syde552-assignment4

April 5, 2024

1 SYDE 552 Assignment 4: Basal Ganglia

1.0.1 Due Monday, April 8, 11:59pm

1.0.2 Value: 15% of total marks for the course

This assignment covers various forms of Action Selection, covering a standard feed-forward neural network model and a winner-take-all model. The last section adds biological details to the winner-take-all model (synapses). The intent is to show the creation a modification of small specific-purpose networks and to explore how timing affects neurons.

You can work in groups to do the assignment, but your answers and code must be original to you. Your submission will be a filled-out copy of this notebook (cells for code and written answers provided).

2 1. Action Selection with Feed-forward Neural Networks

In order for the brain to choose which of many possible actions to select to perform at any given moment, it needs to be able to takke a list of numbers (indicating how good each action is in the current state, sometimes called the Value) and create an output that indicates which one is the largest. For example, with the input `[0.2, 0.5, 0.8, 0.3]` we might want the output `[0, 0, 1, 0]`.

One way to approach this task is to train a normal feed-forward neural network on this task. This would be similar to the digit recognition task from Assignment 2, except we would just have the list of values as input instead of the image.

To create the dataset for training the network, we can just generate a random set of values between 0 and 1, and then compute which one is biggest to produce our target value.

[1]:
```
import numpy as np

N = 4         # how many values in the list
M = 10000     # how many random examples to make in the dataset

X = np.random.uniform(0,1,(M,N))
Y = np.argmax(X, axis=1)
```

In order to train with this data in pytorch, we need to turn that array into a pytorch dataset, and to split it into training and testing.

[2]:
```
import torch
dataset = torch.utils.data.TensorDataset(torch.Tensor(X),torch.Tensor(Y).
 ↪long()) # create your datset

train_dataset, test_dataset = torch.utils.data.random_split(dataset,[0.8,0.2])
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=1000,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1000,␣
 ↪shuffle=True)
```

Now we have to define our network. Here we define a simple neural network with 4 inputs, a hidden layer of 100 neurons, and 4 outputs.

[3]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(4, 100)    # the weights from the input to the new␣
 ↪learned features (hidden layer)
        self.fc2 = nn.Linear(100, 4)    # the weights from the hidden layer to␣
 ↪the output

    def forward(self, x):
        # the processing the network will do
        x = F.relu(self.fc1(x))         # apply the first set of weights, then␣
 ↪apply the ReLU neuron model
        x = self.fc2(x)                 # apply the second set of weights
        return x

network = Net()
```

Finally, we have to train the model. Here is the exact same code from Assignemnt 2 for training

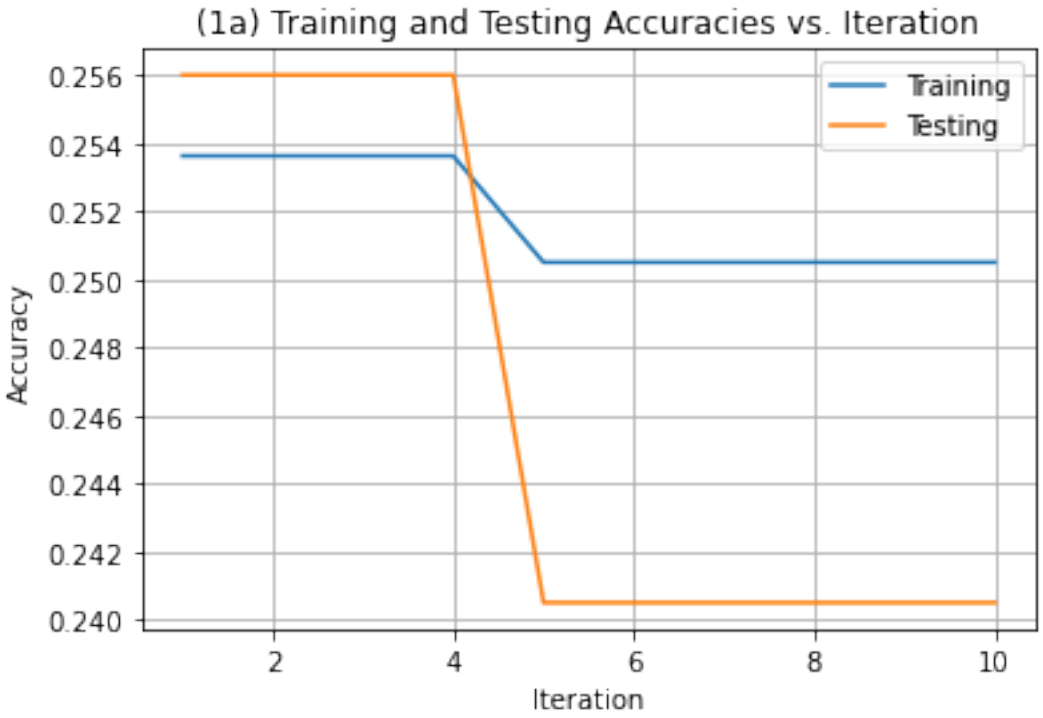
[4]:
```
# create the learning rule
optimizer = torch.optim.SGD(network.parameters(),
                            lr=0.1,   # learning rate
                            momentum=0.5)

# variables to keep track of the training and testing accuracy
accuracy_train = []
accuracy_test = []

def continue_training():
    network.train()      # configure the network for training
    for i in range(10):  # train the network 10 times
```

```
        correct = 0
        for data, target in train_loader:        # working in batchs of 1000
            optimizer.zero_grad()                # initialize the learning system
            output = network(data)               # feed in the data
            loss = F.nll_loss(output, target)    # compute how wrong the output␣
↪is
            loss.backward()                      # change the weights to reduce␣
↪error
            optimizer.step()                     # update the learning rule

            pred = output.data.max(1, keepdim=True)[1]           # compute␣
↪which output is largest
            correct += pred.eq(target.data.view_as(pred)).sum()  # compute the␣
↪number of correct outputs
    # update the list of training accuracy values
    score = float(correct/len(train_loader.dataset))
    accuracy_train.append(score)
    print('Iteration', len(accuracy_train), 'Training accuracy:', score)

    correct = 0
    network.eval()
    for data, target in test_loader:    # go through the test data once (in␣
↪groups of 1000)
        output = network(data)                                  # feed in the data
        pred = output.data.max(1, keepdim=True)[1]              # compute which␣
↪output is largest
        correct += pred.eq(target.data.view_as(pred)).sum()  # compute the␣
↪number of correct outputs
    # update the list of testing accuracy values
    score = float(correct/len(test_loader.dataset))
    accuracy_test.append(score)
    print('Iteration', len(accuracy_test), 'Testing accuracy:', score)
```

1. a) [1 mark] Call `continue_training()` 10 times and then plot the training and testing accuracy (as you did in assignment 2 question 3a.

[5]:
```
import matplotlib.pyplot as plt
for _ in range(10):
    continue_training()
f, pl1 = plt.subplots(1,1)
pl1.plot(np.arange(1, 11), accuracy_train, label="Training")
pl1.plot(np.arange(1, 11), accuracy_test, label="Testing")
pl1.set(title="(1a) Training and Testing Accuracies vs. Iteration",␣
 ↪xlabel="Iteration", ylabel="Accuracy")
pl1.legend()
pl1.grid()
```

```
Iteration 1 Training accuracy: 0.2536250054836273
Iteration 1 Testing accuracy: 0.25600001215934753
Iteration 2 Training accuracy: 0.2536250054836273
Iteration 2 Testing accuracy: 0.25600001215934753
Iteration 3 Training accuracy: 0.2536250054836273
Iteration 3 Testing accuracy: 0.25600001215934753
Iteration 4 Training accuracy: 0.2536250054836273
Iteration 4 Testing accuracy: 0.25600001215934753
Iteration 5 Training accuracy: 0.25049999356269836
Iteration 5 Testing accuracy: 0.24050000309944153
Iteration 6 Training accuracy: 0.25049999356269836
Iteration 6 Testing accuracy: 0.24050000309944153
Iteration 7 Training accuracy: 0.25049999356269836
Iteration 7 Testing accuracy: 0.24050000309944153
Iteration 8 Training accuracy: 0.25049999356269836
Iteration 8 Testing accuracy: 0.24050000309944153
Iteration 9 Training accuracy: 0.25049999356269836
Iteration 9 Testing accuracy: 0.24050000309944153
Iteration 10 Training accuracy: 0.25049999356269836
Iteration 10 Testing accuracy: 0.24050000309944153
```



1. b) [1 mark] In the previous question, the model did not get a very high accuracy. To try to improve the accuracy, let's vary the number of neurons. Try using 200 neurons, 500 neurons, and 1000 neurons. For each one, generate the same plot as in 1a. Does making the network larger in

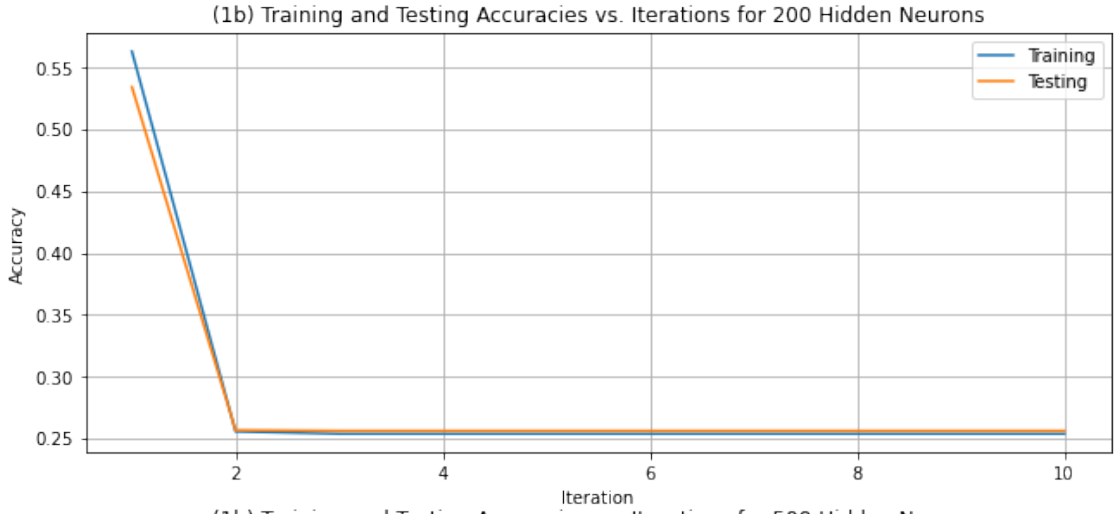
this way improve performance significantly?

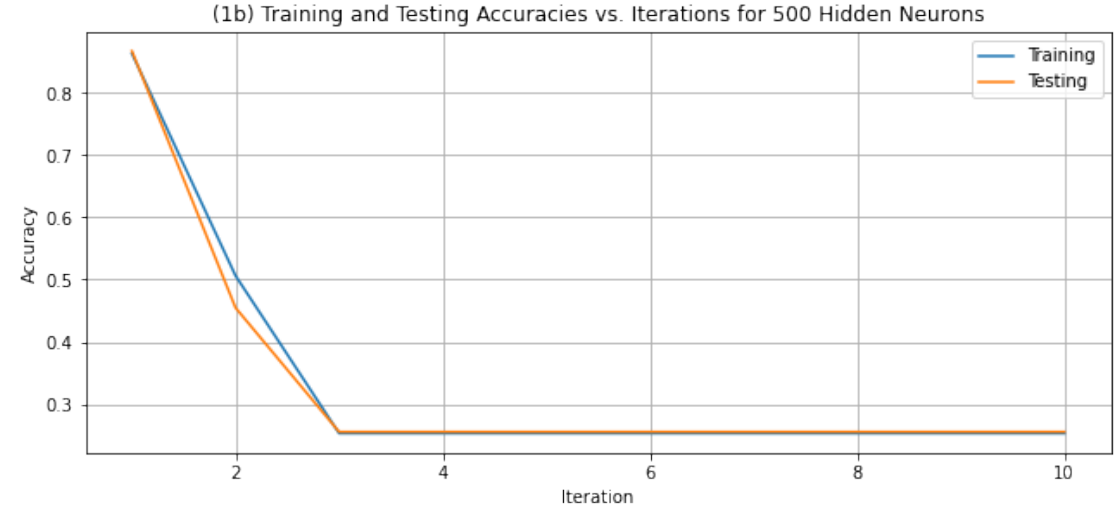
[6]:
```
class AdaptNet(nn.Module):
    def __init__(self, n):
        super(AdaptNet, self).__init__()
        self.fc1 = nn.Linear(4, n)    # the weights from the input to the new learned features (hidden layer)
        self.fc2 = nn.Linear(n, 4)    # the weights from the hidden layer to the output

    def forward(self, x):
        # the processing the network will do
        x = F.relu(self.fc1(x))         # apply the first set of weights, then apply the ReLU neuron model
        x = self.fc2(x)                 # apply the second set of weights
        return x

n_set = [200, 500, 1000]
accuracy_train_set = []
accuracy_test_set = []
for n in n_set:
    network = AdaptNet(n)
    # create the learning rule
    optimizer = torch.optim.SGD(network.parameters(),
                                lr=0.01,    # learning rate
                                momentum=0.5)
    accuracy_train = []
    accuracy_test = []
    for _ in range(10):
        continue_training()
    accuracy_train_set.append(accuracy_train)
    accuracy_test_set.append(accuracy_test)

f, (pl1, pl2, pl3) = plt.subplots(3,1)
f.subplots_adjust(top=3,right=1.5)
pl = [pl1, pl2, pl3]
for i in range(3):
    pl[i].plot(np.arange(1,11),accuracy_train_set[i],label="Training")
    pl[i].plot(np.arange(1,11),accuracy_test_set[i],label="Testing")
    pl[i].set(title=f"(1b) Training and Testing Accuracies vs. Iterations for {n_set[i]} Hidden Neurons",xlabel="Iteration",ylabel="Accuracy")
    pl[i].grid()
    pl[i].legend()
```

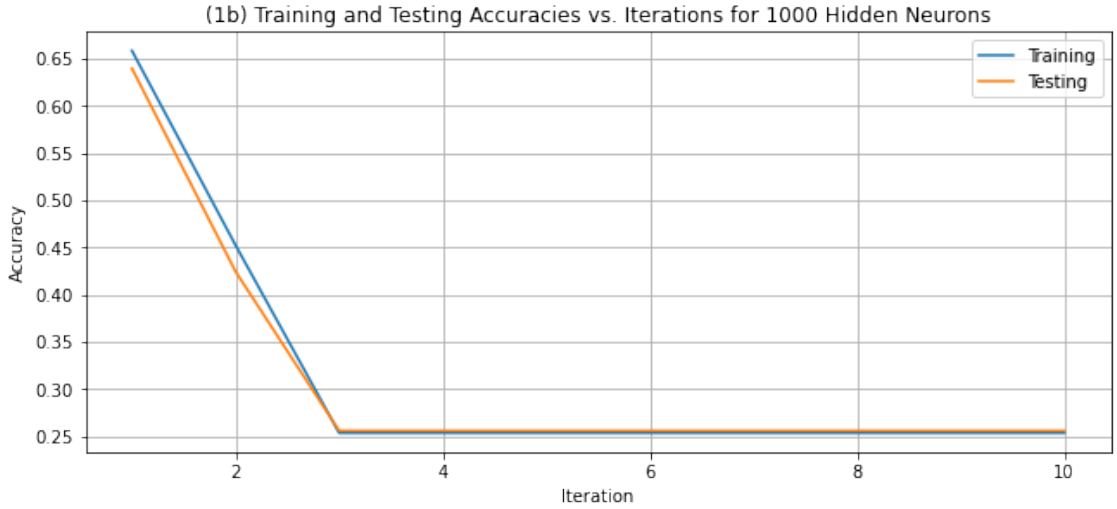
```
Iteration 1 Training accuracy: 0.5632500052452087
Iteration 1 Testing accuracy: 0.534500002861023
Iteration 2 Training accuracy: 0.2554999887943268
Iteration 2 Testing accuracy: 0.2565000057220459
```

```
Iteration 3 Training accuracy: 0.2536250054836273
Iteration 3 Testing accuracy: 0.25600001215934753
Iteration 4 Training accuracy: 0.2536250054836273
Iteration 4 Testing accuracy: 0.25600001215934753
Iteration 5 Training accuracy: 0.2536250054836273
Iteration 5 Testing accuracy: 0.25600001215934753
Iteration 6 Training accuracy: 0.2536250054836273
Iteration 6 Testing accuracy: 0.25600001215934753
Iteration 7 Training accuracy: 0.2536250054836273
Iteration 7 Testing accuracy: 0.25600001215934753
Iteration 8 Training accuracy: 0.2536250054836273
Iteration 8 Testing accuracy: 0.25600001215934753
Iteration 9 Training accuracy: 0.2536250054836273
Iteration 9 Testing accuracy: 0.25600001215934753
Iteration 10 Training accuracy: 0.2536250054836273
Iteration 10 Testing accuracy: 0.25600001215934753
Iteration 1 Training accuracy: 0.862625002861023
Iteration 1 Testing accuracy: 0.8665000200271606
Iteration 2 Training accuracy: 0.5063750147819519
Iteration 2 Testing accuracy: 0.45500001311302185
Iteration 3 Training accuracy: 0.2536250054836273
Iteration 3 Testing accuracy: 0.25600001215934753
Iteration 4 Training accuracy: 0.2536250054836273
Iteration 4 Testing accuracy: 0.25600001215934753
Iteration 5 Training accuracy: 0.2536250054836273
Iteration 5 Testing accuracy: 0.25600001215934753
Iteration 6 Training accuracy: 0.2536250054836273
Iteration 6 Testing accuracy: 0.25600001215934753
Iteration 7 Training accuracy: 0.2536250054836273
Iteration 7 Testing accuracy: 0.25600001215934753
Iteration 8 Training accuracy: 0.2536250054836273
Iteration 8 Testing accuracy: 0.25600001215934753
Iteration 9 Training accuracy: 0.2536250054836273
Iteration 9 Testing accuracy: 0.25600001215934753
Iteration 10 Training accuracy: 0.2536250054836273
Iteration 10 Testing accuracy: 0.25600001215934753
Iteration 1 Training accuracy: 0.6582499742507935
Iteration 1 Testing accuracy: 0.6395000219345093
Iteration 2 Training accuracy: 0.45237499475479126
Iteration 2 Testing accuracy: 0.4244999885559082
Iteration 3 Training accuracy: 0.2536250054836273
Iteration 3 Testing accuracy: 0.25600001215934753
Iteration 4 Training accuracy: 0.2536250054836273
Iteration 4 Testing accuracy: 0.25600001215934753
Iteration 5 Training accuracy: 0.2536250054836273
Iteration 5 Testing accuracy: 0.25600001215934753
Iteration 6 Training accuracy: 0.2536250054836273
Iteration 6 Testing accuracy: 0.25600001215934753
```

```
Iteration 7 Training accuracy: 0.2536250054836273
Iteration 7 Testing accuracy: 0.25600001215934753
Iteration 8 Training accuracy: 0.2536250054836273
Iteration 8 Testing accuracy: 0.25600001215934753
Iteration 9 Training accuracy: 0.2536250054836273
Iteration 9 Testing accuracy: 0.25600001215934753
Iteration 10 Training accuracy: 0.2536250054836273
Iteration 10 Testing accuracy: 0.25600001215934753
```







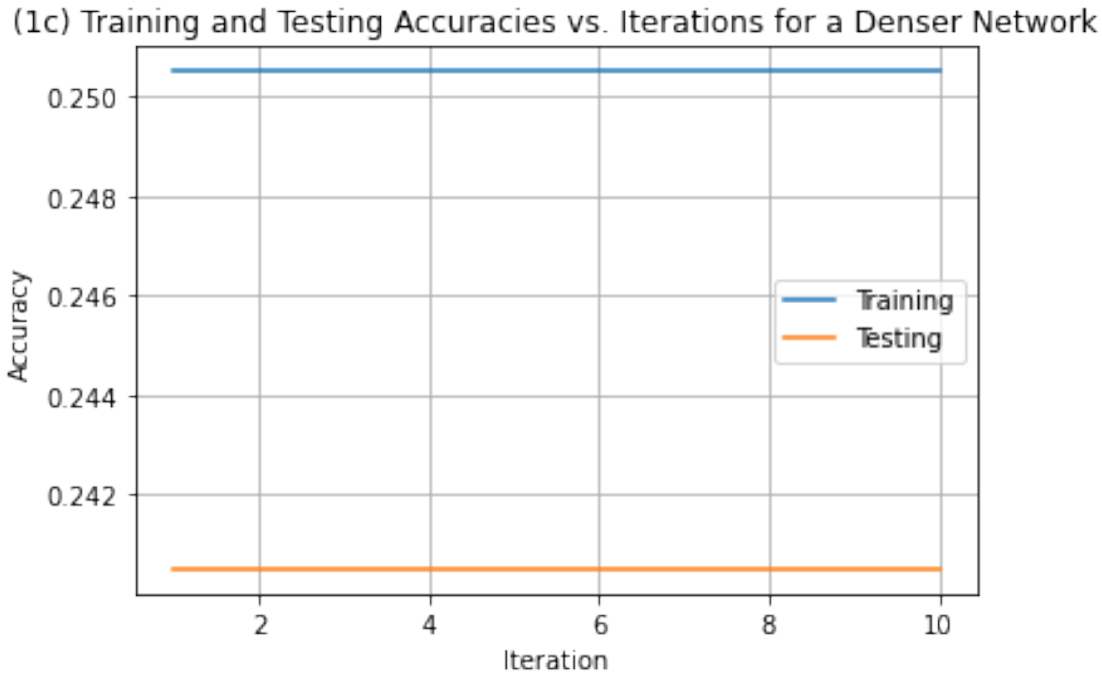
It appears that modifying the network in this way provides limited benefit – it still struggles to reach above 25% accuracy for the testing set. Overall, it does not perform well.

1. c) [1 mark] Try making the network deeper by adding a second layer of neurons. Use 500 neurons in each of the two middle layers. Generate the same plot as in 1a. Does making the network larger in this way improve performance significantly?

[7]:
```
class DeeperNet(nn.Module):
    def __init__(self):
        super(DeeperNet, self).__init__()
        self.fc1 = nn.Linear(4, 500)    # the weights from the input to the new␣
 ↪learned features (hidden layer)
        self.fc2 = nn.Linear(500, 500)    # the weights from the hidden layer␣
 ↪to the output
        self.fc3 = nn.Linear(500, 4)    # the weights from the hidden layer to␣
 ↪the output

    def forward(self, x):
        # the processing the network will do
        x = F.relu(self.fc1(x))         # apply the first set of weights, then␣
 ↪apply the ReLU neuron model
        x = F.relu(self.fc2(x))                 # apply the second set of␣
 ↪weights
        x = self.fc3(x)
        return x


network = DeeperNet()
optimizer = torch.optim.SGD(network.parameters(),
                            lr=0.01,   # learning rate
                            momentum=0.5)
accuracy_train = []
accuracy_test = []
for _ in range(10):
    continue_training()

f, pl1 = plt.subplots(1,1)
pl1.plot(np.arange(1,11),accuracy_train,label="Training")
pl1.plot(np.arange(1,11),accuracy_test,label="Testing")
pl1.set(title="(1c) Training and Testing Accuracies vs. Iterations for a Denser␣
 ↪Network",xlabel="Iteration",ylabel="Accuracy")
pl1.legend()
pl1.grid()
```

```
Iteration 1 Training accuracy: 0.25049999356269836
Iteration 1 Testing accuracy: 0.24050000309944153
Iteration 2 Training accuracy: 0.25049999356269836
Iteration 2 Testing accuracy: 0.24050000309944153
Iteration 3 Training accuracy: 0.25049999356269836
Iteration 3 Testing accuracy: 0.24050000309944153
```

```
Iteration 4 Training accuracy: 0.25049999356269836
Iteration 4 Testing accuracy: 0.24050000309944153
Iteration 5 Training accuracy: 0.25049999356269836
Iteration 5 Testing accuracy: 0.24050000309944153
Iteration 6 Training accuracy: 0.25049999356269836
Iteration 6 Testing accuracy: 0.24050000309944153
Iteration 7 Training accuracy: 0.25049999356269836
Iteration 7 Testing accuracy: 0.24050000309944153
Iteration 8 Training accuracy: 0.25049999356269836
Iteration 8 Testing accuracy: 0.24050000309944153
Iteration 9 Training accuracy: 0.25049999356269836
Iteration 9 Testing accuracy: 0.24050000309944153
Iteration 10 Training accuracy: 0.25049999356269836
Iteration 10 Testing accuracy: 0.24050000309944153
```



Similar to (1b), it appears that adding a layer also does not significantly improve the performance of the network, as it still stuggles to exceed 25% accuracy.

1. d) [1 mark] Starting with the original model in question 1a, let's modify the network in a different way. In particular, in the current version of the model we have no neuron model at the output. This is due to this part of the code:

```
def forward(self, x):
    # the processing the network will do
    x = F.relu(self.fc1(x))          # apply the first set of weights, then apply the ReLU
    x = self.fc2(x)                  # apply the second set of weights
```

```
        return x
```

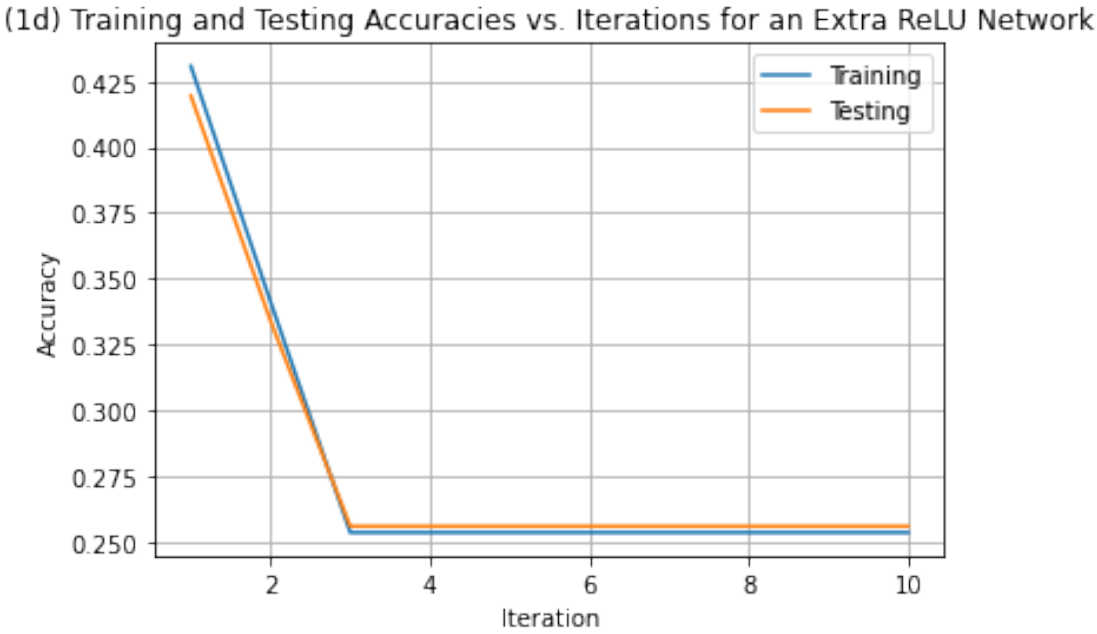
For the first line (the hidden layer), we use a Rectified Linear neuron model (`F.relu`). But we aren't doing that with the second line, where it creates the output. Let's try adding a neuron model there by changing `x = self.fc2(x)` to `x = F.relu(self.fc2(x))`.

Train the model again (by calling `continue_training` 10 times) and make the same plot as above. Does this improve performance significantly?

[8]:
```
class ExtraReLUNet(nn.Module):
    def __init__(self):
        super(ExtraReLUNet, self).__init__()
        self.fc1 = nn.Linear(4, 500)     # the weights from the input to the new␣
 ↪learned features (hidden layer)
        self.fc2 = nn.Linear(500, 4)     # the weights from the hidden layer to␣
 ↪the output

    def forward(self, x):
        # the processing the network will do
        x = F.relu(self.fc1(x))          # apply the first set of weights, then␣
 ↪apply the ReLU neuron model
        x = F.relu(self.fc2(x))                  # apply the second set of␣
 ↪weights
        return x


network = ExtraReLUNet()
optimizer = torch.optim.SGD(network.parameters(),
                            lr=0.01,   # learning rate
                            momentum=0.5)
accuracy_train = []
accuracy_test = []
for _ in range(10):
    continue_training()

f, pl1 = plt.subplots(1,1)
pl1.plot(np.arange(1,11),accuracy_train,label="Training")
pl1.plot(np.arange(1,11),accuracy_test,label="Testing")
pl1.set(title="(1d) Training and Testing Accuracies vs. Iterations for an Extra␣
 ↪ReLU Network",xlabel="Iteration",ylabel="Accuracy")
pl1.legend()
pl1.grid()
```

```
Iteration 1 Training accuracy: 0.4307500123977661
Iteration 1 Testing accuracy: 0.4194999933242798
Iteration 2 Training accuracy: 0.3412500023841858
Iteration 2 Testing accuracy: 0.33399999141693115
Iteration 3 Training accuracy: 0.2536250054836273
```

```
Iteration 3 Testing accuracy: 0.25600001215934753
Iteration 4 Training accuracy: 0.2536250054836273
Iteration 4 Testing accuracy: 0.25600001215934753
Iteration 5 Training accuracy: 0.2536250054836273
Iteration 5 Testing accuracy: 0.25600001215934753
Iteration 6 Training accuracy: 0.2536250054836273
Iteration 6 Testing accuracy: 0.25600001215934753
Iteration 7 Training accuracy: 0.2536250054836273
Iteration 7 Testing accuracy: 0.25600001215934753
Iteration 8 Training accuracy: 0.2536250054836273
Iteration 8 Testing accuracy: 0.25600001215934753
Iteration 9 Training accuracy: 0.2536250054836273
Iteration 9 Testing accuracy: 0.25600001215934753
Iteration 10 Training accuracy: 0.2536250054836273
Iteration 10 Testing accuracy: 0.25600001215934753
```



Once again, there is no significant performance improvement over (1a), (1b), or (1c).

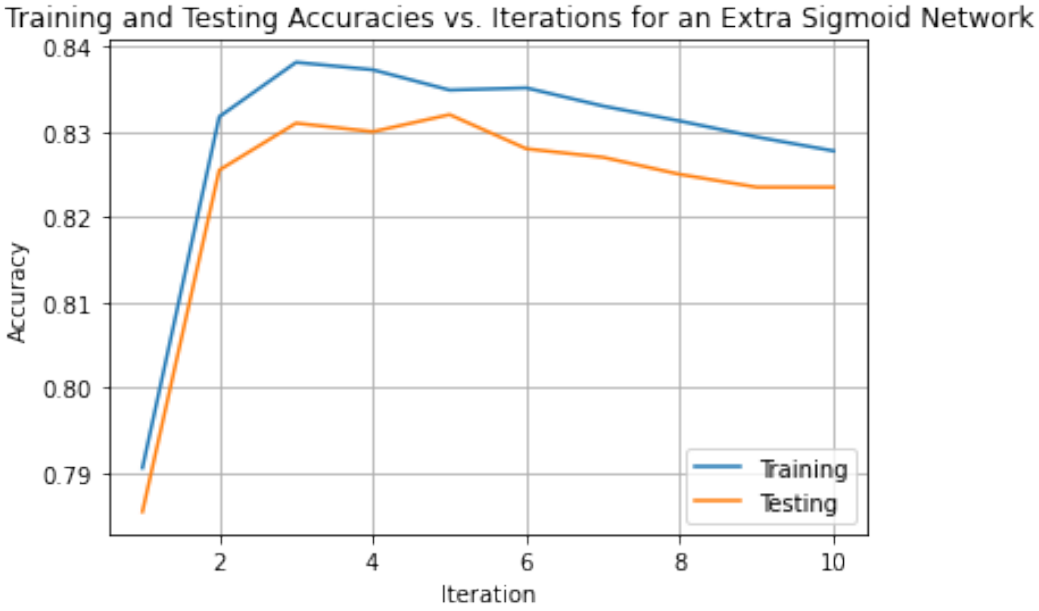
1. e) [1 mark] Repeat 1.d but use `F.sigmoid` instead of `F.relu` for the output. Plot the same graph as before. You should see that this change makes an improvement over the results in the previous questions. Why is this the case?

[9]:
```
class ExtraSigNet(nn.Module):
    def __init__(self):
        super(ExtraSigNet, self).__init__()
        self.fc1 = nn.Linear(4, 500)    # the weights from the input to the new␣
 ↪learned features (hidden layer)
```

```
        self.fc2 = nn.Linear(500, 4)     # the weights from the hidden layer to␣
 ↪the output

    def forward(self, x):
        # the processing the network will do
        x = F.relu(self.fc1(x))          # apply the first set of weights, then␣
 ↪apply the ReLU neuron model
        x = F.sigmoid(self.fc2(x))                  # apply the second set of␣
 ↪weights
        return x


network = ExtraSigNet()
optimizer = torch.optim.SGD(network.parameters(),
                            lr=0.02,   # learning rate
                            momentum=0.5)
accuracy_train = []
accuracy_test = []
for _ in range(10):
    continue_training()

f, pl1 = plt.subplots(1,1)
pl1.plot(np.arange(1,11),accuracy_train,label="Training")
pl1.plot(np.arange(1,11),accuracy_test,label="Testing")
pl1.set(title="(1e) Training and Testing Accuracies vs. Iterations for an Extra␣
 ↪Sigmoid Network",xlabel="Iteration",ylabel="Accuracy")
pl1.legend()
pl1.grid()
```

```
Iteration 1 Training accuracy: 0.7906249761581421
Iteration 1 Testing accuracy: 0.7854999899864197
Iteration 2 Training accuracy: 0.8317499756813049
Iteration 2 Testing accuracy: 0.8255000114440918
Iteration 3 Training accuracy: 0.8381249904632568
Iteration 3 Testing accuracy: 0.8309999704360962
Iteration 4 Training accuracy: 0.8372499942779541
Iteration 4 Testing accuracy: 0.8299999833106995
Iteration 5 Training accuracy: 0.8348749876022339
Iteration 5 Testing accuracy: 0.8320000171661377
Iteration 6 Training accuracy: 0.8351250290870667
Iteration 6 Testing accuracy: 0.828000009059906
Iteration 7 Training accuracy: 0.8330000042915344
Iteration 7 Testing accuracy: 0.8270000219345093
Iteration 8 Training accuracy: 0.831250011920929
Iteration 8 Testing accuracy: 0.824999988079071
Iteration 9 Training accuracy: 0.8293750286102295
Iteration 9 Testing accuracy: 0.8234999775886536
```

```
Iteration 10 Training accuracy: 0.827750027179718
Iteration 10 Testing accuracy: 0.8234999775886536
```



We DO see a significant performance improvement. The sigmoidal activation function scales the output of each output layer neuron to be within $[0, 1]$, which removes the potential increase in error for a more confident answer and reduces the potential error of incorrect answers. This more accurately matches the target values of 0 and 1 and enables better training and performance of the network. For instance, a confident selection of the 2nd class could output $[-1.38, 10.65, 0.65, 2.3]$, but a sigmoidal activation would scale all of these to be within the range $[0, 1]$, producing something like: $[0.06, 0.96, 0.09, 0.11]$ (note that these are not the actual values, just a demonstration of the scaling).

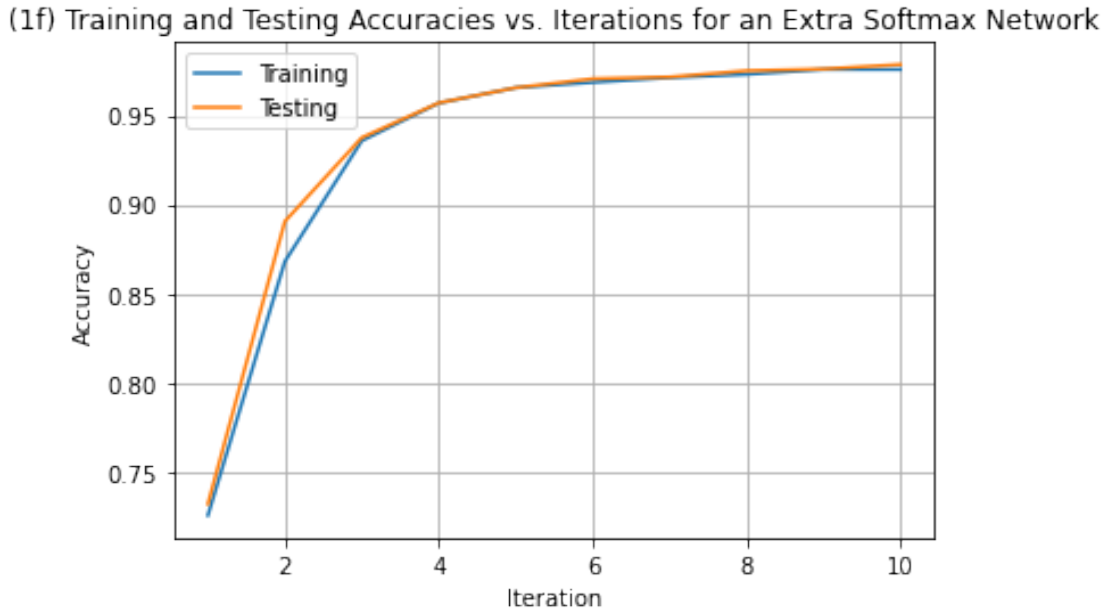
1. f) [1 mark] Try replacing the `F.sigmoid` with `F.softmax`. The softmax function will scale up the largest value while scaling down the smaller values. Repeat the previous question and plot the same graph. You should see an even bigger improvement. Why is this the case?

[10]:
```
class ExtraSoftNet(nn.Module):
    def __init__(self):
        super(ExtraSoftNet, self).__init__()
        self.fc1 = nn.Linear(4, 500)     # the weights from the input to the new␣
↪learned features (hidden layer)
        self.fc2 = nn.Linear(500, 4)     # the weights from the hidden layer to␣
↪the output

    def forward(self, x):
        # the processing the network will do
```

```
        x = F.relu(self.fc1(x))          # apply the first set of weights, then␣
 ↪apply the ReLU neuron model
        x = F.softmax(self.fc2(x))                  # apply the second set of␣
 ↪weights
        return x


network = ExtraSoftNet()
optimizer = torch.optim.SGD(network.parameters(),
                            lr=0.02,   # learning rate
                            momentum=0.5)
accuracy_train = []
accuracy_test = []
for _ in range(10):
    continue_training()

f, pl1 = plt.subplots(1,1)
pl1.plot(np.arange(1,11),accuracy_train,label="Training")
pl1.plot(np.arange(1,11),accuracy_test,label="Testing")
pl1.set(title="(1f) Training and Testing Accuracies vs. Iterations for an Extra␣
 ↪Softmax Network",xlabel="Iteration",ylabel="Accuracy")
pl1.legend()
pl1.grid()
```

```
/tmp/ipykernel_7890/1236685088.py:10: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument.
  x = F.softmax(self.fc2(x))                  # apply the second set of weights
```

```
Iteration 1 Training accuracy: 0.7256249785423279
Iteration 1 Testing accuracy: 0.7319999933242798
Iteration 2 Training accuracy: 0.8679999709129333
Iteration 2 Testing accuracy: 0.890500009059906
Iteration 3 Training accuracy: 0.9357500076293945
Iteration 3 Testing accuracy: 0.9375
Iteration 4 Training accuracy: 0.9568750262260437
Iteration 4 Testing accuracy: 0.9570000171661377
Iteration 5 Training accuracy: 0.965499997138977
Iteration 5 Testing accuracy: 0.965499997138977
Iteration 6 Training accuracy: 0.968500018119812
Iteration 6 Testing accuracy: 0.9704999923706055
Iteration 7 Training accuracy: 0.9711250066757202
Iteration 7 Testing accuracy: 0.9714999794960022
Iteration 8 Training accuracy: 0.9729999899864197
Iteration 8 Testing accuracy: 0.9750000238418579
Iteration 9 Training accuracy: 0.9757500290870667
Iteration 9 Testing accuracy: 0.9760000109672546
Iteration 10 Training accuracy: 0.9757500290870667
Iteration 10 Testing accuracy: 0.9785000085830688
```



This is because the softmax operator actively reduces the output of non-selected neurons and imposes a sort of winner-take-all operation that aims to make the classification operation more binary. This makes the classification and training more effective by providing model outputs that are more aligned with the targets.

1. g) [1 mark] Even though the model you built in 1f) does a good job of solving the task of producing the correct output, it seems to do this only if we include a `softmax` operation. How might this be considered "cheating" (and thus this should not considered to be a good model of action selection in the brain)?

Softmax imposes a sort of winner-take-all operation on the model outputs that biological neurons cannot perform equally. This could be considered cheating, as it isn't biologically viable in this form. A winner-takes-all system would require an entire other model to implement effectively that would need to be additionally trained and then deployed in conjunction with the other model in place of the softmax operation.

3 2. Action Selection with Winner-Take-All

Given the results of the previous section, we need a different approach to do action selection. Since a feed-forward approach led to difficulties, here we will explore a recurrent network: the standard "winner-take-all" circuit. In this system, each neuron excites itself but inhibits the other neurons.

Crucially, since this approach is recurrent, we can't just feed in an input and get an output. Instead, we will feed in an input over time and see what the output is over time. Rather than implement this in `pytorch`, we will just use normal numpy commands.

The main configurable part of the model is how nuch the neurons excite themselves and how much they inhibit other neurons. We can build this as a weight matrix that has the excitation on the diagonal and the inhibition everywhere else:

[11]:
```
w = np.zeros((N,N))
w[:] = -0.9   # set the inhibition amount everywhere
for i in range(N):
    w[i,i] = 1   # set the excitatory connection
print(w)
```

```
[[ 1.  -0.9 -0.9 -0.9]
 [-0.9  1.  -0.9 -0.9]
 [-0.9 -0.9  1.  -0.9]
 [-0.9 -0.9 -0.9  1. ]]
```

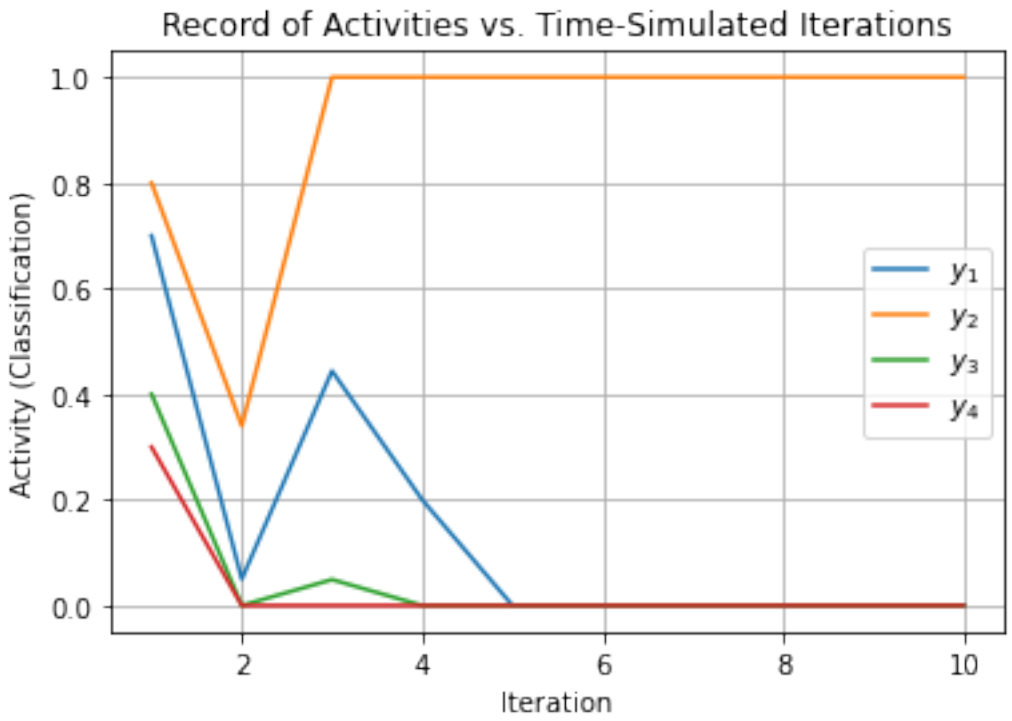
To run the model, we pick a particular input that set that we will provide. We then feed that value into the model, and add the value that the neurons send back to themselves. We also include a `clip` operation that stops the neuron value from going below 0 or above 1. We repeat this process multiple times, and we record the activity of the neurons so we can plot it afterwards.

[12]:
```
record = []                    # for recording the activity values
activity = np.zeros(N)         # neurons are not active initially
input_values = [0.7, 0.8, 0.4, 0.3]
for i in range(10):
    # the new neuron value is the old value times the weight matrix (for the␣
 ↪neurons
    #  inhibiting and exciting each other), plus the input benig added.
    # we also clip the neurons so their activity doesn't go below 0 or above 1
    activity = np.clip(input_values + w @ activity, 0, 1)
    record.append(activity)
```

2. a) [1 mark] Run the model with an input of `[0.7, 0.8, 0.4, 0.3]`. Plot the `record` of the activity. Print the final `activity` value. Does the network successfully output the desired correct results of `[0, 1, 0, 0]`?

[13]:
```
f, pl1 = plt.subplots(1,1)
record = np.array(record)
pl1.plot(np.arange(1,11),record[:,0],label="$y_{1}$");
pl1.plot(np.arange(1,11),record[:,1],label="$y_{2}$");
pl1.plot(np.arange(1,11),record[:,2],label="$y_{3}$");
pl1.plot(np.arange(1,11),record[:,3],label="$y_{4}$");
pl1.set(title="Record of Activities vs. Time-Simulated␣
 ↪Iterations",xlabel="Iteration",ylabel="Activity (Classification)")
pl1.legend()
pl1.grid()

print(record[-1,:])
```

```
[0. 1. 0. 0.]
```



The network *does* output the correct desired result of $[0, 1, 0, 0]$.

2. b) [1 mark] Test the model using all the data in the `X` dataset you created in question 1. (This should be all 10000 sets of 4 randomly generated numbers). Note that since we didn't use any data to train the model, we are using all of it to test the model.

To test the model, loop through all the items (`for input_values in X:`) and run the model as in question 2a. To determine if the output is correct, you can do

```
target = np.zeros(N)
target[np.argmax(input_values)] = 1
if np.all(target == activity):
    correct += 1
```

What proportion of the time does the model produce the correct output?

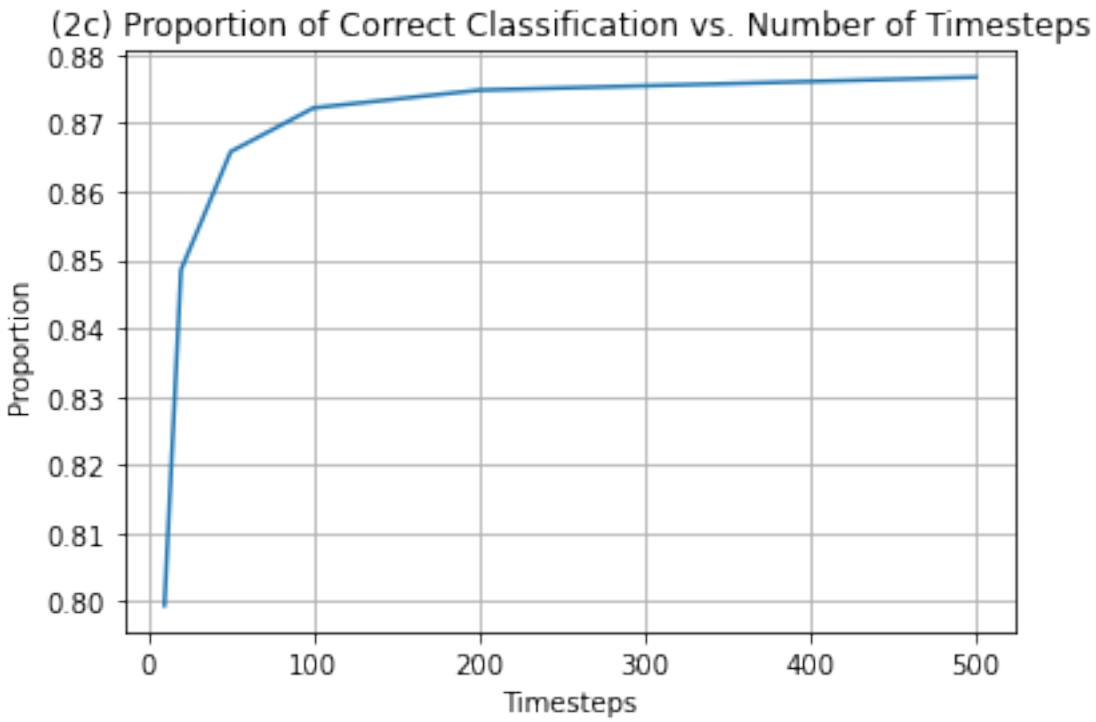
[14]:
```
correct = 0
total = 0
for input_values in X:
    activity = np.zeros(4)
    for i in range(10):
        activity = np.clip(input_values + w @ activity, 0, 1)
    target = np.zeros_like(activity)
    target[np.argmax(input_values)] = 1
    if np.all(target == activity):
        correct += 1
```

```
    total += 1
print(f"TOTAL DATAPOINTS: {total}\nTOTAL CORRECT: {correct}\n\tPROPORTION␣
 ↪CORRECT: {correct/total}")
```

```
TOTAL DATAPOINTS: 10000
TOTAL CORRECT: 7994
        PROPORTION CORRECT: 0.7994
```

2. c) [1 mark] Try running the model for a longer time, rather than the 10 steps in the code provided. Try 10, 20, 50, 100, 200, and 500. Make a plot showing the proportion of time the model produces the ideal output for each of those numbers of timesteps.

[15]:
```
n_set = [10, 20, 50, 100, 200, 500]
results = []
for n in n_set:
    correct = 0
    total = 0
    for input_values in X:
        activity = np.zeros(4)
        for i in range(n):
            activity = np.clip(input_values + w @ activity, 0, 1)
        target = np.zeros_like(activity)
        target[np.argmax(input_values)] = 1
        if np.all(target == activity):
            correct += 1
        total += 1
    results.append(correct/total)

f, pl1 = plt.subplots(1,1)
pl1.plot(n_set, results)
pl1.set(title="(2c) Proportion of Correct Classification vs. Number of␣
 ↪Timesteps",xlabel="Timesteps",ylabel="Proportion")
pl1.grid()
```



2. d) [2 marks] Try improving the model by adjusting the amount of inhibition (-0.9) and the amount of excitation (1). Can you improve the proportion of time the model produces the correction output? To simplify your experimentation, keep the number of timesteps as 20.

[16]:
```
inh = np.arange(-1.3,-0.8,0.1)
exc = np.arange(0.8,1.3,0.1)
cx, cy = np.meshgrid(inh,exc)
cx = cx.ravel()
cy = cy.ravel()

results = []

for c_i in range(cx.shape[0]):
    w = np.zeros((N,N))
    w[:] = cx[c_i]   # set the inhibition amount everywhere
    for i in range(N):
        w[i,i] = cy[c_i]   # set the excitatory connection
    correct = 0
    total = 0
    for input_values in X:
        activity = np.zeros(4)
        for i in range(20):
            activity = np.clip(input_values + w @ activity, 0, 1)
        target = np.zeros_like(activity)
```

```
        target[np.argmax(input_values)] = 1
        if np.all(target == activity):
            correct += 1
        total += 1
    results.append([cx[c_i], cy[c_i], correct/total])
    print(f"{cx[c_i]}, {cy[c_i]}, {correct/total}, {len(results)}/{(cx.
↪shape[0])}")
```

```
-1.3, 0.8, 0.4201, 1/25
-1.2, 0.8, 0.5004, 2/25
-1.0999999999999999, 0.8, 0.6016, 3/25
-0.9999999999999998, 0.8, 0.713, 4/25
-0.8999999999999997, 0.8, 0.7998, 5/25
-1.3, 0.9, 0.4755, 6/25
-1.2, 0.9, 0.5623, 7/25
-1.0999999999999999, 0.9, 0.6643, 8/25
-0.9999999999999998, 0.9, 0.7786, 9/25
-0.8999999999999997, 0.9, 0.836, 10/25
-1.3, 1.0, 0.5289, 11/25
-1.2, 1.0, 0.6225, 12/25
-1.0999999999999999, 1.0, 0.7272, 13/25
-0.9999999999999998, 1.0, 0.829, 14/25
-0.8999999999999997, 1.0, 0.8485, 15/25
-1.3, 1.1, 0.5876, 16/25
-1.2, 1.1, 0.6817, 17/25
-1.0999999999999999, 1.1, 0.7838, 18/25
-0.9999999999999998, 1.1, 0.8505, 19/25
-0.8999999999999997, 1.1, 0.8329, 20/25
-1.3, 1.2, 0.6389, 21/25
-1.2, 1.2, 0.7375, 22/25
-1.0999999999999999, 1.2, 0.8225, 23/25
-0.9999999999999998, 1.2, 0.8491, 24/25
-0.8999999999999997, 1.2, 0.7917, 25/25
```

Some local exploring around the provided inhibition and excitation indicate that it is near a correctness maximum. I can also explore around the point more minutely.

[17]:
```
inh = np.arange(-0.95,-0.80,0.025)
exc = np.arange(0.95,1.05,0.025)
cx, cy = np.meshgrid(inh,exc)
cx = cx.ravel()
cy = cy.ravel()

results = []

for c_i in range(cx.shape[0]):
    w = np.zeros((N,N))
    w[:] = cx[c_i]   # set the inhibition amount everywhere
```

```
    for i in range(N):
        w[i,i] = cy[c_i]    # set the excitatory connection
    correct = 0
    total = 0
    for input_values in X:
        activity = np.zeros(N)
        for i in range(20):
            activity = np.clip(input_values + w @ activity, 0, 1)
        target = np.zeros_like(activity)
        target[np.argmax(input_values)] = 1
        if np.all(target == activity):
            correct += 1
        total += 1
    results.append([cx[c_i], cy[c_i], correct/total])
    print(f"{cx[c_i]}, {cy[c_i]}, {correct/total}, {len(results)}/{(cx.
↪shape[0])}")
```

```
-0.95, 0.95, 0.8394, 1/30
-0.9249999999999999, 0.95, 0.8449, 2/30
-0.8999999999999999, 0.95, 0.8439, 3/30
-0.8749999999999999, 0.95, 0.838, 4/30
-0.8499999999999999, 0.95, 0.8225, 5/30
-0.8249999999999998, 0.95, 0.804, 6/30
-0.95, 0.975, 0.847, 7/30
-0.9249999999999999, 0.975, 0.8491, 8/30
-0.8999999999999999, 0.975, 0.8466, 9/30
-0.8749999999999999, 0.975, 0.8377, 10/30
-0.8499999999999999, 0.975, 0.8226, 11/30
-0.8249999999999998, 0.975, 0.801, 12/30
-0.95, 1.0, 0.8501, 13/30
-0.9249999999999999, 1.0, 0.8515, 14/30
-0.8999999999999999, 1.0, 0.8485, 15/30
-0.8749999999999999, 1.0, 0.8346, 16/30
-0.8499999999999999, 1.0, 0.8203, 17/30
-0.8249999999999998, 1.0, 0.7948, 18/30
-0.95, 1.025, 0.8529, 19/30
-0.9249999999999999, 1.025, 0.8528, 20/30
-0.8999999999999999, 1.025, 0.8468, 21/30
-0.8749999999999999, 1.025, 0.8328, 22/30
-0.8499999999999999, 1.025, 0.8123, 23/30
-0.8249999999999998, 1.025, 0.7869, 24/30
-0.95, 1.05, 0.8544, 25/30
-0.9249999999999999, 1.05, 0.8534, 26/30
-0.8999999999999999, 1.05, 0.8426, 27/30
-0.8749999999999999, 1.05, 0.829, 28/30
-0.8499999999999999, 1.05, 0.8063, 29/30
-0.8249999999999998, 1.05, 0.777, 30/30
```

It appears that the best actually occurs closer to an inhibition value of -0.95 and an excitation value of 1.05, so we certainly *can* improve the correctness proportion by modifying the two values.

4 3. Action Selection with Winner-Take-All and a Synapse

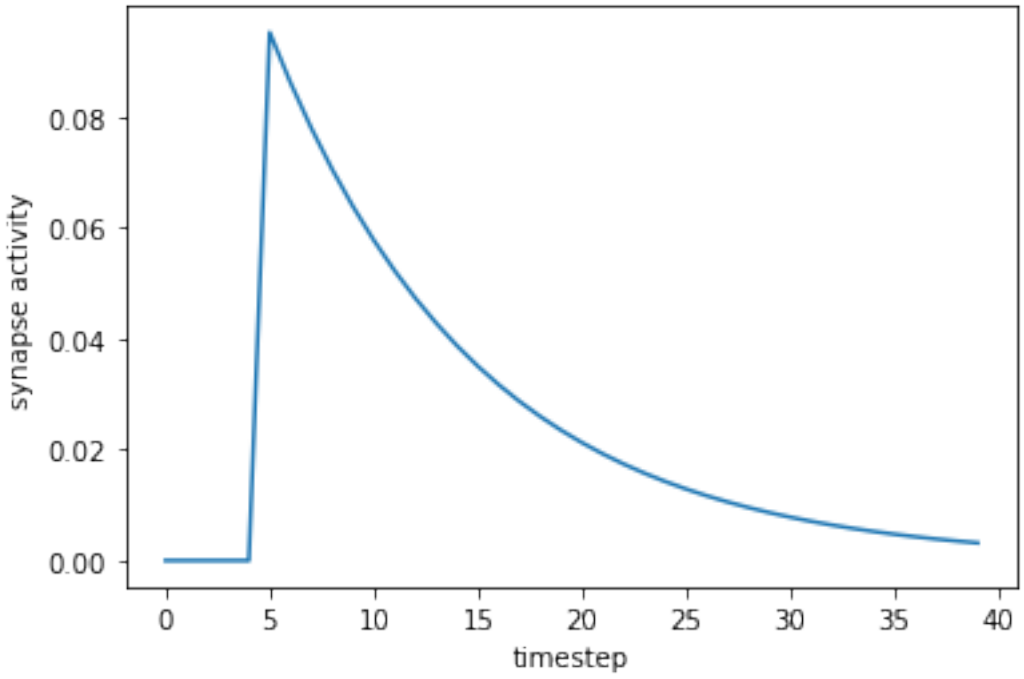
We are now going to add a biological detail to the Winner-Take-All system. In biology, when there is a connection between neurons, there is a synapse, and the activity of that synapse decays slowly over time. This means that when a spike occurs, the input into the next neuron will suddenly increase, and then slowly decrease.

Here we will make a simple approximation of this process. We will be implementing this as a *low-pass filter*. The only parameter is the one that controls how quickly the synaptic activity will decay, and we will call this `tau`. To implement a low-pass filter as part of our neuron model, we can compute the synaptic activity as follows, where `dt` is how long one timestep is in our model:

```
alpha = 1-np.exp(-dt/tau)

activity_syn = activity_syn + alpha * (activity - activity_syn)
```

Here is what that would look like for a single input that gave a single spike on the 5th timestep, where each timestep is 0.001 seconds and the synaptic time constant is 0.01 seconds (10 milliseconds):

[18]:
```
activity = np.zeros(1)
activity_syn = np.zeros(1)

dt = 0.001    # length of a timestep
tau = 0.010   # time constant for the synapse
alpha = 1-np.exp(-dt/tau)
record = []
for i in range(40):
    activity = 0
    if i == 5:
        activity = 1
    activity_syn = activity_syn + alpha * (activity - activity_syn)
    record.append(activity_syn)

import matplotlib.pyplot as plt
plt.plot(record)
plt.xlabel('timestep')
plt.ylabel('synapse activity')
plt.show()
```



While our winner-take-all model is not spiking, we can still add this synapse model to the system. This would be done as follows:

[19]:
```
record = []
activity = np.zeros(N)          # the raw output from the neuron
activity_syn = np.zeros(N)      # the output from the synapse
for i in range(20):
    activity = np.clip(input_values + w @ activity_syn, 0, 1)  # note we use␣
 ↪activity_syn here, not activity
    activity_syn += alpha * (activity - activity_syn)          # implement the␣
 ↪synapse
    record.append(activity_syn)
```

3. a) [1 mark] Test the winner-take-all model with a synapse time constant of 10 milliseconds, inhibition of -0.9, and excitation of 1 against all the data in `X`, running it for 20 timesteps (with each timestep being 1 millisecond). What proportion of time does it get the correct answer? Repeat this for differet synapse time constants (try 0.001, 0.002, 0.005, 0.01, and 0.02 seconds) and generate a plot showing the accuracy as you change the time constant.

[20]:
```
activity = np.zeros(N)
activity_syn = np.zeros(N)

dt = 0.001    # length of a timestep
tau = 0.010   # time constant for the synapse
alpha = 1-np.exp(-dt/tau)
```

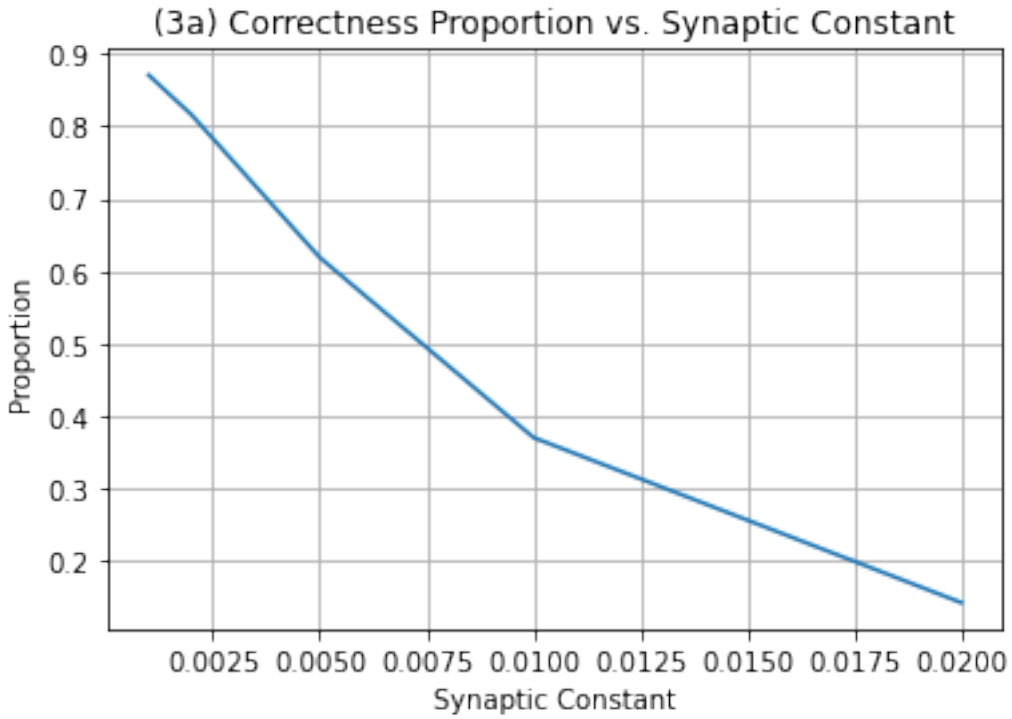
```
w = np.zeros((N,N))
w[:] = -0.9   # set the inhibition amount everywhere
for i in range(N):
    w[i,i] = 1.0   # set the excitatory connection

correct = 0
total = 0
for input_values in X:
    activity = np.zeros(N)
    for i in range(20):
        activity = np.clip(input_values + w @ activity_syn, 0, 1)
        activity_syn += alpha * (activity - activity_syn)
    target = np.zeros_like(activity)
    target[np.argmax(input_values)] = 1
    if np.all(target == activity):
        correct += 1
    total += 1
print(correct/total)
```

```
0.2389
```

This initial setup gives us an approximate 0.24 proportion of correctness.

[21]:
```
tau_set = [0.001, 0.002, 0.005, 0.01, 0.02]
results = []

for tau in tau_set:
    alpha = 1-np.exp(-dt/tau)
    correct = 0
    total = 0
    for input_values in X:
        activity = np.zeros(N)
        activity_syn = np.zeros(N)
        for i in range(20):
            activity = np.clip(input_values + w @ activity_syn, 0, 1)
            activity_syn += alpha * (activity - activity_syn)
        target = np.zeros_like(activity)
        target[np.argmax(input_values)] = 1
        if np.all(target == activity):
            correct += 1
        total += 1
    results.append(correct/total)

f, pl1 = plt.subplots(1,1)
pl1.plot(tau_set, results)
pl1.set(title="(3a) Correctness Proportion vs. Synaptic␣
 ↪Constant",xlabel="Synaptic Constant",ylabel="Proportion")
```

```
pl1.grid()
```



3. b) [2 marks] Try improving the model by adjusting the synaptic time constant, the amount of inhibition and the amount of excitation. Can you improve the proportion of time the model produces the correct output? To simplify your experimentation, keep the number of timesteps as 20. As a hint, you should be able to get the model close to 99% accuracy! This is an example of adding a biological detail to a model and having it improve its performance.

[22]:
```
w = np.zeros((N,N))
w[:] = -1.13
for i in range(N):
    w[i,i] = 0.91

alpha = 1-np.exp(-dt/0.001)

correct = 0
total = 0
for input_values in X:
    activity = np.zeros(N)
    activity_syn = np.zeros(N)
    for i in range(20):
        activity = np.clip(input_values + w @ activity_syn, 0, 1)
        activity_syn += alpha * (activity - activity_syn)
    target = np.zeros_like(activity)
```

```
    target[np.argmax(input_values)] = 1
    if np.all(target == activity):
        correct += 1
    total += 1
print(correct/total)
```

```
0.999
```

Tuned by hand (iterative hill-climbing by increments of 0.01), the best values I could determine for the inhibitory and excitatory weights were -1.13 and 0.91 respectively while also utilizaing a synaptic time constant of 1ms – the best result from the previous section. This resulted in 99.9% correctness for the classifier employing some biological detail – far better than the model with no biological detail.

[ ]: