Autonomous Vehicle DL

License Plate Object Detection

Project Intro:

Object detection is a computer vision technique that allows us to identify and locate objects in an image or video. With this kind of identification and localization, object detection can be used to count objects in a scene and determine and track their precise locations, all while accurately labeling them.

In this project we're building and training a model to find the area in the image that corresponds to a vehicle number plate.
For this task we'll be working with one of the data sets provided on Kaggle website.
The database provides the bounding boxes of the ground truth location of the plate.

Possible approaches:

There are a few ways to tackle such a problem,

A straightforward, but very inefficient, way is to create a training/validation set of images for two classes: plate and background and train a binary classifier for plate/non-plate classification problem for a pre-defined image size.
But in test time we'll be needing to evaluate every position in an image in different scales to find the true position and size of the plate.

Another approach would be to build and train a Neural Network for this task. That could be divided into two categories – classic machine learning architecture, and a more advanced deep learning ones.

For the naïve machine learning architecture we'll need to flatten the entire image and feed it to a few, linear activation functions separated, multilayer perceptron.
In such approach, we'll be losing shape information. And such a feature is radically needed in our task and in other object detection ones.

For the later suggestion, there's quite a few deep learning architectures which can be used to solve this problem,

- Using RCNN, for instance a Faster RCNN architecture with ROI (Region of interest) proposals (could be adding a Resnet-50 to its last few layers).
- A Yolo v4 would do the job well, since it was made and train solely for this specific kind of tasks.
- Build a whole new deep neural net architecture from scratch. The advantages here are that the model well be created and trained for this specific task and for the specific provided data set.

The problem with such an approach is that an object detect task (with continues ambient space of target labels) is a challenging and complex problem (Model wise), and hence would be needing a fair amount of resources – training resources. And our data set has only 237 images. Splitting those to training\validation\ and test subsets would give us roughly ~213 images to train our network upon (considering 10% for validation and testing).

Transfer Learning:

Despite the selected approach, when dealing with such a small amount of data and respectively with such a complex problem, we won't be able to build a well sufficient network with a good generalization. We'll hit over-fitting point regarding the neural weights in the respective model weight ambient space in no time, and we won't be even able to fine-tune the whole network (we won't be able to create a well generalizing spanning base vector for this architecture).

For the above reason, despite the model selection, we'll be needing Transfer Learning.

Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but yet related other problems. For example, knowledge gained while learning to recognize flowers could be applied when trying to recognize palms.

So for our task, we would need to download a pre-trained neural model (preferably for an object detection tasks), add a new network\perceptron and train only this last few added layers.

This gives us the advantage of time saving and the neglect of infrastructure and training resources that we should have in order to handle such a complex and highly dimensional parameter space.

Our Approach:

We didn't want to apply a ready-made and trained model, just a fit-predict approach, and wanted a little challenge. So we wanted to take a pre-trained model, apply a transfer learning by removing its fully connected layers and adding a new dense layers which are more suitable to our goal. Hence, we decided to take a pre-trained VGG-16 architecture (on ImageNet), remove its last few layers (the fully connected ones), add some dense layers of our own, and lastly train our additional layers in addition to fine-tuning the vgg-16 parameters.

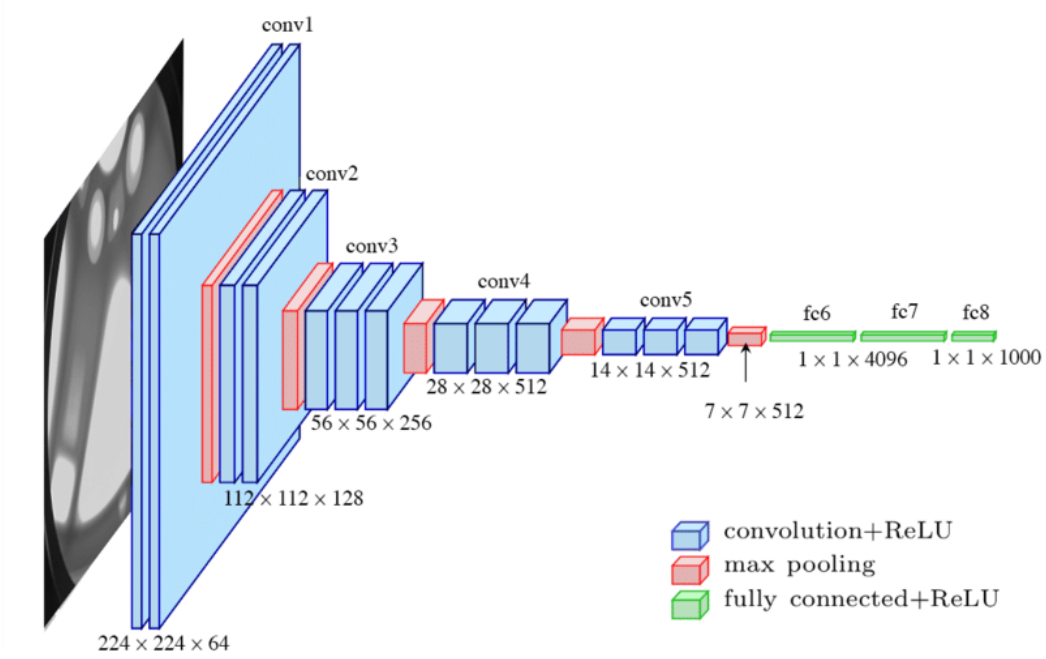

Figure 1: VGG-16 Architecture

Above is the full architecture of the vgg-16 neural network.
In our model, we removed the last 3 fully connected layers, and added 4 new trainable dense layers.

The visualization process is described below,

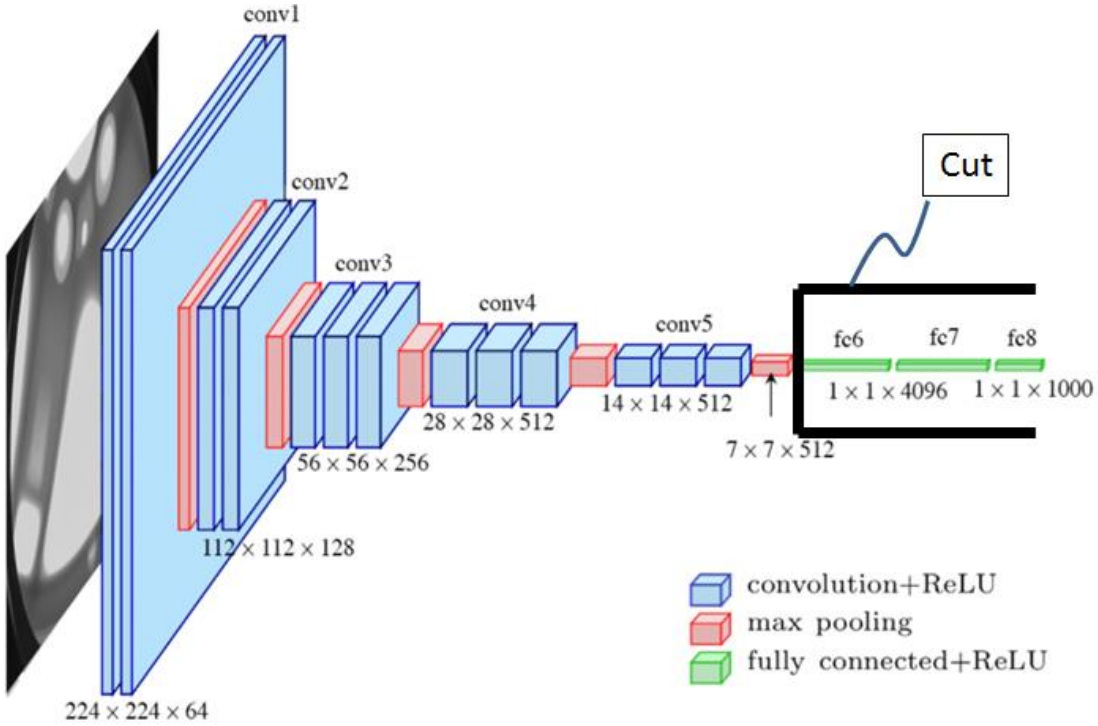

Figure 2: Cutting (removing) the pre-trained last 3 connected layers

After removing the layers,

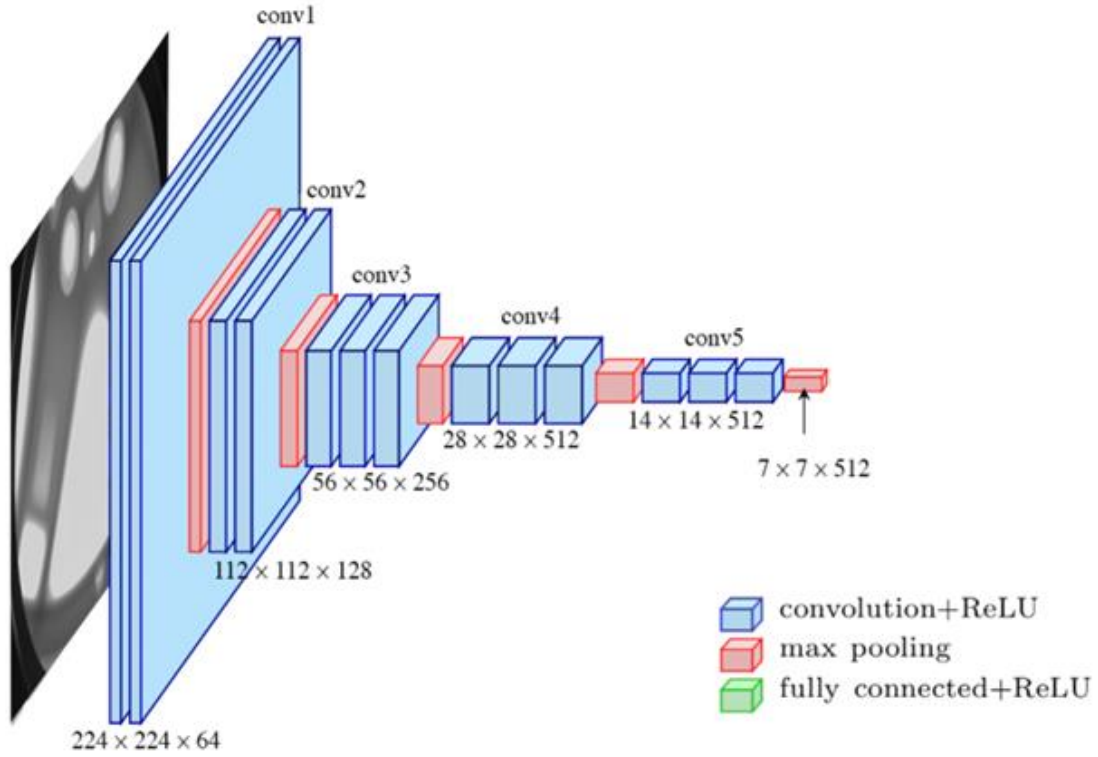

Figure 3: VGG-16 Net after cutting off its dense layers

Thus, resulting that our last layer is a $7\ x\ 7\ x\ 512$ neural layer.

Now, from here we would want to build upon a new set of layers to summarize the features extracted so far and to output an answer. In our case of study, to output a 4 continues parameters which indicates the position of the bounding box on the image.

Adding a new 4 fully connected perceptron layers, we get:

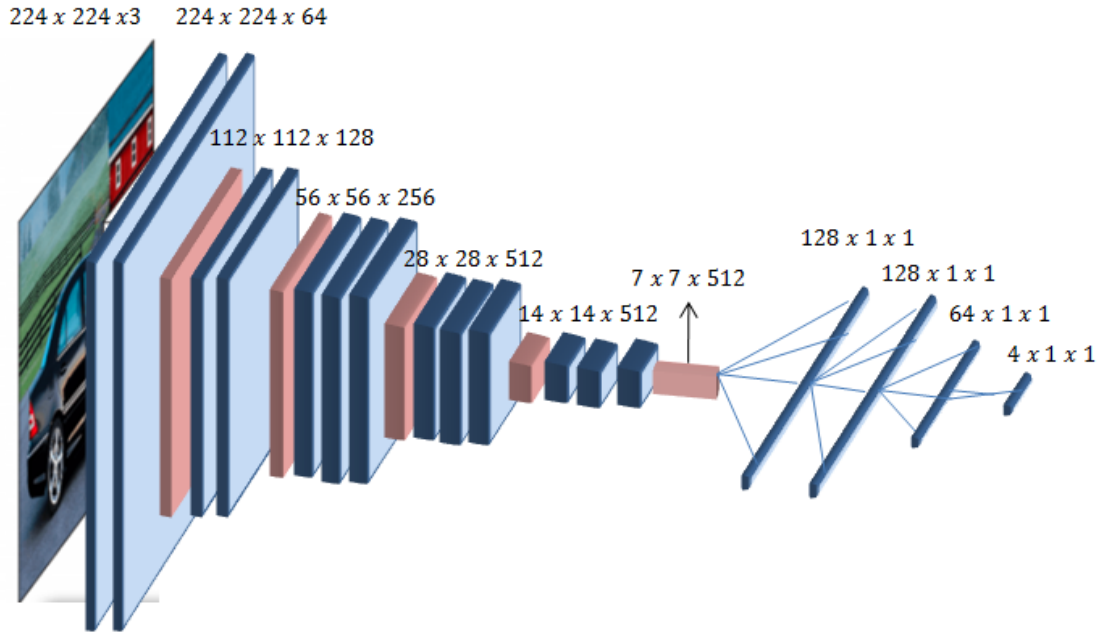

Figure 4: Vgg-16 Architecture together with our added dense layers. Resulting the final architecture

Our last fully connected layers consists of the following dense layers,
$7\ x\ 7\ x\ 512\ \rightarrow 128\ x\ 1\ x\ 1$
$128\ x\ 1\ x\ 1\ \rightarrow 128\ x\ 1\ x\ 1$
$128\ x\ 1\ x\ 1\ \rightarrow 64\ x\ 1\ x\ 1$
$64\ x\ 1\ x\ 1\ \rightarrow 4\ x\ 1\ x\ 1$

Here the last layer $4\ x\ 1\ x\ 1$ is our output layer which contains 4 consecutive parameters that represent a pixel coordinates within the range of $0 \leq \theta \leq 1$. This θ parameter values are our hypothesis for the bounding box relative position regarding the input image.
While being represented as a somehow percentage of the numeric hypothesis, what it represents is it's portion from the image shape (height and width).

This representation for the parameters is easier for the network to back digest and output. And it help the network learn by maintaining a some of regularization on the loss value, hence an inner natural regularization on the gradient of the current learning stage.

Now, let's take a look at the decomposition of our added layers.

The additional dense layers of the architecture are the following,

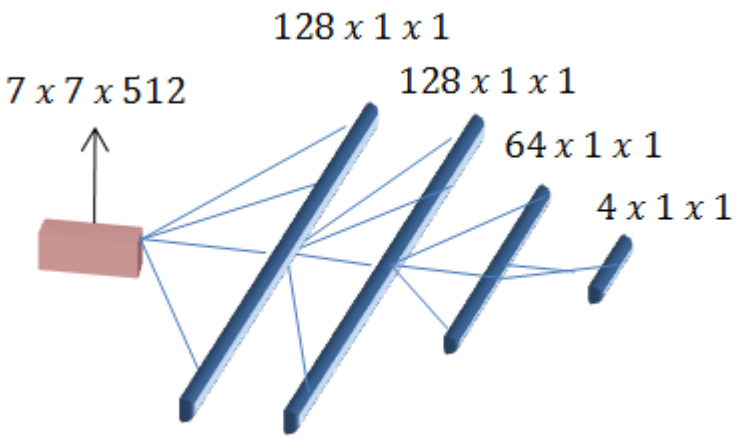

Figure 5: A zoom in to the added layers

The above layers alone contains around $\sim 3M$ learnable parametes. Which is about $\sim \frac{1}{4}$ of the total parameters of the whole deep model.
Such high dimensionality needs some kind of stability. Some of our approaches include a Bach Norm hidden layer, Relu activation function at the middle layers, a Drop Out in order to reduce over-fitting and generalize, and a Sigmoid activation function at the end.

Let's summarize the different components of our added layers network. And visualize the whole process of the feed-forward features.

Network Summary:

$1st\ Layer$:
$Fully\ Connected$: $7\ x\ 7\ x\ 512\ \rightarrow 128\ x\ 1\ x\ 1$
$32\ Batch\ Norm$: $32 - BN(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$
$Relu\ activation\ function$: $Relu(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$
$DropOut - \frac{1}{4}\ ratio$: $Drop(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$

$2nd\ Layer$:
$Fully\ Connected$: $128\ x\ 1\ x\ 1 \rightarrow 128\ x\ 1\ x\ 1$
$32\ Batch\ Norm$: $32 - BN(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$
$Relu\ activation\ function$: $Relu(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$
$DropOut - \frac{1}{4}\ ratio$: $Drop(128\ x\ 1\ x\ 1) \rightarrow\ 128\ x\ 1\ x\ 1$

$3rd\ Layer$:
$Fully\ Connected$: $128\ x\ 1\ x\ 1 \rightarrow 64\ x\ 1\ x\ 1$
$32\ Batch\ Norm$: $32 - BN(64\ x\ 1\ x\ 1) \rightarrow\ 64\ x\ 1\ x\ 1$
$Relu\ activation\ function$: $Relu(64\ x\ 1\ x\ 1) \rightarrow\ 64\ x\ 1\ x\ 1$

$4th\ Layer$:
$Fully\ Connected$: $64\ x\ 1\ x\ 1 \rightarrow 4\ x\ 1\ x\ 1$
$Sigmoid\ activation\ function$: $\sigma(4\ x\ 1\ x\ 1) \rightarrow\ 4\ x\ 1\ x\ 1$

We can see the detailed decomposition of our last layers above.
In code,

```python
class LicensePlateDetectionNN(nn.Module):
    def __init__(self):
        super(LicensePlateDetectionNN, self).__init__()
        self.vgg16 = vgg16(pretrained=True)

        # Neglecting (dis-including) the last 3 fully connected layers (Not including the 7x7x512 Layer)
        self.classifier = Sequential(*list(self.vgg16.features.children()),
                                     self.vgg16.avgpool)

        # Freezing the weights in the transferred VGG16 Neural Network
        # for param in self.classifier.parameters():
        #     param.requires_grad = False

        self.fc1 = Linear((7 * 7 * 512), 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.fc2 = Linear(128, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc3 = Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.fc4 = Linear(64, 4)  # Our output coordinates (x_t, y_t, x_b, y_b)
        self.drop_out = Dropout(0.25)  # Dropout neurons with probability P = 0.25

    def forward(self, x):
        x = self.classifier(x)
        x = torch.flatten(x, 1)
        x = nn_func.relu(self.bn1(self.fc1(x)))
        x = self.drop_out(x)
        x = nn_func.relu(self.bn2(self.fc2(x)))
        x = self.drop_out(x)
        x = nn_func.relu(self.bn3(self.fc3(x)))
        x = sigmoid(self.fc4(x))

        return x
```

Figure 6: The code of the final architecture

For better understanding, let's visualize those layers,

The first layer,

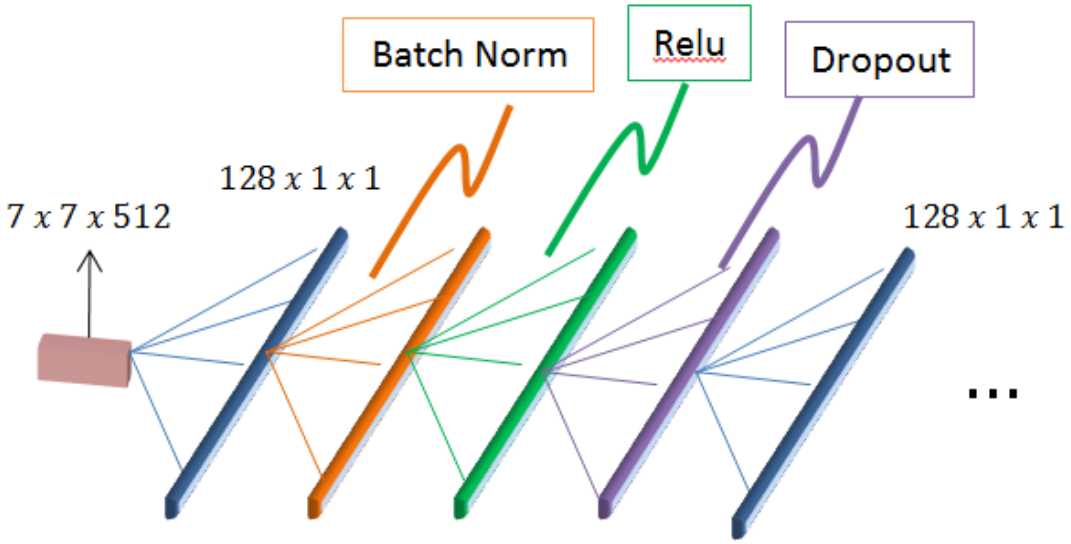

Figure 11: Detailed visualization of the components of the 1st Layer

The second layer,

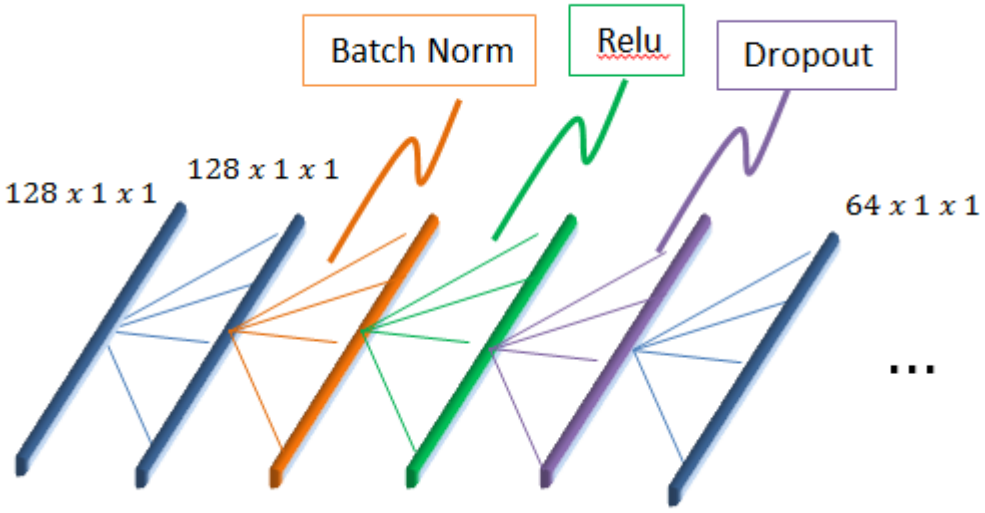

Figure 12: Detailed visualization of the components of the 2nd Layer

The third layer,

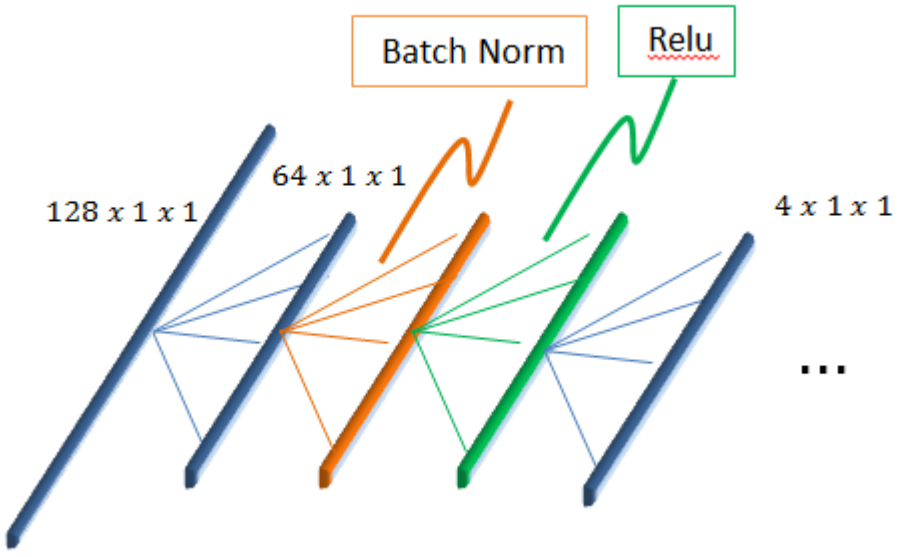

Figure 13: Detailed visualization of the components of the 3rd Layer

The fourth and last layer,

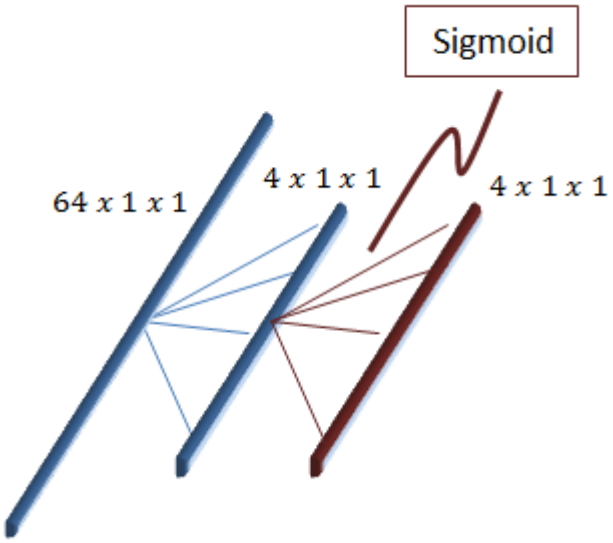

Figure 14: Detailed visualization of the components of the 4th Layer

And for the trainable part of the network, as mention before, we train from scratch our added layers described above and in addition we fine-tune the whole Vgg-16 ImageNet pre-trained network.

As described,

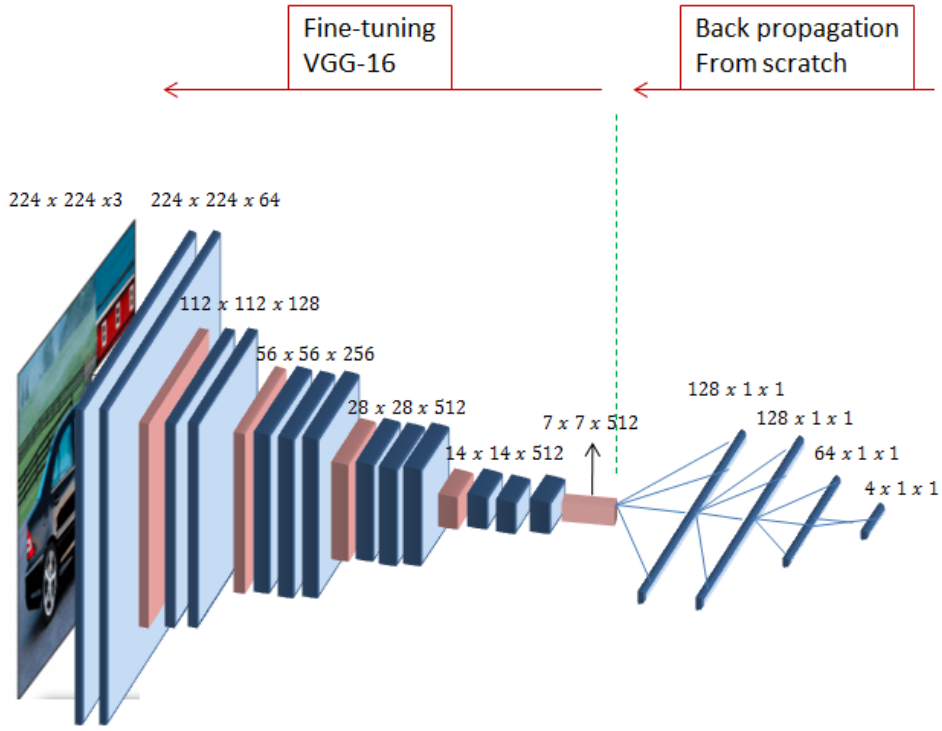

Figure 15: Backpropagation of the deep neural model

Data set:

The data set that was given to us is in the Kaggle turks url page –

https://www.kaggle.com/dataturks/vehicle – number – plate – detection/download

Apparently the turks have blocked the requests and edit the proxy for fetching to data set server. Part of the code for defining an agent and Mozilla token for the browser (Mozilla-compatible),

```
req = urllib.request.Request(row["content"], headers={'User-Agent': 'Mozilla/5.0'})
webpage = urllib.request.urlopen(req, timeout=10).read()
```

Figure 16: Attempt for a server database request with agent and Mozilla token

After searching we've managed to get a similar data set, work on it manually, pre-process and defining names and corresponding labels for the images.

Because of the artificial preparation of the data, not all the images are good for training our model, and some are a bit problematic.

Exploring some of the images in our data and it's corresponding labels,

Figure 17: A sample of the problematic images in the data set

There is quite a good portion of such images in our data, which makes it challenging to train a deep neural model and get good results. Not just that the data size is relatively (to our network\task) very small, but also has a significant amount of problematic images.

Other than that, some of the other images in our data,

Figure 18: Some other images in the data set

Even for the "good" representation of the images, there are such images which its license plate color is the same as the background\car, and others that they're somehow hidden in background by some other noisy objects.

Metrics:

In order for the deep model to adjust, update its parameters, and learn, We need to define some metrics for this kind of task. Some Convex Loss function (after all we're updating the model weights with the gradient descent method. Using Adam optimizer to be specific), regularization terms, accuracy Euclidean metric etc.

Regularization terms,

$\lambda = \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 \quad , \qquad Where,$

$$\lambda_i = \beta \cdot \sum_{j=0}^{k-1} \left\| W^j{}_{FC\_i^{th}\_Layer} \right\|_2 \quad , \quad \forall\, i \in \{1,2,3,4\},\ j \in \{0, \ \dots \ , k\}$$

$k = size(FC\_i^{th}\_Layer)$
$\beta - regularization\ magnitude$
In our case, $\beta = 10^{-6}$

$$\|W\|_n = \|\{w_1, \dots, w_k\}\|_n = \sqrt[n]{|w_1|^n + \cdots + |w_k|^n}$$

Hence in our case,
$$\|W\|_2 = \|\{w_1, \dots, w_k\}\|_2 = \sqrt[2]{|w_1|^2 + \cdots + |w_k|^2}$$

$FC\_i^{th}\_Layer \ - Is\ the\ fully\ connected\ layer\ i^{th}\ in\ number$

Loss objective function,

$$L = \frac{1}{m} \cdot \sum_{i=0}^{n-1} \left(y_{hyp\_i} - y_{label\_i}\right)^2, \qquad n = d^2$$

$d - The\ dimension\ of\ the\ euclidean\ space$
In our case, $d = 2$

Hence, full loss metric,

$L_{total} = L + \lambda$, Where $L\ and\ \lambda\ as\ described\ above$

Optimizer,

We used Adam Optimizer for updating our model weights,

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\eta}{\sqrt{\widehat{v_t} + \varepsilon}} \cdot \widehat{m_t} \quad , \quad \alpha = 3 \cdot 10^{-4} \left(\sim \frac{1}{3} \; of \; 10^{-3} \right)$$

Where,

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t} \quad , \quad \widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

And,

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_{t-1}$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_{t-1}^2$$

$$g_t = \nabla L_{total}(\theta_t) = \frac{\vartheta L_{total}}{\vartheta \theta_t}$$
$$g_t - The \; relative \; gradient \; in \; step \; t$$

Accuracy,

$$\tau = \frac{1}{n \cdot m} \cdot \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} f\left(\bar{y}_{hyp_i} - \bar{y}_i\right) \quad , \quad n = d^2 \quad , \quad m = size(frwd \; data)$$

$$f(\bar{x}) = \begin{cases} 1 & , \quad \dfrac{sum(\bar{x})}{W^{-1} \cdot \|\bar{x}\|} < \rho \cdot W \quad , \quad \rho = 10 \; x10^{-2} \\ 0, & Otherwise \end{cases}$$

$$\rho - \; Accuracy \; Precentage$$

Results:

After pre-processing the data, creating a corresponding labeling, building our model's architecture, determining and applying the Euclidean metrics and loss functions, we obtained the following results under the specified below obligations,

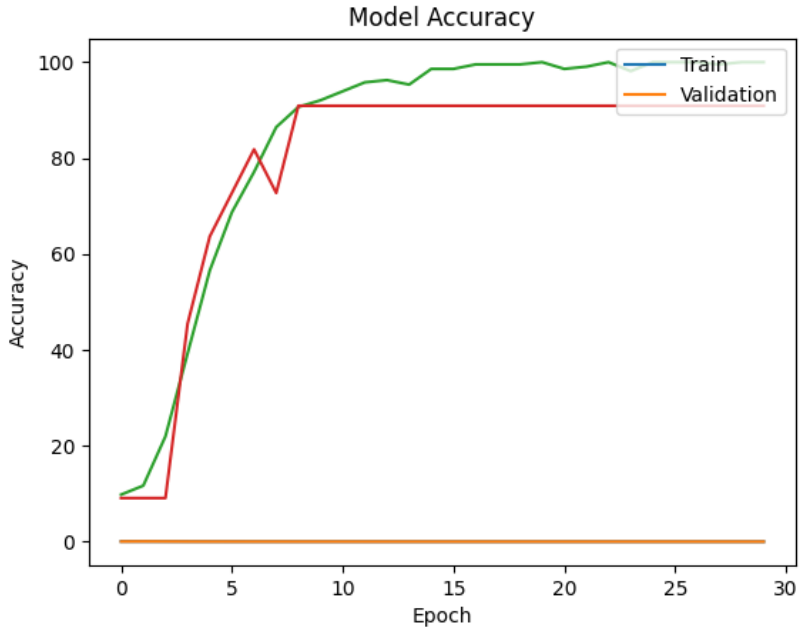

Figure 19: Model's Accuracy graph across 30 Epochs

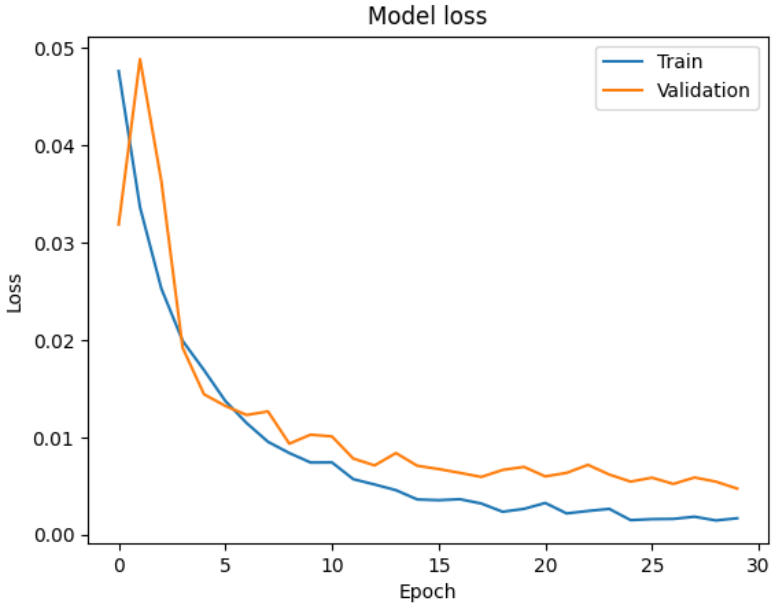

Figure 20: Model's Loss graph across 30 Epochs

Visualize the predicted results,

Figure 21: Model's Predictions on the Validation data set

Figure 22: Model's Predictions on the Test data set

Final Thoughts:

Building such a model was definitely challenging specially with all the different wrappers around.
The data wasn't reachable and we had to improvise (though it was far from being perfect), which took quite some time to get it ready and it was very small for such a complex task. The infrastructure and resources were in lack, each "code run" took hours, which made it feasible only through nights, so architecture correction and bugs fixing were kind of slow.

But yet, despite that, we managed to build and train a good generalizing model. The results were pretty accurate and the optimization graphs during the course of training acts as expected.

We could have taken a pre-trained result-ready off the shelf solution architecture as discussed earlier in the suggested solutions section – "Possible Approaches". But we wanted to challenge our self and not rely on other's work and optimizations (a simple fit-predict process). It was important for us to get our own results and build our own solution with fully functional optimizations, metrics, and a new course of training.

"Artificial intelligence is built upon the idea of a functional neural brain cell. Deep learning will lead to deep understanding, while shallow observations might work just for now, but won't last for the long run".

M.A