**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)

```
def load_dataset(split='train', img_size=(256,256), batch_size=16):
    #Carica il dataset Oxford-IIIT Pet da tfds e applica il #preprocessam

    #Ritorna un dataset in batch.

    # as_supervised=False per mantenere dict
    dataset = tfds.load('oxford_iiit_pet', split=split, as_supervised=Fal
    # Preprocessing
    dataset = dataset.map(lambda sample: preprocess(sample, img_size))
    # Ottimizza il caricamento
    dataset = dataset.shuffle(1024).batch(batch_size).prefetch(tf.data.AU
```

```
12      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

E FAI RUMORE **QUIIIIIIIIIIIIIIIIIIIIIIIIIIII** 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

```
def build_autoencoder(input_shape):

    Costruisce un autoencoder convoluzionale per immagini a colori.
    Parametri:
      input_shape: forma dell'immagine in input (es. (32,32,3))
    Ritorna:
      autoencoder: modello compilato

    input_img = Input(shape=input_shape)

    # Encoder
    x = Conv2D(64, (3,3), activation='relu', padding='same')(input_img)
    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', met

    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=Fa
        super(BasicBlock, self).__init__(**kwargs)
        self.conv = Conv2D(out_channels, kernel_size, strides=stride,
                           padding='same', use_bias=use_bias)
        self.relu = ReLU()

    def call(self, x):
        return self.relu(self.conv(x))

# ResidualBlock: due convoluzioni con skip connection(come lavora cuda co
@register_keras_serializable( ResidualBlock )
class ResidualBlock(Layer):
    def __init__(self, out_channels, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.conv1 = Conv2D(out_channels, 3, strides=1, padding='same')
        self.relu = ReLU()
        self.conv2 = Conv2D(out_channels, 3, strides=1, padding='same')

    def call(self, x):
        residual = self.conv1(x)
        residual = self.relu(residual)
        residual = self.conv2(residual)
        out = Add()([x, residual])
```

```
87          return ReLU()(out)
88
89  # EResidualBlock: versione estesa con convoluzioni a gruppi
90  @register_keras_serializable( EResidualBlock )
91  class EResidualBlock(Layer):
92      def __init__(self, out_channels, groups=1, **kwargs):
93          super(EResidualBlock, self).__init__(**kwargs)
94          self.conv1 = Conv2D(out_channels, 3, strides=1, padding='same', g
95          self.relu = ReLU()
96          self.conv2 = Conv2D(out_channels, 3, strides=1, padding='same', g
97          self.conv3 = Conv2D(out_channels, 1, strides=1, padding='valid')
98
99      def call(self, x):
100         residual = self.conv1(x)
101         residual = self.relu(residual)
102         residual = self.conv2(residual)
103         residual = self.relu(residual)
104         residual = self.conv3(residual)
105         out = Add()([x, residual])
106         return ReLU()(out)
107
108 # Merge_Run_dual: due rami convoluzionali con dilatazioni diverse, poi fu
109 @register_keras_serializable( MergeRunDual )
110 class MergeRunDual(Layer):
111     def __init__(self, out_channels, **kwargs):
112         super(MergeRunDual, self).__init__(**kwargs)
113         # Primo ramo
114         self.conv1a = Conv2D(out_channels, 3, strides=1, padding='same')
115         self.relu1a = ReLU()
116         self.conv1b = Conv2D(out_channels, 3, strides=1, padding='same',
117         self.relu1b = ReLU()
118         # Secondo ramo
119         self.conv2a = Conv2D(out_channels, 3, strides=1, padding='same',
120         self.relu2a = ReLU()
121         self.conv2b = Conv2D(out_channels, 3, strides=1, padding='same',
122         self.relu2b = ReLU()
123         # Gogeta
124         self.conv3 = Conv2D(out_channels, 3, strides=1, padding='same')
125         self.relu3 = ReLU()
126
127     def call(self, x):
128         branch1 = self.relu1a(self.conv1a(x))
129         branch1 = self.relu1b(self.conv1b(branch1))
130
131         branch2 = self.relu2a(self.conv2a(x))
132         branch2 = self.relu2b(self.conv2b(branch2))
133
134         merged = Concatenate()([branch1, branch2])
135         merged = self.relu3(self.conv3(merged))
136         return Add()([merged, x])
```

```
137
138 # CALayer: Channel Attention Layer
139 @register_keras_serializable( CALayer )
140 class CALayer(Layer):
141     def __init__(self, channel, reduction=16, **kwargs):
142         super(CALayer, self).__init__(**kwargs)
143         self.channel = channel
144         self.reduction = reduction
145         self.conv1 = Conv2D(channel // reduction, 1, strides=1, padding='
146         self.relu = ReLU()
147         self.conv2 = Conv2D(channel, 1, strides=1, padding='same', activa
148
149     def call(self, x):
150         # Calcolo della media globale per canale
151         y = tf.reduce_mean(x, axis=[1, 2], keepdims=True)
152         y = self.relu(self.conv1(y))
153         y = self.conv2(y)
154         return Multiply()([x, y])
155
156 # Block: combinazione di MergeRunDual, ResidualBlock, EResidualBlock e CA
157 @register_keras_serializable( Block )
158 class Block(Layer):
159     def __init__(self, out_channels, **kwargs):
160         super(Block, self).__init__(**kwargs)
161         self.merge_run_dual = MergeRunDual(out_channels)
162         self.residual_block = ResidualBlock(out_channels)
163         self.e_residual_block = EResidualBlock(out_channels)
164         self.ca = CALayer(out_channels)
165
166     def call(self, x):
167         r1 = self.merge_run_dual(x)
168         r2 = self.residual_block(r1)
169         r3 = self.e_residual_block(r2)
170         out = self.ca(r3)
171         return out
172
173 # RIDNET: il modello final
174 @register_keras_serializable( RIDNET )
175 def RIDNET(n_feats=64, rgb_range=1.0):
176
177     n_feats: numero di feature channels usate all'interno del modello.<br
178     rgb_range: scala dei valori RGB (ad es. 1.0 se l'input    normalizzat
179
180     rgb_mean = (0.4488, 0.4371, 0.4040)
181
182     input_layer = Input(shape=(None, None, 3))
183
184     # Sottosottrai la media (MeanShift con sign=-1)
185     sub_mean = MeanShift(rgb_mean, sign=-1)
186     x = sub_mean(input_layer)
```

```
187
188     # Testa: BasicBlock (conv + ReLU)
189     head = Conv2D(n_feats, 3, strides=1, padding='same', activation='relu
190
191     # Serie di blocchi
192     b1 = Block(n_feats)(head)
193     b2 = Block(n_feats)(b1)
194     b3 = Block(n_feats)(b2)
195     b4 = Block(n_feats)(b3)
196
197     # Coda: convoluzione finale per ottenere 3 canali
198     tail = Conv2D(3, 3, strides=1, padding='same')(b4)
199
200     # Aggiungi la media (MeanShift con sign=+1)
201     add_mean = MeanShift(rgb_mean, sign=1)
202     res = add_mean(tail)
203
204     # Connessione residua a livello di immagine: somma con l'input origin
205     output = Add()([res, input_layer])
206
207     model = Model(inputs=input_layer, outputs=output)
208     return model
209
210 model = RIDNET(n_feats=32, rgb_range=1.0)
211 model.compile(optimizer='adam', loss='mse', metrics=['accuracy'] )
212 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

### 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 | (None, 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)

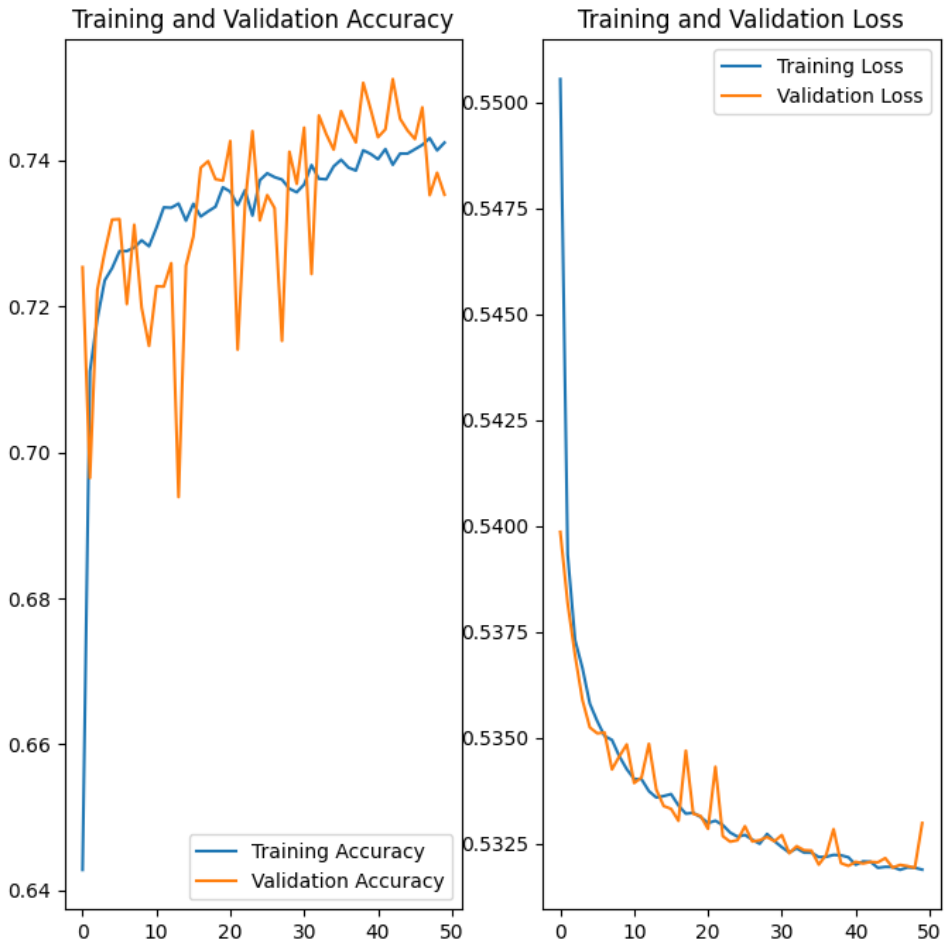


Figure 3.1: First training of the autoencoder

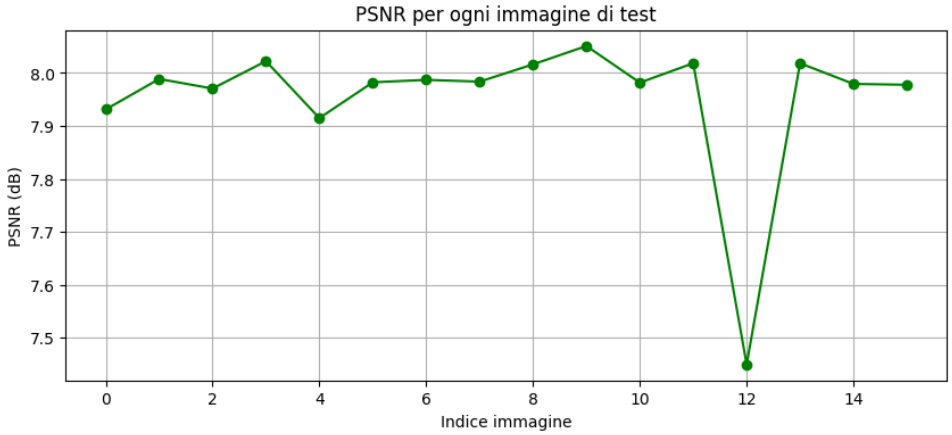


Figure 3.2: First psnr of the autoencoder

### 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

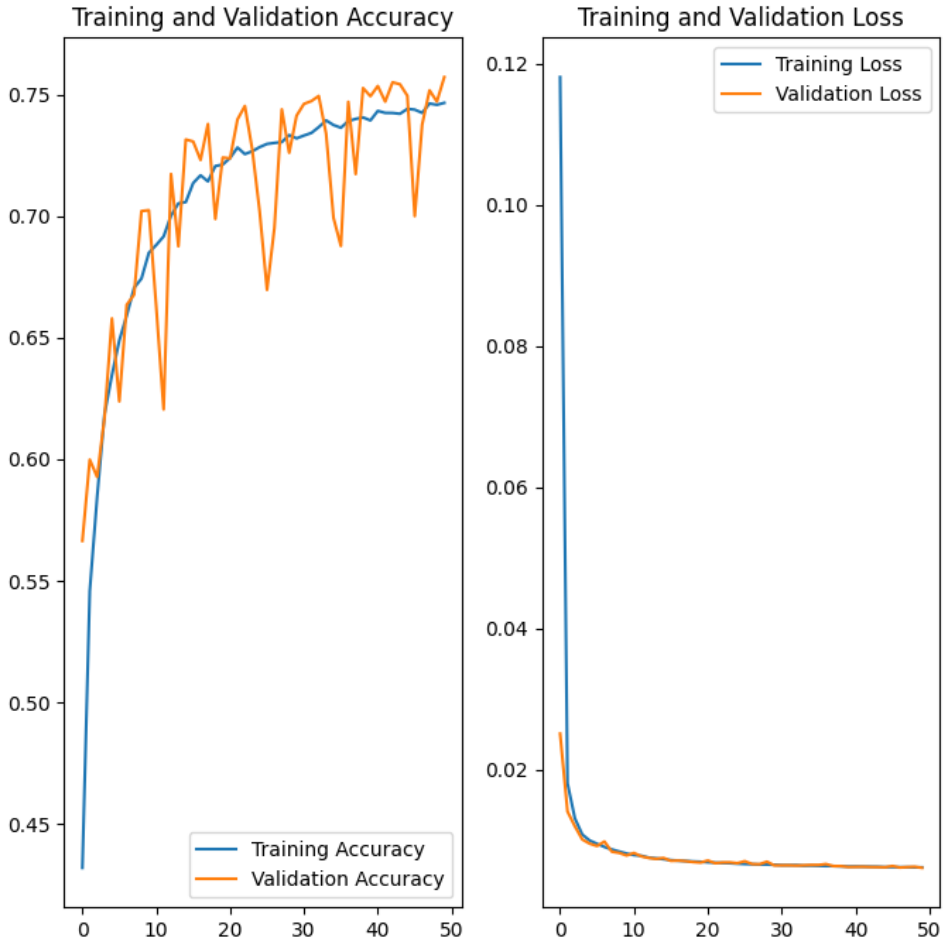


Figure 3.3: First training of the autoencoder

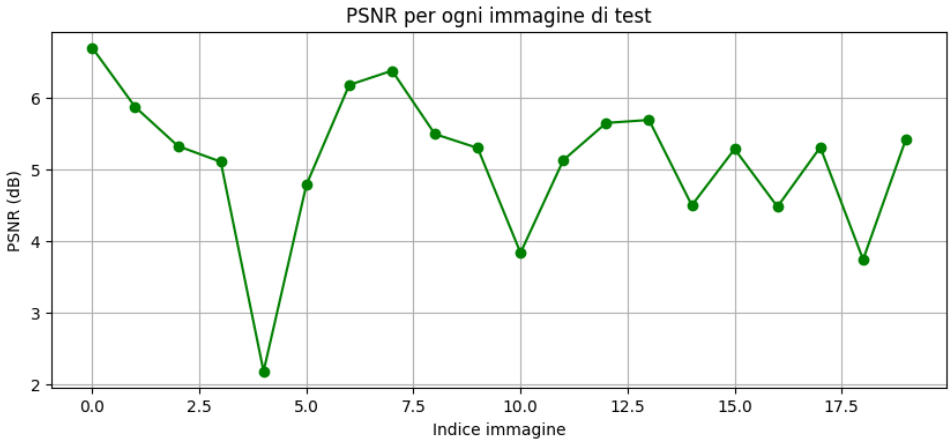


Figure 3.4: First psnr of the autoencoder

# 4 Bibliografia

- ¡nome¿