CNN Assignment Report

Authors:

Abdelmajid Benjelloun

20000203

H130407540

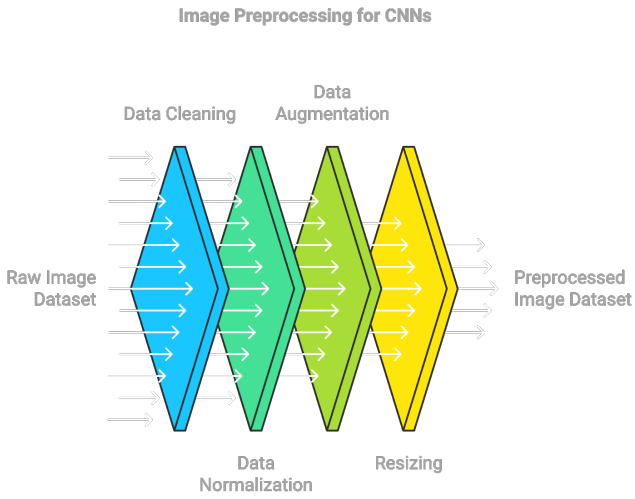
Supervisor:

Pr. ....

Contents

1 Questions for Custom CNN

1.1 What are the steps for data preprocessing to build a custom CNN?



Data preprocessing is crucial in building a Convolutional Neural Network (CNN) to ensure the data is clean, normalized, and ready for efficient training. The main steps for data preprocessing include:

- **Data Cleaning:** Removing or handling missing, incomplete, or corrupted data to maintain consistency.
- **Data Normalization:** Scaling pixel values, usually to a range between 0 and 1 or -1 and 1. This can be mathematically represented as:

$$x_{\text{normalized}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

  where x represents the original pixel value, and $\min(x)$ and $\max(x)$ are the minimum and maximum pixel values, respectively.

- **Data Augmentation:** Applying transformations, such as rotation, flipping, and scaling, to artificially increase dataset diversity. For example, a rotation transformation on an image I by angle θ can be represented as:

$$I' = R(\theta) \cdot I$$

  where $R(\theta)$ is the rotation matrix.

- **Resizing:** Ensuring that all images are resized to a consistent input size, such as 224×224 or 32×32, to maintain uniform input dimensions for the CNN.

1.2 How do we build the different layers of a custom CNN?

Constructing a custom CNN involves defining several types of layers, each responsible for specific operations:

- **Convolutional Layers:** These layers apply a set of filters (kernels) to the input data. The convolution operation between an input X and a kernel K can be represented as:

$$(X * K)(i, j) = \sum_m \sum_n X(i + m, j + n) \cdot K(m, n)$$

  where (i, j) denotes the position in the output feature map.

- **Activation Layers:** Non-linear activation functions like ReLU (Rectified Linear Unit) are applied. ReLU is defined as:

$$f(x) = \max(0, x)$$

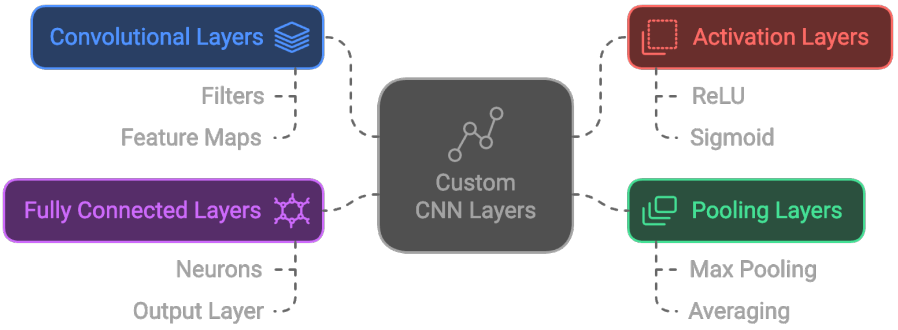
  This introduces non-linearity, enabling the model to learn complex patterns.

- **Pooling Layers:** Max-pooling or average-pooling layers downsample the feature maps. For max-pooling with a 2×2 filter, the output at each position is:

$$\text{max-pool}(i, j) = \max(X_{i:i+2, j:j+2})$$

- **Fully Connected Layers:** These layers connect every neuron in the previous layer to each neuron in the current layer, enabling the model to make final predictions.

- **Dropout Layers:** These layers randomly drop a fraction p of neurons during training to reduce overfitting. If $p = 0.5$, then approximately half of the neurons are randomly dropped.



1.3 Which hyperparameters are critical in a custom CNN?

Critical hyperparameters in a custom CNN directly influence the model's performance and include:

- **Learning Rate:** Controls the step size for updating weights during training. A typical gradient update can be expressed as:

$$w_{t+1} = w_t - \alpha \cdot \nabla L(w_t)$$

  where w represents the weights, α is the learning rate, and $\nabla L(w_t)$ is the gradient of the loss L.

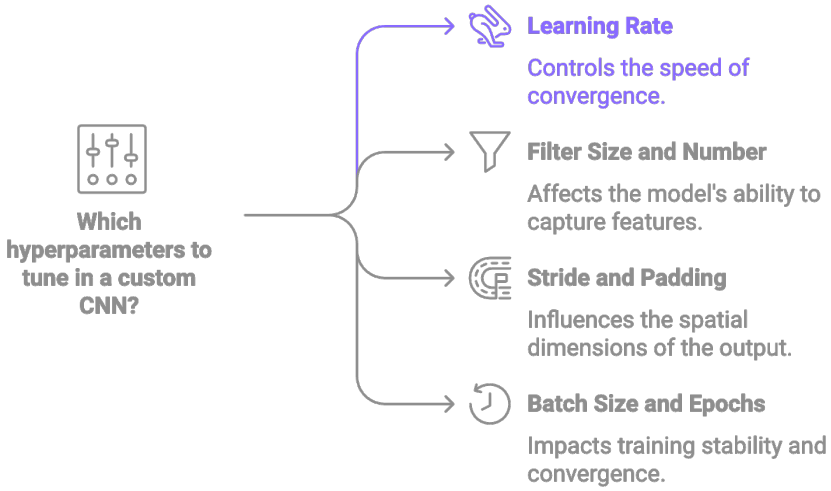
- **Batch Size:** Determines the number of samples processed before updating weights. Larger batch sizes allow for more stable gradient estimates, but smaller batches allow faster updates.

- **Number of Filters:** The number of filters in each convolutional layer impacts the model's capacity to learn features, as each filter captures a unique aspect of the input.

- **Kernel Size:** The size of the filters used in convolutional layers, often represented as 3×3 or 5×5. A kernel size of 3×3 is common due to its efficiency in capturing spatial patterns.

- **Number of Epochs:** Defines the total number of complete passes through the training data.

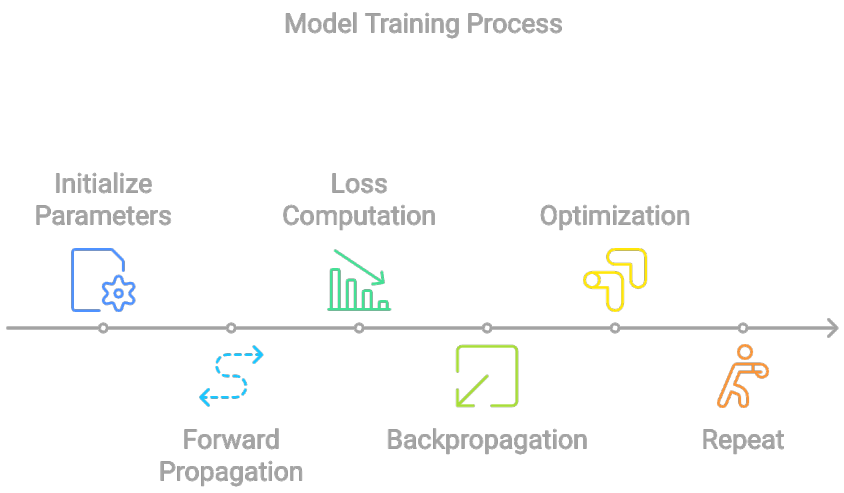
- **Dropout Rate:** Controls the proportion of neurons randomly dropped in dropout layers. For example, a dropout rate $p = 0.5$ will drop half of the neurons.



1.4 How is the model trained from scratch?

Training a CNN from scratch involves several steps:

1. **Initialization:** Initialize weights, commonly using Xavier or He initialization for stability.
2. **Forward Pass:** Compute the output by passing input data through each layer.
3. **Loss Calculation:** Calculate the loss, often cross-entropy for classification tasks:
$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$
where y_c is the true label and $\hat{y}_c$ is the predicted probability for class c.
4. **Backward Pass (Backpropagation):** Compute the gradients of L with respect to each parameter using the chain rule.
5. **Weight Update:** Adjust weights using an optimizer (e.g., SGD, Adam), iteratively reducing the loss.
6. **Repeat:** Iterate through the training process for multiple epochs.
7. *Each step in the training process is designed to ensure that the model effectively learns the underlying patterns present in the data. By meticulously adjusting the model's parameters and continually refining its predictions through feedback, the network becomes increasingly adept at recognizing complex features and relationships within the dataset. This iterative learning process is crucial for improving the model's overall prediction accuracy, as it allows the network to adapt and generalize better to unseen data.*



1.5 How is a custom CNN evaluated?

To evaluate a custom CNN, the following metrics and processes are typically used:

- **Accuracy:** Measures the fraction of correct predictions.
- **Confusion Matrix:** Helps analyze classification performance by showing true positives, false positives, true negatives, and false negatives.
- **Precision, Recall, and F1 Score:** For imbalanced data, these metrics are crucial. For instance, precision is:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

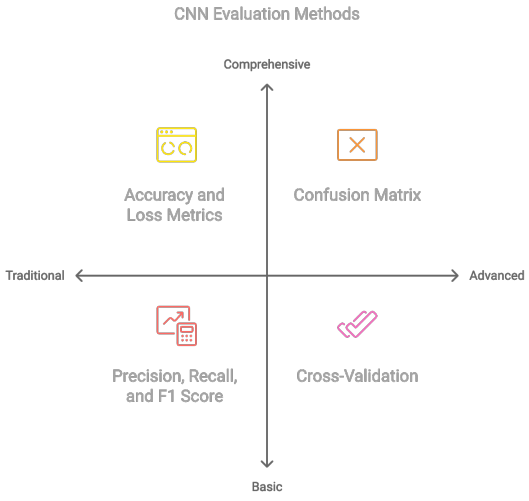
and recall is:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F1 Score is the harmonic mean of precision and recall:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Loss Curve:** By plotting the training and validation loss over epochs, we can visualize the model's learning process.
- **Test Set Evaluation:** The final test set provides an estimate of the model's generalization to unseen data.



2 Exo 1: Image Prediction Using a Pre-Trained Model on ImageNet

ImageNet is a database of over ten million labeled images, produced by the organization of the same name, aimed at research in computer vision.

2.1 Question 01: How to load a pre-trained model on ImageNet with TensorFlow or PyTorch?

To load a pre-trained model on ImageNet, you can use either TensorFlow or PyTorch, which offer predefined functions to access these models.

TensorFlow:

In TensorFlow, you can load a pre-trained model as follows:

This command loads the VGG16 model with weights pre-trained on ImageNet, allowing you to utilize the features extracted by this model in your own application.

PyTorch:

In PyTorch, the process is similar:

```
import torchvision.modet as models
model = models.resnet50(pretrained=True)
```

Here, 'resnet50' is loaded with weights pre-trained on ImageNet, enabling you to benefit from the knowledge accumulated by this model.

2.2 Question: Why is it necessary to resize images before using them with a pre-trained model?

Resizing images is essential because pre-trained models on ImageNet were trained with specific image sizes (typically 224x224 pixels for many models). Reasons for resizing images include:

- **Compatibility:** The input image dimensions must match those expected by the model. For example, if the model expects an image of 224×224, providing an image of a different size will result in an error.
- **Normalization of Features:** By resizing all images to the same size, we ensure that the features extracted during training and prediction are consistent.

2.3 Question: What is the importance of normalizing images before passing them into the pre-trained model?

Normalizing images is crucial for several reasons:

- **Value Scaling:** Pixel values of images should be normalized to avoid scale issues that can affect model performance. For example, if pixel values range from 0 to 255, we can normalize them by dividing by 255, bringing them into the range $[0, 1]$.
- **Distribution Centers:** Additionally, many models pre-trained on ImageNet use specific mean and standard deviation values for normalization. For example, for VGG16 and ResNet, normalization values often include:

$$\text{mean} = [0.485, 0.456, 0.406], \quad \text{std} = [0.229, 0.224, 0.225]$$

This centers the data around zero and ensures that pixel values are within an appropriate range for model training.

2.4 Question: What steps should be followed to pass an image through a pre-trained model to obtain predictions?

The typical steps to obtain predictions from an image using a pre-trained model include:

1. **Load the image:** Use a library like PIL or OpenCV to read the image from disk.
2. **Resize the image:** Adjust the image to the size required by the model, often 224×224 pixels.
3. **Normalize the image:** Apply normalization as described above. For example, to normalize using TensorFlow:

```
from tensorflow.keras.preprocessing import image
import numpy as np

img = image.load_img('path_to_image.jpg', target_size
=(224, 224))
img_array = image.img_to_array(img) / 255.0  # Normalize
to [0, 1]
img_array = (img_array - np.array([0.485, 0.456, 0.406]))
/ np.array([0.229, 0.224, 0.225])  # Standard
normalization
img_array = np.expand_dims(img_array, axis=0)  # Add batch
 dimension
```

4. **Pass the image into the model:** Use the 'predict' method to obtain predictions.

2.5 Question: How to interpret the prediction results of a pre-trained model on ImageNet?

The prediction results of a pre-trained model on ImageNet are usually provided as probabilities for each class. To interpret these results, you can:

- **Predicted class:** Identify the class with the highest probability using the 'argmax' function. For instance, if the model returns a probability vector $[0.1, 0.3, 0.6]$, the predicted class is the one associated with 0.6.
- **Use a class mapping dictionary:** Map the class indices to class names to understand what each prediction means. The ImageNet classes are often available in a mapping file.
- **Visualize scores:** You can also display the probability scores for the top-N classes to get a better idea of other relevant classes. This can be done using a visualization library like Matplotlib.

2.6 Question: Why use a pre-trained model on ImageNet for image prediction instead of building a model from scratch?

Using a pre-trained model offers several advantages, including:

- **Time-saving:** By using a pre-trained model, you save a significant amount of time that would have been needed to train a model from scratch, especially if you have a limited dataset.
- **Performance improvement:** Pre-trained models often have superior performance because they have been trained on a vast amount of varied data.
- **Learning rich features:** These models capture very general features that can be transferred to other tasks, which is crucial when working with smaller datasets.

2.7 Question: What are the steps to fine-tune a pre-trained model on ImageNet for a new classification task? (TL)

Fine-tuning a pre-trained model for a new classification task involves several steps:

1. **Load the pre-trained model:** As mentioned, load a pre-trained model and remove the last layer to adapt the model for your new task.
2. **Add new layers:** Add layers suitable for your task. For example, you might add a Dense layer with a number of neurons corresponding to the number of classes in your task.

3. **Freeze some layers:** You can freeze certain layers to retain the learned weights. This is done by setting ‘layer.trainable = False‘ for the layers you wish to freeze.

4. **Compile the model:** Use a new loss function, such as cross-entropy for classification tasks:

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

   where y_c is the true label and $\hat{y}_c$ is the predicted probability for class c.

5. **Train the model:** Train the model on your new dataset while monitoring performance on a validation set to avoid overfitting.

3 Image Classification with ResNet18

In this subsection, we detail the process of classifying images using a pre-trained ResNet18 model. The model is employed to predict the classes of a set of images from a specified directory.

3.1 Implementation Steps

The following steps outline the implementation:

- Load the pre-trained ResNet18 model.
- Prepare the images by applying necessary transformations.
- Implement a function to make predictions on the images.
- Loop through each image in the directory, process it, and display the predicted labels.

3.2 Code Implementation

Here is the code used for the image classification:

```
import torch
from torchvision import models, transforms
from PIL import Image
import matplotlib.pyplot as plt
import os
import json

# Load the Pre-trained Model
model = models.resnet18(pretrained=True)
model.eval()  # Set the model to evaluation mode

# Load the Label Mapping
```

```
with open("labels.json", "r") as f:
    labels = json.load(f)

# Define Preprocessing Transformations
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225]),
])

# Make Prediction Function
def predict(model, image_tensor):
    with torch.no_grad():
        outputs = model(image_tensor)
    _, predicted_class = outputs.max(1)
    return predicted_class.item()

# Loop Through Each Image in the Folder
image_folder_path = "path/to/images"
for image_name in os.listdir(image_folder_path):
    image_path = os.path.join(image_folder_path, image_name)

    if os.path.isfile(image_path) and image_name.lower().endswith((
    '.png', '.jpg', '.jpeg')):
        img = Image.open(image_path).convert("RGB")
        img_tensor = preprocess(img).unsqueeze(0)
        predicted_index = predict(model, img_tensor)
        predicted_label = labels[predicted_index] if
    predicted_index < len(labels) else "Unknown"

        # Display the image and prediction
        plt.imshow(img)
        plt.title(f"Predicted Class: {predicted_label}")
        plt.axis('off')
        plt.show()
```

Listing 1: Image Classification Code

3.3 Output Results

The following images show the results of the predictions made by the model on the input images:

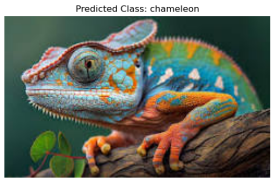


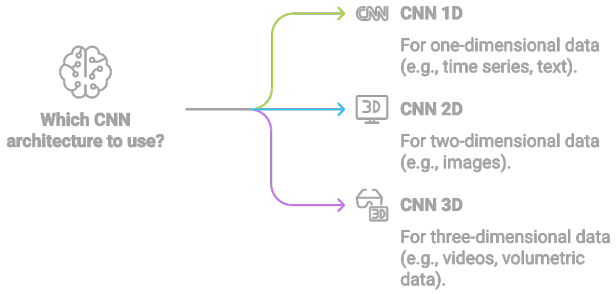
Figure 1: Predicted Class: Chameleon

Figure 2: Predicted Class: Sport Car

Figure 3: Predicted Class: Fox Squirrel

4 Exercice 3 : CNN 1D,2D, and 3D

4.1 Main Difference Between CNN 1D, 2D, and 3D



1D Convolutional Neural Networks: These networks are specifically designed for one-dimensional data, such as time series or text sequences. In CNN 1D, convolution operations are applied along a single axis, enabling the model to capture temporal or sequential patterns effectively.

2D Convolutional Neural Networks: CNNs of this type are utilized for processing images or data that are structured in two dimensions. In 2D CNNs, convolution operations are performed over two axes—height and width—allowing the model to extract spatial features and patterns within the images.

3D Convolutional Neural Networks: These networks are employed for volumetric data or video sequences, where convolution operations are applied across three axes—height, width, and depth. CNN 3D is adept at capturing spatial and temporal information, making it ideal for tasks such as action recognition in videos or processing 3D medical imaging data.

4.2 How a Convolution Layer Works in a CNN 1D

In a CNN 1D, a convolution layer applies a filter (or kernel) that slides along the one-dimensional input. Each position of the filter produces a value by multiplying the filter elements with the corresponding elements in the input region. The result is a feature map that captures the patterns present in the input sequence. The output size is determined by the filter size, stride, and padding.

4.3 When and Why to Use a CNN 3D

A CNN 3D is used when it is necessary to capture information in volumetric data, such as videos, medical images (like MRIs), or 3D simulations. It is

particularly useful when spatial and temporal relationships need to be considered simultaneously, for example, for action classification in video sequences or volume analysis in medical data.

4.4 Examples of 3D Data

An example of 3D data is a video, which can be represented as a sequence of successive 2D images with a temporal dimension. Another example is a volume of medical images, such as an MRI scan, which contains multiple slices (2D) of an organ at different depths.

4.5 Differences in Pooling Layers Between CNN 1D, 2D, and 3D

- **1D Pooling:** Reduces the size of one-dimensional feature maps by applying a downsampling operation (max or average) over consecutive segments of the single dimension.
- **2D Pooling:** Reduces the size of 2D feature maps by applying downsampling over rectangular regions (height and width) within each image.
- **3D Pooling:** Reduces the size of 3D feature maps by applying downsampling over cubic blocks of data (height, width, and depth), which helps preserve spatial and temporal information.

4.6 Specific Challenges of CNN 3D Compared to CNN 2D

The challenges of CNN 3D include:

- **Computational Complexity:** CNN 3D requires more computational resources and memory due to the larger volume of data, making training time-consuming and costly.
- **Overfitting:** With more parameters to learn, there is an increased risk of overfitting, especially if the datasets are limited.
- **Data Preprocessing:** Managing volumetric data may require more sophisticated preprocessing techniques to ensure data quality and consistency in learning.

5 Implementation of CNN 1D on Energy Consumption Data

In this section, we implement a 1D Convolutional Neural Network (CNN) to analyze energy consumption data. The data is read using the following code:

```
df = pd.read_csv('../datasets/d5.csv')
```

All interpretations of the data were made in the previous report where we worked with Long Short-Term Memory (LSTM) networks. To prepare our data for the CNN, we start by defining a Z-score function to standardize our input data:

```
def zscore(s, window, thresh=1, return_all=False):
    roll = s.rolling(window=window, min_periods=1, center=True)
    avg = roll.mean()
    std = roll.std(ddof=0)
    z = s.sub(avg).div(std)
    m = z.between(-thresh, thresh)

    if return_all:
        return z, avg, std, m
    return s.where(m, avg)
```

The Z-score normalization is defined mathematically as:

$$Z = \frac{X - \mu}{\sigma}$$

where Z is the Z-score, X is the original value, μ is the mean of the values, and σ is the standard deviation. This transformation ensures that the data has a mean of 0 and a standard deviation of 1.

Next, we transformed our data into sequences with a window size of 15 days, which allows the model to capture temporal patterns in energy consumption. After this, we standardized our data using MinMaxScaler, which scales the data to a fixed range, typically [0, 1].

We then converted our feature and target variables (X and Y) into tensors for use in PyTorch.

The architecture of our CNN model is defined in the following class:

```
class EnergyCNN1D(nn.Module):
    def __init__(self, input_size=15, output_size=1,
num_filters=64, kernel_size=1, num_layers=1):
        super(EnergyCNN1D, self).__init__()
        self.conv_layers = nn.ModuleList()
        for i in range(num_layers):
            in_channels = input_size if i == 0 else num_filters
            self.conv_layers.append(nn.Conv1d(in_channels,
num_filters, kernel_size=kernel_size))
            self.conv_layers.append(nn.ReLU())
        self.fc = nn.Linear(num_filters, output_size)

    def forward(self, x):
        for layer in self.conv_layers:
            x = layer(x)

        x = x.view(x.size(0), -1)  # Flattening the output
        out = self.fc(x)
        return out
```

The key components of the CNN are:

1. **Convolutional Layer**: This layer applies a convolution operation to the input, which helps to learn spatial hierarchies in the data. The mathematical operation for a 1D convolution is given by:

$$Y[i] = \sum_{j=0}^{k-1} X[i+j] \cdot W[j] + b$$

where $Y[i]$ is the output, X is the input, W is the filter (kernel), b is the bias, and k is the kernel size.

2. **Activation Function**: We used the ReLU (Rectified Linear Unit) activation function, which is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

This function introduces non-linearity to the model, enabling it to learn complex patterns.

3. **Fully Connected Layer**: After the convolutional layers, a fully connected layer is used to produce the final output. It connects every neuron in the last convolutional layer to the output neurons.

We instantiated the model with the following parameters:

```
input_size = 1
hidden_layer_size = 50
output_size = 1
num_layers = 3
model = EnergyCNN1D(input_size=15, output_size=1, num_filters=64,
    kernel_size=1, num_layers=2)
```

To examine the model's output shape, we utilized:

```
outputs = model(X_train)
print(outputs.shape)
```

We defined our loss function and optimizer as follows:

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

The Mean Squared Error (MSE) loss is calculated using:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of observations. This loss function is particularly useful for regression tasks as it measures the average squared difference between the predicted and actual values.

The training process was conducted over 100 epochs, recording training and validation losses:

```
num_epochs = 100
train_losses = []
val_losses = []

for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train)
    train_loss = criterion(outputs, y_train.view(-1, 1))
    train_loss.backward()
    optimizer.step()
    train_losses.append(train_loss.item())

    model.eval()
    with torch.no_grad():
        val_outputs = model(X_test)
        val_loss = criterion(val_outputs, y_test.view(-1, 1))
        val_losses.append(val_loss.item())

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Train Loss: {
    train_loss.item():.4f}, Validation Loss: {val_loss.item():.4f}'
    )
```

After training, we calculated metrics on the validation set:

```
with torch.no_grad():
    final_outputs = model(X_test)

    y_test_np = y_test.view(-1, 1).cpu().numpy()
    final_outputs_np = final_outputs.cpu().numpy()

    mse, rmse, r2 = compute_metrics(y_test_np, final_outputs_np)
```

```
print(f'Validation MSE: {mse:.4f}')
print(f'Validation RMSE: {rmse:.4f}')
print(f'Validation R square: {r2:.4f}')
```

The results obtained after training were as follows:

```
Epoch [10/100], Train Loss: 0.0015, Validation Loss: 0.0018
Epoch [20/100], Train Loss: 0.0015, Validation Loss: 0.0018
Epoch [30/100], Train Loss: 0.0015, Validation Loss: 0.0018
Epoch [40/100], Train Loss: 0.0015, Validation Loss: 0.0018
Epoch [50/100], Train Loss: 0.0014, Validation Loss: 0.0018
Epoch [60/100], Train Loss: 0.0014, Validation Loss: 0.0018
Epoch [70/100], Train Loss: 0.0014, Validation Loss: 0.0018
Epoch [80/100], Train Loss: 0.0014, Validation Loss: 0.0018
Epoch [90/100], Train Loss: 0.0014, Validation Loss: 0.0018
Epoch [100/100], Train Loss: 0.0014, Validation Loss: 0.0018
Validation MSE: 0.0018
Validation RMSE: 0.0426
Validation R square: 0.9682
```

5.0.1 Interpretation of Results

The training and validation losses show a decreasing trend over the epochs, indicating that the model is learning effectively.

- **Training Loss:** The training loss decreased from 0.0015 to 0.0014, demonstrating the model's improvement in fitting the training data. - **Validation Loss:** The validation loss remained relatively stable at around 0.0018 throughout the training process. This stability suggests that the model is generalizing well to unseen data, with no significant **overfitting** occurring.

The validation metrics further indicate strong performance: - **Mean Squared Error (MSE):** The MSE of 0.0018 indicates low average squared differences between predicted and actual values. - **Root Mean Squared Error (RMSE):** An RMSE of 0.0426 suggests that the model's predictions deviate, on average, by approximately 0.0426 from the actual values, which is acceptable in the context of **energy consumption data**. - **R^2 Score:** An R^2 value of 0.9682 signifies that about 96.82% of the variance in the validation dataset can be explained by the model, indicating a good fit.

Overall, the results suggest that the **EnergyCNN1D** model is effective in predicting energy consumption based on the given dataset.