNITS = int(input("MAX_NUMBER_OF_UNITS or 40: ") or 40)

# operations counts
TOTAL_DATASETS = len(DATA_TYPES) * (
    MAX_NUMBER_OF_CLASSES - MIN_NUMBER_OF_CLASSES + 1
)
TOTAL_NETWORKS = (MAX_NUMBER_OF_LAYERS - MIN_NUMBER_OF_LAYERS + 1) * (
    MAX_NUMBER_OF_UNITS - MIN_NUMBER_OF_UNITS + 1
)


def generate_datasets() -> list[dict]:
    pbar = tqdm(total=TOTAL_DATASETS, desc="Generate Datasets")

    datasets = []
    for data_type in DATA_TYPES:
        for number_of_classes in range(
            MIN_NUMBER_OF_CLASSES, MAX_NUMBER_OF_CLASSES + 1
        ):
            X, y = data_type(
                DATA_SIZE,
                number_of_classes,
                NUMBER_OF_FEATURES,
                RANDOM_STATE,
            )

            X_train, X_test, y_train, y_test = train_test_split(X, y)

            datasets.append(
                {
                    "data_type": data_type.__name__,
```

```
                    "number_of_classes": number_of_classes,
                    "X_train": X_train,
                    "X_test": X_test,
                    "y_train": y_train,
                    "y_test": y_test,
                }
            )
            pbar.update(1)

    pbar.close()

    return datasets


def run_experiments(datasets: list[dict]) -> list[tuple]:
    pbar = tqdm(total=TOTAL_NETWORKS * TOTAL_DATASETS, desc="Run Experiments")

    results = []
    for layers in range(MIN_NUMBER_OF_LAYERS, MAX_NUMBER_OF_LAYERS + 1):
        for units in range(MIN_NUMBER_OF_UNITS, MAX_NUMBER_OF_UNITS + 1):
            network = MLPClassifier(hidden_layer_sizes=(units,) * layers)

            for dataset in datasets:
                start_time = time()

                X_train, X_test, y_train, y_test = (
                    dataset["X_train"],
                    dataset["X_test"],
                    dataset["y_train"],
                    dataset["y_test"],
                )

                # train network and get score
                network.fit(X_train, y_train)
                score = network.score(X_test, y_test)

                results.append(
                    (
                        dataset["data_type"],
                        dataset["number_of_classes"],
                        layers,
                        units,
                        score,
                        time() - start_time,
                    )
                )

                pbar.update(1)

```

```python
    pbar.close()

    return results


def download_heat_maps(results: list[tuple]) -> None:
    # converting results to nparray to easily access columns
    results = nparray(
        results,
        dtype=[
            ("type_of_data", "U50"),
            ("classes", int),
            ("layers", int),
            ("units", int),
            ("score", float),
            ("total_time", float),
        ],
    )

    for data_type in DATA_TYPES:
        for number_of_classes in range(
            MIN_NUMBER_OF_CLASSES, MAX_NUMBER_OF_CLASSES + 1
        ):
            data_type_name = data_type.__name__
            # processing data
            mask = (results["type_of_data"] == data_type_name) & (
                results["classes"] == number_of_classes
            )
            filtered_results = results[mask]

            layers = filtered_results["layers"]
            units = filtered_results["units"]
            score = filtered_results["score"]

            score_2d = npzeros(
                (
                    MAX_NUMBER_OF_LAYERS - MIN_NUMBER_OF_LAYERS + 1,
                    MAX_NUMBER_OF_UNITS - MIN_NUMBER_OF_UNITS + 1,
                )
            )

            for l, u, s in zip(layers, units, score):
                score_2d[l - MIN_NUMBER_OF_LAYERS, u - MIN_NUMBER_OF_UNITS] = s

            # setting up plot
            aspect_ratio = score_2d.shape[1] / score_2d.shape[0]
            fig_width = 10
            fig_height = fig_width / aspect_ratio

```

```
            plt.figure(figsize=(fig_width, fig_height), dpi=300)

            im = plt.imshow(
                score_2d,
                cmap="magma",
                extent=(
                    MIN_NUMBER_OF_LAYERS - 0.5,
                    MAX_NUMBER_OF_LAYERS + 0.5,
                    MIN_NUMBER_OF_UNITS - 0.5,
                    MAX_NUMBER_OF_UNITS + 0.5,
                ),
                origin="lower",
                aspect="auto",
                vmin=0,
                vmax=1,
            )

            cbar = plt.colorbar(im, label="Score")
            plt.xlabel("Number of Layers")
            plt.ylabel("Number of Units")
            plt.title(
                f"Score vs. Number of Layers and Units ({data_type_name}, {number_of_classes} Classes)"
            )

            plt.xticks(range(MIN_NUMBER_OF_LAYERS, MAX_NUMBER_OF_LAYERS + 1))
            plt.yticks(range(MIN_NUMBER_OF_UNITS, MAX_NUMBER_OF_UNITS + 1))

            plt.tight_layout()

            # Downloading plots
            makedirs(FOLDER_NAME, exist_ok=True)

            filename = f"size{DATA_SIZE}_features{NUMBER_OF_FEATURES}_random{RANDOM_STATE}
            _type{data_type_name}_classes{number_of_classes}.png"
            filepath = path.join(FOLDER_NAME, filename)

            plt.savefig(filepath, dpi=300, bbox_inches="tight")
            plt.clf()
            plt.close()

            print(f"Heatmap saved as {filepath}")


# main

master_start_time = time()
```

```python

datasets = generate_datasets()
results = run_experiments(datasets)

print(f"Total time: {time() - master_start_time}")

# saving data in any case
write_to_file(results, "rafael_results.txt")

download_heat_maps(results)
```

6.3 data_generators.py

```python
from typing import Tuple

import numpy as np
from sklearn.datasets import make_blobs


def blob(
    samples: int, centers: int, features: int, random_state: int
) -> Tuple[np.ndarray, np.ndarray]:
    """Generates a blob dataset with labels"""

    X, Y = make_blobs(
        n_samples=samples,
        centers=centers,
        n_features=features,
        random_state=random_state,
    )
    return X, Y


def aniso(
    samples: int, centers: int, features: int, random_state: int
) -> Tuple[np.ndarray, np.ndarray]:
    """Generates a aniso dataset with labels"""

    X, Y = blob(samples, centers, features, random_state)
    transformation = [[0.6, -0.6], [-0.4, 0.8]]
    X = np.dot(X, transformation)
    return X, Y
```

6.4 read_and_write.py

```python
from typing import Any, List

```

```python

def write_to_file(arr: List[List[Any]], file_name: str) -> None:
    """Writes a 2D array to file_name with columns separated with tabs
    and rows with newlines"""

    with open(file_name, "w") as f:
        for row in arr:
            f.write("\t".join(map(str, row)) + "\n")


def read_to_array(file_name: str) -> List[List[Any]]:
    """Reads a file and returns a 2D array"""

    with open(file_name, "r") as f:
        return [line.strip().split("\t") for line in f]
```