Deep Learning
Lahore University Of Management Sciences
Spring 2024
Dr. Murtaza Taj

Programming Assignment 3

Autoencoders and Their Applications

DISCLAIMER: None of your models should exceed 500,000 parameters. Furthermore, students are reminded that this assignment is an individual effort, and each participant is expected to submit their own work. Plagiarism is strictly prohibited, including the utilization of chatbots or any other automated systems. Be mindful of academic integrity and ensure that your submission is an authentic representation of your understanding and effort.

1 Introduction

In this assignment, we explore the fascinating domain of AutoEncoders, focusing specifically on convolutional AutoEncoders. This task aims to provide you with a deep understanding of AutoEncoder mechanisms and their applications, including data compression, denoising, and image segmentation. Through hands-on experience with Deep AutoEncoder models and enhancing your PyTorch capabilities, you will acquire valuable skills applicable to a range of deep learning projects.

2 Part 1

2.1 Task 1: CIFAR-10 Classification

Begin by loading the CIFAR-10 dataset from `torchvision.datasets`. Your goal is to train a model on the training set and evaluate its performance on the test set, aiming for an accuracy above 70%. Experimentation is encouraged to achieve this goal.

CNN Architecture

Design a CNN class incorporating multiple convolutional layers. Integrate Batch Normalization and dropout layers to improve training stability and generalization. Use ReLU activation functions within your convolutional blocks. The architecture should conclude with a classifier layer outputting class probabilities. Implement the necessary helper functions provided in `helper.py`.

Training Loop

Develop a supervised learning training loop in `helper.py`. This function should track and return the history of training losses and test accuracies, saving the model iteration with the highest test accuracy.

2.2 Task 2: Image Reconstruction and Denoising

This task involves the MNIST dataset, loaded from `torchvision.datasets`. Construct a hierarchical Linear model to reduce the spatial dimensions from `28x28` to a `15`-feature compact latent representation, then reconstruct the images from this compressed form. This Encoder-Decoder architecture is a foundational experiment in image compression.

Linear AutoEncoder Implementation

Implement the `LinearAutoEncoder` class, which hierarchically reduces the dimensionality of the input images to just 15 features in the latent space, and subsequently reconstructs the images from these compressed features. The class should be defined with an `encoder` and a `decoder` as follows:

- The `encoder` part of your network should consist of a sequential model that starts by flattening the input images. It should then sequentially reduce the dimensionality from 784 (for a flattened 28x28 image) to 128, then to 64, followed by 32, and finally to 15 features. Each linear layer, except for the last one, should be followed by a ReLU activation function to introduce non-linearity.
- The `decoder` part should reconstruct the original image dimensions from the 15-feature representation. It should sequentially increase the dimensionality from 15 back to 32, then to 64, followed by 128, and finally to 784, effectively reconstructing the original image size. Each linear layer, except for the last one, should be followed by a ReLU activation function. The final layer should use a Sigmoid activation function to ensure the output values are between 0 and 1, matching the input image's pixel value range.

The `forward` method should pass the input through the `encoder` to obtain the compressed representation, then through the `decoder` to reconstruct the image. The output should be reshaped back to the original image dimensions (-1, 1, 28, 28). This architecture will allow you to explore the capabilities of linear models in encoding and decoding image data, providing a foundational understanding of AutoEncoder architectures.

Gaussian Noise Addition

Implement a function in `helper.py` to add Gaussian noise to images (ensure the pixel values are clamped between 0 and 1).

Unsupervised Training Loop

Adapt the supervised training loop for unsupervised learning (i.e. image reconstruction and image denoising as specified by the `task` parameter, making necessary adjustments for the task at hand. For denoising, incorporate the Gaussian noise function before inputting images into the model, utilizing default parameters.

Convolutional Encoder-Decoder Model

Implement the `AutoEncoder_CNN` class for the tasks of image reconstruction and denoising. This model utilizes convolutional layers to process image data efficiently, maintaining the

spatial hierarchy of images and significantly reducing the number of parameters compared to fully connected linear models.

Detailed Sample Architecture

- **Encoder:** The encoder uses a sequence of convolutional layers to downsample the image, capturing essential features while reducing dimensionality. It starts with two 3×3 convolutional layers, each followed by a ReLU activation function, to progressively extract features while maintaining spatial dimensions. These layers are configured with a stride of 2 and padding of 1, ensuring the output dimensions are halved after each convolution. Following these layers, a 5×5 convolution further compresses the spatial dimensions before a final max pooling operation, which also saves the indices for later use in the decoder. This design choice efficiently condenses the image into a dense feature representation.
- **Decoder:** The decoder architecture mirrors the encoder in reverse, utilizing the saved max-pooling indices to perform unpooling, followed by a series of convolutional transpose layers that incrementally upsample the encoded features back to the original image size. Starting with a max unpooling layer that restores the spatial dimensions using the saved indices, it then applies a 5×5 convolutional transpose layer, followed by two 3×3 convolutional transpose layers, each with a stride of 2 and padding of 1, and an additional output padding of 1 to match the encoder's downsampling path. A sigmoid activation function is applied at the end to ensure the reconstructed image's pixel values are normalized between 0 and 1.

Note on Implementation Flexibility The described `AutoEncoder_CNN` class serves as a sample implementation to guide you in understanding the core principles of convolutional encoder-decoder architectures. However, you are encouraged to explore and experiment with different configurations of convolutional (and deconvolutional) layers, as well as the positioning of the max pooling and unpooling layers. The key requirements for your custom models are:

- Utilize both convolutional and deconvolutional layers to construct the encoder and decoder, respectively.
- Incorporate at least one max pooling layer in the encoder and one corresponding unpooling layer in the decoder to demonstrate understanding of spatial feature compression and expansion.
- Ensure the total number of parameters in your model does not exceed 35,000. This constraint is intended to encourage efficient model design that balances complexity with performance.

Feel free to experiment within the outlined constraints to discover efficient and effective solutions.

Model Comparison

Compare the number of parameters of both models, and comment on how/why a convolutional neural network performs so much better on image data while using far less parameters.

3 CS 437/5317 - Deep Learning - PA3 - Part 2

In Part 2 of the programming assignment, we further explore autoencoders in deep learning, including supervised learning with the STL-10 dataset, self-supervised learning, and a specialized application in medical imaging for brain tumor segmentation.

Task 1: Supervised Learning

STL-10 Image Recognition

Train a CNN classifier on the STL-10 dataset, which comprises ‘3x96x96‘ images. This dataset includes 500 training and 800 testing images per class, supplemented by an extensive unlabeled set of 100,000 images.

Model Training and Evaluation

Develop a convolutional neural network featuring convolutional layers, batch normalization, and pooling layers, culminating in a classifier layer. Train this model on the STL-10 training data and evaluate its performance on the test set. Plot the test accuracy history.

Exploring Pre-trained Models

Experiment with a pre-trained ResNet-18 model from ‘torchvision.models‘, adapting it to the STL-10 dataset. This involves freezing the model’s gradients to leverage its learned embeddings and adjusting the final classifier layer to reflect the 10 classes of STL-10, in contrast to the original 1000 ImageNet classes. You’ll be only training the final layer, so don’t worry about the computational overhead. Plot the accuracy history after training ResNet-18 on our labeled data.

Task 2: Unsupervised Learning with AutoEncoders

Train a deep convolutional encoder-decoder model on the unlabeled images from the STL-10 dataset. This self-supervised learning approach aims to reconstruct images, leveraging the encoder’s learnt embeddings on the massive corpus of unlableled data for subsequent supervised learning tasks.

Model Specifications

Construct your encoder-decoder model to include 4-5 convolutional and deconvolutional layers, supplemented by batch normalization and ReLU activations. Aim for a model size of approximately 200,000 parameters to accommodate the complexity of the dataset.

Training and Visualization

After training the autoencoder, visualize reconstructions of nine random images alongside their original counterparts using ‘plt.subplots‘. Following this, employ the encoder’s embeddings, with frozen gradients, in a classifier model to assess accuracy on the labeled test data.

Task 3: Brain Tumor Segmentation

This task involves segmenting brain tumors from MRI scans using an autoencoder. The goal is to train a model that can accurately segment tumors, indicated by binary masks, from MRI images.

Model and Training Requirements

Import and preprocess the brain tumor segmentation dataset, which can be found here, adjusting images and masks to a '240x240' resolution. Design an autoencoder that inputs an MRI scan and outputs a binary segmentation mask of the same size. Ensure symmetry in the model architecture and incorporate necessary paddings to maintain consistency in resolution between inputs and outputs.

Custom Loss Function: Dice Loss

For segmentation tasks, the Mean Squared Error (MSE) loss function is often not the best choice as it treats all pixels independently and does not consider the spatial correlation between them. MSE can lead to blurred edges in segmentation maps since it does not effectively handle the imbalance between the region of interest and the background, especially when the region of interest occupies a small part of the image (as is the case with the brain tumor data you're working with).
In contrast, the Dice Loss function is particularly well-suited for segmentation tasks due to its focus on spatial overlap accuracy. It directly measures the similarity between the predicted segmentation and the ground truth, thus encouraging the model to improve the overlap between them.
The Dice Loss is mathematically formulated using the Dice Coefficient as follows:

$$L = 1 - \frac{2 \cdot \sum(p \cdot t) + \epsilon}{\sum(p^2) + \sum(t^2) + \epsilon} \tag{1}$$

where p is the predicted mask, t is the ground truth mask, and ϵ is a small smoothing term, typically on the order of $1e-6$, added to avoid division by zero. By minimizing this loss function, the model is encouraged to increase the overlap between the predicted and true masks, enhancing the segmentation quality.

Evaluation Metrics

Evaluate your segmentation model using two key metrics: the Intersection over Union (IoU) and the Dice Coefficient. Both metrics are critical for quantifying the performance of your segmentation model.
The Dice Coefficient measures the similarity between the predicted segmentation (P) and the ground truth mask (G), and is defined as:

$$\text{Dice} = \frac{2 \times |P \cap G|}{|P| + |G|} \tag{2}$$

The Intersection over Union (IoU), also known as the Jaccard Index, is the ratio of the intersection over the union of the predicted and ground truth masks:

$$\text{IoU} = \frac{|P \cap G|}{|P \cup G|} \tag{3}$$

These have been implemented for you in the Notebook, you just need to use them. Note that for both metrics, a value of 1 indicates perfect segmentation, while a value of 0 indicates no overlap.

Submission Instructions

Please submit the following items for both parts of this assignment:

1. Your Python script files (`.py`) containing all the implemented classes, functions, and model definitions.
2. The Jupyter notebook files (`.ipynb`) where you have performed your experiments, including data preprocessing, model training, and evaluation.
3. The `helper.py` file containing all the required helper functions.
4. The model weights for your best-performing models, as determined by the validation loss or accuracy, for each part of the assignment. Ensure that the weights are saved in `.pth` format).

Submit all the required files in a .zip file on LMS with your roll number.

Godspeed kids :) Reach out to me via Slack in case of any confusion.