Introduction

Deep neural networks are now becoming one of the most popular methods of machine learning. They show better results than alternative methods in such areas as speech recognition, processing images, etc. The main reason for the successful application of deep neural networks is that the network automatically extracts from the data important signs necessary to solve the problem. This circumstance is an important advantage of neural networks when processing large amounts of data.

Recurrent neural networks are a kind of neural networks where the links between elements form a directed sequence. This makes it possible to process a series of events in time or successive spatial chains. Unlike multilayer perceptrons, recurrent networks can use their internal memory to process sequences of arbitrary length. Therefore, networks RNN are applicable in such problems, where the whole is broken down into certain segments (for example, searching for errors in program logs or recognizing human speech).

This report provides a brief overview of the main types of deep recurrent neural networks presented in the Keras framework, as well as an example of the work of recurrent neural networks by the example of an anomaly search problem in data logs.

Keras framework

As a development environment, the Keras framework was used. This framework supports deep learning in Python, providing a convenient way to create and learn for almost any models of in-depth training.

This framework has the following key characteristics, which determined the choice of this framework for solving problems:

-the ability to execute the same code on the CPU or on a GPU;

- the built-in API, which simplifies the development of prototypes of deep learning models;

-the intrinsic support of recurrent and convolutional networks;

- The availability of support for arbitrary network architectures, for example, models with multiple inputs and outputs, sharing models and layers, etc.

Thus, Keras is suitable for creating almost any model of machine learning. It is also used in large companies such as Google, Netflix, Uber for a wide range of tasks.

Tensorflow

Keras is a high-level library that does not implement low-level operations, such as manipulation with tensors and differentiation. To solve these problems, specialized tensor support libraries are used. Keras supports many such libraries. One of them is Tensorflow, developed by Google, which is the most common, scalable and high-quality platform for manipulating tensors. Any code that uses Keras can be run from any of these libraries without having to change anything in the code. If one of the libraries shows better performance during work, then during development you can switch to it.

Recurrent Neural Network (RNN) is a class of machine learning models based on using previous network states to calculate the current [5; 10]. Such networks are useful when the input data of a task is a non-fixed sequence of values, such as text data, where the text fragment is represented by a non-fixed number of sentences, phrases and words. Each character in the text, individual words, punctuation marks and even entire phrases - all this can be an atomic element of the input sequence.

Recurrent neural networks process input sequences one at a time at a time, maintaining the state (in this situation, this is a vector) throughout its entire length.

In the Keras framework, three recurring layers of the corresponding networks are programmatically implemented: SimpleRNN, GRU and LSTM. We consider each of the recurrent layers separately.

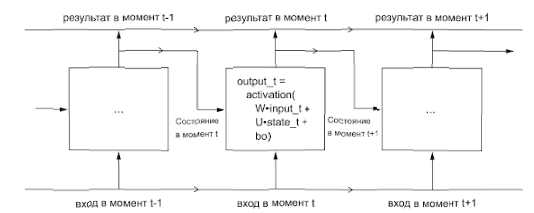
1. SimpleRNN

This layer is the easiest to use in the Keras framework. It processes sequence packets, like other layers in this framework, for example, the Embedding layer. This means that the input data for this layer will have the form: (packet size, time\_intervals, input\_primands). SimpleRNN operates in two different modes:

- Returns complete result sequences for all time intervals (three-dimensional tensor). To select this mode, you need to pass the argument return\_sequences = true to the recurrent layer;

- Returns the last result for each input sequence (two-dimensional tensor with form (packet size, input signs).

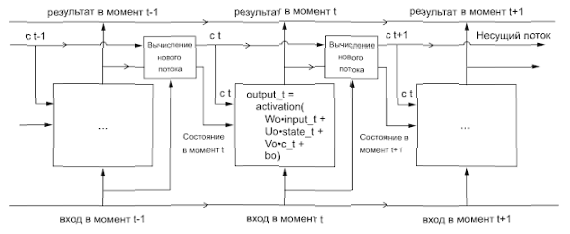
Figure 1 describes an example of a SimpleRNN cell.

This layer is the fastest to use for solving practical problems. However, it has a significant gradient attenuation problem, which makes it impossible to use it for large amounts of data. The essence of this problem lies in the fact that as the number of layers increases, the neural network becomes ineducable.

2. LSTM

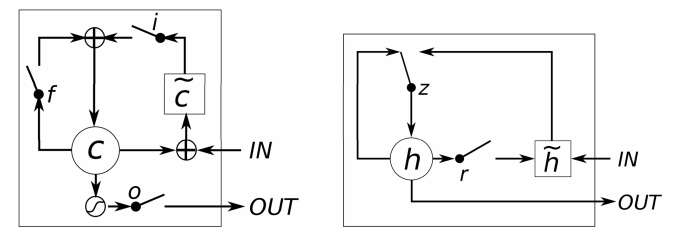
The basis of this recursive layer is algorithms for long short-term memory (LSTM), was developed in 1997. This algorithm was developed to solve the gradient attenuation problem, which is relevant for the SimpleRNN method.

This layer is a variation of the previous method, in which support was added to transfer information at intervals. The calculation of the next value in the data carrier is based on three filters (input i, forgetting f and output o), having the form of the SimpleRNN cell and the own weight matrices. The layout of the LSTM cell is described in Figure 2.

The LSTM model, unlike the SimpleRNN model, is less susceptible to the gradient attenuation problem, but it is more labor-intensive in computing. However, the entire strength of LSTM is manifested in solving more complex problems, for example, the task of pattern recognition.

3. GRU

The method of controlled recurrent blocks was developed in 2014. It is based on the same principle as LSTM layers, but unlike this method it uses fewer filters (two filters are used: update z and reset r), which affects the amount of non-linearity that comes from the input data and ultimately affects the order of calculations. The main differences between these recurrent networks are shown in Figure 3. The GRU method primarily implements a trade-off between the computational cost and the resultant power, which is not unimportant in the field of machine learning.



Solution of the task

Within the framework of the academic discipline, the task was to find anomalies based on logs. The developed code should find anomalies in these logs, such as atypical types of actions, atypical time of action, etc. Search for anomalies should be performed using recurrent neural networks.

To solve this problem, a code was developed using the Keras framework and recurrent neural networks SimpleRNN, GRU, LSTM. This chapter will describe the process of developing a solution for the task at hand.

Data Conversion

As the log file was taken the file "http.csv", which contains the following fields:

1. Id-identifier of the computer

2. Date-visit to the web page

3. User-identifier of the user of the system.

4. The url-site that the user has visited.

5. Activity-type of action

6. Content-content of the web page.

Of these parameters, the anomaly in the system is affected by a set of data consisting of the user of the system, the site on which he entered and the type of action. Therefore, when reading data from a file, we need to select the fields we need, in this case-user, url, activity.

The url field needs to be converted so that only the value of the domain to which the user logs on is left in the field.

To train the system, it is required to manipulate the data in such a way that they are divided into two tensors: input and target. The input tensor is the data on which further training of neural networks takes place. The target tensor is a variable with a certain number of classes.

In this case, the set of user and url fields for each line of the file previously combined into a single load-data combination was taken as the target tensor, and for the target tensor an activity field that can take one of three values: WWWVisit, WWWUpload or WWWDownload.

The code for the data conversion is described below.

def get\_url(url):

return urlparse(url).hostname

df = pd.read\_csv("r6.2/http.csv",nrows=10000,usecols=["user", "url", "activity"],)

df['url'] = df['url'].apply(get\_url)

df['activity'] = df['activity'].apply(delete\_slash)

df['load-data'] = df['user']+" "+ df['url']

del(df['url'])

del(df['user'])

ds = df.sample(frac=1).values

print(ds)

X=ds[:,1]

Y=ds[:,0]

Converting strings to code values

The learning model takes tensors as values ​​of the float type. However, our data is represented as a string type value. To solve this problem, we use the built-in Keras framework utility called Tokenizer. This utility uses direct encoding of words (one-hot encoding). This method consists in assigning to each word a unique integer index i and then converting it to a binary dictionary-sized branch whose elements contain zeros except the i-th element to be assigned to 1. For each word, a specific index is assigned, which depends on the number of occurrences of the word in the list. In particular, the most common word will be the index 1, and the least common word-index N, where N is the number of words in the dictionary. To get the index of all words, we use the method fit\_on\_texts of the Tokenizer utility, to convert strings to integer element lists using the texts\_to\_sentences method.

To normalize and convert the input tensor to a floating-point type, we use the pad\_sequences method of the sequence utility, which completes the number of elements of each section of the tensor with zeros on the left to a certain value, and converts the data into the required kind for learning.

To vectorize the target tensor, we use the to\_categorical method of the utility np\_utils, which allows to convert the list data into N + 1 elements, where N is the maximum value of the list to be converted.

t = Tokenizer()

t2 = Tokenizer()

t.fit\_on\_texts(X)

encoded = t.texts\_to\_sequences(X)

num\_words = len(t.word\_index)+1

max\_len=10

encoded = sequence.pad\_sequences(encoded , maxlen=max\_len, dtype='double')

t2.fit\_on\_texts(Y)

encoded2 = t2.texts\_to\_sequences(Y)

encoded2 = [item for sublist in encoded2 for item in sublist]

nb\_classes=4

encoded2=np\_utils.to\_categorical(encoded2, nb\_classes)

Training

For the training of neural networks, it is required at the beginning to divide the data into training and test data. The system is trained on training data, and then tests the percentage of accuracy on the test data.

To build neural networks, we apply the Sequential model built into the Keras framework, which is based on the assumption that the neural network has only one input and one output and consists of a linear stack of layers.

The initial model, on which the training takes place, consists of three layers:

1. Embedding layer. This layer is a layer of training weights. It is best understood as a dictionary that maps integer indices that denote specific words to dense vectors. The layer takes at least two arguments: the number of possible tokens, for example, the number of words in the dictionary and the dimension of the space. You can also apply additional parameters to the layer's input, for example, the length of the input space.

2. The layer of the neural network. This layer implements one of the recurrent networks, in particular, SimpleRNN, LSTM and GRU. As it was already described earlier, on input these layers take three parameters: packet size, time intervals, input characteristics. The last parameter is mandatory for input in each of the models.

3. The Dense layer is organized as a simple stack of fully connected layers with various activation operations. The argument N passed to each layer of Dense is the number of hidden layer neurons. A hidden neuron is a measurement in the space of representations of a layer. The presence of N hidden neurons means that the weight matrix W will have the form (input\_dimension, N): the scalar product on W projects the input data into the 16-dimensional space of representations. Then addition will be performed with the bias vector b, and an activation operation for the last level is performed, which establishes effective limitations on the result of the network. So, to solve the problem of binary classification, you need to select the activation function sigmoid, and for the multiclass single-valued specification task, the softmax function. The problem we are solving is the problem of binary classification.

After building the model, we compile it. To compile the model, you need to select a loss function and an optimizer. The loss function is a quantity that must be minimized during training, so it must be a measure of success for the task at hand. Since the problem being solved is the task of binary classification and the result of network operation is probability, it is preferable to use the loss function binary\_crossentropy. You can also select the mean\_squared\_error function, but it gives less qualitative results when probabilities are found, unlike the cross entropy method.

The optimizer determines the exact way to use the loss gradient to change the parameters. For example, the RMSProp optimizer implements the method of gradient descent with an impulse. To solve this problem, we applied the optimizer Adam, also based on a gradient descent.

After compiling the model, you can begin learning the network. In the case of using the Keras library, it is enough to call the fit method of the network-it tries to adapt the model to training data. In the learning process, two values ​​are displayed: network losses on training data and network accuracy on training data. For example, for the LSTM model built, the accuracy was 98%. After we check the control set:

scores = model.evaluate(X\_test,Y\_test, batch\_size=32)

print("Accuracy of LSTM: %.2f%%" % (scores[1]\*100))

The accuracy on the control set is 96.82%, which is different from the training set, which is due to the retraining of data, when the machine learning models show the worst accuracy on the new data set compared to the training set.

After the test check, we save the model. The architecture of the constructed model is saved in a file of the .json format, and the scales of the constructed recurrent network in the HDF5 (.h5) format.

Below is an example of a code for constructing a model containing a layer of a recurrent neural network of the LSTM type.

embedding\_vecor\_length = 10

nb\_classes=4

model = Sequential()

model.add(Embedding(num\_words, embedding\_vecor\_length, input\_length=10))

model.add(LSTM(num\_of\_cell))

model.add(Dense(nb\_classes, activation='softmax'))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

model.fit(X\_train,Y\_train,nb\_epoch=200,batch\_size=128)

scores = model.evaluate(X\_test,Y\_test, batch\_size=32)

print("Accuracy of LSTM: %.2f%%" % (scores[1]\*100))

#saving the model

model\_json = model.to\_json()

with open("lstm-model.json", "w") as json\_file:

json\_file.write(model\_json)

# serialize weights to HDF5

model.save\_weights("lstm-model.h5")

print("Saved model to disk")

Table 1 shows the accuracy of recurrent neural networks on training and test data.

Table 1

Result of testing recurrent neural networks for test data (10,000 data, 200 epochs)

|  |  |  |
| --- | --- | --- |
|  | Accuracy on training data | Accuracy on test data |
| LSTM | 98.0 % | 96.82 % |
| SimpleRNN | 98.05 % | 96.77 % |
| GRU | 98.08 % | 96.90 % |

If the amount of data increases, the accuracy of the network may increase, but for all methods, the gradient damping problem (primarily for the SimpleRNN method, but this also applies to the other two methods) is possible.

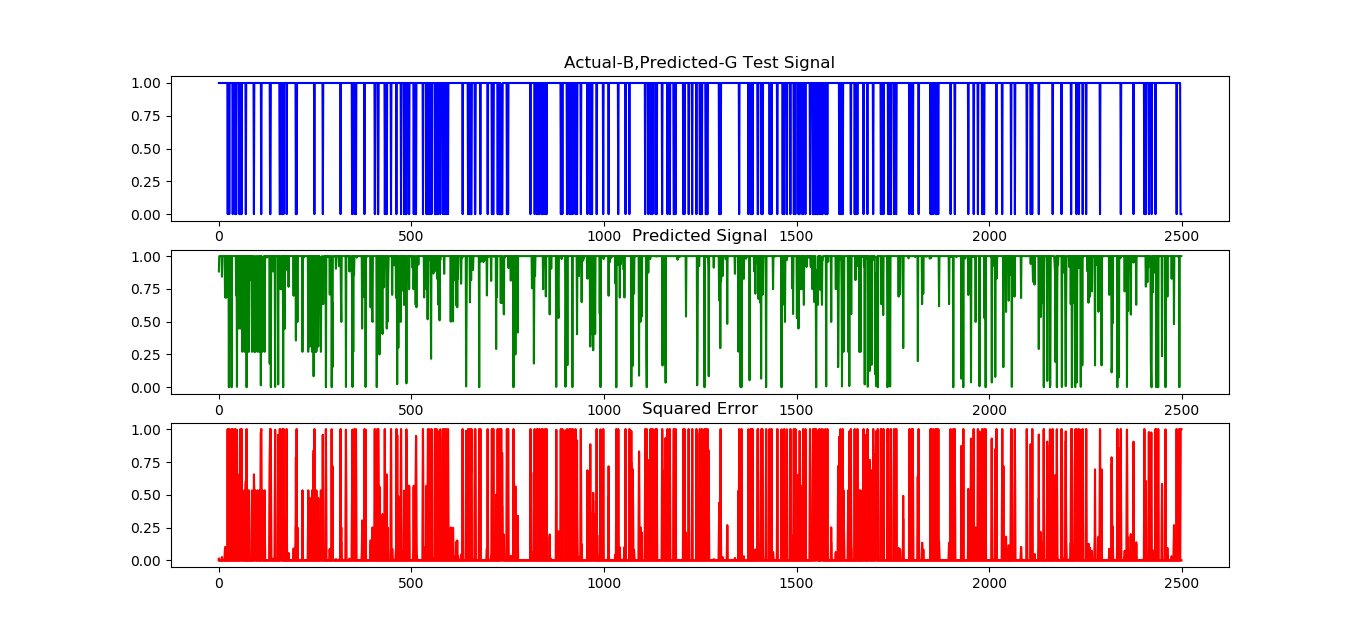
The amount of data and passages was chosen expediently to this task in order to demonstrate the different approaches of recurrent neural networks. In the case of more target data, the use of LSTM and GRU recurrent networks is recommended.

Testing data on specific examples

Now you need to test the data on specific examples. As test data, a specially written log was taken, containing examples of obvious anomalies, such as several downloads in a row or downloading content to a site from which the user usually downloads information.

In order to bring the log to the type required for use in the model, you need to perform manipulations with this log (see the "Data Conversion" and "Converting Rows to Code Values" clauses of the report). Before starting the function of predicting the values ​​of the user's actions in the system, it is necessary to load the model from the files in which it is stored.

The prediction function is used to predict the probability of a particular result. This function returns the probability for each specific value of the target tensor class. For example, if the number of classes of the target tensor is 4, the result of the predict function is a sheet of size Nx4, where N is the number of target data.

For each class, we plot the graph using the matplotlib module. This graph consists of three elements: the probability curve predicted by the model, the real model graph, and the root-mean-square error graph for this class. Using this graph, you can determine which class is most likely for the data. An example of such a graph for the WWWVisit class is described in Figure 4.

To find the anomalies, a special algorithm was written. The input lists the probabilities predicted by the model and the list of real probabilities. For each element of the two lists, the index of the maximum element is found. In the event that the indices of the maximum elements of the corresponding elements from the two lists do not coincide, then the algorithm identifies the anomaly.

If an anomaly is found, it is entered in the dictionary, where the key is the combination of the word encoding of the target tensor, and the value is the number of anomalies identified for the combination of the user and the site that he visited.

Table 2 shows the number of anomalies found for each of the recurrent networks.

Based on this table, we can conclude that the most successful recurrence of anomalies are recurrent networks of the GRU type. The SimpleRNN neural network, although it found about as many anomalies as the GRU, but its predictions were inaccurate in most cases, which makes it less able to find real anomalies than the LSTM and GRU networks.

Table 2

Number of anomalies in logs found using recurrent networks

|  |  |
| --- | --- |
| LSTM | 305 |
| SimpleRNN | 378 |
| GRU | 373 |

Conclusion

As a result of the work, the main types of recurrent networks used in the Keras framework were studied, as well as the main advantages of different types of recurrent neural networks on the example of a specific task.

Currently, recurrent neural networks are experiencing a boom in their development. The main trends were the combination of different architectures and the application of developments from other areas to improve them. From the examples you can name the already mentioned neural networks from Google, in which they use methods taken from reinforcement training, Turing neural machines, optimization algorithms like Batch Normalization and much more - all this deserves a separate article. In general, although recurrent networks do not attract the same attention as convolutional networks, this is only because the objects and tasks with which recurrent networks operate are not as obvious as applications based on convolutional networks, such as DeepDream or Prisma.

Literature

1. Budylskii DV. GRU and LSTM: modern recurrent neural networks // Young scientist. - 2015. - No. 15. - P. 51-54. - URL https://moluch.ru/archive/95/21426/ (reference date: May 22, 2018).
2. LSTM: A Search Space Odyssey / K. Greff [et al.] // arXiv preprint arXiv:1503.04069. — 2015.
3. Multi-resolution linear prediction based features for audio onset detection with bidirectional LSTM neural networks / E. Marchi [et al.] // Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. — IEEE. 2014. — Pp. 2164–2168.
4. Scholle, F. Deep training in Python // SPb- 2018. - 400 p.