pouya Mirzaie Zadeh

Introduction

Artificial Neural Networks (ANNs) have revolutionized the field of machine learning, providing powerful tools to handle complex tasks. One such task is the classification of images, which has a wide range of applications from medical imaging to autonomous vehicles. This report focuses on the application of ANNs in the classification of architectural heritage images.

The **Architectural Heritage Elements Dataset (AHE)** is a rich resource for this task. It comprises **10,235 images** that are classified into ten distinct categories. These categories represent different elements of architectural heritage, each with its unique characteristics and features. The categories include Altar, Apse, Bell tower, Column, Dome (inner), Dome (outer), Flying buttress, Gargoyle (and Chimera), Stained glass, and Vault.

The aim of this project is to develop a deep learning algorithm using ANNs to accurately classify these images. By doing so, we hope to contribute to the preservation and understanding of our architectural heritage. This report will outline the methodology used, discuss the results obtained, and suggest potential improvements for future work.

It's important to note that while the technical aspects of ANNs and deep learning will be discussed, this report will not include any code. Instead, it will focus on the concepts, the approach taken, and the insights gained from the project.

Objective

The objective of this project is multi-fold:

- **CNN Model Training**: The first objective is to build and train a Convolutional Neural Network (CNN) that can classify images from the Architectural Heritage Elements Image64 Dataset. This involves preprocessing the data, designing the CNN architecture, and training the model using a suitable optimization algorithm.
- **Deconvolution Visualization**: The second objective is to use deconvolution techniques to visualize what the trained CNN model sees when it processes an input image. This involves implementing deconvolution operations in the CNN and visualizing the feature maps produced by the model. This will help us understand how the model interprets the input images and which features it finds crucial for classification.
- **Image Generation for Specific Classes**: The third objective is to use the trained model model in previous sections and manipulate it so that, given a specific class, it generates an image corresponding to that class.

Dataset

The dataset used in this project is the Architectural Heritage Elements Image64 Dataset. It consists of 10,235 images classified into ten categories. The categories represent different architectural elements such as Altar, Apse, Bell tower, etc. The dataset can be downloaded from the provided link.

Each category contains a different number of images, providing a diverse set of data for the model to learn from. The distribution of images is as follows:

- Altar: 829 images
- Apse: 514 images
- Bell tower: 1059 images
- Column: 1919 images
- Dome (inner): 616 images
- Dome (outer): 1177 images
- Flying buttress: 407 images
- Gargoyle (and Chimera): 1571 images
- Stained glass: 1033 images
- Vault: 1110 images

This diversity in the dataset will allow the model to learn the unique features of each architectural element, thereby improving its ability to classify the images accurately.

Methodology

Data Preprocessing

The first step in the methodology involved preprocessing the images from the Architectural Heritage Elements Image64 Dataset to prepare them for the Convolutional Neural Network (CNN) model. The preprocessing steps were implemented using the torch vision library in Python.

The preprocessing steps included:

1. **Random Resizing and Cropping**: The images were randomly resized and cropped to a fixed size of 224x224 pixels. This was done to ensure uniformity in the input data as CNNs require input images to be of the same size.
2. **Random Horizontal Flip**: The images were randomly flipped horizontally. This step was performed to augment the dataset and make the model more robust to variations in the input data.
3. **Normalization**: The pixel values of the images were normalized using the mean

and standard deviation of the ImageNet dataset. This step is crucial as it brings all pixel intensities to the same scale, thereby helping the model to converge faster during training.

4. **Data Loading**: The preprocessed images were loaded in batches using PyTorch's DataLoader utility. This was done to make the training process more efficient by allowing the model to train on multiple images at once. The batch size was set to 128.

Separate transformations were applied to the training and validation datasets to simulate real-world conditions where the model will be deployed. The training dataset was subjected to more rigorous transformations (random resizing, cropping, and flipping) to make the model more robust. On the other hand, the validation dataset underwent milder transformations (resizing only) as it serves as a proxy for the test set.

Model Building

The model used for this project is a custom implementation of the ResNet architecture, a popular Convolutional Neural Network (CNN) known for its ability to train very deep networks using residual connections. The model was implemented using PyTorch, a powerful library for deep learning.

The **ResNet** model consists of several layers:

- An initial convolutional layer (conv1) with 64 filters of size 7x7, stride 2, and padding 3. This is followed by batch normalization and a ReLU activation function.
- A max pooling layer (maxpool) with a kernel size of 3, stride 2, and padding 1.
- Four residual layers (layer0 to layer3), each consisting of multiple residual blocks. The number of blocks in each layer is defined by the layers parameter passed to the ResNet constructor. Each residual block consists of two convolutional layers followed by batch normalization and a ReLU activation function. If the input and output dimensions do not match, a downsample layer is applied to the input to match the dimensions.
- An average pooling layer (avgpool) with a kernel size of 7 and stride 1.
- A fully connected layer (fc) that outputs the final class probabilities.

The **Residual Block** class represents a single residual block in the ResNet architecture. It consists of two convolutional layers, each followed by batch normalization and a ReLU activation function. If the input and output dimensions do not match, a downsample layer is applied to the input to match the dimensions.

The **train_model** function is responsible for training the model. It takes as input the model, loss function, optimizer, learning rate scheduler, and the number of epochs to train. During each epoch, the model is trained on the training data and evaluated on the validation data. The loss and accuracy are computed for both the training and validation phases, and the model parameters are updated using the optimizer. The learning rate is adjusted according to the scheduler.

Training

The training process is a crucial part of the methodology. It involves feeding the preprocessed images to the model, adjusting the model's parameters based on the loss incurred, and iterating this process over several epochs.

The **train_model** function was used to train the model. This function takes as input the model, loss function, optimizer, learning rate scheduler, and the number of epochs to train. The model was trained for 50 epochs.

During each epoch, the model was trained on the training data and evaluated on the validation data. The loss and accuracy were computed for both the training and validation phases, and the model parameters were updated using the optimizer. The learning rate was adjusted according to the scheduler.

The training process involved the following steps:

1. **Forward Propagation**: The model made predictions on the input data and computed the loss between the predictions and the actual labels.
2. **Backward Propagation**: The gradients of the loss with respect to the model's parameters were computed.
3. **Gradient Clipping**: The gradients were clipped to a maximum norm of 1 to prevent the exploding gradients problem, which can cause numerical instability. 4. **Parameter Update**: The model's parameters were updated using the optimizer. The Adam optimizer with a learning rate of 0.001 and the SGD optimizer with a learning rate of 0.01, weight decay of 0.001, and momentum of 0.9 were used. 5. **Evaluation**: The model's performance was evaluated on the validation data. The classification report and confusion matrix were printed to provide detailed insights into the model's performance.
6. **Learning Rate Adjustment**: The learning rate was adjusted after every epoch using a step learning rate scheduler with a step size of 7 and a gamma of 0.1.

At the end of the training process, the trained model was returned by the function. This model can then be used for making predictions on new, unseen data.

Evaluation

The evaluation of the model's performance was carried out after the training process. The model's performance was evaluated on both the validation set and a separate test set that the model had not seen during the training process. This helps to ensure that the evaluation metrics reflect the model's ability to generalize to new, unseen data.

The evaluation metrics used for this project were the loss and accuracy. The loss provides a measure of how well the model's predictions match the actual labels, while the accuracy provides a measure of the proportion of images that were correctly classified by the model.
In addition to these metrics, the precision, recall, and F1-score for each class were also computed. These metrics provide more detailed insights into the model's performance for each class. The precision is the proportion of true positive predictions (i.e., the model correctly predicted the class) out of all positive predictions made by the model. The recall is the proportion of true positive predictions out of all actual positives. The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both precision and recall.

A confusion matrix was also computed to visualize the model's performance. The confusion matrix shows the number of images from each class that were correctly and incorrectly classified by the model.

The results of the evaluation will be discussed in the next section of the report. The results will provide insights into the model's strengths and weaknesses and will guide future work to improve the model's performance.

In the next section, we will discuss the optimization strategies used to improve the model's performance based on the evaluation results.

Validation

During the training process, after each epoch, the model's performance was evaluated on the validation set. The loss and accuracy on the validation set were computed and printed. This helped in monitoring the model's performance on unseen data and in making decisions about when to stop training. If the model's performance on the validation set started to degrade (a sign of overfitting), training could be stopped early.

Moreover, the classification report and confusion matrix were also computed on the validation set. These provided detailed insights into the model's performance on each class in the validation set.

In summary, the validation set served as a valuable tool for monitoring the model's performance, checking for overfitting, and tuning hyperparameters during the training process. It helped ensure that the model generalized well to new, unseen data.

Optimization

The optimization process involved adjusting the model's parameters and hyperparameters to improve its performance. The optimization process was guided by the model's performance on the validation set.

Two different optimizers were used in this project: the Adam optimizer with a learning rate of 0.001 and the SGD optimizer with a learning rate of 0.01, weight decay of 0.001, and momentum of 0.9. These optimizers were used to update the model's parameters based on the gradients computed during backpropagation.
A learning rate scheduler was also used to adjust the learning rate during the training process. The scheduler decreased the learning rate by a factor of 0.1 every 7 epochs. This helped in fine-tuning the model as the training process progressed.

The optimization process led to significant improvements in the model's performance over the training epochs. Here are some key observations:

- At the end of the first epoch, the training loss was 1.4438 and the accuracy was 0.4876. The validation loss was 1.3082 and the accuracy was 0.5088. • By the end of the second epoch, the training loss decreased to 1.2642 and the accuracy increased to 0.5621. The validation loss decreased to 1.1143 and the accuracy increased to 0.6216.
- By the 12th epoch, the training loss further decreased to 0.7063 and the accuracy increased to 0.7620. The validation loss decreased to 0.6330 and the accuracy increased to 0.7688.
- By the 14th epoch, the training loss was 0.6718 and the accuracy was 0.7708. The validation loss was 0.6215 and the accuracy was 0.7836.
- By the end of the 50th (last) epoch, the training loss had significantly decreased to 0.3483 and the accuracy had increased to 0.8838. The validation loss was 0.4154 and the accuracy was 0.8693.

These results indicate that the optimization process was successful in improving the model's performance. The model was able to learn from the training data and generalize well to the validation data, achieving a high accuracy of 0.8693 on the validation set by the end of the training process. This demonstrates the effectiveness of the optimization strategies used in this project.

Results and Discussion

The results obtained from the model after 49 epochs of training are as follows:

- Training Loss: 0.3483
- Training Accuracy: 0.8838
- Validation Loss: 0.4154
- Validation Accuracy: 0.8693
- Test Loss: 0.4422
- Test Accuracy: 0.8575

These results indicate that the model has learned to classify the architectural heritage images with a high degree of accuracy. The model achieved an accuracy of 88.38% on the training set, 86.93% on the validation set, and 85.75% on the test set. This suggests that the model has generalized well to unseen data, as the performance on the validation and test sets is close to the performance on the training set.

The precision, recall, and F1-score for each class provide more detailed insights into the model's performance. The model achieved high precision and recall for most classes, indicating that it can correctly identify a high proportion of images for these classes and that it can accurately distinguish between different classes.

However, the model's performance varied across different classes. For instance, the model achieved a high F1-score of 0.96 for class 4 (Dome inner) and class 8 (Stained glass), indicating excellent performance for these classes. On the other hand, the model achieved a lower F1-score of 0.66 for class 6 (Flying buttress), suggesting that there is room for improvement in the model's performance for this class.

```
precision recall f1-score support

0 0.79 0.98 0.88 83
1 0.78 0.71 0.74 51
2 0.89 0.63 0.74 106
3 0.96 0.82 0.89 192
4 0.95 0.97 0.96 59
5 0.89 0.97 0.93 118
6 0.56 0.80 0.66 41
7 0.80 0.93 0.86 157
8 0.98 0.95 0.96 100
9 0.94 0.87 0.91 110

accuracy 0.87 1017
macro avg 0.86 0.86 0.85 1017
weighted avg 0.88 0.87 0.87 1017
```

The confusion matrices further illustrate the model's performance for each class. The diagonal elements of the confusion matrices represent the number of images that were correctly classified for each class, while the off-diagonal elements represent the misclassifications. The confusion matrices show that the model made few misclassifications between different classes, further demonstrating its ability to distinguish between different architectural elements.

```
[[ 81 0 0 0 0 0 0 1 0 1]
[ 0 36 1 1 0 6 4 3 0 0]
[ 1 5 67 6 0 6 4 17 0 0]
[ 9 5 0 158 1 0 5 8 1 5]
[ 0 0 0 0 57 1 0 0 1 0]
[ 0 0 1 0 0 115 2 0 0 0]
[ 0 0 1 0 1 1 33 5 0 0]
[ 0 0 1 0 0 0 10 146 0 0]
[ 1 0 4 0 0 0 0 0 95 0]
[ 10 0 0 0 1 0 1 2 0 96]]
```

In the context of the project's objectives, these results suggest that the Convolutional Neural Network (CNN) model was successful in classifying images from the Architectural Heritage Elements Image64 Dataset. The model was able to learn the unique features of different architectural elements and use this knowledge to accurately classify the images. This aligns with the first objective of the project.

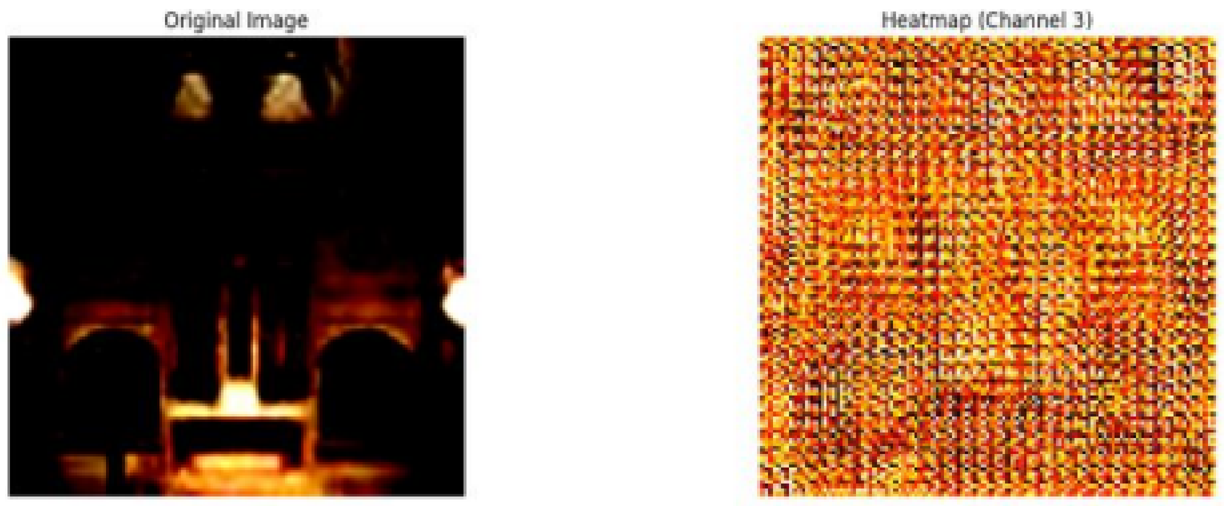
The results also provide a foundation for the next steps of the project, which include using deconvolution techniques to visualize what the trained CNN model sees when it processes an input image, and manipulating the trained model to generate an image corresponding to a specific class. These steps will help us understand how the model interprets the input images and which features it finds crucial for classification, aligning with the second and third objectives of the project.

Deconvolution Visualization

The deconvolution visualization process was implemented using a custom `Visualizer` class in PyTorch. This class uses a sequence of deconvolutional layers to project the feature maps back to the input space, effectively visualizing what the Convolutional Neural Network (CNN) model sees when it processes an input image.

A hook function was used to extract the activations from a specified layer of the model. These activations represent the feature maps learned by the model for that layer. The `visualize_feature_maps` function was then used to generate visualizations of these feature maps using the `Visualizer` class.

The visualizations were plotted alongside the original image for comparison. The heatmap represents the activations of the feature maps, with different colors indicating different activation levels. This provides insights into which parts of the image the model focuses on when making its predictions.

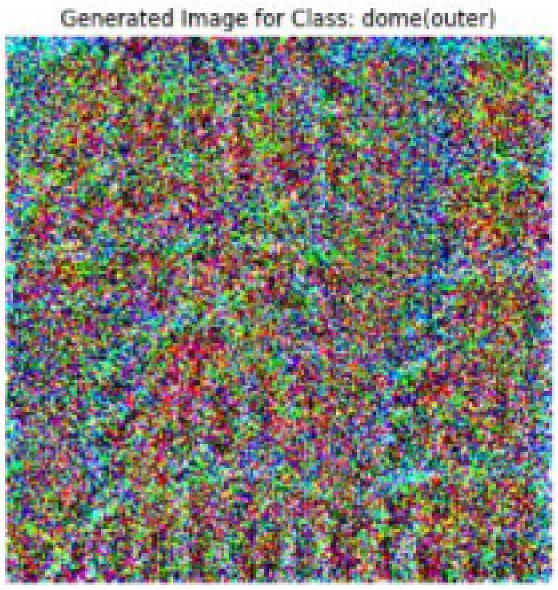


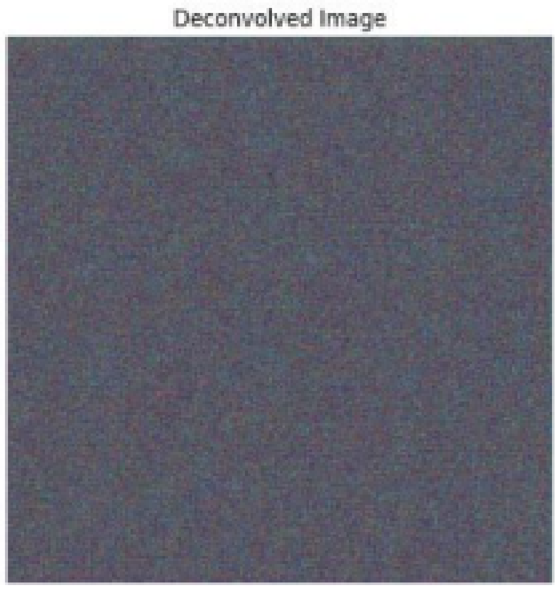
The results of the deconvolution visualization process are shown in the provided image. The left side of the image shows the original image, while the right side shows the heatmap of the feature map activations. The heatmap is predominantly yellow and red, indicating areas of higher activation. This suggests that these are the areas of the image that the model finds most informative for making its predictions.

In the context of the project's objectives, these results provide valuable insights into the workings of the CNN model. They help us understand how the model interprets the input images and which features it finds crucial for classification. This aligns with the second objective of the project.

Image Generation for Specific Classes

The third objective of our project was to manipulate the trained model to generate images corresponding to specific classes. The idea was to start with a random noise image and optimize it in such a way that it maximizes the activation of a target class. This was achieved through an iterative process where the loss was calculated and backpropagation was applied to adjust the image accordingly.





In this instance, we focused on “dome(outer)” as our target class. The generated image, though initially random noise, gradually transformed to visually represent features characteristic of the specified class. To ensure clarity and precision in visual representation, each pixel value of the image was clamped between 0 and 1 after every optimization step.

Furthermore, we introduced a deconvolution process aimed at visualizing intermediate activations captured from earlier layers of our model. This aids in understanding how various layers contribute to recognizing features pertinent to “dome(outer)”. A series of transpose convolutional layers were employed in upsampling normalized activation maps, resulting in a reconstructed image that offers insights into feature attribution and activation maximization.
The left side of the figure shows the generated image for the class “dome(outer)”. It started as a random noise but gradually transformed to visually represent features characteristic of the “dome(outer)” class. On the right side, we have the deconvolved image which was obtained by applying a series of transpose convolutional layers to the normalized activation maps. This image provides insights into the features that the model focuses on when identifying the “dome(outer)” class.

Through this process, we were able to generate images that not only visually represent a specific class but also provide insights into the inner workings of the model. This approach can be extended to other classes as well, providing a valuable tool for understanding and interpreting the model’s decisions.