

IN3050/IN4050 Mandatory Assignment 2, 2022: Supervised Learning

Rules

Before you begin the exercise, review the rules at this website: <https://www.uio.no/english/studies/examinations/compulsory-activities/mn-if-mandatory.html>, in particular the paragraph on cooperation. This is an individual assignment, regarding how the data are generated). We choose to use a synthetic dataset, as a set of natural occurring data--because we are mostly interested in properties of the various learning algorithms, in particular the differences between linear classifiers and multi-layer neural networks together with the difference between binary and multi-class data.

Delivery

Deadline: Friday, March 25, 2022, 23:59

Your submission should be delivered in Devilly. You may redeliver in Devilly before the deadline, but include all files in the last delivery, as only the last delivery will be read. You are recommended to upload preliminary versions (or days) before the final deadline.

What to deliver?

You are recommended to solve the exercise in a Jupyter notebook, but you might solve it in a Python program if you prefer.

If you choose Jupyter, you should deliver the notebook. You should answer all questions and explain what you are doing in Markdown. Still, the code should be properly commented. The notebook should contain results of your runs. In addition, you should make a pdf of your solution which shows the results of the runs. (If you can't export: notebook -> latex -> pdf on your own machine, you may do this on the IF linux machines.)

If you prefer not to use notebooks, you should deliver the code, your run results, and a pdf-report where you answer all the questions and explain your work.

Your report/notebook should contain your name and username.

Deliver one single zipped folder (zip, .tgz or .tar.gz) which contains your complete solution.

Important: If you weren't able to finish the assignment, use the PDF report/Markdown to elaborate on what you've tried and what problems you encountered. Students who have made an effort and attempted all parts of the assignment will get a second chance even if they fail initially. This exercise will be graded PASS/FAIL.

Goals of the assignment

The goal of this assignment is to get a better understanding of supervised learning with gradient descent. It will, in particular, consider the similarities and differences between linear classifiers and multi-layer feed forward networks (multi-layer perceptron, MLP) and the differences and similarities between binary and multi-class classification. A main part will be dedicated to implementing and understanding the backpropagation algorithm.

Tools

The aim of the exercises is to give you a look inside the learning algorithms. You may freely use code from the weekly exercises and the published solutions. You should not use ML libraries like scikit-learn or tensorflow.

You may use tools like NumPy and Pandas, which are not specific ML-tools.

The given precode uses NumPy. You are recommended to use NumPy since it results in more compact code, but feel free to use pure python if you prefer.

Beware

There might occur typos or ambiguities. This is a revised assignment compared to earlier years, and there might be new typos. If anything is unclear, do not hesitate to ask. Also, if you think some assumptions are missing, make your own and explain them!

Initialization

```
In [4]: import numpy as np
import matplotlib.pyplot as plt
import sklearn #for datasets
import sys
import random
```

Part 1: Linear classifiers

Datasets

We start by making a synthetic dataset of 2000 datapoints and five classes, with 400 individuals in each class. (See https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html regarding how the data are generated). We choose to use a synthetic dataset, as a set of natural occurring data--because we are mostly interested in properties of the various learning algorithms, in particular the differences between linear classifiers and multi-layer neural networks together with the difference between binary and multi-class data.

When we are doing experiments in supervised learning, and the data are not already split into training and test sets, we should start by splitting the data. Sometimes there are natural ways to split the data, say training on data from one year and testing on data from a later year, but if that is not the case, we should shuffle the data randomly before splitting. (OK, that is not necessary with this particular synthetic data set, since it is already shuffled by default by scikit, but that will not be the case with real-world data.) We should split the data so that we keep the alignment between X and t , which must be achieved by shuffling the indices. We split into 50% for training, 25% for validation, and 25% for final testing. The set for final testing must not be used until the end of the assignment in part 3.

We fix the seed both for data set generation and for shuffling, so that we work on the same datasets when we rerun the experiments. This is done by the `random_state` argument and the `rng = np.random.RandomState(2822)`.

```
In [1]: from sklearn.datasets import make_blobs
X, t = make_blobs(n_samples=[400,400,400, 400, 400], centers=[(0,1],[4,1],[8,1],[2,0],[6,0]],
                n_features=2, random_state=2019, cluster_std=1.0)
```

```
In [5]: indices = np.arange(X.shape[0])
rng = np.random.RandomState(2022)
rng.shuffle(indices)
Indices = indices
```

```
Out[5]: array([1018, 1295, 643, 1842, 1669, 86, 164, 1653, 1174, 747])
```

```
In [6]: X_train = X[indices[:1000],:]
X_val = X[indices[1000:1500],:]
X_test = X[indices[1500:,:]]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]

print(X_train)

[[ 3.84817372  0.17753467]
 [ 7.3392557  1.3590733]
 [ 1.23913999  0.9230505]
 ...
 [ 1.0515982  1.74618291]
 [ 2.82461945  0.91575814]
 [ 3.85576765 -1.43849511]]
```

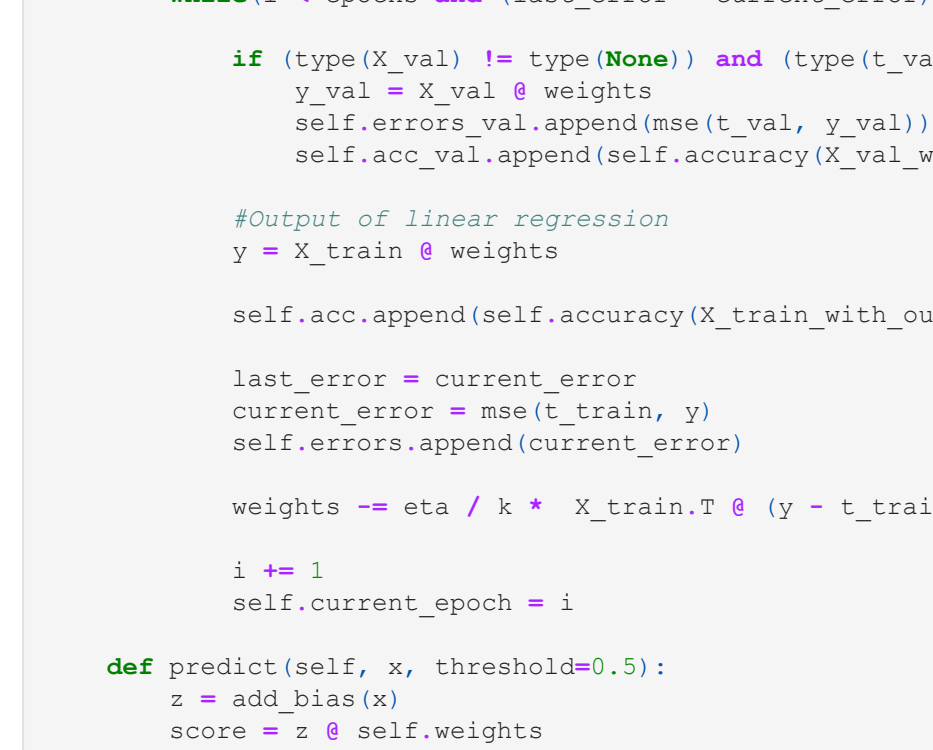
Next, we will make a second dataset by merging the two smaller classes in (X,t) and call the new set (X,t_2) . This will be a binary set.

```
In [7]: t2_train = t_train >= 3
t2_test = t_test.astype('int')
t2_val = (t_val >= 3).astype('int')
t2_test = t2_test >= 3).astype('int')
```

We can plot the two training sets.

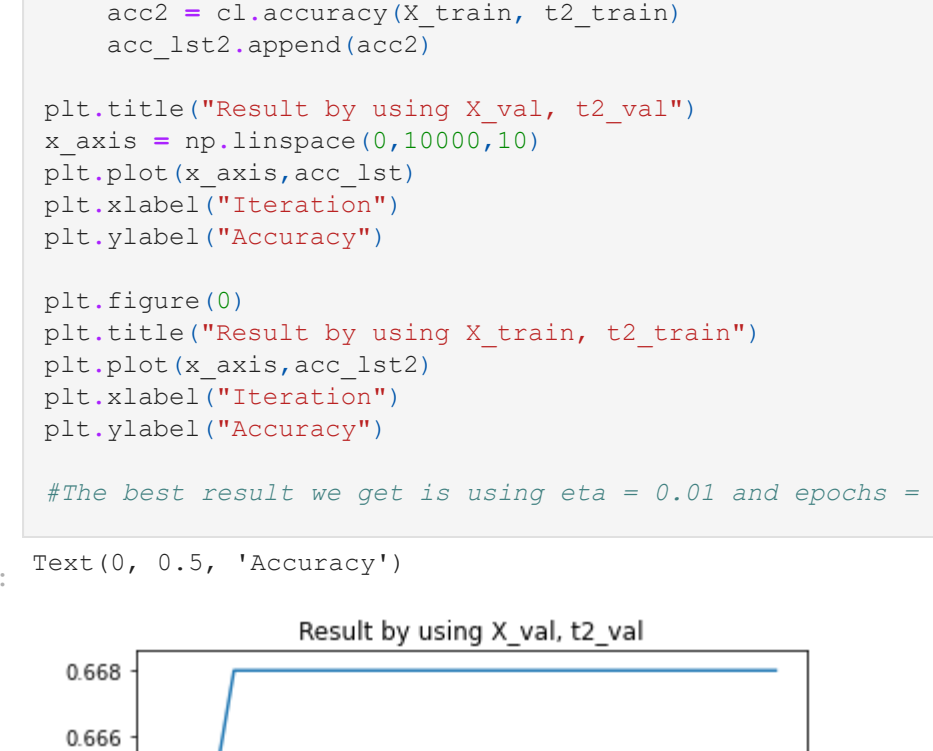
```
In [8]: plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=20.0)
plt.show()

plt.figure()
plt.show()
```



```
<Figure size 432x288 with 0 Axes>
```

```
In [9]: #This is the (X,t2) set, that only have 2 classes
plt.figure(figsize=(8,6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=20.0)
plt.show()
```



Binary classifiers

Linear regression

We see that that set (X,t_2) is far from linearly separable, and we will explore how various classifiers are able to handle this. We start with linear regression. You may make your own implementation from scratch or start with the solution to the weekly exercise set 7, which we include here.

```
In [10]: def mse(t,y):
    """
    Args:
        t = array
        y = array

    Calculation of mean squared error.
    Take a target array t, and a output array y.
    This function will calculate error between them and return the total sum of error.
    """
    sum_tot = 0
    for j in range(len(t)):
        t_j = t[j]
        y_j = y[j]
        sum_tot += (t_j-y_j)**2

    sum_tot = 1/len(t)*sum_tot
    return sum_tot
```

```
In [11]: def add_bias(X):
    """
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X])
    else:
        # X is a matrix
        n = sh[0]
        bias = np.ones((n,1)) # Makes a n*1 matrix of 1's
        return np.concatenate([bias, X], axis = 1)
```

```
In [12]: class NumpyClassifier():
    """Common methods to all numpy classifiers --- if any"""

    def accuracy(self, X_test, y_test, **kwargs):
        pred = self.predict(X_test, **kwargs)
        if len(pred.shape) > 1:
            pred = pred[:,0]
        return np.sum(pred==y_test)/len(pred)
```

```
In [13]: class NumpyLinRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=10, loss_diff = 0.000000001, X_val = None, t_val = None,
           **kwargs):
        """X_train is a NxM matrix, X data points, n features
        t_train are the targets values for training data"""
        X_train = X_train.reshape(-1,X_train.shape[-1])
        X_train_with_out_bias = X_train
        X_train = add_bias(X_train)

        if (type(X_val) != type(None)) and (type(t_val) != type(None)):
            X_val_with_out_bias = X_val
            X_val = add_bias(X_val)

        self.weights = weights = np.zeros(n+1)
        self.errors = []
        self.errors_val = []

        self.acc = []
        self.acc_val = []

        i = 0
        last_error = np.Inf
        current_error = sys.maxsize
        while(i < epochs and (last_error - current_error) > loss_diff):
            if (type(X_val) != type(None)) and (type(t_val) != type(None)):
                y_val = X_val @ weights
                self.errors_val.append(abs(t_val, y_val))
                self.acc_val.append(self.accuracy(X_val_with_out_bias, t_val))

            #Output of linear regression
            y = X_train @ weights

            self.acc.append(self.accuracy(X_train_with_out_bias, t2_train))

            last_error = current_error
            current_error = mse(t_train, y)
            self.errors.append(current_error)

            weights -= eta / k * X_train.T @ (y - t_train)

            i += 1
            self.current_epoch = i

        def predict(self, x, threshold=0.5):
            z = add_bias(x)
            score = z @ self.weights
            return score>threshold
```

We can train and test a first classifier.

```
In [12]: cl = NumpyLinRegClass()
cl.fit(X_train, t2_train)
cl.accuracy(X_val, t2_val)
```

```
Out[12]: 0.572
```

```
In [98]: """
Do not try to run this, it takes way too long,
use this before I begin the assignment to see accuracy in different epochs.
"""
acc_list = []
acc_list2 = []
n = 0
```

```
for i in range(10):
    n += 1000
    cl = NumpyLinRegClass()
    cl.fit(X_train, t2_train, eta = 0.01, epochs = n)
    acc = cl.accuracy(X_val, t2_val)
    acc_list.append(acc)

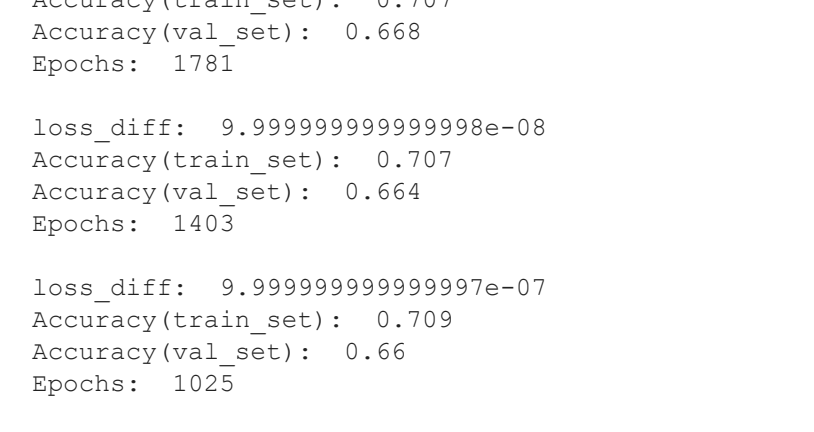
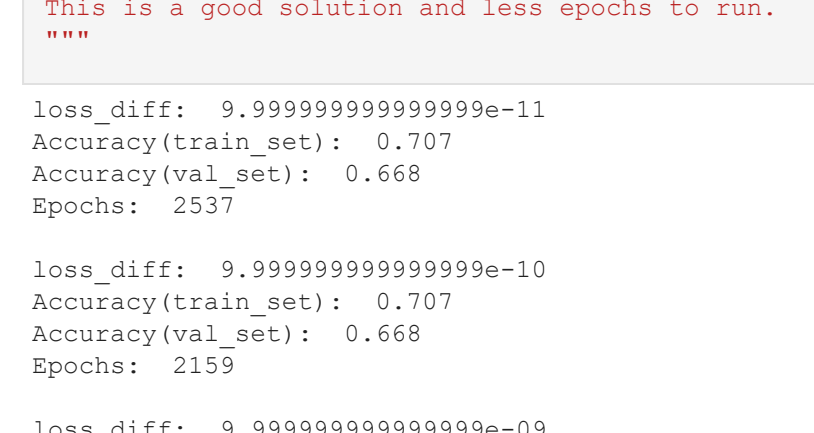
    cl2 = NumpyLinRegClass()
    cl2.fit(X_train, t2_train, eta = 0.01, epochs = n)
    acc2 = cl2.accuracy(X_train, t2_train)
    acc_list2.append(acc2)

plt.title("Result by using X_val, t2_val")
x_axis = np.linspace(0,10000,10)
plt.plot(x_axis,acc_list)
plt.xlabel("Iteration")
plt.ylabel("Accuracy")

plt.figure()
plt.title("Result by using X_train, t2_train")
plt.plot(x_axis,acc_list2)
plt.xlabel("Iteration")
plt.ylabel("Accuracy")

#The best result we get is using eta = 0.01 and epochs = 2000
```

```
Out[98]: Text(0, 0.5, 'Accuracy')
```



The result is far from impressive. Experiment with various settings for the hyper-parameters, eta and epochs. Report how the accuracy vary with the hyper-parameter settings. When you are satisfied with the result, you may plot the decision boundaries, as below.

Feel free to improve the colors and the rest av of the graphics. We have chosen a simple set-up which can be applied to more than two classes without substantial modifications.

```
In [14]: def plot_decision_regions(X, t, clf=[], size=(8,6)):
    """
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    h = 0.02 # step size in the mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    plt.figure(figsize=size) # You may adjust this

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)

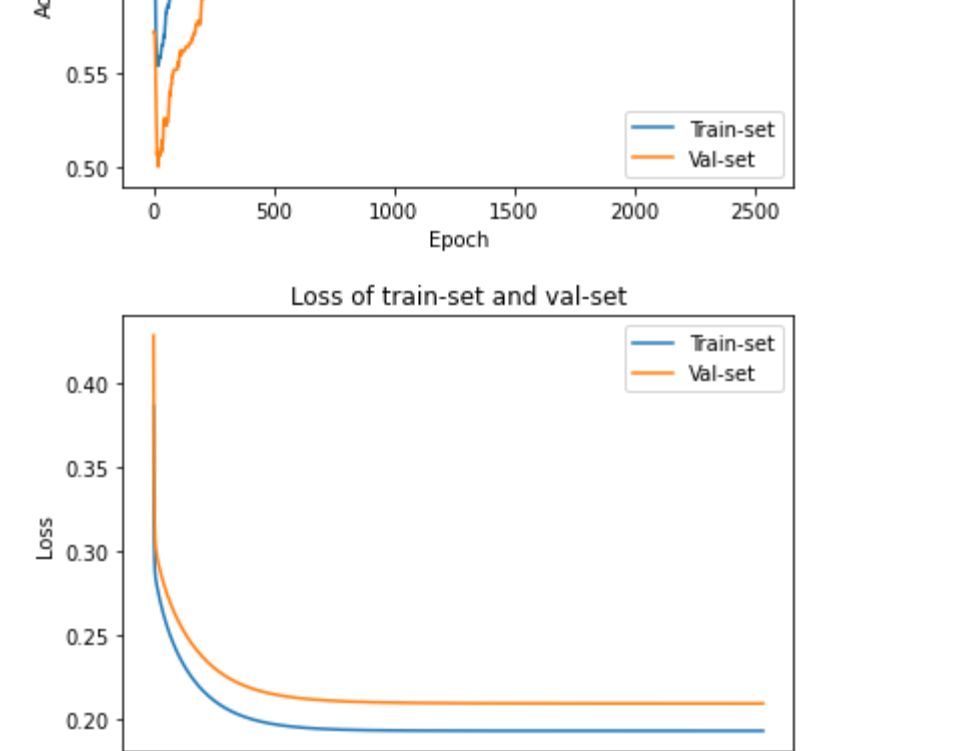
    plt.contourf(xx, yy, Z, alpha=0.2, cmap = 'Paired')

    plt.scatter(X[:,0], X[:,1], c=t, s=20.0, cmap='Paired')

    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision regions")
    plt.xlabel("x0")
    plt.ylabel("x1")

    plt.show()
```

```
In [18]: cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, eta = 0.01, epochs = 2000)
plot_decision_regions(X_train, t2_train, cl)
```



Loss

The linear regression classifier is trained with mean squared error loss. So far, we have not calculated the loss explicitly in the code. Extend the code to calculate the loss on the training set for each epoch and to store the losses such that the losses can be inspected after training.

Train a classifier with your best settings from last point. After training, plot the loss as a function of the number of epochs.

```
In [ ]: #Added function mse, for calculate loss function
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, eta = 0.01, epochs = 2000, loss_diff = 0.000000001)

plt.title("Mean square error")
x_axis = np.linspace(0,2000,2000)
plt.plot(x_axis,cl.errors)
plt.xlabel("Epoch")
plt.ylabel("Errors")
```

Control training

The training runs for a number of epochs. We cannot know beforehand for how many epochs it is reasonable to run the training. One possibility is to run the training until the learning does not improve much. Extend the fit-method with a keyword argument, `loss_diff`, and stop training when the loss has not improved with more than `loss_diff`. Also add an attribute to the classifier which tells us after fitting how many epochs were run.

In addition, extend the fit-method with optional arguments for a validation set (X_{val}, t_{val}) . If a validation set is included in the call to fit, calculate the loss for the validation set, and the accuracy for both the training set and the validation set for each epoch.

Train classifiers with the best value for learning rate η and with varying values for `loss_diff`. For each report, `loss_diff`, accuracy and number of epochs.

After a successful training, plot both training loss and validation loss as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure. Comment on what you see.

```
In [15]: """
Short about the implementation of loss_diff.
The loss_diff will force the program to get out of the loop, if last_error-current_error>loss_diff.
The implementation need a manual start, by initial all values for epoch number 1.
Epoch indexing start from 1, so by setting args epochs = 2000, the last epoch index is 2000.
"""

def different_loss_diff():
    """
    This function try out different loss_diff, and print them.
    """

    acc_list = []
    acc_list2 = []
    epochs_list = []

    cl_list = []
    n = 0.0000000001
    for i in range(10):
        n += 1000
        cl = NumpyLinRegClass()
        #Get epoch i stopped at, when training ends, and get accuracy for train-set and val-set
        cl.fit(X_train, t2_train, eta = 0.01, epochs = 3000, loss_diff = n, X_val = X_val, t_val = t2_val)
        acc = cl.accuracy(X_train, t2_train)
        acc_list.append(acc)

        acc2 = cl.accuracy(X_val, t2_val)
        acc_list2.append(acc2)
        epochs_list.append(cl.current_epoch)

        print("loss_diff: ", n)
        print("Accuracy(train_set): ", acc)
        print("Accuracy(val_set): ", acc2)
        print("Epochs: ", cl.current_epoch)
        plot_acc_loss()
```

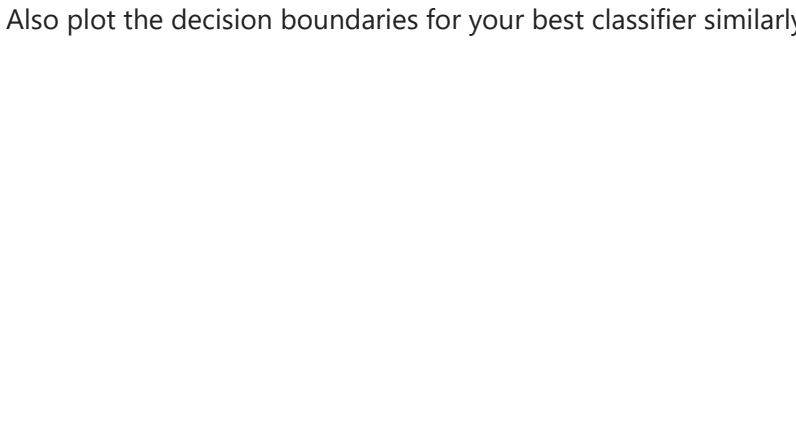
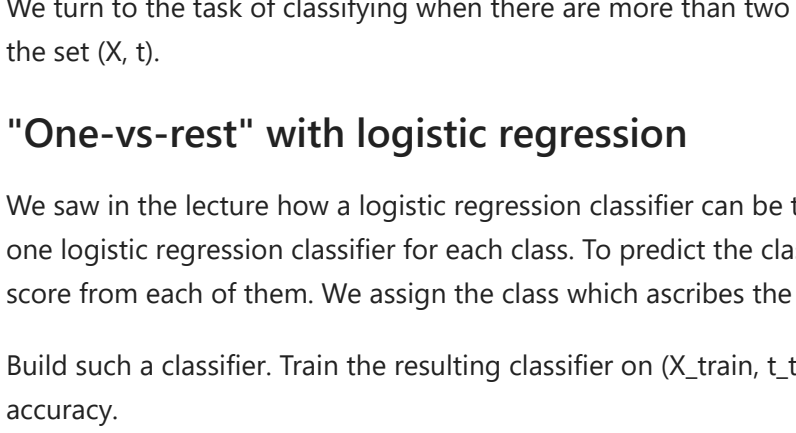
```
def plot_acc_loss():
    """
    Take a cl class, and find the accuracy and loss for each epoch, and plot them.
    """
    acc_train_list = cl.accuracy
    acc_val_list = cl.accuracy_val
    loss_train_list = cl.errors
    loss_val_list = cl.errors_val

    x_axis = np.linspace(1,len(loss_train_list),len(acc_train_list))
    plt.title("Accuracy of train-set and val-set")
    plt.plot(x_axis,acc_train_list)
    plt.plot(x_axis,acc_val_list)
    plt.legend(("Train-set", "Val-set"))
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")

    plt.figure()
    x_axis = np.linspace(1,len(loss_train_list),len(acc_train_list))
    plt.title("Loss of train-set and val-set")
    plt.plot(x_axis,loss_train_list)
    plt.plot(x_axis,loss_val_list)
    plt.legend(("Train-set", "Val-set"))
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
```

```
In [18]: different_loss_diff()
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, eta = 0.01, epochs = 3000, loss_diff = 0.000000001, X_val = X_val, t_val = t2_val)
plot_acc_loss(cl)

"""
I see that the accuracy of logreg is generally a little bit better than linreg,
but the loss to logreg is greater than linreg.
"""
```



Logistic regression

You should now do similarly for a logistic regression classifier. Calculate loss and accuracy for training set and, when provided, also for validation set.

Remember that logistic regression is trained with cross-entropy loss. Hence the loss function is calculated differently than for linear regression.

After a successful training, plot both losses as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure.

Comment on what you see. Do you see any differences between the linear regression classifier and the logistic regression classifier on this dataset?

Starting point: Code from weekly 7

```
In [16]: def cross_entropy(t,y):
    return -np.log(np.exp((1-y)*(1-t)) + (1-y)*(1-t))/len(t)
```

```
In [17]: def logistic(x):
    return 1/(1+np.exp(-x))
```

```
In [18]: class NumpyLogReg(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=10, X_val = X_val, t_val = t2_val):
        """X_train is a NxM matrix, X data points, n features
        t_train are the targets values for training data"""
        (k, m) = X_train.shape
        X_train_with_out_bias = X_train
        X_train = add_bias(X_train)

        if (type(X_val) != type(None)) and (type(t_val) != type(None)):
            X_val_with_out_bias = X_val
            X_val = add_bias(X_val)

        self.weights = weights = np.zeros(m+1)
        self.errors = []
        self.errors_val = []

        self.acc = []
        self.acc_val = []

        for e in range(epochs):
            if (type(X_val) != type(None)) and (type(t_val) != type(None)):
                y_val = self.forward(X_val)
                self.errors_val.append(cross_entropy(t_val, y_val))
                self.acc_val.append(self.accuracy(X_val_with_out_bias, t_val))

            y = self.forward(X_train)
            self.acc.append(self.accuracy(X_train_with_out_bias, t2_train))
            self.errors.append(cross_entropy(t_train, y))

            weights -= eta / k * X_train.T @ (y - t_train)

        def forward(self, X):
            return logistic(X @ self.weights)

        def score(self, x):
            z = add_bias(x)
            score = self.forward(z)
            return score

        def predict(self, x, threshold=0.5):
            z = add_bias(x)
            score = self.forward(z)
            return (score>threshold).astype('int')
```

```
In [51]: cl = NumpyLogReg()
cl.fit(X_train, t2_train, eta = 0.01, epochs = 2000, X_val = X_val, t_val = t2_val)
plot_acc_loss(cl)

"""
I see that the accuracy of logreg is generally a little bit better than linreg,
but the loss to logreg is greater than linreg.
"""
```


Multi-class classifiers

We turn to the task of classifying when there are more than two classes, and the task is to ascribe one class to each input. We will now use the set (X,t) .

"One-vs-rest" with logistic regression

We saw in the lecture how a logistic regression classifier can be turned into a multi-class classifier using the one-vs-rest approach. We train one logistic regression classifier for each class. To predict the class of an item, we run all the binary classifiers and collect the probability score from each of them. We assign the class which ascribes the highest probability.

Build such a classifier. Train the resulting classifier on (X_{train}, t_{train}) , test it on (X_{val}, t_{val}) , tune the hyper-parameters and report the accuracy.

Also plot the decision boundaries for your best classifier similarly to what we plot for the binary case.


```
[20]: class MNINClassifier(X_train, X_test, eta = 0.3, epochs=1000):
    """
    This is a softmax function, that train 5 binary classifier.
    Each classifier will be trained, to each class.

    For example, training of class 0. the classifier will see class 0 as positive class.
    All other classes is negative. In this way we train 5 classifier, one for each class.

    Default of eta is 0.3, I have tried different value of eta, but found eta around 0.2-0.4 gives best accuracy
    eta=0.3 is faster, because it only need to run epochs times when updating weights.
    eta=0.1 is slower, and it with combination of eta = 0.3 give a good accuracy.
    """
    #Transform the data, so all other data that if not the choosen class is 0 (negative)
    #The choosen class set to 1 (positive)

    class0_train = (t_train == 0).astype('int')
    class1_train = (t_train == 1).astype('int')
    class2_train = (t_train == 2).astype('int')
    class3_train = (t_train == 3).astype('int')
    class4_train = (t_train == 4).astype('int')

    #Make logistic regression classifier for each class, and train them
    self.cl0 = NumpyLogReg()
    self.cl1 = NumpyLogReg()
    self.cl2 = NumpyLogReg()
    self.cl3 = NumpyLogReg()
    self.cl4 = NumpyLogReg()

    def predict(self, X_data):
        """
        Args:
            X_data - list/numpy array
            a numpy array

        Using one-vs-rest, for each point in X_data, evaluate the score.
        The score means the probability of the belonging class.
        Then we predicate it to be the class, that give highest score.
        """
        pred_list = []
        for x in X_data:
            #Calculate score to each classifier
            score_list = [self.cl0.score(x), self.cl1.score(x), self.cl2.score(x), self.cl3.score(x), self.cl4.score(x)]

            #Take the highest score
            max_score = max(score_list)
            i = score_list.index(max_score)

            #Predicate x to the class that give highest score
            pred_list.append(i)

        return np.array(pred_list)

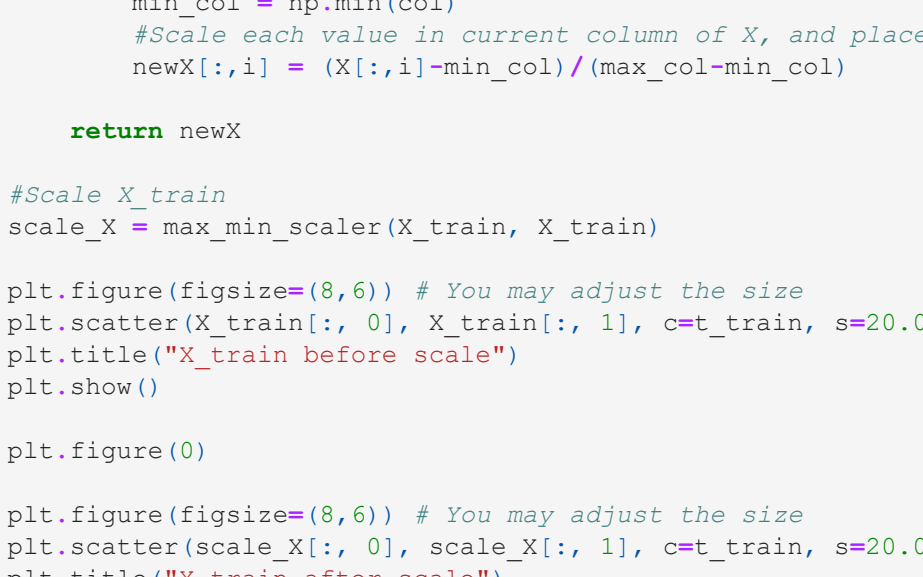
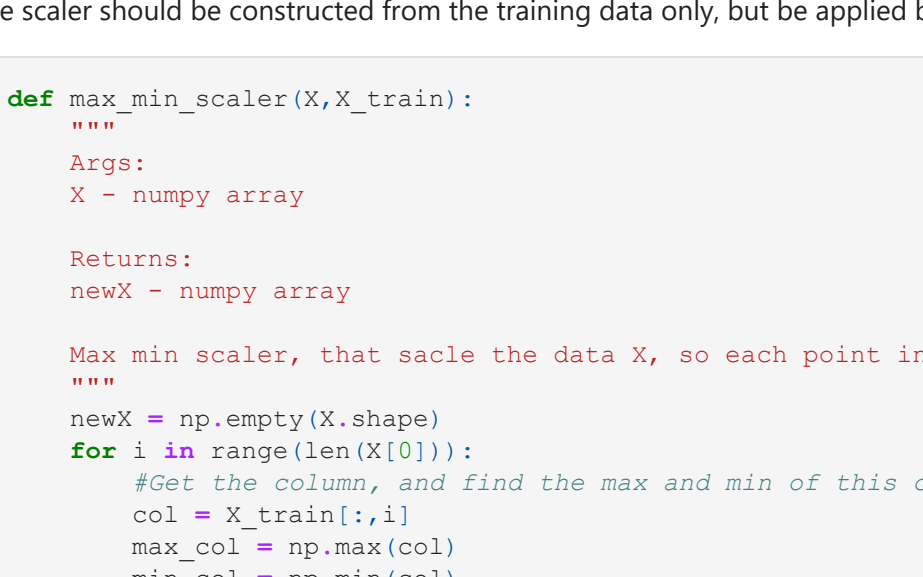
n = 1
acc_list = []
for i in range(10):
    cl = MultiLinRegClass()
    cl.fit(X_train, t_train, epochs=n)
    acc_list.append(cl.accuracy(X_val, t_val))
    n+=200

x_axis = np.linspace(1,2000,len(acc_list))
plt.title("Accuracy of val-set")
plt.plot(x_axis, acc_list)
plt.xlabel("Epoch")
plt.ylabel("Accuracy")

cl = MultiLinRegClass()
cl.fit(X_train, t_train)
acc = cl.accuracy(X_val, t_val)
print("Accuracy of epochs = 1000: ", acc)
plot_decision_regions(X_val, t_val, cl)

Accuracy of epochs = 1000: 0.748
```

Accuracy of val-set



For in4050-students: Multi-nominal logistic regression

The following part is only mandatory for in4050-students. In3050-students are also welcome to make it a try. Everybody has to return for the part 2 on multi-layer neural networks.

In the lecture, we contrasted the one-vs-rest approach with the multinomial logistic regression, also called softmax classifier. Implement also this classifier, tune the parameters, and compare the results to the one-vs-rest classifier. (Don't expect a large difference on a simple task like this.)

Remember that this classifier uses exponentiation followed by softmax in the forward phase. For loss, it uses cross-entropy loss. The loss has a somewhat simpler form than in the binary case. To calculate the gradient is a little more complicated. The actual gradient and update rule is simple, however, as long as you have calculated the forward values correctly.

Part II

Multi-layer neural networks

We will implement the Multi-layer feed forward network (MLP, Marsland sec. 4.2.1), where we use mean squared loss together with logistic activation in both the hidden and the last layer.

Since this part is more complex, we will do it in two rounds. In the first round, we will go stepwise through the algorithm with the dataset (X, t). We will initialize the network and run a first round of training, i.e. one pass through the algorithm at p. 78 in Marsland.

In the second round, we will turn this code into a more general classifier. We can train and test this on (X, t1) and (X, t2), but also on other datasets.

Round 1: One epoch of training

Scaling

First we have to scale our data. Make a standard scaler (normalizer) and scale the data. Remember, not to follow Marsland on this point. The scaler should be constructed from the training data only, but be applied both to training data and later on to validation and test data.

```
In [39]: def max_min_scaler(X,X_train):
    """
    Args:
        X - numpy array
        X_train - numpy array

    Returns:
        maxX - numpy array
        minX - numpy array

    Max min scaler, that scale the data X, so each point in X will be in range 0 to 1.
    """
    newX = np.empty(X.shape)
    for i in range(len(X[0])):
        #Get the column, and find the max and min of this column.
        col_max = np.max(X[:,i])
        col_min = np.min(X[:,i])
        #Scale each value in current column of X, and place it in the new matrix.
        newX[:,i] = (X[:,i]-col_min)/(col_max-col_min)

    return newX

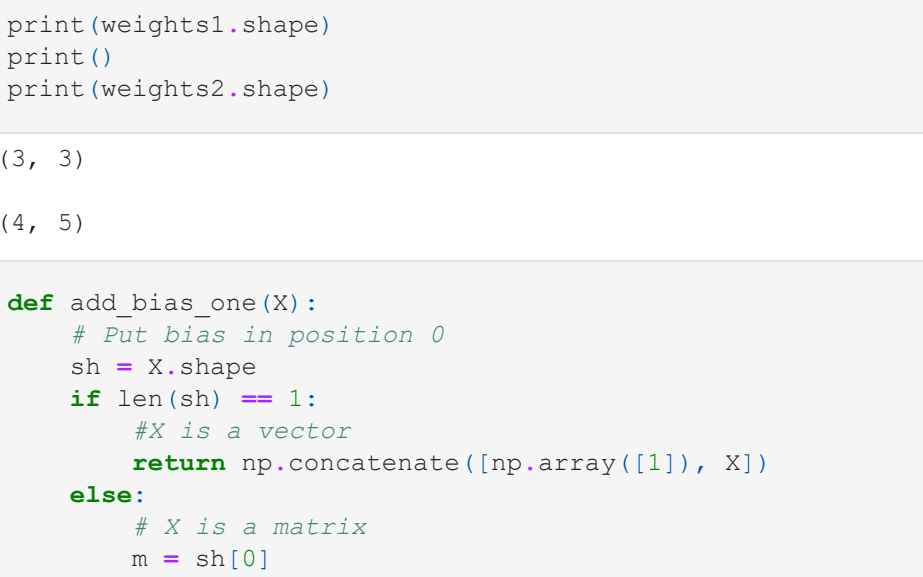
#Scale X_train
scale_X = max_min_scaler(X_train, X_train)

plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(scale_X[:,0], scale_X[:,1], c=t_train, s=20.0)
plt.title("X_train before scale")
plt.show()

plt.figure(0)

plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(scale_X[:,0], scale_X[:,1], c=t_train, s=20.0)
plt.title("X_train after scale")
plt.show()

#Figure size 432x288 with 0 Axes>>
```



Initialization

We will only use one hidden layer. The number of nodes in the hidden layer will be a hyper-parameter provided by the user, let's call it dim_hidden . dim_hidden is called M by Marsland. Initially, we will set it to 3. This is a hyper-parameter where other values may give better results, and the hyper-parameter could be tuned.

Another hyper-parameter set by the user, is the learning rate. We set the initial value to 0.01, but also this may need tuning.

```
In [22]: eta = 0.01 #learning rate
dim_hidden = 3

# We assume that the input X_train (after scaling) is a matrix of dimension P x dim_in, where P is the number of training instances, and dim_in
# is the number of features in the training instances (L in Marsland). Hence we can read dim_in off from X_train.

# The target values have to be converted from simple numbers, 0, 2, ... to "one-hot-encoded" vectors similarly to the multi-class task. After the
# conversion, we can read dim_out from t_train.

In [23]: # convert t_train
print(X_train.shape)
print("P: ", X_train.shape[0])
print("dim_in: ", X_train.shape[1])

dim_in = X_train.shape[1] # Calculate the correct value from the input data
dim_out = 3 # Calculate the correct value from the input data

def one_hot_transform(t):
    one_hot_t = []
    for i in range(len(t)):
        if t[i] == 0:
            one_hot_t.append([1,0,0,0,0])
        elif t[i] == 1:
            one_hot_t.append([0,1,0,0,0])
        elif t[i] == 2:
            one_hot_t.append([0,0,1,0,0])
        elif t[i] == 3:
            one_hot_t.append([0,0,0,1,0])
        else:
            one_hot_t.append([0,0,0,0,1])
    return np.array(one_hot_t)

(1000, 2)
P: 1000
dim_in: 2

We need two sets of weights: weights1 between the input and the hidden layer, and weights2 between the hidden layer and the output.
Make sure that you take the bias terms into consideration and get the correct dimensions. The weight matrices should be initialized to
small random numbers, not to zeros. It is important that they are initialized randomly, both to ensure that different neurons start with
different initial values and to generate different results when you run the classifier. In this introductory part, we have chosen to fix the
random state to make it easier for you to control your calculations. But this should not be part of your final classifier.
```

```
In [27]: # Your code

In [34]: rng = np.random.RandomState(2022)
weights1 = (rng.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.sqrt(dim_in)
weights2 = (rng.rand(dim_hidden+1, dim_out) * 2 - 1)/np.sqrt(dim_hidden)

In [25]: print(weights1.shape)
print()
print(weights2.shape)

(3, 3)

(4, 5)

In [26]: def add_bias_one(X):
    """Put bias in position 0
    # Because y is one-hot encoded
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X])
    else:
        # X is a matrix
        bias = np.ones((m,1)) # Makes a m*1 matrix of 1's
        return np.concatenate([-bias, X], axis = 1)
```

Forwards phase

We will run the first step in the training, and start with the forward phase. Calculate the activations after the hidden layer and after the output layer. We will follow Marsland and use the logistic (sigmoid) activation function in both layers. Inspect whether the results seem reasonable with respect to format and values.

```
In [27]: # Your code

def input_to_hidden_layer(X,V):
    # Add bias one
    Z = X@V
    Y = logistic(Z)
    return Y

def hidden_to_output_layer(A,W):
    # Add bias one
    Z = A@W
    Y = logistic(Z)
    return Y

In [31]: # Your code
hidden_activations = input_to_hidden_layer(scale_X,weights1)
output_activations = hidden_to_output_layer(hidden_activations, weights2)
print(output_activations.shape)

[0.26990052 0.4474561 0.4150512 0.37452743 0.44141406]
```

Control that you are on the right track you may compare your first output value with our result. We have put the bias term -1 in position 0 to both layers. If you have done anything differently from us, you will not get the same numbers. But you still must be on the right track!

```
In [32]: outputs = np.array([0.28969058, 0.44120276, 0.41012141, 0.38135763, 0.44130415])
print(outputs-output_activations[0,:])

***
The difference between given output and my output is small, so for now I just assume it is correct.
The shape of result is 100x5, 1000 instance and 5 classes.
***

[0.01970052 -0.00625334 -0.00492979 0.0068302 -0.00010991]
\nThe difference between given output and my output is small, so for now I just assume it is correct.\n\nThe shape of result is 100x5, 1000 instance and 5 classes.\n"
```

Backwards phase

Calculate the delta terms at the output. We assume, like Marsland, that we use sum of squared errors. (This amounts to the same as using the mean square error).

```
In [28]: # Your code
def delta_output_layer(y,t):
    delta = (y-t)*y*(1-y)
    return delta

Calculate the delta terms in the hidden layer.

In [29]: # Your code
def delta_hidden_layer(y,t,a,w):
    delta = a*(1-a)*delta_output_layer(y,t@w,t)
    return delta

w = weights2
v = weights1
hidden_activations = input_to_hidden_layer(scale_X,v)
output_activations = hidden_to_output_layer(hidden_activations, w)
y = output_activations
t = one_hot_transform(t_train)
a = add_bias(hidden_activations)
X = add_bias(scale_X)
d = delta_hidden_layer(y,t,a,w)

w = weights2 - (eta*a.T@d_delta_output_layer(y,t))
v = weights1 - (eta*a.T@d[:,::-1])

print("w: ", w)
print("v: ", v)
print("delta: ", delta)
print("diff v and old v: ", (v-weights1))

w: [[ 0.1097059 -0.20040025 -0.12731311 -0.03853304 -0.08644044]
 [-0.2291808 -0.384025 -0.53224891 -0.13429652 0.13341971]
 [-0.5534017 -0.0435024 -0.5117888 -0.4761943 -0.0500933 ]
 [-0.6388739 0.32480283 -0.07394431 -0.42684164 -0.76374711]
 [ 0.19392333 0.17555818 -0.56074548]
 [-0.63643285 0.23468113 -0.01222438]
 [-0.63637284 0.16975928 -0.53026643]
 weights1: [[-0.6938717 -0.0013246 -0.54675803]
 [-0.63643285 0.26220593 -0.01840165]
 [ 0.36237224 0.20832872 0.56139063]]
 diff v and old v: [[ 0.
 [-0.0753481 0.0061707]
 [ 0.
 [-0.0376944 -0.011242 ]

Update the weights in both layers. See whether the weights have changed.
```

As an aid, you may compare your new weights with our results. But again, you may have done everything correctly even though you get a different result. For example, there are several ways to introduce the mean squared error. They may give different results after one epoch. But if you run sufficiently many epochs, you will get about the same classifier.

```
In [35]: print("New weights:")
weights = np.array([
[-0.64918987 0.0049323 -0.57494453],
[-0.6388739 0.32480283 -0.08132063],
[ 0.51932333 0.17555818, 0.58007288]])
print(weights)
print("diff v and weights: ", (v-weights))

New weights:
[[ 0.64918987 0.0049323 -0.57494453]
 [-0.6388739 0.32480283 -0.08132063]
 [ 0.51932333 0.17555818 -0.58007288]]
diff v and weights: [[-0.04668183 -0.06615164 0.01419905]
 [-0.0224546 -0.0001217 -0.01934521]
 [ 0.04297991 -0.0057899 -0.02980645]]

Step 2: A Multi-layer neural network classifier
```

Make the classifier

You want to train and test a classifier on (X, t). You could have put some parts of the code in the last step into a loop and run it through some iterations. But instead of copying code for every network we want to train, we will build a general Multi-layer neural network classifier as a class. This class will have some of the same structure as the classifiers we made for linear and logistic regression. The task consists mainly in copying in parts from what you did in step 1 into the template below. Remember to add the self- prefix where needed, and be careful in your use of variable names. And don't fix the random numbers within the classifier.

```
In [38]: def delta_hidden_layer(y,t,a,w):
    delta = a*(1-a)*delta_output_layer(y,t@w,t)
    return delta

def delta_output_layer(y,t):
    delta = (y-t)*y*(1-y)
    return delta

def input_to_hidden_layer(X,V):
    # Add bias one
    Z = X@V
    Y = logistic(Z)
    return Y

def hidden_to_output_layer(A,W):
    # Add bias one
    Z = A@W
    Y = logistic(Z)
    return Y

class MNINClassifier():
    """A multi-layer neural network with one hidden layer"""
    def __init__(self,eta = 0.001, dim_hidden = 6):
        self.X_train_set = None
        #Initialize the hyperparameters
        self.eta = eta
        self.dim_hidden = dim_hidden
        self.dim_in = 2
        self.dim_out = 5

        weights1 = (np.random.rand(self.dim_in + 1, self.dim_hidden) * 2 - 1)/np.sqrt(self.dim_in)
        weights2 = (np.random.rand(self.dim_hidden+1, self.dim_out) * 2 - 1)/np.sqrt(self.dim_hidden)
        self.v = weights1
        self.w = weights2

        # Should you put additional code here?

    def fit(self, X_train, t_train, epochs = 100, X_val = None, t_val = None):
        """
        Fit function for the classifier, and run for given epochs.
        This will first scale X, and add bias to X, and one-hot transform t.
        Then perform forward and backward path, and the end change weights in both layer.
        """
        self.X_train_set = X_train

        # Initialization
        X_scale = max_min_scaler(X_train,X_train)
        X = add_bias_one(X_scale)
        t = one_hot_transform(t_train)

        #List for accuracy and loss each epoch
        self.acc_list = []
        self.errors = []

        for e in range(epochs):
            # Here is the forward path
            if (type(X_val) != type(None)) and (type(t_val) != type(None)):
                # Fit function for the classifier, and run for given epochs.
                y_val = self.predict(X_val)
                self.errors_val.append((t_val, y_val))

            self.acc.append(self.accuracy(X_train,t_train))
            y,a = self.forward(X_scale)

            #Calculation of loss, will be a little bit different then before
            #Because y is one-hot encoded
            #Here for we predict and get the value of y with out encode and calculate loss.
            self.errors.append((t_train, self.predict(X_train)))

            # Here is the backward path
            d_out = delta_output_layer(y,t)
            d_hidden = delta_hidden_layer(y,t,a,self.w)

            #Update weights
            self.v = self.v - (self.eta*a.T@d_out)
            self.w = self.w - (self.eta*a.T@d_hidden[:,::-1])

        def forward(self, X):
            """Perform one forward step.
            Return a pair consisting of the outputs of the hidden layer
            and the outputs on the final layer"""
            hidden_activations = input_to_hidden_layer(X, self.v)
            output_activations = hidden_to_output_layer(hidden_activations, self.w)
            y = output_activations
            a = add_bias_one(hidden_activations)
            return y,a

        def predict(self, X):
            """
            This will decode the one-hot version of y,
            that is the result by sending X into our multi-layer classifier.
            The each row in y contains probability of the value is class 0,1,2,3 or 4.
            We predict after the highest value in the row, and transform it back to a integer.
            """
            y,a = self.forward(X)

            list = []
            for i in range(len(y)):
                if pred_t[i] == t_test[i]:
                    correct += 1

            return correct/len(t_test)

        def accuracy(self, X_test, t_test):
            """Calculate the accuracy of the classifier for the pair (X_test, t_test)
            Return the accuracy"""
            X_test = max_min_scaler(X_test, self.X_train_set)
            pred_t = self.predict(X_test)
            correct = 0

            for i in range(len(pred_t)):
                if pred_t[i] == t_test[i]:
                    correct += 1

            return correct/len(t_test)

acc_list = []
for i in range(10):
    cl = MNINClassifier(eta = 0.01, dim_hidden = 10)
    cl.fit(X_train, t2_train, epochs = 459)
    acc = cl.accuracy(X_val,t2_val)
    acc_list.append(cl)
    acc_list.append(acc)

#Find the highest accuracy and the classifier
highest_acc = max(acc_list)
i = acc_list.index(highest_acc)

print("Accuracy(Highest): ", highest_acc)
X_tmp = max_min_scaler(X_val,X_train)
plot_decision_regions(X_tmp, t2_val, cl_list[i])

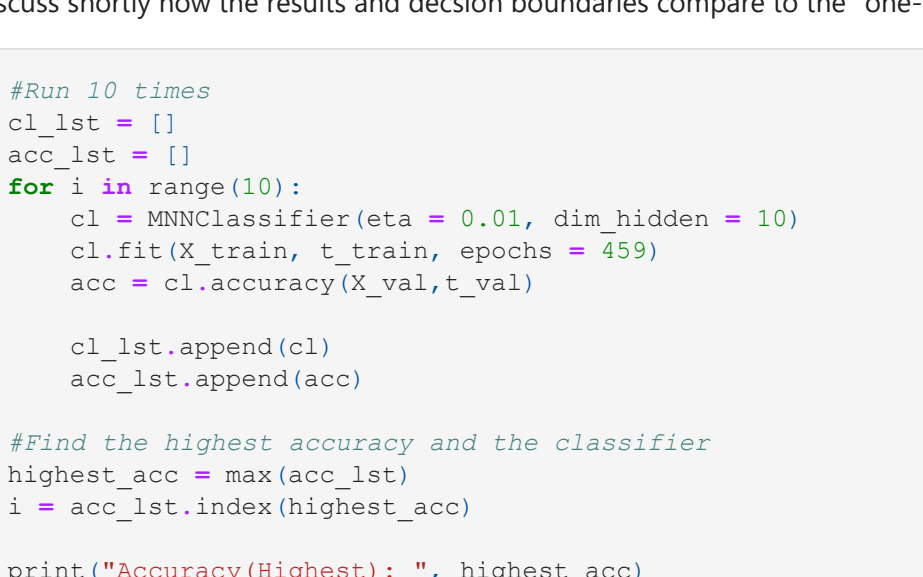
#Mean value of accuracies to all 10 runs
print("Mean of accuracies: ",sum(np.array(acc_list))/len(acc_list))

#Standard deviation
std = np.std(np.array(acc_list))
print("Standard deviation: ", std)

***
From the results, we see that the accuracy of MNINClassifier is worse than one-vs-rest.
But the training of MNINClassifier is faster, because it only need to run epochs times when updating weights.
But the one-vs-rest need to run 5 times of epochs.
***

C:\Users\aler\AppData\Local\Programs\Python\Python37\Lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
overflow encountered in exp

Accuracy(Highest): 0.73
```



Mean of accuracies: 0.6258000000000001
Standard deviation: 0.0954549177487667
From the results, we see that the accuracy of MNINClassifier is worse than one-vs-rest. But the training of MNIN is faster, because it only need to run epochs times when updating weights. But the one-vs-rest need to run 5 times of epochs.

Binary class

Let us see whether a multilayer neural network can learn a non-linear classifier. Train a classifier on (X_train, t2_train) and test it on (X_val, t2_val). Tune the hyper-parameters for the best result. Run ten times with the best setting and report mean and standard deviation. Plot the decision boundaries.

```
In [42]: #Run 10 times
acc_list = []
for i in range(10):
    cl = MNINClassifier(eta = 0.01, dim_hidden = 10)
    cl.fit(X_train, t2_train, epochs = 459)
    acc = cl.accuracy(X_val,t2_val)
    acc_list.append(cl)
    acc_list.append(acc)

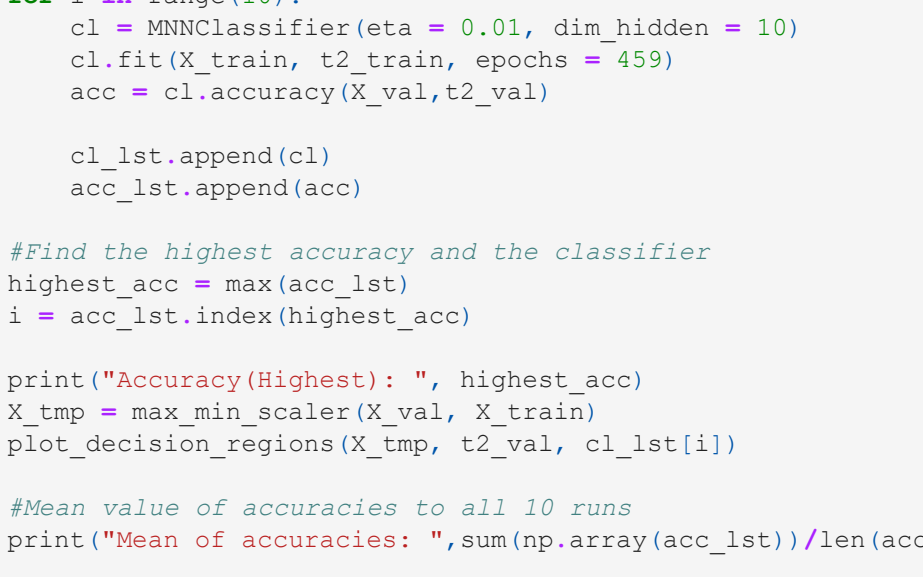
#Find the highest accuracy and the classifier
highest_acc = max(acc_list)
i = acc_list.index(highest_acc)

print("Accuracy(Highest): ", highest_acc)
X_tmp = max_min_scaler(X_val,X_train)
plot_decision_regions(X_tmp, t2_val, cl_list[i])

#Mean value of accuracies to all 10 runs
print("Mean of accuracies: ",sum(np.array(acc_list))/len(acc_list))

#Standard deviation
std = np.std(np.array(acc_list))
print("Standard deviation: ", std)

Accuracy(Highest): 0.672
```



Mean of accuracies: 0.6258000000000001
Standard deviation: 0.0954549177487667
From the results, we see that the accuracy of MNINClassifier is worse than one-vs-rest. But the training of MNIN is faster, because it only need to run epochs times when updating weights. But the one-vs-rest need to run 5 times of epochs.

For in4050-students: Early stopping

The following part is only mandatory for in4050-students. In3050-students are also welcome to make it a try. Everybody has to return for the part 2 on multi-layer neural networks.

There is a danger of overfitting if we run too many epochs of training. One way to control that is to use early stopping. We can use (X_val, t2_val) as validation set when training on (X_train, t_train).

Let $e=50$ or $e=70$ (You may try both or choose some other number) After e number of epochs, calculate the loss for both the training set (X_train, t_train) and the validation set (X_val, t_val), and store them.

Train a classifier for many epochs. Plot the losses for both the training set and the validation set in the same figure and see whether you get the same effect as in figure 4.11 in Marsland.

Modify the code so that the training stops if the loss on the validation set is not reduced by more than ϵ after e many epochs, where ϵ is a threshold you provide as a parameter.

Run the classifier with various values for ϵ and report the accuracy and the number of epochs ran.

Part III: Final testing

We can now perform a final testing on the held-out test set.

Binary task (X, t2)

Consider the linear regression classifier, the logistic regression classifier and the multi-layer network with the best settings you found. Train each of them on the training set and evaluate on the held-out test set, but also on the validation set and the training set. Report in a 3 by 3 table.

Comment on what you see. How do the three different algorithms compare? Also, compare the result between the different data sets. In cases like these, one might expect slightly inferior results on the held-out test data compared to the validation data. Is this the case?

Also report precision and recall for class 1.

Multi-class task (X, t)

For In3050-students compare the one-vs-rest classifier to the multi-layer perceptron. Evaluate on test, validation and training set as above. In4050-students should also include results from the multi-nomial logistic regression.

Comment on the results.

```
In [43]: cl_mnn = MNINClassifier(eta = 0.01, dim_hidden = 10)
cl_mnn.fit(X_train, t2_train, epochs = 459)
acc_mnn_test = cl_mnn.accuracy(X_test,t2_test)
acc_mnn_val = cl_mnn.accuracy(X_val,t2_val)
acc_mnn_train = cl_mnn.accuracy(X_train,t2_train)

cl_log_reg = NumpyLogReg()
cl_log_reg.fit(X_train, t2_train, eta = 0.01, epochs = 2000)
acc_log_reg_test = cl_log_reg.accuracy(X_test, t2_test)
acc_log_reg_val = cl_log_reg.accuracy(X_val, t2_val)
acc_log_reg_train = cl_log_reg.accuracy(X_train, t2_train)

cl_lin_reg = NumpyLinRegClass()
cl_lin_reg.fit(X_train, t2_train, eta = 0.01, epochs = 3000, loss_diff = 0.0000000001)
acc_lin_reg_test = cl_lin_reg.accuracy(X_test, t2_test)
acc_lin_reg_val = cl_lin_reg.accuracy(X_val, t2_val)
acc_lin_reg_train = cl_lin_reg.accuracy(X_train, t2_train)

print(" MNIN | LOG | LIN")
print("(Test | ",acc_mnn_test," | ", acc_log_reg_test," | ", acc_lin_reg_test)
print("(Val | ",acc_mnn_val," | ", acc_log_reg_val," | ", acc_lin_reg_val)
print("(Train | ",acc_mnn_train," | ", acc_log_reg_train," | ", acc_lin_reg_train)

MNIN | LOG | LIN
Test | 0.724 | 0.724 | 0.724
Val | 0.664 | 0.662 | 0.662
Train | 0.708 | 0.708 | 0.708

In [46]: """
Here we report precision and recall for cl_mnn
"""
y_pred_test = cl_mnn.predict(X_test)
tp = 0
fn = 0
fp = 0
tn = 0

for i in range(len(y_pred_test)):
    if y_pred_test[i] == 1:
        tp += 1
    else:
        fp += 1
else:
    if y_pred_test[i] == t2_test[i]:
        tn += 1
    else:
        fn += 1

print("Precision: ", tp/(tp+fn))
print("Recall: ", (tp/(tp+fp)))

Precision: 0.57
Recall: 0.7125
```



```
In [47]: cl_mnn = MNNCClassifier(eta = 0.01, dim_hidden = 10)
cl_mnn.fit(X_train, t_train, epochs = 450)
acc_mnn_test = cl_mnn.accuracy(X_test, t_test)
acc_mnn_val = cl_mnn.accuracy(X_val, t_val)
acc_mnn_train = cl_mnn.accuracy(X_train, t_train)

cl_multi = MulticlassClassifier()
cl_multi.fit(X_train, t_train)
acc_multi_test = cl_multi.accuracy(X_test, t_test)
acc_multi_val = cl_multi.accuracy(X_val, t_val)
acc_multi_train = cl_multi.accuracy(X_train, t_train)

print("      MNN      | Multi |")
print("Test | ", acc_mnn_test, "|", acc_multi_test)
print("Val  | ", acc_mnn_val, "|", acc_multi_val)
print("Train| ", acc_mnn_train, "|", acc_multi_train)

c:\users\elcr\appdata\local\programs\python\python37-32\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning:
sing: overflow encountered in exp

      MNN      | Multi |
Test | 0.694 | 0.768
Val  | 0.676 | 0.748
Train| 0.674 | 0.746
```

```
In [ ]: """
In binary we see that the test result of MNN is better than other classifier.
And the test result to MNN is alot better than val and train.

But in multiple class case, the one-vs-test gives a better result.
The reason might be overfitting and init weights choose.
"""
```