# TP2 : Multi-Layer Perceptrons

In [1]:

```
#imports
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn import metrics
from time import time
import matplotlib.pyplot as plt
import numpy as np
import warnings
```

In [2]:

```
#load dataset
mnist = fetch_openml('mnist_784')
```

## [1] Echantillonage du jeu de données et séparation en entraînement/test

*NB* : Au vu des temps d'exécution constatés, nous nous sommes limité à 1/10 du jeu de données initial, soit 7000 images au lieu des 70000 demandées.

In [3]:

```
#choose sample size
dataset_length = len(mnist.data)
sample_size = 7000
sample_size = min(dataset_length,sample_size)

#extract sample from dataset
sample_indexes = np.random.randint(dataset_length, size= sample_size)
data, target = np.array([mnist.data[i] for i in sample_indexes]), np.array([mnist.targe
t[i] for i in sample_indexes])

#extract train/test according to the proportion given in the subject
train_size = 49000/70000 #keep the given proportion
d_train, d_test, l_train, l_test = train_test_split(data,target,train_size = train_size
, random_state = 42)
```

## [2] Premiers réseaux de neurones

**Construire un modèle de classification ayant comme paramètre :hidden_layer_sizes = (50) puis calculez la précision du classifieur**

In [4]:

```
mlp = MLPClassifier(hidden_layer_sizes=(50), random_state=42)
mlp.fit(d_train, l_train)
score = mlp.score(d_test, l_test)

print(score)
```

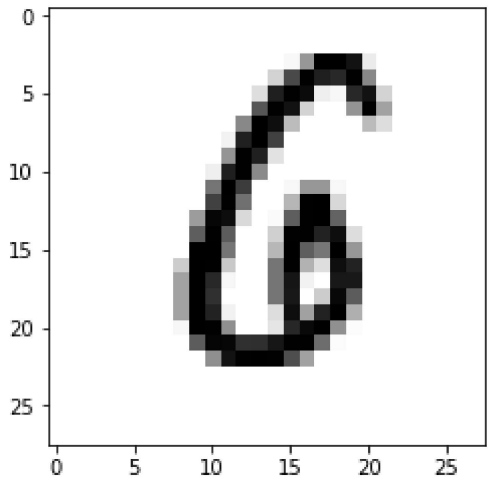
```
0.8790476190476191
```

**Afficher la classe de l'image 4 et sa classe prédite**

In [5]:

```
#Test de prediction = valeur réel
img_index = 3 #4th image of the dataset
images = data.reshape((-1, 28, 28))
plt.imshow(images[img_index],cmap=plt.cm.gray_r,interpolation="nearest")
res = mlp.predict(data[img_index:img_index+1])[0]
print("prediction :",res)
print("actual :",target[img_index])
```

```
prediction : 6
actual : 6
```




**Calculez la précision en utilisant le package : metrics.precision_score(ytest_pr, ypredTest_pr,average='micro').**

In [6]:

```
res = mlp.predict(d_test)
precision = metrics.precision_score(l_test,
                                    res,
                                    average='micro')
print(precision)
```

```
0.8790476190476191
```

# [3] Analyse : Nombres de couches et neurones

**Varier le nombre de couches de 1 entre (2 et 100) couches, et recalculer la précision du classifieur.**

In [7]:

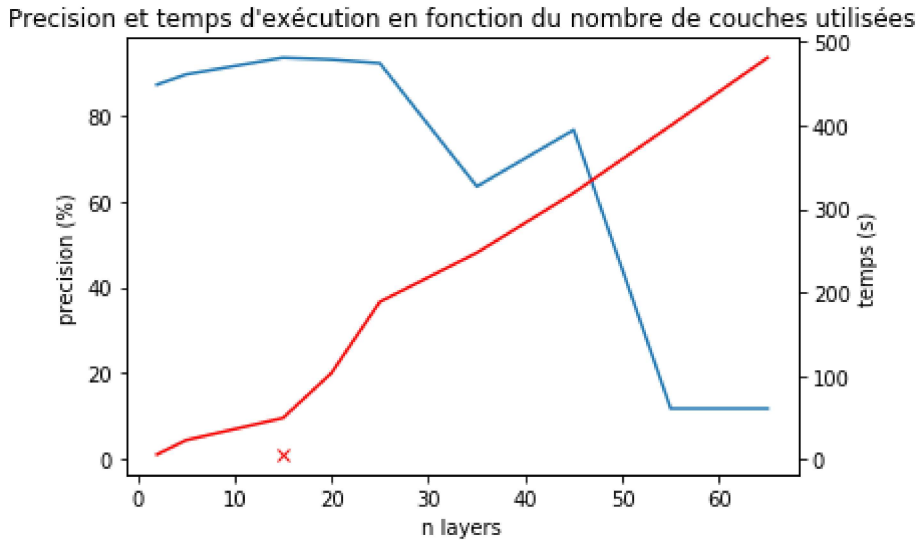
```
best_layers_nb = 0
layers_nb_set = [2,5,15,20,25,35,45,55,65]
best_score = 0
plot = ([],[],[])

neurons = np.geomspace(300, 40, num=100, dtype=int)#reduce neuron number logarithmicall
y to ensure convergence

start = time() #TIMER START
for layers_nb in layers_nb_set:
    start_iter = time()
    mlp = MLPClassifier(hidden_layer_sizes=neurons[:layers_nb])
    mlp.fit(d_train, l_train)
    prediction = mlp.predict(d_test)
    score = metrics.precision_score(l_test, prediction, average='micro')
    duration_iter = time()-start
    if score > best_score:
        best_score = score
        best_layers_nb = layers_nb
    plot[0].append(layers_nb)
    plot[1].append(100*score)
    plot[2].append(duration_iter)
    print("Execution time for",layers_nb,duration_iter,"s")

duration = time()-start #TIMER END
```

```
Execution time for 2 5.950281620025635 s
Execution time for 5 23.064556121826172 s
Execution time for 15 49.5737726688385 s
Execution time for 20 103.50929617881775 s
Execution time for 25 188.7758231163025 s
Execution time for 35 247.38115525245667 s
Execution time for 45 319.152809381485 s
Execution time for 55 399.71495246887207 s
Execution time for 65 480.9455759525299 s
```

In [8]:

```
fig=plt.figure()
ax=fig.add_subplot(111, label="precision")
ax2=ax.twinx()

ax.set_xlabel("n layers")
ax.set_ylabel("precision (%)")
ax.set_title("Precision et temps d'exécution en fonction du nombre de couches utilisée
s")
ax.plot(plot[0],plot[1])
ax.plot([best_layers_nb],[best_score], marker= 'x', color='r')

ax2.set_ylabel("temps (s)")
ax2.plot(plot[0],plot[2],color='r')

print("Optimal layers number :",best_layers_nb, "( score :",best_score,")")
```

```
Optimal layers number : 15 ( score : 0.9352380952380952 )
```




**Interprétation**

On observe que la précision augmente graduellement puis chute rapidement lorsque l'on dépasse 25 couches.

Puisque le temps d'exécution augmente lui fortement avec le nombre de couches, il semble préférable de se limiter à moins d'une vingtaine de couches pour notre analyse, voire moins d'une dizaine pusique là différence en précision est très faible.

On notera cependant que le nombre de neurones par couche joue probablement un rôle déterminant dans cette évolution, d'où la nécessité de l'analyse suivante.

**Construire cinq modèles de classification des données mnist avec des réseaux qui ont respectivement de 1 à 10 couches cachées, et des tailles de couches entre 10 et 300 neurones au choix d'une façon aléatoire.**

In [9]:

```
def build_random_models(n,layers_nb_range, layers_size_range):
    layers_nb = np.random.randint(low = layers_nb_range[0],high = layers_nb_range[1], s
ize = n)
    layers_sizes = np.random.randint(low = layers_size_range[0],high = layers_size_rang
e[1], size = n)
    res = []
    for i in range(n):
        nb = layers_nb[i]
        size = layers_sizes[i]
        model = MLPClassifier(hidden_layer_sizes=[size for j in range(nb)])
        res.append((model, nb, size))
    return res
```

In [10]:

```
models = build_random_models(5,(1,11),(10,301))
results = []
for model, nb, size in models:
    start = time()
    model.fit(d_train, l_train)
    prediction = model.predict(d_test)
    score = metrics.precision_score(l_test, prediction, average='micro')
    duration = time()-start
    print("Executed",nb,size,"score :",score,"in",duration,"second")
    results.append((score,nb,size,duration))
results = sorted(results)
for score, nb, size, duration in results:
    print("Layers :",nb,"Neurons :",size,"Precision :",score*100,"%","Exec :",duration,
"s")
```

```
Executed 4 132 score : 0.8890476190476191 in 18.50693154335022 second
Executed 8 48 score : 0.9276190476190476 in 17.8991961479187 second
Executed 4 65 score : 0.8738095238095238 in 16.002991437911987 second
Executed 6 49 score : 0.8971428571428571 in 17.431270599365234 second
Executed 7 131 score : 0.9242857142857143 in 23.361974000930786 second
Layers : 4 Neurons : 65 Precision : 87.38095238095238 % Exec : 16.00299143
7911987 s
Layers : 4 Neurons : 132 Precision : 88.90476190476191 % Exec : 18.5069315
4335022 s
Layers : 6 Neurons : 49 Precision : 89.71428571428571 % Exec : 17.43127059
9365234 s
Layers : 7 Neurons : 131 Precision : 92.42857142857143 % Exec : 23.3619740
00930786 s
Layers : 8 Neurons : 48 Precision : 92.76190476190476 % Exec : 17.89919614
79187 s
```

**Interprétation**

La précision du modèle augmente avec le nombre de couches(comme vu précedemment) mais surtout avec le nombre de neurones par couche. Etendre ce nombre de neurones représente cependant un coût en temps. Pour une itération, nous avons obtenu un modèle optimal à 94% de précision, avec 10 couches et 251 neurones par courche mais dont l'entraînement + prédiction prenait plus d'une minute sur 7000 données.

Un bon compromis semble être trouvé avec un modèle une cinquantaine de neurones sur 6 couches, soit 300 neurones au total, qui obtient un score > 90% pour 20 sec de temps d'exécution total.

Par la suite, on conservera ce nombre de 6 couches et 300 neurones comme référence. En revanche, on répartira les neurones de façon moins uniforme entre les couches, par exemple en utilisant np.geomspace pour obtenir une répartition logarithmique.

# [4] Analyse des paramètres supplémentaires

In [11]:

```
#Common parameters for the rest of the analysis
layers_nb = 6
layers_size = 50

up_size = 1.25 * layers_size
down_size = 0.75*layers_size

neurons = np.geomspace(up_size, down_size, num=100, dtype=int)#reduce neuron number gra
dually to ensure convergence
neurons = neurons[:layers_nb]
```

**Algorithmes d'optimisation**

In [12]:

```
algo_set = ['lbfgs', 'sgd', 'adam']
plot = ([],[],[],[],[])
for algo in algo_set:
    print("Algo :",algo, end='')
    start = time()#TIMER START

    mlp = MLPClassifier(hidden_layer_sizes=neurons, solver = algo)
    mlp.fit(d_train, l_train)
    prediction = mlp.predict(d_test)

    duration = time() - start #TIMER END

    recall = metrics.recall_score(l_test, prediction, average = 'micro')
    score = metrics.precision_score(l_test, prediction, average='micro')
    error = metrics.zero_one_loss(l_test, prediction)

    plot[0].append(algo)
    plot[1].append(score)
    plot[2].append(recall)
    plot[3].append(error)
    plot[4].append(duration)
    print(" | Done")
```

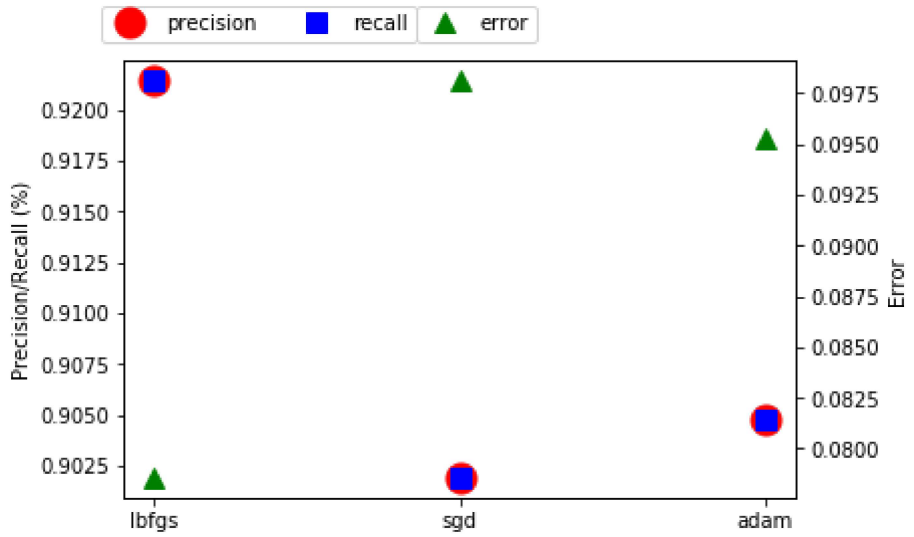
```
Algo : lbfgs | Done
Algo : sgd | Done
Algo : adam | Done
```

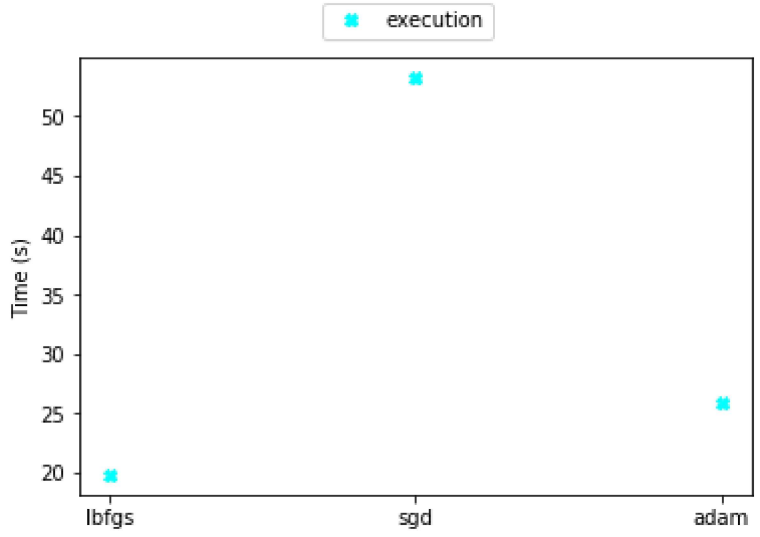
In [13]:

```
fig=plt.figure()
ax1=fig.add_subplot(111, label="precision")
ax2=ax1.twinx()


ax1.plot(plot[0],plot[1],linestyle='None',marker='o',ms=15.0,color='r', label='precisio
n')
ax1.plot(plot[0],plot[2],linestyle='None',marker='s',ms=10,color='b', label='recall')
ax2.plot(plot[0],plot[3],linestyle='None',marker='^',ms=10,color='g', label='error')

ax1.set_ylabel("Precision/Recall (%)")
ax2.set_ylabel("Error")
ax1.legend(ncol = 2, bbox_to_anchor=(0.45,1.15))
ax2.legend(bbox_to_anchor = (0.63, 1.15))

fig2=plt.figure()
ax3=fig2.add_subplot(111, label="error")
ax3.plot(plot[0],plot[4],linestyle='None',marker='X',color='cyan',label='execution')
ax3.set_ylabel("Time (s)")
ax3.legend(bbox_to_anchor = (0.63, 1.15))
```

Out[13]:

```
<matplotlib.legend.Legend at 0x2bd4f3481d0>
```




**Interprétation**

On observe des scores supérieurs sur tous les points pour lbfgs, avec, de plus, un temps d'exécution inférieurs.

En revanche, selon le jeu d'entraînement, nous avons observé des cas de non-convergence pour cet algorithme(voire plus bas). Les scores obtenus restaient supérieurs aux deux autres. L'algorithme itère donc plus vite au vu du temps d'exécution, mais converge moins rapidement. Cependant, même dans les cas de non-convergence les résultats sont supérieurs aux deux autres, lbfgs sera donc l'algorithme retenu.

**Fonctions d'activations**

In [14]:

```
algo = 'lbfgs'
function_set = ['identity', 'logistic', 'tanh', 'relu']
plot = ([],[],[],[],[])
for function in function_set:
    print("Function :",function,end='')
    start = time()#TIMER START

    mlp = MLPClassifier(hidden_layer_sizes=neurons, solver = algo, activation = functio
n)
    mlp.fit(d_train, l_train)
    prediction = mlp.predict(d_test)

    duration = time() - start #TIMER END

    recall = metrics.recall_score(l_test, prediction, average = 'micro')
    score = metrics.precision_score(l_test, prediction, average='micro')
    error = metrics.zero_one_loss(l_test, prediction)

    plot[0].append(function)
    plot[1].append(100*score)
    plot[2].append(100*recall)
    plot[3].append(error)
    plot[4].append(duration)
    print(" | Done")
```

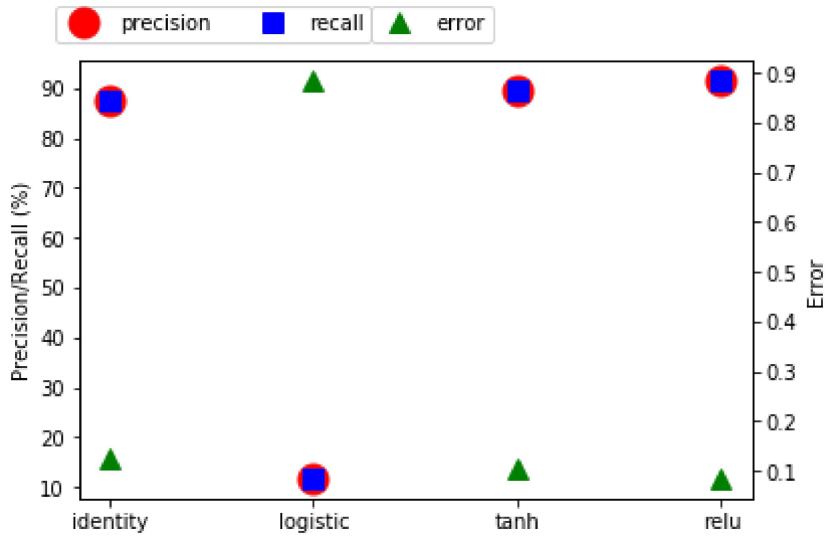
```
Function : identity

c:\dev\python\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:471: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown i
n:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)

 | Done
Function : logistic | Done
Function : tanh

c:\dev\python\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:471: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown i
n:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)

 | Done
Function : relu | Done
```

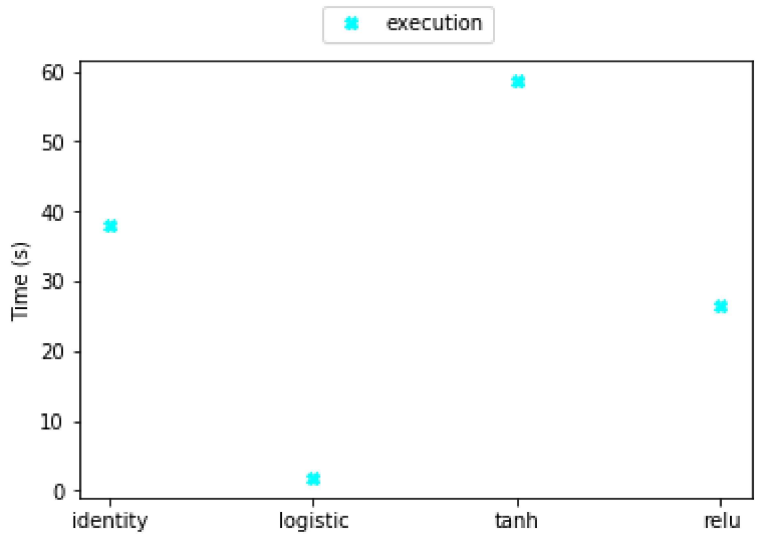
In [15]:

```
fig=plt.figure()
ax1=fig.add_subplot(111, label="precision")
ax2=ax1.twinx()


ax1.plot(plot[0],plot[1],linestyle='None',marker='o',ms=15.0,color='r', label='precisio
n')
ax1.plot(plot[0],plot[2],linestyle='None',marker='s',ms=10,color='b', label='recall')
ax2.plot(plot[0],plot[3],linestyle='None',marker='^',ms=10,color='g', label='error')

ax1.set_ylabel("Precision/Recall (%)")
ax2.set_ylabel("Error")
ax1.legend(ncol = 2, bbox_to_anchor=(0.45,1.15))
ax2.legend(bbox_to_anchor = (0.63, 1.15))

fig2=plt.figure()
ax3=fig2.add_subplot(111, label="error")
ax3.plot(plot[0],plot[4],linestyle='None',marker='X',color='cyan',label='execution')
ax3.set_ylabel("Time (s)")
ax3.legend(bbox_to_anchor = (0.63, 1.15))
```

Out[15]:

```
<matplotlib.legend.Legend at 0x2bd53786278>
```



### Interprétation

La fonction la plus intéressante semble demeurer relu, très comparable à identity cependant(logique puisque relu est une légère variante de cette dernière). Son temps d'exécution est comparable à celui de la fonction identity, mais les scores de relu sont plus intéressants. La fonction logistic est plus rapide mais produit une qualité de prédiction rédhibitoire, la fonction than obtient des scores comparables mais est moins rapide.

### Régulation L2

In [16]:

```
algo = 'lbfgs'
function = 'relu'
alpha_set = [10**(-i) for i in range(8,0,-1)]
plot = ([],[],[],[],[])
for p,alpha in enumerate(alpha_set):
    print("Regulation L2 :",alpha,end='')
    start = time()#TIMER START

    mlp = MLPClassifier(hidden_layer_sizes=neurons, solver = algo, activation = functio
n, alpha = alpha)
    mlp.fit(d_train, l_train)
    prediction = mlp.predict(d_test)

    duration = time() - start #TIMER END

    recall = metrics.recall_score(l_test, prediction, average = 'micro')
    score = metrics.precision_score(l_test, prediction, average='micro')
    error = metrics.zero_one_loss(l_test, prediction)

    plot[0].append(p+1)#Logarithmic scale
    plot[1].append(100*score)
    plot[2].append(100*recall)
    plot[3].append(error)
    plot[4].append(duration)
    print(" | Done")
```

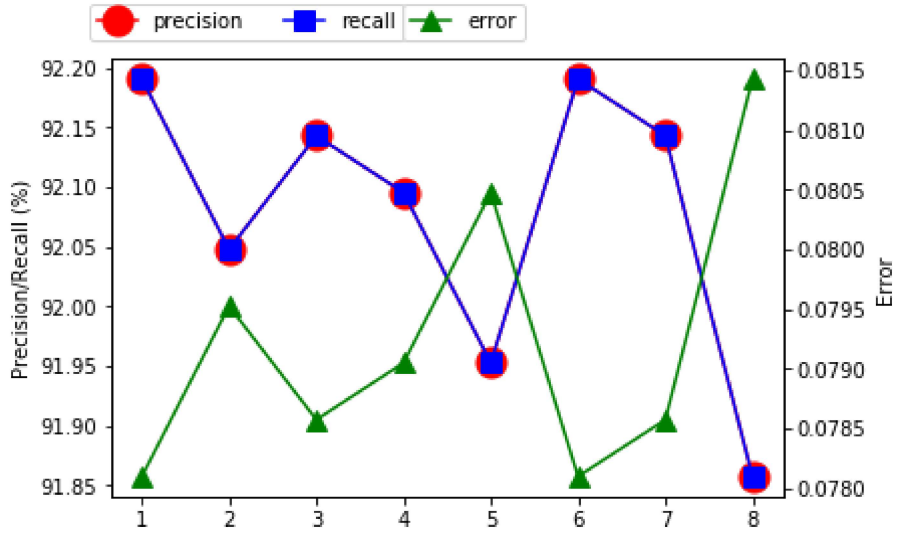
```
Regulation L2 : 1e-08 | Done
Regulation L2 : 1e-07 | Done
Regulation L2 : 1e-06 | Done
Regulation L2 : 1e-05 | Done
Regulation L2 : 0.0001 | Done
Regulation L2 : 0.001 | Done
Regulation L2 : 0.01 | Done
Regulation L2 : 0.1 | Done
```

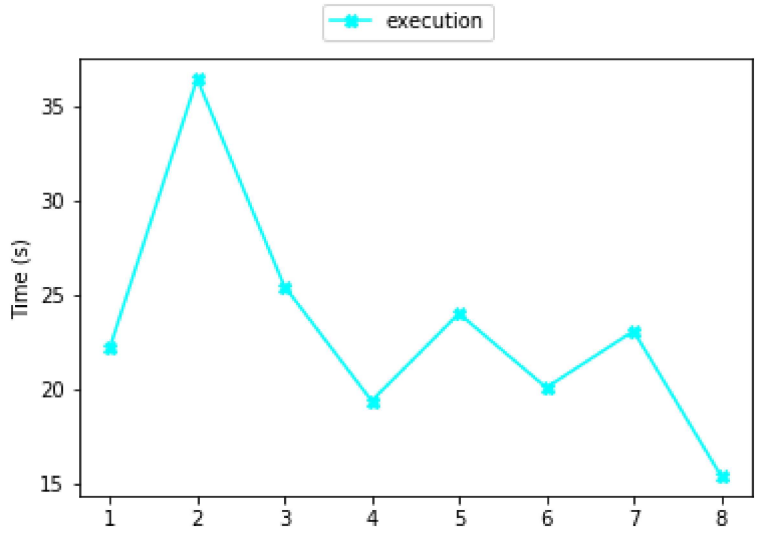
In [17]:

```
fig=plt.figure()
ax1=fig.add_subplot(111, label="precision")
ax2=ax1.twinx()


ax1.plot(plot[0],plot[1],marker='o',ms=15.0,color='r', label='precision')
ax1.plot(plot[0],plot[2],marker='s',ms=10,color='b', label='recall')
ax2.plot(plot[0],plot[3],marker='^',ms=10,color='g', label='error')

ax1.set_ylabel("Precision/Recall (%)")
ax2.set_ylabel("Error")
ax1.legend(ncol = 2, bbox_to_anchor=(0.45,1.15))
ax2.legend(bbox_to_anchor = (0.63, 1.15))

fig2=plt.figure()
ax3=fig2.add_subplot(111, label="error")
ax3.plot(plot[0],plot[4],marker='X',color='cyan',label='execution')
ax3.set_ylabel("Time (s)")
ax3.legend(bbox_to_anchor = (0.63, 1.15))
```

Out[17]:

```
<matplotlib.legend.Legend at 0x2bd5368e470>
```



**Interprétation**

Les scores de classification semblent aller croissant avec la valeur du paramètre alpha, atteignant néanmoins un plateau autour de 92.3% de précision à partir de 10e-3. Les temps d'exécution sont eux cependant bien moindre pour les valeurs plus élevées, on choisira donc la valeur max 0.1.

# Paramètres retenus

Pour le classifieur MLP, nos résultats indiquent donc qu'il serait optimal d'utiliser l'algorithme d'optimisation LBFGS, la fonction d'activation relu, et une valeur de régulation L2 à 0.1

Le nombre de neurones/couches idéal semble être aux alentours de 300 neurones et 5 couches, répartis non-uniformément pour aider à la convergence.

# FIN TP2