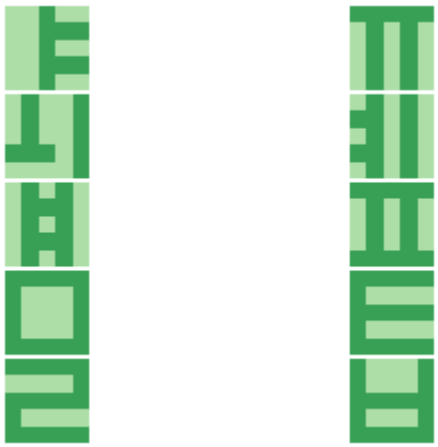
First test to simulate figure 3 of In-sensor reservoir computing for language learning via two-dimensional memristors

```
import numpy as np
import matplotlib.pyplot as plt
from glob import glob
from collections import Counter

filt=lambda x:'_' in x
get_letter=lambda x:x.split('.')[0]
#for i in range(10):
letters = list(filter(filt,glob('*.npy')))

letters= ['l0_ya.npy',  'l1_yu.npy',  'l2_oi.npy', 'l3_yoi.npy', 'l4_yai.npy', 'l5_p.npy',
 'l6_m.npy', 'l7_t.npy', 'l8_r.npy', 'l9_b.npy',
 'letsbuy.npy', 'letsgo.npy', 'letsride.npy', 'kick.npy','getout.npy']

letters = letters[:10]
ls = list(map(np.load, letters)) # letters[:10], letters[10:]
d = dict(zip(list(map(get_letter,letters)),ls))
d
```


```
{'l0_ya': array([[0., 0., 1., 0., 0.],
        [0., 0., 1., 1., 1.],
        [0., 0., 1., 0., 0.],
        [0., 0., 1., 1., 1.],
        [0., 0., 1., 0., 0.]]), 'l1_yu': array([[1., 1., 1., 1., 1.],
        [0., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.]]), 'l2_oi': array([[0., 1., 0., 0., 1.],
        [0., 1., 0., 0., 1.],
        [0., 1., 0., 0., 1.],
        [1., 1., 1., 0., 1.],
        [0., 0., 0., 0., 1.]]), 'l3_yoi': array([[0., 1., 0., 1., 0.],
        [1., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.],
        [1., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.]]), 'l4_yai': array([[0., 1., 0., 1., 0.],
        [0., 1., 1., 1., 0.],
        [0., 1., 0., 1., 0.],
        [0., 1., 1., 1., 0.],

        [0., 1., 0., 1., 0.]]), 'l5_p': array([[1., 1., 1., 1., 1.],
        [0., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.],
        [0., 1., 0., 1., 0.],
```

```
[1., 1., 1., 1., 1.]]), 'l6_m': array([[1., 1., 1., 1., 1.],
[1., 0., 0., 0., 1.],
[1., 0., 0., 0., 1.],
[1., 0., 0., 0., 1.],
[1., 1., 1., 1., 1.]]), 'l7_t': array([[1., 1., 1., 1., 1.],
[1., 0., 0., 0., 0.],
[1., 1., 1., 1., 1.],
[1., 0., 0., 0., 0.],
[1., 1., 1., 1., 1.]]), 'l8_r': array([[1., 1., 1., 1., 1.],
[0., 0., 0., 0., 1.],
[1., 1., 1., 1., 1.],
[1., 0., 0., 0., 0.],
[1., 1., 1., 1., 1.]]), 'l9_b': array([[1., 0., 0., 0., 1.],
[1., 0., 0., 0., 1.],
[1., 1., 1., 1., 1.],
[1., 0., 0., 0., 1.],
[1., 1., 1., 1., 1.]])}
```

```
letters = [ 'l0_ya.npy', 'l1_yu.npy', 'l2_oi.npy', 'l3_yoi.npy', 'l4_yai.npy',
 'l5_p.npy', 'l6_m.npy', 'l7_t.npy', 'l8_r.npy', 'l9_b.npy']
fig, ax = plt.subplots(len(letters)//2,2)
ax = [a for ae in ax for a in ae]
for i, lett in enumerate(letters):
    ax[i].imshow(np.load(lett), cmap=plt.cm.Greens, clim=[-1,2])
    ax[i].axis('off')

plt.subplots_adjust(wspace=0.05,hspace=0.05)
```

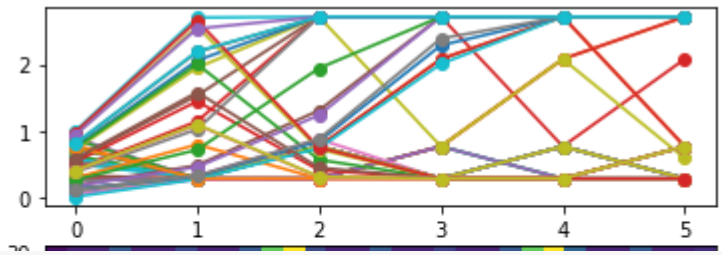


```
import warnings
warnings.filterwarnings('ignore')

fig, ax = plt.subplots(2,1)

t = np.arange(6)
# plt.plot(t,  np.cumsum(1/150*np.exp(-t)),'-o')

inp =  np.array([1., 0., 1., 0., 1.])
```

```
def output_row(initial_state, input_signal):
    """
    This function emulates the behavior of a memristor PoC.
    ┼nsert here a more detailed model or integrate a differential equation
    for better behavior.
    Ideally the experimental data should be here.

    Parameters
    ----------
    initial_state : conductance value before any pulse.
    input_signal : signal of the applied pulses.
    Returns
    -------
    a : the output of the conductance. this has to be extracted of the device.
    """
    a = [initial_state]

    for i in range(5):
        if input_signal[i] > 0 :
            a.append(np.clip(a[i],0.1,1)*np.exp(1))
        else:
            a.append(np.clip(a[i],1,10)*(3-np.exp(1)))
    return a

matrix = np.zeros((10,31))
nr=0
for nl, lett in enumerate(d.keys()):
    print (nl,nr)
    for nr, row in enumerate(d[lett]):
        initial_state = np.random.random(1)
        output = output_row(initial_state,row)
        ax[0].plot(output+np.random.random(1)*1e-4,'-o')
        matrix[nl,nr*6:(nr+1)*6] = output

matrix[:,30] = 2.5* np.random.random((10,))

# plt.figure()
ax[1].imshow(matrix, extent= [10,1,1,31],aspect='auto')
```

```
0 0
1 4
2 4
```

```
3 4
4 4
5 4
6 4
7 4
8 4
9 4
<matplotlib.image.AxesImage at 0x7f25ecc24310>
```




```
"""
one hot encoding, softmax function activation and training procedure
"""

def one_hot(y, c):
    """
    # y--> label/ground truth.
    # c--> Number of classes.
    """

    # A zero matrix of size (m, c)
    y_hot = np.zeros((len(y), c))

    # Putting 1 for column where the label is,
    # Using multidimensional indexing.
    y_hot[np.arange(len(y)), y] = 1

    return y_hot


one_hot([5,2,3], 10)

def softmax(z):
    # z--> linear part.

    # subtracting the max of z for numerical stability.
    exp = np.exp(z - np.max(z))

    # Calculating softmax for all example letters.
    for i in range(len(z)):
        exp[i] /= np.sum(exp[i])

    return exp

# fit(X,np.arange(10),0.1,10,100)
def fit(X, y, lr, c, epochs):
    """
```

```
    # X --> Input.
    # y --> true/target value.
    # lr --> Learning rate.
    # c --> Number of classes.
    # epochs --> Number of iterations.
    """

    # m-> number of training examples
    # n-> number of features
    m, n = X.shape

    # Initializing weights and bias randomly.
    w = np.random.random((n, c))
    b = np.random.random(c)
    # Empty list to store losses.
    losses = []

    np.save('initial_w.npy',w)
    # Training loop.
    for epoch in range(epochs):

        # Calculating hypothesis/prediction.
        z = X@w + b
        y_hat = softmax(z)

        # One-hot encoding y.
        y_hot = one_hot(y, c)

        # Calculating the gradient of loss w.r.t w and b.
        w_grad = (1/m)*np.dot(X.T, (y_hat - y_hot))
        b_grad = (1/m)*np.sum(y_hat - y_hot)

        # Updating the parameters.
        w = w - lr*w_grad
        b = b - lr*b_grad

        np.save('w' + str(epoch) + '.npy',w)
        # Calculating loss and appending it in the list.
        loss = -np.mean(np.log(y_hat[np.arange(len(y)), y]))
        losses.append(loss)
        # Printing out the loss at every 100th iteration.
        if epoch%1==0:
            print('Epoch {epoch}==> Loss = {loss}'
                  .format(epoch=epoch, loss=loss))
    return w, b, losses
```

```
"""  10 examples for the training, each example has 6*5 features, or 30 virtual reservoir noc
"""

X = np.zeros((10,30))
```

```
X = np.zeros((10,30))
for i, letter in enumerate(d.keys()):
    initial_state = np.random.random(1)
    output = []
    for row in d[letter]:
        output.append(output_row(initial_state,row))
    X[i,:] = np.concatenate(output)[:,0]


w, b, losses = fit(X,np.arange(10),0.1,10,100)

plt.figure()
plt.plot(losses)
```

```
Epoch 0==> Loss = 3.5213475677678057
Epoch 1==> Loss = 2.9720440287922787
Epoch 2==> Loss = 2.6232867411860665
```

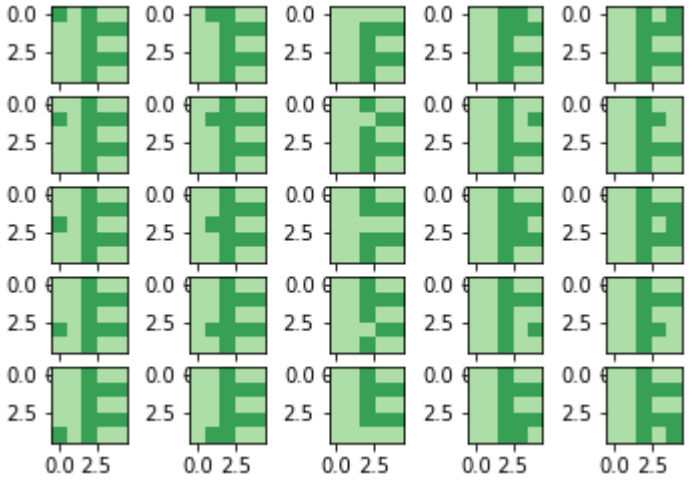
```
Epoch 3==> Loss = 2.362738891209903
Epoch 4==> Loss = 2.1519414413470654
Epoch 5==> Loss = 1.979704575495553
Epoch 6==> Loss = 1.8367285790049586
Epoch 7==> Loss = 1.7136239426882454
Epoch 8==> Loss = 1.6042674114293276
Epoch 9==> Loss = 1.5055095186528382
Epoch 10==> Loss = 1.4156131351217367
Epoch 11==> Loss = 1.3334194053268187
Epoch 12==> Loss = 1.2580555223586956
Epoch 13==> Loss = 1.188824848423791
Epoch 14==> Loss = 1.1251501614639603
Epoch 15==> Loss = 1.0665377288456648
Epoch 16==> Loss = 1.01255358831892
Epoch 17==> Loss = 0.9628081251272314
Epoch 18==> Loss = 0.91694630962375
Epoch 19==> Loss = 0.8746416950998673
Epoch 20==> Loss = 0.8355928122120877
Epoch 21==> Loss = 0.7995209933104942
Epoch 22==> Loss = 0.7661689645929212
Epoch 23==> Loss = 0.7352997822344217
Epoch 24==> Loss = 0.7066958682414773
Epoch 25==> Loss = 0.6801580267654674
Epoch 26==> Loss = 0.6555043998408424
Epoch 27==> Loss = 0.6325693644243666
Epoch 28==> Loss = 0.611202392010732
Epoch 29==> Loss = 0.5912668976722616
Epoch 30==> Loss = 0.5726391038425067
Epoch 31==> Loss = 0.5552069396341206
Epoch 32==> Loss = 0.5388689911779094
Epoch 33==> Loss = 0.5235335134670185
Epoch 34==> Loss = 0.5091175099237313
Epoch 35==> Loss = 0.4955458824878889
Epoch 36==> Loss = 0.4827506524156746
Epoch 37==> Loss = 0.47067025007382146
Epoch 38==> Loss = 0.45924887069692105
Epoch 39==> Loss = 0.44843589222688485
Epoch 40==> Loss = 0.43818535086776417
Epoch 41==> Loss = 0.4284554697755726
Epoch 42==> Loss = 0.4192082362867634
Epoch 43==> Loss = 0.4104090232109444
Epoch 44==> Loss = 0.4020262499267101
Epoch 45==> Loss = 0.3940310792887177
Epoch 46==> Loss = 0.38639714665329306
Epoch 47==> Loss = 0.3791003176398894
Epoch 48==> Loss = 0.3721184715536468
Epoch 49==> Loss = 0.3654313076911607
Epoch 50==> Loss = 0.35902017203197295
Epoch 51==> Loss = 0.3528679020789166
Epoch 52==> Loss = 0.34695868784993344
Epoch 53==> Loss = 0.3412779472419981
Epoch 54==> Loss = 0.33581221418482476
Epoch 55==> Loss = 0.33054903817912024
Epoch 56==> Loss = 0.3254768939726215
Epoch 57==> Loss = 0.32058510026840736
Epoch 58==> Loss = 0.31586374648563453
Epoch 59==> Loss = 0.31130362670424355
```

```
Epoch 60==> Loss = 0.3068961800239603
Epoch 61==> Loss = 0.3026334366552369
Epoch 62==> Loss = 0.29850796913698974
Epoch 63==> Loss = 0.2945128481442094
Epoch 64==> Loss = 0.29064160240876696
Epoch 65==> Loss = 0.28688818232994223
Epoch 66==> Loss = 0.28324692689819486
Epoch 67==> Loss = 0.2797125335971915
Epoch 68==> Loss = 0.2762800309858108
Epoch 69==> Loss = 0.27294475369424637
Epoch 70==> Loss = 0.26970231959703933
Epoch 71==> Loss = 0.26654860895124266
Epoch 72==> Loss = 0.26347974531042695
Epoch 73==> Loss = 0.2604920780451522
Epoch 74==> Loss = 0.25758216631824155
Epoch 75==> Loss = 0.2547467643788638
Epoch 76==> Loss = 0.2519828080534127
Epoch 77==> Loss = 0.2492874023235641
```

```
"""
here I add noise in each position of the 25 positions of the matrix
"""
cc = ['l0_ya.npy', 'l1_yu.npy', 'l2_oi.npy', 'l3_yoi.npy',
 'l4_yai.npy',  'l5_p.npy', 'l6_m.npy', 'l7_t.npy',
 'l8_r.npy',  'l9_b.npy']

# fig345, ax345 = plt.subplots(5,5)
# test_letter  = d[cc[0].split('.')[0]]
# for i in range(5):
#     for j in range(5):
#         test_letter  = d[cc[0].split('.')[0]].copy()
#         test_letter[i,j] = 1 if not test_letter[i,j] else 0
#         ax345[i,j].imshow(np.array(test_letter.copy()), cmap='jet')

# for i in range(5):
#     for j in range(5):
#         print (5*i+j)

"""  10 examples for the training, each example has 6*5 features, or 30 virtual reservoir noc
"""
X = np.zeros((25,30))


num_letter =  0# between 0 and 9
# test_letter  = d[cc[0].split('.')[0]]
fig345, ax345 = plt.subplots(5,5)
for i in range(5):
    for j in range(5):
        test_letter  = d[cc[num_letter].split('.')[0]].copy()
        test_letter[i,j] = 1 if not test_letter[i,j] else 0
        initial_state = np.random.random(1)
        output = []
        case_letter = np.array(test_letter.copy())
        ax345[i,j].imshow(np.array(case_letter), cmap=plt.cm.Greens, clim=[-1,2])
```

```
        ax345[i,j].imshow(np.array(case_letter),cmap=plt.cm.Greens, clim=[-1,2])
        for row in case_letter:
            output.append(output_row(initial_state,row))
        X[5*i+j,:] = np.concatenate(output)[:,0]


def predict(X, w, b):
    """ X --> Input.
        w --> weights.
        b --> bias."""

    # Predicting
    z = X@w + b
    y_hat = softmax(z)
    # print (y_hat.shape)
    # Returning the class with highest probability.
    return np.argmax(y_hat, axis=1)
    # return y_hat

# for i in range(30)
print ('prediction of the corresponding letter', num_letter, ':', predict(X, w, b))
```

```
prediction of the corresponding letter 0 : [0 0 0 0 0 0 0 0 4 4 0 0 0 0 0 0 0 0 0
```

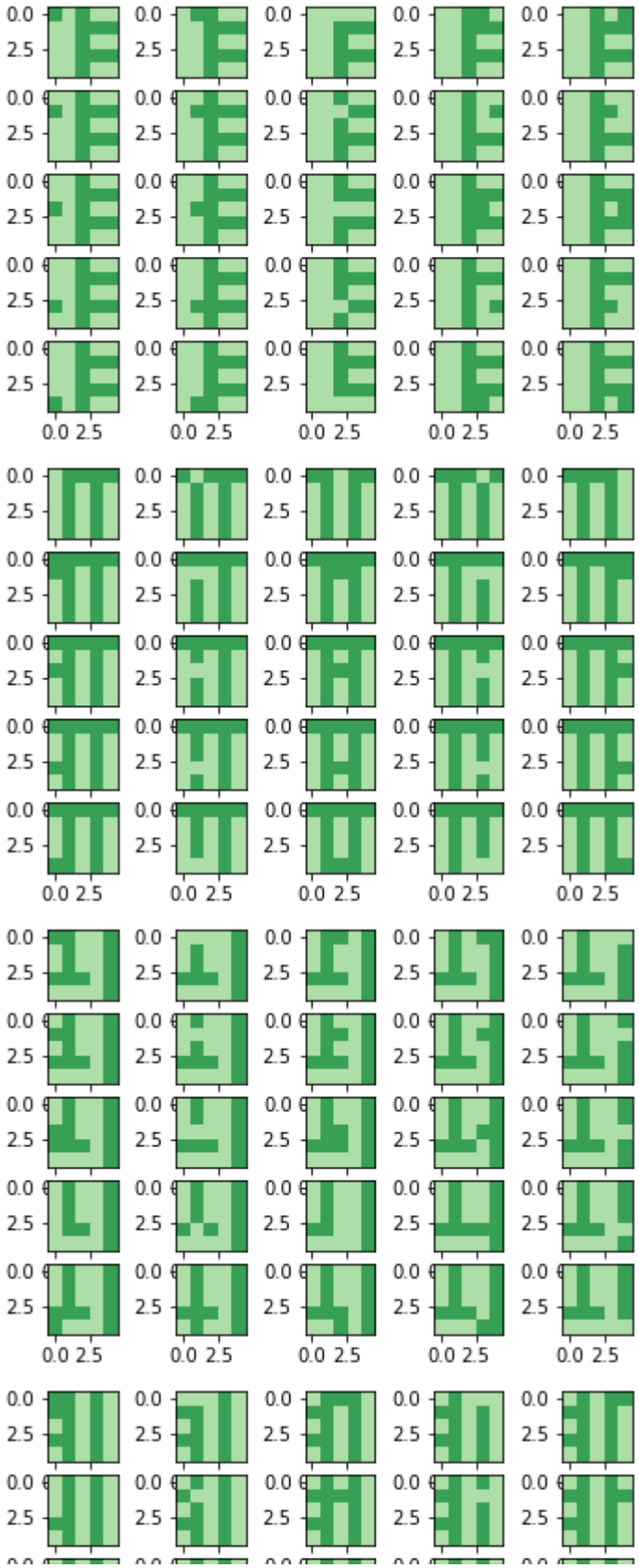


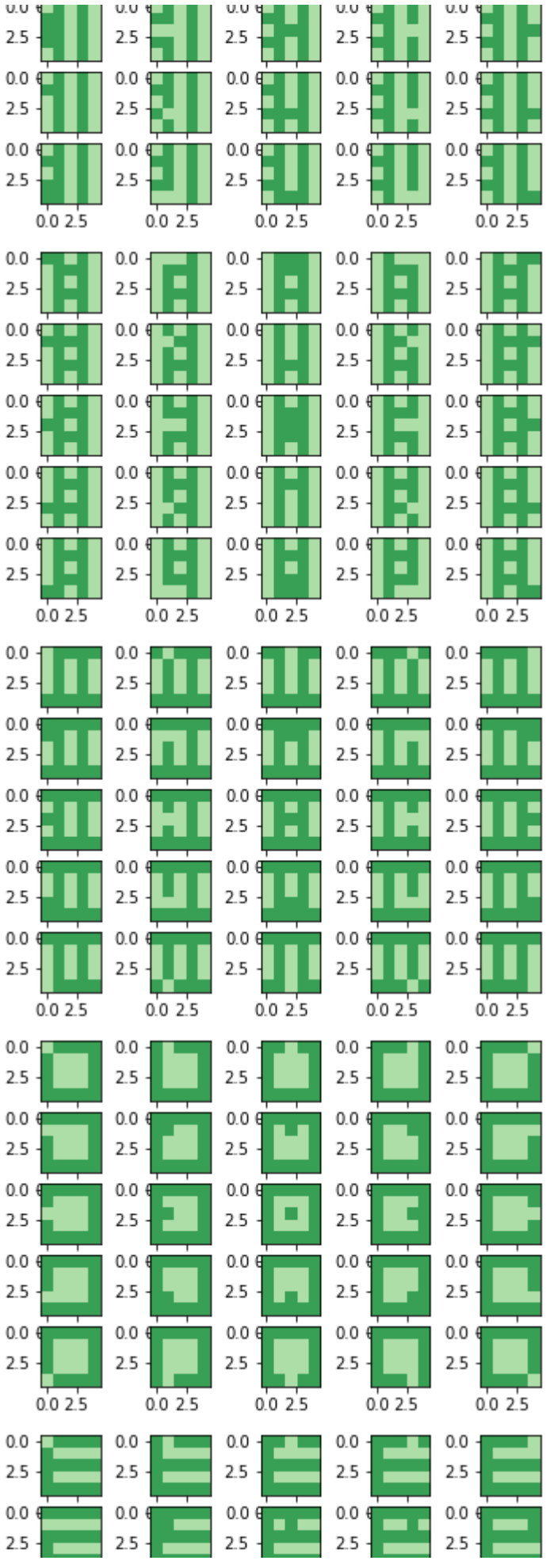

```
def predict(X, w, b):
    """ X --> Input.
        w --> weights.
        b --> bias."""

    # Predicting
    z = X@w + b
    y_hat = softmax(z)
    # print (y_hat.shape)
    # Returning the class with highest probability.
    return np.argmax(y_hat, axis=1)
```

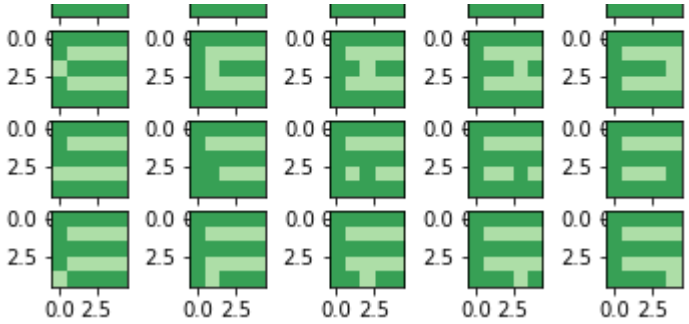
```
"""  test the 10 cases for making the confusion matrix
"""
X = np.zeros((25,30))

confusion_matrix = np.zeros((10,10))

for num_letter in range(10):# between 0 and 9
    # test_letter  = d[cc[0].split('.')[0]]
    fig345, ax345 = plt.subplots(5,5)
    for i in range(5):
        for j in range(5):
            test_letter  = d[cc[num_letter].split('.')[0]].copy()
            test_letter[i,j] = 1 if not test_letter[i,j] else 0
            initial_state = np.random.random(1)
            output = []
            case_letter = np.array(test_letter.copy())
            ax345[i,j].imshow(np.array(case_letter),cmap=plt.cm.Greens, clim=[-1,2])
            for row in case_letter:
                output.append(output_row(initial_state,row))
            X[5*i+j,:] = np.concatenate(output)[:,0]

    predictions = predict(X, w, b)
    print ('prediction of the corresponding letter', num_letter, ':', predictions)
    for n_lett, prob in Counter(predictions).items():
        confusion_matrix[num_letter, n_lett] = prob/25*100.

plt.figure()
plt.imshow(confusion_matrix)
```

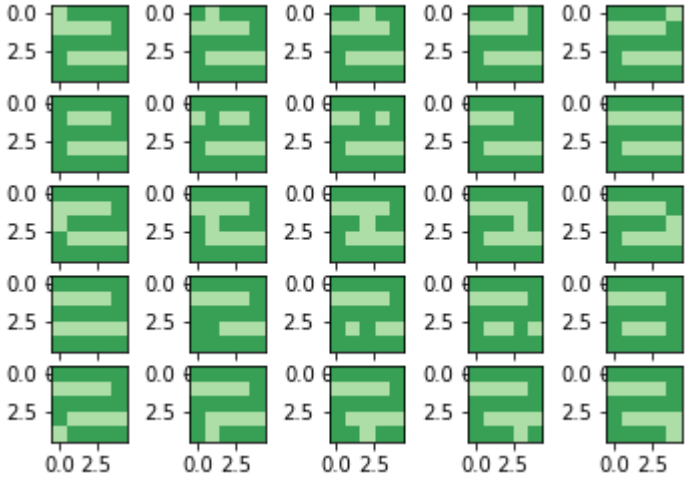
```
prediction of the corresponding letter 0 : [0 0 0 0 0 0 0 0 4 4 0 0 0 0 0 0 0 0 0
prediction of the corresponding letter 1 : [1 1 1 5 1 1 1 1 1 1 1 1 7 1 1 1 1 1 1
prediction of the corresponding letter 2 : [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

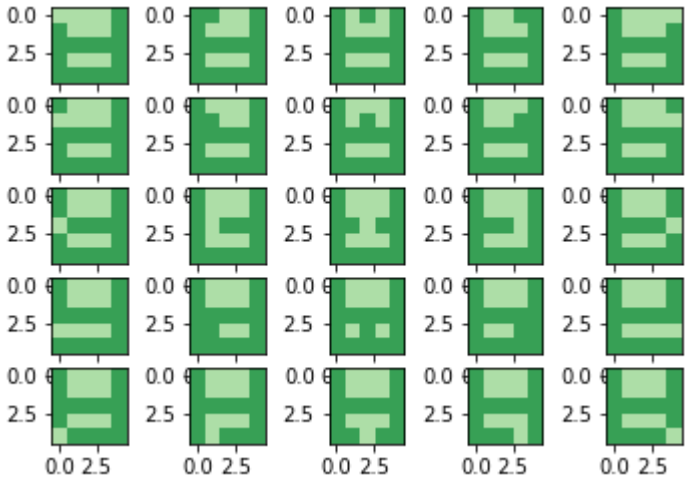
```
prediction of the corresponding letter 3 : [3 3 1 3 3 3 2 3 3 3 3 3 3 3 4 3 3 3 3
prediction of the corresponding letter 4 : [4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
prediction of the corresponding letter 5 : [5 6 5 5 5 5 5 5 6 5 8 6 8 5 5 5 5 5 5
prediction of the corresponding letter 6 : [6 6 6 6 5 8 6 6 6 6 6 6 8 6 6 6 6 6 6
prediction of the corresponding letter 7 : [7 8 7 7 7 7 7 7 8 7 7 6 6 8 7 7 7 8 7
prediction of the corresponding letter 8 : [8 8 8 7 7 7 8 8 8 8 8 6 8 8 7 7 7 8 7
prediction of the corresponding letter 9 : [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
<matplotlib.image.AxesImage at 0x7f25e915c710>
```

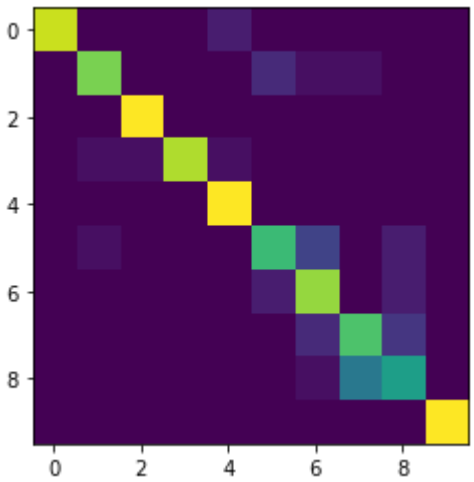



0.0
2.5
0.0 2.5


0.0
2.5
0.0 2.5


0.0
2.5
0.0 2.5


0.0
2.5
0.0 2.5


0
2
4
6
8