Computer Vision

Mid-Term Report

Afnan Abdul Gafoor

June 24, 2024

Introduction

This report details my activities over the past month, primarily focused on studying the theoretical underpinnings of deep learning models slated for implementation. The foundational knowledge was acquired from Stanford Lectures (https://youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv)), supplemented by other relevant references. Initial implementations were predominantly carried out in Keras, with efforts also made to replicate these models using PyTorch. The report follows a chronological order reflecting my learning progression. Additionally, the assistance of ChatGPT was utilized to refine and enhance the text where necessary.**The revised PoA is given at the end.**

Contents

1. Learnings from the first Assignment

1.1 Tensors

In the first part of our exercises, we delved into the basics of PyTorch tensors. We learned how to create and manipulate these multi-dimensional arrays, essential for any deep learning task. Starting with simple tensor initialization and reshaping, we explored various operations such as filling tensors with specific values, performing element-wise computations, and even executing matrix multiplications. This foundation is crucial for understanding how data is handled in PyTorch and lays the groundwork for more complex operations in neural networks.

1.2 Autograd

The next set of exercises focused on PyTorch's powerful automatic differentiation feature, Autograd. We experimented with tensors that require gradient computation, performing basic mathematical operations and observing how gradients are automatically tracked. By defining functions and computing their derivatives, we saw firsthand how PyTorch simplifies the process of gradient calculation, which is pivotal for training machine learning models. These exercises highlighted the ease with which PyTorch handles backpropagation, making it an invaluable tool for model optimization.

1.3 Dataset Class

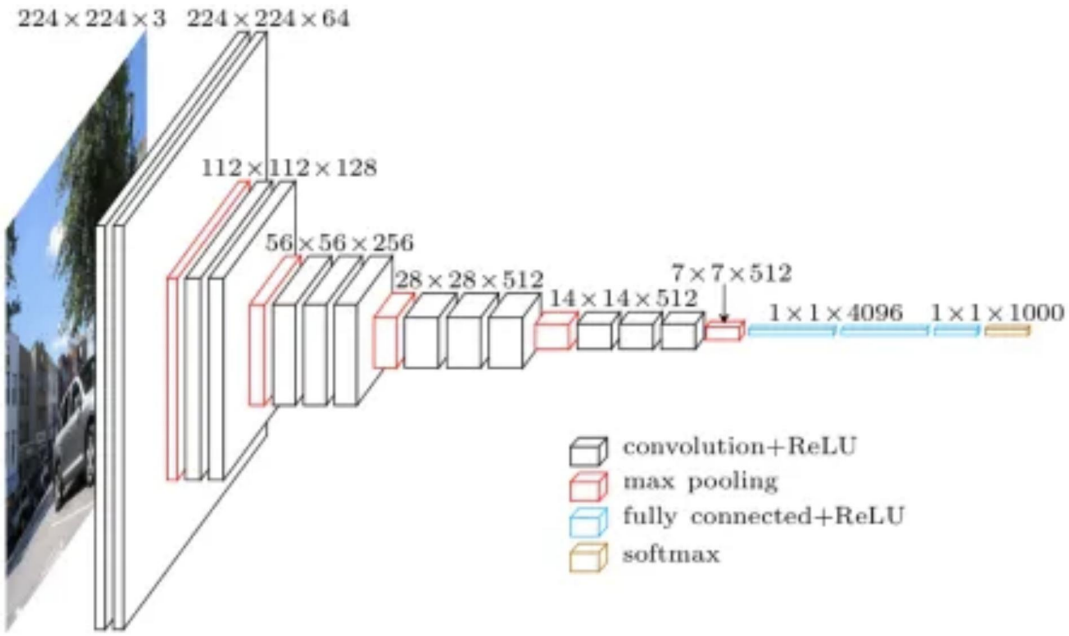
We then moved on to creating custom Dataset classes. This allowed us to efficiently load and preprocess data, a critical step in any machine learning workflow. We built datasets that could handle different data formats, including images and their associated labels or annotations. By incorporating transformations, we learned how to prepare data on-the-fly for training, ensuring that our models receive the right kind of input. This flexibility in data handling is key to managing diverse datasets and ensuring robust model performance.

1.4 DataLoader

Finally, we explored the use of PyTorch's DataLoader, which is essential for managing how data is fed into models. We learned how to batch and shuffle data, apply augmentations, and even handle varying image sizes within batches. We also tackled more advanced techniques like stratified sampling and maintaining order in data loading. These exercises underscored the importance of efficient data loading and processing, ensuring that models are trained on well-organized and pre-processed data, which is crucial for achieving high performance in practical applications.

2. Implementing VGG16 in Keras: Step-by-Step

In this section, we explore the implementation of the VGG16 architecture using Keras. VGG16, a deep convolutional neural network, gained prominence by securing the 2014 ImageNet competition with its simple yet effective design. The model comprises 16 layers with weights and emphasizes the use of small 3x3 convolutional filters and max-pooling layers.



Architecture of VGG16

Ref: https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c

Steps for Implementation:

1. Library Imports and Initialization:

- We begin by importing essential libraries, including Keras modules for constructing and training the neural network, and utilities for data preprocessing. These libraries provide the necessary functions to build a robust deep learning model and handle image data efficiently.

2. Dataset Preparation:

- The dataset used is the "Dogs vs Cats" dataset from Kaggle, which is a common benchmark for binary image classification tasks. Using Keras' 'ImageDataGenerator', we load and preprocess the images, which includes resizing them to 224x224 pixels to match the input size required by VGG16.

- The ‘ImageDataGenerator’ also supports on-the-fly data augmentation techniques such as rotation, scaling, and flipping, enhancing the model's ability to generalize from the training data without permanently altering the images stored on disk.

3. Model Construction:

The VGG16 model is built sequentially, layer by layer. This architecture involves:

- Convolutional layers with small 3x3 filters and ReLU activation functions, arranged in a sequence to extract detailed features from the input images.
- Max-pooling layers following each set of convolutional layers to down-sample the spatial dimensions, reducing the computational load and controlling overfitting.
- The pattern of convolutional and pooling layers repeats, with the number of filters increasing in each stage (starting from 64 and going up to 512).
- After the convolutional blocks, the output is flattened into a single vector and passed through two fully connected (dense) layers with 4096 units each, followed by a final dense layer with 2 units for binary classification (dog or cat).

4. Compiling the Model:

- The model is compiled using the Adam optimizer, which is effective in navigating the complex loss landscape of deep networks. The loss function chosen is categorical cross-entropy, appropriate for multi-class classification problems, and accuracy is set as the metric to monitor.

5. Training the Model:

- To train the model, we use ‘ModelCheckpoint’ to save the best-performing model based on validation accuracy and ‘EarlyStopping’ to halt training if there is no improvement over several epochs. This helps in preventing overfitting and ensures that the model's weights are saved at their optimal state.

6. Evaluating and Visualizing Performance:

- After training, we visualize the training and validation accuracy and loss over epochs to evaluate the model’s performance. This graphical representation helps in understanding how well the model is learning and whether adjustments are needed.

7. Making Predictions:

- For making predictions, we load the saved best model and preprocess the input images to match the expected format. The model then predicts

the class of the image, indicating whether it is a dog or a cat based on the learned features.

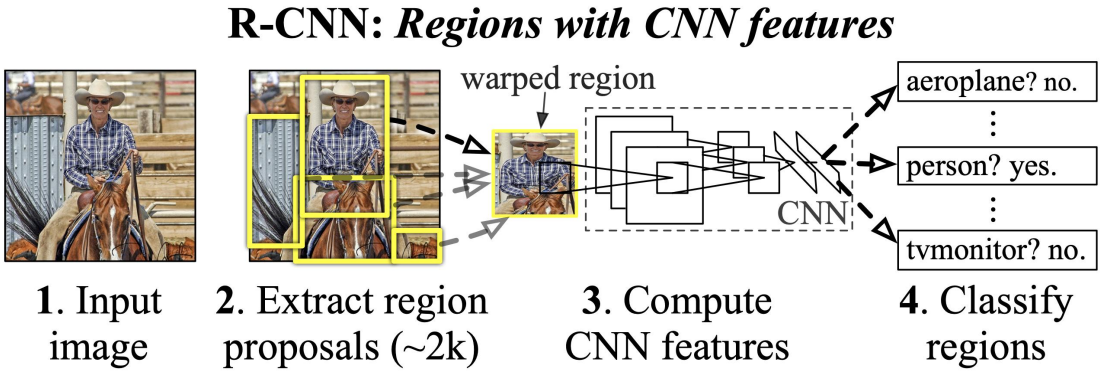
3. Simple Summary of the RCNN Paper

This paper introduces **R-CNN**, a new method for detecting and identifying objects in images. Here's a simplified breakdown of what the paper is about and the key ideas:

1. Problem Background:

- Detecting objects in images (like finding and recognizing cars, people, or animals) has been challenging and slow in terms of improvement.
- Previous methods used complex systems combining many techniques, but progress was minimal.

2. R-CNN Approach:



Ref: https://arxiv.org/pdf/1311.2524v5

- The authors propose a simpler and more effective method called R-CNN (Regions with Convolutional Neural Network features).
- R-CNN works in a few steps:

1. Region Proposals: It first identifies possible areas in the image that might contain objects (about 2000 such areas per image).
2. Feature Extraction: For each of these areas, it uses a powerful model called a Convolutional Neural Network (CNN) to analyze and extract important details.
3. Classification: These details are then used to determine what object (if any) is in each area.

3. Key Innovations:

- Using CNNs for Detailed Analysis: CNNs are good at processing images and finding patterns. By applying them to specific regions in an image, R-CNN can effectively locate and recognize objects.

- Pre-training and Fine-tuning: They first train the CNN on a large set of general images and then adjust (fine-tune) it using a smaller, specific set of images to make it better at detecting specific objects.

4. Performance:

- R-CNN significantly improved object detection accuracy compared to previous methods.
- On a standard test (the PASCAL VOC dataset), R-CNN achieved an accuracy of 53.3%, which was more than 30% better than the previous best method.

5. Efficiency:

- The R-CNN method is also efficient in terms of computation. It shares most of the work across different object classes, making it scalable and faster than many other methods.

Why It's Important:

This paper was a significant step forward in computer vision because it introduced a way to use deep learning effectively for object detection. R-CNN laid the groundwork for future improvements in how machines see and understand images.

4. RCNN Implementation on custom dataset

The script is designed to detect airplanes in images using a combination of selective search for region proposals and a deep learning model (VGG16) for classification.

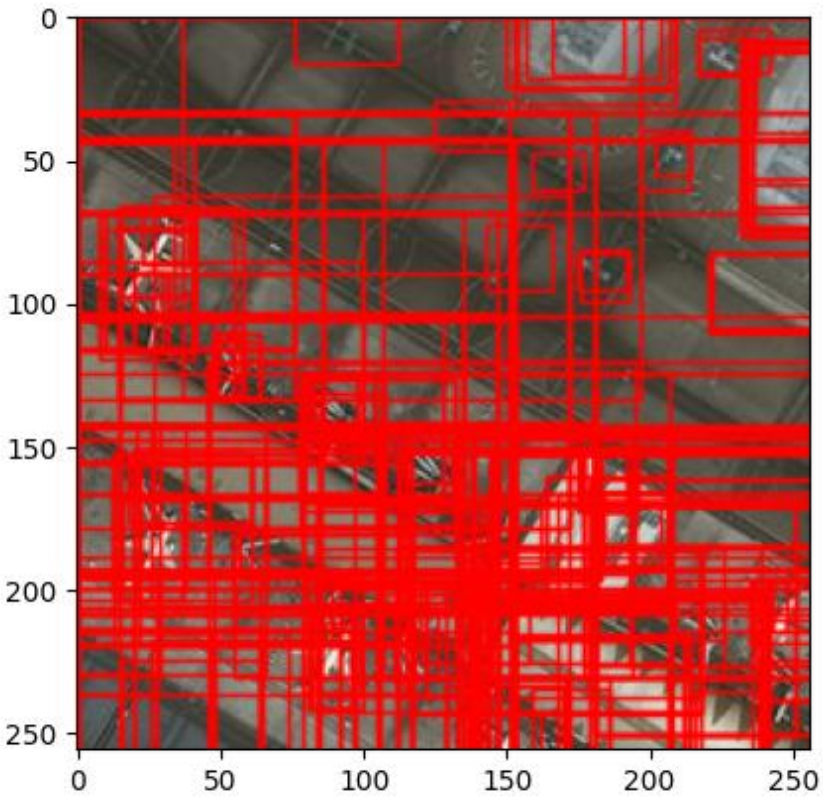
Some of the key steps are:

Region proposal:

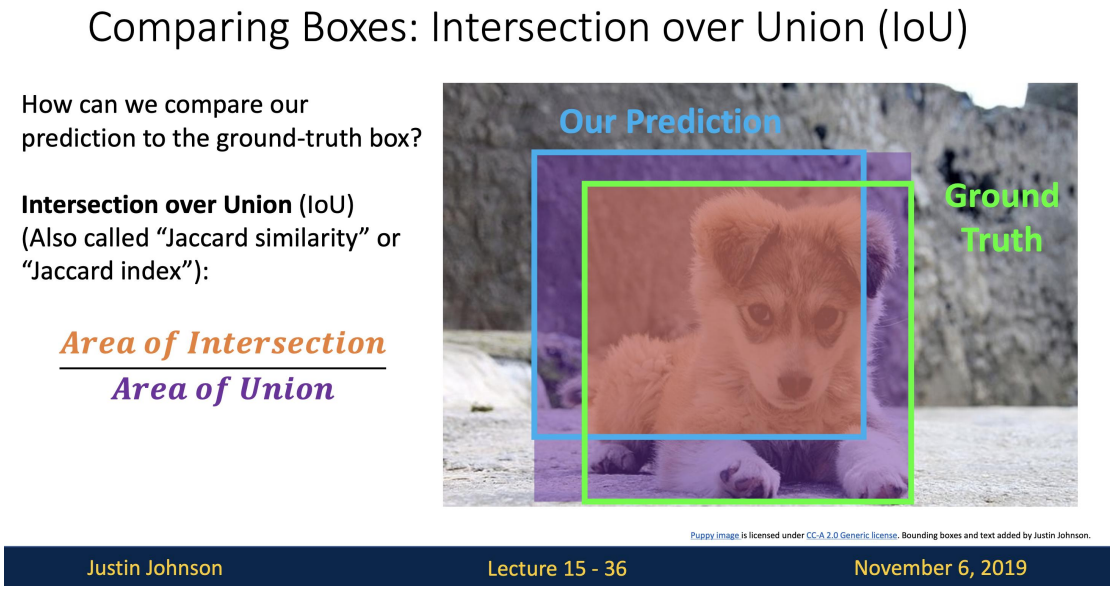
```
im = cv2.imread(os.path.join(path,"42845.jpg"))
ss.setBaseImage(im)
ss.switchToSelectiveSearchFast()
rects = ss.process()
imOut = im.copy()
for i, rect in (enumerate(rects)):
x, y, w, h = rect
# print(x,y,w,h)
# imOut = imOut[x:x+w,y:y+h]
```

```
cv2.rectangle(imOut, (x, y), (x+w, y+h), (255, 0, 0), 1,
cv2.LINE_AA)
# plt.figure()
plt.imshow(imOut)
```

This code snippet demonstrates the application of Selective Search for object proposal generation, specifically for detecting airplanes in images. First, the script optimizes the OpenCV library for improved performance. Then, it initializes Selective Search and loads an example image, "42845.jpg", from a specified directory. Selective Search is configured to use this image as its base and switches to a fast mode for generating region proposals. The resulting regions (rectangles) are computed and overlaid onto a copy of the original image, highlighting potential airplane locations with red rectangles. This visualization aids in understanding how Selective Search identifies candidate regions, crucial for subsequent steps in object detection workflows.



Implementation of IoU



Ref: http://myumi.ch/r80Pz

```
def get_iou(bb1, bb2):
assert bb1['x1'] < bb1['x2']
assert bb1['y1'] < bb1['y2']
assert bb2['x1'] < bb2['x2']
assert bb2['y1'] < bb2['y2']

x_left = max(bb1['x1'], bb2['x1'])
y_top = max(bb1['y1'], bb2['y1'])
x_right = min(bb1['x2'], bb2['x2'])
y_bottom = min(bb1['y2'], bb2['y2'])
```

```
if x_right < x_left or y_bottom < y_top:
return 0.0
```

```
intersection_area = (x_right - x_left) * (y_bottom -
y_top)
```

```
bb1_area = (bb1['x2'] - bb1['x1']) * (bb1['y2'] -
bb1['y1'])
bb2_area = (bb2['x2'] - bb2['x1']) * (bb2['y2'] -
bb2['y1'])
```

```
iou = intersection_area / float(bb1_area + bb2_area -
intersection_area)
assert iou >= 0.0
assert iou <= 1.0
return iou
```

The function begins by asserting that the coordinates of both bounding boxes (‘bb1’ and ‘bb2’) are correctly formatted (‘x1 < x2’ and ‘y1 < y2’). It then computes the intersection area between the two bounding boxes by determining the overlapping region's dimensions in terms of its top-left (‘x_left’, ‘y_top’) and bottom-right (‘x_right’, ‘y_bottom’) corners.

If the bounding boxes do not intersect (‘x_right < x_left’ or ‘y_bottom < y_top’), the function returns an IoU of 0.0, indicating no overlap.

If there is an intersection, it calculates the intersection area using the formula:

intersection_area = (x_right - x_left) x (y_bottom - y_top)

It then computes the areas of each bounding box (‘bb1_area’ and ‘bb2_area’). The IoU is calculated as:

IoU = intersection_area/(bb1_area + bb2_area - intersection)

The function ensures that the IoU value is within the range [0, 1] and returns it. This function is critical in object detection tasks, as IoU is commonly used to evaluate the accuracy of predicted bounding boxes compared to ground truth annotations. It helps determine how well the predicted region aligns with the actual object location, essential for assessing the performance of algorithms like Selective Search in proposing accurate object regions.

Model Architecture

```
for layers in (vggmodel.layers)[:15]:

print(layers)
layers.trainable = False
X= vggmodel.layers[-2].output
predictions = Dense(2, activation="softmax")(X)
model_final = Model(inputs = vggmodel.input, outputs =
predictions)
opt = Adam(lr=0.0001)
```

```
model_final.compile(loss =
keras.losses.categorical_crossentropy, optimizer = opt,
metrics=["accuracy"])
model_final.summary()
```

The code snippet initializes a deep learning model for binary classification using transfer learning with the VGG16 architecture, which has been pre-trained on the ImageNet dataset. This approach leverages the learned representations of the VGG16 model to perform a specific task of distinguishing between images containing airplanes and those that do not.

Firstly, the script selects and freezes the first 15 layers of the VGG16 model. This freezing ensures that the weights within these layers, which have already been optimized for general image feature extraction tasks, remain unchanged during subsequent training. By freezing these layers, computational resources are conserved, and the model benefits from the pre-learned features without modifying them.

Next, it captures the output of the penultimate layer of the VGG16 model. This output serves as the input to a new dense layer configured for binary classification. The dense layer consists of two units, representing the two classes (airplane or not), and employs a softmax activation function. This setup ensures that the model's output provides a probability distribution across the two classes, facilitating classification decision-making.

Subsequently, a new Keras ‘Model’ is instantiated, where the input is specified as the input layer of the VGG16 model (‘vggmodel.input’), and the output is set to the predictions made by the dense layer previously defined. This step finalizes the architecture of the transfer learning model, bridging the pre-trained VGG16 layers with the new classification layer.

After defining the model architecture, the script compiles the model for training. It uses the Adam optimizer with a learning rate of 0.0001, chosen for its efficiency in optimizing deep neural networks. The categorical cross-entropy loss function is specified, appropriate for multi-class classification tasks where each class is mutually exclusive.

Finally, a summary of the model architecture is printed, detailing the layers, their output shapes, and the number of trainable parameters. This summary provides insights into how the VGG16 model has been adapted and configured for the specific task of binary classification, demonstrating transparency in model configuration and readiness for training.

In conclusion, this code snippet showcases a systematic approach to leveraging transfer learning with the VGG16 model for binary classification tasks, ensuring efficient use of pre-trained features and effective adaptation to new classification requirements.

5. Revised PoA

Week 1:Debug/modify previously written codes (like CNN)

Week 2-4:Learn about Gabor Filters,Semantic Segmentation

Week 5:Documentation and Submission