towards
data science

Follow 608K Followers

Deep neural networks: How to define?

Correctly defining what makes a neural network deep is important. This post proposes an alternative definition to deep neural networks.

 Valdivino Santiago Júnior Sep 29 · 8 min read

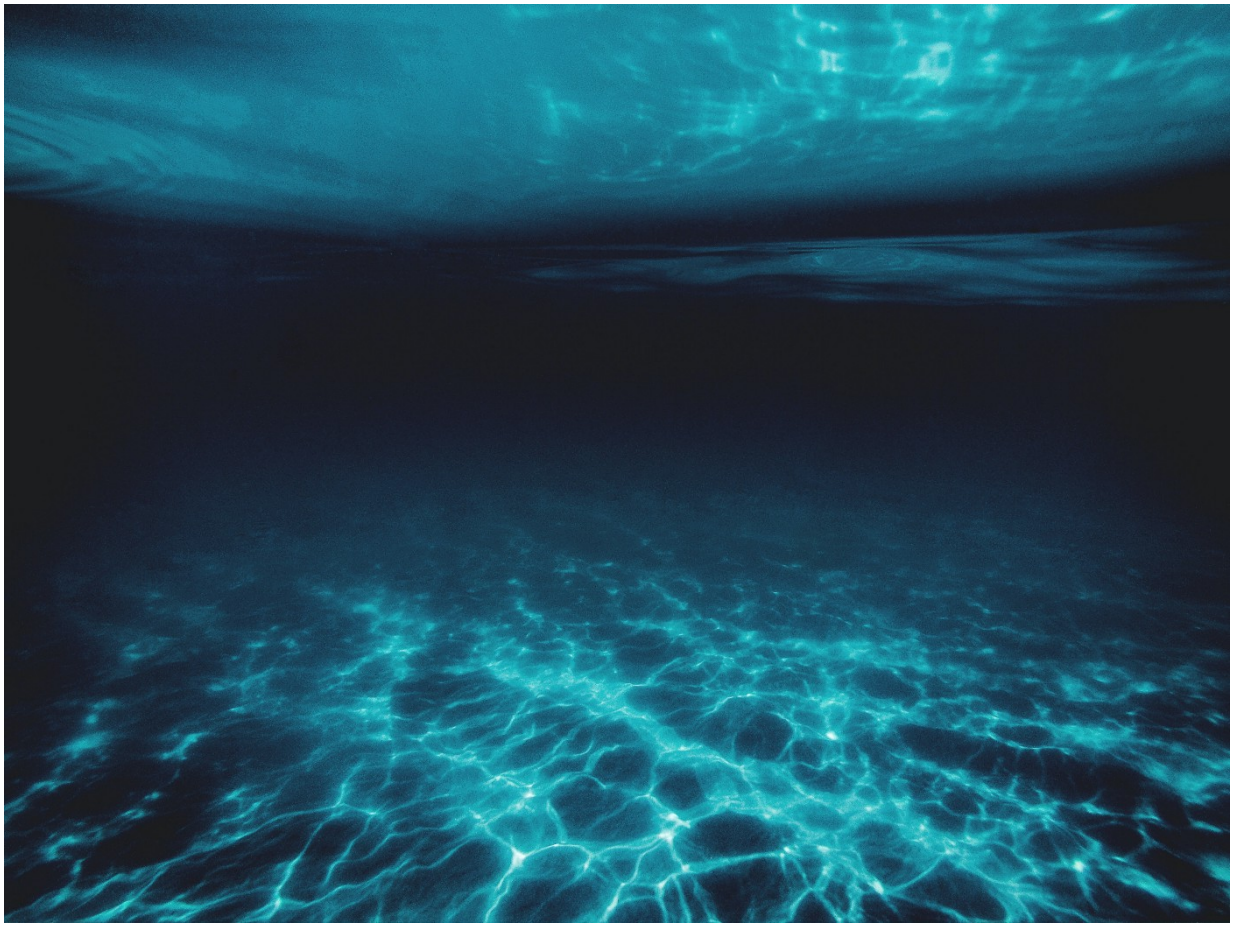


Photo by Jonathan Borba on Unsplash.

• • •

The success of artificial intelligence (AI) nowadays is basically due to deep learning (DL)

layers, where each type of layer has specific purposes.

However, deep neural networks (DNNs), such as deep convolutional neural networks (CNNs), are based on multilayer perceptron (MLP), a class of feed-forward artificial neural network that has been used for quite some time, even before the advent of the first CNN in 1989. Hence, it comes the question: when a model/network is considered "deep" and not "shallow"?

Shallow x deep neural networks

Traditionally, a shallow neural network (SNN) is one with one or two hidden layers. Thus, a deep neural network (DNN) is one with more than two hidden layers. This is the most accepted definition. Below, we show an example of an SNN and DNN (hidden layers are in red).

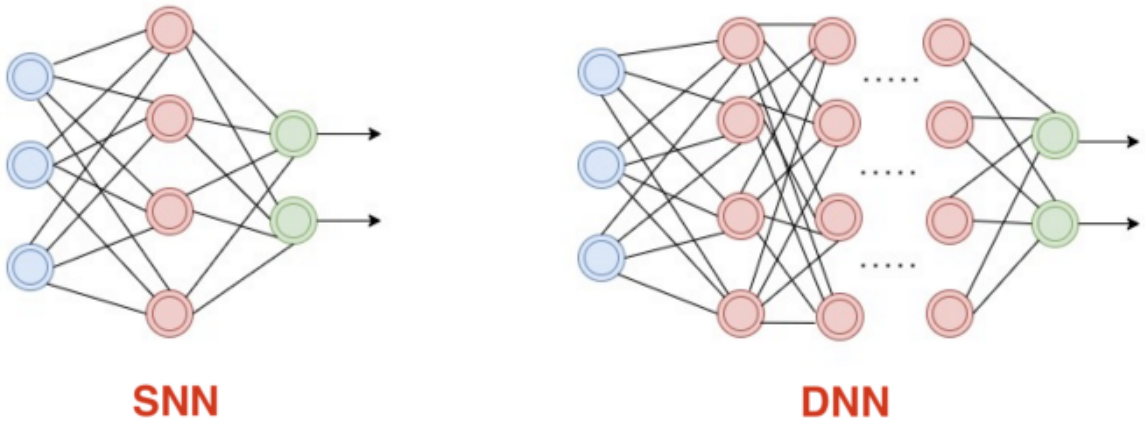


Image by author.

But, considering only the number of hidden layers is a good approach? Correctly defining what makes a model deep is important for an unambiguous understanding of the concept especially from the standpoint of DL adoption by the industry. In this post, we propose an alternative definition to DNNs. We suggest that a model to be considered "deep" should be analysed not only considering the number of hidden layers but also:

a.) considering the time necessary to train and/or evaluate it;

b.) taking into account the demand in terms of computational platform (e.g. GPU).

Classical benchmark datasets (e.g. CIFAR-10, ImageNet) and standard environments/platforms (Google Colab, kaggle) can be used to get these measures. In

MNIST database

Regarding the MNIST database, we considered three neural networks: SNN500 by Aviv Shamsian, CNN3L by Nutan, and LeNet-5 by Bolla Karthikeya. SNN500 is an SNN with a single hidden layer containing 500 neurons. CNN3L is also considered an SNN because it has two hidden (convolutional) layers and the output layer. The third network is the classical LeNet-5 with five layers where four are hidden ones. Let us see the code (access here).

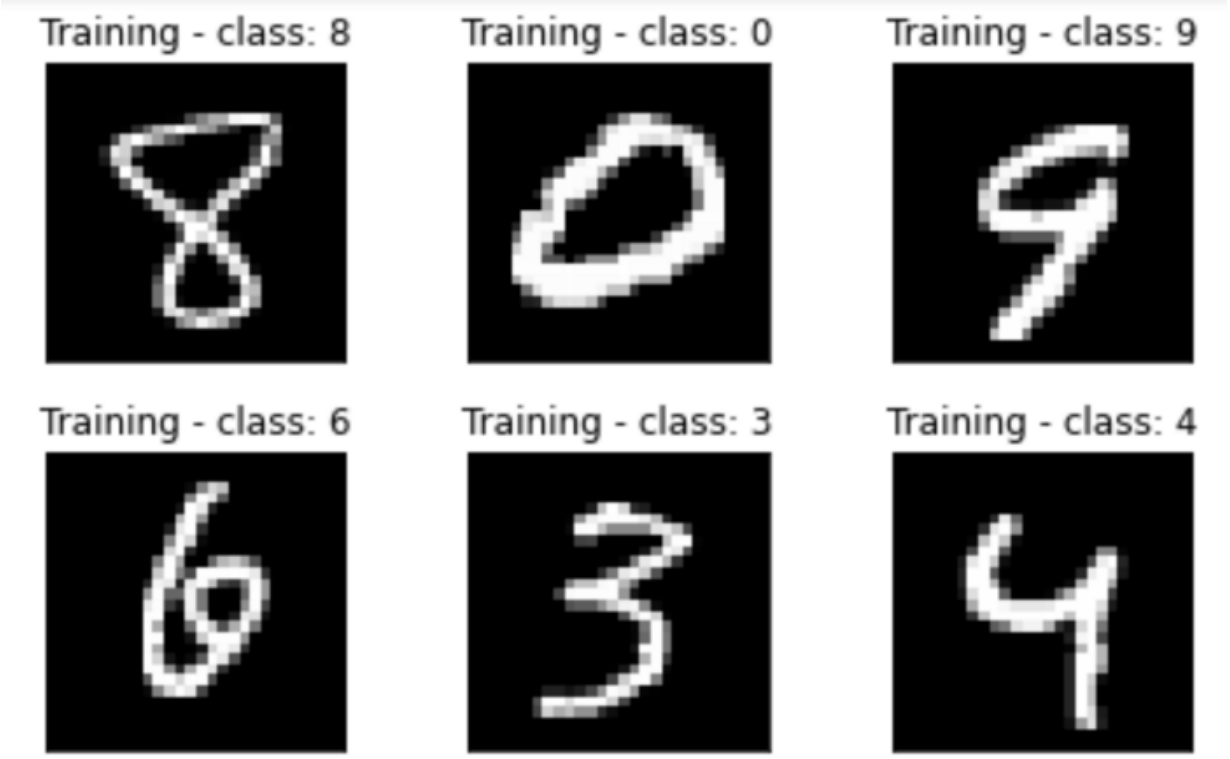


Image by author.

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
from prettytable import PrettyTable
import matplotlib.pyplot as plt
import time

# This function obtains the number of trainable parameters of the
# model/network.
def count_parameters(model):
    table = PrettyTable(["Modules", "Parameters"])
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad: continue
        param = parameter.numel()
```

```
    print(f"Total trainable params: {total_params}")
    return total_params

# Just visualising some images
def visualise_images(img, lab, t):
   fig = plt.figure()
   for i in range(6):
     plt.subplot(2,3,i+1)
     plt.tight_layout()
     plt.imshow(img[i][0], cmap='gray', interpolation='none')
     plt.title("{} - class: {}".format(t,lab[i]))
     plt.xticks([])
     plt.yticks([])
```

Now, we define the number of classes and also some hyper-parameters. Note that the batch size is 100.

```
num_classes = 10 # Number of output classes, discrete range [0,9]

# Hyper-parameters
num_epochs = 20 # Number of epochs
batch_size = 100 # The size of input data took for one iteration
lr = 1e-3 # Learning rate
```

We download and handle the MNIST dataset.

```
# Downloading MNIST dataset
train_data = dsets.MNIST(root = './data', train = True,
                         transform = transforms.ToTensor(), download =
True)

test_data = dsets.MNIST(root = './data', train = False,
                        transform = transforms.ToTensor())

print('#'*20)
print('Training dataset: ', train_data)
print('Test dataset: ', test_data)

# Wrap an iterable around the dataset to enable easy access to the
samples.
train_gen = torch.utils.data.DataLoader(dataset = train_data,
                                        batch_size = batch_size,
                                        shuffle = True)

test_gen = torch.utils.data.DataLoader(dataset = test_data,
                                       batch_size = batch_size,
                                       shuffle = False)


device = torch.device("cuda:0" if torch.cuda.is_available() else
```

Just taking a quick look at the training dataset.

```
batch_train = enumerate(train_gen)
batch_idx, (batch_train_data, batch_train_classes) =
next(batch_train)
print('One batch - training dataset:', batch_train_data.shape)

print('\nEach image of the batch:')
for i in range(batch_train_classes.shape[0]):
  print('Image: {} - Input shape: {} - Class: {}'.format(i,
batch_train_data[i].shape, batch_train_classes[i]))
  if i == (batch_train_classes.shape[0]-1):
    print('The "image" itself: ', batch_train_data[i])

visualise_images(batch_train_data, batch_train_classes, 'Training')
```

Note that the shape of one batch is: [100, 1, 28, 28]. This means 100 images in one batch, one channel (images in MNIST are in greyscale), and each image has a dimension of 28 x 28 pixels. Now, we define the three neural networks. Their performances are measured based on the accuracy of the test dataset.

```
class SNN500(nn.Module):
  def __init__(self, input_sz, hidden_sz, num_clas):
    super(SNN500,self).__init__()
    self.fc1 = nn.Linear(input_sz, hidden_sz)
    self.relu = nn.ReLU()
    self.fc2 = nn.Linear(hidden_sz, num_clas)

  def forward(self,x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    return out

class CNN3L(nn.Module):
    def __init__(self, num_clas):
        super(CNN3L, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
```

```
        )
        # Fully-connected layer
        self.out = nn.Linear(32 * 7 * 7, num_clas)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # Flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output

class LeNet5(nn.Module):
    def __init__(self, num_clas):
        super(LeNet5, self).__init__()
        # Convolution
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6,
kernel_size=5, stride=1, padding=2, bias=True)
        # Max-pooling
        self.max_pool_1 = nn.MaxPool2d(kernel_size=2)
        # Convolution
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16,
kernel_size=5, stride=1, padding=0, bias=True)
        # Max-pooling
        self.max_pool_2 = nn.MaxPool2d(kernel_size=2)
        # Fully-connected layers
        self.fc1 = nn.Linear(16*5*5, 120)   # convert matrix with
16*5*5 (= 400) features to a matrix of 120 features (columns)
        self.fc2 = nn.Linear(120, 84)       # convert matrix with 120
features to a matrix of 84 features (columns)
        self.fc3 = nn.Linear(84, num_clas)        # convert matrix
with 84 features to a matrix of 10 features (columns)

    def forward(self, x):
        # Convolve, then perform ReLU non-linearity
        x = nn.functional.relu(self.conv1(x))
        # Max-pooling with 2x2 grid
        x = self.max_pool_1(x)
        # Convolve, then perform ReLU non-linearity
        x = nn.functional.relu(self.conv2(x))
        # Max-pooling with 2x2 grid
        x = self.max_pool_2(x)
        # First flatten 'max_pool_2_out' to contain 16*5*5 columns
        x = x.view(-1, 16*5*5)
        # FC-1, then perform ReLU non-linearity
        x = nn.functional.relu(self.fc1(x))
        # FC-2, then perform ReLU non-linearity
        x = nn.functional.relu(self.fc2(x))
        # FC-3
        x = self.fc3(x)
        return x
```

Now, we can select one of the models/networks to train and evaluate. We also define the loss function, optimiser, and show the number of trainable parameters of the selected model.

```
print('Checking trainable parameters:
{}'.format(count_parameters(net)))
```

Now, we can train the model.

```
train_losses = []
train_acc = []
train_time_init = time.time()
for epoch in range(num_epochs):
  net.train()
  running_loss = 0.0
  running_corrects = 0
  for images,labels in train_gen: # Iterate over data: begin
    if opt == '1':
      images = Variable(images.view(-1,28*28)).to(device) # Send to
GPU
    elif (opt == '2') or (opt == '3'):
      images = Variable(images).to(device) # Send to GPU

    labels = Variable(labels).to(device) # Send to GPU
    optimizer.zero_grad()
    with torch.set_grad_enabled(True):
        outputs = net(images)
        _, preds = torch.max(outputs, 1)
        loss = loss_function(outputs, labels)
        loss.backward()
        optimizer.step()

    running_loss += loss.item() * images.size(0)
    running_corrects += torch.sum(preds == labels.data)
  # Iterate over data: end

  epoch_loss = running_loss / len(train_data)
  epoch_acc = running_corrects.double() / len(train_data)
  print('Epoch [%d/%d], Loss: %.4f, Accuracy: %.4f'
                %(epoch+1, num_epochs, epoch_loss, epoch_acc))

  train_losses.append(epoch_loss)
  train_acc.append(epoch_acc)

train_time_end = time.time() - train_time_init
```

In the inference phase, we measure the performance of the selected model.

```
am_training = net.training
print('Am I training? ', am_training)
net.eval()
am_training = net.training
print('Am I training? ', am_training)
inference_loss = 0.0
```

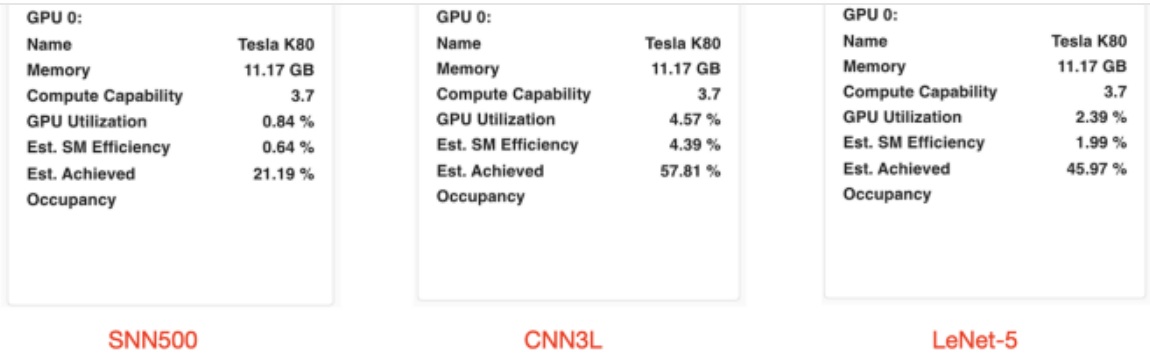
```
with torch.no_grad():
  for images,labels in test_gen: # Iterate over data: begin
      if opt == '1':
        images = Variable(images.view(-1,28*28)).to(device) # Send to
GPU
      elif opt == '2' or opt == '3':
        images = Variable(images).to(device) # Send to GPU

      labels = labels.to(device) # Send to GPU
      outputs_infer = net(images)
      _, preds_infer = torch.max(outputs_infer,1)
      loss_infer = loss_function(outputs_infer, labels)

      inference_loss += loss_infer.item() * images.size(0)
      inference_corrects += torch.sum(preds_infer == labels.data)
  # Iterate over data: end

final_inference_loss = inference_loss / len(test_data)
final_inference_acc = inference_corrects.double() / len(test_data)

infer_time_end = time.time() - infer_time_init
print('\nTraining and inference in {:.0f}m {:.0f}s  OR
{:.0f}s'.format(
        (train_time_end + infer_time_end) // 60,
        (train_time_end + infer_time_end) % 60,
         train_time_end + infer_time_end))

print('\nLoss of {}: {:.4f}'.format(opt_name, final_inference_loss))
print()
print('Accuracy of {}: {:.4f}'.format(opt_name, final_inference_acc))
```

Hence, we ran each of the three models for 20 epochs, and measured the required total time (train_time_end + infer_time_end. Note that the total time includes only these two phases) in seconds, as shown below. Time and accuracy are the average of the three runs.

	#Layers	#Train Par	Time (s)	Accuracy	Type
SNN500	2	397,510	172.00	0.9832	Shallow
CNN3L	3	28,938	228.33	0.9908	Shallow
LeNet-5	5	61,706	199.33	0.9891	Deep

MNIST: results. Image by author.

Note that even if CNN3L has the smallest number of trainable parameters (#Train Par), it demanded more time to run (train and evaluate). CNN3L is also the best of all models (even if accuracies are very close). Below, we show the GPU summary obtained via the TensorBoard Plugin with PyTorch Profiler (access here).



MNIST: GPU summary. Image by author.

In all models, the GPU utilisation is very low. However, note that CNN3L has the highest GPU utilisation (4.57%). The bottom of line is that, according to the classical definition, an SNN (CNN3L) demanded more time to run, got higher accuracy, and utilised more GPU than a DNN (LeNet-5).

Important to mention that as higher the GPU utilisation, the better. But since here we are comparing models with the same configurations (same batch size, same number of workers on DataLoader's construction, …), a higher GPU utilisation may be interpreted as a model demanding more computational resources than other. We would expect that deeper models would demand higher GPU utilisation than the shallower ones.

CIFAR-10 database

The CIFAR-10 database consists of 32 x 32 colour images (three bands). Here, we used two out the three previous models: CNN3L by Nutan and LeNet-5 presented in the Training a Classifier PyTorch tutorial. We have made some minor modifications in these two models in order to properly handle the CIFAR-10 images. The optimiser, learning rates are identical, but now the batch size is 4 and we ran each model three times for 10 epochs. Access the code here.

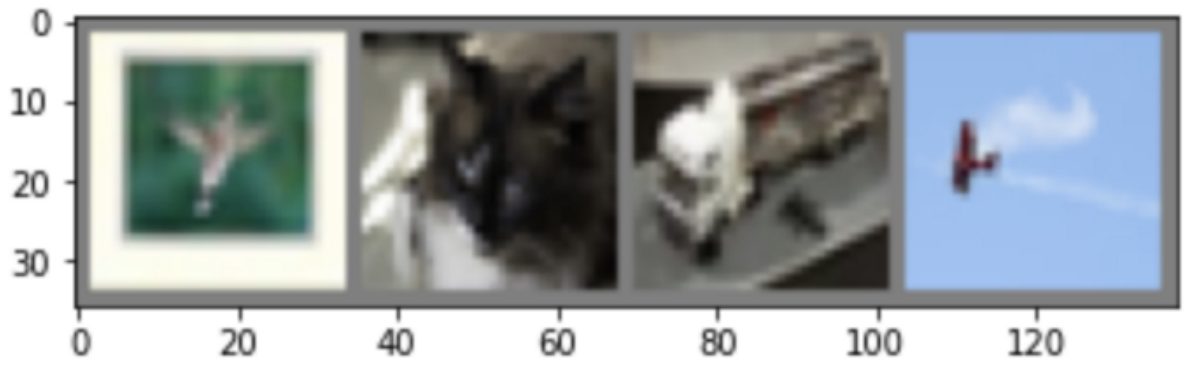


Image by author.

	#Layers	#Train Par	Time (s)	Accuracy	Type
CNN3L	3	22,058	770.33	0.6345	Shallow
LeNet-5	5	62,006	880.33	0.6146	Deep

CIFAR-10: results. Image by author.

Now, we have a more “natural” situation. In other words, LeNet-5 has higher number of trainable parameters and demanded more time to run. But, again, CNN3L achieved better perfomance. The GPU summary obtained via the TensorBoard Plugin with PyTorch Profiler (access here) is shown below.

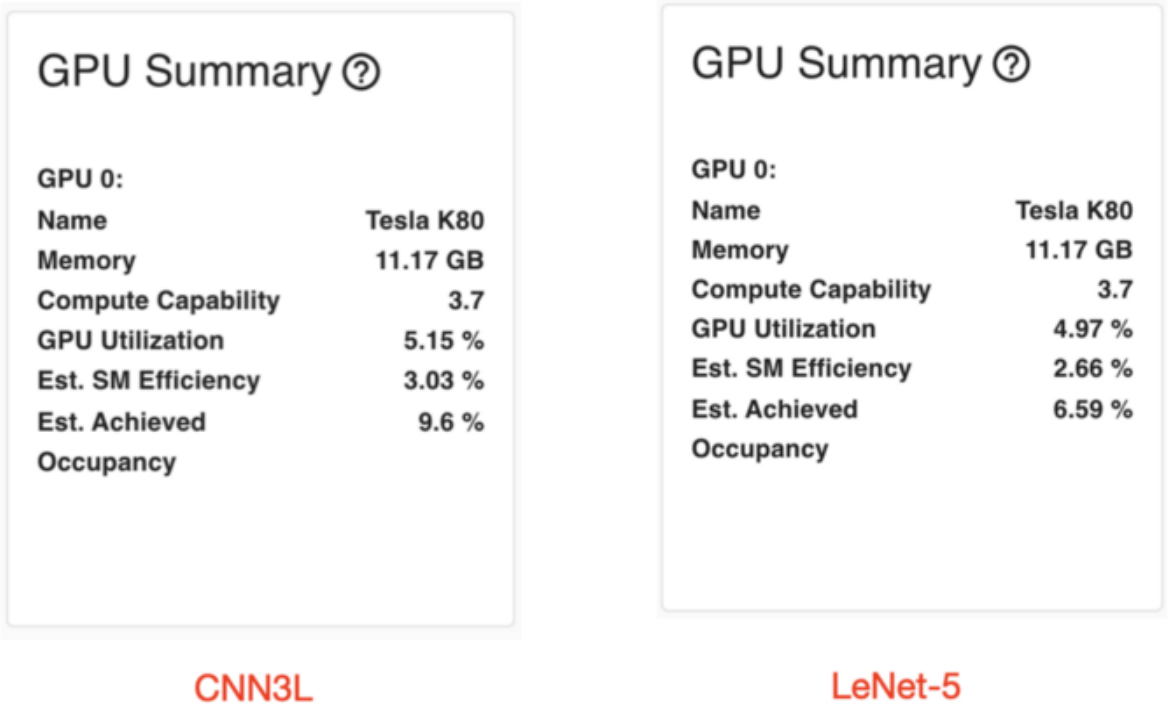


Image by author.

Again, CNN3L (shallow) presented more GPU utilisation than LeNet-5 (deep).

Conclusions

In this post, we suggest an alternative definition to consider a model as a DNN. We believe that a definition that encompass not only the number of hidden layers but also the the time necessary to train and/or evaluate the model, and also the requirements in terms of computational platform might be more suitable.

and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

Deep Learning Deep Neural Networks