Grasp Detection using CNNs

COMP341 - Robot Perception and Manipulation - Assignment 1
Report documentation

Team Members		
Student Name	**Student ID**	**Contributions**
Conor O'Sullivan	201411716	Rectangle Metric Implementation & Report
Elena Manoli	201426062	Grasp Visualization and Report
Hadi Aljishi	201387758	Background Augmentation, Dataset Loading, Object Classification, Depth Image Network
Mihai Tranca	201437789	Neural Network Design
Mike Semple	201352715	Neural Network Design, Network Training and Testing, Accuracy Improvement

Table of contents:

Step 1. Grasp Detection using CNN + RGB image

The first step is to set the seed of the random number generator. This ensures that the same permutation of numbers is produced every time in order to get reproducible results. We also make sure that our architecture is making use of the GPU (Cuda).

```
def set_seed(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():  # GPU operation have separate seed
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)


set_seed(42)
```

```
try:
    from google.colab import drive
    drive.mount('/content/drive')
    DATASET_PATH = "drive/MyDrive/Liverpool/Assignments/Year 3/COMP341/COMP34
    device = torch.device("cpu") if not torch.cuda.is_available() else torch.
    !pip install shapely
except:
    print('not in drive')
    DATASET_PATH = 'ass1_data/Data'
    device = torch.device("cpu") if not torch.cuda.is_available() else torch.
print("Using device", device)
```

Figure 1.1 Seeding the random number generator

The Dataset:

The dataset used in this project consists of a number of ‘scenes’ for different objects in different orientations. Each scene contains an RGB image of the object in question, a text file listing possible grasps, as well as a mask and depth images for extra training.

```
427.89104;439.45935;68.815;24.0;26.0995
427.89104;439.45935;68.815;24.0;13.0498
427.89104;439.45935;68.815;24.0;39.1493
```

Figure 1.2 Grasp file example

Each line in the grasp file represents one grasp using 5 numbers. The x,y position is represented by the first two numbers. The angle is the third number, while the jaw opening and size are the fourth and fifth numbers respectively. With that in mind, we were able to load the grasps into python lists as follows.

```
def load_grasps(grasp_file):
  processLine = lambda line: np.array([float(number) for number in line.split(";")])
  with open(grasp_file) as file:
    grasps = np.array([processLine(line) for line in file.readlines()], dtype=np.float32)
  return grasps
```

Figure 1.3 Loading the grasps

Each data point in our set is a single scene represented by an RGB image, the grasps for that scene padded to the maximum grasp number, the object label, and optionally, a depth image of the user's choosing (either perfect or stereo depth).

Background Augmentation:

The dataset provided in this assignment proved too small for training our network. To resolve this, we used the image masks provided to perform background augmentation and expand the training set. This is done by first performing a bitwise AND operation on the original RGB image and its corresponding mask to get the masked foreground. We then invert the mask and perform another bitwise AND, this time with the background image to get the masked background. Finally, we carry out a bitwise OR to combine the masked foreground and background images together.

```
def generate_augmented_images(root_dir, bk_dir, num_bk):
    for object_name in os.listdir(root_dir):
        for scene_index in range(5):
            scene_root = f"{root_dir}/{object_name}/{scene_index}_{object_name}_"

            # Make sure a scene with this index exists for this object
            rgb_path = scene_root + "RGB.png"
            if not os.path.exists(rgb_path): continue

            # Load original image and mask
            rgb = cv2.imread(rgb_path)
            mask = cv2.imread(scene_root + "mask.png")

            for bk_index, background in enumerate(os.listdir(bk_dir)):

                # Load the background image
                bk = cv2.imread(os.path.join(bk_dir, background))[:1024, :1024]

                # Get masked foreground
                fg_masked = cv2.bitwise_and(rgb, mask)

                # Get masked background
                mask = cv2.bitwise_not(mask) # invert mask
                bk_masked = cv2.bitwise_and(bk, mask)
                mask = cv2.bitwise_not(mask)  # revert mask to original

                # Combine masked foreground and masked background
                final = cv2.bitwise_or(fg_masked, bk_masked)
                cv2.imwrite(scene_root + f"RGB{bk_index}.png", final)
```

Figure 1.4 Background Augmentation

With a total of 5 new background images for each existing RGB image, we can effectively expand our dataset to 6 times its original size.

Network Architectures:

The first architecture we chose to implement is the Convolution Neural Network. The convolution layer is the one that applies the filter to the original image. A feature map is created by applying a filter to the image many times. Pooling is used to minimize the size of the feature map after the convolutional layers have been applied. The main purpose of pooling is to reduce the number of parameters of the input tensor. Batch Normalization is a layer that normalizes the inputs of each batch to make the network more stable and faster to train. The input to the fully connected layers is the output from the final pooling layers which is flattened and then fed into the fully connected layer.

```
class ConvNN(nn.Module):
    def __init__(self):
        super(ConvNN, self).__init__()

        # Input image is size 1024x1024
        self.conv1 = nn.Conv2d(3, 64, kernel_size=6, stride=2, padding=0) # output is 510x510
        self.batchN = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # output is 255x255

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1) # output is 128x128
        self.batchN2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # output is 64x64

        self.conv3 = nn.Conv2d(128, 128, kernel_size=2, stride=2, padding=0) # output is 32x32
        self.batchN3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # output is 16x16

        self.fc1 = nn.Linear(128 * 16 * 16, 5)
```

Figure 1.5 CNN

The forward function is called which applies all the layers sequentially including the fully connected.

```
def forward(self, x):

    x = self.conv1(x)
    x = self.batchN(x)
    x = F.relu(x)
    x = self.pool1(x)

    x = self.conv2(x)
    x = self.batchN2(x)
    x = F.relu(x)
    x = self.pool2(x)

    x = self.conv3(x)
    x = self.batchN3(x)
    x = F.relu(x)
    x = self.pool3(x)

    # flatten tensor
    x = x.view(x.size()[0], -1)

    x = self.fc1(x)

    return x
```

Figure 1.6 Forward Function

The Deep Residual Network was the second network architecture we tested. Skip connections are used by ResNet. The skip connection skips a few layers of training and connects directly to the output. This means that if any layer has an impact on architecture performance, it will be skipped by regularization. To begin, we created a block to represent the skip connection. As you can see in the figure below we applied an activation function after skip connection.

```
class ResNetBlock(nn.Module):
    def __init__(self, depth_in, activation_func, depth_out=-1):
        super().__init__()
        self.downSampleResidul = True
        if depth_out == -1:
            depth_out = depth_in
            self.downSampleResidul = False

        self.resBlock = nn.Sequential(
            nn.BatchNorm2d(depth_in),
            activation_func(),
            nn.Conv2d(depth_in, depth_out, kernel_size=3, padding=1, stride=1 if not self.downSampleResidul else 2, bias=False),
            nn.BatchNorm2d(depth_out),
            activation_func(),
            nn.Conv2d(depth_out, depth_out, kernel_size=3, padding=1, bias=False)
        )

        self.downsampleRes = nn.Sequential(
            nn.BatchNorm2d(depth_in),
            activation_func(),
            nn.Conv2d(depth_in, depth_out, kernel_size=1, stride=2, bias=False)
        )

    def forward(self, x):
        z = self.resBlock(x)
        if self.downSampleResidul:
            x = self.downsampleRes(x)
        return z + x
```

Figure 1.7 Single ResNet Block

The overall ResNet architecture is made up of several ResNet blocks of which some are downsampling the input.

```
self.block2 = ResNetBlock(32, nn.SiLU, 64)      #out = 127x127
self.block3 = ResNetBlock(64, nn.SiLU, 128)     #out = 64x64
self.block4 = ResNetBlock(128, nn.SiLU, 256)    #out = 32x32
self.block5 = ResNetBlock(256, nn.SiLU, 256)    #out = 16x16
```

Figure 1.8 ResNet Architecture

This network also performs object classification alongside regression, as knowing the type of object can give us information about how it could be grasped. This is done by splitting the network into two branches after feature extraction is done. Each branch is then fed to a different block, one for regression and one for classification.

```
        self.regressionBlock = nn.Sequential(
            nn.Dropout(0.3),
            nn.Linear(self.flattened_size, 1024),
            nn.Dropout(0.2),
            nn.SiLU(inplace=True),
            nn.Linear(1024, 5 if mode == 'train' else 1000),
        )

        self.classificationBlock = nn.Sequential(
            nn.Linear(self.flattened_size, 1024),
            nn.ReLU(),
            nn.Linear(1024, 10),
        )

    def forward(self, x):
        x = self.featureExtractionBlock(x)
        x = self.pool(x)
        x = x.view(-1, self.flattened_size)
        if self.applyFC:
            grasp = self.regressionBlock(x)
            classification = self.classificationBlock(x)
            return (grasp, classification)
        return x
```

Figure 1.9 Multi-head ResNet

The 3rd and final architecture we implemented was a De-Convolution (or transposed convolution) Residual Network. This uses a very similar first section to the Res Net described above, with slightly different channel sizes and less downsampling.

```
class DeConvResNet(nn.Module):
    def __init__(self, dropout=0.3):
        super(DeConvResNet, self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=0), #out = 511x511
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(16),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=0), #out = 509x509
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0), #out = 507x507
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 32, kernel_size=3, stride=2, padding=0) #out = 253x253
            # nn.MaxPool2d(kernel_size=3, stride=2, padding=0) #out = 253x253
        )
        self.featureExtractionBlock = nn.Sequential(
            ResNetBlock(32, nn.SiLU, 64),      #out = 127x127
            ResNetBlock(64, nn.SiLU, 128),      #out = 64x64
            ResNetBlock(128, nn.SiLU, 128),     #out = 32x32
            ResNetBlock(128, nn.SiLU),    #out = 32x32
            ResNetBlock(128, nn.SiLU, 128),    #out = 16x16
        )
```

Figure 1.10 De-Convolutional Network

After a similar initial section, we then add a deconvolution block, which up-samples the outputs from 16x16 to 64x64.

```
self.convTrasposeBlock = nn.Sequential(
    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, output_padding=1),
    nn.BatchNorm2d(64),
    nn.SELU(inplace=True),
    nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=2, output_padding=1),
    nn.BatchNorm2d(32),
    nn.SELU(inplace=True),
    nn.ConvTranspose2d(32, 32, kernel_size=9, stride=1, padding=4)  #out = 64x64
)
```

Figure 1.11 - De-Convolution Block

Then after this, the network splits into 4 heads, one for the position vector (x,y), one for the angle, one for the height, and one for the width.

```
self.pos_conv = nn.Conv2d(32, 2, kernel_size=2)
self.angle_conv = nn.Conv2d(32, 1, kernel_size=2)
self.width_conv = nn.Conv2d(32, 1, kernel_size=2)
self.hight_conv = nn.Conv2d(32, 1, kernel_size=2)

self.pos_output = nn.Sequential(nn.Linear(self.flattened_size*2, 512), nn.Linear(512, 2))
self.angle_output = nn.Sequential(nn.Linear(self.flattened_size, 512), nn.Linear(512, 1))
self.width_output = nn.Sequential(nn.Linear(self.flattened_size, 512), nn.Linear(512, 1))
self.hight_output = nn.Sequential(nn.Linear(self.flattened_size, 512), nn.Linear(512, 1))
```

Figure 1.12 - Network Head Split

Each head is very similar in the fact it is passed through a convolution then flattened then passed through a linear layer.

```
def forward(self, x):
    x = self.block1(x)
    x = self.featureExtractionBlock(x)
    x = self.convTrasposeBlock(x)
    pos_output = self.pos_output(self.pos_conv(self.dropout(x)).view(-1, self.flattened_size*2))
    angle_output = self.angle_output(self.angle_conv(self.dropout(x)).view(-1, self.flattened_size)).flatten()
    width_output = self.width_output(self.width_conv(self.dropout(x)).view(-1, self.flattened_size)).flatten()
    hight_output = self.hight_output(self.hight_conv(self.dropout(x)).view(-1, self.flattened_size)).flatten()
    return pos_output, angle_output, width_output, hight_output
```

Figure 1.13 - Head Flattening

Having one head for each different type of output makes the network better able to predict each section of the grasp, with a better position, angle and height/width predictions.

Training:

An RGB image is fed into the network as input. There are three channels in a color image: red, green, and blue. We must normalize the image before passing it to our models, which involves converting the range. We used Pytorch vision to be able to make adjustments to the image. The transforms.ToTensor() method converts the image to a tensor. The transforms.Resize() and transforms.CenterCrop() methods resize the image. Finally, we used transforms.Normalize() to normalize the image by passing the sequence of means and the sequence of standard deviation for each channel.

```
def gen_transforms(resize=None, imgs_mean=(0.5,), imgs_std=(0.5,)):
    tst_trans = [tv.transforms.ToTensor()]
    if resize is not None:
        tst_trans.append(tv.transforms.Resize(resize[0]))
        tst_trans.append(tv.transforms.CenterCrop(resize[1]))
    tst_trans.append(tv.transforms.Normalize(mean=imgs_mean, std=imgs_std))
    return tv.transforms.Compose(tst_trans)
```

Figure 1.14 RGB image input

The get_train_loader() function is then defined to load the training data. PyTorch's DataLoader class is used to load and loop through the dataset. The dataset that will be used to train the model is the first parameter in the DataLoader class. The batch size is the second parameter, which defines how many training samples are used in each iteration.

```
    trainData = GraspDataset(root_dir=f'{DATASET_PATH}/training', transforms=
    trainLoader = td.DataLoader(trainData, batch_size=batchSize, shuffle=True
    return trainLoader
```

Figure 1.15 Data loading

We have two separate functions to train our models: *train()* for our ResNet model, and *train_no_classification()* for our CNN model. The model to be trained is the first parameter. The batch size specifies how many samples from the dataset will be used in the neural network's training. The number of epochs refers to how many times the learning algorithm will loop over the full training dataset. While progressing towards the loss function's minimum, the learning rate sets the step size at each iteration.

```
def train(model, batch_size, n_epochs, learning_rate, resize=None):
```

Figure 1.16 Train function

The get_train_loader() function is used to get the training data, while the getLoss() function is used to receive the loss function. The optimizer can also be obtained by using the getOptimizer() method.

```
trainLoader = get_train_loader(batch_size, resize)
lossFn = getLoss()
optimizer = getOptimiser(model, learning_rate)
```

Figure 1.17 Loading training data, loss Function, and optimizer

The getLoss() function lets us choose between MSELoss for regression and CrossEntropyLoss for classification.

```
def getLoss(regression=True):
    if regression:
        return nn.MSELoss()
    else:
        return nn.CrossEntropyLoss()
```

Figure 1.18 Getting Loss Function

For the getOptimizer() function, we used the optim.Adam by PyTorch which is an extension to gradient descent. As you can see, we attempted to use opt.SGD, however, it did not work as planned because Adams' method is an adaptive learning methodology. In standard SGD, however, the learning rate has a comparable influence on all of the model's weights.

```
def getOptimiser(model, learningRate):
    #optimiser = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9
    optimiser = optim.Adam(model.parameters(), lr=learningRate)
    return optimiser
```

Figure 1.19 Getting Optimiser Function

The first loop is used to repeatedly go through the entire training dataset. One iteration of this loop is called an epoch. The second loop is used to go over all the batches in a single epoch. Batches of data are fed into the model at the start. The outputs are acquired by forward propagating the input RGB images through the model. After that, the loss between the outputs and the ground truths is calculated. If classification is included in the training, the regression and classification losses are calculated separately and then summed together.

The backpropagation step is then completed using the backward() method, which computes the gradient during the neural network's backward run. The optimizer iterates through all of the parameters that need to be updated, changing their values as it goes. By adding the loss to the existing number, the total training loss and running loss are updated.

```
for epoch in range(n_epochs):  # loop over the dataset for n_epoch
    epoch_loss = 0
    epoch_accuracy = 0

    with tqdm(trainLoader, unit="batch") as tepoch:
        for rgb, grasp, label, sceneRoot, depth in tepoch:   #for each batch
            tepoch.set_description(f"Epoch {epoch}")
            # Move tensors to gpu
            rgb, depth, grasp = rgb.to(device), depth.to(device), grasp.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()
            # forward + backward + optimize
            outputs = model(rgb, depth)
            loss = lossFn(outputs,  grasp)
            loss.backward()
            optimizer.step()
            # add to running totals
            epoch_loss += loss.item()
            with torch.no_grad():
                accs = grasp_accuracy(outputs, grasp)
                epoch_accuracy += accs
    epoch_loss = epoch_loss / len(trainLoader)
    train_history.append(epoch_loss)
    accuracy_history.append(epoch_accuracy/len(trainLoader))
with torch.no_grad():
    accuracy_history = [i.cpu().numpy().item() for i in accuracy_history]
print(f"Training Finished, took {round(time.time() - training_start_time,1)}s")
return train_history, accuracy_history
```

Figure 1.20 Forward and Backward propagation

Step 2. Evaluate Results

We output two things from our network, the object classification and the scoring metric between our predicted grasp and the ground truth grasp.

Object Classification:

Like in the most traditional deep learning problem, we attempt to differentiate the different objects in our training data: the figure, the football, the phone, etc… This allows the network to achieve a better grasp, by understanding the type of object we're grasping. A classical example of this is when training a robot to grasp a cup. While many grasps may be valid, the location of the grasp may cause the contents of the glass to spill, thereby achieving a failure. By understanding the object to be a cup, we can train our robot to grasp the correct area.

Figure 2.1 - Grasp contextualization

This same observation can apply to many objects, so we can use it in our own solution.
We do this by splitting our output into two groups, one of which will find a grasping point on the object, while the other will attempt to identify that object. We then combine the loss from both of these groups in order to more neatly train the neural network.

Identifying the object can also help when checking the accuracy of the grasp. If we can find the correct object and its scene (as we have multiple images for each object), we can load that object's grasp text file, and compare the possible truthful grasps, with the grasp our model has output. Clearly it would not be possible for our model to attain a good accuracy with only one ground truth grasp, as there is no guarantee that the model will predict the same grasping point as the arbitrarily chosen ground truth from our list. By looping through our entire list of ground truths for a given object, we can return the highest accuracy found when comparing our prediction to each truth, thus we have a much higher chance of achieving a good accuracy.

Rectangle Metric:

To find the accuracy of our predicted grasp, we used the Rectangle Metric to evaluate the results of our predicted grasps from Step 1. This was found by taking as input the dimensions of the ground truth grasp and our model's predicted grasp, and taking the Jaccard index of these two rectangles. The formula for the Jaccard index for two rectangles A and B is:

$$\text{Jaccard Index} = \frac{(A \quad B)}{(A \quad B)}$$

Figure 2.2 Jaccard Formula

That is, the area of the intersection of both rectangles over the area of the union of both rectangles, naturally, if a perfect intersection of ground truth and prediction is achieved, this ratio will be equal to 1. We can visualize the Jaccard index like so:

(centerX, centerY, theta, height, width)

In this example:

A: (567.16686, 545.16015, 50.0129, 75.5, 45.4924)

B: (554.1098, 544.7016 , 57.21771 , 83.99756, 50.15486)

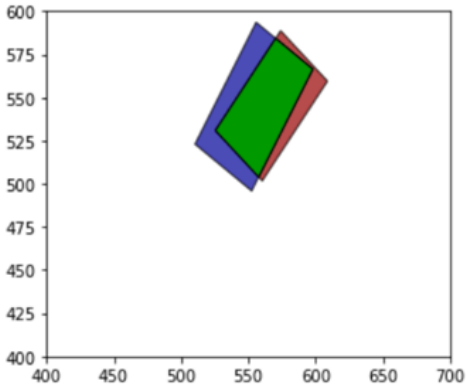

Figure 2.3 Jaccard Visualization

Where rectangle A, rectangle B, and the intersection are highlighted separately for clarity.

```
r1 = RotatedRect(cx1, cy1, angle1, w1, h1)
r2 = RotatedRect(cx2, cy2, angle2, w2, h2)

intersection = r1.intersection(r2).area
union = (r1.width*r1.height) + (r2.width*r2.height) - intersection
score = intersection/union
```

intersection area: 2687.210926398685
union area: 4960.351135742915
Jaccard Index: intersection/union = 0.5417380449209307

Figure 2.4 Score calculation

We encountered some challenges in implementing this scoring mechanism in raw python, as there was no consistent shape that the resulting overlap could be, a random rotation between the two input rectangles, nor any guarantee that the two rectangles overlap at all.

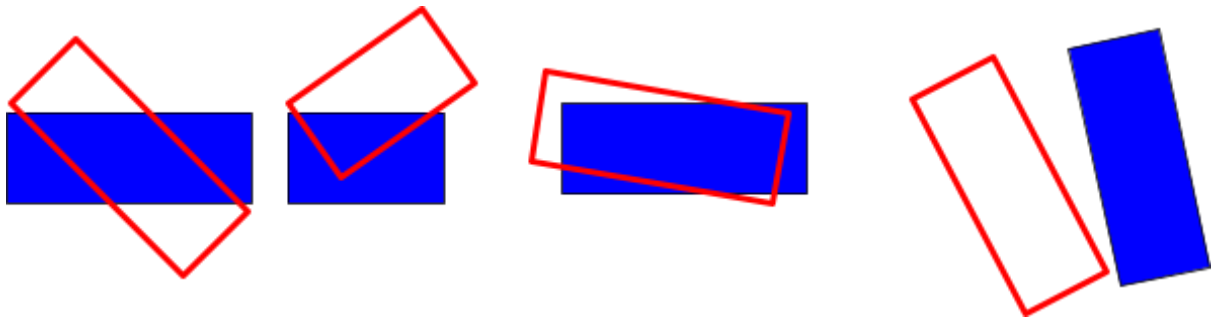

Figure 2.5 - overlap combinations (trapezium, triangle, compound shape, no overlap)

This meant that there wasn't a simple formula that could be used to find the area of the resulting polygon. Many of the online resources were dependent on the two rectangles having the same orientation, which was clearly insufficient for our purposes.

This led us to use python's shapely library to more easily track the areas of each rectangle, as well as that of our intersection, this means that we can easily evaluate the results regardless of the size or overlapping shape of the two rectangles.

The conditions which quantified a "good" grasp prediction were when the Jaccard index was greater than 0.25 (25% overlap), and the angle of rotation between both rectangles is less than 30%. These are the same metrics as in the slides, implemented with the following lines:

```
if math.fabs(angle1 - angle2) > 30:
    print("angle of rotation between ground truth grasp and predicted grasp is greater than 30 degrees, this is a poor grasp estimation")
    print()
if score < 0.25:
    print("The Jaccard index is less than 25%, this is a poor grasp estimation")
```

Figure 2.6 Additional Constraints

The plot_epochs() is the function used for plotting the training loss and validation loss. As you can see we use the plot function available in matplotlib to achieve that.

```
def plot_epochs(train_history, validation_history=None, plotType="loss"):
    x = np.arange(1, len(train_history) + 1)
    plt.figure(figsize=(8, 6))
    if validation_history:
        plt.plot(x, val_history, color=colors[1], label=f"Validation {plotTyp
    plt.plot(x, train_history, color=colors[0], label=f"Training {plotType}",
    plt.ylabel(f'{plotType}')
    plt.xlabel('Epoch')
    plt.legend(loc='upper right')
    plt.title(f"Evolution of the training and validation {plotType}")
    plt.show()
```

Figure 2.7 Plot_epochs function

We apply this to each architecture we instantiated in step 1: the Visual Geometry Group, and the Deep Convolutional Neural Network. Testing their accuracy on the training data gives the following results:

Convolutional Network: ResNet: De-Convolutional:

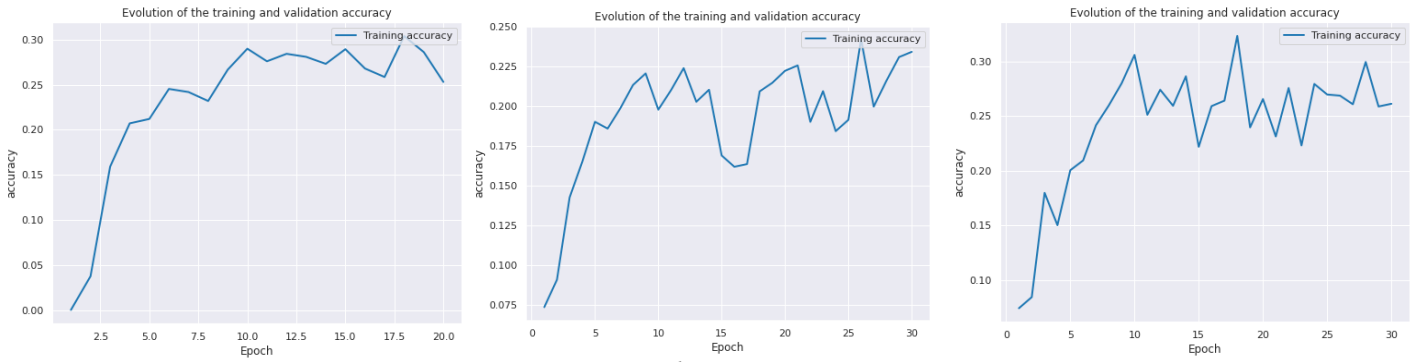

Figure 2.8 - Architecture Accuracies

Convolutional Network: ResNet: De-Convolutional:

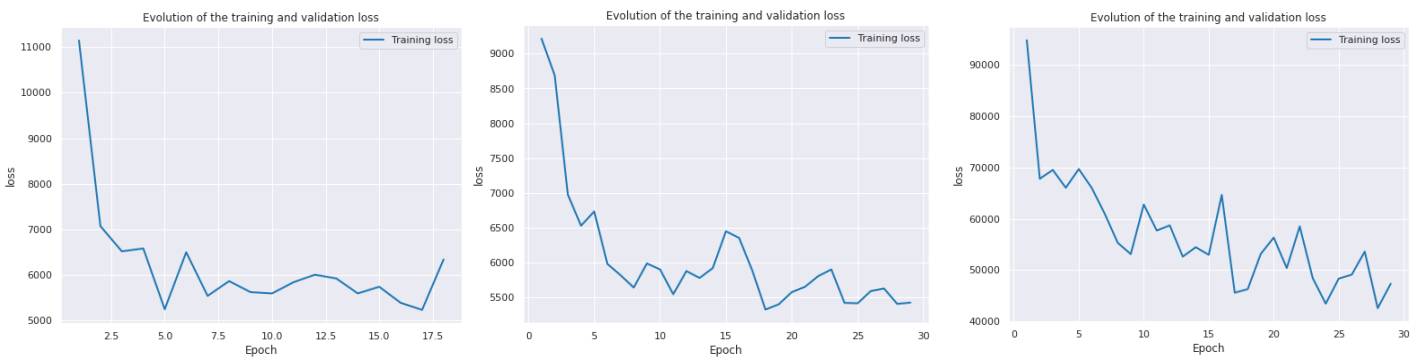

Figure 2.9 Architecture Loss

Final Accuracy Percentages:

Network	Training	Testing
Conv Net	28.9%	27.4%
Res Net	24.9%	21.4%
DeConvRes Net	32.3%	28.9%

Visualizing the results:

We can view both the ground truth grasp and our prediction over the input image itself for further clarity. By loading the image into python, as well as taking the inputs of both grasps, we simply use the function draw_box() to reference our results:

```
def draw_box(image,center_point, rotate, width, height, color):


    angle = np.radians(rotate)
    # Determine the coordinates of the 4 corner points
    rotated_rect_points = []
    x = center_point[0] + ((width / 2) * cos(angle)) - ((height / 2) * sin(angle))
    y = center_point[1] + ((width / 2) * sin(angle)) + ((height / 2) * cos(angle))
    rotated_rect_points.append([x,y])
    x = center_point[0] - ((width / 2) * cos(angle)) - ((height / 2) * sin(angle))
    y = center_point[1] - ((width / 2) * sin(angle)) + ((height / 2) * cos(angle))
    rotated_rect_points.append([x,y])
    x = center_point[0] - ((width / 2) * cos(angle)) + ((height / 2) * sin(angle))
    y = center_point[1]- ((width / 2) * sin(angle)) - ((height / 2) * cos(angle))
    rotated_rect_points.append([x,y])
    x = center_point[0] + ((width / 2) * cos(angle)) + ((height / 2) * sin(angle))
    y = center_point[1] + ((width / 2) * sin(angle)) - ((height / 2) * cos(angle))
    rotated_rect_points.append([x,y])

    rotatedImg = cv2.polylines(image, np.array([rotated_rect_points], np.int32), True, color, thickness)
    return rotatedImg
```

Figure 2.10 - draw_box() function

Below are some images from an early epoch of the project, with the ground truth highlighted red, and our prediction in green:

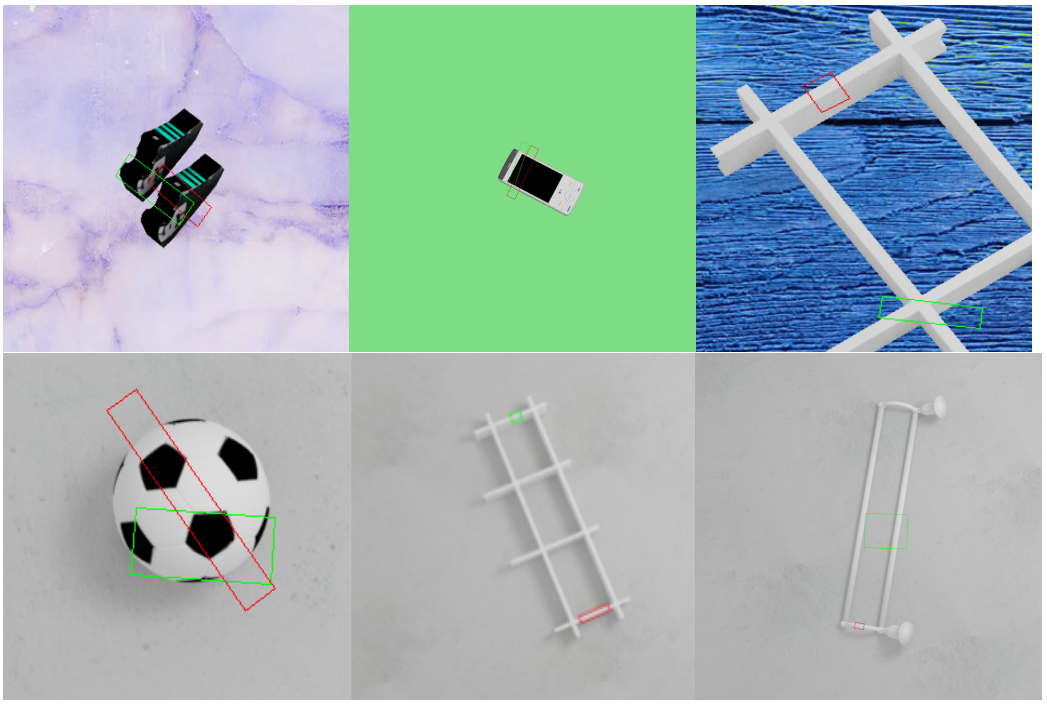

Figure 2.11 - Visualizing on training data

Clearly with these images, our model has predicted a fairly accurate grasp, albeit in the wrong areas. One major issue we had over the course of this project was, even when compared to each of the grasps featured in the images' file, we were unable to find one with a suitable match. This meant that on paper, our accuracy was fairly low, despite being a good estimate for a grasping point when looking at the object in isolation.

Step 3. Grasp Detection using CNN + RGB and Depth Image

Our RGB and depth image model uses the same ResNet RGB model from step 1, duplicated to take two input images instead of one, RGB and depth. The only difference between the RGB side and depth image side is the number of input channels in the first layer which must be changed to account for the difference in dimensionality. The inputs of the two sides are concatenated directly after extracting the features and flattening to a one-dimensional tensor. The result is then fed to the final regression and classification blocks.

```
def forward(self, rgb, x):
  rgb_output = self.rgb_side(rgb) # Feed first input to RGB side
  x = self.featureExtractionBlock(x)
  x = self.pool(x)
  x = x.view(-1, self.flattened_size)
  x = torch.cat((rgb_output, x), 1) # Merge the two sides into one tensor
  grasp = self.regressionBlock(x)
  classification = self.classificationBlock(x)
  return (grasp, classification)
```

Figure 3.1 Combine RGB and depth

To train this network, we use an additional training function: *train_depth()*. This is similar to our normal training function, but it feeds the network depth images as well as RGB images.

Including the depth means that the network has more data to work with, as such, we should generalize to more accurate grasps, as the learner can more easily distinguish the object from the image's background. Comparing the accuracy of this combined network with our network from step 1 yields the following results:

Step 1: Step 3:

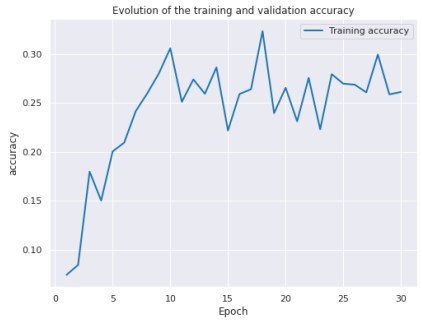

Figure 3.2 Step 1 & Step 3 Comparison

Comparing the Step 3 results with our best performing Step 1 architecture, the de-convolutional network, shows better results.

Step 4. Grasp Detection in the Wild

In Practice, it is imperative that the end model be able to predict a good grasp on unseen data points, not just on the images that we were given to train, to this effect, we found more images to test our completed model on:

Figure 4.1 - sample of Cornell Images

These images are from the Cornell Grasp Dataset, and are a great fit for our purposes. There are a variety of objects to test, which should provide a good example as to the competency of our dataset. However, there is no depth map accompanying each of these images, so feeding them into the our trained model from step 1, we have:

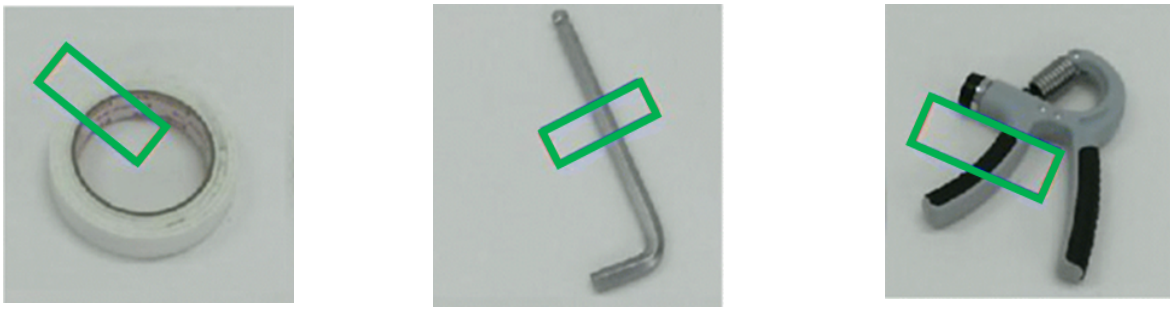

Figure 4.2 Grasp Visualization on unseen data

These predictions are good estimates of grasping points, as, for the most part they appear over the object in a reasonable location for the object to be picked up from. While our model had a poor accuracy when compared to the ground truth grasps of the training data, its effect on unseen data is far better, so it generalizes well as a grasp detection system.

Model Improvements:

We discovered numerous ways to improve our findings over the course of the project. One way was by switching out the activation function of our network from RELU to SELU, that

is, switching from the Rectified Linear Unit to the Scaled Exponential Linear Unit. SELU was first proposed in 2017 and uses a different approach to prevent vanishing gradients and dead neurons. It does this by maintaining the mean and variance between layers, thereby internally normalizing the network. The SELU function is defined as the following:

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

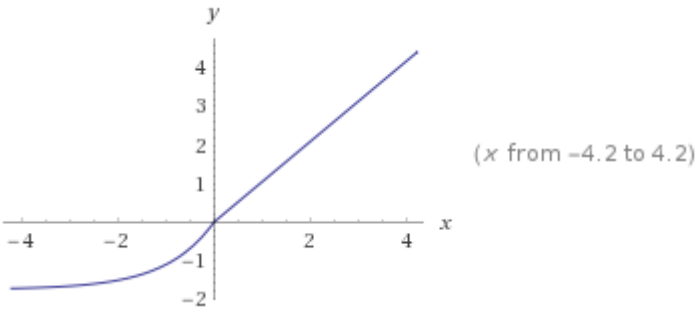

Figure 4.3 - SELU Function

In practice, this function is identical to RELU for inputs ≥ 0, but takes a different approach to negative values, it has a variable gradient which is at times, less than 1 at others greater than one, which we can use to manipulate the gradient, and consequently the variance.

By utilizing self-normalization over controlling the gradient, SELU neuron's will never die, and hence the rate of learning will stagnate. In practice, SELU has been known to greatly outperform RELU, so it is a reasonable idea to switch to using this to improve our networks.

Another way was through the use of background Augmentation, explained above in Step 1's area. One of our main issues during training was a severe lack of data points to train with. Deep learning requires a vast amount of data to learn from in order to generalize that knowledge to new, unseen points. This means that we would benefit from a higher number of images to attempt grasp detection. We did this by removing and changing the backgrounds behind each object.

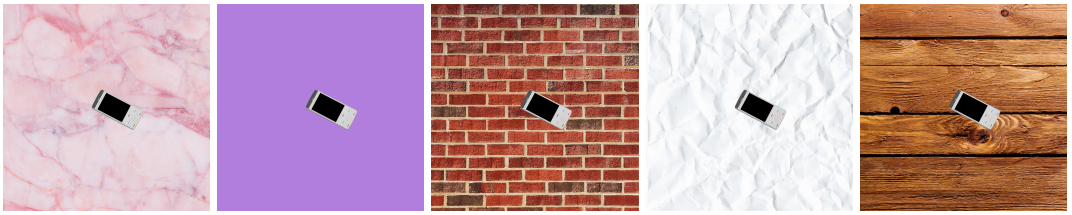

Figure 4.4 - Multiple Backgrounds

With the total number of datapoints now 6-fold from what we were given to start with, training the model became a lot easier. Obviously an approach like this needs to be careful in order to prevent overfitting, so we kept the number of backgrounds to a relatively small number.

Thirdly for the ResNet model, we pre trained the feature extraction block on ImageNet prior to training using the given images. This helps the model in the feature extraction of rgb images prior to being challenged with grasp detection. Wd did this as follows:

```python
def pretrain(model, batch_size, n_epochs, learning_rate, resize=None, mode = 'pretrain'): #mode = 'pretrain' or 'tra[40]
    print(f"batch size: {batch_size}\nn epochs: {n_epochs}\nlearning rate: {learning_rate}\n", "="*20)

    trainLoader = get_train_loader(batch_size, resize, 'grasp' if mode == 'train' else 'imageNet')
    lossFn = getLoss(mode == 'train')
    optimizer = getOptimiser(model, learning_rate)

    train_history = []

    model = model.to(device) # Move model to gpu
    model.train() # Set the model to training mode (for DropOut)
    model.featureExtractionBlock.load_state_dict(torch.load(featureExtractionPretrainModel))

    for epoch in range(n_epochs):  # loop over the dataset for n_epoch
        epoch_loss = 0
        with tqdm(trainLoader, unit="batch") as tepoch:
            tepoch.set_description(f"Epoch {epoch}")
            for rgb, grasp, label  in tepoch:   #for each batch
                rgb, grasp, label = rgb.to(device), grasp.to(device), label.to(device)   # Move tensors to gpu
                optimizer.zero_grad()   # zero the parameter gradients
                outputs = model(rgb)
                loss = lossFn(outputs,  grasp)
                loss.backward()
                optimizer.step()
                epoch_loss += loss.item()# add to running totals
        epoch_loss = epoch_loss / len(trainLoader)
        train_history.append(epoch_loss)
    torch.save(model.featureExtractionBlock.state_dict(), featureExtractionPretrainModel)
    return train_history
```

Figure 4.5 - Pre-train Function

This is the main function which governs the pre-training mechanism of the model, the key features being the get_train_loader function, which returns either our grasp detection model or ImageNet depending on the mode passed into the main function, we “pretrain” the model using ImageNet before we “train” the model in grasp detection. We also make sure to use torch.save() to keep the feature-extraction techniques learned from ImageNet safe for future use.

We also used hyper parameter tuning to find the optimum hyper-parameters for the deep ResNet model. We made a function to test using a simple list of learning rates, trying each learning rate.

```python
def learning_rate_search(lrs):
    results = {}
    for lr in lrs:
        train_loss, train_acc = train(nnet, BATCH_SIZE, EPOCHS, lr)
        results[lr] = (train_loss, train_acc)
    return results
lrResults = learning_rate_search([0.1, 0.01, 0.001, 0.0001, 0.00001])
```

Figure 4.6 - Learning Rate Search

Then after this we used the library optuna, which used bayesian tree algorithms to find the optimum hyper-parameters.

```
def optuna_objective(trial):
    EPOCHS = 8
    dropOut=trial.suggest_uniform("dropout", 0.1, 0.5)
    linearSize=trial.suggest_int("linear_layer_size", 256, 8192)
    pool = trial.suggest_int("pool_num", 1, 2)
    model = DeepGraspResNet(dropOut=dropOut, linearSize=linearSize, pool=pool)
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "RMSprop", "SGD"])
    lr = trial.suggest_loguniform("lr", 1e-4, 5e-2)
    batchSize=trial.suggest_int("batch_size", 8, 16)
    optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=lr)
    usePreTrain = trial.suggest_int("pre_trainer?", 1, 2)

    _, train_acc = train(nnet, batchSize, EPOCHS, lr, usePretrainedBlock = True if usePreTrain == 1 else False, optimizer=optimizer)
    return train_acc

def run_optuna():
    import optuna
    study = optuna.create_study(direction="maximize")
    study.optimize(optuna_objective, n_trials=100)

    pruned_trials = [t for t in study.trials if t.state == optuna.structs.TrialState.PRUNED]
    pruned_trials = [t for t in study.trials if t.state == optuna.structs.TrialState.PRUNED]
    complete_trials = [t for t in study.trials if t.state == optuna.structs.TrialState.COMPLETE]
    print("Study statistics: ")
    print("  Number of finished trials: ", len(study.trials))
    print("  Number of pruned trials: ", len(pruned_trials))
    print("  Number of complete trials: ", len(complete_trials))

    print("Best trial:")
    trial = study.best_trial

    print("  Value: ", trial.value)
    print("  Params: ")
    for key, value in trial.params.items():
        print("    {}: {}".format(key, value))
# run_optuna()
```

Figure 4.7 - Oputna Library

This improved our accuracy by about 4%.

References:

Sulabh Kumra, Christopher Kanan, 2017, <Accessed 30.03.2022>, url:https://ieeexplore.ieee.org/document/8202237/authors#authors

Hu Cheng, Yingying Wang, Max Q.-H. Meng, 2021, <Accessed 25.03.2022>, url:https://ieeexplore.ieee.org/document/9636511/authors#authors

Joseph Redmon, Anelia Angelova, Real-Time Grasp Detection Using Convolutional Neural Networks, url: https://arxiv.org/pdf/1412.3128.pdf