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-ifi-mandatory.html , in particular the paragraph on cooperation. This is an individual assignment. You are not allowed to deliver together or copy/share source-code/answers with others. By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

Delivery

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

Your submission should be delivered in Devilry. You may redeliver in Devilry 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 hours (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 IFI 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 [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import sklearn #for datasets
```

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---and not a set of natural occuring 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 may 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* till 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(2022)`.

In [2]:
```python
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 [3]:
```python
indices = np.arange(X.shape[0])
rng = np.random.RandomState(2022)
rng.shuffle(indices)
indices[:10]
```

Out[3]:
```
array([1018, 1295,  643, 1842, 1669,   86,  164, 1653, 1174,  747])
```

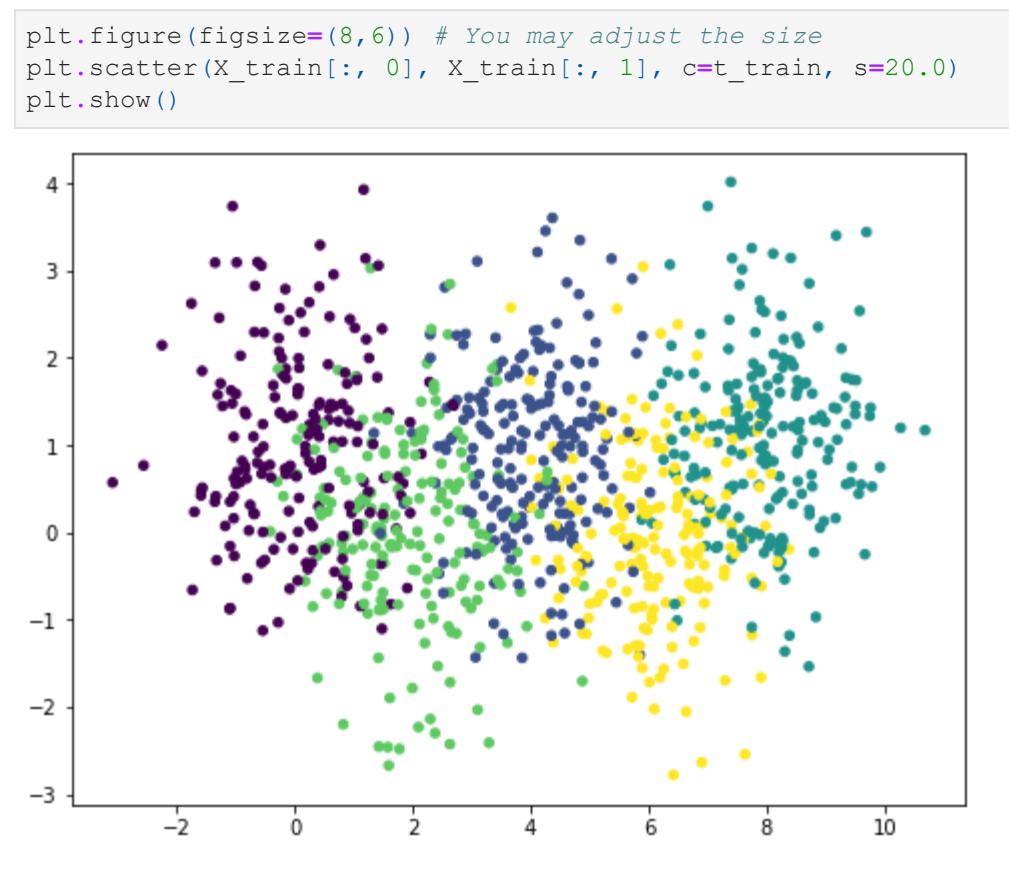
In [4]:
```python
X_train = X[indices[:1000],:]
X_val = X[indices[1000:1500],:]
X_test = X[indices[1500:],:]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]
```

Next, we will make a second dataset by merging the two smaller classes in (X,t) and call the new set (X, t2). This will be a binary set.

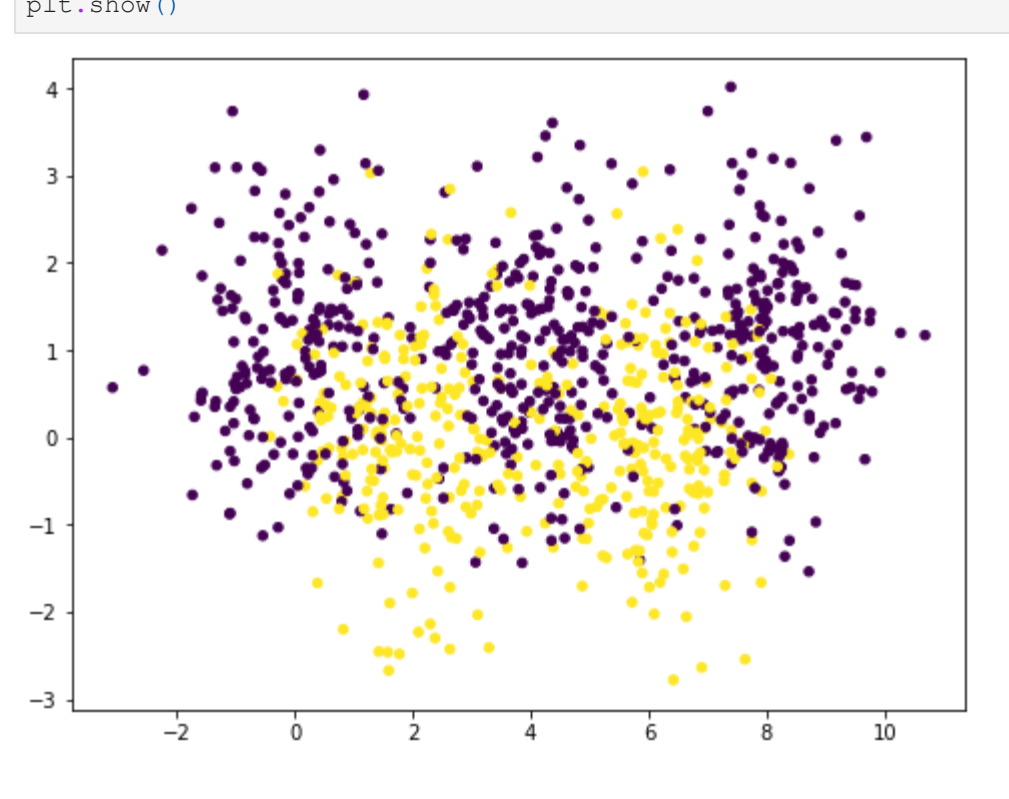
In [5]:
```python
t2_train = t_train >= 3
t2_train = t2_train.astype('int')
t2_val = (t_val >= 3).astype('int')
t2_test = (t_test >= 3).astype('int')
```

We can plot the two traing sets.

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



In [7]:
```python
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, t2) 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.

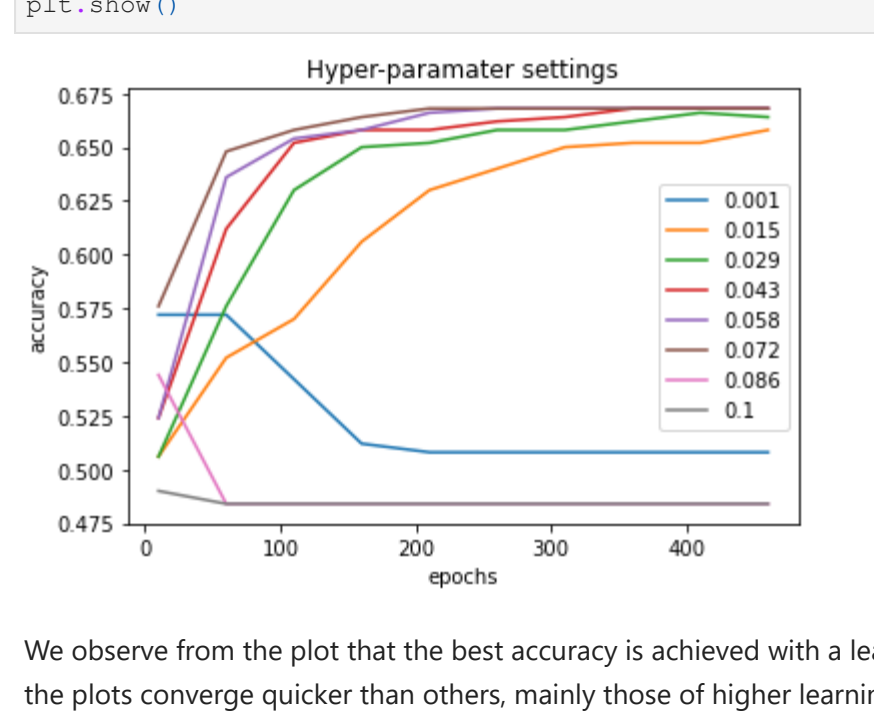
In [8]:
```python
def add_bias(X):
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X])
    else:
        # X is a matrix
        m = sh[0]
        bias = np.ones((m,1)) # Makes a m*1 matrix of 1-s
        return np.concatenate([bias, X], axis  = 1)
```

In [72]:
```python
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)

    def recall(self, X_test, y_test):
        pred = self.predict(X_test)
        true_p = 0
        false_n = 0
        for j in range(pred.shape[0]):
            if pred[j]==1:
                if pred[j]==y_test[j]:
                    true_p += 1
            else:
                if pred[j]!=y_test[j]:
                    false_n += 1

        return round(true_p/(true_p+false_n), 3)

    def precision(self, X_test, y_test):
        pred = self.predict(X_test)
        true_p = 0
        false_p = 0
        for j in range(pred.shape[0]):
            if pred[j]==1:
                if pred[j]==y_test[j]:
                    true_p += 1
                else:
                    false_p += 1

        return round(true_p/(true_p+false_p), 3)
```

In [73]:
```python
class NumpyLinRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, eta, epochs, loss_diff, X_val=None, t_val=None):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        (k, m) = X_train.shape
        X_train = add_bias(X_train)
        if X_val is not None:
            X_val = add_bias(X_val)

        self.weights = weights = np.zeros(m+1)
        self.loss_train = []
        self.loss_val = []
        self.acc_val = []
        self.acc_train = []
        self.epochs_ran = 0

        for e in range(epochs):
            step = X_train.T@(X_train @ weights - t_train)
            current_loss_diff = abs(np.average(step))
            self.loss_train.append(current_loss_diff)

            if abs(current_loss_diff) <= loss_diff:
                break

            weights -= eta / k * step
            self.epochs_ran += 1

            if X_val is not None and t_val is not None:
                self.acc_val.append(self.accuracy(X_val[:,1:], t_val))
                self.acc_train.append(self.accuracy(X_train[:,1:], t_train))
                self.loss_val.append(abs(np.average(X_val.T@(X_val@weights-t_val))))


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

    def getLossTrain(self):
        return self.loss_train

    def getLossVal(self):
        return self.loss_val

    def getEpochs(self):
        return self.epochs_ran

    def getAccVal(self):
        return self.acc_val

    def getAccTrain(self):
        return self.acc_train
```

We can train and test a first classifier.

In [11]:
```python
cl = NumpyLinRegClass()


for eta in np.linspace(0.001, 0.1, 8):
    accs = []
    for epoch in range(10, 500, 50):
        cl.fit(X_train, t2_train, eta, epoch, loss_diff=0.2)
        accs.append(cl.accuracy(X_val, t2_val))
    plt.plot(range(10, 500, 50), accs, label=str(round(eta, 3)))
    plt.legend()

plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("Hyper-paramater settings")
plt.show()
```



We observe from the plot that the best accuracy is achieved with a learning rate of eta=0.072 and an epoch above the 200 mark. Some of the plots converge quicker than others, mainly those of higher learning rates (eta at ca 0.086) as these will jump over the optimal solution and converge to local optima during gradient descent. On the opposite end we have eta=0.001, which updates the weights very slowly and would need a lot more epochs to become optimal.

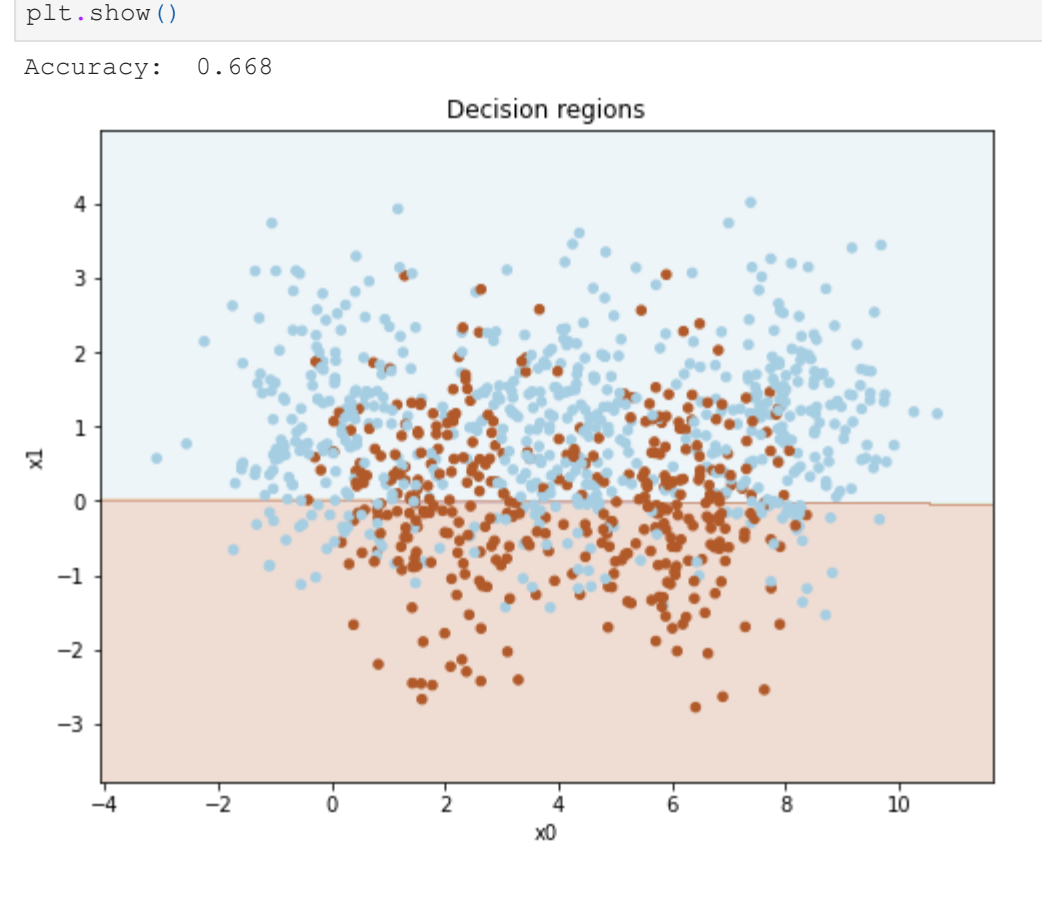
The result is far from impressive. Experiment with various settings for the hyper-parameters, eta and epochs. Report how the accuracy vary with the hyper-parameter settings. When you are satisfied with the result, you may plot the decision boundaries, as below.

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 substanial modifications.

In [12]:
```python
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 [13]:
```python
cl.fit(X_train, t2_train, eta=0.072, epochs=300, loss_diff=0.2)
print("Accuracy: ", cl.accuracy(X_val, t2_val))
plot_decision_regions(X_train, t2_train, cl)
plt.show()
```

```
Accuracy:  0.668
```

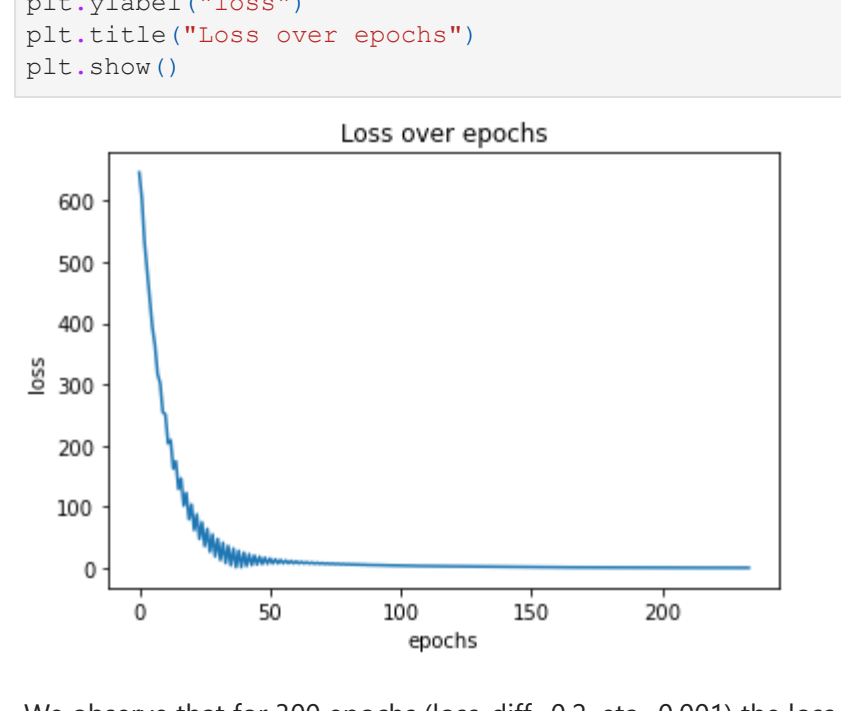


Loss

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 [14]:
```python
loss = cl.getLossTrain()
plt.plot(range(len(loss)), loss)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("Loss over epochs")
plt.show()
```



We observe that for 300 epochs (loss_diff=0.2, eta=0.001) the loss will be greatly reduced during the first 50 epochs. After this, the loss will slowly approach 0.2, where it stops due to to early stop function (at epoch 234). This is expected behaviour.

Control training

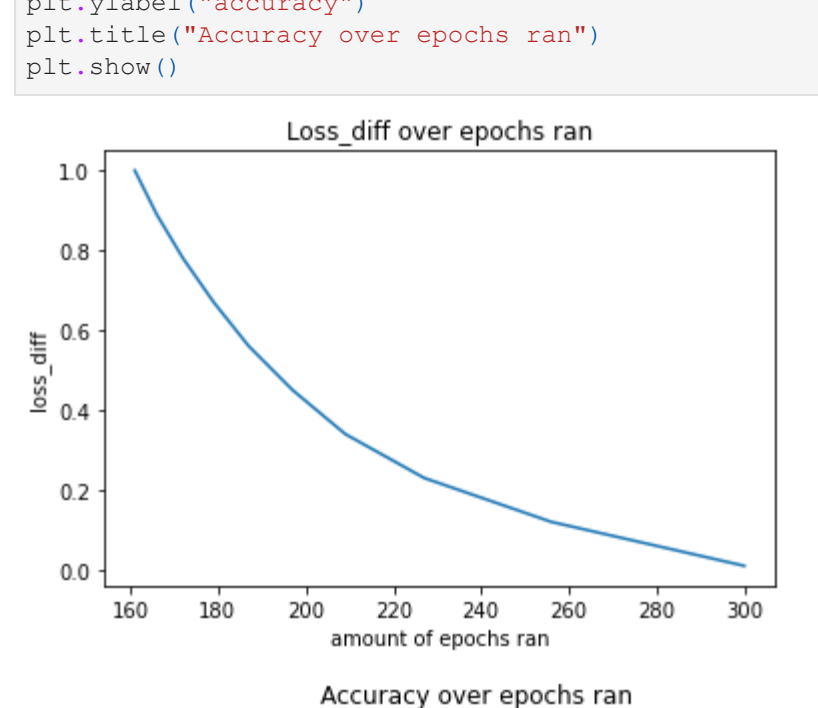
The training runs for a number of epochs. We cannot know beforehand for how many epochs it is reasonable to run the training. One possibility is to run the training until the learning does not improve much. Extend the fit-method with a keyword argument, `loss_diff`, and stop training when the loss has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

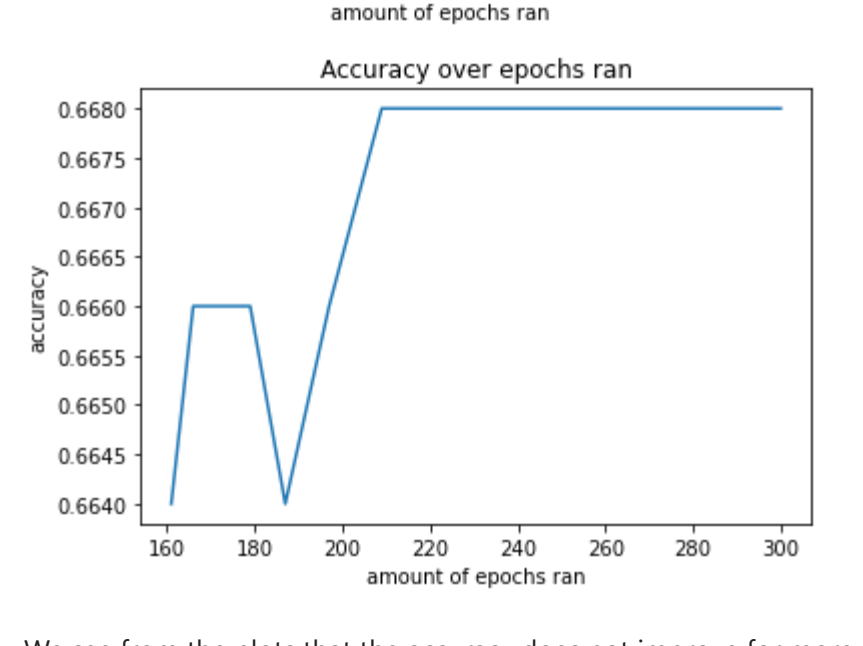
In addition, extend the fit-method with optional arguments for a validation set (X_val, t_val). If a validation set is included in the call to fit, calculate the loss for the validation set, and the accuracy for both the training set and the validation set for each epoch.

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

After a succesful training, plot both training loss snd vslidation 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.

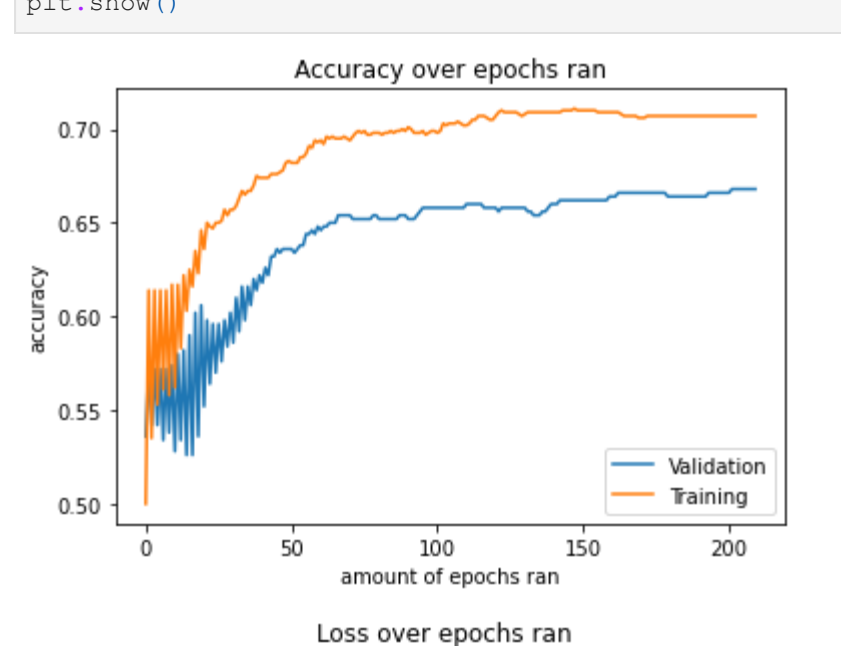
In [15]:
```python
cl = NumpyLinRegClass()

epochs_ran = []
accs = []
loss_ds = np.linspace(0.01, 1, 10)
for loss_d in loss_ds:
    cl.fit(X_train, t2_train, eta=0.072, epochs=300, loss_diff=loss_d)
    epochs_ran.append(cl.getEpochs())
    accs.append(cl.accuracy(X_val, t2_val))

plt.plot(epochs_ran, loss_ds)
plt.xlabel("amount of epochs ran")
plt.ylabel("loss_diff")
plt.title("Loss_diff over epochs ran")
plt.show()

plt.plot(epochs_ran, accs)
plt.xlabel("amount of epochs ran")
plt.ylabel("accuracy")
plt.title("Accuracy over epochs ran")
plt.show()
```

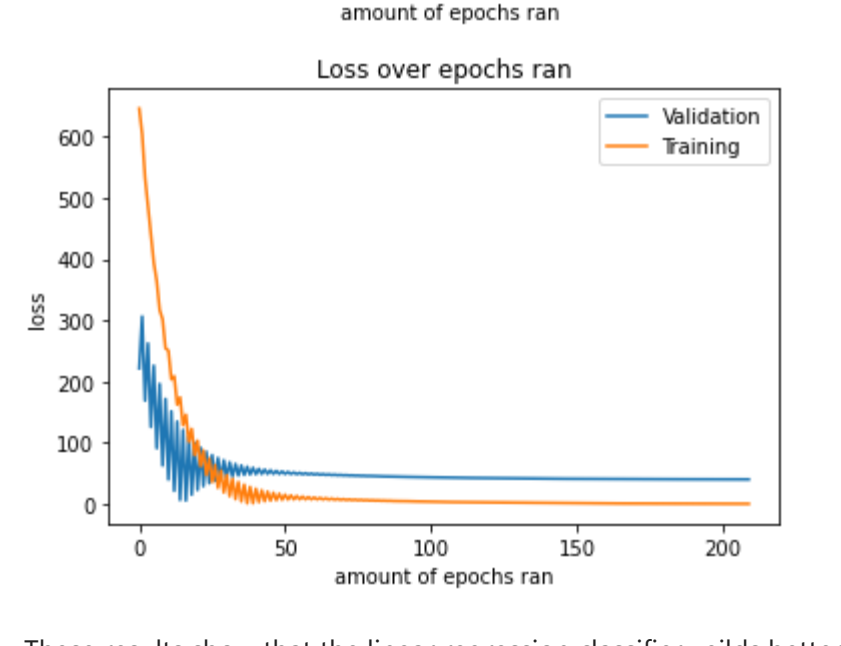




We see from the plots that the accuracy does not improve for more than 210 epochs ran.

In [16]:
```python
cl.fit(X_train, t2_train, eta=0.072, epochs=210, loss_diff=0.2, X_val=X_val, t_val=t2_val)
plt.plot(range(len(cl.getAccVal())), cl.getAccVal(), label="Validation")
plt.title("Accuracy over epochs ran")
plt.xlabel("amount of epochs ran")
plt.ylabel("accuracy")
plt.plot(range(len(cl.getAccTrain())), cl.getAccTrain(), label="Training")
plt.legend()
plt.show()

plt.plot(range(len(cl.getLossVal())), cl.getLossVal(), label="Validation")
plt.title("Loss over epochs ran")
plt.xlabel("amount of epochs ran")
plt.ylabel("loss")
plt.plot(range(len(cl.getLossTrain())), cl.getLossTrain(), label="Training")
plt.legend()
plt.show()
```





These results show that the linear regression classifier yeilds better results for the training dataset than for the validation dataset. This is explained by the validation set having a slightly different spread of the different class, which the algorithm is untrained for. The linear regression classifier can achieve a greater accuracy on the training set because, well, it was trained on this dataset, creating optimal decision regions for this dataset. It's not that all new datasets would give inferior results, for example if a test dataset had all points of the negative class on one side of the decision boundary and all points of the positive class on the other, the linear regression classifier would achieve 100% accuracy, which is greater than the 0.71 achieved by the training set. The validation data however only yields a maximum accuracy of 0.65.

The loss is greater for the validation data (greater value is worse), but becomes stabile at the same rate as the training data.

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 succesful 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.

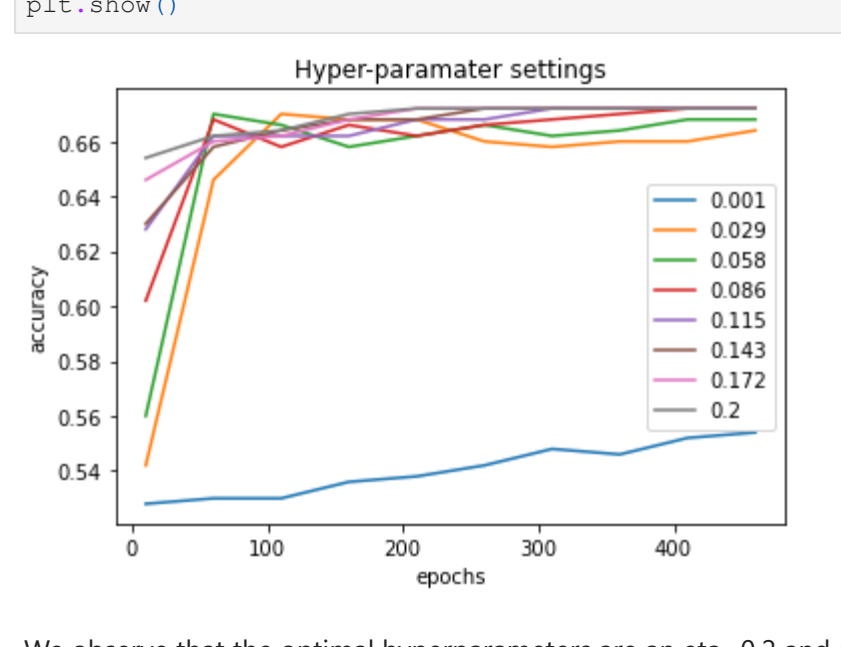
Comment on what you see. Do you see any differences between the linear regression classifier and the logistic regression classifier on this dataset?

Starting point: Code from weekly 7

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

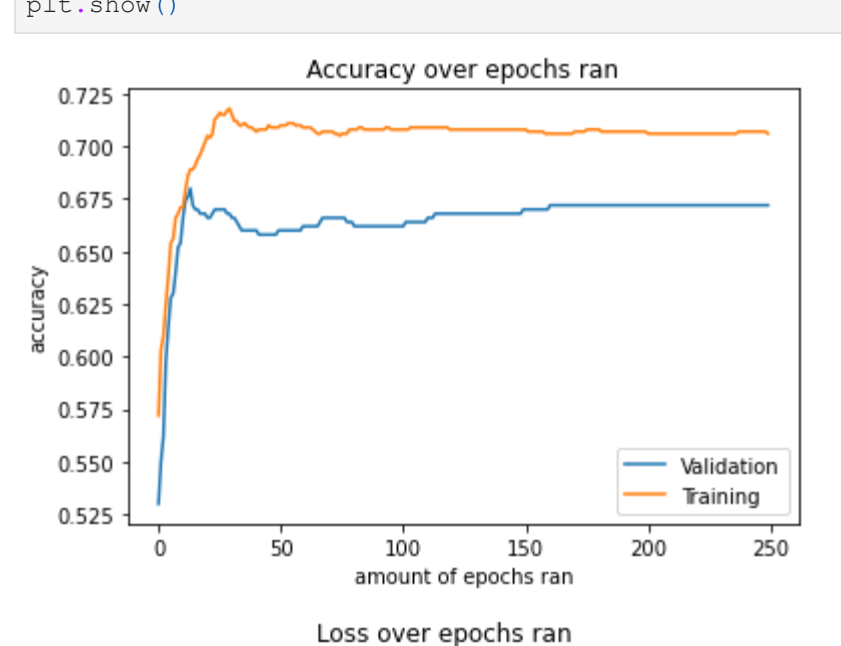
In [74]:
```python
class NumpyLogReg(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=10, X_val=None, t_val=None):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        (k, m) = X_train.shape
        X_train = add_bias(X_train)
        if X_val is not None and t_val is not None:
            X_val = add_bias(X_val)

        self.weights = weights = np.zeros(m+1)
        self.loss_train = []
        self.loss_val = []
        self.acc_val = []
        self.acc_train = []
        self.epochs_ran = 0

        for e in range(epochs):

            step = X_train.T @ (self.forward(X_train) - t_train)
            weights -= eta / k * step

            current_loss_diff = np.average(step)
            self.loss_train.append(abs(current_loss_diff))

            if X_val is not None and t_val is not None:
                self.acc_val.append(self.accuracy(X_val[:,1:], t_val))
                self.acc_train.append(self.accuracy(X_train[:,1:], t_train))
                self.loss_val.append(abs(np.average(X_val.T@(self.forward(X_val)-t_val))))

    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')

    def getLossTrain(self):
        return self.loss_train

    def getLossVal(self):
        return self.loss_val

    def getEpochs(self):
        return self.epochs_ran

    def getAccVal(self):
        return self.acc_val

    def getAccTrain(self):
        return self.acc_train
```

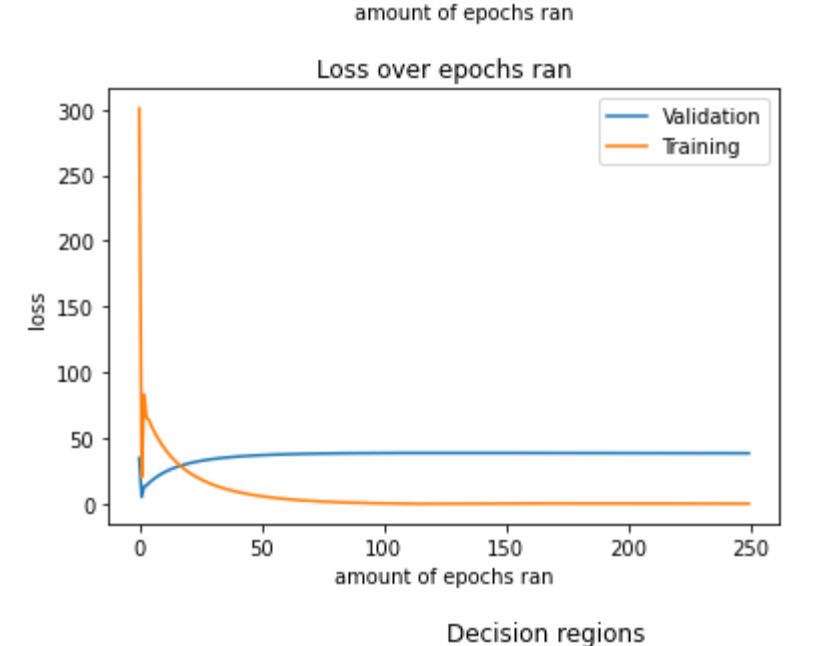
In [19]:
```python
cl = NumpyLogReg()

# finner beste hyperparametre for logreg
for eta in np.linspace(0.001, 0.2, 8):
    accs = []
    for epoch in range(10, 500, 50):
        cl.fit(X_train, t2_train, eta, epoch)
        accs.append(cl.accuracy(X_val, t2_val))
    plt.plot(range(10, 500, 50), accs, label=str(round(eta, 3)))
    plt.legend()

plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("Hyper-paramater settings")
plt.show()
```

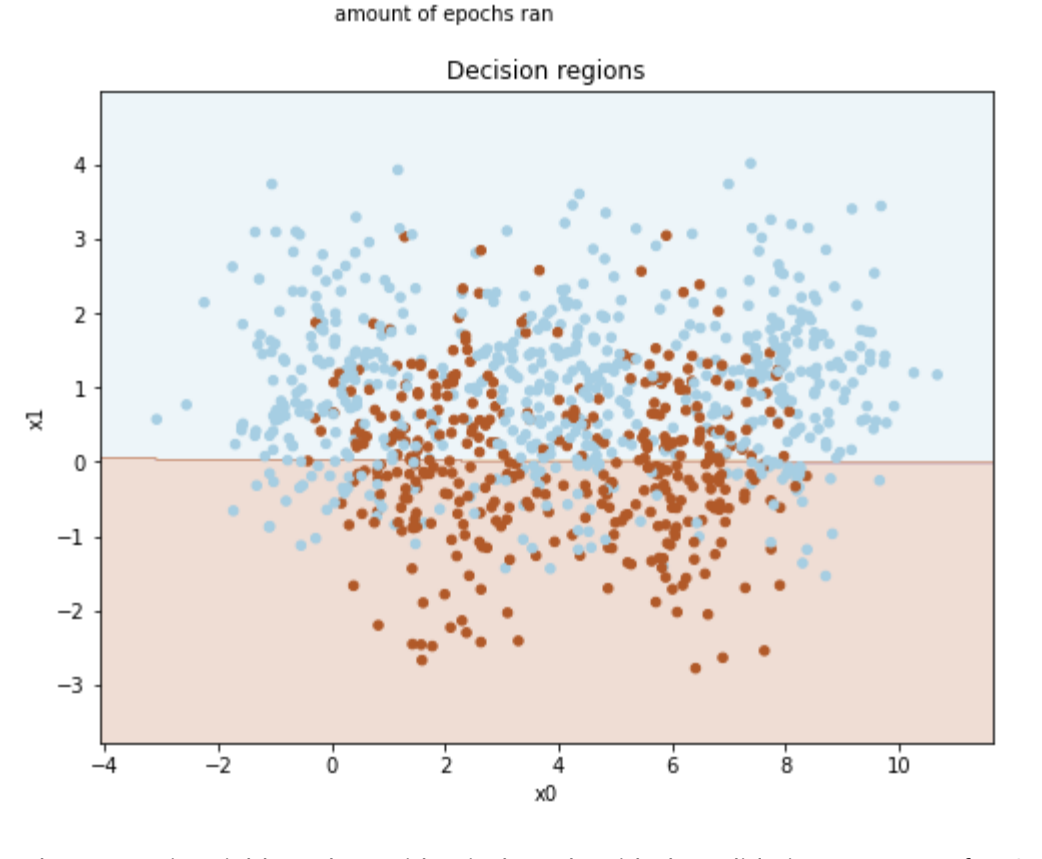


We observe that the optimal hyperparameters are an eta=0.2 and about 250 epochs

In [20]:
```python
cl.fit(X_train, t2_train, eta=0.2, epochs=250, X_val=X_val, t_val=t2_val)

plt.plot(range(len(cl.getAccVal())), cl.getAccVal(), label="Validation")
plt.title("Accuracy over epochs ran")
plt.xlabel("amount of epochs ran")
plt.ylabel("accuracy")
plt.plot(range(len(cl.getAccTrain())), cl.getAccTrain(), label="Training")
plt.legend()
plt.show()

plt.plot(range(len(cl.getLossVal())), cl.getLossVal(), label="Validation")
plt.title("Loss over epochs ran")
plt.xlabel("amount of epochs ran")
plt.ylabel("loss")
plt.plot(range(len(cl.getLossTrain())), cl.getLossTrain(), label="Training")
plt.legend()
plt.show()

plot_decision_regions(X_train, t2_train, cl)
plt.show()
```







The accuracies yield an almost identical result, with the validation accuracy after 250 epochs being 0.672 for the logistic regression classifier and 0.668 for the linear regression classifier. For the training data it's even closer, being 0.706 for the logreg and 0.707 for linreg. There are seemingly random, small peaks at the beginning epochs in accuracy. I don't believe this to be overfitting, rather points where the decision boundaries happen to favour the dataset.

The loss functions similarly to the linear regression classifier even though we are now using cross entropy loss instead of mean squared error. Again, the loss seems to dip very early on, and eventually stabilizes at a point near zero for the training set. The validation set dips, then has an increasing loss. I did not expect this result, and the loss function seems to work perfectly fine for other eta values than 0.2. I suppose the validation data just fits well with the beginning weights.

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

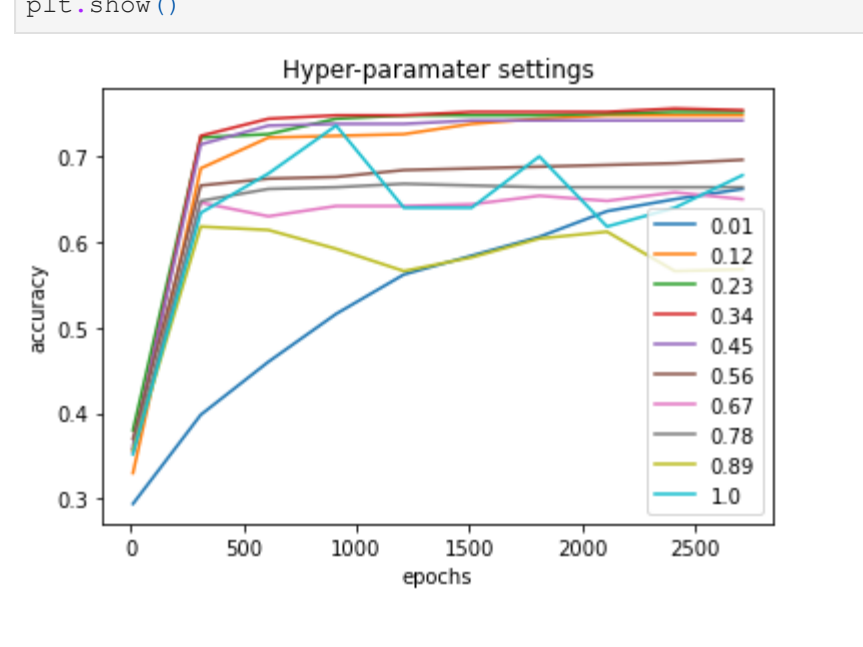
We saw in the lecture how a logistic regression classifier can be turned into a multi-class classifier using the one-vs-rest approach. We train one logistic regression classifier for each class. To predict the class of an item, we run all the binary classifiers and collect the probability score from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

In [21]:
```python
class OnevsRest(NumpyClassifier):

    def fit(self, X_train, t_train, eta, epochs):
        # trener for hver klasse
        self.classifiers = [NumpyLogReg() for i in range(5)]
        for cl in self.classifiers:
            t = (t_train==self.classifiers.index(cl)).astype("int")
            cl.fit(X_train, t, eta, epochs)


    def predict(self, x):
        # faktiske prediksjoner
        preds = []
        for e in x:
            # finner probability score
            probs = []
            for cl in self.classifiers:
                z = add_bias(e)
                probs.append(np.average(cl.forward(z)))
            preds.append(probs.index(max(probs)))
        preds = np.array(preds)
        return preds
```

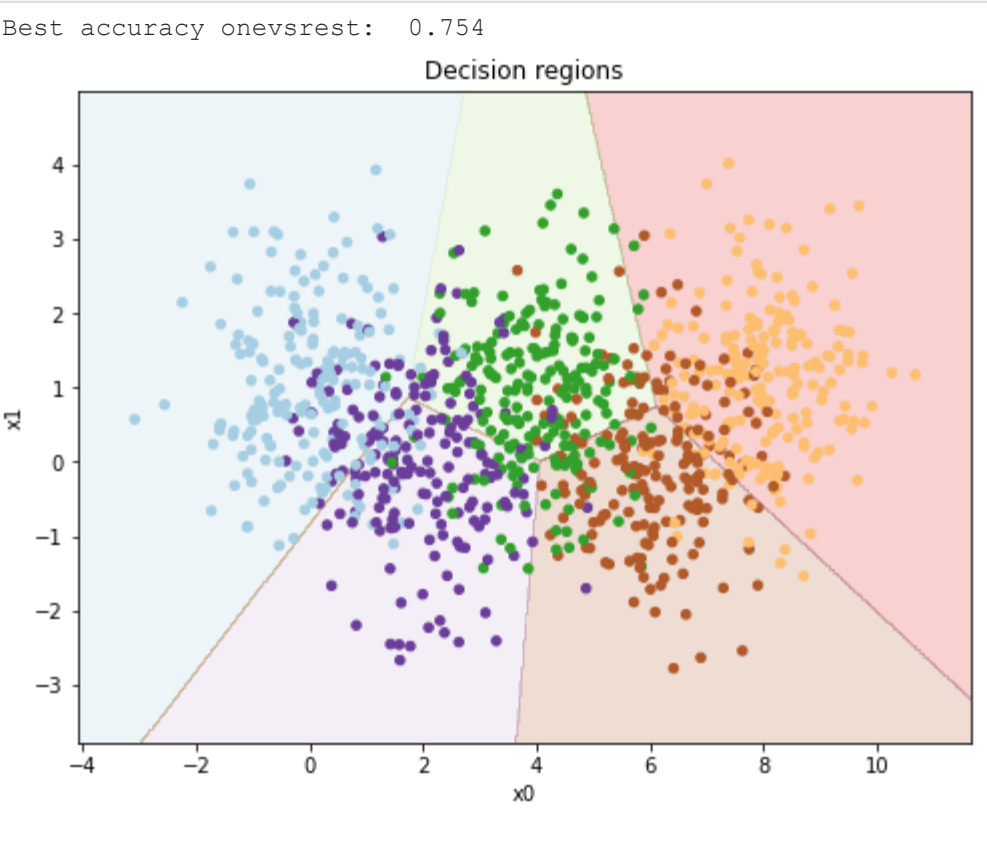
In [22]:
```python
cl = OnevsRest()

# finner beste eta og epochs for OnevsRest
for eta in np.linspace(0.01, 1, 10):
    accs = []
    for epoch in range(10, 3000, 300):
        cl.fit(X_train, t_train, eta, epoch)
        accs.append(cl.accuracy(X_val, t_val))
    plt.plot(range(10, 3000, 300), accs, label=str(round(eta, 3)))
    plt.legend()
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.title("Hyper-paramater settings")
plt.show()
```



We observe from the graph that the best accuricies are achieved by using a learning rate eta between 0.12 to 0.45 for apporiximately 2000 epochs. Slower learning rates would require many more epochs, and greater learning rates quickly diverge.

In [23]:
```
cl = OnevsRest()
cl.fit(X_train, t_train, 0.34, 2500)
plot_decision_regions(X_train, t_train, cl)
print("Best accuracy onevsrest: ", cl.accuracy(X_val, t_val))
```

```
Best accuracy onevsrest:  0.754
```



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 exponetiation 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, t) 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 [24]:
```
# Your code

m = [np.average(X_train[:,i-1:i]) for i in range(1, int(X_train.shape[1])+1)]
s = [np.std(X_train[:,i-1:i]) for i in range(1, int(X_train.shape[1])+1)]

def scale(x, m, s):
    return (x-m)/s
```

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 [25]:
```
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* off from *t_train*.

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 rerun 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 [26]:
```
# Your code

def OneHotEncode(t):
    classnum = len(set(t))
    encoded = []
    for e in t:
        vec = np.zeros((1, classnum), dtype="int")
        vec[0, e] = 1
        encoded.append(vec)
    return np.array(encoded)[:, 0,]


t = OneHotEncode(t_train)

dim_in = X_train.shape[1]
dim_out = len(t[0])
```

In [27]:
```
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 [28]:
```
weights1
```

Out[28]:
```
array([[-0.6938717 , -0.00133246, -0.54675803],
       [-0.63643285,  0.26220593, -0.01840165],
       [ 0.56237224,  0.20852872,  0.56139063]])
```

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 [29]:
```
# Your code

def add_bias_minus(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
        m = sh[0]
        bias = np.ones((m,1)) # Makes a m*1 matrix of 1-s
        return np.concatenate([(-bias), X], axis  = 1)


# hidden activations =
def hidden_activations(X_train, weights1, dim_hidden):
    biased_input = add_bias_minus(X_train)
    hidden_output = np.zeros(biased_input.shape)

    for node in range(dim_hidden):
        z = biased_input@weights1[:, node:node+1]
        hidden_output[:,node:node+1] = np.array([logistic(z)]).T

    return hidden_output
```

In [30]:
```
# Your code
# output_activations =
def output_activations(hidden_output, weights2, dim_out):
    biased_hidden_input = add_bias_minus(hidden_output)
    output = np.zeros((biased_hidden_input.shape[0], dim_out))

    for node in range(dim_out):
        z = biased_hidden_input@weights2[:, node:node+1]
        output[:, node:node+1] = np.array([logistic(z)]).T

    return output
```

To 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 in both layers. If you have done anything differently from us, you will not get the same numbers. But you may still be on the right track!

In [31]:
```
print(output_activations(hidden_activations(scale(X_train, m, s), weights1, dim_hidden), weights2, dim_out)[0,
```

```
[0.28969058 0.44120276 0.41012141 0.38135763 0.44130415]
```

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 [32]:
```
# Your code
hidden_output = hidden_activations(X_train, weights1, dim_hidden)
output = output_activations(hidden_activations(scale(X_train, m, s), weights1, dim_hidden), weights2, dim_out)

delta_output = (output-t)*output*(1-output)
```

Calculate the delta terms in the hidden layer.

In [33]:
```
# Your code
delta_hidden = (hidden_output*(1-hidden_output))*(delta_output@weights2[1:].T)

eta = 0.01
biased_input = add_bias_minus(X_train)
biased_hidden_input = add_bias_minus(hidden_output)

weights1 -= eta*biased_input.T@delta_hidden
weights2 -= eta*biased_hidden_input.T@delta_output
```

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 [34]:
```
print("New weights:")
print(weights1)
```

```
New weights:
[[-0.65021197  0.01584468 -0.57261548]
 [-0.70595067  0.40262588  0.351424  ]
 [ 0.51794875  0.17655113  0.59547216]]
```

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 classfier 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 [71]:
```
class MNNClassifier():
    """A multi-layer neural network with one hidden layer"""

    def __init__(self,eta = 0.001, dim_hidden = 6):
        """Initialize the hyperparameters"""
        self.eta = eta
        self.dim_hidden = dim_hidden

        # Should you put additional code here?

    def fit(self, X_train, t_train, epochs = 2000):
        """Initialize the weights. Train *epochs* many epochs."""

        # Initilaization
        # Fill in code for initalization
        t = OneHotEncode(t_train)
        self.dim_in = X_train.shape[1]
        self.dim_out = len(t[0])

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

        biased_input = add_bias_minus(X_train)


        for e in range(epochs):
            # Run one epoch of forward-backward
            #Fill in the code
            hid_out, out = self.forward(X_train)
            biased_hid_out = add_bias_minus(hid_out)

            delta_output = (out-t)*out*(1-out)
            delta_hidden = (hid_out*(1-hid_out))*(delta_output@self.weights2[1:].T)

            self.weights1 -= self.eta*biased_input.T@delta_hidden
            self.weights2 -= self.eta*biased_hid_out.T@delta_output

    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"""
        #Fill in the code
        biased_input = add_bias_minus(X)
        hidden_output = np.zeros((biased_input.shape[0], self.dim_hidden))
        for node in range(self.dim_hidden):
            z = biased_input@self.weights1[:, node:node+1]
            hidden_output[:,node:node+1] = np.array([logistic(z)]).T

        biased_hidden_input = add_bias_minus(hidden_output)
        output = np.zeros((biased_hidden_input.shape[0], self.dim_out))

        for node in range(self.dim_out):
            z = biased_hidden_input@self.weights2[:, node:node+1]
            output[:, node:node+1] = np.array([logistic(z)]).T

        return hidden_output, output

    def accuracy(self, X_test, t_test):
        """Calculate the accuracy of the classifier for the pair (X_test, t_test)
        Return the accuracy"""
        #Fill in the code
        out = self.argmax(self.forward(X_test)[1])
        preds = []
        for i in range(out.shape[0]):
            preds.append(np.argmax(out[i]))
        preds = np.array(preds)
        ratio = np.average((t_test==preds))
        return ratio

        # bare for binary dataset
    def recall(self, X_test, t_test):
        out = self.argmax(self.forward(X_test)[1])
        pred = []
        for i in range(out.shape[0]):
            pred.append(np.argmax(out[i]))
        pred = np.array(pred)
        true_p = 0
        false_n = 0
        for j in range(pred.shape[0]):
            if pred[j]==1:
                if pred[j]==t_test[j]:
                    true_p += 1
            else:
                if pred[j]!=t_test[j]:
                    false_n += 1

        return round(true_p/(true_p+false_n), 3)

        # bare for binary dataset
    def precision(self, X_test, t_test):
        out = self.argmax(self.forward(X_test)[1])
        pred = []
        for i in range(out.shape[0]):
            pred.append(np.argmax(out[i]))
        pred = np.array(pred)
        true_p = 0
        false_p = 0
        for j in range(pred.shape[0]):
            if pred[j]==1:
                if pred[j]==t_test[j]:
                    true_p += 1
                else:
                    false_p += 1
        return round(true_p/(true_p+false_p), 3)

    def argmax(self, output):
        preds = []
        for pred in output:
            out = np.zeros(pred.shape, dtype="int")
            out[np.argmax(pred)] = 1
            preds.append(out)
        return np.array(preds)

    def predict(self, X):
        return np.array([np.argmax(i) for i in self.forward(X)[1]])


    def add_bias_minus(self, 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
            m = sh[0]
            bias = np.ones((m,1)) # Makes a m*1 matrix of 1-s
            return np.concatenate([(-bias), X], axis  = 1)
```

Multi-class

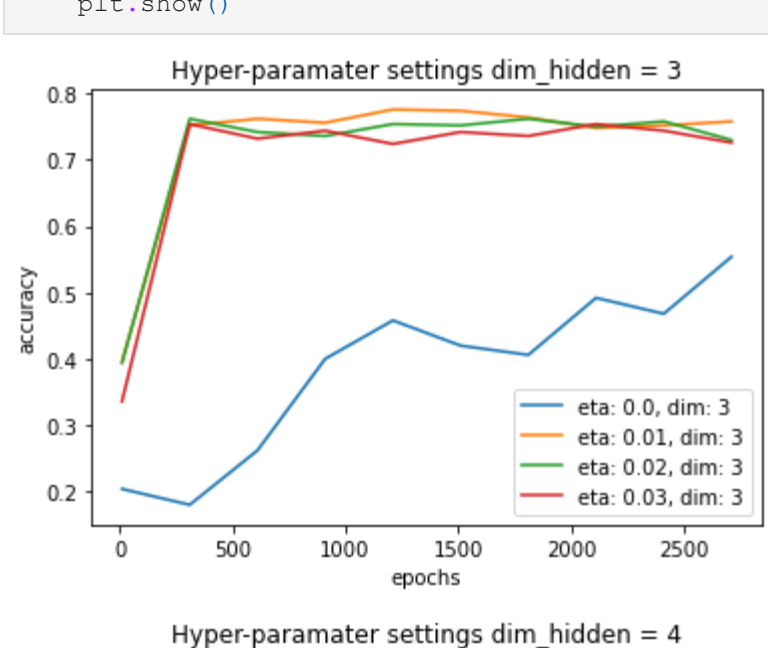
Train the network on (X_train, t_train) (after scaling), and test on (X_val, t_val). Tune the hyperparameters to get the best result:

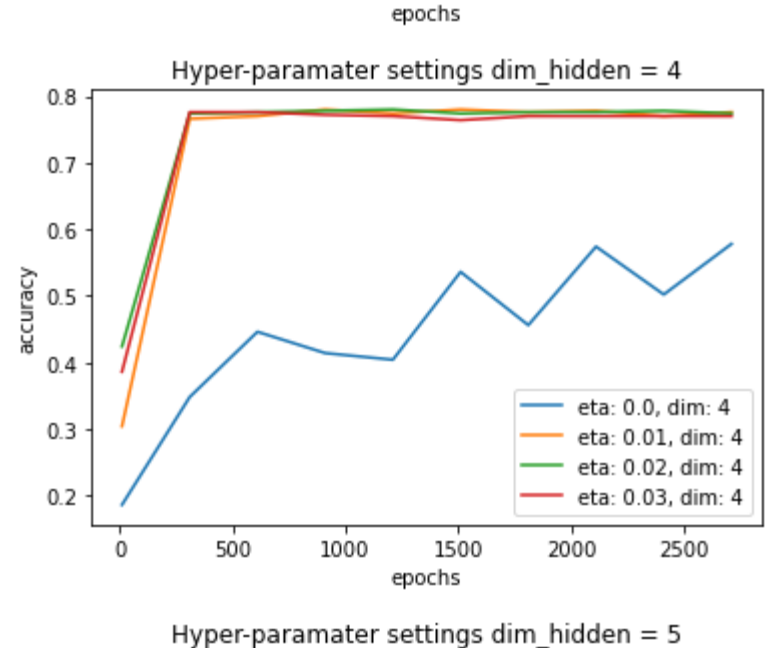
- number of epochs
- learning rate
- number of hidden nodes.

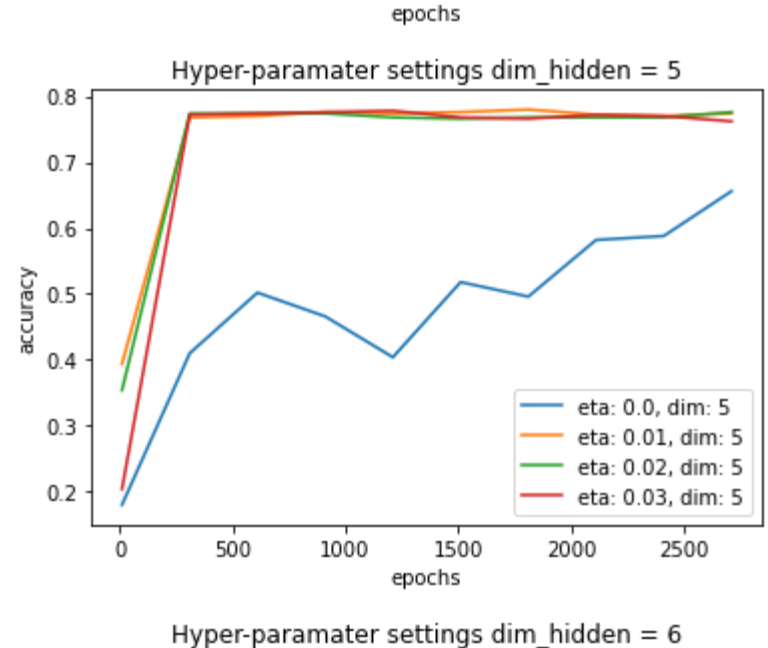
When you are content with the hyperparameters, you should run the same experiment 10 times, collect the accuracies and report the mean value and standard deviation of the accuracies across the experiments. This is common practise when you apply neural networks as the result may vary slightly between the runs. You may plot the decision boundaries for one of the runs.

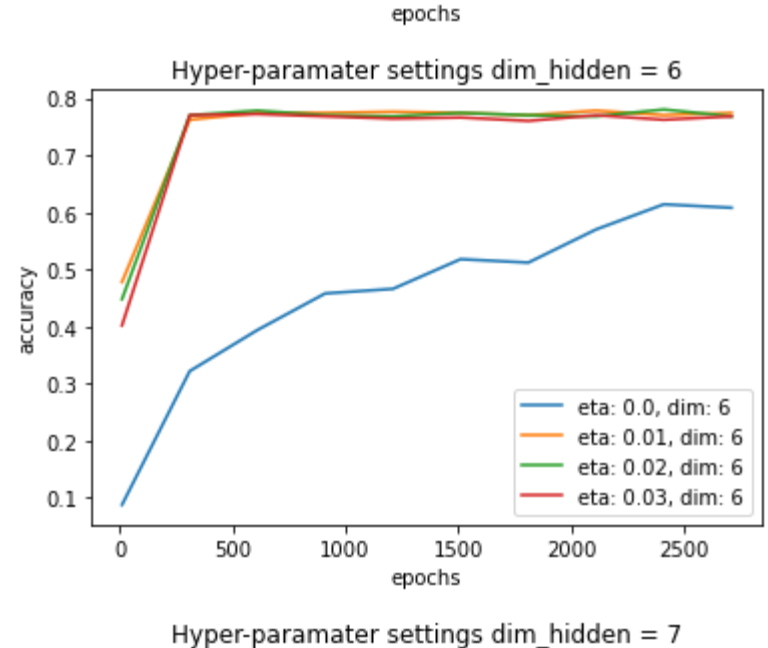
Discuss shortly how the results and decsion boundaries compare to the "one-vs-rest" classifier.

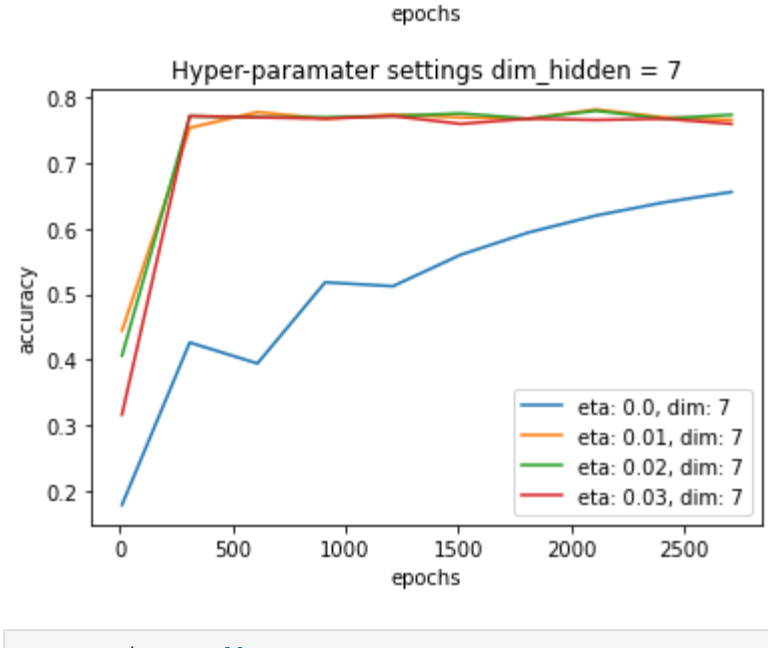
In [36]:
```
X = scale(X_train, m, s)
X_val_scaled = scale(X_val, m, s)
for dim_hidden in np.arange(3, 8):
    for eta in np.linspace(0.0001, 0.03, 4):
        cl = MNNClassifier(eta, dim_hidden)
        accs = []
        for epoch in range(10, 3000, 300):
            cl.fit(X, t_train, epoch)
            accs.append(cl.accuracy(X_val_scaled, t_val))
        plt.plot(range(10, 3000, 300), accs, label="eta: "+str(round(eta, 3))+", dim: "+str(dim_hidden))
        plt.legend()
    plt.xlabel("epochs")
    plt.ylabel("accuracy")
    plt.title("Hyper-paramater settings dim_hidden = " + str(dim_hidden))
    plt.show()
```





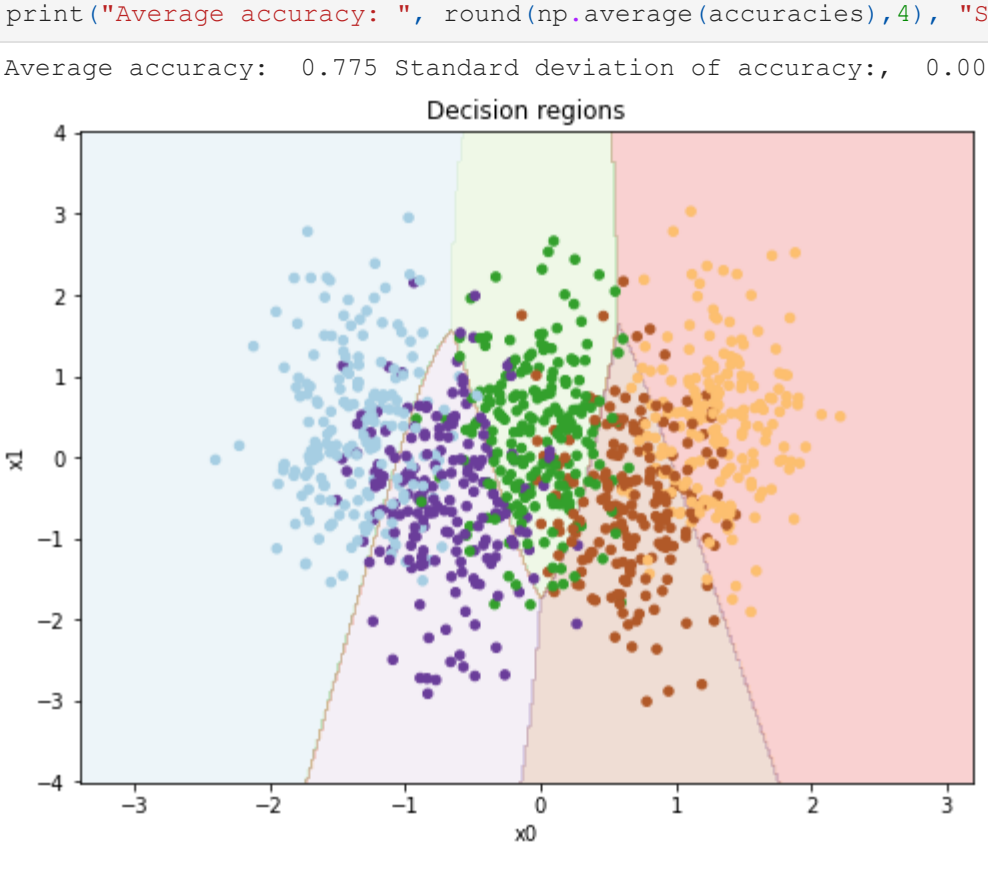






In [37]:
```
accuracies = []
X = scale(X_train, m, s)

for i in range(10):
    cl = MNNClassifier(0.01, dim_hidden=4)
    cl.fit(X, t_train, 1500)
    X_val_scaled = scale(X_val, m, s)
    accuracies.append(cl.accuracy(X_val_scaled, t_val))

plot_decision_regions(X, t_train, cl)
print("Average accuracy: ", round(np.average(accuracies),4), "Standard deviation of accuracy:, ", round(np.std(
```

```
Average accuracy:  0.775 Standard deviation of accuracy:,  0.0022
```



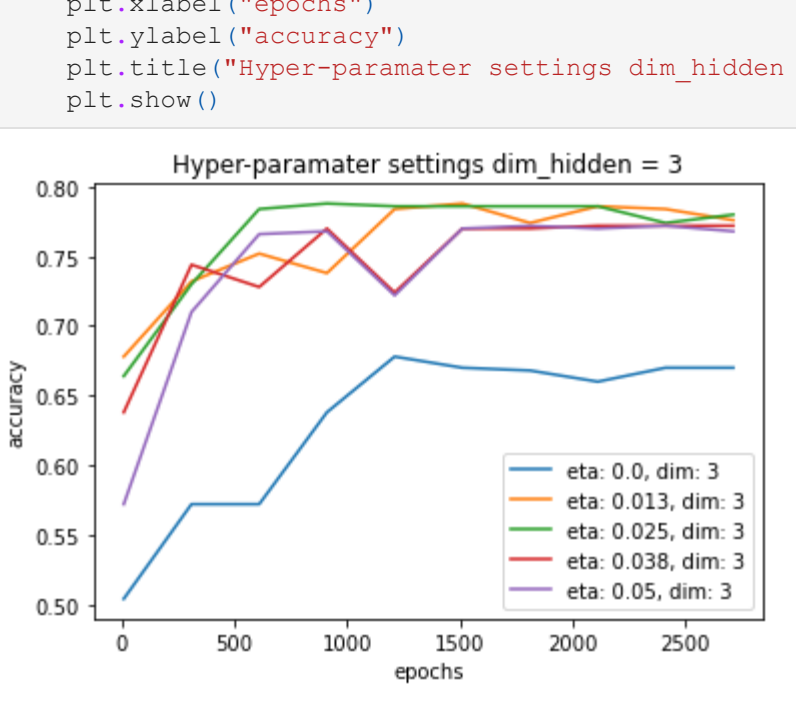
Running multiple tests for the hyperparameters, I concluded that the best fit was found with a learning rate of 0.01 and that the accuracy stabilized for epochs greater than 1500. The classifier did performed equally for these settings on different dimentions of the hidden layers between the value of 4 and 7. I will thus use 4 layers as this will run slightly faster.

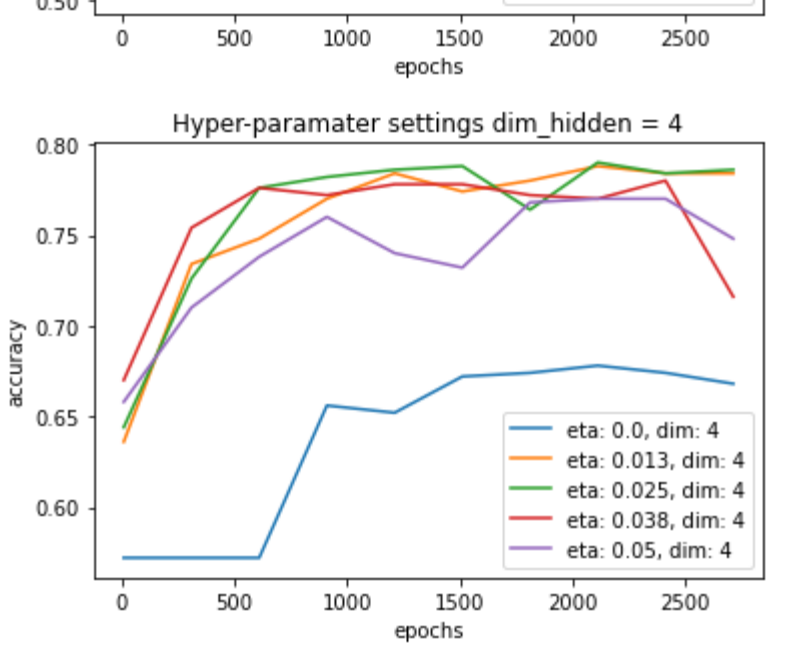
The reported accuracy is slightly greater than One-vs-rest, coming in at an average accuracy of 77.66 with a low standard deviation of 0.0031, with the One-vs-rest classifier coming in at a maximum accuracy of 75.4. This small increase comes from the MLPs ability to draw non-linear lines to thread the needle between different classes. This does not offer a great increase since the data is difficult to seperate, but picks up a few extra objects.

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 [ ]:
```
X = scale(X_train, m, s)
X_val_scaled = scale(X_val, m, s)
for dim_hidden in np.arange(3, 8):
    for eta in np.linspace(0.0001, 0.05, 5):
        cl = MNNClassifier(eta)
        accs = []
        for epoch in range(10, 3000, 300):
            cl.fit(X, t2_train, epoch)
            accs.append(cl.accuracy(X_val_scaled, t2_val))
        plt.plot(range(10, 3000, 300), accs, label="eta: "+str(round(eta, 3))+", dim: "+str(dim_hidden))
        plt.legend()
    plt.xlabel("epochs")
    plt.ylabel("accuracy")
    plt.title("Hyper-paramater settings dim_hidden = " + str(dim_hidden))
    plt.show()
```

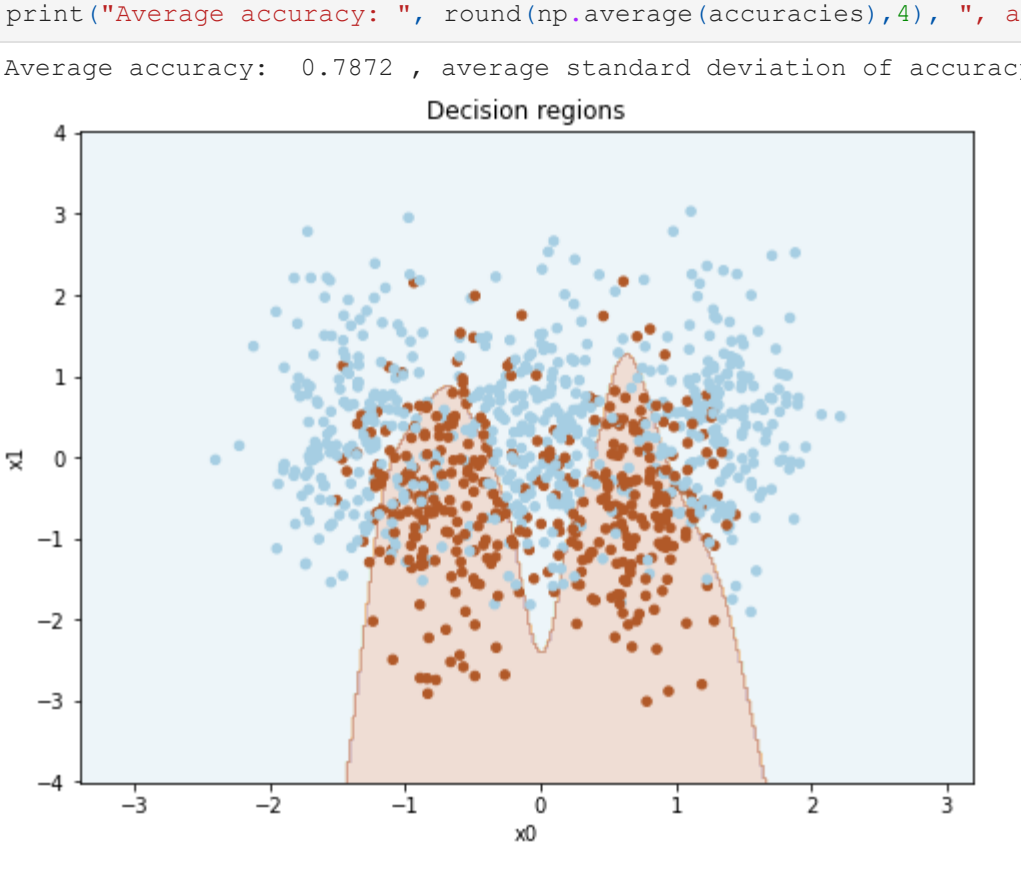




We observe that the best settings of hyperparameter are a learning rate of 0.025, with an epoch greater than 1000. Dim_hidden seems to make little difference as long as above 3. I would have liked to group the plots better, but making a 2x3 grid via subplot only made the plots smaller and unreadable due to the legends covering the graph. I would much appreciate feedback on how to make better plots.

In [79]:
```
accuracies = []
for i in range(10):
    cl = MNNClassifier(eta=0.022, dim_hidden=6)
    X = scale(X_train, m, s)
    cl.fit(X, t2_train, 1500)
    X_val_scaled = scale(X_val, m, s)
    accuracies.append(cl.accuracy(X_val_scaled, t2_val))

plot_decision_regions(X, t2_train, cl)
print("Average accuracy: ", round(np.average(accuracies),4), ", average standard deviation of accuracy:, ", rou
```

```
Average accuracy:  0.7872 , average standard deviation of accuracy:,  0.0013
```



The MNN classifier does a much better job than the regression classifiers, with an average accuracy of 0.786 vs 0.668. Again the strength comes from the ability to make nonlinear decision boundaries which better encapsulate the desired classes.

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, t_val) as valuation set when training on (X_train, t_train).

Let *e=50* or *e=10* (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 *t* after *e* many epochs, where *t* is a threshold you provide as a parameter.

Run the classifier with various values for *t* and report the accuracy and the numberof 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 so 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 preceptron. 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 [77]:
```
# won't load if you don't have tabulate, see PDF for results otherwise
from tabulate import tabulate

head = ["LinReg", "LogReg", "MNN"]
side = ["train", "val", "test"]

linreg_cl = NumpyLinRegClass()
logreg_cl = NumpyLogReg()
MNN_cl = MNNClassifier(0.022)

linreg_cl.fit(X_train, t2_train, eta=0.072, epochs=300, loss_diff=0.1)
logreg_cl.fit(X_train, t2_train, eta=0.02, epochs=250)
MNN_cl.fit(scale(X_train, m, s), t2_train, epochs=1500)


linreg_acc = []
logreg_acc = []
mnn_acc = []

linreg_rec = []
logreg_rec = []
mnn_rec = []

linreg_pre = []
logreg_pre = []
mnn_pre = []


linreg_acc.append(linreg_cl.accuracy(X_train, t2_train))
logreg_acc.append(logreg_cl.accuracy(X_train, t2_train))
mnn_acc.append(MNN_cl.accuracy(scale(X_train,m,s), t2_train))

linreg_acc.append(linreg_cl.accuracy(X_val, t2_val))
logreg_acc.append(logreg_cl.accuracy(X_val, t2_val))
mnn_acc.append(MNN_cl.accuracy(scale(X_val,m,s), t2_val))

linreg_acc.append(linreg_cl.accuracy(X_test, t2_test))
logreg_acc.append(logreg_cl.accuracy(X_test, t2_test))
mnn_acc.append(MNN_cl.accuracy(scale(X_test,m,s), t2_test))

linreg_rec.append(linreg_cl.recall(X_train, t2_train))
logreg_rec.append(logreg_cl.recall(X_train, t2_train))
mnn_rec.append(MNN_cl.recall(scale(X_train,m,s), t2_train))

linreg_rec.append(linreg_cl.recall(X_val, t2_val))
logreg_rec.append(logreg_cl.recall(X_val, t2_val))
mnn_rec.append(MNN_cl.recall(scale(X_val,m,s), t2_val))

linreg_rec.append(linreg_cl.recall(X_test, t2_test))
logreg_rec.append(logreg_cl.recall(X_test, t2_test))
mnn_rec.append(MNN_cl.recall(scale(X_test,m,s), t2_test))

linreg_pre.append(linreg_cl.precision(X_train, t2_train))
logreg_pre.append(logreg_cl.precision(X_train, t2_train))
mnn_pre.append(MNN_cl.precision(scale(X_train,m,s), t2_train))

linreg_pre.append(linreg_cl.precision(X_val, t2_val))
logreg_pre.append(logreg_cl.precision(X_val, t2_val))
mnn_pre.append(MNN_cl.precision(scale(X_val,m,s), t2_val))

linreg_pre.append(linreg_cl.precision(X_test, t2_test))
logreg_pre.append(logreg_cl.precision(X_test, t2_test))
mnn_pre.append(MNN_cl.precision(scale(X_test,m,s), t2_test))

print("Binary data accuracy over datasets:")
print(tabulate(np.array([side, linreg_acc, logreg_acc, mnn_acc]).T, headers=head))

print("\nBinary data recall for class 1 over datasets:")
print(tabulate(np.array([side, linreg_rec, logreg_rec, mnn_rec]).T, headers=head))

print("\nBinary data precision for class 1 over datasets:")
print(tabulate(np.array([side, linreg_pre, logreg_pre, mnn_pre]).T, headers=head))



head = ["OnevsRest", "MNN"]

OVR_cl = OnevsRest()
MNN_cl = MNNClassifier(eta=0.01, dim_hidden=4)


OVR_cl.fit(X_train, t_train, 0.34, 2500)
MNN_cl.fit(scale(X_train, m, s), t_train, epochs=2000)


ovr_acc = []
mnn_acc = []

ovr_acc.append(OVR_cl.accuracy(X_train, t_train))
mnn_acc.append(MNN_cl.accuracy(scale(X_train,m,s), t_train))

ovr_acc.append(OVR_cl.accuracy(X_val, t_val))
mnn_acc.append(MNN_cl.accuracy(scale(X_val,m,s), t_val))

ovr_acc.append(OVR_cl.accuracy(X_test, t_test))
mnn_acc.append(MNN_cl.accuracy(scale(X_test,m,s), t_test))

print("\nMulti-class data accuracy over datasets:")
print(tabulate(np.array([side, ovr_acc, mnn_acc]).T, headers=head))
```

Binary data accuracy over datasets:

	LinReg	LogReg	MNN
train	0.707	0.713	0.798
val	0.668	0.67	0.786
test	0.726	0.724	0.802

Binary data recall for class 1 over datasets:

	LinReg	LogReg	MNN
train	0.503	0.433	0.715
val	0.444	0.407	0.692
test	0.515	0.45	0.715

Binary data precision for class 1 over datasets:

	LinReg	LogReg	MNN
train	0.658	0.711	0.75
val	0.669	0.696	0.783
test	0.72	0.763	0.773

Multi-class data accuracy over datasets:

	OnevsRest	MNN
train	0.75	0.794
val	0.754	0.776
test	0.778	0.798

For the binary data the regression classifiers are more or less equal, with the MNN classifier outperforming both of them by a 0.1 margin. The classifiers actually did the best on the test data, contrary to the belief that the classifiers would perform the worst on the test data. The MNN classifier also performs the best when it comes to recall and precision for class 1, with logreg performing especially poorly coming in at a 0.43 average.

The fact that the logreg classifier has high precision and low recall for class 1 means that it classifies too few elements as the positive class 1, but the elements classified as the positive class are usually classified correctly. The linreg classifier has a similar relationship with a higher precision than recall, but the logreg classifier has a larger difference.

The MNN classifier has both a high precision and a high recall, meaning that it classifies a high amount of elements that are positive as the positive class, and that these are often correctly classified as they fit with the labels.

For the multi-class data the results are very consistant accross the different data sets for the different classifiers. The MNN classifier gives the best result with an average accuracy of 0.79, which is a noteable increase in accuracy over the OnevsRest's 0.75. Again, there is no decrease in performance for the test data, in fact it is where the multi-class classifiers usually perform the best.