COMP2005 Laboratory Sheet 9: CNN

1. PyTorch & Torchvision

In the previous lab session, OpenCV was the main library we used to implement most image processing algorithms. However, OpenCV does not provide efficient methods for the development of deep neural networks. There are many deep learning frameworks (e.g., PyTorch, Caffe and TensorFlow) integrated with forward propagation, backpropagation, and gradient descent, which enable us to materialise neural networks' training and inference in a short time.

PyTorch is famous for its simplicity and ease of use, which is widely used by researchers in both academia and industry. Therefore, in this lab session, we will learn to develop deep neural networks with PyTorch. Before you start, you need to install PyTorch in your environment. You can follow the instructions at PyTorch's official website: https://pytorch.org/get-started/locally/. You may select different installation commands of PyTorch as per your devices and requirements (Figure 1).

Figure 1

A PyTorch of only CPU is enough for this lab session since we will not implement a large model and training the model of this lab session with CPU only takes a few minutes. If your device is equipped with a NVIDIA GPU which supports CUDA and cuDNN, you can install a CUDA version which will significantly increase the training and inference speed of the neural network through hardware acceleration. In terms of installing PyTorch with CUDA support, Anaconda is highly suggested (especially if you are using Windows) because Anaconda automatically installs PyTorch's dependencies of CUDA and cuDNN.

On top of that, we will use Torchvision which provides an abundance of APIs for image processing related to deep learning and OpenCV and NumPy for some image processing steps.

2. Overview of Deep Learning Developments

There are five core components of the development of deep learning techniques:

1) **Data Preparation**: organising the data for training and testing the model, including loading images and annotations from the local directory, preprocessing the images, and matching each image with the corresponding annotation.
2) **Network Construction**: defining the structure of the deep neural network.
3) **Loss Function**: implementing the loss function measuring the distance between the model's output and the ground truth (also known as annotations and labels).
4) **Training**: integrating the above three components to materialise the training pipeline of the model.
5) **Inference**: implementing the prediction pipeline of the well-trained model.

In this lab session, we will implement a classification model for handwritten numeral recognition, which covers all the five components introduced above. In the following contents of this lab session, each component will be implemented one by one.

3. Data Preparation

In this lab session, we use MNIST [1] dataset to train and test our model, a famous benchmark for handwritten numeral recognition. It contains 60000 images for training and 10000 images for testing, where every image is 28 by 28 pixels. This dataset is available on Moodle. Download the dataset and observe how it is organised. As for this dataset, it follows the structure below:

```
MNIST
|___mnist_train
|    |___0
|    |    |___image0
|    |    |___image1
|    |    ......
|    |
|    |___1
|    |___2
|    |___3
|    |___4
|    |___5
|    |___6
|    |___7
|    |___8
|    |___9
|
|___mnist_test
     |___0
     |___1
     |___2
     |___3
     |___4
     |___5
     |___6
     |___7
     |___8
     |___9
```

We can see that the dataset is divided into two folders: one is for training and the other is for testing. In each folder, there are 10 sub-folders containing images. For the images in a sub-folder, the name of the sub-folder represents its category, i.e., the training set for images of "0" is under the sub-folder name "MNIST/mnist_train/0/". After observing the dataset structure, we can start to write the code for data preparation.

First, let's create a Python file called *DataLoader.py* and import the libraries used in this component:

```
from torch.utils.data import Dataset
import os, random, cv2
import torchvision.transforms.functional as TF
```

torch.utils.data.Dataset is required to customise our data preprocessing. We need to write a class that inherits from it to define how the data is loaded from the disk and how the images and labels are matched. *torchvision.transforms.functional* provides a series of APIs of image processing and data type conversion from NumPy/OpenCV to PyTorch tensor (similar to that of ndarray in NumPy), and vice versa. Remember, the input and output of the neural network should always be a PyTorch tensor, and they should be converted to another format (e.g., NumPy array) if further pre-processing and post-processing are required.

After importing those libraries, we need to write a class inherited from *torch.utils.data.Dataset*. Specifically, three methods should be rewritten by us, which means we are not allowed to change the names of these three methods, as shown below.

```
class MNIST(Dataset):
    def __init__(self):
        super(MNIST, self).__init__()
    def __getitem__(self, index):
        …
    def __len__(self):
        …
```

In *self.__init__()*, we need to load the paths of all the images (you can also load every image instead if your device has enough RAM, but it is not recommended because we can do it in *self.__getitem__()*), and match each image with its ground truth label). First of all, let's add a parameter to *self.__init__()* to indicate which part of the dataset. It has two values: "train" or "test" depending on if we want to load the training set or test set, respectively.

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
```

After that, let's set the root path of our dataset according to the partition. Here we are using a relative path since our .py scripts and dataset are under the same folder or you can change it to an absolute path according to the location of the dataset in your devices.

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
        if partition == "train":
            root_path = "./MNIST/mnist_train/"
        else:
            root_path = "./MNIST/mnist_test/"
```

Since we aim to load all the image paths and labels and match them, we can initialise an empty list whose elements are tuples (image path, label).

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
        # create a list of image path - label pair: [(image_path, label), ...]
        self.image_path_label_pairs = []
```

By observing the dataset structure, we can see that each subfolder's name is the category of images under it. Hence, we can use *os.listdir()* to obtain the class list of all categories.

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
        categories = os.listdir(root_path)
```

Since images of the same class are under the sub-folder of the class name, we can summarise the image path and label pair by getting each image's path and the name of the corresponding sub-folder.

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
        # load images of each sub-folder (category)
        for category in categories:
            # load image names under the current sub-folder
            image_names = os.listdir(root_path + category)
            # load image paths and corresponding categories
            self.image_path_label_pairs.extend([(root_path + category + "/" + image_name, int(category)) for image_name in image_names])
```

Finally, we can shuffle the image path - label pairs, but this is not necessary because it can be done in other components.

```
class MNIST(Dataset):
    def __init__(self, partition="train"):
        ...
        random.shuffle(self.image_path_label_pairs)
```

Then let's rewrite the *self.__getitem__()* method. This method is called when a mini-batch of training/testing data is going to be generated. In this method, we should retrieve an image path - label pair, and load and pre-process the image and label. *self.__getitem__()* has a parameter *index* which indicates the index of the image path - label pair among all image path - label pairs. First, let's retrieve an image path - label pair according to the index.

```
class MNIST(Dataset):
    def __getitem__(self, index):
        image_path_label_pair = self.image_path_label_pairs[index]
```

After that, obtain the image path and label of the image, respectively.

```
class MNIST(Dataset):
    def __getitem__(self, index):
        …
        image_path = image_path_label_pair[0]
        label = image_path_label_pair[1]
```

Then, let's read the image by loading the image path with OpenCV. Note that the image should be loaded in greyscale.

```
class MNIST(Dataset):
    def __getitem__(self, index):
        ...
        image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
```

Next, the image should be converted from OpenCV (NumPy array) to PyTorch tensor.

```
class MNIST(Dataset):
    def __getitem__(self, index):
        ...
        torch_image = TF.to_tensor(image)
```

Finally, the image is normalised with the given mean and standard deviation, and the processed image and its label are returned.

```
class MNIST(Dataset):
    def __getitem__(self, index):
        ...
        torch_image = TF.normalize(torch_image, mean=[0.1307],
        std=[0.3081])
        return torch_image, label
```

To finish the data preparation, let's rewrite *self.__len__()*. In this method, we need to set the total number of image path - label pairs.

```
class MNIST(Dataset):
    def __len__(self):
        return len(self.image_path_label_pairs)
```

4. Network Construction

In this component, we need to initialise all the layers and their parameters and organise the data flow of the forward propagation. In this lab session, we will create a small neural network which comprises two convolutional layers, two max-pooling layers, and three fully connected layers. Create a Python file called *net.py* and import the necessary dependencies:

```
import torch.nn as nn
```

torch.nn consists of different kinds of layers, such as CNN, pooling, normalisation, and activation functions. To customise our model, we need to create a class inheriting from *torch.nn.Module* which integrates the backpropagation so that we do not have to consider the calculation of the gradients of parameters of the model. What we need to consider is only the structure of the model and its forward propagation.

```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        …
    def forward(self, x):
        …
```

In the example above, we have created a neural network named *ToyModel*. We shall initialise all the layers of the model in *self.__init__()*, and define the forward propagation in *self.forward()*. Now, let's create two convolutional layers using *nn.Conv2d* where in_*channels* and *out_channels* are the number of channels of the input and output tensor respectively and *kernel_size* is the size of the convolutional kernel:

```
class ToyModel(nn.Module):
    def __init__(self):
        ...
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=8,
        kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,
        kernel_size=5)
```

Then let's create a max pooling layer and an activation layer (we will use ReLU as the activation function). We will create a max pooling layer using *nn.MaxPool2d* where *kernel_size* is the size of the window and *stride* is the stride of the window's movement.

Since there are no parameters to be learned in these two layers, we will only create one of each and they can be reused again if necessary.

```
class ToyModel(nn.Module):
    def __init__(self):
        ...
        self.pooling = nn.MaxPool2d(kernel_size=2, stride=2)
        self.relu = nn.ReLU()
```

After that, three consecutive fully connected layers are created to generate the confidence score of each category of the input image. This is done by using *nn.Linear* where *in_features* and *out_features* represent the number of dimensions of the input and output feature vectors respectively. Since there are 10 categories in MNIST (from 0 to 9), the *out_features* of the classification head is set to 10.

```
class ToyModel(nn.Module):
    def __init__(self):
        ...
        # create fully connected layers
        self.FC1 = nn.Linear(in_features=16*4*4, out_features=64)
        self.FC2 = nn.Linear(in_features=64, out_features=32)
        # create classification head
        self.cls = nn.Linear(in_features=32, out_features=10)
```

Now let's rewrite the *self.forward()* function. It receives one parameter *x* as the input which is a 4D tensor *(mini-batch size, channel, height, width)*, where *mini-batch size* is the number of images in the mini-batch, *channel* is the number of channels in each image and *height* and *weight* represents the height and width of the image. Since we pre-processed the MNIST images to grayscale earlier, *channel* will be set to 1. Both *height* and *width* will be set to 28 since each image of MNIST is 28 by 28 pixels.

Next, we shall define the execution sequence of the two predefined convolutional layers. Each CNN layer is a combination of a convolutional layer and an activation function layer. A max-pooling layer is created after each CNN layer to reduce the size of feature map outputs by the CNN layer.

```
class ToyModel(nn.Module):
    def forward(self, x):
        # conv layer 1
        x = self.conv1(x)
        # activation
        x = self.relu(x)
        # max pooling
        x = self.pooling(x)
```

```
        # conv layer 2
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pooling(x)
```

The output feature map till this point is of the size *(mini-batch size, 16, 4, 4)*. However, the input size of the following fully connected layer should be *(mini-batch size, 16x4x4)*. Therefore, the current output is required to be flattened to the size of *(mini-batch size, 16x4x4)*.

```
class ToyModel(nn.Module):
    def forward(self, x):
        ...
        x = x.reshape(x.size(0), 16*4*4)   # batch size, in_features, out_features
```

Note that *x.size(0)* returns the size of the 0^{th} dimension of *x*. Finally, let's connect the fully connected layers to the end of the convolutional layers. Activation functions are required after each fully connected layer except for the classification head.

```
class ToyModel(nn.Module):
    def forward(self, x):
        ...
        # fully connected layer 1
        x = self.FC1(x)
        # activation
        x = self.relu(x)
        # fully connected layer 2
        x = self.FC2(x)
        x = self.relu(x)
        # classification layer, i.e., the output layer
        x = self.cls(x)
        return x
```

5. Loss Function

As for classification tasks, cross-entropy loss is widely used to measure the difference between the prediction of the model and the ground truth. PyTorch has provided us with an API of cross-entropy loss, i.e. *torch.nn.CrossEntropyLoss*. However, in your further research, you may witness many kinds of loss functions which are not built-in by PyTorch. Therefore, you may be interested to know how to customise your own loss function. In this section, we will implement cross-entropy loss with basic PyTorch operations to show you how to

customise the loss function with PyTorch. You can also skip this section and come back at any time in the future.

First, let's create a Python file called *LossFunction.py* and import the dependencies:

```
import torch
import torch.nn as nn
```

Similar to what we have done in the last section, we can customise a loss function by creating a class inheriting from *torch.nn.Module* so that the calculation of gradients of parameters can be calculated automatically with PyTorch instead of materialising it by ourselves. We need to define the procedure of the calculation of the loss function by rewriting the *self.forward()* method.

```
class MyCrossEntropyLoss(nn.Module):
    def __int__(self):
        super(MyCrossEntropyLoss, self).__int__()
    def forward(self):
        …
```

As for our customised cross-entropy loss, it receives three parameters: *predictions, labels* and *reduction*. *predictions* is the predicted confidence scores output by the model and is of the size *(mini-batch size, 10)*. *labels* is a list of the size *(mini-batch size)* where each element is the ground truth category of each sample. *reduction* has two optional values: *"mean"* which refers to averaging the loss of each image in the mini-batch and *"sum"* which calculates the summation of it.

```
class MyCrossEntropyLoss(nn.Module):
    def forward(self, predictions, labels, reduction="mean"):
        ...
```

In terms of cross-entropy loss, firstly the predictions of the model are softmaxed:

```
class MyCrossEntropyLoss(nn.Module):
    def forward(self, predictions, labels, reduction="mean"):
        probabilities = torch.exp(predictions) /
        torch.sum(torch.exp(predictions), dim=1).reshape(predictions.size(0),
        1)
```

After that, the probability of each image's ground truth category predicted by the model after being softmaxed is selected by fancy indexing of PyTorch:

```
class MyCrossEntropyLoss(nn.Module):
    def forward(self, predictions, labels, reduction="mean"):
        ...
        probabilities = probabilities[range(probabilities.size(0)), labels]
```

Finally, let's calculate the cross-entropy loss and average or sum among the mini-batch:

```
class MyCrossEntropyLoss(nn.Module):
    def forward(self, predictions, labels, reduction="mean"):
        ...
        # calculate cross-entropy
        loss = - torch.log(probabilities)
        # average the loss of each sample or calculate summation
        if reduction == "mean":
            loss = torch.sum(loss) / loss.size(0)
        elif reduction == "sum":
            loss = torch.sum(loss)
        return loss
```

6. Training

In this section, we will integrate all the above three components to conduct the training of the model. First, let's create a Python file called *train.py* and import the three modules developed by us earlier:

```
from DataLoader import MNIST
from net import ToyModel
from LossFunction import MyCrossEntropyLoss
```

We also need to import other necessary dependencies:

```
import torch
import torch.nn as nn
import torch.utils.data as Data
```

Then we need to set some hyperparameters where *BATCH_SIZE* is the number of images in a mini-batch for each iteration, *EPOCH* is the number of epochs (1 epoch means the **entire** training dataset is passed through once) and *LR* is the learning rate.

```
BATCH_SIZE = 32
EPOCH = 10
LR = 0.0001
```

After that, let's initialise each component one by one. Firstly, we shall initialise the neural network.

```
model = ToyModel()
```

Then, we are encouraged to set the training device, e.g., CPU or GPU. You can use GPU to accelerate the training process if you have installed a PyTorch supporting CUDA. But training our toy model on the CPU is sufficient.

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

By default, the model is on the CPU when it is initialised, and we need to move it to the training device.

```
model.to(train_device)
```

We also need to set the model to training mode.

```
model.train()
```

Now let's initialise the training set. Specifically, we need an iterator which returns images and labels of a mini-batch per iteration/loop. We can easily achieve this goal with *torch.utils.data*. First, let's create an instance of the training set.

```
mnist = MNIST(partition="train")
```

Then we use *torch.utils.data* to get an iterator. Set the parameters of *torch.utils.data.DataLoader* where *dataset* receives the instance of our training set, *batch_size* is the size of the mini-batch returned per iteration, and *shuffle* indicates whether to shuffle the datasets. Setting *pin_memory* as True can reduce the time taken for each iteration. When the number of samples not yet visited during an epoch is less than the mini-batch size, they will be dropped if *drop_last* is True. The time taken to load the training set can be reduced by using multi-threads, where *num_workers* is the number of threads.

```
loader = Data.DataLoader(
        dataset=mnist,
        batch_size=BATCH_SIZE,
        shuffle=True,  # it is good to shuffle the dataset
        pin_memory=True,
        drop_last=True,
        num_workers=1,
)
```

Now, it's time to initialise the loss function. Here you have two options: using cross-entropy loss provided by PyTorch or using our customised cross-entropy loss:

```
# initialise loss function provided by PyTorch
loss_func = nn.CrossEntropyLoss(reduction="mean")
# initialise our customised cross-entropy loss
my_loss_func = MyCrossEntropyLoss()
```

The last step before training is to initialise an optimizer to conduct backpropagation and gradient descent. PyTorch has provided many different kinds of optimizers, such as SGD[2], RMSprop[3], and Adam[4]. In this lab session, we will use Adam as our optimizer.

```
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
```

Finally, we can launch the training. A *for* loop is used to go through all epochs, and for each epoch, another *for* loop is used to get a mini-batch of images and labels per iteration.

```
for epoch in range(EPOCH):
    for torch_images, labels in loader:
        ...
```

It is necessary to move them to the training device, too, i.e., the model, images, and labels should be on the same device.

```
for epoch in range(EPOCH):
    for torch_images, labels in loader:
        …
        torch_images = torch_images.to(train_device)
        labels = labels.to(train_device)
```

Then we can obtain the model's prediction by calling its forward method:

```
for epoch in range(EPOCH):
    for torch_images, labels in loader:
        ...
        predictions = model.forward(torch_images)
```

After getting the model's prediction, you can calculate the cross-entropy loss with PyTorch's built-in method or with our customised method.

```
for epoch in range(EPOCH):
    for torch_images, labels in loader:
        ...
        # calculate loss function with PyTorch's method
        loss = loss_func(predictions, labels)
        # calculate loss function with our customised method
        loss = my_loss_func(predictions, labels)
```

Next, we can conduct the backpropagation and gradient descent:

```
for epoch in range(EPOCH):
    for torch_images, labels in loader:
        ...
        optimizer.zero_grad()  # clear the history accumulated gradients
        loss.backward()  # back propagation
        optimizer.step()  # gradient descent
        print('epoch:', epoch, '|my train loss:%.4f' % loss, '|official train
        loss:%.4f' % loss_func(predictions, labels))
```

After the execution of each epoch, we can save the weights of the model.

```
for epoch in range(EPOCH):
    …
    torch.save(model.state_dict(), './module/net' + str(epoch) + '.pkl')
```

Now you can run the code for training. You are expected to see the following output (You will find that the loss function value returned by our customised cross-entropy loss and that returned by the official cross-entropy loss provided by PyTorch are the same):

```
epoch: 0 |my train loss:0.3601 |official train loss:0.3601
epoch: 0 |my train loss:0.6156 |official train loss:0.6156
epoch: 0 |my train loss:0.3401 |official train loss:0.3401
epoch: 0 |my train loss:0.3496 |official train loss:0.3496
epoch: 0 |my train loss:0.4090 |official train loss:0.4090
epoch: 0 |my train loss:0.2741 |official train loss:0.2741
epoch: 0 |my train loss:0.4032 |official train loss:0.4032
epoch: 0 |my train loss:0.2809 |official train loss:0.2809
epoch: 0 |my train loss:0.3494 |official train loss:0.3494
```

7. Inference

After training the model, we shall make the model predict test images which are not in the training dataset. First, let's create a Python file called *inference.py* and import the necessary dependencies.

```
from DataLoader import MNIST
from net import ToyModel
from torchvision.transforms.functional import normalize
import torch.utils.data as Data
import torch
import numpy as np
import cv2
```

Then create the *denormalize()* method which is the inverse of normalisation. This is needed because the image was normalised by the data loader and it is difficult to visualise a normalised image.

```
def denormalize(tensor, mean, std):
    mean = np.array(mean)
    std = np.array(std)
    _mean = -mean / std
    _std = 1 / std
    return normalize(tensor, _mean, _std)
```

Besides the preparation of the optimizer and the loss function, the procedure before inference is similar to that of training the model.

```
# initialise inference devices
test_device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# initialise the dataset
mnist = MNIST(partition="test")
# initialise the dataloader
loader = Data.DataLoader(
        dataset=mnist,
        batch_size=1,
        shuffle=True,
        pin_memory=True,
        drop_last=True,
        num_workers=1,
)
```

For loading the well-trained model, we can create an untrained model and load the saved weights to it before moving it to the test device.

```
# load the model's well-trained weight
model = ToyModel()
model.load_state_dict(torch.load("./module/net9.pkl"))
# move the model to the test device
model.to(test_device)
```

For inference, we need to set the model to “evaluation” mode.

```
model.eval()
```

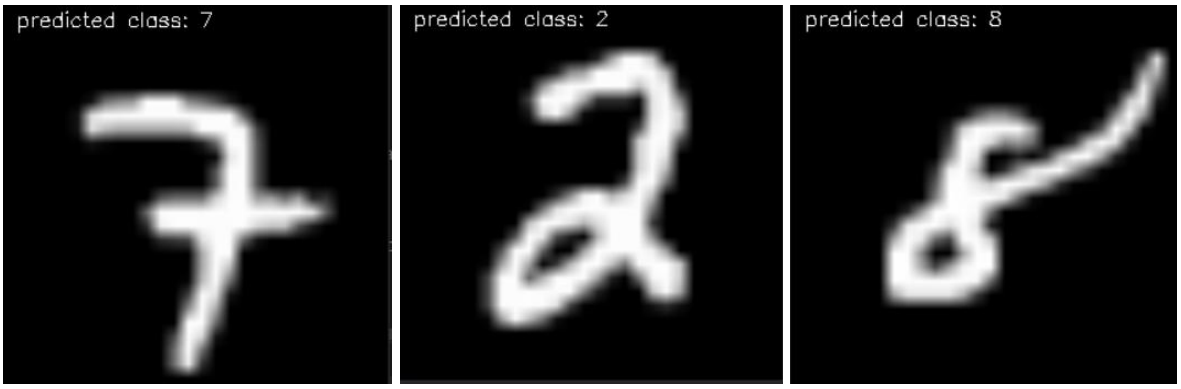
Then we can use the following statements to obtain the predicted category of a test image:

```
for torch_images, labels in loader:
        # move the images to the test device
        torch_images = torch_images.to(test_device)
        labels = labels.to(test_device)
        # obtain the output of the model
        predictions = model.forward(torch_images)
        # obtain the predicted category
        _, predicted_class = torch.max(predictions, dim=1)
```

Lastly, we use the following statements to visualise the prediction result.

```
for torch_images, labels in loader:
    …
    # denormalise the torch image for visualisation
    image = denormalize(torch_images[0], mean=[0.1307], std=[0.3081])
    # change the image from PyTorch tensor to OpenCV (NumPy array)
    image = (image.cpu().numpy() * 255).astype(np.uint8)[0]
    # enlarge the image for visualisation (the original image size is 28x28)
    image = cv2.resize(image, (280, 280))
    # put the predicted category on the image for visualisation
    cv2.putText(image, "predicted class: " + str(predicted_class.cpu().numpy()[0]),
    (10, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, 255, 1)
    # visualisation
    cv2.imshow("", image)
    # wait for user input
    pressedKey = cv2.waitKey(0) & 0xFF
    if pressedKey == ord('q'):
        # press q to quit
        break
    else:
        continue
cv2.destroyAllWindows()
```

You are expected to see the following output (You can press “Q” to quit or any other key to show the next image and its prediction):



8. References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2323, 1998.

[2] Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." International conference on machine learning. PMLR, 2013.

[3] Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[4] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).