Domuta Alessia 2009986

Simulazione python

L'obiettivo è quello di implementare una simulazione con modelli di reti neurali utilizzando l'ambiente di sviluppo python.
La simulazione è basata sul dataset MNIST fornito nel terzo laboratorio, quindi la risoluzione è simile.

Riconoscimento cifre

Il seguente codice descrive l'inizializzazione del database. L'architettura adottata segue lo stesso approccio utilizzato nei laboratori, che prevede 784 neuroni di input, due strati nascosti composti da 500 neuroni ciascuno, e 10 neuroni di output corrispondenti al numeri delle classi.
Si procede dunque ad addestrare la rete.

```
[1] from sklearn.neural_network import MLPClassifier
    from torchvision.datasets import MNIST
    from torchvision.transforms import Lambda
```

```
    %%capture
    mnist_train = MNIST(root="../mnist",
                        train=True,
                        download=True)
    mnist_test = MNIST(root="../mnist",
                        train=False,
                        download=True)
```

```
[3] mnist_train_data, mnist_train_targets = mnist_train.data.numpy(), mnist_train.targets.numpy()
    mnist_test_data, mnist_test_targets = mnist_test.data.numpy(), mnist_test.targets.numpy()
```

```
[4] mnist_train_data = mnist_train_data.reshape(60000, 28*28)
    mnist_test_data = mnist_test_data.reshape(10000, 28*28)
```

```
[5] mnist_train_data = mnist_train_data / 255
    mnist_test_data = mnist_test_data / 255
```

```
[6] # con 10 iterazioni la convergenza è già abbastanza buona e l'algoritmo
    # impiega circa 3 minuti a completare l'esecuzione. Volendo una convergenza
    # migliore si può aumentare il numero di iterazioni

    random_state = 0

    MLP = MLPClassifier(hidden_layer_sizes=(500, 500),
                        max_iter = 10,
                        random_state=random_state)
```

```
[7] MLP = MLP.fit(mnist_train_data, mnist_train_targets)
```

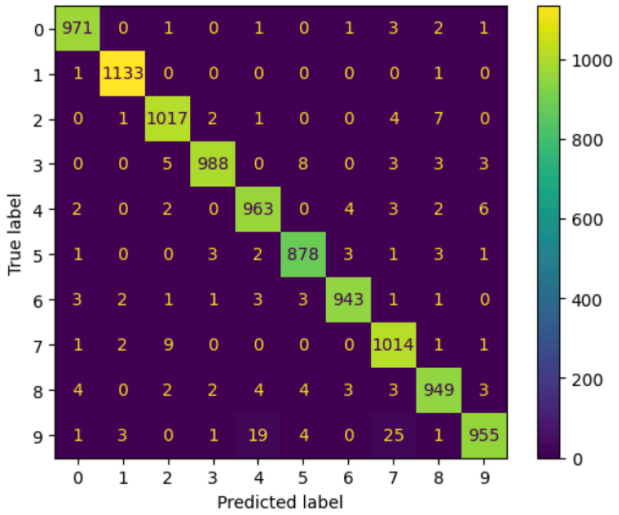
```
    ConvergenceWarning: Stochastic Optimizer: Maximum iterations (10) reached and the optimization hasn't converged yet.
```

Domuta Alessia 2009986

```
[8] import matplotlib.pyplot as plt
    import sklearn.metrics as metrics
```

```
[10] accuratezza = MLP.score(mnist_test_data, mnist_test_targets)
     accuratezza
```

```
_ = metrics.ConfusionMatrixDisplay.from_predictions(mnist_test_targets,
                                                    test_predictions)
```

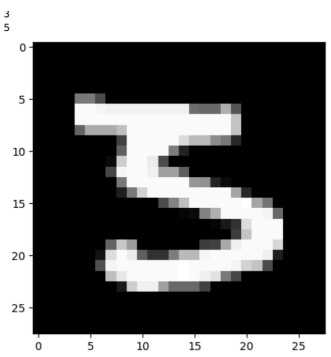


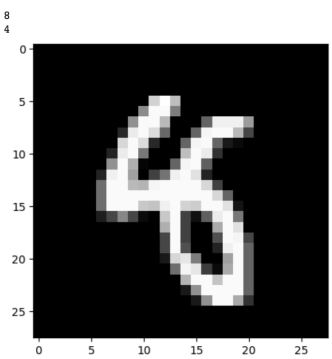
Per capire quali sono le cifre più difficili da riconoscere è necessario calcolare il True Positive rate:

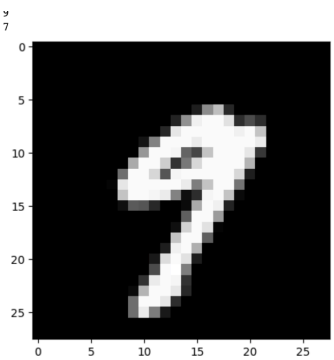
- Classe 0: 0,990816326530612
- Classe 1: 0,998237885462555
- Classe 2: 0,98546511627907
- Classe 3: 0,978217821782178
- Classe 4: 0,980651731160896
- Classe 5: 0,984304932735426
- Classe 6: 0,984342379958246
- Classe 7: 0,986381322957198
- Classe 8: 0,974332648870637
- Classe 9: 0,946481665014866

Le cifre più difficili da riconoscere sono 3, 8, 9. Di seguito qualche pattern classificato in modo errato.

```
for i in range(len(mnist_test_targets)):
  if mnist_test_targets[i] == 3 and mnist_test_targets[i] != test_predictions[i]:
    _ = plt.imshow(mnist_test_data[i].reshape(28, 28), cmap="gray")
    print(mnist_test_targets[i])
    print(test_predictions[i])
    break
```







Domuta Alessia 2009986

Le cifre sono effettivamente ambigue e anche un umano farebbe fatica a riconoscerle. Inoltre i 3 sono spesso confusi con i 5, gli 8 con i 4 e i 9 con i 7.

Introduzione del rumore

Si vuole introdurre del rumore sui dati, usando la funzione vista in laboratorio.

```
[22] import numpy as np
```

```
[23] def _inject_Gaussian_noise(mnist_data, noise_level):
       # creiamo una matrice di rumore della dimensione di un'immagine
       random_gaussian_tensor = np.random.normal(loc = 0, scale = noise_level, size = (1,784))
       # aggiungiamo il rumore ai pixel originali, tagliando i valori minori di 0 o maggiori di 1
       noisy_images = mnist_data + random_gaussian_tensor
       noisy_images = np.clip(noisy_images,0,1)
       return noisy_images
```

```
[24] noise_level = 0.3
     mnist_test_with_noise = _inject_Gaussian_noise(mnist_test_data, noise_level)
```

Effettuiamo la valutazione 10 volte, ognuna delle quali prevede un aumento di 0.1 di rumore nelle immagini.

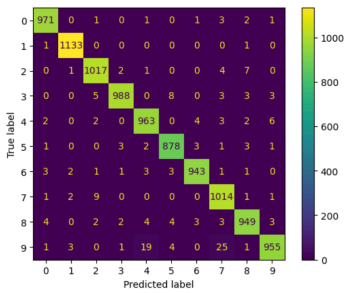
```
livelli_di_rumore = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
percentuale_errori = []

# testiamo il modello su stimoli contenenti un livello di rumore crescente:
for livello in livelli_di_rumore:
  mnist_con_rumore = _inject_Gaussian_noise(mnist_test_data, livello)
  accuratezza = MLP.score(mnist_con_rumore, mnist_test_targets)
  percentuale_errori.append(1-accuratezza)
  rec=metrics.recall_score(mnist_test_targets, MLP.predict(mnist_con_rumore), average=None)
  print("\n livello di rumore: ", livello, "\n accuratezza:", accuratezza, "\n recall per lable:")
  r=0
  while r<10:
    print (r, ": %3f" %rec[r])
    r=r+1
  metrics.ConfusionMatrixDisplay.from_predictions(mnist_test_targets, MLP.predict(mnist_con_rumore))
```

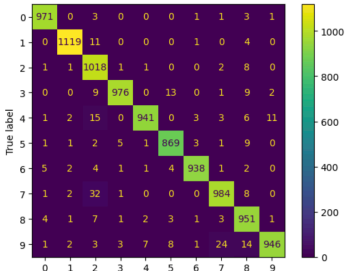
```
 livello di rumore:  0.0
 accuratezza: 0.9811
 recall per lable:
0 : 0.990816
1 : 0.998238
2 : 0.985465
3 : 0.978218
4 : 0.980652
5 : 0.984305
6 : 0.984342
7 : 0.986381
8 : 0.974333
9 : 0.946482
```

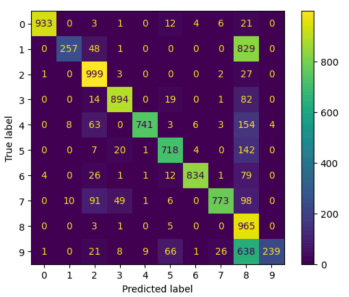
```
 livello di rumore:  0.2
 accuratezza: 0.9007
 recall per lable:
0 : 0.991837
1 : 0.739207
2 : 0.990310
3 : 0.958416
4 : 0.901222
5 : 0.936099
6 : 0.963466
7 : 0.888132
8 : 0.973306
9 : 0.695738
```

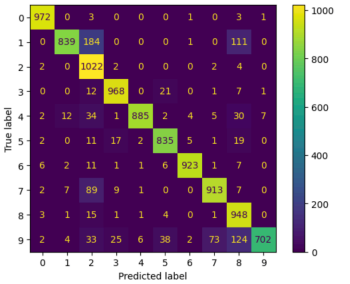
```
 livello di rumore:  0.3
 accuratezza: 0.7353
 recall per lable:
0 : 0.952041
1 : 0.226432
2 : 0.968023
3 : 0.885149
4 : 0.754582
5 : 0.804933
6 : 0.870564
7 : 0.751946
8 : 0.990760
9 : 0.236868
```

```
 livello di rumore:  0.4
 accuratezza: 0.6493
 recall per lable:
0 : 0.841837
1 : 0.113656
2 : 0.913760
3 : 0.638614
4 : 0.547862
5 : 0.857623
6 : 0.836117
7 : 0.900778
8 : 0.925051
9 : 0.019822
```

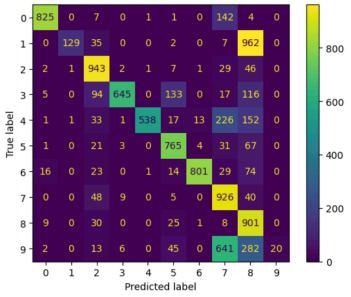




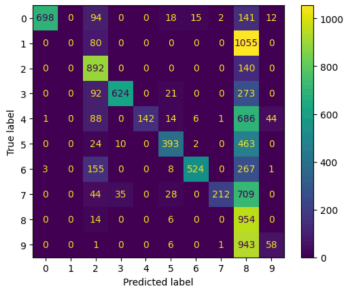




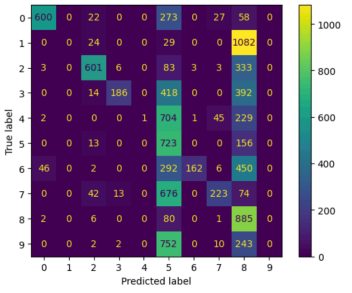
```
livello di rumore:  0.5
accuratezza: 0.4497
recall per lable:
0 : 0.712245
1 : 0.000000
2 : 0.864341
3 : 0.617822
4 : 0.144603
5 : 0.440583
6 : 0.546973
7 : 0.206226
8 : 0.979466
9 : 0.057483
```



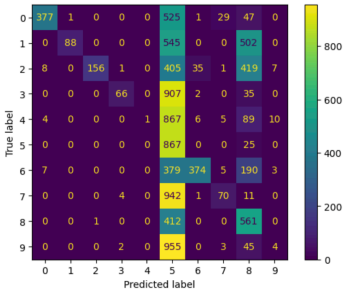

```
livello di rumore:  0.6
accuratezza: 0.3381
recall per lable:
0 : 0.612245
1 : 0.000000
2 : 0.582364
3 : 0.184158
4 : 0.001018
5 : 0.810538
6 : 0.169102
7 : 0.216926
8 : 0.908624
9 : 0.000000
```



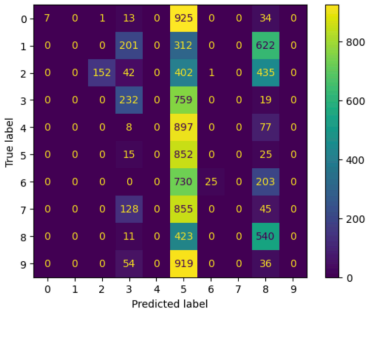

```
livello di rumore:  0.7
accuratezza: 0.2564
recall per lable:
0 : 0.384694
1 : 0.077533
2 : 0.151163
3 : 0.065347
4 : 0.001018
5 : 0.971973
6 : 0.390397
7 : 0.068093
8 : 0.575975
9 : 0.003964
```



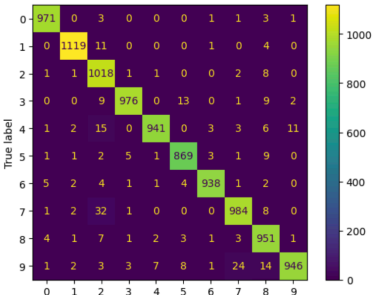

```
livello di rumore:  0.8
accuratezza: 0.2934
recall per lable:
0 : 0.974490
1 : 0.000000
2 : 0.002907
3 : 0.000000
4 : 0.000000
5 : 0.627803
6 : 0.473904
7 : 0.074903
8 : 0.908624
9 : 0.000000
```




```
livello di rumore:  0.9
accuratezza: 0.1808
recall per lable:
0 : 0.007143
1 : 0.000000
2 : 0.147287
3 : 0.229703
4 : 0.000000
5 : 0.955157
6 : 0.026096
7 : 0.000000
8 : 0.554415
9 : 0.000000
```



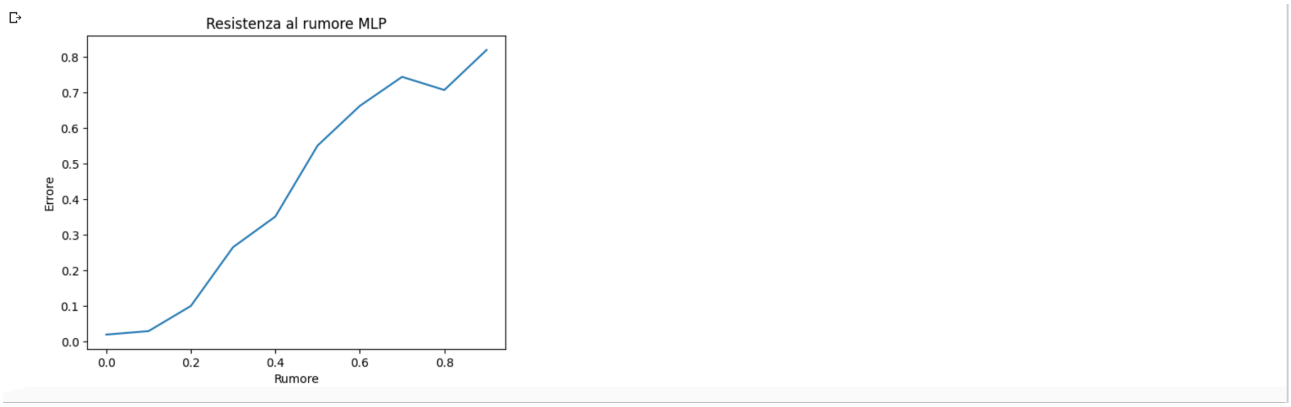

```
livello di rumore:  0.1
accuratezza: 0.9713
recall per lable:
0 : 0.990816
1 : 0.985903
2 : 0.986434
3 : 0.966337
4 : 0.958248
5 : 0.974215
6 : 0.979123
7 : 0.957198
8 : 0.976386
9 : 0.937562
```



Si può notare che il modello è sempre meno accurato rispetto al progressivo aumento del rumore. Il modello diventa sempre meno in grado di classificare correttamente le cifre presenti nelle immagini al crescere del rumore. Sebbene l'accuratezza inizi a diminuire anche con piccole percentuali di rumore, è osservabile un deterioramento significativo a partire da 0,3 di rumore, come si nota dal seguente grafico.

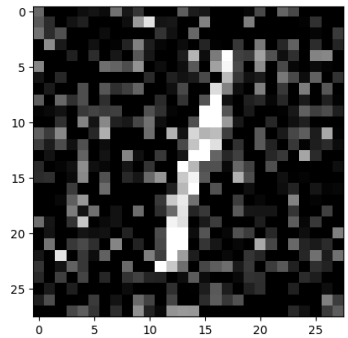
```
_ = plt.plot(livelli_di_rumore, percentuale_errori)
_ = plt.xlabel("Rumore")
_ = plt.ylabel("Errore")
_ = plt.title("Resistenza al rumore MLP")
```

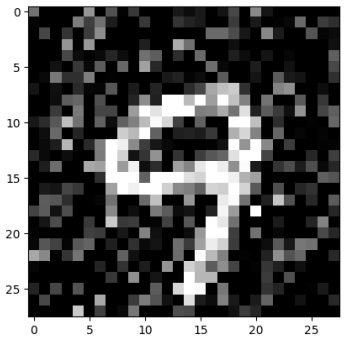
Domuta Alessia 2009986

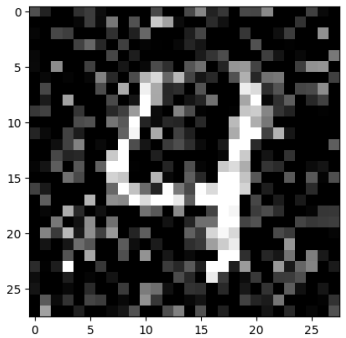


Di seguito sono presenti alcuni esempi di pattern classificati in modo errato a seguito dell'introduzione del rumore a 0.3 .

```
a = _inject_Gaussian_noise(mnist_test_data, 0.3)
b = MLP.predict(a)

x = 0
for i in range(len(mnist_test_targets)):
  if mnist_test_targets[i] == test_predictions[i] and mnist_test_targets[i] != b[i]:
    _ = plt.imshow(a[i].reshape(28, 28), cmap="gray")
    print(mnist_test_targets[i])
    print(b[i])
    x = x+1
    if x == 1:
      break
```







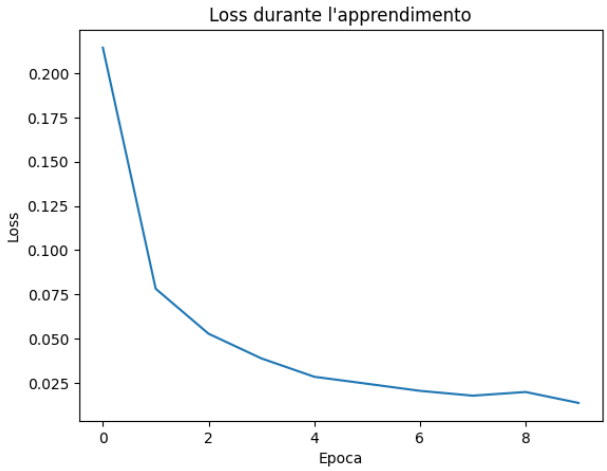
Variazione numero di neuroni e strati nascosti

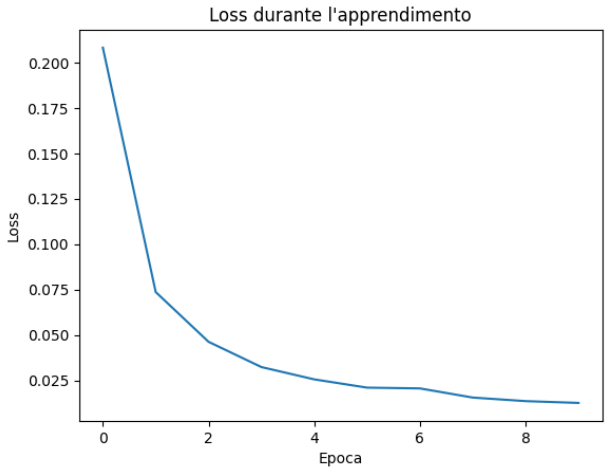
Domuta Alessia 2009986

La prestazione del modello scelto cambia al variare del numero dei neuroni e degli strati nascosti. É stato infatti valutato al raddoppiare nel numero di neuroni, al raddoppiare degli strati nascosti.
É stato valutato anche aumentando il massimo numero di iterazioni, raddoppiandolo.
Infine si è valutata anche l'introduzione del learning rate per verificare gli effetti sull'apprendimento.

```
random_state = 0

MLP2 = MLPClassifier(hidden_layer_sizes=(1000, 1000),     Modificabile
                     max_iter = 10,
                     random_state=random_state)

MLP2 = MLP2.fit(mnist_train_data, mnist_train_targets)

import matplotlib.pyplot as plt
import sklearn.metrics as metrics

_ = plt.plot(range(MLP2.n_iter_), MLP2.loss_curve_)
_ = plt.xlabel("Epoca")
_ = plt.ylabel("Loss")
_ = plt.title("Loss durante l'apprendimento")

accuratezza = MLP2.score(mnist_test_data, mnist_test_targets)
print("accuracy: ",accuratezza)

test_predictions2 = MLP2.predict(mnist_test_data)

_ = metrics.ConfusionMatrixDisplay.from_predictions(mnist_test_targets,
                                                    test_predictions2)
```

Numero di neuroni raddoppiato: da (500, 500) a (1000,1000). Numero massimo di iterazioni 10.

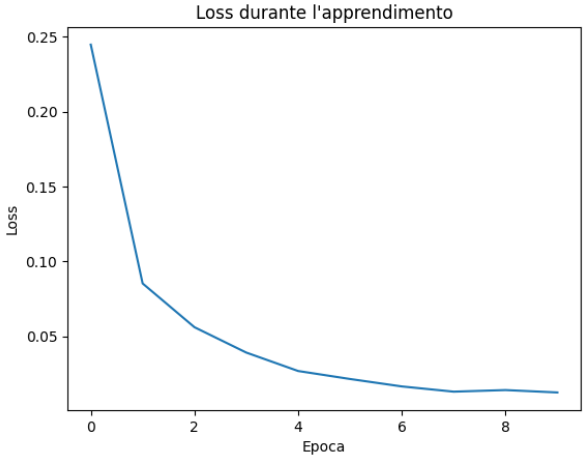


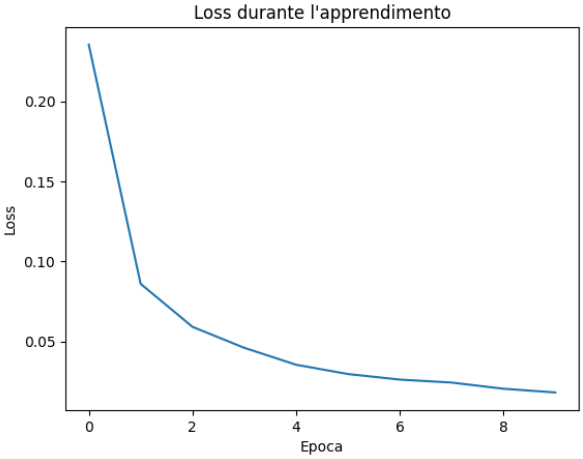


Accuracy passa da 0.9811 a 0.9787. In questo caso peggiora, dunque aumentare il numero dei neuroni non sempre è una buona idea perché è importante che il modello riesca a generalizzare e non ci si imbatta nel problema di overfitting.

Domuta Alessia 2009986

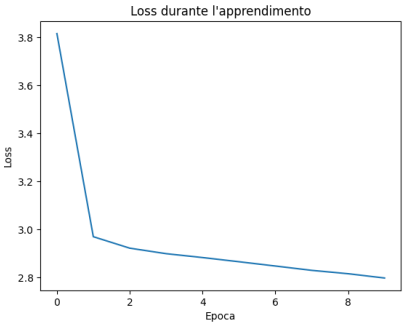
Raddoppiato il numero di strati nascosti da (500, 500) a (500, 500, 500, 500), numero massimo di interazioni invariato.

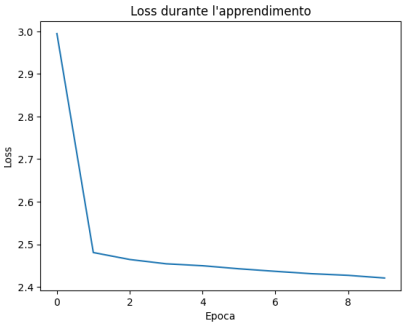


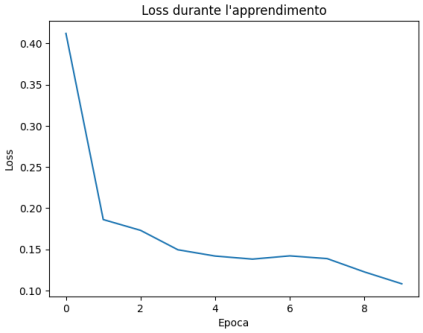


L’accuracy passa da 0.9811 a 0.9812 quindi è leggermente migliorata.
La curva che descrive la loss dell’apprendimento sembra convergere più lentamente ma in modo migliore. Il miglioramento nell’accuratezza ci suggerisce che in questo caso il problema di overfitting non si è verificato e che l’equilibrio si è mantenuto, permettendo dunque al modello di migliorare.

Un altro iper parametro da considerare è il learning rate. Questo controlla l’entità degli aggiornamenti dei pesi durante il processo di addestramento e può influenzare la precisione di convergenza. Si testano dunque diversi valori: 0.5, 0.25, 0.02.





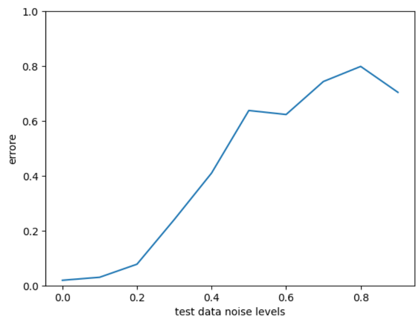


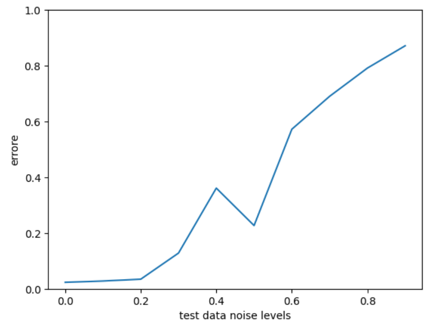
L’accuracy è rispettivamente: 0.101, 0.101, 0.9571.
Il learning rate troppo alto porta a un’accuratezza insufficiente nonostante raggiunga la convergenza in modo più lineare. Si nota che con un learning rate basso l’accuratezza migliora sebbene in grafico appaia meno stabile.

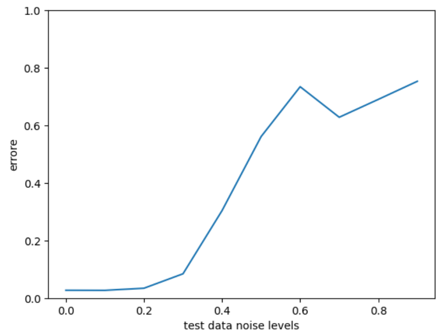
Domuta Alessia 2009986

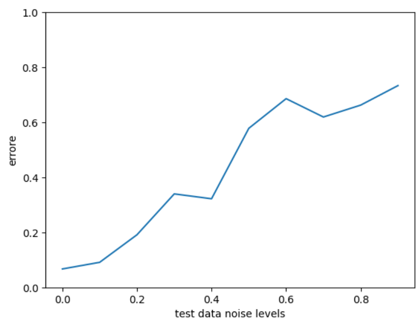
Introduzione del rumore

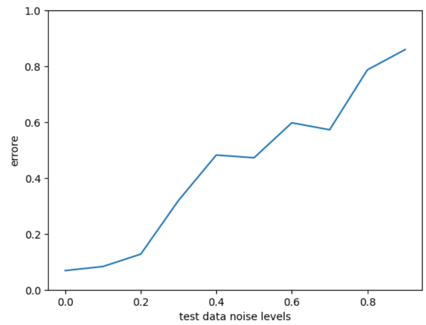
Si è voluto verificare come cambia l'accuratezza di riconoscimento una volta introdotto del rumore sui pattern di training, per vedere se migliora qualcosa nel riconoscimento di patter rumorosi.
Si utilizzano i livelli di rumore già visti e ad ogni iterazione il rumore viene aumentato di 0.1.
Infine si eseguono i test come in precedenza e si verifica l'accuracy ad ogni livello, mostrandolo con un grafico. In questo caso è stato scelto di valutare sull'errore in modo da avere una migliore idea su come si comporta.

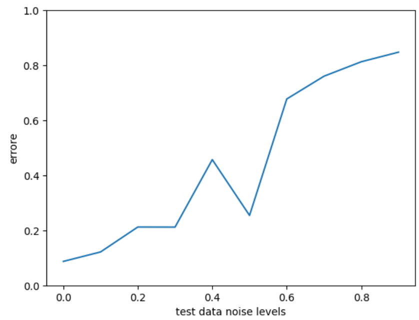


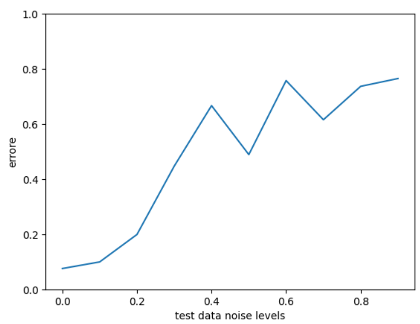


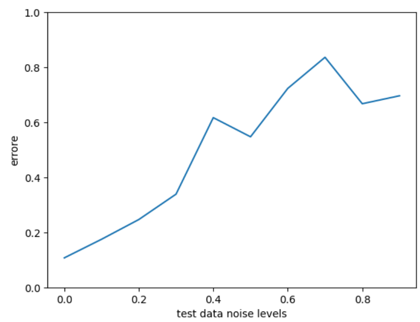


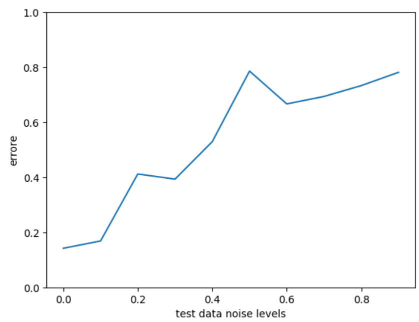


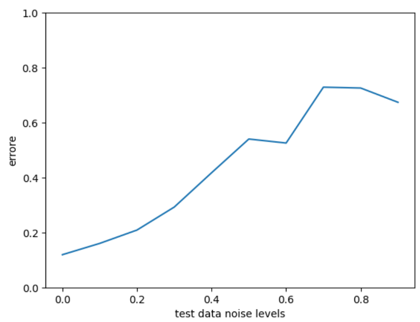










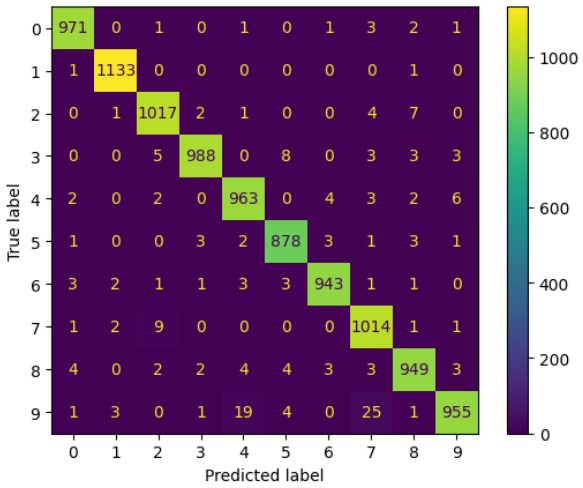


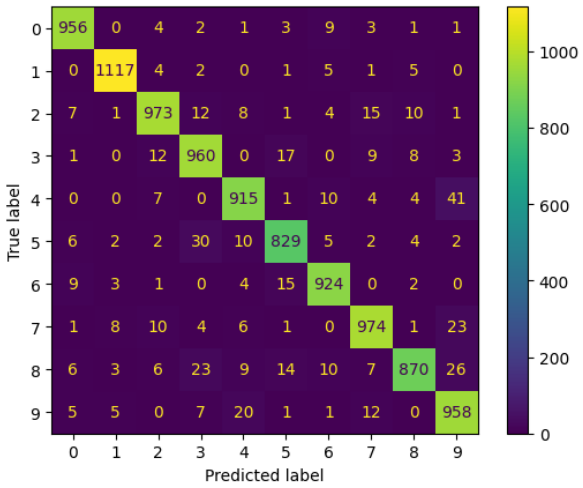
Si nota che per valori di rumore basso sul training set l'errore aumenta molto sui test set con rumore alto, ma è gestibile fino a circa 0.3, ottimale attorno a 0.2. Invece per dei valori di rumore alti del training set, il modello fa più fatica discriminare valori bassi sul test set.

Domuta Alessia 2009986

Riduzione pattern di training

Si vuole infine osservare come cambia l'accuracy di riconoscimento sui pattern di test topo aver tenuto solo il 10% dei pattern di training.





L'accuratezza con il 100% dei dati in training è di 0.9811 mentre una volta ridotti al 10% è di 0.9417 quindi è peggiorata. Nonostante ciò rimane un buon risultato contando che il training set è stato ridotto considerevolmente.