06_PixelRNN

December 12, 2023

1 Welcome to assignment #6!

Please submit your solution of this notebook in the Whiteboard at the corresponding Assignment entry. We need you to upload the .ipynb-file and the exported .pdf of this notebook.

If you have any questions, ask them in either in the tutorials or in the "Mattermost" channel. The channel is called SSL_WS_2324, you can join the server using this Link and can search for the public channel.

This week we want to to implement a PixelRNN on MNIST.

2 PapagAI

From the second week onwards we started the reflective study. Register on the PapagAI website and write your first reflection about your impressions and challenges in the context of the lectures and tutorials you had this and previous week. The size of reflection can be anywhere bigger than 100 words. You can check out this YouTube video with instructions on how to register, create a reflection and get an ai feedback.

Please note, that this task is an obligatory one for this course and make sure each of you does the reflection, not only one person per group.

Please state both names of your group members here: Authors: Manasi Acharya, Namrata De

3 Assignment 6: PixelRNN

Paper: `https://arxiv.org/pdf/1601.06759.pdf` (Probably) Useful Blogpost: `https://towardsdatascience.com/auto-regressive-generative-models-pixelrnn-pixelcnn-32d192911173`

3.1 Ex. 6.1 Architecture

Pixel Recurrent Neural Networks perform a generative task using a recurrent network architecture. Working on images, we usually work with 2 spatial dimensions (+ color channel dimension in case of colored images). The paper presents 2 different versions of LSTM layers to make it work, but you don't have to. You are free to use any other recurrent layer and any library you want. Whatever works best for you.

Implement your model architecture here. **(RESULT)**

[38]:
```python
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, Dense, LSTM, Reshape

# Define the PixelRNN model
def pixel_rnn_model(input_shape, num_lstm_units=128, num_output_channels=1):
    # Input layer
    input_layer = Input(shape=input_shape, name='input_layer')

    # Convolutional layer to capture spatial dependencies
    conv_layer = Conv2D(filters=num_lstm_units, kernel_size=(1, 1),␣
 ↪padding='same', activation='relu')(input_layer)

    # Reshape the convolutional output to feed into LSTM layer
    reshaped_layer = Reshape((-1, num_lstm_units))(conv_layer)

    # LSTM layer
    lstm_layer = LSTM(units=num_lstm_units,␣
 ↪return_sequences=True)(reshaped_layer)

    # Dense layer to predict pixel values
    output_layer = Dense(units=num_output_channels,␣
 ↪activation='sigmoid')(lstm_layer)

    # Reshape the output to match the input shape
    final_output = Reshape((input_shape[0], input_shape[1],␣
 ↪num_output_channels))(output_layer)

    # Define the model
    model = tf.keras.Model(inputs=input_layer, outputs=final_output,␣
 ↪name='pixel_rnn_model')

    return model

# Example usage
input_shape = (28, 28, 1)  # Adjust based on your image dimensions
pixel_rnn = pixel_rnn_model(input_shape)
pixel_rnn.summary()
```

```
Model: "pixel_rnn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 28, 28, 1)]       0

 conv2d_15 (Conv2D)          (None, 28, 28, 128)       256

 reshape_16 (Reshape)        (None, 784, 128)          0
 lstm_12 (LSTM)              (None, 784, 128)          131584

 dense_9 (Dense)             (None, 784, 1)            129

 reshape_17 (Reshape)        (None, 28, 28, 1)         0

=================================================================
Total params: 131969 (515.50 KB)
Trainable params: 131969 (515.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

3.2 Ex. 6.2 Training on MNIST

Train your model on the MNIST dataset. **(RESULT)**

[39]:
```python
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Add channel dimension (1 for grayscale images)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

# One-hot encode the labels
y_train = x_train  # Just use the input images as labels for pixel-wise␣
 ↪prediction
y_val = x_val
y_test = x_test

# Compile the model
pixel_rnn.compile(optimizer='adam', loss='mse')  # Use mean squared error for␣
 ↪pixel-wise prediction

# Train the model
batch_size = 64
epochs = 5
```

```
history = pixel_rnn.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
↪validation_data=(x_val, y_val))

# Evaluate the model on the test set
test_loss = pixel_rnn.evaluate(x_test, y_test)
print(f'Test Loss: {test_loss}')
```

```
Epoch 1/5
938/938 [==============================] - 814s 866ms/step - loss: 0.0067 -
val_loss: 6.5098e-04
Epoch 2/5
938/938 [==============================] - 796s 848ms/step - loss: 3.8696e-04 -
val_loss: 1.7288e-04
Epoch 3/5
938/938 [==============================] - 1093s 1s/step - loss: 1.0089e-04 -
val_loss: 6.4131e-05
Epoch 4/5
938/938 [==============================] - 787s 838ms/step - loss: 4.7200e-05 -
val_loss: 3.3863e-05
Epoch 5/5
938/938 [==============================] - 737s 786ms/step - loss: 2.5808e-05 -
val_loss: 2.0050e-05
313/313 [==============================] - 44s 141ms/step - loss: 2.0153e-05
Test Loss: 2.0153369405306876e-05
```

3.3 Ex. 6.3 Complete some samples

Using your trained model, are you able to complete MNIST samples that are partially masked? Show your best results and compare them to the original. **(RESULT)**

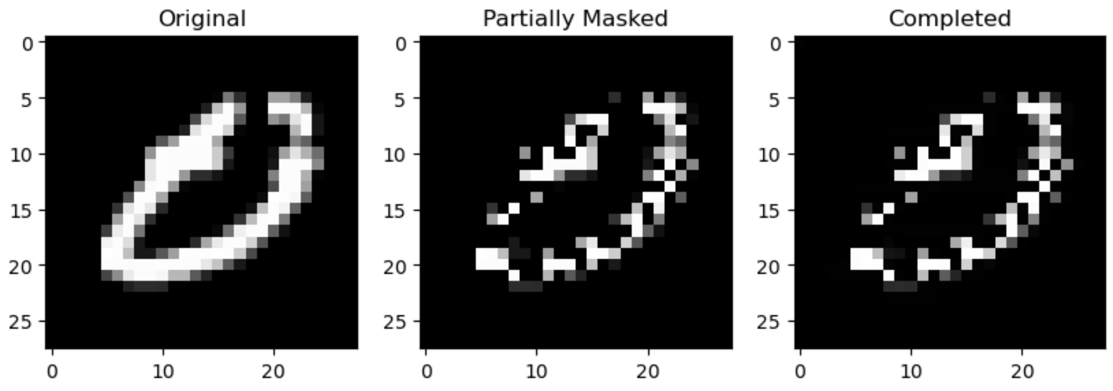
```
[40]: # pixel_rnn.load_weights(model_path)

import numpy as np
import matplotlib.pyplot as plt

# Function to create partially masked inputs
def create_partially_masked_inputs(images, mask_fraction=0.5):
    masked_images = np.copy(images)
    num_masked_pixels = int(mask_fraction * images.size)
    mask_indices = np.random.choice(images.size, num_masked_pixels,␣
↪replace=False)
    masked_images.flat[mask_indices] = 0  # Mask the pixels by setting them to 0
    return masked_images


# Choose some images from the test set for completion
num_samples = 5
```

```
sample_indices = np.random.choice(len(x_test), num_samples, replace=False)
sample_images = x_test[sample_indices]

# Create partially masked inputs
masked_images = create_partially_masked_inputs(sample_images)

# Generate completed samples using the model
completed_samples = pixel_rnn.predict(masked_images)
```

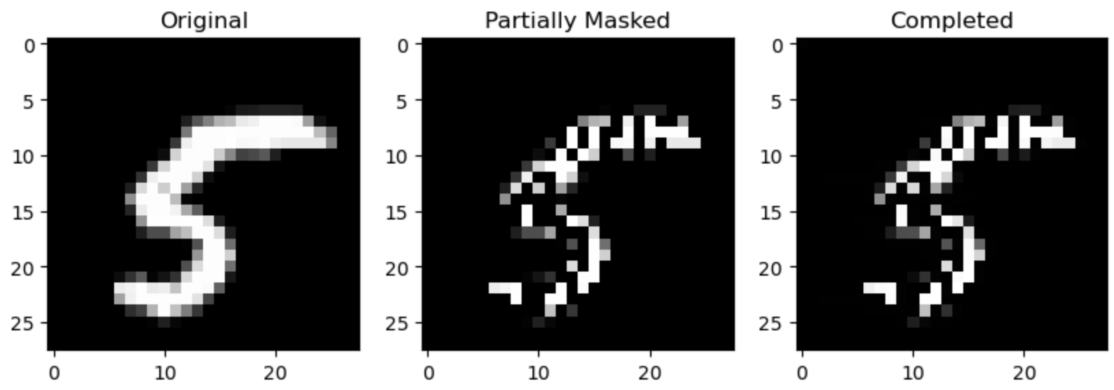
```
1/1 [==============================] - 0s 447ms/step
```

[41]:
```
# Plot the original, masked, and completed samples
for i in range(num_samples):
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 3, 1)
    plt.imshow(sample_images[i, :, :, 0], cmap='gray')
    plt.title('Original')

    plt.subplot(1, 3, 2)
    plt.imshow(masked_images[i, :, :, 0], cmap='gray')
    plt.title('Partially Masked')

    plt.subplot(1, 3, 3)
    completed_image = completed_samples[i]
    print(completed_image.shape)
    plt.imshow(completed_samples[i, :, :, 0], cmap='gray')
    plt.title('Completed')

    plt.show()
```

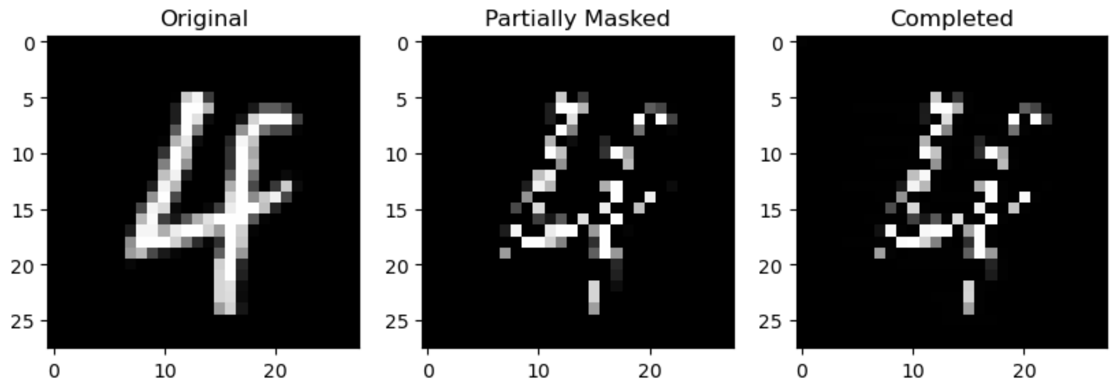
```
(28, 28, 1)
```



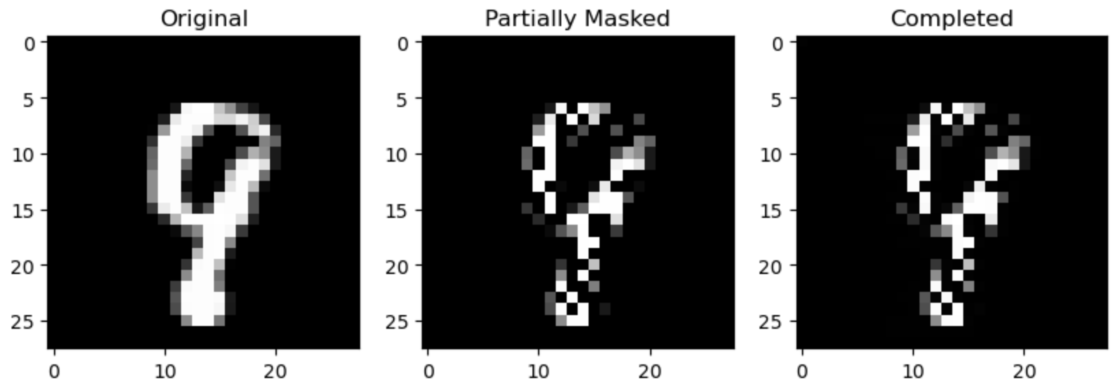
```
(28, 28, 1)
```



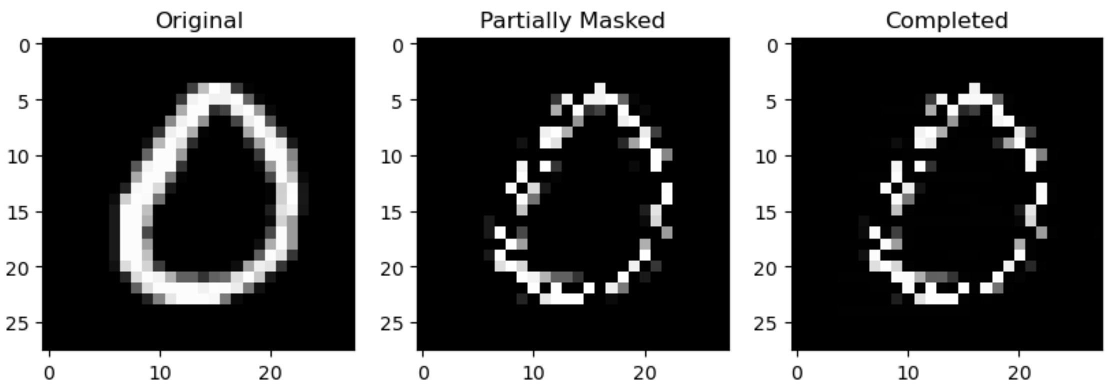
```
(28, 28, 1)
```



```
(28, 28, 1)
```



```
(28, 28, 1)
```


Original
Partially Masked
Completed
0
5
10
15
20
25
0
10
20

```
[ ]:
```