Lab report – TNM112

Deep learning for media technology, Lab 3

Emma Zettervall (emmze999)

Sunday 29th December, 2024 (12:15)

Abstract

This report explores the implementation of an inpainting model using a U-Net-based convolutional neural network (CNN). The model is trained and evaluated on the MNIST and CIFAR-10 datasets, with performance measured using the Structural Similarity Index Measure (SSIM). Results show that the model achieves an average SSIM of 0.8795 on MNIST and 0.8546 on CIFAR-10, demonstrating its effectiveness in reconstructing missing image regions.

1 Introduction

This lab offered a choice of several mini projects, each focusing on different AI techniques. For this report, inpainting was selected and implemented. Inpainting is a process where missing pixel values in an image are filled in, often with the help of AI. For this project, a convolutional neural network (CNN) was used to perform the inpainting task.

The model was tested on two datasets: MNIST, which consists of grayscale images of handwritten digits, and CIFAR-10, which contains color images of objects. The CNN used in this project is an image-to-image network, meaning it takes a masked image as input and attempts to reconstruct the missing parts to match the original image.

2 Background

Traditionally, inpainting relied on methods like patch-based or diffusion-based techniques, which often required manual input or strong assumptions about the image [1]. However, with the rise of deep learning, CNNs have become the preferred method for inpainting. CNNs excel at learning patterns directly from data, making them highly effective for generating realistic reconstructions [2].

For this project, a U-Net-inspired CNN architecture was chosen. U-Net is a popular choice for image-to-image tasks like inpainting because it captures both large-scale patterns and fine details using skip connections. Skip connections allow high-resolution spatial information from the encoder to flow directly to the decoder, improving reconstruction quality [3].

For evaluating, the Structural Similarity Index Measure (SSIM) is suitable.

SSIM is a perceptual metric that quantifies the similarity between two images by evaluating luminance, contrast, and structure. Unlike pixel-wise metrics like Mean Squared Error (MSE), SSIM aligns more closely with human visual perception, making it ideal for assessing image reconstruction tasks [4].

3 Method

This section describes the implementation of the inpainting pipeline developed in Python using Keras.

3.1 Data Preparation

To train the model, a masking function was implemented to remove pixels from the center of input images. The file **data_generator.py** from previous labs was copied and the **mask_image** function was added. The masked images were then used as input to the network, while the original images served as the ground truth. The masking function is shown below:

```
def mask_images(self, mask_size=8):
    self.masked_x_train = self.x_train.copy()
    self.masked_x_valid = self.x_valid.copy()
    self.masked_x_test = self.x_test.copy()

    def mask_center(img):
        h, w = img.shape[:2]
        start = h // 2 - mask_size // 2
        img[start:start + mask_size, start:start + mask_size] = -1
        return img

    for i in range(self.masked_x_train.shape[0]):
        self.masked_x_train[i] = mask_center(self.masked_x_train[i
    ])
    for i in range(self.masked_x_valid.shape[0]):
        self.masked_x_valid[i] = mask_center(self.masked_x_valid[i
    ])
    for i in range(self.masked_x_test.shape[0]):
        self.masked_x_test[i] = mask_center(self.masked_x_test[i])
```

Next, the data for either MNIST or CIFAR-10 was generated and masked, using 1000 training samples:

```
data_gen = data_generator3.DataGenerator()
dataset_type = 'mnist'  # 'mnist' or 'cifar10'
data_gen.generate(dataset=dataset_type, N_train=1000)
data_gen.mask_images(mask_size=8)
```

3.2 Model Architecture

A U-Net-based autoencoder was implemented. The architecture consisted of three main parts:

- **Encoder**: The encoder included convolutional layers to extract features from the input image, followed by pooling layers to reduce spatial dimensions. Dropout layers were added to the encoder to prevent overfitting by randomly deactivating certain neurons during training, which encouraged the model to generalize better.

- **Bottleneck**: This stage used convolutional layers to capture high-level abstract features from the input image. The bottleneck acted as the model's compressed representation of the data, focusing on essential features while discarding extraneous information.
- **Decoder**: The decoder reconstructed the images using upsampling and transposed convolutional layers. Skip connections were employed to transfer information directly from the encoder to the decoder. These connections helped preserve fine-grained details and spatial accuracy.

To improve the model's generalization capability, data augmentation techniques such as random rotation and zooming were applied to the input images during training.

The architecture was designed to accommodate datasets with varying input sizes by using the input shape as a parameter. The convolutional layers employed the ReLU activation function, which introduced non-linearity and helped the model learn complex patterns. The output layer used the tanh activation function to ensure the pixel values of the reconstructed image lay within the desired range (-1 to 1), matching the normalized input data.

The model was compiled using the Adam optimizer. The loss function used was Mean Squared Error (MSE), which measured the average squared differences between the predicted and true pixel values. MSE is equivalent to the L2 loss but normalizes the result by dividing by the number of pixels, making it more interpretable in image reconstruction tasks.
The U-Net-based autoencoder is shown below:

```
def build_unet_autoencoder(input_shape):
    input_img = Input(shape=input_shape)

    # Data augmentation
    x = RandomRotation(0.1)(input_img)
    x = RandomZoom(0.1)(x)

    # Encoder
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(c1)
    c1 = Dropout(0.2)(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(p1)
    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(c2)
    c2 = Dropout(0.2)(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    # Bottleneck
    b = Conv2D(128, (3, 3), activation='relu', padding='same')(p2)

    # Decoder with skip connections
    u2 = UpSampling2D((2, 2))(b)
    u2 = Conv2DTranspose(64, (3, 3), activation='relu', padding='
    same')(u2)
    u2 = Concatenate()([u2, c2])
    u2 = Conv2D(64, (3, 3), activation='relu', padding='same')(u2)

    u1 = UpSampling2D((2, 2))(u2)
```

```
    u1 = Conv2DTranspose(32, (3, 3), activation='relu', padding='
    same')(u1)
    u1 = Concatenate()([u1, c1])
    u1 = Conv2D(32, (3, 3), activation='relu', padding='same')(u1)

    # Output layer dynamically matches the input channels
    decoded = Conv2D(input_shape[-1], (3, 3), activation='tanh',
    padding='same')(u1)
    model = Model(inputs=input_img, outputs=decoded)
    model.compile(optimizer='adam', loss='mse')
    return model
```

3.3 Training

After experimenting with dropout layers, skip connections, epochs, and batch size, the final model was trained on 1000 samples from the MNIST and CIFAR-10 datasets. Key hyperparameters included 100 epochs, a batch size of 64, the Mean Squared Error (MSE) as the loss function, and the Adam optimizer, as below.

```
input_shape = data_gen.x_train.shape[1:]
autoencoder = build_unet_autoencoder(input_shape)
autoencoder.summary()

history = autoencoder.fit(
    data_gen.masked_x_train, data_gen.x_train,
    validation_data=(data_gen.masked_x_valid, data_gen.x_valid),
    epochs=100, batch_size=64
)
```

3.4 Evaluation

The model's accuracy was evaluated using the Structural Similarity Index Measure (SSIM) between the reconstructed and original images using the following function:

```
def calculate_ssim(original, predicted):
    ssim = tf.image.ssim(original, predicted, max_val=2.0)
    ssim_normalized = (ssim + 1) / 2  # Normalize SSIM from [-1, 1]
     to [0, 1]
    return ssim_normalized.numpy()
```

The average SSIM score across all test images was calculated:

```
all_ssim = []
for i in range(data_gen.x_test.shape[0]):
    original_tensor = tf.convert_to_tensor(np.expand_dims(data_gen.
    x_test[i], axis=0), dtype=tf.float32)
    predicted_tensor = tf.convert_to_tensor(np.expand_dims(
    autoencoder.predict(np.expand_dims(data_gen.masked_x_test[i],
    axis=0)), axis=0), dtype=tf.float32)
    ssim_acc = calculate_ssim(original_tensor, predicted_tensor)[0]
    all_ssim.append(ssim_acc)
print("Average SSIM for all test images: ", np.mean(all_ssim))
```

To visualize the results, the first five images from the test set were plotted alongside their masked inputs and reconstructed outputs. The SSIM for each reconstructed image was also displayed:

```python
# Plotting function
def plot_results(original, masked, predicted, num_images=5):
    plt.figure(figsize=(10, 5))
    for i in range(num_images):
        # Expand dimensions for SSIM calculation
        original_tensor = tf.convert_to_tensor(np.expand_dims(original[i], axis=0), dtype=tf.float32)
        predicted_tensor = tf.convert_to_tensor(np.expand_dims(predicted[i], axis=0), dtype=tf.float32)

        # Calculate SSIM accuracy
        ssim_acc = calculate_ssim(original_tensor, predicted_tensor)[0]
        # Original
        plt.subplot(3, num_images, i+1)
        plt.imshow((original[i] + 1) / 2, cmap='gray')
        plt.title("Original")
        plt.axis('off')

        # Masked
        plt.subplot(3, num_images, i+1+num_images)
        plt.imshow((masked[i] + 1) / 2, cmap='gray')
        plt.title("Masked")
        plt.axis('off')

        # Reconstructed
        plt.subplot(3, num_images, i+1+2*num_images)
        plt.imshow((predicted[i] + 1) / 2, cmap='gray')
        plt.title(f"Reconstructed\nSSIM: {ssim_acc:.4f}")
        plt.axis('off')

    plt.tight_layout()
    plt.show()

# Predict and visualize the first 5 test images
predicted = autoencoder.predict(data_gen.masked_x_test[:5])
plot_results(data_gen.x_test[:5], data_gen.masked_x_test[:5], predicted)
```

4 Results

4.1 MNIST

The model performed well on MNIST, achieving an average SSIM of 0.8795 and visually accurate reconstructions of the masked regions. The reconstructions maintained the structure of the handwritten digits, effectively filling in the missing pixel values. An example of the results is shown in Figure 1.

4.2 CIFAR-10

On CIFAR-10, the model achieved an average SSIM of 0.8546. While the reconstructions were less accurate compared to MNIST, the model was able to restore general features and colors of the masked regions. The increased complexity of CIFAR-10 images, which include diverse objects and colors, made a greater challenge for the network. An example of the results is shown in Figure 2.

Figure 1: Results on MNIST dataset. Original, masked, and reconstructed images and SSIM-values are shown.

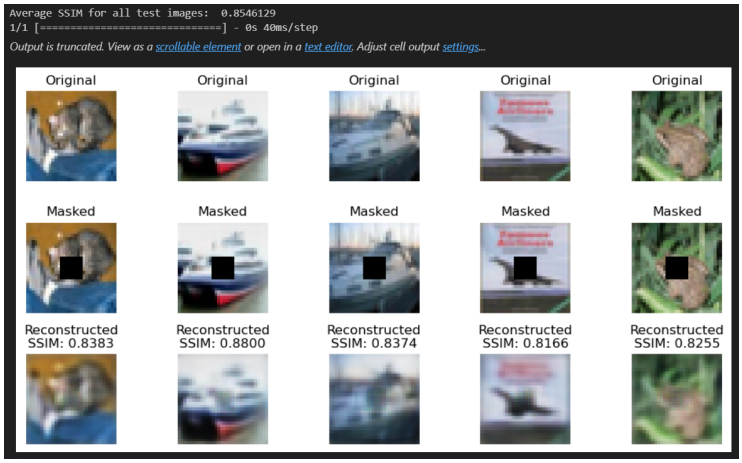

Figure 2: Result on CIFAR-10 dataset. Original, masked, and reconstructed images and SSIM-values are shown.

5 Conclusion

- The U-Net-based autoencoder successfully performed inpainting on both MNIST and CIFAR-10 datasets, achieving high SSIM scores for reconstructed images.
- Results on MNIST were more accurate due to the simpler nature of the dataset compared to CIFAR-10.
- Data augmentation and dropout were effective in improving the model's generalization, particularly on unseen validation data.
- SSIM and MSE were used to evaluate the model's performance. SSIM

measured how similar the reconstructed images were to the originals in terms of visual quality, while MSE measured the accuracy of individual pixel values. Together, these metrics provided a clear understanding of how well the model performed.

Use of generative AI

The abstract, background and conclusion was written with the help of ai, aswell as some rephrasing in other parts of the report. The functions in the code are made together with ai.

References

[1] Marcelo Bertalmío, Guillermo Sapiro, Vicent Caselles, and Coloma Ballester. Image inpainting. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 417–424. ACM Press/Addison-Wesley Publishing Co., 2000.

[2] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

[3] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241. Springer, 2015.

[4] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.