Keras autoencoder

Asked 4 years, 6 months ago Active 4 years ago Viewed 4k times

2

I've worked a long time ago with neural networks in Java and now I'm trying to learn to use TFLearn and Keras in Python.

I'm trying to build an autoencoder, but as I'm experiencing problems the code I show you hasn't got the bottleneck characteristic (this should make the problem even easier).

On the following code I create the network, the dataset (two random variables), and after train it plots the correlation between each predicted variable with its input.

What the network should learn, is to output the same input that receives.

```
import matplotlib.pyplot as plt
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras.models import load_model
from loaders.nslKddCup99.nslKddCup99Loader import NslKddCup99

def buildMyNetwork(inputs, bottleNeck):
    inputLayer = Input(shape=(inputs,))
    autoencoder = Dense(inputs*2, activation='relu')(inputLayer)
    autoencoder = Dense(inputs*2, activation='relu')(autoencoder)
    autoencoder = Dense(bottleNeck, activation='relu')(autoencoder)
    autoencoder = Dense(inputs*2, activation='relu')(autoencoder)
    autoencoder = Dense(inputs*2, activation='relu')(autoencoder)
    autoencoder = Dense(inputs, activation='sigmoid')(autoencoder)
    autoencoder = Model(input=inputLayer, output=autoencoder)
    autoencoder.compile(optimizer='adadelta', loss='mean_squared_error')
    return autoencoder


dataSize = 1000
variables = 2
data = np.zeros((dataSize,variables))
data[:, 0] = np.random.uniform(0, 0.8, size=dataSize)
data[:, 1] = np.random.uniform(0, 0.1, size=dataSize)

trainData, testData = data[:900], data[900:]

model = buildMyNetwork(variables,2)
model.fit(trainData, trainData, nb_epoch=2000)
predictions = model.predict(testData)

for x in range(variables):
    plt.scatter(testData[:, x], predictions[:, x])
    plt.show()
    plt.close()
```

Even though some times the result is acceptable, many others isn't, I know neural networks have weight random initialization and therefore it may converge to different solutions, but I think this is too much and there may be some mistake in my code.

Sometimes correlation is acceptable

[Others is quite lost]

**

UPDATE:

**

Thanks Marcin Możejko!

Indeed that was the problem, my original question was because I was trying to build an autoencoder, so to be coherent with the title here comes an example of autoencoder (just making a more complex dataset and changing the activation functions):

```
import matplotlib.pyplot as plt
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras.models import load_model
from loaders.nslKddCup99.nslKddCup99Loader import NslKddCup99

def buildMyNetwork(inputs, bottleNeck):
    inputLayer = Input(shape=(inputs,))
    autoencoder = Dense(inputs*2, activation='tanh')(inputLayer)
    autoencoder = Dense(inputs*2, activation='tanh')(autoencoder)
    autoencoder = Dense(bottleNeck, activation='tanh')(autoencoder)
    autoencoder = Dense(inputs*2, activation='tanh')(autoencoder)
    autoencoder = Dense(inputs*2, activation='tanh')(autoencoder)
    autoencoder = Dense(inputs, activation='tanh')(autoencoder)
    autoencoder = Model(input=inputLayer, output=autoencoder)
    autoencoder.compile(optimizer='adadelta', loss='mean_squared_error')
    return autoencoder


dataSize = 1000
variables = 6
data = np.zeros((dataSize,variables))
data[:, 0] = np.random.uniform(0, 0.5, size=dataSize)
data[:, 1] = np.random.uniform(0, 0.5, size=dataSize)
data[:, 2] = data[:, 0] + data[:, 1]
data[:, 3] = data[:, 0] * data[:, 1]
data[:, 4] = data[:, 0] / data[:, 1]
data[:, 5] = data[:, 0] ** data[:, 1]

trainData, testData = data[:900], data[900:]

model = buildMyNetwork(variables,2)
model.fit(trainData, trainData, nb_epoch=2000)
predictions = model.predict(testData)

for x in range(variables):
    plt.scatter(testData[:, x], predictions[:, x])
    plt.show()
    plt.close()
```

For this example I used TanH activation function, but I tried with others and worked aswell. The dataset has now 6 variables but the autoencoder has a bottleneck of 2 neurons; as long as variables 2 to 5 are formed combining variables 0 and 1, the autoencoder only needs to pass the information of those two and learn the functions to generate the other variables on the decoding phase. The example above shows how all functions are learnt but one, the division... I don't know yet why.

python machine-learning neural-network keras autoencoder Edit tags

Share Edit Follow Close Flag

edited Aug 31 '17 at 16:00
Marcin Możejko
36.1k 9 97 112

asked Feb 28 '17 at 17:17
Cae
81 2 7

stackoverflow.com/questions/47822154/... any suggestions? – Dexter Dec 15 '17 at 8:22

2 Answers

Active Oldest Votes

0

This is a really cool usecase to see and study the difficulties of training a Neural Net.

Indeed I see multiple possibilities :

1) If there are very different results between 2 different runs, it can come from the initialization. But it can also come from your data set which isn't the same at every run.

2) Something that will make it difficult for the network to learn the correlation is your activations, more precisely the sigmoid. I would change that non linearity to another 'relu'. No reason to use a Sigmoid here, I know it looks linear around 0 but it isnt really linear. To produce a 0.7, the raw output will have to be quite high. The "easy" relation that you have in mind for this network isnt so easy since you constrain it.

3) if it is sometimes not good, maybe it needs more epochs to converge?

4) maybe you'd need a bigger dataset? In theory you're good since there is 900 samples for < 200 parameters in your network but who knows...

I can't try all of this since I'm on my phone right now but feel free to try and troubleshout with the hints I gave you :) I hope this helps.

EDIT :

After some trials, as Marcin Możejko was saying, the issue comes from the activations. You can read his answer to have more info on what is going wrong. The way to fix it is to change your activations. If you are a fan of `'relu'`, you can use a special version of this activation. For this, use a LeakyRelu layer and do not set an activation to the previous layer, like this :

```
autoencoder = Dense(inputs*2)(inputLayer)
autoencoder = LeakyReLU(alpha=0.3)(autoencoder)
```

This will solve the case where you get stuck in a nonoptimal solution. Otherwise, as I said above, you can try not to use any non-linearities. Your loss will go down way faster and doesn't get stuck.

Keep in mind that the non linearities were introduced to get the networks to find more complex patterns in the data. In your case you have the simplest linear pattern.

Share Edit Follow Flag edited Mar 1 '17 at 8:35 answered Feb 28 '17 at 21:11 Nassim Ben 10.5k 1 30 47

4

I think that your case is relatively easy to explain why your network might fail to learn an identity function. Let's go through your example:

1. Your input comes from `2d` space - and it doesn't lie on a `1d` or `0d` submanifold - due to uniform distribiution. From this it's easy to see that in order to get an identity function from your autoencoder **every** layer should be able to represent a function which range is at least **two dimensional**, beacuse the output of your last layer should also lie on a `2d` manifold.
2. Let's go through your network and check if it satisfy the condtion need:

```
inputLayer = Input(shape=(2,))
autoencoder = Dense(4, activation='relu')(inputLayer)
autoencoder = Dense(4, activation='relu')(autoencoder)
autoencoder = Dense(2, activation='relu')(autoencoder) # Possible problems here
```

   You may see that the bottleneck might cause a problem - for this layer it might be hard to satisfy the condition from the first point. For this layer - in order to get the 2-dimensional output range you need to have weights which will make all the examples not falling into saturation region of `relu` (in this case all this samples will be squashed to `0` in one of the units - what makes impossible for range to be "fully" `2d`). So basically - the probability that this will not happen is relatively small. Also the probability that backpropagation will not move this unit to this region also cannot be neglected.

UPDATE:

In a comment the question was asked why optimizer fail to prevent or undo the saturation. It's an example of a one of the important `relu` downsides - once an example falls into a `relu` saturation region - this example doesn't directly take part in learning of a given unit. It could affect it by influencing previous units - but due to `0` derivative - this influence is not direct. So basically *unsaturating an example* comes from a side effect - not the direct action of an optimizer.

Share Edit Follow Flag edited Feb 28 '17 at 22:30 answered Feb 28 '17 at 21:55 Marcin Możejko 36.1k 9 97 112

Cool explaination! But I'm still not really certain to understand why the network couldn't learn that the weight should all be positives?! This is basic backpropagation, the updates should go in the right direction. From a math point of view, I see a global minimum that can be achieved, there are even several of them due to the higher dimensions of the first layers. We can even find the weight to achieve this minimum manually. How come that the optimizer can't find it? Maybe we should use another optimizer? – Nassim Ben Feb 28 '17 at 22:23

Or just linear activations? As I mentionned in the answer below, the nonlinearities don't really make sense here right?! We don't need to activate or desactivate neurons here... no need for complex patterns, only propagate the data without loosing information – Nassim Ben Feb 28 '17 at 22:35