ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Corso di Laurea triennale in Ingegneria e Scienze Informatiche

Image Denoising Technique with Neural Network

Relatore:
Prof. Lazzaro Damiana

Co/Contro Relatore
Dott. Ezio Greggio

Candidato:
Matteo Vanni
Matricola: 0000935584

MMMCAINA 20256

Contents

1 Articolanzio

1.1 Panoramica sul problema

L'image denoising è il processo di rimozione di rumore da un'immagine.
Il rumore, che è causato da svariate fonti, quali foto fatte in condizioni di scarsa illuminazione o problemi che corrompono i file, causa perdita d'informazione sull'immagine.

Cos è il rumore? Un aggiunta casuale di pixel che non appartengono all'immagine originale e ce ne sono di varie tipologie:
Impulse Noise(IN) dove i pixel sono completamente diversi da quelli attorno. Esistono due categorie di IN: Salt and Pepper Noise(SPN) e Random Valued Impulse Noise(RVIN).
Additive White Gaussian Noise(AWGN) cambia ogni pixel dall'originale di una piccola quantità.

1.2 Utilizzo di modelli di Deep Learning

É essenziale rimuovere il rumore e ristabilire l'immagine originale dove riottenere l'immagine originale è importante per prestazioni robuste o ricostruire le informazioni mancanti è molto utile, come immagini astronomiche di oggetti molto lontani.
Le reti neurali convoluzionali lavorano bene con le immagini e ne utilizzeremo N, menzionate in alcuni paper di ricerca e compareremo i risultati di ogni modello.

1.3 Dataset utilizzati

Il primo dataset usato per addrestrare i modelli sarà Oxford-IIIT Pet da Tensorflow, in modo poi da testare i modelli con immagini che non conoscono da altri dataset(colonscopie)

```
1 def load_dataset(split='train', img_size=(256,256), batch_size
    ↪ =16):
2     #Carica il dataset Oxford-IIIT Pet da tfds e applica il #
        ↪ preprocessamento.
3
4     #Ritorna un dataset in batch.
5
6     # as_supervised=False per mantenere dict
7     dataset = tfds.load('oxford_iiit_pet', split=split,
        ↪ as_supervised=False)
8     # Preprocessing
```

```
    dataset = dataset.map(lambda sample: preprocess(sample,
        ↪ img_size))
    # Ottimizza il caricamento
    dataset = dataset.shuffle(1024).batch(batch_size).prefetch(tf
        ↪ .data.AUTOTUNE)
    return dataset
```

```
import numpy as np

def incmatrix(genl1,genl2):
    m = len(genl1)
    n = len(genl2)
    M = None #to become the incidence matrix
    VT = np.zeros((n*m,1), int)  #dummy variable

    #compute the bitwise xor matrix
    M1 = bitxormatrix(genl1)
    M2 = np.triu(bitxormatrix(genl2),1)

    for i in range(m-1):
        for j in range(i+1, m):
            [r,c] = np.where(M2 == M1[i,j])
            for k in range(len(r)):
                VT[(i)*n + r[k]] = 1;
                VT[(i)*n + c[k]] = 1;
                VT[(j)*n + r[k]] = 1;
                VT[(j)*n + c[k]] = 1;

                if M is None:
                    M = np.copy(VT)
                else:
                    M = np.concatenate((M, VT), 1)

                VT = np.zeros((n*m,1), int)

    return M
```

Kvasir-seg: 1000(MILLEH) immagini di polipi,
gli animali di arvard, le colonscopie

2 Decsrizione delle reti

2.1 Rumore

La prima funzione di rumore è u'aggiunta di rumore casuale a ogni pixel dell'immagine.

```
13   def add_noise(x, noise_factor=0.1):
14
15     x: immagine in input
16     noise_factor: fattore di rumore da aggiungere
17     Ritorna:
18       x_noisy: immagine con rumore aggiunto
19
20     noise = tf.random.normal(shape=tf.shape(x), mean=0.0, stddev
         ↪ =1.0)
21     x_noisy = x + noise_factor * noise
22     return x_noisy
```

E FAI RUMORE **QUIIIIIIIIIIIIIIIIIIIIIIIIIIIII** Prima funzione di rumore -¿ Random noise con fattore 0.3 massimo Rumore gaussiano Rumore a più layer

2.2 Autoencoder

Il primo approccio è stato quello di usare l'autencoder dell'articolo da cui sto copiando paro paro tutto.
Spiegazione del numero di layer usati e del tipo di mse loss fun, dam e learning rate di 1e-3

```
23 def build_autoencoder(input_shape):
24
25     Costruisce un autoencoder convoluzionale per immagini a
         ↪ colori.
26     Parametri:
27       input_shape: forma dell'immagine in input (es. (32,32,3))
28     Ritorna:
29       autoencoder: modello compilato
30
31     input_img = Input(shape=input_shape)
32
33     # Encoder
34     x = Conv2D(64, (3,3), activation='relu', padding='same')(
         ↪ input_img)
35     x = MaxPooling2D((2,2), padding='same')(x)
```

```
    x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2,2), padding='same')(x)

    # Decoder
    x = Conv2D(64, (3,3), activation='relu', padding='same')(
        ↪ encoded)
    x = UpSampling2D((2,2))(x)
    x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
    x = UpSampling2D((2,2))(x)
    decoded = Conv2D(3, (3,3), activation='sigmoid', padding='
        ↪ same')(x)

    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer='adam', loss='
        ↪ binary_crossentropy', metrics=['accuracy'])

    return autoencoder

autoencoder = build_autoencoder(input_shape=(img_size[0],
    ↪ img_size[1], 3))
autoencoder.summary()
```

2.3 RIDNet

stessa cosa dell'autencoder (vedere se aggiungere tutto il codice gigante del setup)

```
  # MeanShift: sottrae (o aggiunge) la media RGB
@register_keras_serializable( MeanShift )
class MeanShift(Layer):
    def __init__(self, rgb_mean, sign=-1, **kwargs):

        rgb_mean: tupla con la media dei canali R, G, B.
        sign: -1 per sottrarre la media, 1 per aggiungerla.

        super(MeanShift, self).__init__(**kwargs)
        self.rgb_mean = tf.constant(rgb_mean, dtype=tf.float32)
        self.sign = sign

    def call(self, x):
        # x    atteso in formato (batch, altezza, larghezza, 3)
        # Sfruttiamo Lambda per eseguire l'operazione per ogni
            ↪ elemento
        # Nota: non stiamo scalando per std in questo esempio
        return x + self.sign * self.rgb_mean

# BasicBlock: Conv2D seguita da attivazione ReLU
@register_keras_serializable( BasicBlock )
class BasicBlock(Layer):
    def __init__(self, out_channels, kernel_size=3, stride=1,
        ↪ use_bias=False, **kwargs):
```

```
75          super(BasicBlock, self).__init__(**kwargs)
76          self.conv = Conv2D(out_channels, kernel_size, strides=
               ↪ stride,
77                             padding='same', use_bias=use_bias)
78          self.relu = ReLU()
79
80      def call(self, x):
81          return self.relu(self.conv(x))
82
83  # ResidualBlock: due convoluzioni con skip connection(come lavora
       ↪  cuda con 2 thread -> 2 filtri applicati e poi si uniscono
       ↪ i risultati)
84  @register_keras_serializable( ResidualBlock )
85  class ResidualBlock(Layer):
86      def __init__(self, out_channels, **kwargs):
87          super(ResidualBlock, self).__init__(**kwargs)
88          self.conv1 = Conv2D(out_channels, 3, strides=1, padding='
               ↪ same')
89          self.relu = ReLU()
90          self.conv2 = Conv2D(out_channels, 3, strides=1, padding='
               ↪ same')
91
92      def call(self, x):
93          residual = self.conv1(x)
94          residual = self.relu(residual)
95          residual = self.conv2(residual)
96          out = Add()([x, residual])
97          return ReLU()(out)
98
99  # EResidualBlock: versione estesa con convoluzioni a gruppi
100 @register_keras_serializable( EResidualBlock )
101 class EResidualBlock(Layer):
102     def __init__(self, out_channels, groups=1, **kwargs):
103         super(EResidualBlock, self).__init__(**kwargs)
104         self.conv1 = Conv2D(out_channels, 3, strides=1, padding='
               ↪ same', groups=groups)
105         self.relu = ReLU()
106         self.conv2 = Conv2D(out_channels, 3, strides=1, padding='
               ↪ same', groups=groups)
107         self.conv3 = Conv2D(out_channels, 1, strides=1, padding='
               ↪ valid')
108
109     def call(self, x):
110         residual = self.conv1(x)
111         residual = self.relu(residual)
112         residual = self.conv2(residual)
113         residual = self.relu(residual)
114         residual = self.conv3(residual)
115         out = Add()([x, residual])
116         return ReLU()(out)
```

```
117
118 # Merge_Run_dual: due rami convoluzionali con dilatazioni diverse
    ↪ , poi fusione e skip connection
119 @register_keras_serializable( MergeRunDual )
120 class MergeRunDual(Layer):
121     def __init__(self, out_channels, **kwargs):
122         super(MergeRunDual, self).__init__(**kwargs)
123         # Primo ramo
124         self.conv1a = Conv2D(out_channels, 3, strides=1, padding=
            ↪ 'same')
125         self.relu1a = ReLU()
126         self.conv1b = Conv2D(out_channels, 3, strides=1, padding=
            ↪ 'same', dilation_rate=2)
127         self.relu1b = ReLU()
128         # Secondo ramo
129         self.conv2a = Conv2D(out_channels, 3, strides=1, padding=
            ↪ 'same', dilation_rate=3)
130         self.relu2a = ReLU()
131         self.conv2b = Conv2D(out_channels, 3, strides=1, padding=
            ↪ 'same', dilation_rate=4)
132         self.relu2b = ReLU()
133         # Gogeta
134         self.conv3 = Conv2D(out_channels, 3, strides=1, padding='
            ↪ same')
135         self.relu3 = ReLU()
136
137     def call(self, x):
138         branch1 = self.relu1a(self.conv1a(x))
139         branch1 = self.relu1b(self.conv1b(branch1))
140
141         branch2 = self.relu2a(self.conv2a(x))
142         branch2 = self.relu2b(self.conv2b(branch2))
143
144         merged = Concatenate()([branch1, branch2])
145         merged = self.relu3(self.conv3(merged))
146         return Add()([merged, x])
147
148 # CALayer: Channel Attention Layer
149 @register_keras_serializable( CALayer )
150 class CALayer(Layer):
151     def __init__(self, channel, reduction=16, **kwargs):
152         super(CALayer, self).__init__(**kwargs)
153         self.channel = channel
154         self.reduction = reduction
155         self.conv1 = Conv2D(channel // reduction, 1, strides=1,
            ↪ padding='same')
156         self.relu = ReLU()
157         self.conv2 = Conv2D(channel, 1, strides=1, padding='same'
            ↪ , activation='sigmoid') #sigmoid: risultato tra 0 e
            ↪  1
```

```
158
159     def call(self, x):
160         # Calcolo della media globale per canale
161         y = tf.reduce_mean(x, axis=[1, 2], keepdims=True)
162         y = self.relu(self.conv1(y))
163         y = self.conv2(y)
164         return Multiply()([x, y])
165
166 # Block: combinazione di MergeRunDual, ResidualBlock,
    ↪ EResidualBlock e CALayer
167 @register_keras_serializable( Block )
168 class Block(Layer):
169     def __init__(self, out_channels, **kwargs):
170         super(Block, self).__init__(**kwargs)
171         self.merge_run_dual = MergeRunDual(out_channels)
172         self.residual_block = ResidualBlock(out_channels)
173         self.e_residual_block = EResidualBlock(out_channels)
174         self.ca = CALayer(out_channels)
175
176     def call(self, x):
177         r1 = self.merge_run_dual(x)
178         r2 = self.residual_block(r1)
179         r3 = self.e_residual_block(r2)
180         out = self.ca(r3)
181         return out
182
183 # RIDNET: il modello final
184 @register_keras_serializable( RIDNET )
185 def RIDNET(n_feats=64, rgb_range=1.0):
186
187     n_feats: numero di feature channels usate all'interno del
        ↪ modello.<br>
188     rgb_range: scala dei valori RGB (ad es. 1.0 se l'input
        ↪ normalizzato [0,1]).
189
190     rgb_mean = (0.4488, 0.4371, 0.4040)
191
192     input_layer = Input(shape=(None, None, 3))
193
194     # Sottosottrai la media (MeanShift con sign=-1)
195     sub_mean = MeanShift(rgb_mean, sign=-1)
196     x = sub_mean(input_layer)
197
198     # Testa: BasicBlock (conv + ReLU)
199     head = Conv2D(n_feats, 3, strides=1, padding='same',
        ↪ activation='relu')(x)
200
201     # Serie di blocchi
202     b1 = Block(n_feats)(head)
203     b2 = Block(n_feats)(b1)
```

```
204     b3 = Block(n_feats)(b2)
205     b4 = Block(n_feats)(b3)
206 
207     # Coda: convoluzione finale per ottenere 3 canali
208     tail = Conv2D(3, 3, strides=1, padding='same')(b4)
209 
210     # Aggiungi la media (MeanShift con sign=+1)
211     add_mean = MeanShift(rgb_mean, sign=1)
212     res = add_mean(tail)
213 
214     # Connessione residua a livello di immagine: somma con l'
        ↪ input originale
215     output = Add()([res, input_layer])
216 
217     model = Model(inputs=input_layer, outputs=output)
218     return model
219 
220 model = RIDNET(n_feats=32, rgb_range=1.0)
221 model.compile(optimizer='adam', loss='mse', metrics=['accuracy']
    ↪ )
222 model.summary()
```

3 Analisi ed ottimizzazione

3.1 Prestazioni

PSNR è il metodo più comune per misurare la qualità delle immagini.
Il PSNR è definito come il rapporto tra il massimo valore possibile di un segnale e il valore del rumore che disturba la qualità della sua rappresentazione.
Normalmente misurato in una scala logaritmica in decibel(dB).
Data l'immagine originale(g) e l'immagine rumorosa(f), il PSNR è definito come:

$$PSNR = 20log_{10}(\frac{MAX_f}{\sqrt{MSE}})$$

dove MAX_f è il valore massimo del pixel dell'immagine e si calcola come

$$MSE = \frac{1}{mn}\sum_{0}^{m-1}\sum_{0}^{n-1}||f(i,j) - g(i,j)||^2$$

mentre MSE(*Mean Square Error*) è l'errore quadratico medio tra l'immagine originale e quella rumorosa.

3.2 Prima run

Il primo allenamento è stato sottoposto a entrambi i modelli con immagini RGB con risoluzione 112x112 pixel, un noise_factor di 0.4 e 50 epoche di training.
Per l'autoenconcoder è stata usata una batch_size di 16 mentre per la RIDNet una di 20.

3.2.1 Autoencoder

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer	(None, 112, 112, 3)	0
conv2d	(None, 112, 112, 64)	1,792
max_pooling2d	(None, 56, 56, 64)	0
conv2d_1	(None, 56, 56, 64)	36,928
max_pooling2d_1	(None, 28, 28, 64)	0
conv2d_2	(None, 28, 28, 64)	36,928
up_sampling2d	(N one, 56, 56, 64)	0
conv2d_3	(None, 56, 56, 64)	36,928
up_sampling2d_1	(None, 112, 112, 64)	0
conv2d_4	(None, 112, 112, 3)	1,731

Total params: 114,307 (446.51 KB)
Trainable params: 114,307 (446.51 KB)
Non-trainable params: 0 (0.00 B)

3.2.2 RIDNet

Model: "functional_5"

Layer (type)	Output Shape	Param #	Connected to
input_layer_5	(None, None, None, 3)	0	-
mean_shift_10	(None, None, None, 3)	0	input_layer_5[0][0]
conv2d_250 (Conv2D)	(None, None, None, 32)	896	mean_shift_10[0][0]
block_20 (Block)	(None, None, None, 32)	93,666	conv2d_250[0][0]
block_21 (Block)	(None, None, None, 32)	93,666	block_20[0][0]
block_22 (Block)	(None, None, None, 32)	93,666	block_21[0][0]
block_23 (Block)	(None, None, None, 32)	93,666	block_22[0][0]
conv2d_299 (Conv2D)	(None, None, None, 3)	867	block_23[0][0]
mean_shift_11	(None, None, None, 3)	0	conv2d_299[0][0]
add_413 (Add)	(None, None, None, 3)	0	mean_shift_11[0][0]

Total params: 376,427 (1.44 MB)
Trainable params: 376,427 (1.44 MB)
Non-trainable params: 0 (0.00 B)

3.3 Quantizzazione dei modelli

3.4 Analisi dei modelli

3.4.1 Performance su dataset sconosciuti

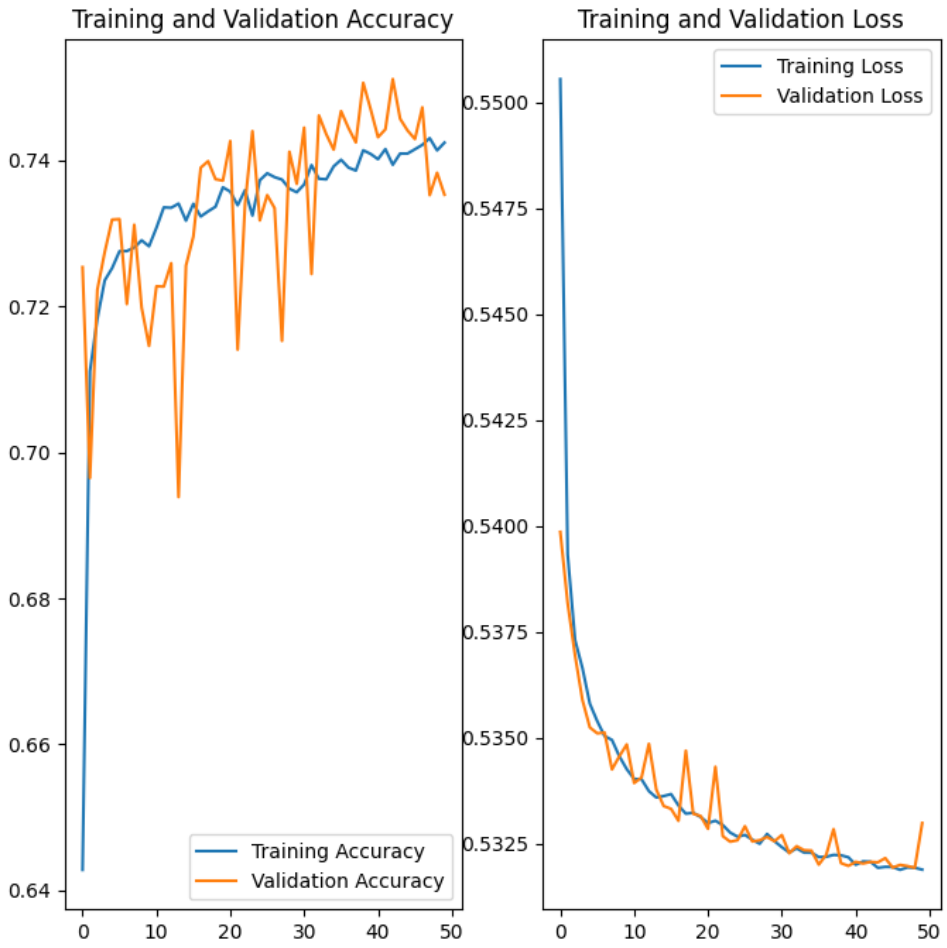

Figure 3.1: First training of the autoencoder

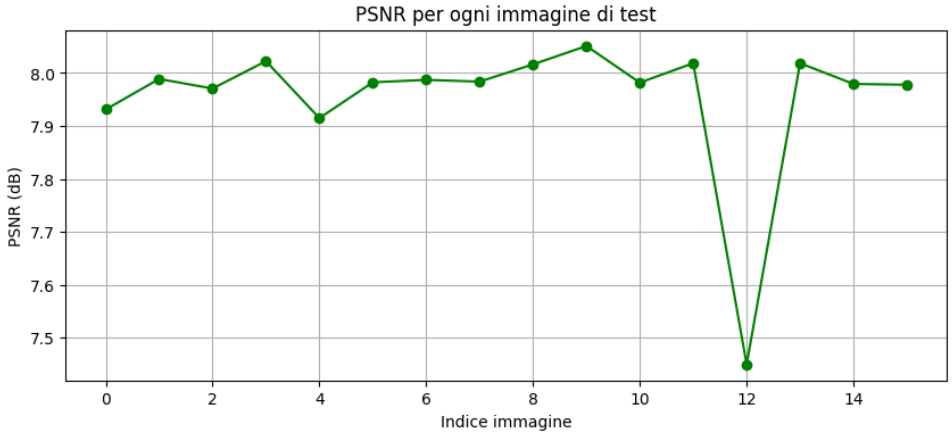

Figure 3.2: First psnr of the autoencoder

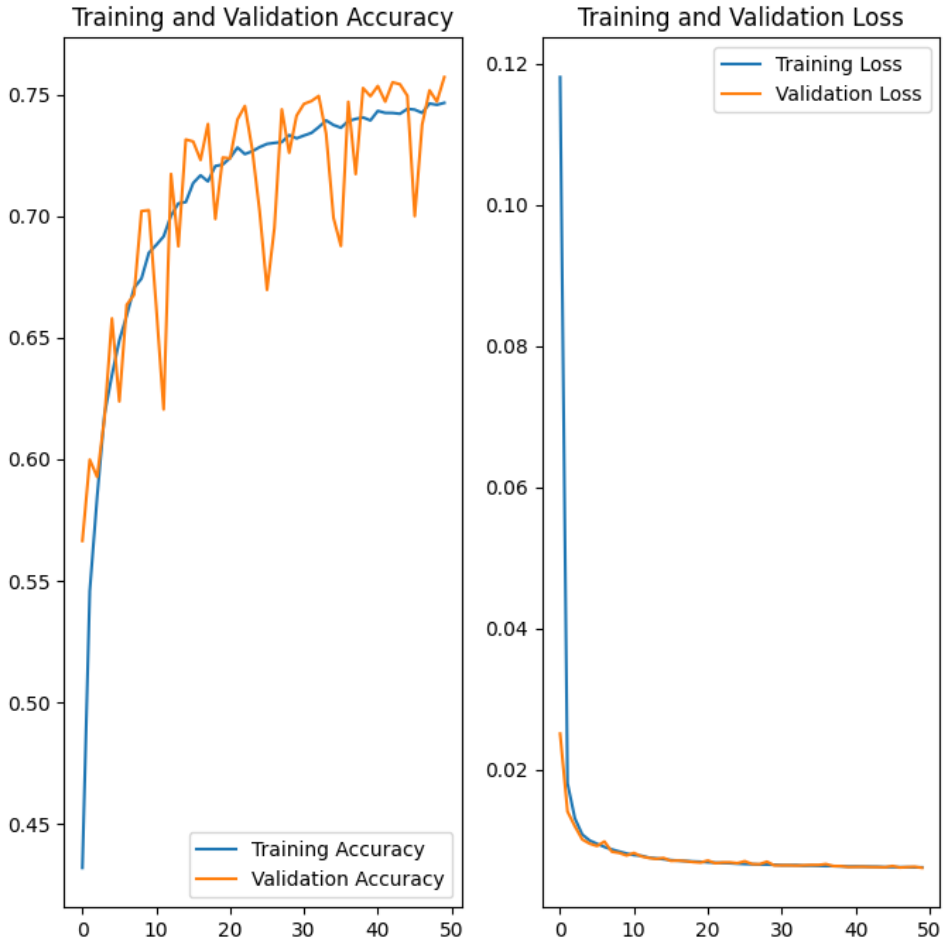

Figure 3.3: First training of the autoencoder

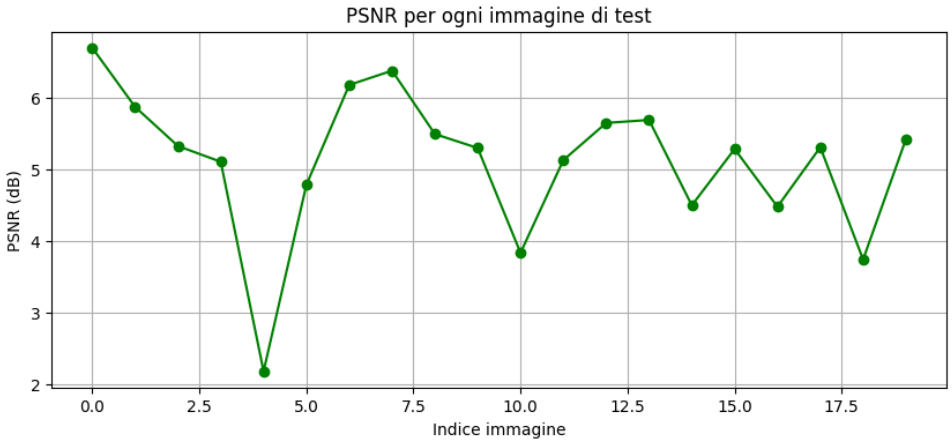

Figure 3.4: First psnr of the autoencoder

4 Bibliografia

- ¡nome¿