European IT Certification Curriculum
Self-Learning Preparatory Materials

EITC/AI/DLPTFK

Deep Learning with Python, TensorFlow and Keras

This document constitutes European IT Certification curriculum self-learning preparatory material for the EITC/AI/DLPTFK Deep Learning with Python, TensorFlow and Keras programme.

This self-learning preparatory material covers requirements of the corresponding EITC certification programme examination. It is intended to facilitate certification programme's participant learning and preparation towards the EITC/AI/DLPTFK Deep Learning with Python, TensorFlow and Keras programme examination. The knowledge contained within the material is sufficient to pass the corresponding EITC certification examination in regard to relevant curriculum parts. The document specifies the knowledge and skills that participants of the EITC/AI/DLPTFK Deep Learning with Python, TensorFlow and Keras certification programme should have in order to attain the corresponding EITC certificate.

Disclaimer

This document has been automatically generated and published based on the most recent updates of the EITC/AI/DLPTFK Deep Learning with Python, TensorFlow and Keras certification programme curriculum as published on its relevant webpage, accessible at:

https://eitca.org/certification/eitc-ai-dlptfk-deep-learning-with-python,-tensorflow-and-keras/

As such, despite every effort to make it complete and corresponding with the current EITC curriculum it may contain inaccuracies and incomplete sections, subject to ongoing updates and corrections directly on the EITC webpage. No warranty is given by EITCI as a publisher in regard to completeness of the information contained within the document and neither shall EITCI be responsible or liable for any errors, omissions, inaccuracies, losses or damages whatsoever arising by virtue of such information or any instructions or advice contained within this publication. Changes in the document may be made by EITCI at its own discretion and at any time without notice, to maintain relevance of the self-learning material with the most current EITC curriculum. The self-learning preparatory material is provided by EITCI free of charge and does not constitute the paid certification service, the costs of which cover examination, certification and verification procedures, as well as related infrastructures.

TABLE OF CONTENTS

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: INTRODUCTION
TOPIC: DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS

Deep learning with Python, TensorFlow, and Keras is a powerful combination that allows for the development and implementation of complex neural networks. In this material, we will provide an introduction to deep learning and explore the basics of neural networks.

Neural networks are machine learning models that aim to map input data to a desired output. For example, we may want to determine whether an image contains a dog or a cat. The input data, represented as X1, X2, X3, is fed into the neural network. The output layer consists of neurons that represent the possible outputs, such as dog or cat.

To map the inputs to the output, we use hidden layers. Each input is connected to the neurons in the hidden layer, and each connection has a unique weight associated with it. By introducing hidden layers, we can capture nonlinear relationships between the inputs and the desired output. A neural network with one hidden layer is considered a basic neural network, while a neural network with two or more hidden layers is referred to as a deep neural network.

At the individual neuron level, the inputs are summed together, taking into account the unique weights associated with each input. This sum is then passed through an activation function, which simulates the firing of a neuron. Common activation functions include step functions and sigmoid functions, which return values between 0 and 1.

Python, TensorFlow, and Keras provide high-level APIs that simplify the implementation of deep learning models. These tools allow users to focus on the design and architecture of the neural network, rather than getting caught up in low-level TensorFlow code.

To get started with deep learning, it is recommended to have Python 3.6 or later installed. TensorFlow currently supports Python 3.6, but support for later versions may be available soon. Once the necessary software is installed, users can begin exploring the world of deep learning and building their own neural networks.

Deep learning with Python, TensorFlow, and Keras offers a simplified approach to developing and working with neural networks. By understanding the basics of neural networks, including the use of hidden layers and activation functions, users can leverage these tools to solve complex machine learning problems.

In deep learning, the final output layer of a neural network typically uses a sigmoid activation function. This function assigns probabilities to different classes or categories. For example, let's say we have two classes: dog and cat. The output layer might assign a probability of 0.79 to dog and 0.21 to cat. These probabilities add up to 1.0. To determine the predicted class, we take the class with the highest probability, which in this case is dog with 79% confidence.

To build a neural network, we need to import the TensorFlow library. You can do this by running the command "pip install --upgrade tensorflow". Make sure you have TensorFlow version 1.1 or greater. Once imported, you can check the current version using "import tensorflow as tf" and "tf.version".

Next, we need to import a dataset. In this example, we will use the MNIST dataset, which consists of 28x28 images of handwritten digits from 0 to 9. Each image is a unique representation of a digit. The goal is to feed the pixel values of these images into the neural network and have it predict the corresponding digit.

To import the dataset, we can use the "tensorflow.keras.datasets.mnist" module. We will unpack the dataset into training and testing variables, namely "X_train", "Y_train", "X_test", and "Y_test".

To visualize the dataset, we can use the "matplotlib" library. Import it using "import matplotlib.pyplot as plt". Then, use "plt.imshow(X_train[0])" to display the first image in the training set. The image will be in black and white, as it is a binary representation of the digit.

Before training the neural network, it is important to normalize the data. In this case, the pixel values range

from 0 to 255. We can scale these values between 0 and 1 using the "tf.keras.utils.normalize" function. Apply this function to both the training and testing data.

Now, let's build the model. We will use the "tf.keras.models.Sequential" type of model, which is a common choice for feed-forward neural networks. The first layer of the model is the input layer. Since our images are 28x28, we need to flatten them into a 1D array. This can be done using the "model.add(tf.keras.layers.Flatten())" syntax.

After the input layer, we can add additional layers to the model using the "model.add" syntax. These layers can be fully connected layers, convolutional layers, or other types of layers depending on the problem at hand.

We can use the "flattened" layer from the "chaos" library as one of the built-in layers in our deep learning model. This layer is specifically useful when working with Convolutional Neural Networks (CNNs) as it helps to flatten the output before passing it to the next layer. In our case, we will use it as the input layer to simplify our model.

Moving on to the hidden layers, we will add two dense layers using the "model.add" syntax. In each dense layer, we will specify the number of units (neurons) and the activation function. For our model, we will use 128 units in each hidden layer and the rectified linear activation function (ReLU) as it is a commonly used default activation function.

Next, we will add the output layer, which will also be a dense layer. The number of units in the output layer will depend on the classification task at hand. In our case, we have 10 classifications, so we will use 10 units. However, the activation function for the output layer will be different. Since we want to obtain a probability distribution, we will use the softmax activation function.

With the model architecture defined, we now need to set some parameters for training the model. We will use the "model.compile" function to specify the optimizer, loss metric, and the metrics we want to track during training. The optimizer we will use is the Adam optimizer, which is a commonly used optimizer in deep learning. For the loss metric, we will use categorical cross-entropy, which is suitable for multi-class classification tasks. Finally, we will track the accuracy metric during training.

Once all the parameters are defined, we can proceed to train the model using the "model.fit" function. We will pass the training data (X_train and y_train) and specify the number of epochs (total number of training iterations). In this case, we will use 3 epochs.

After training the model, we can evaluate its performance. In this case, we achieved a 97% accuracy after only three epochs, which is quite good. However, it is important to note that this accuracy is obtained on the training data, and we need to check if the model has overfit the data.

Deep learning is a powerful technique in the field of artificial intelligence that allows models to learn patterns and generalize from data. The goal is for the model to understand the underlying attributes that distinguish different classes or categories, rather than simply memorizing every single sample it is trained on.

To evaluate the performance of a deep learning model, it is important to calculate the validation loss and validation accuracy. This can be done using the `evaluate` function in TensorFlow and Keras. By passing the test data (`X_test` and `Y_test`) to this function, we can obtain the loss and accuracy values. In the example given, the loss is approximately 0.11 and the accuracy is around 96.5%. It is worth noting that the out-of-sample accuracy may be slightly lower and the loss slightly higher compared to the training accuracy and loss. This is expected and indicates that the model is generalizing well.

It is important to monitor the difference between the training and validation metrics. If there is a large difference, it suggests that the model may be overfitting the training data. In such cases, it is advisable to adjust the model to prevent overfitting.

In addition to evaluating the model, it is also useful to know how to save and load a trained model. This can be done using the `save` and `load_model` functions in TensorFlow and Keras. By saving the model, we can reuse it later for predictions or further training. The saved model can be loaded using the `load_model` function, providing the exact model name.

To make predictions with a loaded model, we can use the `predict` function. It is important to note that the `predict` function expects a list as input. In the given example, the predictions are stored in the variable `predictions`, and the input is the test data `X_test`. The predictions are in the form of one-hot arrays, which represent probability distributions for each class. To extract the predicted class, we can use the `argmax` function from the NumPy library.

In the example, the predicted class for the first test sample (`X_test[0]`) is 7. This means that the model predicts the sample belongs to class 7. To visualize the sample, the `plt` library can be used. By plotting the pixel values of the sample, we can see the image representation of the digit.

This covers the basics of deep learning with Python, TensorFlow, and Keras. It is important to note that this is just an introduction and there are many more advanced topics to explore. Some of these topics include loading external datasets, using TensorBoard for model visualization, and troubleshooting models when they don't perform as expected.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: DATA
TOPIC: LOADING IN YOUR OWN DATA

In this tutorial, we will be discussing how to load an external dataset in deep learning using Python, TensorFlow, and Keras. Specifically, we will be using the cats and dogs dataset from Microsoft, which was initially a Kaggle challenge. The objective of this tutorial is to train a neural network to identify whether an image contains a cat or a dog.

To begin, you will need to download the cats and dogs dataset. Once downloaded, you should see two directories: "cat" and "dog". These directories contain images of cats and dogs, respectively. Each directory contains approximately 12,500 samples, providing ample examples to train a model.

Before we proceed, let's ensure that we have the necessary libraries installed. We will be using numpy for array operations, matplotlib for visualizing images, and OpenCV (cv2) for image operations. If you do not have these libraries installed, you can use the following commands to install them:
- numpy: `pip install numpy`
- matplotlib: `pip install matplotlib`
- OpenCV: `pip install opencv-python`

Now that we have the required libraries, let's start by specifying the data directory. In this case, the data is located in the "data sets" folder under "pip images" in the "X files" directory. We will also define the categories we need to deal with, which are "dog" and "cat".

Next, we will iterate through all the examples of dogs and cats. To do this, we will use the `os` library to join the data directory path with the category path. Then, we will use `os.listdir` to iterate through all the images in that path. For each image, we will convert it to an array using `cv2.imread`, which reads the image from the specified path. We will also convert the image to grayscale using `cv2.cvtColor` since we do not believe that color is essential for this specific task.

To visualize the images, we will use `plt.imshow` from the matplotlib library. We will set the colormap to "gray" to display the grayscale image. This step is optional but can help us verify that the images are as expected.

In the end, we will have an array representation of the images, which can be used for further processing and training our deep learning model.

It's important to note that the tutorial assumes basic familiarity with Python and deep learning concepts.

In the field of artificial intelligence, deep learning is a popular technique that involves training neural networks with large amounts of data to perform complex tasks. One important aspect of deep learning is loading and preprocessing data, which we will explore in this didactic material.

When working with images in deep learning, it is often necessary to normalize them to a consistent shape. Although it is possible to use variable-sized images for classification, it is generally simpler to resize all images to the same shape. For example, we can choose an image size of 50 by 50 pixels.

To resize the images, we can use the `resize` function from the `cv2` library. This function takes the image array and the desired image size as parameters, and returns the resized image array. We can then display the resized image using the `imshow` function from the `matplotlib` library, specifying a gray color map.

It is important to note that the choice of image size depends on the specific task and the characteristics of the images. Smaller image sizes may lead to loss of details, while larger image sizes may capture more fine-grained features. In the example discussed, an image size of 50 by 50 pixels was deemed suitable for classification.

Once the desired image size is determined, we can proceed to create the training dataset. We start by initializing an empty list called `training_data`. Then, we define a function called `create_training_data` to iterate through the images and build the dataset.

In this function, we need to map the class labels (e.g., "dog" and "cat") to numerical values. We can do this by assigning the index of each class label in a list of categories to a corresponding numerical value. For example, we can assign 0 to "dog" and 1 to "cat". This mapping allows us to convert the class labels to numerical values that can be processed by the neural network.

Next, we iterate over the images, resize them using the previously determined image size, and append the resized image array and its corresponding class label to the `training_data` list. It is also recommended to handle any potential errors that may occur during the resizing process.

After creating the training dataset, it is important to ensure that the data is properly balanced. In the case of binary classification, such as distinguishing between cats and dogs, it is ideal to have an equal number of samples for each class. This balance helps prevent biases in the training process. However, if the number of samples for each class is different, class weights can be used during model training to account for the imbalance.

To check the balance of the training data, we can print the length of the `training_data` list. This will give us an indication of how many samples we have for each class and whether any class imbalances need to be addressed.

When working with deep learning and image classification tasks, it is important to normalize the images to a consistent size. This can be achieved by resizing the images using the appropriate libraries and functions. Additionally, creating a balanced training dataset is crucial for accurate model training.

To ensure the effectiveness of deep learning models, it is important to properly handle imbalanced datasets. One approach is to balance the dataset by equalizing the number of samples for each class. This can prevent the model from favoring the majority class and improve its overall performance.

Another important step is to shuffle the data before feeding it into the neural network. Shuffling the data ensures that the model does not learn patterns based on the order of the samples. Instead, it learns from a random mix of samples, which can enhance its ability to generalize to unseen data.

To balance the dataset, you can import the 'random' module and use the 'shuffle' function. This function shuffles the training data, ensuring that it is evenly distributed between the different classes. For example, you can use the following code:

```
1. import random
2.
3. random.shuffle(training_data)
```

After shuffling the data, you can proceed to pack it into variables that will be used as input for the neural network. Typically, you will have a feature set (X) and corresponding labels (Y). To store these, you can create empty lists for X and Y. For example:

```
1. X = []
2. Y = []
```

Next, you can iterate over the shuffled training data and append the features and labels to the respective lists. This will organize the data in a format that can be fed into the neural network. Here is an example of how you can achieve this:

```
1. for features, label in training_data:
2.     X.append(features)
3.     Y.append(label)
```

Before feeding the data into the neural network, it is important to convert the feature set (X) into a NumPy array and reshape it appropriately. The reshape operation ensures that the data is in the correct format for the neural network to process. For example, if your images are grayscale and have a fixed size, you can use the following code:

```
1. import numpy as np
```

```
2.
3. X = np.array(X).reshape(-1, image_size, image_size, 1)
```

In this code, 'image_size' represents the size of the images in your dataset. The '-1' in the reshape function indicates that the number of samples can vary, while '1' represents the number of channels (grayscale images have one channel).

Finally, it is a good practice to save the preprocessed data to avoid repeating the preprocessing steps in future runs. You can use various methods to save the data, such as pickle or saving it in a specific file format. Saving the data allows you to load it quickly for subsequent model training or evaluation.

When working with your own data in deep learning, it is crucial to balance the dataset and shuffle the data before feeding it into the neural network. Balancing the dataset ensures equal representation of each class, while shuffling prevents the model from learning patterns based on sample order. Additionally, converting the feature set into a NumPy array and reshaping it correctly prepares the data for the neural network. Saving the preprocessed data can also save time and resources in future experiments.

In the process of working with neural networks, it is common to continuously make adjustments to the model. This means that you may not have all the answers right away, especially when dealing with more complex problems. Therefore, it is important to avoid rebuilding your dataset every time you make a change.

To address this issue, we can make use of a Python library called "pickle". Pickle allows us to save our data in a serialized format, so that we can easily load it back when needed.

To save our training data, we can start by importing the "pickle" module. Then, we can open a file using the "open()" function and assign it to a variable, let's say "pickle_out". We can then use the "dump()" function from the "pickle" module to save our data. In this case, we want to save the variable "X", which represents our features. Finally, we can close the file using the "close()" method. We can follow the same process to save the variable "Y", which represents our labels.

To read the data back in, we can open the file using the "open()" function and assign it to a variable, let's say "pickle_in". We can then use the "load()" function from the "pickle" module to load the data. In this case, we want to load the variable "X". We can assign the loaded data to a new variable, let's say "x1", which represents our images. Similarly, we can load the variable "Y" and assign it to a new variable, let's say "y1", which represents our labels.

In the next tutorial, we will take the dataset that we have compiled and feed it through a convolutional neural network. Before that, we will cover some basics of convolution and related concepts. If you have any questions or suggestions, please feel free to leave them below. Otherwise, I will see you in the next tutorial.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: CONVOLUTIONAL NEURAL NETWORKS (CNN)
TOPIC: INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS (CNN)

Convolutional neural networks (CNNs) are a type of deep learning model that are particularly effective in processing and analyzing image data. In this tutorial, we will explore the basics of CNNs and how they can be applied to classify dogs versus cats using the Python programming language, TensorFlow, and Keras.

To understand how CNNs work, let's start with the basic steps involved. The first step is convolution, which involves applying a convolutional window to the input image. This window, often referred to as a filter or kernel, is a small matrix that slides over the image, performing a mathematical operation at each position. The purpose of convolution is to extract useful features from the image.

After convolution, the next step is pooling. Pooling is a downsampling operation that reduces the dimensionality of the feature maps. The most common form of pooling is max pooling, which involves taking the maximum value within a small window. This helps to retain the most salient features while discarding irrelevant details.

The process of convolution and pooling is repeated multiple times to extract increasingly complex features from the image. The initial layers of a CNN typically detect simple features like edges and lines, while deeper layers can recognize more complex patterns like shapes and objects.

It's important to note that CNNs are trained using a large dataset of labeled images. During training, the model learns to automatically adjust the weights of the filters to optimize the classification accuracy. This process is known as backpropagation.

In our example, we will be using a pre-built dataset of dog and cat images. The images will be converted to pixel data, and a convolutional window will be applied to extract features. The resulting feature maps will then undergo pooling to reduce dimensionality. This process of convolution and pooling will be repeated multiple times to extract relevant features from the images.

By training the CNN on this dataset, we aim to teach the model to accurately classify images as either dogs or cats. The model will learn to recognize patterns and features that are indicative of each class, enabling it to make accurate predictions on new, unseen images.

Convolutional neural networks are a powerful tool for image classification tasks. By leveraging the concepts of convolution and pooling, CNNs can effectively extract features from images and learn to classify them accurately. In the next part of this tutorial series, we will delve deeper into the implementation details of CNNs using Python, TensorFlow, and Keras.

Convolutional neural networks (CNNs) are a type of deep learning model commonly used for image classification tasks. In this didactic material, we will introduce the basic concepts of CNNs and demonstrate how to build a simple CNN using Python, TensorFlow, and Keras.

To begin, we need to import the necessary libraries. We will import TensorFlow as TF and the required modules from TensorFlow and Keras. Specifically, we will import Sequential from TensorFlow.keras.models, Dense, Dropout, Activation, Flatten, Conv2D, and MaxPooling2D from TensorFlow.keras.layers. Additionally, we will import the pickle module for data loading.

After importing the necessary modules, we will load the data using pickle. The data consists of images and their corresponding labels. To prepare the data for training, we will normalize the pixel values. Since the pixel values range from 0 to 255, we can easily normalize the data by dividing it by 255.

Next, we will build our CNN model. We will start by creating a Sequential model using the syntax model = Sequential(). This model allows us to stack layers sequentially.

The first layer we will add is a Conv2D layer. This layer performs convolutional filtering on the input data. We can specify the number of filters, the filter size, and the input shape. In our case, we will use 64 filters with a filter size of 3x3 and an input shape of (50, 50, 1).

Following the Conv2D layer, we will add an Activation layer. The activation function we will use is the rectified linear activation function (ReLU). This function introduces non-linearity into the model.

After the Activation layer, we will add a MaxPooling2D layer. This layer performs max pooling, which reduces the spatial dimensions of the input data. In our case, we will use a pooling size of 2x2.

We can repeat the above steps to add more convolutional layers to our model. Each additional Conv2D layer will have the same structure as the first one, but without the need to specify the input shape.

Once we have added all the convolutional layers, we need to flatten the data before passing it to the fully connected layers. We can achieve this by adding a Flatten layer.

Finally, we will add a Dense layer as the output layer. The number of nodes in this layer will depend on the number of classes in our classification task. In our case, we will use 64 nodes. We will also add an Activation layer to introduce non-linearity to the output.

With the model built, we can now compile and train it using the appropriate optimizer and loss function. However, these steps are beyond the scope of this didactic material.

We have introduced the basic concepts of convolutional neural networks (CNNs) and demonstrated how to build a simple CNN using Python, TensorFlow, and Keras. CNNs are powerful deep learning models commonly used for image classification tasks. By stacking convolutional, activation, and pooling layers, we can effectively extract features from images and make accurate predictions.

Convolutional neural networks (CNNs) are a type of deep learning model commonly used in image recognition and computer vision tasks. In this tutorial, we will provide an introduction to CNNs and discuss their basic structure and components.

CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. These layers work together to extract and learn features from input images. The convolutional layers apply filters to the input image, which helps to detect different patterns and features. The pooling layers downsample the output of the convolutional layers, reducing the spatial dimensions of the input. Finally, the fully connected layers are responsible for making predictions based on the learned features.

To build a CNN model using Python, TensorFlow, and Keras, we first need to define the architecture of the model. This involves specifying the number and type of layers, as well as the parameters for each layer. In this tutorial, we will focus on a simple CNN architecture with one convolutional layer.

To define the model architecture, we use the Keras Sequential model. We start by creating an instance of the Sequential class:

```
1. model = Sequential()
```

Next, we add the convolutional layer using the `Conv2D` function. This layer applies a specified number of filters to the input image. We can also specify the size of the filters and the activation function to be used. For example, to add a convolutional layer with 64 filters, a filter size of 3x3, and a ReLU activation function, we can use the following code:

```
1. model.add(Conv2D(64, (3, 3), activation='relu'))
```

After adding the convolutional layer, we can add more layers to the model, such as pooling layers or additional convolutional layers. These layers help to further extract and learn features from the input image.

Once the model architecture is defined, we need to compile the model. This involves specifying the loss function, optimizer, and metrics to be used during training. For example, to use the categorical cross-entropy loss function, the Adam optimizer, and accuracy as the metric, we can use the following code:

```
1. model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'
```

```
])
```

After compiling the model, we can train the model using the `fit` function. This function takes the input data and labels, as well as other parameters such as the batch size and number of epochs. For example, to train the model with a batch size of 32 and 10 epochs, we can use the following code:

```
1. model.fit(X, Y, batch_size=32, epochs=10)
```

During training, the model will iterate over the input data in batches, adjusting the model's parameters to minimize the loss function. The number of epochs determines how many times the model will iterate over the entire dataset.

Finally, we can evaluate the performance of the trained model using the `evaluate` function. This function takes the test data and labels and returns the loss and accuracy of the model on the test set. For example, to evaluate the model on the test set, we can use the following code:

```
1. loss, accuracy = model.evaluate(X_test, Y_test)
```

In this tutorial, we have provided an introduction to convolutional neural networks (CNNs) and discussed the basic steps involved in building and training a CNN model using Python, TensorFlow, and Keras. We have also briefly mentioned the importance of evaluating the model's performance and discussed some common patterns and red flags to watch out for during training.

Convolutional neural networks (CNN) are a powerful technique used in the field of Artificial Intelligence (AI) for image recognition and processing. In this tutorial, we will introduce the concept of CNN and its importance in deep learning with Python, TensorFlow, and Keras.

CNNs are specifically designed to process visual data, making them highly effective in tasks such as object detection, image classification, and facial recognition. They are inspired by the human visual system, which is known for its ability to recognize patterns and features in images.

One key component of CNNs is the convolutional layer. This layer applies a set of filters to the input image, extracting relevant features by performing convolution operations. These filters are learned through the training process and are responsible for detecting edges, corners, and other visual patterns.

Another important component of CNNs is the pooling layer. This layer reduces the spatial dimensions of the input by downsampling, which helps in reducing the computational complexity of the network. Common pooling techniques include max pooling and average pooling.

The fully connected layer is the last layer in a CNN, responsible for making predictions based on the extracted features. This layer takes the output from the previous layers and applies a set of weights to produce the final prediction.

To implement CNNs in Python, we can utilize popular libraries such as TensorFlow and Keras. TensorFlow provides a powerful framework for building and training neural networks, while Keras offers a high-level API that simplifies the process of constructing CNN architectures.

In the next tutorial, we will delve into the topic of TensorBoard, a tool that is essential for training and analyzing CNN models. TensorBoard provides visualizations and metrics that help in understanding the behavior and performance of the network during training.

By understanding the fundamentals of CNNs and utilizing tools like TensorFlow, Keras, and TensorBoard, we can unlock the potential of deep learning for image-related tasks. Stay tuned for the next tutorial, where we will explore the capabilities of TensorBoard in more detail.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: TENSORBOARD
TOPIC: ANALYZING MODELS WITH TENSORBOARD

TensorBoard is a powerful tool for analyzing and optimizing deep learning models in Python using TensorFlow and Keras. It allows us to visualize the training process of our models over time, making it easier to understand and improve their performance.

One of the main uses of TensorBoard is to monitor metrics such as accuracy and loss during training. By visualizing these metrics, we can identify patterns and trends that can help us make informed decisions about our models.

To use TensorBoard, we need to import the necessary libraries. In this case, we import the TensorBoard callback from the Keras library. This callback allows us to save the necessary logs for visualization in TensorBoard.

Next, we need to define our model and give it a useful name. This is important because we may train multiple models and it's helpful to be able to identify them easily. For example, we can name our model "2 times 64 confident".

Once we have our model defined, we can add the TensorBoard callback to our training process. This callback will save the necessary logs for visualization. In this example, we only specify the log directory, but there are other parameters that can be customized depending on our needs.

After adding the TensorBoard callback, we can start training our model. As the training progresses, TensorBoard will update the logs and we can visualize the metrics in real-time. We can see how accuracy and loss change over time, and use this information to make adjustments and improvements to our model.

In addition to monitoring metrics, TensorBoard also offers other useful features. For example, we can use the ModelCheckpoint callback to save the best model based on certain criteria, such as the best validation accuracy or the lowest loss. This can be helpful when we want to avoid overtraining our model.

TensorBoard is a valuable tool for analyzing and optimizing deep learning models. By visualizing the training process and monitoring metrics, we can gain insights into our models and make informed decisions to improve their performance.

In this didactic material, we will discuss how to analyze models using TensorBoard in the context of Artificial Intelligence, specifically Deep Learning with Python, TensorFlow, and Keras.

When working with models, it is important to assign a unique name to each model. This can be achieved by including a timestamp in the name, ensuring that each model is distinct. By doing this, we prevent overwriting or appending models with the same name, which can lead to confusion in the analysis.

To specify the callback object for TensorBoard, we use the TensorBoard object. This object allows us to define where the log directory should be located. We can use the name of the model as part of the log directory path. Additionally, it is worth noting that customizing or creating our own callbacks is possible, providing flexibility in the analysis process.

To pass the TensorBoard callback into the fitment, we use the "callbacks" parameter. This parameter takes in a list of callbacks, and in our case, we have only one callback, which is the TensorBoard callback.

Once the model is trained, a new directory called "logs" will be created. This directory contains the model's information. To view the model in real-time, we need to open the command window (terminal or command prompt) and navigate to the directory that houses the logs. From there, we can run the TensorBoard command by typing "tensorboard --logdir=logs" and accessing the provided address in a browser.

It is important to note that on Windows, specifying the log directory can be a bit finicky. However, by following the correct syntax and ensuring no quotes are used, the TensorBoard should run smoothly.

After successfully running TensorBoard, we can visualize the graphs of the model. This includes the in-sample accuracy, in-sample loss, out-of-sample accuracy, and out-of-sample loss. Ideally, we want to see an increase in accuracy and a decrease in loss for both in-sample and out-of-sample data. However, it is not uncommon to observe validation loss creeping back up after a certain number of epochs. This can be attributed to the model transitioning from generalizing to memorizing the input samples.

Analyzing models using TensorBoard is an essential step in the Deep Learning process. By assigning unique names to models and utilizing the TensorBoard callback, we can effectively visualize and evaluate the performance of our models.

When evaluating a model, it is important to focus on the validation loss. This metric provides valuable insights into the model's performance. In this tutorial, we will explore how to analyze models using TensorBoard.

To begin, we can experiment with different model configurations. For example, we can remove the dense layer from our model and train it for twenty epochs. However, it is essential to note that removing layers may impact the model's performance negatively. Additionally, it is recommended to update the model's name to reflect any changes made.

During the training process, we may encounter errors or crashes. In such cases, it is crucial to ensure that the recording and other processes are not affected. Restarting the training process and monitoring TensorBoard for updates can help us analyze the model's performance.

Analyzing the model's performance, we observe that without the dense layer, the model's accuracy is worse. However, the out-of-sample loss is better. This indicates that the model without the dense layer performs better on unseen data, which is more important than the in-sample performance. Overfitting is a concern when the in-sample performance is significantly better than the out-of-sample performance.

In TensorBoard, we can utilize various features to analyze models. For instance, we can toggle the visibility of individual runs or select multiple runs to compare their performance. Additionally, we can adjust the smoothing to visualize the data with different levels of smoothness. This helps us understand the trends and patterns in the model's performance.

Furthermore, we can filter and search for specific models based on criteria such as model architecture or hyperparameters. This enables us to efficiently navigate through a large number of models and focus on the ones that meet our requirements.

In the next tutorial, we will explore how to optimize models using automated processes. This approach simplifies the iterative process of modifying and retraining models. If you have any questions or concerns, please feel free to leave them in the comments below or join our Discord community at discord.gg/Centex.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: TENSORBOARD
TOPIC: OPTIMIZING WITH TENSORBOARD

TensorBoard is a powerful tool that can be used to optimize models in deep learning. By visualizing different attempts at models, we can identify areas for improvement and make changes accordingly. In this tutorial, we will explore how to optimize models using TensorBoard.

When optimizing a model, there are several aspects that we can test and tweak. Some of these include the optimizer, learning rate, number of dense layers, units per layer, activation units, kernel size, stride, decay rate, and more. With so many possible combinations, the number of models to test can quickly become overwhelming.

To simplify the optimization process, it is recommended to start with the most obvious changes. In this case, we will focus on tweaking the number of layers, nodes per layer, and whether or not to include a dense layer at the end. These changes can have a significant impact on the performance of the model.

To implement these changes, we can use Python code. We will define some choices for the number of dense layers (0, 1, or 2) and layer sizes (32, 64, 128). These choices are based on previous successful models. We will iterate through these choices to create different combinations of models.

Next, we need to assign names to each model combination. This can be done using a naming convention that includes the number of convolutional layers, dense layers, and layer sizes. Additionally, we can include a timestamp to differentiate between different runs.

Once we have all the model combinations and their respective names, we can proceed to apply these changes to our model. This involves modifying the code to include the desired changes for each model combination.

By utilizing TensorBoard, we can visualize the performance of each model and compare the results. This allows us to identify the best performing models and make further improvements if necessary.

TensorBoard is a valuable tool for optimizing models in deep learning. By systematically testing different combinations of model parameters, we can improve the performance of our models. Through visualization and analysis, we can make informed decisions on how to further optimize our models.

In this didactic material, we will discuss the process of optimizing deep learning models using TensorBoard, a powerful visualization tool provided by TensorFlow. TensorBoard allows us to track and analyze various aspects of our models, such as loss, accuracy, and computational graph visualization.

Before diving into the optimization process, it is important to understand the structure of our deep learning model. The model consists of convolutional layers (conv layers) and dense layers. The conv layers extract features from the input data, while the dense layers perform classification based on these extracted features.

To begin, we need to define the input shape for the first layer of our model. This is necessary to ensure compatibility between the input data and the model architecture. Additionally, the first dense layer must be flattened to accommodate the output of the conv layers.

To handle the conv layers, we iterate through a range of "com layer - 1" (the number of conv layers minus one), adding the necessary components for each conv layer. Similarly, we iterate through a range of "dense layer" to add the dense layers. It is important to note that the output layer is considered a dense layer, but we refer to the dense layers before the output layer.

In our example, we use a fixed size of 64 for the dense layers. However, in practice, it is common to use different sizes for the dense layers compared to the conv layers. This is because the conv layers extract features, while the dense layers perform classification, and these tasks have different requirements.

Once the model architecture is defined, we can proceed with the optimization process. It is recommended to adjust the size of the layers and experiment with different configurations to achieve optimal performance.

To run the code, it is necessary to have TensorFlow installed. For those using the CPU version, installation instructions can be found in the tutorial mentioned in the text-based version of this material. For GPU users, the installation process is similar, but requires additional steps, such as installing CUDA toolkit and extracting cuDNN into the CUDA toolkit.

To speed up the training process, it is also possible to run the code on a cloud-based platform like PaperSpace, which offers a variety of GPUs at competitive prices. A referral link to PaperSpace is provided in the description, offering a $10 credit to get started.

After running the code, we can visualize the training process using TensorBoard. By specifying the log directory, TensorBoard will generate interactive visualizations of our model's performance. These visualizations include metrics like loss and accuracy over time, as well as the computational graph that represents the model's structure.

Optimizing deep learning models involves defining the model architecture, adjusting layer sizes, and experimenting with different configurations. TensorBoard is a valuable tool for visualizing and analyzing the training process, allowing us to make informed decisions to improve model performance.

When working with Artificial Intelligence (AI) and specifically Deep Learning, it is important to optimize and fine-tune the models to achieve the best possible performance. In this didactic material, we will explore the use of TensorBoard, a powerful visualization tool that comes bundled with TensorFlow, to optimize our deep learning models.

TensorBoard allows us to visualize and analyze various aspects of our models, such as training and validation metrics, model architectures, and even the distribution of weights and biases. By using TensorBoard, we can gain insights into the behavior of our models and make informed decisions on how to improve them.

To begin, let's assume that we have already trained a deep learning model and have logged the training and validation metrics using TensorBoard. We can then load these logs into TensorBoard and start analyzing the results.

One common issue that we might encounter is the incorrect path to the log files. If the logs are not being loaded properly, we can try changing the path to the correct location. It is important to ensure that the log files are stored in the designated directory and that the path is correctly specified in the code.

Once the logs are successfully loaded, we can visualize various metrics such as validation loss and accuracy. These metrics provide valuable insights into the performance of our model. For example, we can identify the best performing models by examining the validation loss and accuracy curves. The models with the lowest validation loss and highest accuracy are usually considered the best.

In addition to the metrics, we can also analyze the model architecture. TensorBoard provides a graphical representation of the model, allowing us to visualize the layers, their connections, and the flow of data through the network. This visualization helps us understand the complexity and structure of our model.

Another useful feature of TensorBoard is the ability to compare different models. By running multiple experiments with different hyperparameters or architectures, we can compare their performance and identify the best configuration. TensorBoard allows us to overlay multiple curves on the same graph, making it easy to compare and analyze the results.

To optimize our model, we can experiment with different configurations, such as the number of convolutional layers, the number of nodes per layer, and the presence or absence of dense layers. By analyzing the results in TensorBoard, we can identify patterns and make informed decisions on which configurations yield the best performance.

It is important to note that overfitting can be a common issue in deep learning models. Overfitting occurs when the model performs well on the training data but fails to generalize to new, unseen data. To avoid overfitting, we should carefully monitor the validation metrics and ensure that the model is not memorizing the training data. If the validation metrics start to deteriorate while the training metrics continue to improve, it is a sign of

overfitting.

TensorBoard is a powerful tool that can greatly assist in optimizing deep learning models. By visualizing and analyzing various aspects of the model's performance, architecture, and metrics, we can make informed decisions on how to improve the model's performance. Experimenting with different configurations and monitoring validation metrics are crucial steps in optimizing deep learning models.

Deep learning is a powerful technique in the field of artificial intelligence that allows machines to learn and make predictions from large amounts of data. In this didactic material, we will discuss the use of Python, TensorFlow, and Keras for deep learning, specifically focusing on TensorBoard and optimizing with TensorBoard.

One important aspect of deep learning is the architecture of the neural network. In particular the use of dense and convolutional layers. Dense layers are fully connected layers where each neuron is connected to every neuron in the previous layer. Convolutional layers, on the other hand, are used for image processing tasks and are particularly effective in recognizing patterns in images.

Also the concept of validation loss should be mentioned. Validation loss is a metric used to evaluate the performance of a model during training. It represents the difference between the predicted and actual values on a validation dataset. It is important to monitor the validation loss to ensure that the model is not overfitting or underfitting the data.

The number of dense and convolutional layers can be adjusted based on the specific task and dataset. It should be stressed that adding a dense layer might help in memorizing information, but it is important to note that this may not be applicable to all datasets. The size of the model and the number of samples in the dataset play a crucial role in determining the effectiveness of the model.

The importance of trial and error in optimizing the model should also be highlighted. Making incremental changes and observing the impact on the model's performance is a common approach. This process can be time-consuming, especially when training large models.

Activation functions are another important aspect of deep learning. The rectified linear activation function (ReLU) is a good choice. The choice of activation function depends on the specific task and the type of data being processed.

Scaling the data should also be mentioned as an important step. Scaling ensures that all features have a similar range, which can improve the performance of the model.

In terms of tools, what should be stressed is the use of TensorBoard for visualizing and optimizing models. TensorBoard is a powerful tool that provides visualizations of various metrics, including loss, accuracy, and model architecture. It helps in understanding the behavior of the model and identifying areas for improvement.

Finally, it should be added that even professionals in the field of deep learning follow a similar trial and error approach. This highlights the iterative nature of model optimization and the importance of experimentation.

This didactic material has provided an overview of deep learning with Python, TensorFlow, and Keras. It has discussed the use of TensorBoard for model visualization and optimization. It has also highlighted the importance of architectural choices, trial and error, activation functions, data scaling, and the iterative nature of model optimization.

In deep learning with Python, TensorFlow, and Keras, TensorBoard is a powerful tool for optimizing models. It allows us to visualize and analyze various aspects of our model's performance. In this didactic material, we will explore the process of optimizing with TensorBoard.

When working with deep learning models, we start by defining our model and specifying the activation functions. These functions determine the output of each neuron in the network. Then, we move on to the training phase. During training, we compile the model, selecting an optimizer and providing the necessary data and target values. This is where the model learns from the data and adjusts its parameters to minimize the loss.

After training, we can test the model to evaluate its performance. This can be done using a command-line

interface or any other suitable method. However, it is important to note that the code used for testing in the provided material is not considered optimal or Pythonic.

This didactic material provides an overview of optimizing deep learning models using TensorBoard. It emphasizes the importance of defining the model, selecting appropriate activation functions, and utilizing the right optimizer during training. The material also highlights preference for higher-level APIs. Finally it explores the visualization capabilities of TensorBoard.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: TENSORBOARD
TOPIC: USING TRAINED MODEL

Deep learning models are trained using large amounts of data to make accurate predictions. In this context, the concept of a "Data saver variable" is mentioned. This variable is used to store external images that are utilized for making predictions. The process involves training the model on a dataset and then evaluating its performance on these external images.

To train a deep learning model, it is crucial to have a diverse and representative dataset. This dataset should contain a wide range of examples that the model will learn from. Once the model is trained, it can be used to make predictions on new, unseen data.

The "Data saver variable" mentioned in the transcript refers to a mechanism for storing these external images. This variable allows the model to access and use these images for prediction purposes. By using this variable, the model can utilize the information contained in these external images to make accurate predictions.

After training the model and storing the external images in the "Data saver variable," the next step is to make predictions on these images. The trained model can be used to process these images and generate predictions based on the learned patterns and features. This process allows the model to generalize its knowledge and make predictions on new, unseen data.

The "Data saver variable" is a mechanism used in deep learning to store external images that are used for prediction purposes. By training the model on a dataset and evaluating its performance on these external images, the model can make accurate predictions on new, unseen data.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: RECURRENT NEURAL NETWORKS
TOPIC: INTRODUCTION TO RECURRENT NEURAL NETWORKS (RNN)

A recurrent neural network (RNN) is a type of artificial neural network that is capable of processing sequential data, where the order of the data is important. This is particularly useful for analyzing time series data or natural language processing tasks, where the meaning of a sentence depends on the order of the words. In this didactic material, we will explore the basic concepts of RNNs and how they work.

RNNs consist of recurrent cells, which are responsible for processing sequential data. There are different types of recurrent cells, such as the basic recurrent cell, long short-term memory (LSTM) cell, and gated recurrent unit (GRU). However, LSTMs are commonly used due to their effectiveness. These cells receive sequential data as input and output to the next layer or the next node in the recurrent layer.

To understand how an LSTM cell works, let's consider a simple example. Imagine a green box representing a cell in the recurrent layer. The data from the previous cell is passed to this cell, and it performs three main operations: forget, input, and output. First, it decides what information to forget from the previous node. Then, it takes input data and determines what information to add to the current bundle of information. Finally, based on this bundle of information, it decides what to output to the next layer and the next node.

RNNs can be unidirectional or bidirectional. In unidirectional RNNs, the data flows in one direction, while in bidirectional RNNs, the data flows in both directions. This allows for more complex patterns and relationships to be captured.

Implementing an RNN involves structuring the dataset appropriately, as sequential data typically lacks explicit targets. Pre-processing the data is often necessary. In this didactic material, we will use a simple example to demonstrate the basics of building an RNN. However, keep in mind that the challenging part of working with RNNs lies in preparing the dataset.

Now, let's start by importing the necessary libraries. We will import TensorFlow as tf and the Sequential module from the Keras models.

```
1. import tensorflow as tf
2. from tensorflow.keras.models import Sequential
```

Please note that the above code assumes that TensorFlow is already installed. If not, you can install it using the appropriate method.

In the next tutorial, we will dive into a realistic example using time series data, specifically crypto currency prices. This will provide a more practical understanding of how RNNs can be applied.

In this didactic material, we will introduce the concept of Recurrent Neural Networks (RNN) in the context of deep learning with Python, TensorFlow, and Keras.

To begin, we need to import the necessary modules from TensorFlow and Keras. Specifically, we will import the Dense layer, Dropout, and LSTM cell. The Dense layer is used for the output layer, and it is common to include a dense layer before the output layer as well. Dropout is used to prevent overfitting, and the LSTM cell is a type of recurrent neural network cell that we will be using. If you are using the GPU version of TensorFlow, you may also want to consider using the KUDNNLSTM cell, which is optimized for GPU and can provide faster performance.

Next, we need a dataset to train our model. For this example, we will use the MNIST dataset, which is a collection of handwritten digits. We can easily load the dataset using the TF.keras.datasets.mnist function and unpack it into training and testing data.

Now that we have the necessary modules and dataset, we can start building our model. We will use a sequential model, which is a linear stack of layers. First, we add an LSTM layer with 128 cells. We specify the input shape as 28 by 28, which corresponds to the dimensions of the MNIST images. We use the rectified linear activation function for this layer. Additionally, we set the return_sequences parameter to True, indicating that we want this

layer to return sequences.

After the LSTM layer, we add a dropout layer with a dropout rate of 20%. This helps prevent overfitting by randomly dropping out some connections during training. Then, we add another LSTM layer with 128 cells, using the same activation function and dropout rate.

Next, we add a dense layer with 32 nodes and the rectified linear activation function. Finally, we add the output layer, which is a dense layer with 10 nodes (corresponding to the 10 possible digits). We do not include dropout for this layer.

To summarize, our model consists of two LSTM layers, each with 128 cells, followed by a dropout layer, another LSTM layer, a dense layer with 32 nodes, and the output layer with 10 nodes.

Now that we have built our model, we can feed the training data through it. The training data is already in sequences, with each sequence representing a row of pixels from the MNIST images. The hope is that the recurrent neural network can learn to recognize patterns in these sequences and accurately classify the digits.

We have introduced the concept of Recurrent Neural Networks (RNN) and demonstrated how to build a basic RNN model using Python, TensorFlow, and Keras. This model is capable of processing sequences of data, making it suitable for tasks such as image recognition. In the next parts, we will explore more advanced topics related to RNNs.

In this didactic material, we will introduce the concept of Recurrent Neural Networks (RNN) in the context of deep learning with Python, TensorFlow, and Keras. RNNs are a type of artificial neural network that are particularly well-suited for processing sequential data, such as time series or natural language.

To begin, let's discuss the structure of an RNN model. RNN models consist of multiple recurrent layers, which allow the network to retain information from previous inputs. Each recurrent layer contains a number of units, also known as cells or memory blocks. These units are responsible for processing the sequential data and maintaining a hidden state that carries information from previous time steps.

When building an RNN model, it is important to choose an appropriate activation function. In this case, we will use the softmax activation function, which is suitable for multi-class classification problems. The softmax function outputs a probability distribution over the different classes, allowing us to make predictions.

After defining the model structure, we need to compile the model. To do this, we use the Adam optimizer from TensorFlow and specify a learning rate of 1e-3. Additionally, we can introduce a decay factor to gradually reduce the learning rate over time. This can help the model converge to a better solution by taking smaller steps as it progresses.

Next, we specify the loss function to be used for training the model. In this case, we will use the sparse categorical cross-entropy loss, which is commonly used for multi-class classification problems. We can also define the metrics that we want to track during training, such as accuracy.

Once the model is compiled, we can train it using the fit() function. We provide the training data (X_train and Y_train) and specify the number of epochs, which determines how many times the model will iterate over the entire training dataset. We can also specify a validation dataset (X_test and Y_test) to monitor the model's performance during training.

It is important to note that preprocessing the data can have a significant impact on the model's performance. In this case, we can normalize the input data by dividing it by 255, which scales the values between 0 and 1. This normalization step can help the model converge faster and improve its accuracy.

Additionally, we can explore other variations of RNNs, such as the CuDNNLSTM layer, which is optimized for GPU acceleration. This layer uses the hyperbolic tangent (tanh) activation function and can provide faster training times compared to the standard LSTM layer.

This didactic material provided an introduction to Recurrent Neural Networks (RNN) in the context of deep learning with Python, TensorFlow, and Keras. We discussed the structure of an RNN model, the choice of

activation function, the compilation and training process, and the importance of data preprocessing. We also explored alternative RNN layers, such as the CuDNNLSTM layer. By understanding these concepts, you can begin to apply RNNs to various sequential data tasks.

Recurrent neural networks (RNNs) are a powerful type of artificial neural network that can effectively process sequential data. In this didactic material, we will introduce the concept of RNNs and their application in deep learning using Python, TensorFlow, and Keras.

RNNs are particularly useful when dealing with data that has a temporal or sequential nature, such as time series data or natural language. Unlike traditional feedforward neural networks, RNNs have a feedback connection that allows them to retain information from previous steps in the sequence. This makes them well-suited for tasks that require memory or context, such as language translation or speech recognition.

One key advantage of RNNs is their ability to handle variable-length input sequences. This is achieved through the use of recurrent connections, which allow information to flow from one step to the next. By maintaining an internal state, RNNs can capture long-term dependencies in the data and make predictions based on the entire sequence.

Let's discuss the accuracy achieved by the RNN model. Accuracy is a common metric used to evaluate the performance of a classification model. It represents the percentage of correctly classified samples out of the total number of samples. In this case, the model achieved an accuracy of 96% after a certain number of epochs.

Let's now considers the difference between accuracy and validation accuracy. Validation accuracy refers to the accuracy of the model on a separate validation dataset, which is used to assess the generalization ability of the model. It is common for the validation accuracy to be higher than the accuracy on the training dataset, as the model has not seen the validation data during training.

The validation accuracy is usually higher than the accuracy at the end of each epoch. This can be attributed to the fact that the accuracy is computed at the end of each epoch, while the validation accuracy is an average over the entire epoch. It is worth noting that the validation accuracy can fluctuate during training, and it is important to monitor it to avoid overfitting.

The consideration highlights the importance of continuing training if the validation accuracy is higher than the accuracy. This suggests that the model could potentially benefit from further training to improve its performance. However, it is crucial to strike a balance and avoid overfitting the model to the training data.

In the next topic a more complex example will be covered, involving a realistic time series dataset. This highlights the versatility of RNNs in handling various types of sequential data. The pre-processing steps were not discussed in detail in the current material, as the focus was on the implementation of RNNs.

RNNs are a powerful tool in the field of deep learning, particularly for tasks involving sequential data. They allow for the processing of variable-length input sequences and can capture long-term dependencies. Monitoring accuracy and validation accuracy is important during training to assess model performance and avoid overfitting. In the next topic, a more complex example involving a realistic time series dataset will be explored.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: RECURRENT NEURAL NETWORKS
TOPIC: INTRODUCTION TO CRYPTOCURRENCY-PREDICTING RNN

In this didactic material, we will be discussing how to apply recurrent neural networks (RNNs) to a realistic example of predicting cryptocurrency prices using Python, TensorFlow, and Keras. RNNs are a type of artificial intelligence algorithm that can process sequential data by retaining information from previous steps.

In this case, we will be working with a time series dataset consisting of prices and volumes for various cryptocurrencies. However, the concepts we cover can be applied to other types of sequential data as well. Our goal is to use RNNs to track the price sequences of four major cryptocurrencies: Bitcoin, Litecoin, Ethereum, and Bitcoin Cash.

To achieve this, we will take the last 60 minutes of price and volume data for each cryptocurrency and use it to predict the future price of Litecoin. We will create sequences of this data, updating it every minute, and attempt to predict whether the price of Litecoin will rise or fall three minutes into the future.

This example can be generalized to other applications, such as predicting server overheating or website traffic based on sensor data and time of day. The ultimate objective is to either classify the data (e.g., predicting price movement) or perform regression analysis (e.g., predicting price or percentage change).

Working with sequential data presents several challenges. First, we need to preprocess the data by building sequences and normalizing it. Since the prices and volumes of different cryptocurrencies vary, we need to ensure that the data is in relative terms. Additionally, we need to scale the data, which is more complex than simply dividing by 255 as in image data.

Furthermore, working with sequential data requires us to address the challenge of out-of-sample prediction. This means predicting data points that fall outside the training set, which requires additional considerations.

To get started, we have provided a dataset that you can download from the link provided in the description. The dataset consists of four files, each containing the price data for one of the cryptocurrencies. The data includes UNIX timestamps, low/high/open/close prices, and volume information. For our purposes, we will focus on the closing price and volume columns.

To begin, we will read in the data and ensure that it is formatted correctly. Once the data is loaded, we can proceed with the necessary preprocessing steps.

We need to import the pandas library. If you don't have it installed, you can open a terminal or command prompt and use the command "pip install pandas". We will be using pandas to work with our data.

Next, we will read in our data from a CSV file called "crypto_data/LTC_USD.csv". The columns in the CSV file are not named, so we will specify the column names as "time", "low", "high", "open", and "volume". We will use the pandas function "read_csv" to read in the data and store it in a DataFrame called "DF".

Now, let's print the first few rows of the DataFrame using the "head" function to see what the data looks like.

We have successfully read in the data and can now analyze it. We are primarily interested in the "close" and "volume" columns. However, we have multiple CSV files with similar data, and we want to combine them based on the common index, which is the "time" column.

To achieve this, we will create an empty DataFrame called "main_DF" using the pandas function "DataFrame". We will then iterate over a list of file names that we intend to use, which are "BTC_USD.csv", "LTC_USD.csv", "ethereum_USD.csv", and "Bitcoin_cash_BCH_USD.csv".

For each file, we will read in the data using the pandas function "read_csv" and store it in a DataFrame called "DF". We will rename the "close" and "volume" columns to include the ratio in the column name, such as "BTC_close" and "BTC_volume", using f-strings. This will help us identify the data from each file when we merge them.

Next, we will set the index of the DataFrame to be the "time" column using the pandas function "set_index". This will ensure that all the DataFrames have the same index, allowing us to merge them.

Finally, we will merge all the DataFrames into the "main_DF" DataFrame using the pandas function "merge". This will combine the data based on the common index.

In this didactic material, we will introduce the concept of Recurrent Neural Networks (RNNs) and their application in predicting cryptocurrency prices. Specifically, we will use Python, TensorFlow, and Keras to implement an RNN model for predicting the future price of Litecoin (LTC/USD) based on historical price and volume data.

To start, we need to prepare the data for training our model. We will create a DataFrame containing two columns: "close" (price) and "volume". By printing the DataFrame, we can ensure that the data is correctly loaded.

Next, we will merge the columns of the DataFrame to prepare the sequential data. If the main DataFrame is empty, we assign it the value of the DataFrame. Otherwise, we join the main DataFrame with the DataFrame. By printing the head of the main DataFrame, we can verify that the columns are merged correctly.

Now, let's discuss the requirements for training a supervised machine learning model. For any supervised machine learning problem, we need both the sequences and the targets. In our case, the sequences are the historical price and volume data, and the target is the future price of Litecoin. We will define some constants, including the sequence length, the future period to predict, and the ratio to predict. For example, we can set the sequence length to 60 minutes and predict the future price for the next three minutes.

To determine the targets, we will define a classification rule. If the future price is higher than the current price, we assign a label of 1 (indicating a good buying opportunity). Otherwise, we assign a label of 0. By applying this classification rule, we can train the model to learn the relationship between the sequence of features and the future price.

To calculate the future price, we will use the "close" column of the main DataFrame. We shift this column by the negative value of the future period to predict. By printing the future price, we can verify that it is correctly calculated.

We have discussed the process of preparing the data for training an RNN model to predict the future price of Litecoin. We have merged the necessary columns, defined the sequence length and future period to predict, and calculated the future price based on the classification rule. With this information, we are ready to proceed with training the RNN model.

In this topic, we will continue our exploration of cryptocurrency-predicting recurrent neural networks (RNNs). We have already obtained the future price of the cryptocurrency based on the current price, and now we want to map this function to a new column called "target". To do this, we will convert the output to a list and assign it as a column in our main DataFrame.

To begin, we need to convert the output to a list using the "list" function. Then, we will use the "map" function to apply our classify function to each pair of current and future prices. In this case, the current column is represented by "main DF close" and the future column is represented by "main DF future". We will assign the result to the "target" column.

After applying the function, we can check if the mapping worked as expected. We can observe the current price and the price in three periods in the future. If the future price is less than the current price, the target will be zero. Otherwise, it will be one.

Next, let's print the first ten values of the "target" column using the "head" function. This will allow us to verify if the mapping was successful. We can compare the current price with the price in three periods in the future to determine if the target is correct.

At this point, everything looks great, and we are ready to move on to the next steps. However, it is important to

note that there is still a lot of work ahead of us. We need to build sequences, balance the data, normalize the data (as the prices and volumes of cryptocurrencies can vary significantly), and scale the data. Additionally, we may need to consider out-of-sample testing and other tasks that we might have overlooked.

In the next topic, we will continue with these remaining steps. If you have any questions, comments, or concerns, please feel free to leave them below. If you notice any errors in the code or have any suggestions, your feedback is greatly appreciated. We would like to thank our recent sponsors, including Billy, Harsh, Soft, Mark Zuckerberg, JT, and Newcastle Geek. We appreciate your support.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: RECURRENT NEURAL NETWORKS
TOPIC: NORMALIZING AND CREATING SEQUENCES CRYPTO RNN

In this deep learning topic, we will continue working on our project of implementing a recurrent neural network (RNN) to predict the future price movements of a cryptocurrency. Our goal is to use the sequences of the cryptocurrency's prices and volumes, along with the prices and volumes of three other cryptocurrencies, to make accurate predictions.

So far, we have obtained the necessary data and merged it together. Now, we need to create sequences from this data and perform various preprocessing steps such as normalization and scaling. However, before we proceed with these steps, we need to address the issue of out-of-sample testing.

In the case of temporal data, time series data, and other sequential data, simply shuffling the data and randomly selecting a portion as the out-of-sample set is not appropriate. This is because the sequences in the out-of-sample set would still be very similar to those in the in-sample set, making it easy for the model to overfit.

Instead, we need to separate a chunk of data as our out-of-sample set. For time series data, it is recommended to use a chunk of data from the future as the out-of-sample set. In our case, we will take the last 5% of the historical data and set it aside as our out-of-sample data. This approach is similar to building the model 5% of the time ago and forward testing it.

It is important to note that many people overlook or mishandle out-of-sample testing, especially in finance and time series data analysis. Some may not perform out-of-sample testing at all, leading to overfitting. Others may use incorrect methods for out-of-sample testing. Therefore, it is crucial to keep this in mind and follow the proper procedure.

To implement the separation of the last 5% of the data as our out-of-sample set, we will first sort the data in ascending order. Then, we will find the index value that corresponds to the threshold of the last 5% of the data. This threshold index value can be calculated as 0.05 times the length of the data. Finally, we will separate the out-of-sample data by selecting the rows with index values greater than the threshold.

Once we have separated the out-of-sample data, we can proceed with the remaining preprocessing steps. It is important to note that while we will not address the issues related to normalization and scaling at this point, ensuring the correct separation of the out-of-sample data is of utmost importance.

In this topic, we have discussed the importance of out-of-sample testing for sequential data analysis. We have explained the recommended approach of separating a chunk of data from the future as the out-of-sample set. We have also emphasized the significance of correctly implementing this separation to avoid overfitting and ensure accurate testing.

In this didactic material, we will discuss the process of normalizing and creating sequences for a recurrent neural network (RNN) using Python, TensorFlow, and Keras. Specifically, we will focus on applying these techniques to cryptocurrency data.

To begin, we need to split our data into training and validation sets. We will use the last 5% of the data as the validation set. This can be achieved by setting a threshold value and selecting all data points with a timestamp greater than or equal to this threshold for validation. The remaining data points will be used for training.

Once the data is split, we can proceed to create sequences for our RNN. In order to do this, we need to perform several steps including balancing, scaling, and potentially other preprocessing tasks. To simplify this process, we can create a function called "pre_process_DF" that will handle all these tasks for both the training and validation sets.

The "pre_process_DF" function takes a DataFrame as input and performs the necessary preprocessing steps. First, we import the "preprocessing" module from the scikit-learn library. If you don't have this module installed, you can use the command "pip install scikit-learn" to install it.

Next, we drop the "future" column from the DataFrame as it is no longer needed. This column was used to generate the target variable and including it in the training process would lead to overfitting.

After dropping the "future" column, we iterate over the remaining columns and scale them using the percent change method. This normalization step ensures that the data is comparable across different cryptocurrencies and avoids bias towards larger values. We then drop any NaN values that may have been generated during the scaling process.

Finally, we use the "scale" function from the "preprocessing" module to scale the values of each column in the DataFrame. This ensures that the values fall within a certain range, typically between 0 and 1, which is desirable for neural network training.

By applying the "pre_process_DF" function to both the training and validation sets, we can obtain the preprocessed data required for training our RNN. The resulting data will be stored in the variables "train_X", "train_y", "validation_X", and "validation_y" respectively.

We have discussed the process of normalizing and creating sequences for an RNN using Python, TensorFlow, and Keras. We have seen how to split the data into training and validation sets, as well as how to preprocess the data using the "pre_process_DF" function. These steps are crucial for preparing the data for training an RNN model.

To normalize and create sequences for a recurrent neural network (RNN) in the context of deep learning with Python, TensorFlow, and Keras, we follow a specific process. The goal is to scale the data between 0 and 1 and create sequences of a certain length.

First, we start by scaling the data. We use a simple formula to scale the data to be between 0 and 1. This can be done by dividing each value by the minimum-maximum range of the data.

Next, we handle any missing or invalid values. If the percent change column contains NaN (not a number) values, we drop those rows. Additionally, if the sequence creation process generates NaN values, we also drop those rows.

To create sequences, we initialize an empty list called "sequential_data" and another variable called "prev_days" to store the previous days' data. We set the maximum length of the sequence, which is typically 60.

To handle the sequence creation logic, we import the "deque" class from the "collections" module. The "deque" class allows us to create a list-like structure with a maximum length. As new items are added, old items are automatically removed.

We wait until "prev_days" has at least 60 values before starting to populate the sequence. For each row in the dataset, we iterate over the columns and append the values to "prev_days". However, we exclude the target column from the sequence.

Once "prev_days" has reached the desired length, we append the features and labels to the "sequential_data" list. The features are the previous 60 days' data, and the label is the current value.

Finally, we shuffle the "sequential_data" list randomly to ensure that the data is not biased. This is done using the "random.shuffle" function.

It's important to note that this didactic material assumes prior knowledge of Python programming, deep learning concepts, and the basics of TensorFlow and Keras libraries.

After preparing the sequences and targets, we are now ready to feed them through a model. However, there are still a few more steps we need to complete before training the model. In the next topic, we will continue working towards our goal, although it is unlikely that we will train the model in full.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: RECURRENT NEURAL NETWORKS
TOPIC: BALANCING RNN SEQUENCE DATA

In this didactic material, we will discuss the process of balancing RNN sequence data in the context of building a recurrent neural network for predicting cryptocurrency price movements. Balancing the data is an important step to ensure that the model does not waste time and gets stuck in a rut.

Before we balance the data, we need to pre-process it. In the pre-processing step, we normalize and scale the data. Once the data is pre-processed, we proceed to balance it. Balancing the data means having an equal number of buys and sells. If the data is already balanced or has a slight imbalance, there is no need to worry. However, if the data has a significant imbalance, it is crucial to balance it.

We can pass class weights to Keras, which allows us to assign different weights to different classes. However, in practice, this method does not always solve the balancing issue effectively. Therefore, it is recommended to manually balance the data.

To balance the data, we create two lists: "buys" and "sells." We iterate over the sequential data and append the sequence targets to either the "buys" or "sells" list based on their value. After that, we shuffle both lists to ensure randomness.

Next, we determine the minimum length between the "buys" and "sells" lists. We assign this minimum length to the variable "lower." Then, we slice the "buys" and "sells" lists up to the "lower" length to ensure both lists have the same length.

Once the data is balanced, we combine the "buys" and "sells" lists into the "sequential data" list. We shuffle the "sequential data" list to avoid having all buys followed by all sells, which could confuse the model.

Now that the data is balanced and shuffled, we need to split it into input (X) and output (Y) lists. We iterate over the "sequential data" list and append the sequences to the "X" list and the targets to the "Y" list.

Finally, we return the numpy arrays of "X" and "Y" as the pre-processed data. At this point, the pre-processing data frame function is complete, and we are ready to proceed with further steps.

To summarize, balancing RNN sequence data is essential to ensure that the model does not waste time and get stuck in a rut. By balancing the data, we create an equal number of buys and sells. This can be achieved by creating separate lists for buys and sells, shuffling them, and then slicing them to have the same length. Additionally, we split the balanced data into input (X) and output (Y) lists.

The training data consists of 69,000 samples, while the validation data has 3,000 samples. It is important to note that there is a balanced distribution between the "buys" and "don't buys" in both the training and validation sets. This balance ensures that the model is trained on a representative dataset.

Moving forward, the next step is to build and train the model. This will be covered in the upcoming topic.

EITC/AI/DLPTFK DEEP LEARNING WITH PYTHON, TENSORFLOW AND KERAS DIDACTIC MATERIALS
LESSON: RECURRENT NEURAL NETWORKS
TOPIC: CRYPTOCURRENCY-PREDICTING RNN MODEL

In this topic, we will continue our exploration of deep learning with Python, TensorFlow, and Keras by building a recurrent neural network (RNN) to predict future price movements of a cryptocurrency. This RNN will be trained using historical price and volume data of the cryptocurrency, as well as data from other major cryptocurrencies.

To begin, we need to import the necessary libraries. We will import TensorFlow as TF and Keras' Sequential model from tensorflow.keras.models. Additionally, we will import various layers and callbacks from tensorflow.keras.layers and tensorflow.keras.callbacks, respectively. These include Dense, Dropout, LSTM, and BatchNormalization.

Next, we define some constants. The first constant is "epochs," which determines the number of training epochs for our model. We will start with a value of 64, but this can be adjusted later. The second constant is "batch_size," which determines the number of samples per gradient update. We will start with a value of 64 as well. Lastly, we define a name for our model using an F-string. This name should be descriptive and unique, allowing for easy comparison of different models.

With the necessary libraries imported and constants defined, we can now proceed to build our model. We create a Sequential model using "model = tf.keras.models.Sequential()". We then add layers to our model using "model.add()". The first layer we add is an LSTM layer with 128 nodes. We set "return_sequences=True" to indicate that the next layer will also be an LSTM layer. This ensures that the output of each LSTM layer is fed as input to the next LSTM layer.

After the LSTM layer, we add a Dropout layer with a dropout rate of 0.2 to prevent overfitting. We then add a BatchNormalization layer to normalize the data between layers. Batch normalization is useful for ensuring that the input data to each layer is normalized, improving the overall performance of the model.

Once we have added all the layers to our model, we can proceed to train it. We will use the "fit()" function to train our model using the training data. We will also use the "evaluate()" function to evaluate the model's performance on the validation data. Additionally, we can use callbacks such as TensorBoard and ModelCheckpoint to save checkpoints of the model during training.

By following these steps, we can build a recurrent neural network model using Python, TensorFlow, and Keras to predict future price movements of a cryptocurrency based on historical data.

Batch normalization is a technique used in deep learning to normalize the inputs of each layer, which helps in improving the performance and stability of the model. In the given code snippet, batch normalization is applied before the dense layer. This technique does not have any parameters.

The code snippet also includes the addition of a dense layer with 128 units. The return sequences parameter is removed, indicating that this layer is not a recurrent layer. Then, another dense layer with 128 units is added, followed by a final dense layer with 128 units. The dropout parameter is set to 0.1, although 0.2 might also work well.

A dense layer with 32 units and a rectified linear activation function is added to the model. Either tanh or rectified linear activation functions can be used, as they are commonly used with the CuDNNLSTM layer.

The next step is to specify the optimizer. In this case, the Adam optimizer from the TensorFlow Keras library is used. The learning rate is set to 0.001, the decay rate to 1e-6, and the DK (decay step) to 1e-3.

The model is then compiled, with the loss function set to sparse categorical cross-entropy. The binary cross-entropy can also be used. The metric chosen for evaluation is accuracy.

Callbacks are defined next. The first callback is for TensorBoard, which is used for visualizing the training process. The log directory is specified as 'logs'.

The second callback is for model checkpointing. The file path is defined as 'checkpoint/epoch-{epoch:02d}-{val_accuracy:.2f}'. This callback saves the model weights at the end of each epoch.

Finally, the model is trained using the fit function. The training data (train_X and train_Y) and validation data (validation_X and validation_Y) are passed as arguments. The batch size and number of epochs are set according to the variables defined earlier. The callbacks for TensorBoard and model checkpointing are also passed.

After training, the model is saved using the save function.

This code snippet demonstrates the process of building a recurrent neural network model for predicting cryptocurrency prices. It includes the use of batch normalization, dense layers, activation functions, optimizers, loss functions, and callbacks for monitoring the training process.

In this didactic material, we discuss the application of recurrent neural networks (RNNs) in predicting cryptocurrency prices. Specifically, we will focus on the implementation of a cryptocurrency-predicting RNN model using Python, TensorFlow, and Keras.

Before we dive into the details of the RNN model, let's first address the importance of properly naming and organizing our model files. It is crucial to include a timestamp or some form of unique identifier in the file name to avoid overwriting previous models. This will help us keep track of different versions of our model and avoid potential data loss.

Now, let's move on to the implementation of the RNN model. We start by defining the log directory for our model, which will store the training logs. It is important to note that the log directory should be consistent throughout the code. In this case, we encounter a discrepancy in the variable name, which can be confusing. We should ensure that the variable name is consistent to avoid any confusion during the implementation.

To monitor the progress of our model, we can use a browser-based tool such as TensorBoard. By accessing the provided URL, we can visualize various metrics such as validation loss and accuracy. It is worth noting that validation accuracy is calculated at the end of each epoch, while accuracy is calculated for the entire epoch. This can sometimes result in slightly higher validation accuracy due to the model's learning progress during the epoch.

Moving on to the results, we observe that the validation loss and accuracy for different cryptocurrencies vary. For example, when predicting Ethereum (ETH) prices, we see a consistent trend in validation accuracy and loss. The same applies to other cryptocurrencies like Litecoin (LTC) and Bitcoin Cash (BCH). However, Bitcoin (BTC) exhibits a more fluctuating pattern, indicating potential challenges in accurately predicting its prices.

It is important to mention that the results presented here are specific to the dataset used and the implementation of the RNN model. Different datasets or variations in the model architecture may yield different results. Therefore, it is crucial to experiment with different approaches and datasets to obtain the most accurate predictions.

This didactic material has provided an overview of the implementation of a cryptocurrency-predicting RNN model using Python, TensorFlow, and Keras. We have discussed the importance of proper file naming and organization, monitoring the model's progress using TensorBoard, and analyzing the results obtained for different cryptocurrencies. Remember that further experimentation and exploration are necessary to improve the accuracy of cryptocurrency price predictions.

It is important to note that while the initial results of this model seem promising, caution must be exercised. Projects like these are prone to mistakes, and there may be bugs or limitations in the model. Therefore, it is recommended to use this model for educational purposes only and at your own risk.

To improve the accuracy of the model, additional features beyond pricing and volume can be incorporated. This may include factors such as market sentiment, news sentiment, or social media trends. By expanding the feature set, the model can capture more nuanced patterns and potentially improve its predictive capabilities.

Once the model is trained, it can be used to make predictions on new, unseen data. The model takes in the relevant features and outputs a prediction of the cryptocurrency movement. These predictions can be further analyzed and used to inform trading decisions.

The use of deep learning techniques, specifically recurrent neural networks, in predicting cryptocurrency movements shows promising results. By leveraging Python, TensorFlow, and Keras, we can develop a cryptocurrency-predicting RNN model that can potentially assist in making informed trading decisions. However, it is crucial to exercise caution and understand the limitations and potential risks associated with such models.