Open Lab 1

Single-Layer Networks

Grade: 100/100

CSCI 7850 - Deep Learning

Due: Sep. 12 @ 11:00pm

Assignment

Here are the details of what you need to do for this assignment:

- Create a python script named `OL1.py` that takes three command line arguments: `[Data] [Optimizer] [Standardize]`
  - `[Data]` can be either `iris` or `wdbc` : your script should use the appropriate data set selected using this argument
  - `[Optimizer]` can be either `adam` or `rmsprop` or `sgd` to select the optimizer used by the model (Adam(), RMSprop(), or SGD())
  - `[Standardize]` can be either `0` or `1` indicated whether to perform standardization on the data (`0` for unstandardized and `1` for standardized)
  - Your code should **always** perform a permutation (or shuffle) of the data set prior to training
  - Other settings should be maintained as specified in the tutorial below (number of epochs=101, etc.) so that your script prints the results line (validation accuracy) **exactly matching the format provided below**.
- Use your `OL1.py` script to run your models using all of the different combinations of command-line arguments - **perform 100 independent runs for each combination**.

- Compile your data into *uniquely named* text files for each combination of arguments that can be read in using `np.loadtxt` .

- Create an iPython Notebook file named `OL1.ipynb` which reads in the compiled results for Adam, RMSprop, and SGD to produce a learning curve with mean and standard error. An example of one such plot is provided below, but your notebook should contain four plots in the end: (WDBC-unstandardized), (WDBC-standardized),(Iris-unstandardized), (Iris-standardized). No code for model training/testing should be in this notebook file, it should only read in the results text files and plot them.

- At the end of your notebook file, create a Markdown cell and compile answers to the following questions:

  1. Which of the chosen optimizers seems to benefit the most from standardization of the data sets?
  2. Why do you think this may be happening?
  3. What other choices (hyperparameters, architecture changes, data prep, etc.) do you think might be explored which could impact the performance?
  4. What outcome would you expect from changing the code in this way (hypothesis)?
  5. What process would you use to attempt to confirm your hypothesis and what steps would testing it involve?
  6. Which part(s) of the lab/code/experiments are still *unclear* to you after finishing this assignment?
  7. Which parts are you interested in learning more about?

Example results plot (illustration for `OL1.ipynb`)

In [136…

```
results1 = np.loadtxt("wdbc-unstandardized-adam-results.txt")
results2 = np.loadtxt("wdbc-unstandardized-rmsprop-results.txt")
results3 = np.loadtxt("wdbc-unstandardized-sgd-results.txt")
```

In [8]:

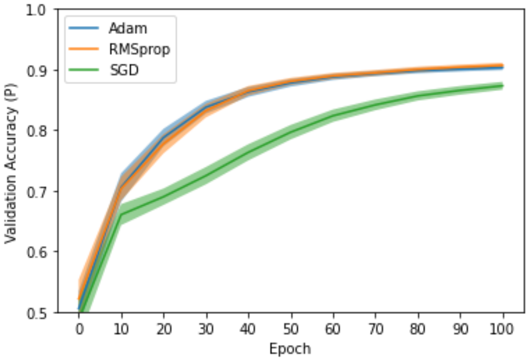
```
# All have this shape
results1.shape
```

Out[8]:

```
(100, 11)
```

In [27]:

```
plt.plot(np.arange(0,results1.shape[1]),
         np.mean(results1,0),label='Adam')
plt.fill_between(np.arange(0,results1.shape[1]),
                 np.mean(results1,0)-(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
                 np.mean(results1,0)+(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results2.shape[1]),
         np.mean(results2,0),label='RMSprop')
plt.fill_between(np.arange(0,results2.shape[1]),
                 np.mean(results2,0)-(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 np.mean(results2,0)+(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results3.shape[1]),
         np.mean(results3,0),label='SGD')
plt.fill_between(np.arange(0,results3.shape[1]),
                 np.mean(results3,0)-(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 np.mean(results3,0)+(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 alpha=0.5)
plt.ylim([0.5,1.0])
plt.ylabel('Validation Accuracy (P)')
plt.xlabel('Epoch')
plt.legend()
plt.xticks(ticks=np.arange(0,results1.shape[1]),
           labels=np.arange(0,results1.shape[0]+1,10))
plt.show()
```



Submission

Create a zip archive which contains the following contents:

- `OL1.py`
- `OL1.ipynb`
- `*-results.txt` : **All** processed results text files (needed by np.loadtxt in your notebook)
- `sources.pdf` : **Optional**, if you used any electronic resources other than those provided in the course, include full documentation of how/why they were utilized

Upload your zip archive to the course assignment system by the deadline at the top of this document.

Computational Explorations Using JupyterLab: *Inside and Outside the Notebook*

JupyterLab is a great interface for prototyping, interactive coding, and explanatory documentation, so we will utilize it often for performing analysis and making plots. However, it's also not always the best tool for the job. Your work below will consist of utilizing a Jupyter Notebook to explore the creation and training of single-layer neural networks for binary- and multi-class data sets.

Let's start by loading a data set, and doing some preliminary exploration of what information seems to be in that data set - effectively trying to determine the task (T) that this particular data is aiming to solve. Next, we will parse the data into distinct

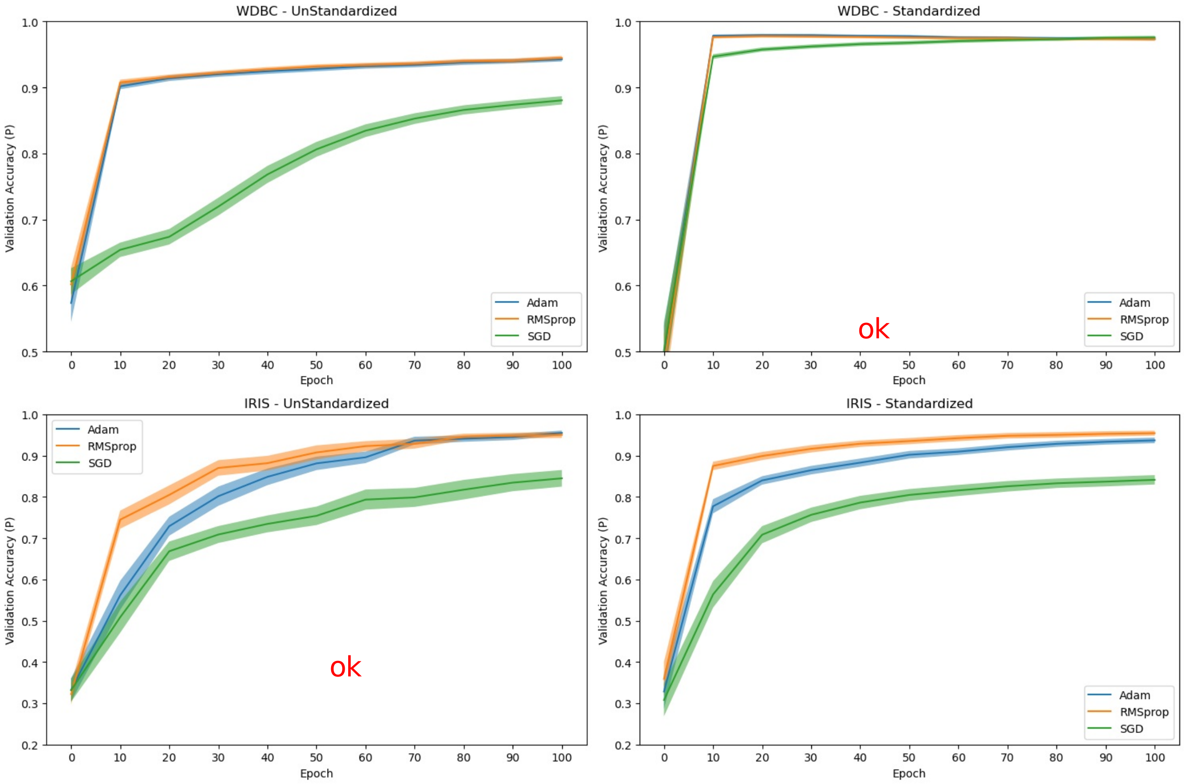
In [1]:

```python
# Setup Imports
import numpy as np
import matplotlib.pyplot as plt

# Loading Results from OL1.py Application outputs
results = [{
        "title":"WDBC - UnStandardized",
        "limits": [0.5, 1.0],
        "adam": np.loadtxt("wdbc-unstandardized-adam-results.txt"),
        "rmsprop": np.loadtxt("wdbc-unstandardized-rmsprop-results.txt"),
        "sgd": np.loadtxt("wdbc-unstandardized-sgd-results.txt"),
    },{
        "title":"WDBC - Standardized",
        "limits": [0.5, 1.0],
        "adam": np.loadtxt("wdbc-standardized-adam-results.txt"),
        "rmsprop": np.loadtxt("wdbc-standardized-rmsprop-results.txt"),
        "sgd": np.loadtxt("wdbc-standardized-sgd-results.txt"),
    },{
        "title":"IRIS - UnStandardized",
        "limits": [0.2, 1.0], # On IRIS the limit is lower than on WDBC
        "adam": np.loadtxt("iris-unstandardized-adam-results.txt"),
        "rmsprop": np.loadtxt("iris-unstandardized-rmsprop-results.txt"),
        "sgd": np.loadtxt("iris-unstandardized-sgd-results.txt"),
    },{
        "title":"IRIS - Standardized",
        "limits": [0.2, 1.0], # On IRIS the limit is lower than on WDBC
        "adam": np.loadtxt("iris-standardized-adam-results.txt"),
        "rmsprop": np.loadtxt("iris-standardized-rmsprop-results.txt"),
        "sgd": np.loadtxt("iris-standardized-sgd-results.txt"),
    }]
```

```
Matplotlib is building the font cache; this may take a moment.
```

In [2]:

```python
# Create a 2x2 grid for the Plots (subplots)
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Default Values for all Plots
for i, axis in enumerate(axes.ravel()):

    # Setup Data Layers for Plots
    axis.set_title(results[i]["title"])

    # Setup Adam - Optimized
    data = results[i]["adam"]
    axis.plot(np.arange(0, data.shape[1]), np.mean(data, 0),label='Adam')
    axis.fill_between(np.arange(0, data.shape[1]), np.mean(data, 0)-(1.96*np.std(data, 0)/np.sqrt(data.shape[0]

    # Setup RMSprop - Optimized
    data = results[i]["rmsprop"]
    axis.plot(np.arange(0, data.shape[1]), np.mean(data, 0),label='RMSprop')
    axis.fill_between(np.arange(0, data.shape[1]), np.mean(data, 0)-(1.96*np.std(data, 0)/np.sqrt(data.shape[0]

    # Setup SGD - Optimized
    data = results[i]["sgd"]
    axis.plot(np.arange(0, data.shape[1]), np.mean(data, 0),label='SGD')
    axis.fill_between(np.arange(0, data.shape[1]), np.mean(data, 0)-(1.96*np.std(data, 0)/np.sqrt(data.shape[0]

    # General Setting and UI configs that are the same on each Plot
    axis.set_ylabel('Validation Accuracy (P)')
    axis.set_ylim(results[i]["limits"])
    axis.set_xlabel('Epoch')
    axis.set_xticks(ticks=np.arange(0, 11), labels=np.arange(0, 110, 10))
    axis.legend()

# Adjust spacing between subplots
plt.tight_layout()

# Show the plots
plt.show()
```



ok

ok

Ansers to OL1 Questions:

Answer 1

Looking over the plots from above, it seems that the SGD optimizer benefited the most from standardizing the input values. This was particularly true for the WDBC dataset.

ok

Answer 2

The main advantages of helping the SGD optimizer by standardization are that the convergence is faster because the input features (especially in the WDBC dataset) have different scales on the main dataset. Using standardization made the inputs all the same scale, thus making adjustments during the learning/optimization process helped to not over- and under-correct on epochs, and hence, get the best prediction result in a smoother and probably faster way.

ok

Answer 3

A few things come to mind:

1. Work on a more detailed learning rate (lr) for the different optimizers (all of them used the same for the most part). Maybe adjust it based on whether I did or did not use standardization of the input data. *(this is Hyperparameter adjustment)*
2. In my case, I should try and clean up the input features better, especially on the multiclass classification model (IRIS). It seems like I had to adjust the `y_pred` to be a different datatype when processing the training and validation steps in the model.
3. Determine how many learning iterations (epochs) I really need to train the models. For example, I believe that after about ~50-75 epochs, the prediction rates did not get materially better.
4. Look at different standardization for the data. It seems that on the WDBC data, standardization helped immensely; however, on the IRIS data, it had less of an impact. I, therefore, think I could look into **Normalization** rather than standardization techniques.

ok

Answer 4

Since I listed a few details above, I would have a few expected answers:

1. I would see less of a "spread" between the epochs' predictions. Meaning I think the learning would be more uniform

as I run my "OL1.py" app 100 times.

2. It would make the code a bit cleaner and easier to read if I clean it up before loading it into the trainers.
3. I think it would improve the processing time for OL1.py and therefore use fewer resources.
4. Might help on the IRIS with the SGD optimizer being more in line with RMSprop and ADAM optimizers.

ok

Answer 5

As stated above, since I listed 4x, I will try to list my processes related to it:

1. I would compare the prediction curves, like the above plots, to each other to see if one moves quicker to a better validation rate of 1.0 in comparison to the others.
2. Since I suggested a simple code change, it would just be more readable...
3. I would add some date and timestamp logging to see if I can materially speed up the process. I would probably use the **logging** module (https://docs.python.org/3/library/logging.html).
4. I could try Min and Max - Normalization instead of standardization like I previously did. This would be something like this: `x_new = (x - min) / (max - min)`. I would have to test and see if it performs better than the standardization I used.

ok

Answer 6

Overall, I had a hard time running the `OL1.py` efficiently. I was able to log into the clusters, but I feel like it didn't really work out for me in parallelization. It took a long time to run, and overall, I think it was as fast simply letting it run on `biosim` overnight rather than using the `sbatch` process. I could really use more help in testing or understanding how to use it.

It's true that the benefit is mimized on this assignment since it's mainly a training lab - future assignments will -require- GPU cluster resources (of some kind) for completion.

Answer 7

From the above mentioned, I would really like to have a "prebuilt" application to test locally, on `biosim`, and in the cluster to gain some more knowledge on how to use these resources.

In addition, I think some input on **Normalization** that I mentioned above might be interesting as well.

Sure, there are some techniques that we will be using in future lab assignments that will help with this.

```
#!/usr/bin/env python3

import sys
import numpy as np
import torch
import lightning.pytorch as pl
import pandas as pd
import torchmetrics

class Config:
    def __init__(self):
        self.data_type = None
        self.optimizer_name = None
        self.use_standardization = None
        self.max_epochs = 100
        self.num_classes = 0

    def parse_args(self):
        self.data_type = (sys.argv[1] or "").lower() # Make sure it is lower case
        self.optimizer_name = (sys.argv[2] or "").lower() # Make sure it is lower case
        self.use_standardization = (sys.argv[3] == "1") # Conver to bool
        self.device = "gpu" if torch.cuda.is_available() else "cpu" # Setup Device

    @property
    def is_multi_classification(self):
        return (self.num_classes > 2)

    @property
    def data_url(self):
        match self.data_type:
            case "iris":
                return "https://www.cs.mtsu.edu/˜jphillips/courses/CSCI7850/public/iris-data.txt"
            case "wdbc":
                return "https://www.cs.mtsu.edu/˜jphillips/courses/CSCI7850/public/WDBC.txt"
            case _:
                return ""

    @property
    def result_filename(self):
        _filename = ""
        _filename += self.data_type
        _filename += "-"
        _filename += "standardized" if self.use_standardization else "unstandardized"
        _filename += "-"
        _filename += self.optimizer_name
        _filename += "-results.txt"
        return _filename

    @property
    def to_string(self):
        return f’’’
        *** Config Settings ***
        Data Type:\t{self.data_type}
        Optimizer:\t{self.optimizer_name}
        Class Cnt:\t{self.num_classes}
        Is Multi-Class:\t{self.is_multi_classification}
        Epoch Cnt:\t{self.max_epochs}
        Standarize:\t{self.use_standardization}
        Filename:\t{self.result_filename}
        ’’’
```

```
# Define Neural Netowrk Model
class NeuralNetwork(torch.nn.Module):
    def __init__(self, input_size, output_size, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor

        # This is an actual neural layer...
        self.output_layer = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        # Here is where we use the layers to compute something...
        y = x # Start with the input
        y = self.output_layer(y) # y replaces y (stateful parameters)
        return y # Final calculation returned

    # Separate the final activation function out because
    # binary_cross_entropy assumes you are using a sigmoid
    # (outputs are considered logits - more later on this...)
    def predict(self, x):
        # Here is where we use the layers to compute something...
        y = x
        y = self.forward(y) # Start with the input
        y = torch.softmax(y, -1) if config.is_multi_classification else torch.sigmoid(y)
        return y # Final calculation returned


# Define Trainable Module
class PLModel(pl.LightningModule):
    def __init__(self, module, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor
        self.module = module

        # This creates an Accuracy and loss Functions
        # based on teh num_class being passed
        if(config.is_multi_classification):
            self.network_acc = torchmetrics.classification.Accuracy(task=’multiclass’, num_cla
sses=config.num_classes)
            self.network_loss = torch.nn.CrossEntropyLoss()
        else:
            self.network_acc = torchmetrics.classification.Accuracy(task=’binary’)
            self.network_loss = torch.nn.BCEWithLogitsLoss()

    def forward(self, x):
        return self.module.forward(x)

    def predict(self, x):
        return self.module.predict(x)

    def configure_optimizers(self):
        match config.optimizer_name:
            case "adam":
                return torch.optim.Adam(self.parameters(), lr=0.01)
            case "rmsprop":
                return torch.optim.RMSprop(self.parameters(), lr=0.01)
            case "sgd":
                return torch.optim.SGD(self.parameters(), lr=0.01)
            case _:
                return None
    def training_step(self, train_batch, batch_idx):
        x, y_true = train_batch
        # Handle Cross Entropy to Long vs. BCE is a Float
        y_true = torch.Tensor(y_true).type(torch.long) if config.is_multi_classification else
y_true
        y_pred = self(x)
        acc = self.network_acc(y_pred, y_true)
        loss = self.network_loss(y_pred, y_true)
        self.log(’train_acc’, acc, on_step=False, on_epoch=True)
        self.log(’train_loss’, loss, on_step=False, on_epoch=True)
        return loss

    def validation_step(self, val_batch, batch_idx):
        x, y_true = val_batch
        # Handle Cross Entropy to Long vs. BCE is a Float
        y_true = torch.Tensor(y_true).type(torch.long) if config.is_multi_classification else
y_true
        y_pred = self(x)
        acc = self.network_acc(y_pred, y_true)
        loss = self.network_loss(y_pred, y_true)
        self.log(’val_acc’, acc, on_step=False, on_epoch=True)
        self.log(’val_loss’, loss, on_step=False, on_epoch=True)
        return loss

    def testing_step(self, test_batch, batch_idx):
        x, y_true = test_batch
        # Handle Cross Entropy to Long vs. BCE is a Float
        y_true = torch.Tensor(y_true).type(torch.long) if config.is_multi_classification else
y_true
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log(’test_acc’, acc, on_step=False, on_epoch=True)
        self.log(’test_loss’, loss, on_step=False, on_epoch=True)
        return loss



# Main Section of the Code Application
if __name__ == ’__main__’:
    config = Config()
    config.parse_args()

    data = np.loadtxt(config.data_url)

    X = data[ : ,    : -1]
    Y = data[ : , -1 :   ]

    # Setup Number of Classes
    config.num_classes = len(np.unique(Y))

    # Display Config Settings
    print(config.to_string)

    def preprocess(x):
        return (x - np.mean(x)) / np.std(x)

    if (config.use_standardization):
        X = np.apply_along_axis(preprocess, 0, X)

    # Setting the Out Feature based onteh Classificaiotn Model used (Binary or Multi-Class)
    out_features = config.num_classes if config.is_multi_classification else Y.shape[-1]
    neural_net = NeuralNetwork(X.shape[-1], out_features).to(config.device)

    predictions = neural_net.predict(torch.Tensor(X[:5])) if config.is_multi_classification el
se neural_net.predict(torch.Tensor(X[:5,:]))
    predictions.detach().numpy()

    model = PLModel(neural_net).to(config.device)

    # Define a permutation needed to shuffle both # inputs and targets in the same manner....
    shuffle = np.random.permutation(X.shape[0])
    X_shuffled = X[shuffle, :]
    Y_shuffled = Y[shuffle, :]

    # Keep 70% for training and remaining for validation
    split_point = int(X_shuffled.shape[0] * 0.7)
    x_train = X_shuffled[:split_point]
    y_train = Y_shuffled[:split_point,0] if config.is_multi_classification else Y_shuffled[:sp
lit_point]

    x_val = X_shuffled[split_point:]
    y_val = Y_shuffled[split_point:,0] if config.is_multi_classification else Y_shuffled[split
_point:]

    # The dataloaders handle shuffling, batching, etc...
    xy_train = torch.utils.data.DataLoader(
        list(
            zip(
                torch.Tensor(x_train).type(torch.float),
                torch.Tensor(y_train).type(torch.float)
            )
        ),
        num_workers=4,
        shuffle=True,
        batch_size=32)

    xy_val = torch.utils.data.DataLoader(
        list(
            zip(
                torch.Tensor(x_val).type(torch.float),
                torch.Tensor(y_val).type(torch.float)
            )
        ),
        num_workers=4,
        shuffle=False,
        batch_size=32)

    logger = pl.loggers.CSVLogger("logs", name = "Single-Layer-Network",)

    trainer = pl.Trainer(
        max_epochs=config.max_epochs,
        logger=logger,
        enable_progress_bar=True,
        log_every_n_steps=0,
        callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])

    preliminary_result = trainer.validate(model, dataloaders=xy_val)

    trainer.fit(model, train_dataloaders=xy_train, val_dataloaders=xy_val)
    final_result = trainer.validate(model, dataloaders=xy_val)

    # Get Results from Logger
    results = pd.read_csv(f"{logger.log_dir}/metrics.csv", delimiter=’,’)
    # Calculate the expression and store it in a variable
    result_acc = ["%.8f"%(x) for x in results[’val_acc’][np.logical_not(np.isnan(results["val_
acc"]))][0::10]]
    result_acc_string = " ".join(result_acc)

    # Open the file in append mode
    with open(config.result_filename, ’a’) as file:
        # Append the result_string to a new line in the file
        file.write(result_acc_string + "\n")

    print("Validation accuracy:",*["%.8f"%(x) for x in results[’val_acc’][np.logical_not(np.is
nan(results["val_acc"]))][0::10]])
```

Looks good.

```
Testing Arguments - Data: iris Optimizer: adam, Standardization: 0
Validation accuracy: 0.37777779 0.77777779 0.82222223 0.86666667 0.86666667 0.88888890 0.91111
112 0.88888890 0.91111112 0.95555556 0.95555556
Testing Arguments - Data: iris Optimizer: adam, Standardization: 1
Validation accuracy: 0.66666669 0.80000001 0.88888890 0.91111112 0.93333334 0.93333334 0.93333
334 0.95555556 0.95555556 0.95555556 0.95555556
Testing Arguments - Data: iris Optimizer: rmsprop, Standardization: 0
Validation accuracy: 0.40000001 0.75555557 0.97777778 0.86666667 0.97777778 0.93333334 0.93333
334 0.95555556 0.95555556 0.95555556 0.95555556
Testing Arguments - Data: iris Optimizer: rmsprop, Standardization: 1
Validation accuracy: 0.06666667 0.86666667 0.88888890 0.88888890 0.88888890 0.88888890 0.88888
890 0.88888890 0.88888890 0.91111112 0.91111112
Testing Arguments - Data: iris Optimizer: sgd, Standardization: 0
Validation accuracy: 0.40000001 0.51111114 0.57777780 0.77777779 0.68888891 0.73333335 0.80000
001 0.93333334 0.82222223 0.95555556 0.82222223
Testing Arguments - Data: iris Optimizer: sgd, Standardization: 1
Validation accuracy: 0.46666667 0.68888891 0.71111113 0.75555557 0.75555557 0.80000001 0.82222
223 0.86666667 0.86666667 0.88888890 0.88888890
Testing Arguments - Data: wdbc Optimizer: adam, Standardization: 0
Validation accuracy: 0.57894737 0.91228068 0.92982459 0.92982459 0.92982459 0.93567252 0.93567
252 0.96491230 0.95906430 0.95906430 0.97076023
Testing Arguments - Data: wdbc Optimizer: adam, Standardization: 1
Validation accuracy: 0.81286550 0.95906430 0.97076023 0.97076023 0.96491230 0.95906430 0.96491
230 0.96491230 0.96491230 0.95906430 0.95906430
Testing Arguments - Data: wdbc Optimizer: rmsprop, Standardization: 0
Validation accuracy: 0.64327484 0.92982459 0.92982459 0.93567252 0.94736844 0.94736844 0.93567
252 0.96491230 0.95321637 0.95906430 0.97076023
Testing Arguments - Data: wdbc Optimizer: rmsprop, Standardization: 1
Validation accuracy: 0.12865497 0.98830408 0.98245615 0.97660822 0.98245615 0.97660822 0.98245
615 0.97660822 0.97660822 0.98245615 0.97660822
Testing Arguments - Data: wdbc Optimizer: sgd, Standardization: 0
Validation accuracy: 0.42105263 0.59064329 0.59649122 0.64327484 0.67251462 0.74269009 0.78947
371 0.83040935 0.85964912 0.85964912 0.87134504
Testing Arguments - Data: wdbc Optimizer: sgd, Standardization: 1
Validation accuracy: 0.13450292 0.94152045 0.95906430 0.96491230 0.96491230 0.96491230 0.96491
230 0.96491230 0.96491230 0.97076023 0.97076023
```