First Implementation: A Single-Node "Network":

- Optimizing one connection for one input :

$$O=\sigma(\phi w+b),$$

Example : $\sigma((-0.23)w+b)=0.74$, Given an input of −0.23, produce the output 0.74.
→ There are 2 unknown parameters w and b and just one equation, so we have infinity of solutions.

To solve this the idea is to find a solution by descending the error surface E defined by

$$E=12(O-0.74)_2,(29)$$

Libraries for Anaconda :

- TensorFlow: is an open source software library for high performance numerical computation. It comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

- the PyTorch deep learning library :
  It has nifty features such as dynamic computational graph construction as opposed to the static computational graphs present in TensorFlow and Keras.

The basics in Pytorch:

- Computational graphs : A computational graph is a set of calculations, which are called *nodes*, and these nodes are connected in a directional ordering of computation.
  The benefits of using a computational graph is that each node is like its own independently functioning piece of code, so it's useful for building neural network operations and making gradient back-propagation
- Tensors : Tensors are matrix-like data structures.

Some code to declare Tensors in Pycorch :

```
import torch
x = torch.Tensor(2, 3)    %tensor of zeros (2 lines,3 columns)
x = torch.rand(2, 3)      %tensor of random floats (2 lines,3 columns)
x = torch.ones(2,3)    %tensor of ones (2 lines,3 columns)
x[:,1]  %return the second column
```

→ Multiplying tensors, adding them and so forth is also available.

- **Autograd in PyTorch:**

The need of a mechanism where error gradients are calculated and back-propagated through the computational graph which is called Autograd in Pycorch.

The Autograd is done via the Variable Class. It contains a Tensor and allows the calculation of the gradients with the call of .backward() function.

- Declaration of Variable :

x = Variable(torch.ones(2, 2) * 2, requires_grad=True).
The requires_grad parameter means if it's true that the calculation of the gradient is allowed and this means that this Variable would be trainable.
If requires_grad = False, the Variable would not be trained.
We can declare Variable by operations on a tensor.
Example: z = 2 * (x * x) + 5 * x this is a Variable.

z.backward(torch.ones(2, 2)) → To calculate the gradient dz/dx.
The gradient is stored in the x Variable, in the property .grad.
When we do x.grad a tensor(2,2) is shown.

- **Creating a neural network in PyTorch :**

Object : create a neural network with 4-layer fully connected to classify the hand-written digits.

Neural Network Class :

Using the base Class : nn.Module and the need of including it's functionality.
Code :

```
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module):       % Inheritance from nn.Module Class
   def __init__(self):          %  initialization
```

```
super(Net, self).__init__()    %creation of an instance
self.fc1 = nn.Linear(28 * 28, 200)
self.fc2 = nn.Linear(200, 200)
self.fc3 = nn.Linear(200, 10)
```

→ A fully connected neural network layer is represented by the nn.Linear object, with the first argument in the definition being the number of nodes in layer l and the next argument being the number of nodes in layer l+1.

At this stage the skeleton of our neural network is ready. We need to define how data flows through out network.
→ Defining the forward() method which overwrites a method in the base class.

Code:

```
def forward(self, x):      %self= our neural network , x= the data
   x = F.relu(self.fc1(x))
   x = F.relu(self.fc2(x))
   x = self.fc3(x)
   return F.log_softmax(x)
```

Comments :
self.fc1(x) : we are feeding the first layer fully connected self.fc1 with the data x.
Then We apply a ReLU activation to the nodes in this layer using *F.relu()*.
We replace *x* at each stage, feeding it into the next layer.
Except for the last one – instead of a ReLU activation we return a log softmax "activation".

Definitions:

- The rectifier is an activation function defined as the positive part of its argument:
  f (x) = x + = max (0 , x).
  A unit employing the rectifier is also called a **rectified linear unit** (**ReLU**).

- The softmax function is a normalized exponential function.

Training Part :

net = Net() %Instanciation.

- first step: Setting an optimizer and a loss criterion:

```
# create a stochastic gradient descent optimizer
optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)
```

→ we need to supply to our optimizer is all the parameters of our network and this is available with pytorch by .parameters().

```
# create a loss function
criterion = nn.NLLLoss()
```

→ Criterions are helpful to train a neural network. Given an input and a target, they compute a gradient according to a given loss function(or error function, the difference between the actual output and the predicted output).

- the full training code:

```
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        # resize data from (batch_size, 1, 28, 28) to (batch_size, 28*28)
        data = data.view(-1, 28*28)
        optimizer.zero_grad()
        net_out = net(data)
        loss = criterion(net_out, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
```

Convolutional Neural Network : CNN

- CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing.[1] They are also known as shift invariant or space invariant

artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

- Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

Practical image segmentation with Unet :

Object: given an image of something and a mask, we want to extract the image from its background.
We will use a Unet neural network which will learn how to automatically create the masks:

1. By feeding into the neural net the images of the cars.
2. By using loss function comparing the output of the neural net with the appropriate masks and backpropagating the error for the network to learn.

Basics :

- Epoch: In training neural network, one epoch means one pass of the full training set. Usually it may contain a few iterations.
- Batch : Usually we divide the training set into batches, each epoch go through the whole training set. Each iteration goes through on batch.

Some code:

tb_viz_cb = TensorboardVisualizerCallback(os.path.join(script_dir, '../logs/tb_viz'))
→ TensorboardVisualizerCallback is the class which will save the predictions to tensorboard at each epochs of the training step.

tb_logs_cb = TensorboardLoggerCallback(os.path.join(script_dir, '../logs/tb_logs'))
→ TensorboardLoggerCallback will save the losses and pixel-wise "accuracies" to tensorboard.

model_saver_cb =
ModelSaverCallback(os.path.join(script_dir,'../output/models/model_' +
helpers.get_model_timestamp()), verbose=True)
→ ModelSaverCallback will save your model once the training step is finished.

Downloading and extracting the dataset from Kaggle or downloading the data manually by putting them in the input folder of the project :

```
From Kaggle :
ds_fetcher = DatasetFetcher()
ds_fetcher.download_dataset()
```

Splitting the training set into train/validation and retrieve the test set:

```
# Get the path to the files for the neural net
X_train, y_train, X_valid, y_valid =
ds_fetcher.get_train_files(sample_size=sample_size,
validation_size=validation_size)
full_x_test = ds_fetcher.get_test_files(sample_size)
```

Callback for the test (or predict) pass:

```
pred_saver_cb = PredictionsSaverCallback(os.path.join(script_dir,
'../output/submit.csv.gz'), origin_img_size, threshold)
```

→ It will store the predictions into a gzip file each time a new batch of prediction is made. This way the predictions are not stored into memory as they are very big. You can submit the resulting submit.csv.gz from the output folder to Kaggle when the predictions are finished.

Defining our neural net architecture :

```
net = unet_origin.UNetOriginal((3, *img_resize))
classifier = nn.classifier.CarvanaClassifier(net, epochs)
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.99)
```

→ Creating our network and the optimizer : net and optimizer.

```
train_ds = TrainImageDataset(X_train, y_train, input_img_resize, output_img_resize,
X_transform=aug.augment_img)
train_loader = DataLoader(train_ds, batch_size,
                          sampler=RandomSampler(train_ds),
                          num_workers=threads,
                          pin_memory=use_cuda)

valid_ds = TrainImageDataset(X_valid, y_valid, input_img_resize,
output_img_resize, threshold=threshold)
valid_loader = DataLoader(valid_ds, batch_size,
```

```
                sampler=SequentialSampler(valid_ds),
                num_workers=threads,
                pin_memory=use_cuda)

# Train the classifier
classifier.train(train_loader, valid_loader, epochs, callbacks=[tb_viz_cb, tb_logs_cb,
model_saver_cb])
```

→ The training step by passing in the train/valid dataset loaders and the training callback we defined.