

Convolutional Neural Network Applications: License Plate Recognition

Alireza Honardoost, Morteza Hajiabadi, Ksra Davoodi

Directed by Mahdi Lotfi

Automatic License Plate Recognition (ALPR) is a critical area of research in computer vision with wide-ranging applications in traffic management, law enforcement, electronic toll collection, and security.

In this exploration, we will leverage our knowledge of computer vision and machine learning to develop a Convolutional Neural Network (CNN)-based system for detecting and extracting data from Iranian license plates.

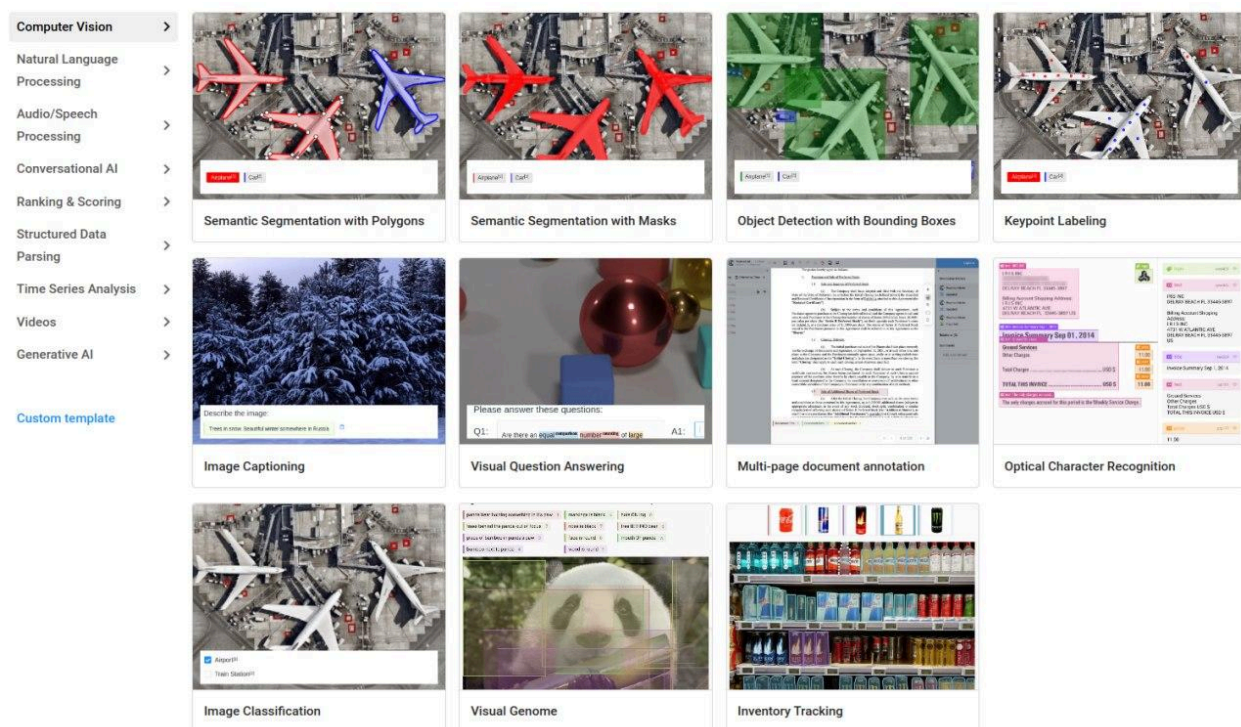
1. Preparing Dataset

We will start our journey from scratch, so we need to label and prepare our dataset.

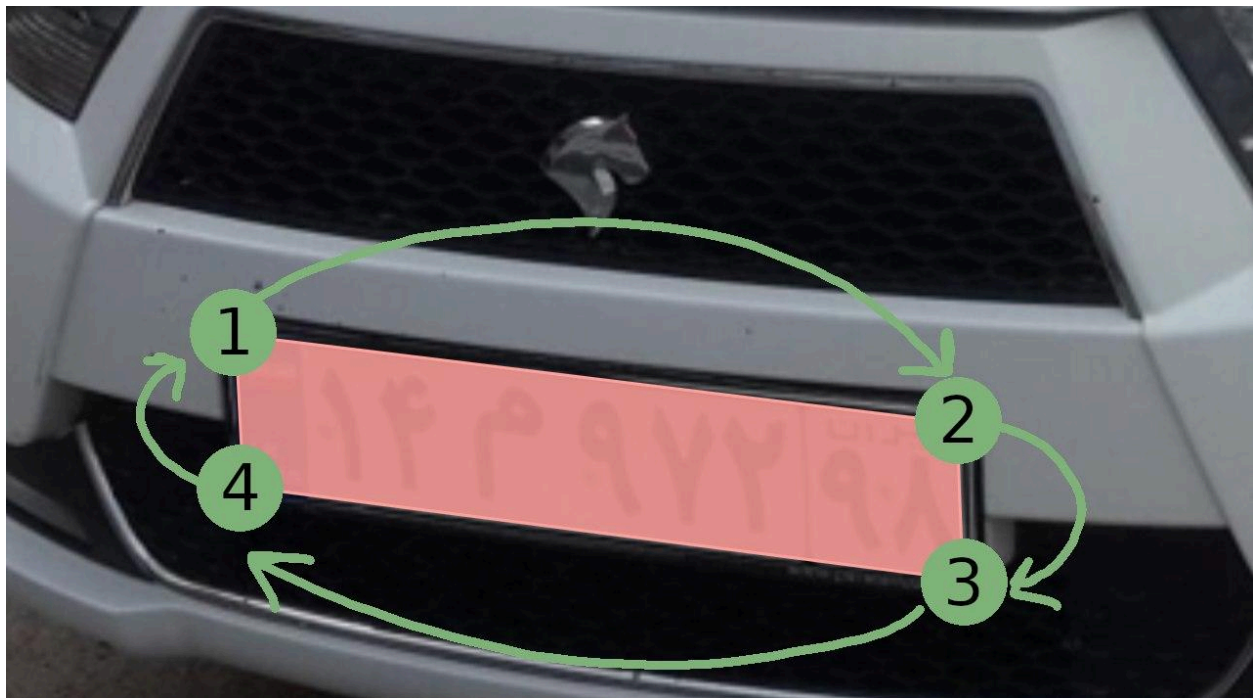
1.1. Label your dataset

Dataset labeling is a crucial step in supervised learning tasks. Many tools are available for labeling datasets, such as CVAT, Labellmg, and Label Studio. These tools support a variety of dataset types and labeling tasks.

For labeling the four corners of the license plates, we will use the "Semantic Segmentation with Polygons" interface in Label Studio



To label an image, navigate to the relevant project and choose the image. If you're labeling multiple images, utilize the "Label All Tasks" option for batch labeling. Once you've opened the annotation tab, select the label category you want to annotate (e.g., Plate) and begin annotating from the top left corner of the plate, continuing clockwise until you return to the starting point.



Ensure you identify exactly four corner points and return to the starting point from the last one.

1.2. Application: Car plate masking

Now that we have identified the four corners of the license plate within the image, let's delve into some exciting applications!

You might have come across websites like Divar (a platform for advertisements) that conceal car license plates in vehicle listings for security purposes.

In this task, we will leverage our knowledge of geometric transformations to obscure the license plate area in our labeled images with the logo of KNTU University!





To accomplish this, we'll create a function in the `masking.py` file that takes the original image, the coordinates of the four corners of the license plate along with a cover image, and returns a new image where the license plate is seamlessly covered by the cover image.

You can export some labeled samples from Label Studio and use `kntu.jpg` for testing your function.



2. Extract the Plate Image

Let's advance our game with the four corners! Our next move involves extracting the plate image.

2.1. Transform the plate

Use your knowledge of geometric transformations to complete the function in the `extract.py` file. This function should take an image and the four corner points of a license plate as inputs, and return the license plate image with a corrected perspective and an aspect ratio of 4.5 (the aspect ratio of Iranian license plates).





2.2. Application: Car plate blurring

Referring back to part 1.2, there are other approaches to conceal the license plate such as blurring. In this task, you need to use the previous parts in the [blurring.py](#) file to blur the license plate in an image.



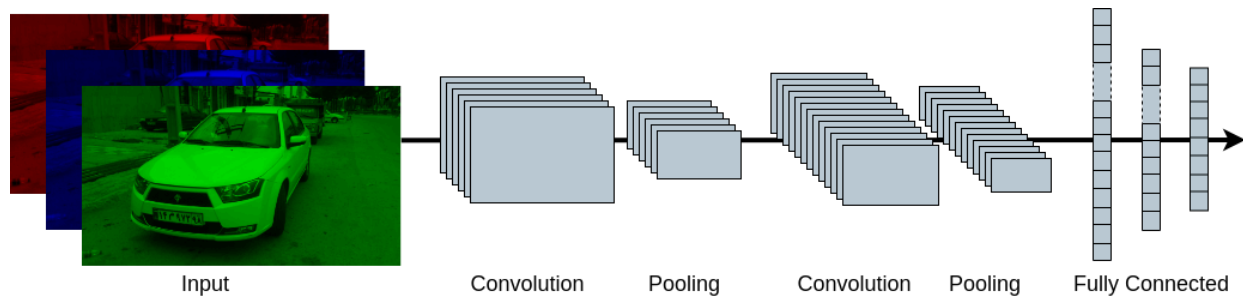
3. Building a Regression CNN

In the previous part, we explored various applications of identifying the four corners of a license plate in a car image. But how can we automatically obtain these points without human intervention? Here, we do this by training a CNN regressor on the previously annotated images.


3.1. Design the model architecture

In this task, we aim to design a deep Convolutional Neural Network that takes color images of cars as input and outputs the four corners of the license plate in each image.

Chain convolutional, pooling, and fully connected layers in the [regressor.py](#) file to build your model. Enhance layer outputs with appropriate activation functions. Note that the first layer should input a constant-size image with three color channels, and the final layer should output 8 numbers (representing the 4 coordinates).




You can also include dropout layers to prevent the model from overfitting. Additionally, utilize regularization techniques within your layers to further mitigate overfitting. It is also recommended to use normalization layers to ensure computational stability.

 **Implementation note:** You are not allowed to use pre-trained or pre-designed model architectures for this project. You **MUST** design the model yourself and provide an explanation for the architecture and different layers of your model. You are free to use either *TensorFlow Keras (Sequential or Functional) API* or *PyTorch* layers to design, train, and test your model.

3.2. Enhance dataset with augmentation

Data augmentation artificially increases the size of the training set and helps reduce overfitting by generating many realistic variants of each training instance. For example, you can apply transformations such as slight shifts, blurs, crops, resizes, rotations, contrast adjustments, and perspective changes to each image in the training set. By adding these transformed images to the training set, you can create a model that is more robust and tolerant to these variations.

Complete the **augment** function in the **loader.py** file to randomly apply small variations to the images. Ensure that the point values are updated accordingly to correctly reflect the transformed four corners of the license plate. Keep in mind that the transformations should not result in invalid or out-of-bound corners or unrealistic car image, and the license plate should remain intact and within the image boundaries.

 **Implementation note:** You are not allowed to use any third-party libraries to handle transformations. You **MUST** use the techniques learned in the course and *OpenCV functions* to apply augmentation.

3.3. Normalize dataset samples

Take a look at some images from our annotated dataset. Are these images all the same size? As we know, our model requires input images to be of a specific, predetermined size. Therefore, to feed these images into our model, we must resize them to a common size.

To complete this task, fill out the `loader.py` file and complete the `resize` function to transform the image and its annotation points to a specific, constant size.

Can you visualize some output images in the new size? By disrupting the aspect ratio of the images, we may end up with distorted and odd-looking car images. Do you think the model can still accurately output the four corners of the plate in this case?

3.4. Train the model

Now, it's time for the model to meet the data!

Finalize the `loader.py` file by completing the `load` function to handle dataset loading, augmentation, and normalization. Ensure this function prepares the data for feeding into the model defined in `regressor.py`. Utilize the built-in features of your selected framework to store model checkpoints during training. Remember to perform a train-test-validation split before starting the training process.

✂ **Option:** Loading the entire dataset using our traditional method is a heavy and costly operation, which is infeasible for low-resource computers. Instead of completing the `load` function, you can earn an extra score by implementing a custom Keras data generator or PyTorch dataset that only loads and processes a subset of images in each epoch.

💡 **Note:** It is recommended to use cloud-based environments such as Google's Colab for training your model. These platforms provide various processing units (CPU, GPU, and TPU) that can significantly speed up your training process.

3.5. Test the model

Now, let's evaluate what our model has learned by working on the `evaluate.py` file. Implement the `corners` function to use your model for automatically obtaining the four corners of the license plate.

Remember to normalize the image size before inputting it into the network and de-normalize the output points to match the original size.

Evaluate your model's accuracy by visualizing its outputs and measuring them using the Mean Squared Error (MSE) metric. Experiment with different model layer parameters, architectures, and augmentation techniques to achieve optimal accuracy on the test data.

4. Building a Classification CNN

Now that we have accurately extracted the car plate image, it's time to read its characters and identify its content. To achieve this, we will use a classifier CNN to detect and extract the characters at each position in the car plate image.

4.1. Generate your dataset!

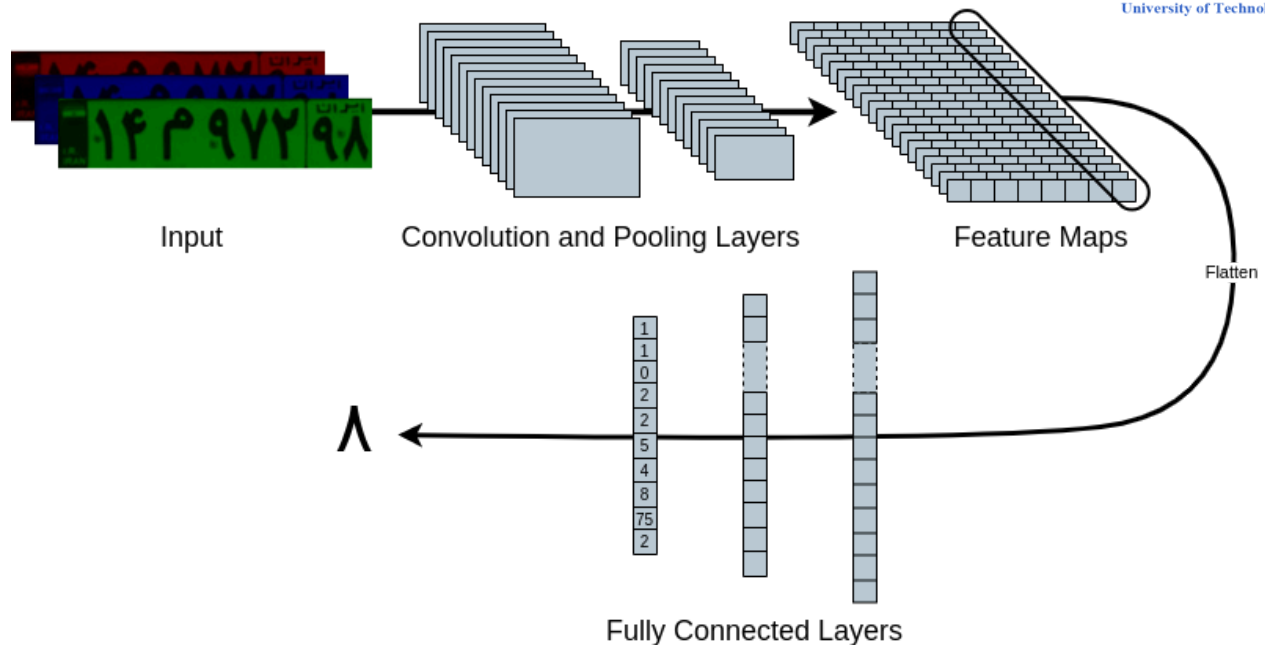
In Part 1, we labeled some real images and, after augmentation, fed them into our CNN to learn and extract features. Another approach is to generate and augment a virtual dataset that can model our problem accurately.

Use the scripts provided in this [repository](#) to generate realistic license plate images. Don't forget to augment these license plates using the script's built-in features and what you have learned. Apply small variations such as slight shifts, blurs, crops, resizes, rotations, contrast adjustments, perspective changes, and add dirt to each image to generate new samples from each original image.

4.2. Design the model architecture

Now we want to train a new classifier CNN to read the content of the extracted license plate image.

In the `classifier.py` file, use a deep convolutional layer structure to create a feature map that generates a deep feature vector for each of the 8 characters. Feed each deep feature vector into a dense layer structure to predict the class of each character. Repeat this process for all 8 deep feature vectors to extract the final 8-character license plate content.



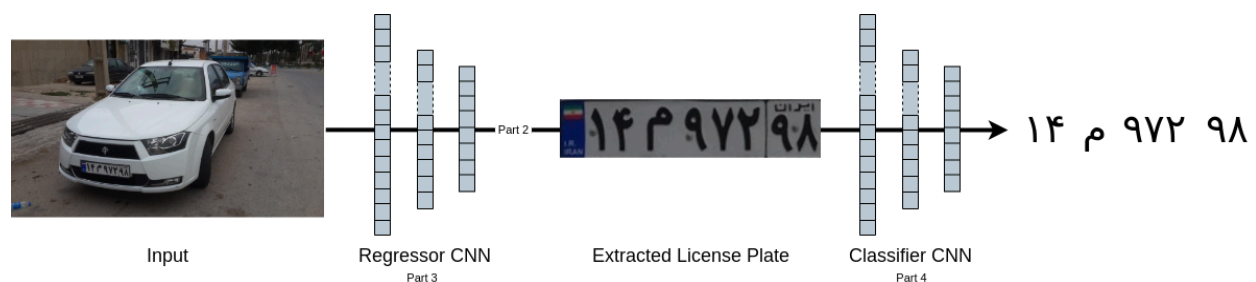
4.3. Train and test the model

Similar to Part 3.4, we will train our model using the generated dataset. Complete the `classifier.py` file to train the model, and then test the model in the `integrate.py` file by completing the `read_plate` function, which outputs the license plate content for a given license plate image. You should report the accuracy of your model by identifying and using an appropriate metric to evaluate its performance in the `integrate.py` file.

Remember to experiment with different hyperparameters and model architectures to find the most effective model.

4.4. Integrate with the previous parts

In our final step, we will combine our previous code to process a single car image, extract the license plate image, and read its content.



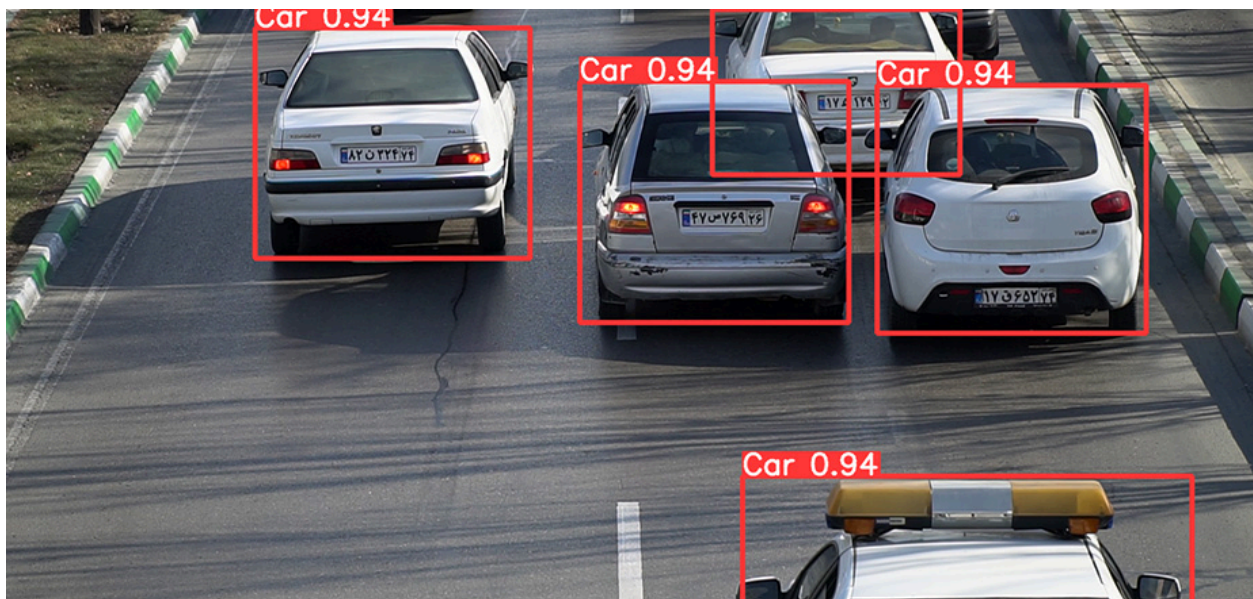
Utilize the previous parts in the `integrate.py` file and complete the `read_single` function. This function should take an image of a single car with an arbitrary size, extract and read the content of the license plate, and return the plate content as a string.

5. Optional: Images with Multiple Cars!

In the previous parts, we learned how to extract and read the content of a license plate from a single car image. Now, we want to extend our work to detect multiple cars in an image and extract the license plate data for each one.

5.1. Clone and prepare a pretrained model

Detecting multiple cars in an image is a complex task that requires a large model trained on a rich dataset. For this task, we will clone and prepare a pretrained model such as YOLO or SSD in the `integrate.py` file.



5.2. Integrate with previous part

Finally, complete the `read_multiple` function to process an image containing multiple cars. This function should output the bounding boxes and the corresponding license plate data for each detected car in the image.



References & Resources

- <https://labelstud.io/guide/install>
- https://youtu.be/A0cob_f5BmM?si=50bU4e8eJCqUdrMy
- Lab 8
- <https://keras.io/api/layers>
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow
- <https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly>
- https://keras.io/api/callbacks/model_checkpoint
- <https://github.com/kntu-utils/license-plate-generator>
- https://keras.io/guides/functional_api
- <https://docs.ultralytics.com/>
- <https://arxiv.org/abs/1506.02640>
- <https://paperswithcode.com/method/ssd>