IPS ESTB

Convolution Neural Networks and its applications

Author: Bernardo Antunes Gomes Augusto

Student number: 201800128

Curricular Unit: Machine Learning

Teachers: Jacinto Estima and Vala Rohani

Index

Introduction

Machine Learning is the most widely used branch of computer science nowadays (Behl, 2020). It is used by many industries for automating tasks and doing complex data analysis. Nowadays, we are using devices benefit from them. For Example, an intelligent assistant like Google Home, wearable fitness trackers like Fitbit (Behl, 2020). In the last few years, the field of machine learning has started to demonstrate his full potential, with the rise of the Artificial Neural Network (ANN). These computational models can outperform, by far, previous forms of artificial intelligence in common machine learning tasks. One of the most outstanding forms of ANN architecture is the Convolutional Neural Network (CNN). CNN's are mainly used to solve difficult image-driven pattern recognition tasks and with their precise, yet simple architecture, they offer a simple approach of getting started with ANNs. This research work starts with a scanty introduction to CNNs, a section called Literature Review where will be explored the existing related works to CNN's, small project with Pytorch about CNN in the section Implementation, after this section, we will move forward to the part where will be a discussion on the findings and finally, but not least, a short conclusion to finish this research work. For this section of the implementation, I will be using a known dataset called FashionMNIST. The MNIST dataset, *Modified National Institute of Standards and Technology database*, is a famous dataset of handwritten digits that is commonly used for training image processing systems for machine learning. NIST stands for *National Institute of Standards and Technology*. The *M* in MNIST stands for *modified*, and this is because there was an original *NIST* dataset of digits that was modified to give us *MNIST* (deeplizard.com). In this implementation I will present images of the code and the output of each piece of the code. Along with a brief explanation of what that part of the code does. For the realization of this implementation I followed an online course available in deeplizard.com (You can access the link by just pressing the CTRL button and clicking the mouse in this **link**).

Literature Review

For the development of this section of the research work I've consulted a book called "*Recent Trends and Advances in Artificial Intelligence and Internet of Things*" chapter 36 (Fundamental Concepts of Neural Network). In this section I will exploit some of the existing related applications of CNN's and give a more meticulous explanation to what is CNN. Like

said in the introduction CNN's are a form of ANN architecture. CNN's are mainly used to solve difficult image-driven pattern recognition tasks and with their precise, even with their simple architecture, they are a simple approach to someone getting started with ANNs. "*It can learn highly abstracted features of objects especially spatial data and can identify them more efficiently. A deep CNN model consists of a finite set of processing layers that can learn various features of input data (e.g. image) with multiple levels of abstraction. The initiatory layers learn and extract the high-level features (with lower abstraction), and the deeper layers learn and extract the low-level features (with higher abstraction)*" (Valentina E. Balas, 2019). One question that always comes up is "Why Convolutional Neural Networks is more considerable over other classical neural networks in the context of computer vision?". One of the main reasons to consider CNN is the weight sharing feature of CNN, that reduces the number of trainable parameters in the network, which prevents overfitting and improve generalization. The implementation of a large network is harder since it uses other types of neural networks rather than using CNN. At present CNN has come to light as a tool to achieve a promising result in a variety of computer vision-based applications like image classification, object detection, speech recognition, facial expression recognition, text recognition, etc. Some application areas that apply CNN to achieve state-of-the-art performance including image classification, text recognition, object detection, human pose estimation, image captioning, etc. I will give a brief explanation of how CNN can be used in these areas. Starting with image classification. CNN is the first choice for image classification problem since CNN has been achieving hugely better classification accuracy compared to other methods especially in the case of large-scale datasets. Next comes the text recognition. The text detection and text recognition inside of an image have been widely studied for a few years. CNN plays a vital role to identify the text inside the image. In the medical image analysis, CNN has rapidly proved to be a state-of-the-art foundation, by achieving outstanding performances in the diagnosis of diseases by processing medical images e.g. X-rays. About the automatic colorization of image and style transfer. In the last years, with the deep learning ascension, some CNN models gave an automation way to convert black and white images or gray images to equivalent colorful RGB images. And the satellite imagery. Nowadays, CNN has a vital role to identify different natural hazards like e.g. tsunamis and hurricanes. By the usage of satellite image analysis, we can do smart city plan, roadway and river extraction, land classification, and many others. There are many more applications of CNN and I'm sure that the number will continue to rise.

Implementation

We've arrived at the implementation part. In this section I will show the project I've developed. I have chosen to go with PyTorch has a lot of code available on the internet for some error I may encounter along the way, sometimes is faster than TensorFlow, easy to debug, easy to use and is developer-friendly. This implementation could have been done with other tools like, Tensorflow or ScikitLearn. I just have chosen PyTorch for reasons I've already mentioned and because I wanted to learn a new tool. Before we jump right to the presentation of my project, I will give a quick background of PyTorch. Pytorch is a relatively new framework. The initial release of PyTorch was in October 2016, and before PyTorch was created, there was and there still exists another framework called Torch. Torch is a machine learning framework that is based on a programming language called Lua. The connection between PyTorch and Torch exists because many of the developers who maintain the Lua version are the ones who created PyTorch (lizard, 2018). The Lua version was aging and so it was necessary to create a new framework. The creator of this new framework, called PyTorch, was Soumith Chintala. It's also important to salient that PyTorch is written in Python. In figure 1 we can see the main packages that I'm going to use and a brief explanation of each one.

Now that we have a basic understanding of PyTorch let's move for the real implementation.

Package	Description
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

Figure 1 – Main Packages PyTorch (original table in deeplizard.com)

In this import section I will import all the modules and packages needed to the ETL process, build, train, and analyze our CNN. The difference between a module and a package is that a module is a single file (or files) that are imported under one import and used. A package is a collection of modules in directories that give a package hierarchy. In figure 2 we can observe the necessary imports for our CNN. There is not much to say about this part of the implementation. I divided the imports into 3 rectangles, a green, a blue, and a red. The green one (that is the first one counting from the start) contains all the packages and modules that we are going to need for the ETL, build, train process. An important point to make is that the torch.set_grad_enabled(True) is true by default but I've followed the course and at some point at the course had to disable. The torch.set_printoptions(linewidth=120) sets display options for output. The blue rectangle (the one in the middle) has modules and packages that we need to build the confusion matrix and plotting it. Finally, the red rectangle is optional. This will only be used because with the import of plot_confusion_matrix I was running into some error, so I decided to insert the entire function. And it solves the error.

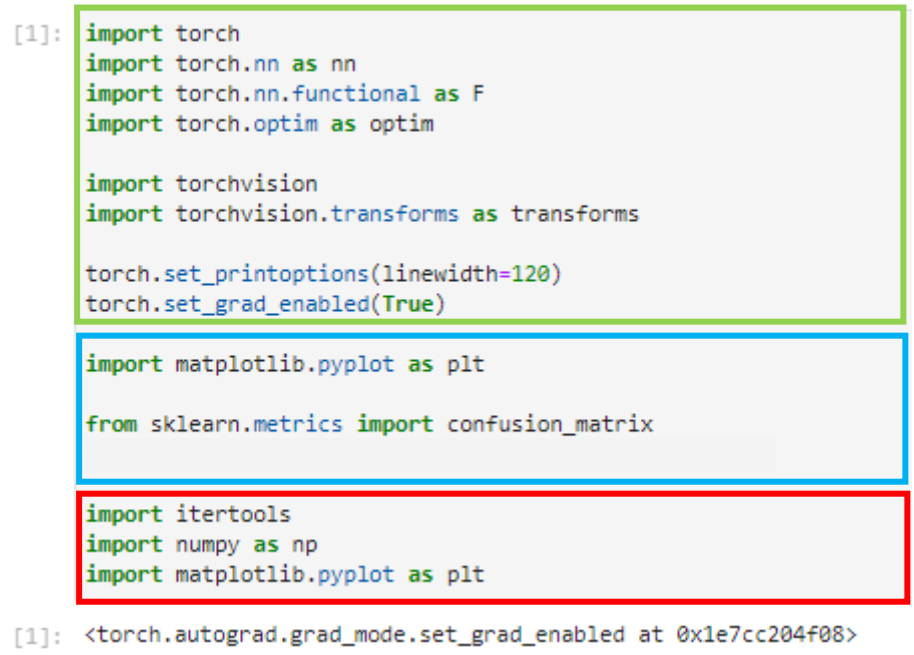

Figure 2 - imports

The next section was the building process. This process is where we are going to build our CNN. But before that in figure 3 we can see the version of torch and torchvision that I was using in this research work.

```
[3]: # Versions that I am working with
     print(torch.__version__)
     print(torchvision.__version__)

     1.5.0
     0.6.0
```

Figure 3 - versions

Now we can observe the figure 4. In this figure we can see a function (get_num_correct) that will be used to tells us the number of correct predictions giving a predictions tensor and a labels tensor.

```
[4]: def get_num_correct(pred, labels):
         return preds.argmax(dim=1).eq(labels).sum().item()
```

Figure 4 - get_num_correct

In figure 5, we can observe our CNN. Our Network class is an extension of Module class from PyTorch. It will inherit the constructor method from the Module class. Then this class will have two convolutional layers and three linear layers, in the red (first) and green (second) rectangle respectively. We can see that the convolutional layers have three parameters while the linear layers have two parameters. The in_channel will receive the values from out_channel from the previous layer, so as the in_features will receive the out_features value. In figure 6 is a table from deeplizard that has all a brief explanation of what each parameter corresponds to. In the class Network we have a method called forward, has shown in figure 5. What this method does is accepts a tensor as input, and then, it returns a tensor as output. In the method forward is where the convolutional and linear layers are going to be used.
The input tensor is going to be transformed and passed through the convolutional layers. The Conv1 and Conv2 have a convolutional operation and then have a relu activation operation, where its output is going to be passed to the max polling operation with kernel_size=2 and stride=2. The second convolutional layer will receive the output from the first convolutional layer. Each layer contains a collection of weight, that is the data, and a collection of operations, the code. The weights are stored inside the nn.Conv2d() class instance. We must reshape or flatten our tensor before we pass our input to the first hidden layer. This is a must when passing a linear layer receives an input from a convolutional layer output. About the reshape operation in figure 5, the number 12 corresponds to the number of output channels

coming from the previous convolutional layer. And the 4*4 is the height and width of each output channel.

```
[5]: class Network(nn.Module):
         def __init__(self):
             super().__init__()

             # this function have 2 convolutional layers
             self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
             self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)

             # and 3 Linear layers
             self.fc1 = nn.Linear(in_features=12 * 4 * 4, out_features=120)
             self.fc2 = nn.Linear(in_features=120, out_features=60)
             self.out = nn.Linear(in_features=60, out_features=10)

         def forward(self, t):

             # in the foward method is where the linear
             # and the convolutional layers are gonna be used

             t = F.relu(self.conv1(t))
             t = F.max_pool2d(t, kernel_size=2, stride=2)

             t = F.relu(self.conv2(t))
             t= F.max_pool2d(t, kernel_size=2, stride=2)

             t= t.reshape(-1, 12 * 4 * 4)
             t = F.relu(self.fc1(t))

             t = F.relu(self.fc2(t))

             t = self.out(t)

             return t
```

Figure 5 - Class Network

Layer	Param name	Param value	The param value is
conv1	in_channels	1	the number of color channels in the input image.
conv1	kernel_size	5	a hyperparameter.
conv1	out_channels	6	a hyperparameter.
conv2	in_channels	6	the number of out_channels in previous layer.
conv2	kernel_size	5	a hyperparameter.
conv2	out_channels	12	a hyperparameter (higher than previous conv layer).
fc1	in_features	12*4*4	the length of the flattened output from previous layer.
fc1	out_features	120	a hyperparameter.
fc2	in_features	120	the number of out_features of previous layer.
fc2	out_features	60	a hyperparameter (lower than previous linear layer).
out	in_features	60	the number of out_channels in previous layer.
out	out_features	10	the number of prediction classes.

Figure 6 - table of the deeplizard of what each parameter corresponds to

In figure 7, we can see how to access our dataset. It will extract from where is stored in our machine or download it directly from the source.

```
[6]: # here we are going to initialize our training set and we
     # are using torchvision to acess the FashionMNIST dataset

     train_set = torchvision.datasets.FashionMNIST(root='./data'
                                                   , train=True
                                                   , download=True
                                                   , transform=transforms.Compose([
                                                       transforms.ToTensor()
                                                   ])
                                                   )
```

Figure 7 - access to the dataset

In figure 8 we can observe the complete training loop. The main goal of this process is that the number of correct values goes up and the value of the loss goes down. One thing to notice is that we get distinct results every time we run this code. This happens because the model is created each time at the top, and the model weights are randomly initialized. Its recommended that we pull the network, optimizer, and the train_loader out of the training loop cell so we are able to run the training loop without resetting the network's weights. Inserting a bigger number of cycles doesn't justify because besides the fact that, we can suffer from overfitting, we also will not get much benefit from that (in terms of time and terms of increase of the accuracy). That's the reason I decided to go with just 5 cycles for the training loop. In figure 9, we can see that we have an accuracy of around 88%.
After the training process, we move to the analytic part of this implementation. For this I will build a confusion matrix and plotting it. Before we do this, we must check the length of our training set and the targets in our training set. As we can see in figure 10, the length of both is the same. This is because we have 6000 images in the dataset. To get the predictions of the entire dataset we need a function, and its name is get_all_preds(). We can see this function in figure 11. In this function we are going to use a data loader to generate batches. Because we can't load all our dataset since the machine would not be able to handle the process of all the data in the dataset. I have decorated this function with the function torch.no_grad(). Because we want this function execution to omit gradient tracking. With this we use less memory. To collect our results will be used the torch.cat() function.

```
[7]: network = Network()
optimizer = optim.Adam(network.parameters(), lr=0.01)
train_loader = torch.utils.data.DataLoader(
    train_set
    ,batch_size=100
    ,shuffle=True
)
```

```
[8]: for epoch in range(5):

    total_loss = 0
    total_correct = 0

    for batch in train_loader: # Get Batch
        images, labels = batch

        preds = network(images) # Pass Batch
        loss = F.cross_entropy(preds, labels) # Calculate Loss

        optimizer.zero_grad()
        loss.backward() # Calculate Gradients
        optimizer.step() # Update Weights

        total_loss += loss.item()
        total_correct += get_num_correct(preds, labels)

    print(
        "epoch", epoch,
        "total_correct:", total_correct,
        "loss:", total_loss
    )
```

```
epoch 0 total_correct: 46142 loss: 358.7937639057636
epoch 1 total_correct: 51164 loss: 238.6852857619524
epoch 2 total_correct: 51819 loss: 219.81513564288616
epoch 3 total_correct: 52278 loss: 210.05714961886406
epoch 4 total_correct: 52526 loss: 203.2397604137659
```

Figure 8 - Complete trainning loop

```
[9]: # accuracy
     total_correct / len(train_set)
```

```
[9]: 0.8754333333333333
```

Figure 9 - Accuracy

```
[9]: # 60000 because we have 60000 images in the dataset
     len(train_set)

[9]: 60000

[10]: # 60000 because we have 60000 images in the dataset
      len(train_set.targets)

[10]: 60000
```

Figure 10 - check the lenght of the dataset

```
[11]: # we are gonna use a data loader to generate batches
      # Because we cant load all our dataset because the machine would not handle the process of all the data
      @torch.no_grad()
      def get_all_preds(model, loader):
          all_preds = torch.tensor([])
          for batch in loader:
              images, labels = batch

              preds = model(images)
              all_preds = torch.cat(
                  (all_preds, preds)
                  ,dim=0
              )
          return all_preds
```

Figure 11 – get_all_preds() function

I also used “with torch.no_grad()”, in figure 12 because, as I mentioned before, with this we use less memory. In figure 13 we can see the total correct and accuracy. I have to adjust the original code since I was running into an error. I had to insert the .view(60000, 1) and it solves the error. But at one cost, the total correct and the accuracy became perfect and they shouldn’t be.

```
[13]: with torch.no_grad():
          prediction_loader = torch.utils.data.DataLoader(train_set, batch_size=10000)
          train_preds = get_all_preds(network, prediction_loader)
```

Figure 12 - with torch.no_grad()

```
[14]: # Had to put view(60000, 1) because it was giving me an error saying:
      # RuntimeError: The size of tensor a (100) must match the size of tensor b (60000) at non-singleton dimension 0
      # PS: view(60000, -1) also work

      preds_correct = get_num_correct(train_preds, train_set.targets.view(60000, 1))

      print('total correct: ', preds_correct)
      print('accuracy: ', preds_correct / len(train_set))

      total correct:  600000
      accuracy:  10.0
```

Figure 13 - total correct and accuracy

Now I have reached the final part of the implementation, the analytic part. In this part I will be showing how I've built the confusion matrix and how I plot it. We can observe the figure 14, where I have **to** build the confusion matrix using the function "confusion_matrix()" and the number of predicted labels vs the target labels, the values inside the two tensors act as coordinates for our matrix. We can also see that the type of this matrix is **the** data type of this matrix is int64. This is the default data type when we don't specify.

```
[16]: cm = confusion_matrix(train_set.targets, train_preds.argmax(dim=1))

[17]: print(type(cm))
      <class 'numpy.ndarray'>

[18]: cm

[18]: array([[4496,   14,   45,  319,   23,    3, 1051,    0,   49,    0],
             [   4, 5869,    2,  107,    9,    0,    5,    0,    4,    0],
             [  28,    5, 4241,   86, 1178,    0,  412,    0,   50,    0],
             [  49,   25,   15, 5707,   81,    4,  107,    0,   10,    2],
             [   4,    3,  182,  379, 5247,    2,  162,    0,   21,    0],
             [   0,    0,    0,    0,    0, 5916,    0,   39,    2,   43],
             [ 459,    5,  351,  226,  791,    4, 4078,    0,   86,    0],
             [   0,    0,    0,    0,    0,  349,    0, 5575,    0,   76],
             [   5,    0,    6,   19,   28,   36,   28,   22, 5855,    1],
             [   0,    0,    0,    1,    0,   46,    0,  241,    1, 5711]],
            dtype=int64)
```

Figure 14 - build confusion matrix

In figure 15, we have the function plot_confusion_matrix(). This function could be imported but I decided to do the import since I was running into some errors. I'm not going to go deep in this figure since it could have been done by one import. In figure 15 we can also observe the labels that I gave to each line and column of the matrix.

[19]:
```
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[20]:
```
names = (
    'T-shirt/top'
    ,'Trouser'
    ,'Pullover'
    ,'Dress'
    ,'Coat'
    ,'Sandal'
    ,'Shirt'
    ,'Sneaker'
    ,'Bag'
    ,'Ankle boot'
)
```

Figure 15 - plot_confusion_matrix() and names

Lastly, in figure 16 we can see that I defined that the figure size with line 21 (10 by 10). And I plot the confusion matrix. It's important to notice that with the confusion matrix we can conclude where our CNN is getting more confused. By observing this confusion matrix, we can say that there are there classes that it gets confused most of the time. These are Shirt with a T-shirt/top, Coat with Shirts and Coat with Pullover. It’s important to salient that I have read the confusion matrix through predicted labels and then true labels, like x and y. All this code is available in my GitHub (https://github.com/bernar0507/Machine-Learning/tree/master/Research%20Work%20ML).

```
[21]: plt.figure(figsize=(10,10))

[21]: <Figure size 720x720 with 0 Axes>
      <Figure size 720x720 with 0 Axes>

[22]: plot_confusion_matrix(cm, names)
```

```
Confusion matrix, without normalization
[[4496   14   45  319   23    3 1051    0   49    0]
 [   4 5869    2  107    9    0    5    0    4    0]
 [  28    5 4241   86 1178    0  412    0   50    0]
 [  49   25   15 5707   81    4  107    0   10    2]
 [   4    3  182  379 5247    2  162    0   21    0]
 [   0    0    0    0    0 5916    0   39    2   43]
 [ 459    5  351  226  791    4 4078    0   86    0]
 [   0    0    0    0    0  349    0 5575    0   76]
 [   5    0    6   19   28   36   28   22 5855    1]
 [   0    0    0    1    0   46    0  241    1 5711]]
```

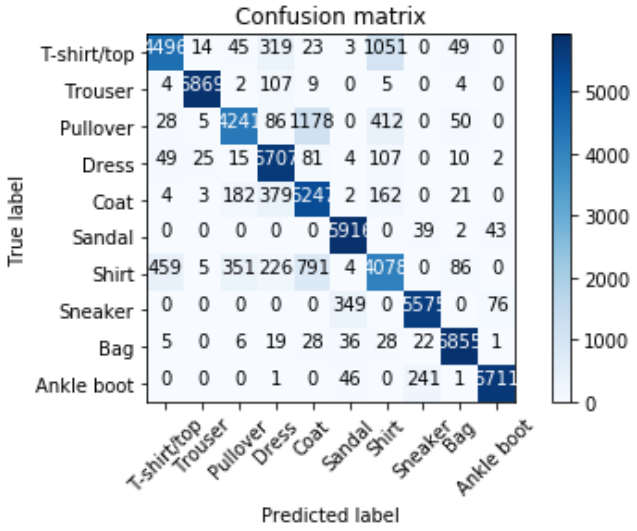

Figure 16 - plotted confusion matrix

Conclusion

CNN is the first choice for image classification problems since CNN has been achieving hugely better classification accuracy compared to other methods especially in the case of large-scale datasets. Next comes the text recognition. The text detection and text recognition inside of an image have been widely studied for a few years. CNN plays a vital role to identify the text inside the image. In the medical image analysis, CNN has rapidly proved to

be a state-of-the-art foundation, by achieving outstanding performances in the diagnosis of diseases by processing medical images e.g. X-rays. About the automatic colorization of image and style transfer.

Nowadays, CNN has a vital role to identify different natural hazards like e.g. tsunamis and hurricanes. By the usage of satellite image analysis, we can do smart city plan, roadway and river extraction, land classification, and many others. There are many more applications of CNN and I'm sure that the number will continue to rise.

In this implementation I could see with my own eyes the true power of CNN. I was able to discover another incredibly useful tool, PyTorch. I could deeply understand how to build, train, and analyze CNN. Although CNN was unable to distinguish some images, I can say with confidence that this was an impressive result. It's no wonder that CNN is the first choice for image classification problem since CNN. since CNN has been achieving hugely better classification accuracy compared to other methods especially in the case of large-scale datasets. If CNN continues to grow as they have been growing in these last few years, I'm sure they will help a lot not just in the medical department but also in the anti-criminal service. For example, avoiding and identifying fake signatures and other types of fraud.

I am no expert in this area but in this challenge of the research work, I could discover a lot of new things and could consolidate some concepts already addressed in class by my teachers.

I hope that this research work not only provides a better understanding of CNNs but also facilitates future research activities and application developments in the field of CNNs.

Acknowledgment

This research was developed by me, Bernardo Antunes Gomes Augusto, and with the supervision of the professors, Jacinto Estima and Vala Rohani and carried out at Polytechnic Institute of Setubal, Portugal.

References

Behl, A. (2020, April 22). *An Introduction to Machine Learning*. Retrieved from Medium:

https://becominghuman.ai/an-introduction-to-machine-learning-33a1b5d3a560

BILLINGS, G. L. ((Received 30 May 1995; accepted 26 October 1995, October 26). Radial Basis Fuction Network Configuration Mutual Information and the Orthogonal Least Squares Algorithm. p. 19.

deeplizard. (2018, September 03). *youtube*. Retrieved from Neural Network Programming - Deep Learning with PyTorch: https://www.youtube.com/watch?v=v5cngxo4mIg&list=PLZbbT5o_s2xrfNyHZsM6ufI0iZENK9xgG

Dubovikov, K. (2017, June 20). *Medium*. Retrieved from PyTorch vs TensorFlow — spotting the difference: https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b

Engineer, P. (2019, December 13). *youtube*. Retrieved from PyTorch Tutorials - Complete Beginner Course: https://www.youtube.com/watch?v=EMXfZB8FVUA&list=PLqnslRFeH2UrcDBWF5mfPGpqQDSta6VK4

Jiuxiang Gu, Z. W. (2017, October 9). Recent Advances in Convolutional Neural Networks. *Recent Advances in Convolutional Neural Networks*, p. 38.

lizard, d. (2018, September 05). *DEEPLIZARD*. Retrieved from Neural Network Programming - Deep Learning with PyTorch: https://deeplizard.com/learn/video/iTKbyFh-7GM

Nielsen, M. (2019, December). *Neural networks and deep learning*. Retrieved from Deep learning: http://neuralnetworksanddeeplearning.com/chap6.html#introducing_convolutional_networks

Valentina E. Balas, R. K. (2019). *Recent Trends and Advances in Artificial Intelligence and Internet of Things.* Springer.

Wikipedia. (2020, May 14). Retrieved from Extract, transform, load :

https://en.wikipedia.org/wiki/Extract,_transform,_load