A3: NeuralNetwork Class

Requirements

In this assignment, you will complete the implementation of the `NeuralNetwork` class, based on your solution to `A2` and 08a Optimizers, using the code included in the next code cell. Your implementation must meet the requirements described in the doc-strings.

Run the code in 08a Optimizers to create the file `optimizers.py` for use in this assignment.

Then apply your `NeuralNetwork` class to the problem of predicting the rental of bicycles in Seoul as described below.

Code for `NeuralNetwork` Class Saved in File `neuralnetworkA3.py`

```
%%writefile neuralnetworkA3.py

import numpy as np
import optimizers as opt


class NeuralNetwork():
    """
    A class that represents a neural network for nonlinear regression.
    """

    def __init__(self, n_inputs, n_hiddens_each_layer, n_outputs):
        """Creates a neural network with the given structure."""

        self.n_inputs = n_inputs
        self.n_hiddens_each_layer = n_hiddens_each_layer
        self.n_outputs = n_outputs

        # Create list of shapes of weight matrices for each layer
        shapes = []
        input_size = n_inputs
        for n_hidden in n_hiddens_each_layer:
            shapes.append((input_size + 1, n_hidden))  # +1 for bias
term
            input_size = n_hidden
        shapes.append((input_size + 1, n_outputs))  # Output layer

        # Build one-dimensional vector of all weights and weight
matrices
        self.all_weights, self.Ws =
self._make_weights_and_views(shapes)

        # Build gradients similarly
```

```
        self.all_gradients, self.Grads =
self._make_weights_and_views(shapes)

        self.X_means = None
        self.X_stds = None
        self.T_means = None
        self.T_stds = None

        self.n_epochs = 0
        self.error_trace = []

    def _make_weights_and_views(self, shapes):
        """Creates vector of all weights and views for each layer."""
        # Create one-dimensional numpy array of all weights with
random initial values between -1 and 1.
        total_weights = sum((rows * cols) for rows, cols in shapes)
        all_weights = np.random.uniform(-1, 1, total_weights)

        # Build weight matrices as views by reshaping corresponding
elements from vector of all weights
        Ws = []
        start = 0
        for rows, cols in shapes:
            end = start + rows * cols
            W = np.reshape(all_weights[start:end], (rows, cols))
            W /= np.sqrt(rows)  # Divide by sqrt of number of inputs
            Ws.append(W)
            start = end

        return all_weights, Ws

    def __repr__(self):
        return f'NeuralNetwork({self.n_inputs},
{self.n_hiddens_each_layer}, {self.n_outputs})'

    def __str__(self):
        if self.n_epochs > 0:
            return f'{self.__repr__()} trained for {self.n_epochs}
epochs with a final RMSE of {self.error_trace[-1]}'
        else:
            return f'{self.__repr__()} has not been trained.'

    def train(self, Xtrain, Ttrain, Xvalidate, Tvalidate, n_epochs,
batch_size=-1,
              method='sgd', learning_rate=None, momentum=0,
weight_penalty=0, verbose=True):
        """Updates the weights."""

        self.batch_size = batch_size
```

```
        # Standardize Xtrain, Ttrain, Xvalidate and Tvalidate
        self.X_means = Xtrain.mean(axis=0)
        self.X_stds = Xtrain.std(axis=0)
        Xtrain_standardized = (Xtrain - self.X_means) / self.X_stds
        Xvalidate_standardized = (Xvalidate - self.X_means) /
self.X_stds

        self.T_means = Ttrain.mean(axis=0)
        self.T_stds = Ttrain.std(axis=0)
        Ttrain_standardized = (Ttrain - self.T_means) / self.T_stds
        Tvalidate_standardized = (Tvalidate - self.T_means) /
self.T_stds

        # Instantiate Optimizers object by giving it vector of all
weights
        optimizer = opt.Optimizers(self.all_weights)

        # Select optimization method
        if method == 'sgd':
            self.error_trace = optimizer.sgd(Xtrain_standardized,
Ttrain_standardized,
                                             Xvalidate_standardized,
Tvalidate_standardized,
                                             self.error_f,
self.gradient_f,
                                             n_epochs=n_epochs,
batch_size=batch_size,

learning_rate=learning_rate,
                                             momentum=momentum,
weight_penalty=weight_penalty,
                                             verbose=verbose)

        elif method == 'adam':
            self.error_trace = optimizer.adam(Xtrain_standardized,
Ttrain_standardized,
                                              Xvalidate_standardized,
Tvalidate_standardized,
                                              self.error_f,
self.gradient_f,
                                              n_epochs=n_epochs,
batch_size=batch_size,

learning_rate=learning_rate,

weight_penalty=weight_penalty, verbose=verbose)

        elif method == 'scg':
            self.error_trace = optimizer.scg(Xtrain_standardized,
Ttrain_standardized,
```

```
                                                            Xvalidate_standardized,
Tvalidate_standardized,
                                                            self.error_f,
self.gradient_f,
                                                            n_epochs=n_epochs,
batch_size=batch_size,

weight_penalty=weight_penalty, verbose=verbose)

        else:
            raise Exception("Method must be 'sgd', 'adam', or 'scg'")

        self.n_epochs += len(self.error_trace)
        self.best_epoch = optimizer.best_epoch

        return self

    def _add_ones(self, X):
        return np.insert(X, 0, 1, axis=1)

    def _forward(self, X):
        """Calculate outputs of each layer given inputs in X."""
        self.Zs = [self._add_ones(X)]
        for W in self.Ws[:-1]:
            Z = np.tanh(self.Zs[-1] @ W)
            self.Zs.append(self._add_ones(Z))
        self.Zs.append(self.Zs[-1] @ self.Ws[-1])
        return self.Zs

    def error_f(self, X, T):
        """Calculate mean squared error."""
        Y = self._forward(X)[-1]
        return np.mean((T - Y) ** 2)

    def gradient_f(self, X, T):
        """Return gradients with respect to all weights."""
        n_samples = X.shape[0]
        delta = -(T - self.Zs[-1]) / n_samples

        for layeri in range(len(self.Ws) - 1, -1, -1):
            self.Grads[layeri][:] = self.Zs[layeri].T @ delta
            if layeri > 0:
                delta = (delta @ self.Ws[layeri].T)[:, 1:] * (1 -
self.Zs[layeri][:, 1:] ** 2)

        return self.all_gradients

    def use(self, X):
        """Return the output of the network for input samples."""
        X_standardized = (X - self.X_means) / self.X_stds
```

```
        Y = self._forward(X_standardized)[-1]
        return Y * self.T_stds + self.T_means

    def get_error_trace(self):
        """Returns list of root-mean square error for each epoch."""
        return self.error_trace
```

```
Writing neuralnetworkA3.py
```

Example Results

Here we test the `NeuralNetwork` class with some simple data.

```
%load_ext autoreload
%autoreload 2

import numpy as np
import matplotlib.pyplot as plt

import neuralnetworkA3 as nn  # Your file produced from the above code
cell.
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
X = np.arange(0, 2, 0.5).reshape(-1, 1)
T = np.sin(X) * np.sin(X * 10)

nnet = nn.NeuralNetwork(X.shape[1], [2, 2], 1)

# Set all weights here to allow comparison of your calculations
# Must use [:] to overwrite values in all_weights.
# Without [:], new array is assigned to self.all_weights, so self.Ws
no longer refer to same memory
nnet.all_weights[:] = np.arange(len(nnet.all_weights)) * 0.001

nnet.train(X, T, X, T, n_epochs=1, batch_size=-1, method='sgd',
learning_rate=0.1)

nnet.Ws
```

```
SGD: Epoch 1 MSE=1.00008,1.00008
```

```
[array([[-1.51603124e-07,  9.99801782e-04],
        [ 2.00719909e-03,  3.00940664e-03]]),
 array([[0.00398889, 0.00498788],
        [0.00600106, 0.00700115],
        [0.00800157, 0.00900172]]),
 array([[0.00898958],
```

```
         [0.01099768],
         [0.01199691]])]
```

```
nnet.Zs
```

```
[array([[ 1.        , -1.34164079],
        [ 1.        , -0.4472136 ],
        [ 1.        ,  0.4472136 ],
        [ 1.        ,  1.34164079]]),
 array([[ 1.00000000e+00, -2.69308526e-03, -3.03773156e-03],
        [ 1.00000000e+00, -8.97798084e-04, -3.46045767e-04],
        [ 1.00000000e+00,  8.97494878e-04,  2.34564504e-03],
        [ 1.00000000e+00,  2.69278206e-03,  5.03730187e-03]]),
 array([[1.        , 0.0039484 , 0.00494164],
        [1.        , 0.00398071, 0.00497843],
        [1.        , 0.00401302, 0.00501523],
        [1.        , 0.00404533, 0.00505203]]),
 array([[0.00909229],
        [0.00909308],
        [0.00909388],
        [0.00909468]])]
```

```
nnet.Grads
```

```
[array([[ 1.51603124e-06,  1.98217653e-06],
        [-7.19909102e-05, -9.40663907e-05]]),
 array([[ 1.11145896e-04,  1.21249680e-04],
        [-1.05587719e-05, -1.15185542e-05],
        [-1.57268813e-05, -1.71564392e-05]]),
 array([[1.01041953e-02],
        [2.32194989e-05],
        [3.09340617e-05]])]
```

```
Y = nnet.use(X)
Y
```

```
array([[-0.06308723],
       [-0.06308687],
       [-0.06308651],
       [-0.06308615]])
```

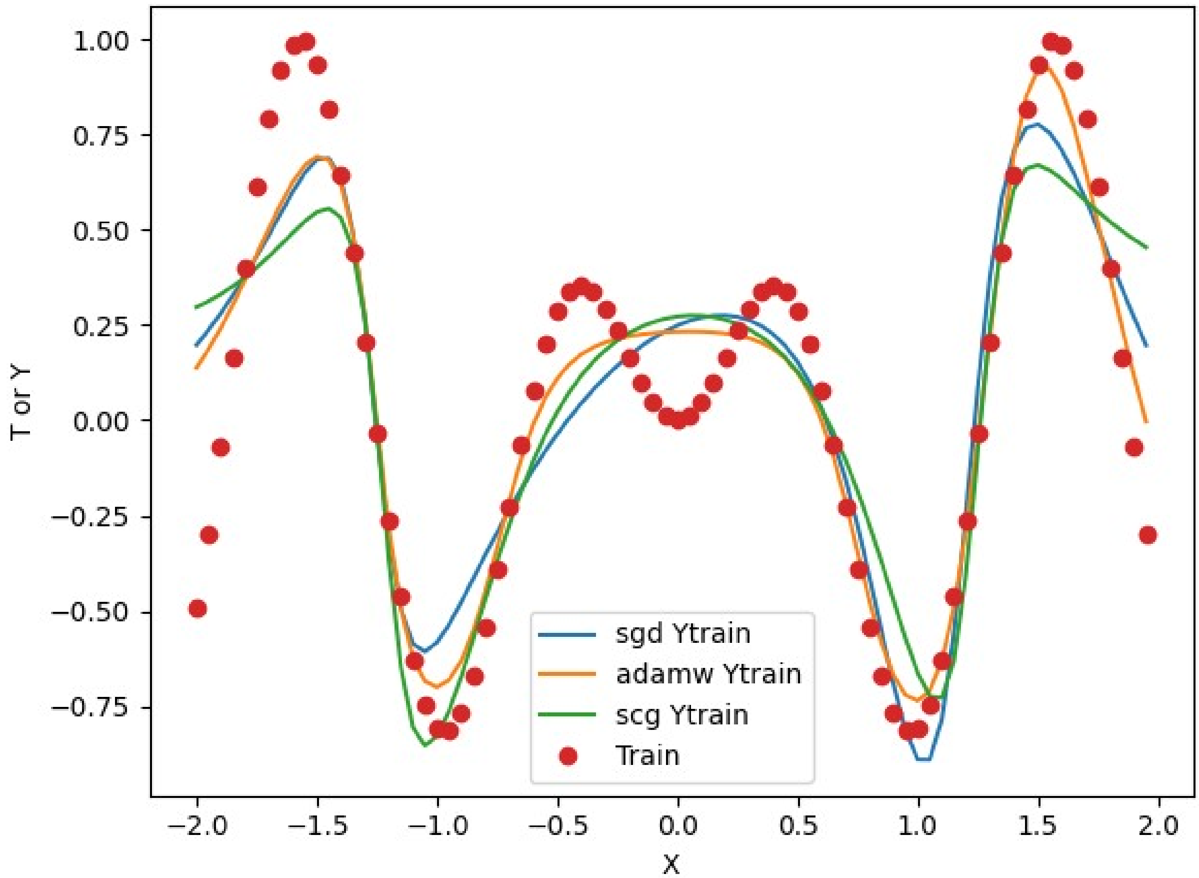
More Detailed Example Use

```
Xtrain = np.arange(-2, 2, 0.05).reshape(-1, 1)
Ttrain = np.sin(Xtrain) * np.sin(Xtrain * 5)

Xval = Xtrain * 1.1
Tval = Ttrain + 0.2 * Xtrain
Xtest = Xtrain * 0.97
```

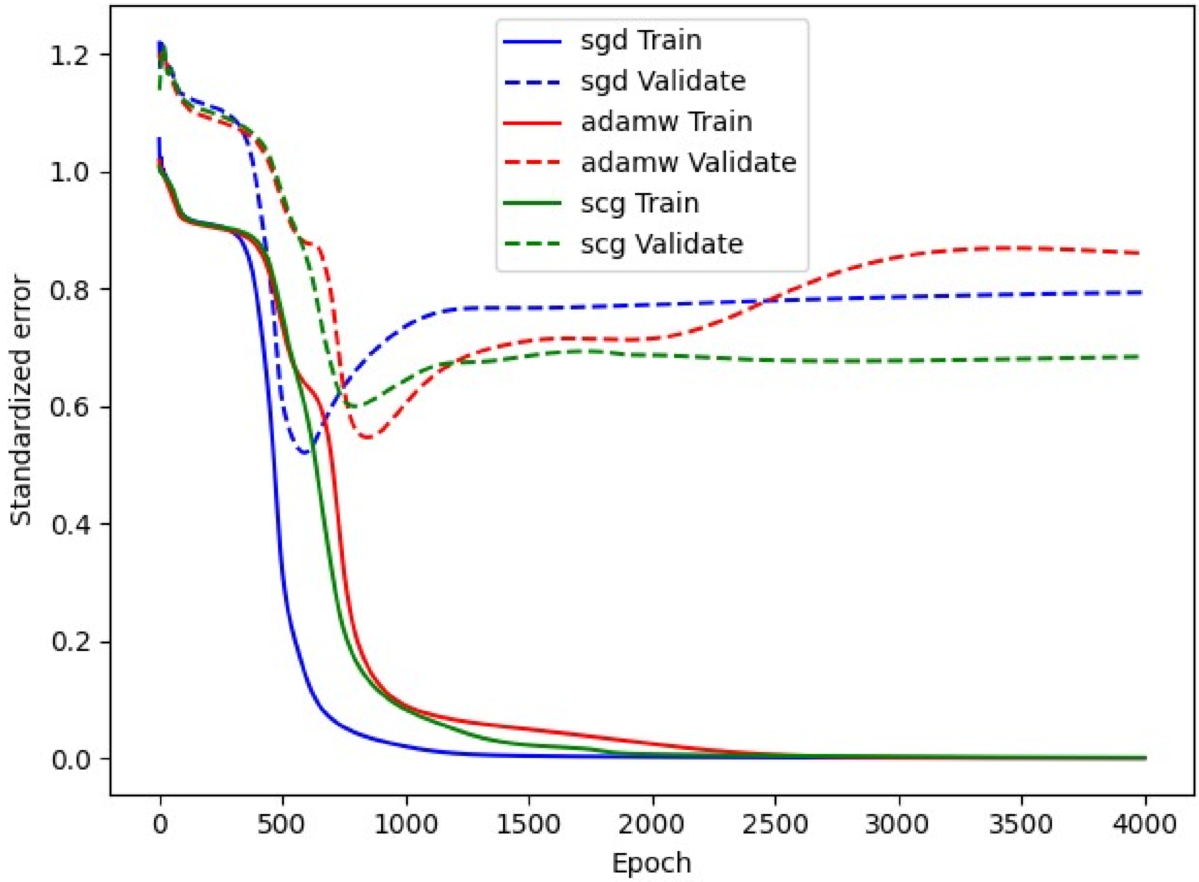
```
Ttest = Ttrain + 0.15 * Xtrain # + np.random.uniform(-0.05, 0.05,
Ttrain.shape)

errors = []
Ytests = []
n_epochs = 4000
method_rhos = [('sgd', 0.05),
               ('adamw', 0.005),
               ('scg', None)]

for method, rho in method_rhos:
    nnet = nn.NeuralNetwork(Xtrain.shape[1], [10, 10], 1)
    nnet.train(Xtrain, Ttrain, Xval, Tval, n_epochs, batch_size=-1,
method=method, learning_rate=rho,
               momentum=0.9)  # momentum only affects sgd)
    Ytrain = nnet.use(Xtrain)
    plt.plot(Xtrain, Ytrain, '-', label=method + ' Ytrain')
    errors.append(nnet.get_error_trace())
    Ytests.append(nnet.use(Xtest))

plt.plot(Xtrain, Ttrain, 'o', label='Train')
plt.xlabel('X')
plt.ylabel('T or Y')
plt.legend()
plt.tight_layout()
```

```
SGD: Epoch 400 MSE=0.88142,1.05762
SGD: Epoch 800 MSE=0.16605,0.60035
SGD: Epoch 1200 MSE=0.05000,0.67452
SGD: Epoch 1600 MSE=0.02011,0.69130
SGD: Epoch 2000 MSE=0.00745,0.68737
SGD: Epoch 2400 MSE=0.00538,0.67984
SGD: Epoch 2800 MSE=0.00408,0.67743
SGD: Epoch 3200 MSE=0.00308,0.67889
SGD: Epoch 3600 MSE=0.00230,0.68161
SGD: Epoch 4000 MSE=0.00174,0.68458
AdamW: Epoch 400 MSE=0.02240,0.69452
AdamW: Epoch 800 MSE=0.01500,0.72812
AdamW: Epoch 1200 MSE=0.00082,0.72829
AdamW: Epoch 1600 MSE=0.00022,0.73659
AdamW: Epoch 2000 MSE=0.00016,0.73690
AdamW: Epoch 2400 MSE=0.00010,0.73939
AdamW: Epoch 2800 MSE=0.00008,0.73959
AdamW: Epoch 3200 MSE=0.00006,0.73808
AdamW: Epoch 3600 MSE=0.00005,0.73775
AdamW: Epoch 4000 MSE=0.00004,0.73847
SCG: Epoch 0 MSE=1.01037,1.32067
SCG: Epoch 400 MSE=0.00023,0.81317
SCG: Epoch 800 MSE=0.00001,0.75998
SCG: Epoch 1200 MSE=0.00000,0.75285
```

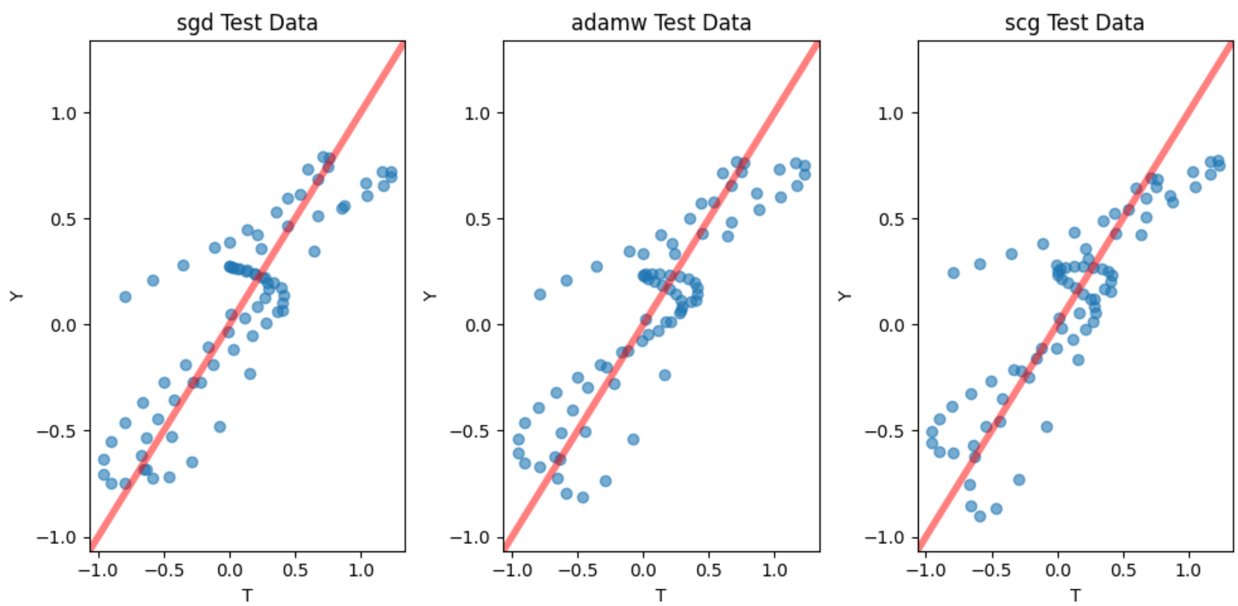
```
SCG: Epoch 1600 MSE=0.00000,0.75314
SCG: Epoch 2000 MSE=0.00000,0.75426
SCG: Epoch 2400 MSE=0.00000,0.75446
SCG: Epoch 2800 MSE=0.00000,0.75401
SCG: Epoch 3200 MSE=0.00000,0.75324
SCG: Epoch 3600 MSE=0.00000,0.75347
```




```
errors = np.stack(errors)
# errors is now 3 x n_epochs x 2
colors = ['b', 'r', 'g']
styles = ['-', '--']
for methodi, method in enumerate([mr[0] for mr in method_rhos]):
    for train_val_i, train_val in enumerate(['Train', 'Validate']):
        plt.plot(errors[methodi, :, train_val_i], f'{colors[methodi]}
{styles[train_val_i]}',
                 label=f'{method} {train_val}')
plt.xlabel('Epoch')
plt.ylabel('Standardized error')
plt.legend()
plt.tight_layout()
```




```
def plot_Y_vs_T(Y, T, title):
    plt.plot(T, Y, 'o', alpha=0.6)
    a = min(min(T), min(Y))[0]
    b = max(max(T), max(Y))[0]
    plt.axline((a, a), (b, b), linewidth=4, color='r', alpha=0.5)
    plt.xlabel('T')
    plt.ylabel('Y')
    plt.title(title)

plt.figure(figsize=(10, 5))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plot_Y_vs_T(Ytests[i], Ttest, f'{method_rhos[i][0]} Test Data')
plt.tight_layout()
```



Application

Use your neural network implementation to create a model for predicting the critical temperature of superconductive materials based on attributes of the materials. Download and extract the data from this UCI ML Repository site. This site explains the data and has a link to an introductory paper. The data consists of 81 attributes extracted from 21,263 superconductors in the first 81 columns and the critical temperature for each in the 82nd column. So, the first 81 columns will form your input X matrix and the last column will be your target T matrix.

1. Your task is to do the following. Partition the data into partitions of 60%, 20% and 20% for the training, validation and test sets, respectively. Try training with each of the three optimization methods and reasonable values for the other parameters. Plot the `error_trace`s for example runs of each of three methods. Discuss what you see in the plots.
1. Write code using nested for loops to iterate over all three optimization methods, several hidden layer structures, several numbers of epochs, several learning rates, and several batch sizes. In a list of lists, collect the method, number of epochs, learning rate, batch size, and RMSEs for training, validation, and test data. After all for loops have completed, convert the resulting list of lists into a `pandas.DataFrame` with appropriate column names. Sort it by ascending test set RMSEs and print the `DataFrame`. It may be helpful to also do this for each iteration of the outer-most for loop. You should set `verbose=False` in the call to `NeuralNetwork.train` to reduce the amount of printing. Discuss the set of parameter values and all three RMSE values that produce some of the lowest test RMSEs. To debug this code, use very small numbers of epochs.
1. Train another network using the best parameter values shown in your results. In three separate plots, plot the predicted critical temperature versus the actual (target) critical temperatures for the training, validation, and test sets. Discuss what you see. How well does your neural network predict the critical temperatures?

Grading

Your notebook will be run and graded automatically. Test this grading process by first downloading A3grader.zip and extract `A3grader.py` from it. Run the code in the following cell to demonstrate an example grading session. As always, a different, but similar, grading script will be used to grade your checked-in notebook. It will include additional tests. You should design and perform additional tests on all of your functions to be sure they run correctly before checking in your notebook.

For the grading script to run correctly, you must first name this notebook as 'A3solution.ipynb' (lower case s) and then save this notebook. Check in your notebook in Canvas.

```
%run -i A3grader.py
```

```
======================= Code Execution =======================

Extracting python code from notebook named 'A3solution.ipynb' and
storing in notebookcode.py
Removing all statements that are not function or class defs or import
statements.


======================================================================
==========
Testing this for 5 points:

def check_weight_views(nnet):
    results = []
    for layeri, W in enumerate(nnet.Ws):
        if np.shares_memory(nnet.all_weights, W):
            print(f'nnet.Ws[{layeri}] correctly shares memory with
nnet.all_weights')
            results.append(True)
        else:
            print(f'nnet.Ws[{layeri}] does not correctly share memory
with nnet.all_weights')
            results.append(False)

    return np.all(results)

n_inputs = 3
n_hiddens = [12, 8, 4]
n_outputs = 2

nnet = nn.NeuralNetwork(n_inputs, n_hiddens, n_outputs)


#  and test result with    check_weight_views(nnet)
```

```
----------------------------------------------------------------------
----  5/5 points. Weight views are correctly defined
----------------------------------------------------------------------


======================================================================
==========
Testing this for 5 points:

nnet = nn.NeuralNetwork(3, [], 4)


#  and test result with    check_weight_views(nnet)

----------------------------------------------------------------------
----  5/5 points. Weight views are correctly defined
----------------------------------------------------------------------


======================================================================
==========
Testing this for 5 points:

def check_gradient_views(nnet):
    results = []
    for layeri, G in enumerate(nnet.Grads):
        if np.shares_memory(nnet.all_gradients, G):
            print(f'nnet.Grads[{layeri}] correctly shares memory with
nnet.all_gradients')
            results.append(True)
        else:
            print(f'nnet.Grads[{layeri}] does not correctly share
memory with nnet.all_gradients')
            results.append(False)

    return np.all(results)

n_inputs = 3
n_hiddens = [5, 10, 20]
n_outputs = 2

nnet = nn.NeuralNetwork(n_inputs, n_hiddens, n_outputs)


#  and test result with    check_gradient_views(nnet)

----------------------------------------------------------------------
----  5/5 points. Gradient views are correctly defined
----------------------------------------------------------------------
```

```
======================================================================
==========
Testing this for 15 points:

n_inputs = 3
n_hiddens = [5, 10, 20]
n_outputs = 2
n_samples = 10

X = np.arange(n_samples * n_inputs).reshape(n_samples, n_inputs) * 0.1

nnet = nn.NeuralNetwork(n_inputs, n_hiddens, n_outputs)
nnet.all_weights[:] = 0.1  # set all weights to 0.1
nnet.X_means = np.mean(X, axis=0)
nnet.X_stds = np.std(X, axis=0)
nnet.T_means = np.zeros((n_samples, n_outputs))
nnet.T_stds = np.ones((n_samples, n_outputs))

Y = nnet.use(X)

Y_answer = np.array([[0.14629519, 0.14629519],
                     [0.24029528, 0.24029528],
                     [0.33910878, 0.33910878],
                     [0.43981761, 0.43981761],
                     [0.53920896, 0.53920896],
                     [0.63421852, 0.63421852],
                     [0.72233693, 0.72233693],
                     [0.80186297, 0.80186297],
                     [0.87195874, 0.87195874],
                     [0.93254   , 0.93254   ]])


#  and test result with    np.allclose(Y, Y_answer, 0.1)

----------------------------------------------------------------------
----  15/15 points. nnet.use returned correct values.
----------------------------------------------------------------------


======================================================================
==========
Testing this for 20 points:

n_inputs = 3
n_hiddens = [6, 3]
n_samples = 5

X = np.arange(n_samples * n_inputs).reshape(n_samples, n_inputs) * 0.1
T = np.log(X + 0.1)
n_outputs = T.shape[1]
```

```
def rmse(A, B):
    return np.sqrt(np.mean((A - B)**2))

results = []
for rep in range(20):
    nnet = nn.NeuralNetwork(n_inputs, n_hiddens, n_outputs)
    nnet.train(X, T, X, T, 2000, batch_size=-1, method='adamw',
learning_rate=0.001, verbose=False)
    Y = nnet.use(X)
    err = rmse(Y, T)
    print(f'Net {rep+1} RMSE {err:.5f}')
    results.append(err)

mean_rmse = np.mean(results)
print(mean_rmse)

Net 1 RMSE 0.01317
Net 2 RMSE 0.00885
Net 3 RMSE 0.01277
Net 4 RMSE 0.01115
Net 5 RMSE 0.01299
Net 6 RMSE 0.00767
Net 7 RMSE 0.01423
Net 8 RMSE 0.01402
Net 9 RMSE 0.01332
Net 10 RMSE 0.01348
Net 11 RMSE 0.00780
Net 12 RMSE 0.00789
Net 13 RMSE 0.01483
Net 14 RMSE 0.01425
Net 15 RMSE 0.01416
Net 16 RMSE 0.01335
Net 17 RMSE 0.01247
Net 18 RMSE 0.01310
Net 19 RMSE 0.01351
Net 20 RMSE 0.01499
0.012401367053791534

#  and test result with    0.0 < mean_rmse < 0.1

----------------------------------------------------------------------
----  20/20 points. mean_rmse is correct value.
----------------------------------------------------------------------


======================================================================
==========
Testing this for 20 points:
```

```
n_inputs = 3
n_hiddens = [10, 10, 5]
n_samples = 5

X = np.arange(n_samples * n_inputs).reshape(n_samples, n_inputs) * 0.1
T = 2 + np.log(X + 0.1)
Xval = X + np.random.normal(0.0, 0.1, size=X.shape)
Tval = 2.1 + np.log(Xval + 0.1)
n_outputs = T.shape[1]

def rmse(A, B):
    return np.sqrt(np.mean((A - B)**2))

results = []
for rep in range(20):
    nnet = nn.NeuralNetwork(n_inputs, n_hiddens, n_outputs)
    nnet.train(X, T, Xval, Tval, 3000, batch_size=-1, method='adamw',
learning_rate=0.1, verbose=False)
    Y = nnet.use(X)
    err = rmse(Y, T)
    print(f'Net {rep+1} RMSE {err:.5f} best epoch {nnet.best_epoch}')
    results.append(err)

mean_rmse = np.mean(results)
print(mean_rmse)

Net 1 RMSE 0.10264 best epoch 42
Net 2 RMSE 0.10706 best epoch 32
Net 3 RMSE 0.07375 best epoch 1847
Net 4 RMSE 0.15587 best epoch 26
Net 5 RMSE 0.10362 best epoch 34
Net 6 RMSE 0.06920 best epoch 2453
Net 7 RMSE 0.09906 best epoch 526
Net 8 RMSE 0.11209 best epoch 25
Net 9 RMSE 0.07596 best epoch 20
Net 10 RMSE 0.11298 best epoch 275
Net 11 RMSE 0.03252 best epoch 1902
Net 12 RMSE 0.11016 best epoch 815
Net 13 RMSE 0.07999 best epoch 34
Net 14 RMSE 0.06414 best epoch 1065
Net 15 RMSE 0.14334 best epoch 23
Net 16 RMSE 0.07890 best epoch 46
Net 17 RMSE 0.10997 best epoch 20
Net 18 RMSE 0.08706 best epoch 25
Net 19 RMSE 0.12045 best epoch 30
Net 20 RMSE 0.12027 best epoch 22
0.09795178394637534

#  and test result with    0.005 < mean_rmse < 0.2
```

```
--------------------------------------------------------------------------
----  20/20 points. mean_rmse returned correct value.
--------------------------------------------------------------------------

==========================================================================
A3 Execution Grade is 70 / 70

 REMEMBER, YOUR FINAL EXECUTION GRADE MAY BE DIFFERENT,
 BECAUSE DIFFERENT TESTS WILL BE RUN.
==========================================================================

Application Results:

___ / 10  1. Train with each of the three optimization, plot the
error_traces for each
             of the three methods.  Discussion of what you see in the
plots.
___ / 10  2. Use nested for loops to test various parameter values.
Collect results in a
             DataFrame. Discussion of the set of parameter values and
all three RMSE values
             that produce some of the lowest test RMSEs
___ / 10  3. Using best parameter values found, plot predicted
critical temperature versus
             the actual (target) critical temperatures for the
training, validation, and test sets.
             Discuss what you see. How well does your neural network
predict the critical temperatures?

==========================================================================
A3 Experiments and Discussion Grade is __ / 30
==========================================================================

==========================================================================
A3 FINAL GRADE is  _  / 100
==========================================================================

Extra Credit:  Code and discussion showing most significant input
features, and results after removing half of the least significant
features.


A3 EXTRA CREDIT is 0 / 1
```

Extra Credit

Using a network that gives you pretty good test RMSE results, try to figure out which input features are most significant in predicting the critical temperature. Remember, that our neural networks are trained with standardized inputs, so you can compare the magnitudes of weights in the first layer to help you determine which inputs are most significant.

To visualize the weights, try displaying the weights in the first layer as an image, with `plt.imshow` with `plt.colorbar()`. Discuss which weights have the largest magnitudes and discuss any patterns you see in the weights in each hidden unit of the first layer.

Retrain your neural network after removing half of the inputs for which the first layer of your network has the lowest mean absolute weights. Discuss how this affects the three RMSE values.