Accuracy using Random Calssifier:

When random classifier is used for the calssification problem involving 5 output label, the probability of choosing correct label out of the 5 labels is (1/5) = 20%

So the accuracy will be almost equal to 20%.

Accuracy using Majority Classifier:

When the majority calssifier is used, the majority label in the traning set is applied to all cross validation data set. So, the only the examples having the majority label will be correct using the Mojority classification model. Here I'm considering the same split in data as Project part 1 (Training: 40000, Cross validation: 5000, Test: 5000)

In the given dataset, the training split of the data has majority label of 2.

Once this label is predicted for all validation set data, the accuracy = 1022/5000 = 0.20 = 20%

In [906]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F

#Prepare the data.
import torch
import numpy as np
from torch.autograd import Variable
import torch.nn as nn
from numpy import linalg as LA

dtype = torch.FloatTensor

images = np.load("./data/images.npy")
labels = np.load("./data/labels.npy")

images = np.reshape(images, (images.shape[0], images.shape[1] * image
s.shape[2]))

images = images - images.mean()
images = images/images.std()

train_seqs = images[0:40000]
val_seqs = images[40000:50000]

train_labels = labels[0:40000]
cv_labels = labels[40000:50000]
```

In [907]:
```
HEIGHT, WIDTH, NUM_CLASSES, NUM_OPT_STEPS = 26, 26, 5, 5000
learning_rate = 0.001

class LinearModel(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(LinearModel, self).__init__()
        self.Linear = torch.nn.Linear(D_in, D_out)

    def forward(self, x):
        y_pred = self.Linear(x)
        return y_pred
```

torch.Linear Parameters:

```
Number of parameters = total weights + biases = 676 * 5 + 5= 3385
```

In [908]:
```
model = LinearModel(HEIGHT * WIDTH, NUM_CLASSES)
```

torch.optim:

In the previous homeworks, we have used, Stochastic Gradient Descent (SGD) and Adam variant of SGD.

Parameters required for ADAM:

1. learning rate
2. betas(to reduce the learning rate as we approach convergence) ,
3. eps(epsilon: to make sure the denominator doesnt go to zero) and
4. weight_decay

Parameters required for SGD:

1. learning rate: rate at which weights are updated
2. weight decay: to include l2 regulariser in the update
3. Momentum

In [909]:
```
optimizer = torch.optim.Adam(model.parameters(), lr= learning_rate)
```

In [910]:
```
def train(batch_size):
    model.train()

    i = np.random.choice(train_seqs.shape[0], size = batch_size, repl
ace=False)
    x = Variable(torch.from_numpy(train_seqs[i].astype(np.float32)))
    y = Variable(torch.from_numpy(train_labels[i].astype(np.int)))

    optimizer.zero_grad()
    y_hat = model(x)
    loss = F.cross_entropy(y_hat, y)
    loss.backward()
    optimizer.step()

    return loss.data[0]
```

In [911]:
```
def accuracy(y, y_hat):
    count = 0
    for i in range(y.shape[0]):
        if y[i] == y_hat[i]:
            count += 1
    return count/y.shape[0]
```

In [912]:
```
import random
def approx_train_accuracy():
    i = np.random.choice(train_seqs.shape[0], size = 1000, replace=Fa
lse)
    x = train_seqs[i].astype(np.float32)
    y = train_labels[i].astype(np.int)
    y_hat = np.empty(1000)

    index = 1
    for param in model.parameters():
        if (index%2 != 0):
            weights = param.data.numpy()
            index +=1
        else:
            bias = param.data.numpy()

    for i in range(1000):
        y_pred = x[i].dot(weights.transpose()) + bias
        res = np.argmax(y_pred)
        y_hat[i] = res
    acc = accuracy(y,y_hat)
    return acc


def val_accuracy():
    y_hat = np.empty(1000)

    i = np.random.choice(val_seqs.shape[0], size = 1000, replace=Fals
e)
    x = val_seqs[i].astype(np.float32)
    y = cv_labels[i].astype(np.int)

    index = 1
    for param in model.parameters():
        if (index%2 != 0):
            weights = param.data.numpy()
            index +=1
        else:
            bias = param.data.numpy()
    for i in range(1000):
        y_pred = x[i].dot(weights.transpose()) + bias
        res = np.argmax(y_pred)
        y_hat[i] = res
    acc = accuracy(y,y_hat)
    return acc
```

In [913]:
```
train_accs, val_accs = [], []
batch_size = 500
for i in range(5000):
    l = train(batch_size)
    if i % 100 == 0:
        train_accs.append(approx_train_accuracy())
        val_accs.append(val_accuracy())
        print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
```

```
   0  0.23  0.26
 100  0.80  0.77
 200  0.79  0.78
 300  0.79  0.77
 400  0.79  0.77
 500  0.80  0.78
 600  0.77  0.77
 700  0.77  0.76
 800  0.80  0.78
 900  0.82  0.81
1000  0.80  0.79
1100  0.78  0.77
1200  0.81  0.77
1300  0.79  0.78
1400  0.78  0.77
1500  0.79  0.77
1600  0.79  0.80
1700  0.78  0.79
1800  0.77  0.78
1900  0.78  0.79
2000  0.81  0.77
2100  0.78  0.77
2200  0.78  0.78
2300  0.79  0.80
2400  0.81  0.79
2500  0.79  0.77
2600  0.80  0.79
2700  0.80  0.78
2800  0.81  0.77
2900  0.79  0.80
3000  0.77  0.77
3100  0.80  0.76
3200  0.80  0.78
3300  0.80  0.79
3400  0.80  0.81
3500  0.79  0.78
3600  0.79  0.77
3700  0.80  0.79
3800  0.79  0.78
3900  0.80  0.78
4000  0.76  0.78
4100  0.81  0.77
4200  0.78  0.78
4300  0.80  0.77
4400  0.79  0.76
4500  0.80  0.76
4600  0.79  0.78
4700  0.79  0.77
4800  0.79  0.80
4900  0.80  0.78
```

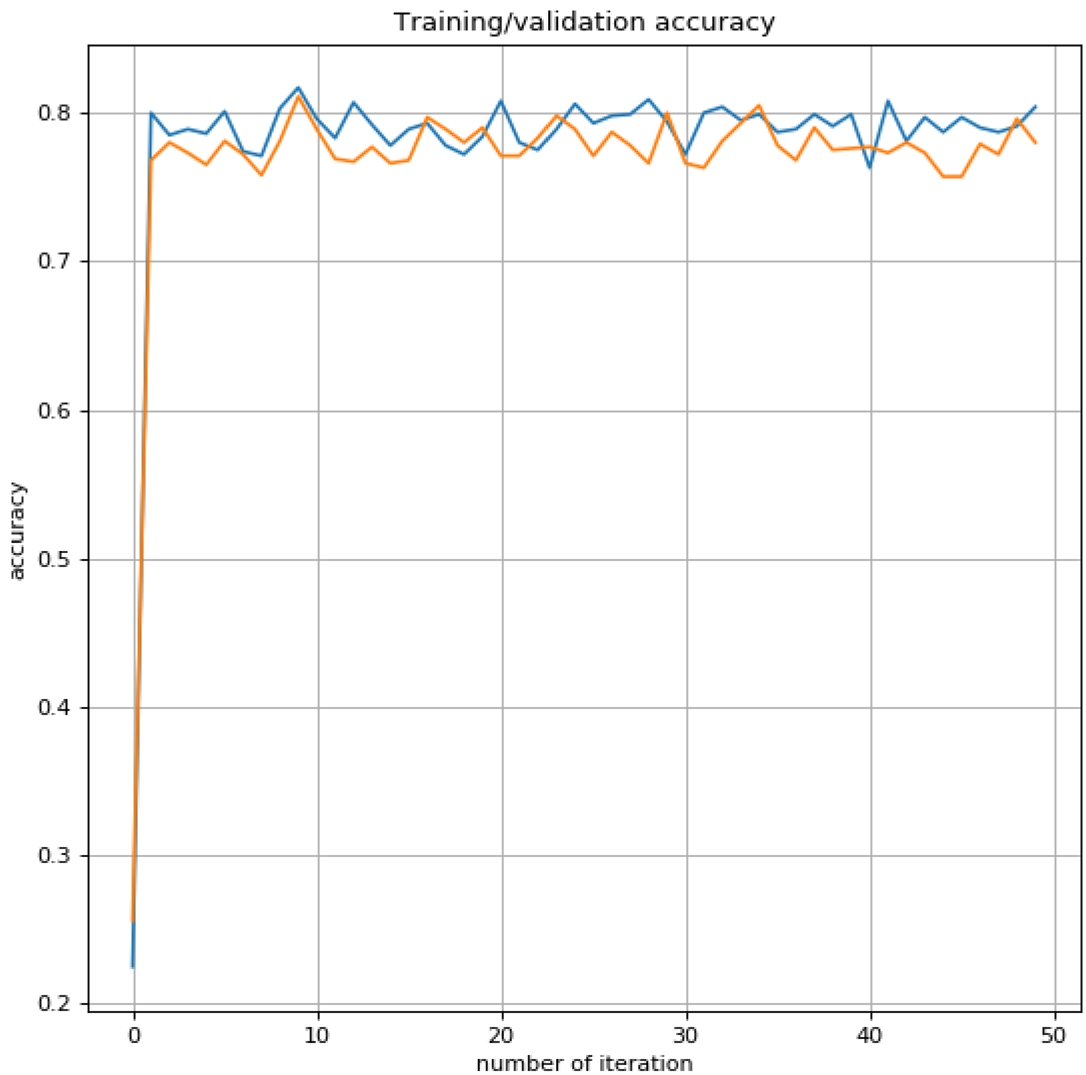
In [914]:
```
import matplotlib.pyplot as plt


t = np.arange(0,len(train_accs),1)

s = train_accs
k = val_accs
print("max_train accuracy: ", max(train_accs))
print("max_val accuracy: ", max(val_accs))
plt.figure(figsize=(8,8), dpi = 80)
plt.plot(t, s, t, k)

plt.xlabel('number of iteration')
plt.ylabel('accuracy')
plt.title('Training/validation accuracy')
plt.grid(True)
plt.show()
```

```
max_train accuracy:  0.817
max_val accuracy:  0.811
```



Accuracies with batch size =1 and 5000 steps:

With batch size =1, Optimization steps = 5000, using SGD and learning rate=1e-6, the highest accuracy reached is around 24%.

Reason for low accuracy: The model is underfitting here.

The learning is too less for the loss function to converge. There is significantly high loss in the predictions and weights are not optimal. So, the accuracy is very less.

Optimizations:

Highest Accuracy Achieved: 82%

Learning rate = 0.001 Optimiser = Adam Batch size = 5000