ADVENTURES IN MACHINE LEARNING

ython TensorFlow utorial – Build a Neural etwork

April 8, 2017 Andy Deep learning, Neural networks, nsorFlow 12

The TensorFlow logo

Google's TensorFlow has been a hot topic in deep learning recently. The open source software, designed to allow efficient computation of data flow graphs, is especially suited to deep learning tasks. It is designed to be executed on single or multiple CPUs and GPUs, making it a good option for complex deep learning tasks. In it's most recent incarnation – version 1.0 – it can even be run on certain mobile operating systems. This introductory tutorial to TensorFlow will give an overview of some of the basic concepts of TensorFlow in Python. These will be a good stepping stone to building more complex deep learning networks, such as Convolution Neural Networks, natural language models and Recurrent Neural Networks in the package. We'll be creating a simple three-layer neural network to

POPULAR TUTORIALS

Neural Networks Tutorial – A Pathway to Deep Learning
Python TensorFlow Tutorial – Build a Neural Network
Convolutional Neural Networks Tutorial in TensorFlow
Keras tutorial – build a convolutional neural network in 11 lines
Word2Vec word embedding tutorial in Python and TensorFlow

CATEGORIES

CNTK
Convolutional Neural Networks
Deep learning
gensim
Keras
Neural networks
NLP
Optimisation
TensorFlow
Word2Vec

classify the MNIST dataset. This tutorial assumes that you are familiar with the basics of neural networks, which you can get up to scratch with in the ral networks tutorial if required. To install TensorFlow, ow the instructions here. The code for this tutorial can be nd in this site's GitHub repository. Once you're done, you might want to check out a higher level deep learning library sits on top of TensorFlow called Keras – see my Keras rial.

Recommended online course: Once you've read this post, and you'd like to learn more in a video course, I'd recommend the following inexpensive Udemy course: Data Science: Practical Deep Learning in Theano + TensorFlow

---

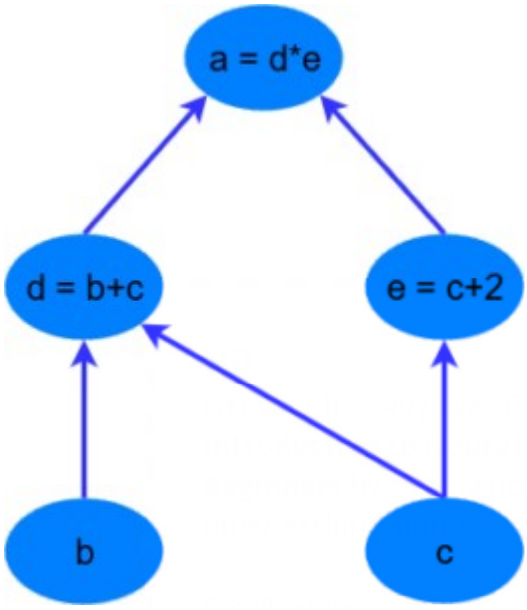
First, let's have a look at the main ideas of TensorFlow.

1.0 TensorFlow graphs

TensorFlow is based on graph based computation – "what on earth is that?", you might say. It's an alternative way of conceptualising mathematical calculations. Consider the following expression $a = (b + c) * (c + 2)$. We can break this function down into the following components:

$$d = b + c$$
$$e = c + 2$$
$$a = d * e$$

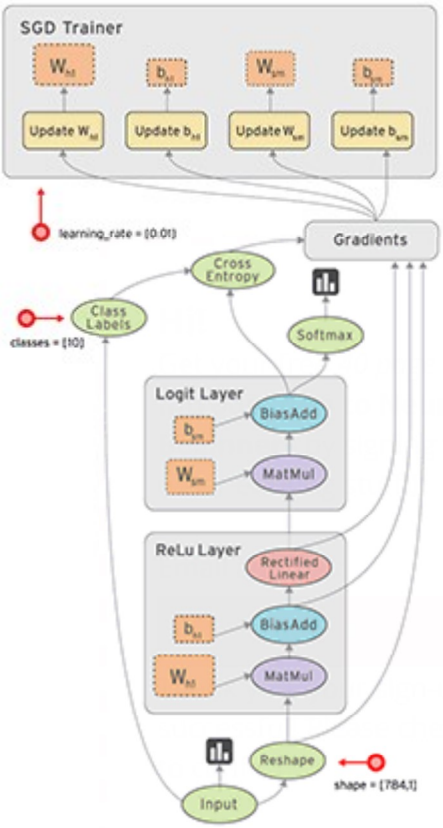
Now we can represent these operations graphically as:



Simple computational graph

; may seem like a silly example – but notice a powerful idea in ressing the equation this way: two of the computations = $b + c$ and $e = c + 2$) can be performed in parallel. By :ting up these calculations across CPUs or GPUs, this can give ignificant gains in computational times. These gains are a :t for big data applications and deep learning – especially for complicated neural network architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). The idea behind TensorFlow is to ability to create these computational graphs in code and allow significant performance improvements via parallel operations and other efficiency gains.

We can look at a similar graph in TensorFlow below, which shows the computational graph of a three-layer neural network.



TensorFlow data flow graph

The animated data flows between different nodes in the graph are *tensors* which are multi-dimensional data arrays. For instance, the input data tensor may be 5000 x 64 x 1, which represents a 64 node input layer with 5000 training samples. After the input layer there is a hidden layer with rectified linear units as the activation function. There is a final output layer (called a "logit layer" in the above graph) which uses cross entropy as a cost/loss function. At each point we see the relevant tensors flowing to the "Gradients" block which finally flow to the Stochastic Gradient Descent optimiser which performs the back-propagation and gradient descent.

Here we can see how computational graphs can be used to represent the calculations in neural networks, and this, of course, is what TensorFlow excels at. Let's see how to perform some basic mathematical operations in TensorFlow to get a feel for how it all works.

2.0 A Simple TensorFlow example

Let's first make TensorFlow perform our little example calculation above – $a = (b + c) * (c + 2)$. First we need to introduce ourselves to TensorFlow *variables* and *constants*. Let's declare some then I'll explain the syntax:


```
port tensorflow as tf

793 first, create a TensorFlow constant
nst = tf.constant(2.0, name="const")

155
create TensorFlow variables
= tf.Variable(2.0, name='b')
= tf.Variable(1.0, name='c')
```

an be observed above, TensorFlow constants can be lared using the *tf.constant* function, and variables with the *riable* function. The first element in both is the value to be gned the constant / variable when it is initialised. The second n optional name string which can be used to label the stant / variable – this is handy for when you want to do alisations (as will be discussed briefly later). TensorFlow will infer the type of the constant / variable from the initialised value, but it can also be set explicitly using the optional *dtype* argument. TensorFlow has many of its own types like *tf.float32*, *tf.int32* etc. – see them all here.

It's important to note that, as the Python code runs through these commands, the variables haven't actually been declared as they would have been if you just had a standard Python declaration (i.e. b = 2.0). Instead, all the constants, variables, operations and the computational graph are only created when the initialisation commands are run.

Next, we create the TensorFlow *operations*:

```
# now create some operations
d = tf.add(b, c, name='d')
e = tf.add(c, const, name='e')
a = tf.multiply(d, e, name='a')
```

TensorFlow has a wealth of operations available to perform all sorts of interactions between variables, some of which we'll get to later in the tutorial. The operations above are pretty obvious,

and they instantiate the operations $b + c$, $c + 2.0$ and $d * e$.

The next step is to setup an object to initialise the variables and the graph structure:


```
setup the variable initialisation
it_op = tf.global_variables_initializer()
```

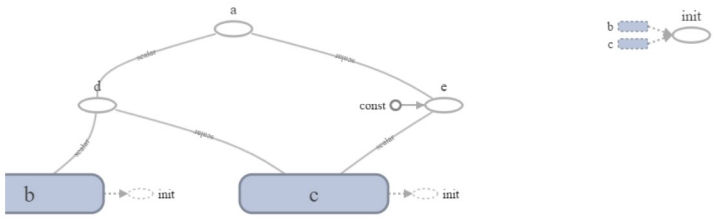

so now we are all set to go. To run the operations between

variables, we need to start a TensorFlow session – *tf.Session.* TensorFlow session is an object where all operations are . Using the *with* Python syntax, we can run the graph with the owing code:

```
start the session
th tf.Session() as sess:
  # initialise the variables
  sess.run(init_op)
  # compute the output of the graph
  a_out = sess.run(a)
  print("Variable a is {}".format(a_out))
```

The first command within the *with* block is the initialisation, which is run with the, well, *run* command. Next we want to figure out what the variable *a* should be. All we have to do is run the operation which calculates *a* i.e. *a = tf.multiply(d, e, name='a').* Note that *a* is an *operation*, not a variable and therefore it can be *run*. We do just that with the *sess.run(a)* command and assign the output to *a_out,* the value of which we then print out.

Note something cool – we defined operations *d* and *e* which need to be calculated before we can figure out what *a* is. However, we don't have to explicitly run *those* operations, as TensorFlow knows what other operations and variables the operation *a* depends on, and therefore runs the necessary operations on its own. It does this through its data flow graph which shows it all the required dependencies. Using the TensorBoard functionality, we can see the graph that TensorFlow created in this little program:



ole TensorFlow graph

v that's obviously a trivial example – what if we had an array values that we wanted to calculate the value of *a* over?

1 The TensorFlow placeholder

s also say that we didn't know what the value of the array *b* uld be during the declaration phase of the TensorFlow blem (i.e. before the *with tf.Session() as sess*) stage. In this e, TensorFlow requires us to declare the basic structure of the a by using the *tf.placeholder* variable declaration. Let's use it b:

```
# create TensorFlow variables
b = tf.placeholder(tf.float32, [None, 1], name='b')
```

Because we aren't providing an initialisation in this declaration, we need to tell TensorFlow what data type each element within the *tensor* is going to be. In this case, we want to use *tf.float32*. The second argument is the shape of the data that will be "injected" into this variable. In this case, we want to use a (? x 1) sized array – because we are being cagey about how much data we are supplying to this variable (hence the "?"), the placeholder is willing to accept a *None* argument in the size declaration. Now we can inject as much 1-dimensional data that we want into the *b* variable.

The only other change we need to make to our program is in the *sess.run(a,...)* command:

```
a_out = sess.run(a, feed_dict={b: np.arange(0, 10)[:,
np.newaxis]})
```

Note that we have added the *feed_dict* argument to the *sess.run(a,...)* command. Here we remove the mystery and

specify exactly what the variable *b* is to be – a one-dimensional range from 0 to 10. As suggested by the argument name, *feed_dict,* the input to be supplied is a Python dictionary, with each key being the name of the *placeholder* that we are

ıg.

ɜn we run the program again this time we get:


```
riable a is [[  3.]
  6.]
  9.]
 12.]
 15.]
 18.]
 21.]
 24.]
 27.]
 30.]]
```

ice how TensorFlow adapts naturally from a scalar output (i.e. ıgular output when *a=9.0*) to a *tensor* (i.e. an array/matrix)? This is based on its understanding of how the data will flow through the graph.

Now we are ready to build a basic MNIST predicting neural network.

3.0 A Neural Network Example

Now we'll go through an example in TensorFlow of creating a simple three layer neural network. In future articles, we'll show how to build more complicated neural network structures such as convolution neural networks and recurrent neural networks. For this example though, we'll keep it simple. If you need to scrub up on your neural network basics, check out my popular tutorial on the subject. In this example, we'll be using the MNIST dataset (and its associated loader) that the TensorFlow package provides. This MNIST dataset is a set of 28×28 pixel grayscale images which represent hand-written digits. It has 55,000 training rows, 10,000 testing rows and 5,000 validation rows.

We can load the data by running:

```
from tensorflow.examples.tutorials.mnist import
input_data
ist = input_data.read_data_sets("MNIST_data/",
e_hot=True)
```


one_hot=True argument specifies that instead of the labels
ociated with each image being the digit itself i.e. "4", it is a

or with "one hot" node and all the other nodes being zero i.e.
), 0, 0, 1, 0, 0, 0, 0, 0]. This lets us easily feed it into the output
r of our neural network.

1 Setting things up

t, we can set-up the placeholder variables for the training
a (and some training parameters):

```
Python optimisation variables
arning_rate = 0.5
pochs = 10
batch_size = 100

# declare the training data placeholders
# input x - for 28 x 28 pixels = 784
x = tf.placeholder(tf.float32, [None, 784])
# now declare the output data placeholder - 10 digits
y = tf.placeholder(tf.float32, [None, 10])
```

Notice the *x* input layer is 784 nodes corresponding to the 28 x 28 (=784) pixels, and the *y* output layer is 10 nodes corresponding to the 10 possible digits. Again, the size of *x* is (? x 784), where the ? stands for an as yet unspecified number of samples to be input – this is the function of the *placeholder* variable.

Now we need to setup the weight and bias variables for the three layer neural network. There are always *L-1* number of weights/bias tensors, where *L* is the number of layers. So in this case, we need to setup two tensors for each:

```
# now declare the weights connecting the input to the
```

```
hidden layer
W1 = tf.Variable(tf.random_normal([784, 300],
stddev=0.03), name='W1')
b1 = tf.Variable(tf.random_normal([300]), name='b1')
and the weights connecting the hidden layer to the
tput layer
 = tf.Variable(tf.random_normal([300, 10],
ddev=0.03), name='W2')
 = tf.Variable(tf.random_normal([10]), name='b2')
```


so let's unpack the above code a little. First, we declare some ables for W1 and b1, the weights and bias for the connections ween the input and hidden layer. This neural network will e 300 nodes in the hidden layer, so the size of the weight sor W1 is [784, 300]. We initialise the values of the weights ıg a random normal distribution with a mean of zero and a ıdard deviation of 0.03. TensorFlow has a replicated version ıe numpy random normal function, which allows you to ıte a matrix of a given size populated with random samples wn from a given distribution. Likewise, we create W2 and b2 ables to connect the hidden layer to the output layer of the neural network.

Next, we have to setup node inputs and activation functions of the hidden layer nodes:

```
# calculate the output of the hidden layer
hidden_out = tf.add(tf.matmul(x, W1), b1)
hidden_out = tf.nn.relu(hidden_out)
```

In the first line, we execute the standard matrix multiplication of the weights (*W1*) by the input vector *x* and we add the bias *b1*. The matrix multiplication is executed using the *tf.matmul* operation. Next, we finalise the *hidden_out* operation by applying a rectified linear unit activation function to the matrix multiplication plus bias. Note that TensorFlow has a rectified linear unit activation already setup for us, *tf.nn.relu*.

This is to execute the following equations, as detailed in the neural networks tutorial:

$$z^{(l+1)} = W^{(l)}x + b^{(l)}$$
$$h^{(l+1)} = f(z^{(l+1)})$$

Now, let's setup the output layer, *y_*:

```
# now calculate the hidden layer output - in this
case, let's use a softmax activated
output layer
 = tf.nn.softmax(tf.add(tf.matmul(hidden_out, W2),
))
```


in we perform the weight multiplication with the output from

hidden layer (*hidden_out*) and add the bias, *b2*. In this case,
are going to use a softmax activation for the output layer –
can use the included TensorFlow softmax function
n.softmax.

also have to include a cost or loss function for the
misation / backpropagation to work on. Here we'll use the
s entropy cost function, represented by:

$$= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_j^{(i)} log(y_j\_^{(i)}) + (1-y_j^{(i)}) log(1-y_j\_^{(i)})$$

Where $y_j^{(i)}$ is the ith training label for output node *j*, $y_j\_^{(i)}$ is the ith predicted label for output node *j*, *m* is the number of training / batch samples and *n* is the number . There are two operations occurring in the above equation. The first is the summation of the logarithmic products and additions *across all the output nodes*. The second is taking a mean of this summation *across all the training samples*. We can implement this cross entropy cost function in TensorFlow with the following code:

```
y_clipped = tf.clip_by_value(y_, 1e-10, 0.9999999)
cross_entropy = -tf.reduce_mean(tf.reduce_sum(y *
tf.log(y_clipped)
                        + (1 - y) * tf.log(1 -
y_clipped), axis=1))
```

Some explanation is required. The first line is an operation converting the output *y_* to a clipped version, limited between 1e-10 to 0.999999. This is to make sure that we never get a case were we have a *log(0)* operation occurring during training – this would return NaN and break the training process. The second line is the cross entropy calculation.

To perform this calculation, first we use TensorFlow's *tf.reduce_sum* function – this function basically takes the sum of a given axis of the tensor you supply. In this case, the tensor that is supplied is the element-wise cross-entropy calculation for a

ile node and training sample i.e.: $log(y_j\_^{(i)}) + (1-y_j^{(i)})log(1-y_j\_^{(i)})$. Remember that *y* and *pped* in the above calculation are (*m* x *10*) tensors – therefore

need to perform the first sum over the second axis. This is cified using the axis=1 argument, where "1" actually refers to

second axis when we have a zero-based indices system like ion.

r this operation, we have an (*m* x *1*) tensor. To take the mean his tensor and complete our cross entropy cost calculation execute this part $\frac{1}{m}\sum_{i=1}^{m}$), we use TensorFlow's *duce_mean* function. This function simply takes the mean of itever tensor you provide it. So now we have a cost function we can use in the training process.

s setup the optimiser in TensorFlow:

```
# add an optimiser
optimiser =
tf.train.GradientDescentOptimizer(learning_rate=learn
ing_rate).minimize(cross_entropy)
```

Here we are just using the gradient descent optimiser provided by TensorFlow. We initialize it with a learning rate, then specify what we want it to do – i.e. minimise the cross entropy cost operation we created. This function will then perform the gradient descent (for more details on gradient descent see here and here) and the backpropagation for you. How easy is that? TensorFlow has a library of popular neural network training optimisers, see here.

Finally, before we move on to the main show, were we actually run the operations, let's setup the variable initialisation operation and an operation to measure the accuracy of our predictions:

```
# finally setup the initialisation operator
init_op = tf.global_variables_initializer()

# define an accuracy assessment operation
```

```
correct_prediction = tf.equal(tf.argmax(y, 1),
tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
```


correct prediction operation *correct_prediction* makes use of TensorFlow *tf.equal* function which returns *True* or *False* ending on whether to arguments supplied to it are equal. *tf.argmax* function is the same as the numpy argmax ction, which returns the index of the maximum value in a :or / tensor. Therefore, the *correct_prediction* operation irns a tensor of size (*m* x *1*) of *True* and *False* values ignating whether the neural network has correctly predicted digit. We then want to calculate the mean accuracy from this sor – first we have to cast the type of the *correct_prediction* ration from a Boolean to a TensorFlow float in order erform the *reduce_mean* operation. Once we've done that, now have an *accuracy* operation ready to assess the formance of our neural network.

2 Setting up the training

We now have everything we need to setup the training process of our neural network. I'm going to show the full code below, then talk through it:

```
 # start the session
 with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    total_batch = int(len(mnist.train.labels) /
batch_size)
    for epoch in range(epochs):
         avg_cost = 0
         for i in range(total_batch):
             batch_x, batch_y =
mnist.train.next_batch(batch_size=batch_size)
              _, c = sess.run([optimiser,
cross_entropy],
                             feed_dict={x: batch_x, y:
batch_y})
             avg_cost += c / total_batch
         print("Epoch:", (epoch + 1), "cost =",
```

```
"{:.3f}".format(avg_cost))
    print(sess.run(accuracy, feed_dict={x:
mnist.test.images, y: mnist.test.labels}))
```


ɔping through the lines above, the first couple relate to
ing up the *with* statement and running the initialisation
ration. The third line relates to our mini-batch training
793 eme that we are going to run for this neural network. If you
ıt to know about mini-batch gradient descent, check out this
155 t. In the third line, we are calculating the number of batches
ɹn through in each training epoch. After that, we loop
ɔugh each training epoch and initialise an *avg_cost* variable to
ɒ track of the average cross entropy cost for each epoch. The
t line is where we extract a randomised batch of samples,
:h_x and *batch_y*, from the MNIST training dataset. The
sorFlow provided MNIST dataset has a handy utility function,
:_batch, that makes it easy to extract batches of data for
ıing.

following line is where we run two operations. Notice that
.run is capable of taking a list of operations to run as its first
argument. In this case, supplying *[optimiser, cross_entropy]* as the list means that both these operations will be performed. As such, we get two outputs, which we have assigned to the variables _ and *c*. We don't really care too much about the output from the *optimiser* operation but we want to know the output from the *cross_entropy* operation – which we have assigned to the variable *c*. Note, we run the *optimiser* (and *cross_entropy*) operation on the batch samples. In the following line, we use *c* to calculate the average cost for the epoch.

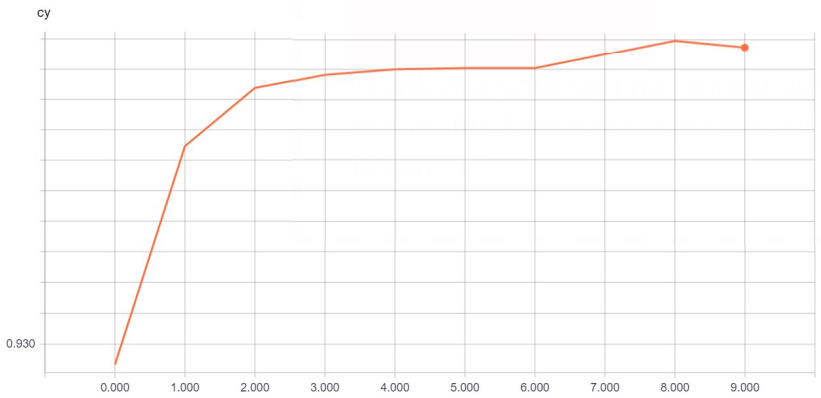
Finally, we print out our progress in the average cost, and after the training is complete, we run the *accuracy* operation to print out the accuracy of our trained network on the test set. Running this program produces the following output:

```
Epoch: 1 cost = 0.586
Epoch: 2 cost = 0.213
Epoch: 3 cost = 0.150
Epoch: 4 cost = 0.113
Epoch: 5 cost = 0.094
Epoch: 6 cost = 0.073
Epoch: 7 cost = 0.058
```

```
Epoch: 8 cost = 0.045
Epoch: 9 cost = 0.036
Epoch: 10 cost = 0.027

aining complete!
9787
```


re we go – approximately 98% accuracy on the test set, not . We could do a number of things to improve the model, ı as regularisation (see this tips and tricks post), but here we just interested in exploring TensorFlow. You can also use sorBoard visualisation to look at things like the increase in ıracy over the epochs:



TensorBoard plot of the increase in accuracy over 10 epochs

In a future article, I'll introduce you to TensorBoard visualisation, which is a really nice feature of TensorFlow. For now, I hope this tutorial was instructive and helps get you going on the TensorFlow journey. Just a reminder, you can check out the code for this post here. I've also written an article that shows you how to build more complex neural networks such as convolution neural networks and Word2Vec natural language models in TensorFlow. You also might want to check out a higher level deep learning library that sits on top of TensorFlow called Keras – see my Keras tutorial. I'll also be writting a new article on recurrent neural networks and LSTMs soon. So stay tuned.

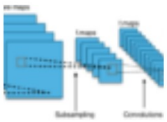
Have fun!

Recommended online course: If you'd like to dive a little deeper I'd recommend the following inexpensive Udemy video course: Data Science: Practical Deep Learning in Theano + TensorFlow

---

« PREVIOUS
Stochastic Gradient Descent – Mini-batch and more

NEXT »
Convolutional Neural Networks Tutorial in TensorFlow



COMMENTS

tomas

APRIL 14, 2017 AT 8:28 AM

hi, like the idea of explaining using the simple equation, great idea. I didn't get the tensor/array output could you past all the code. Also the code for the tensorboard visualization would be nice (I know you are planning to go into that in more detail in another tutorial, but would be great to take a look at now.

REPLY

Andy ☆

APRIL 15, 2017 AT 5:51 AM

Hi Tomas – no problems, you can find the code here : https://github.com/adventuresinML/adventures-in-ml-code. I've put another link to this repository in the article to make it clearer. Thanks for the feedback

REPLY

Bablofil

APRIL 19, 2017 AT 3:19 AM

Th-nks, great article.

EPLY

Lwebzem

MAY 5, 2017 AT 1:16 AM

ed the code from this post and it worked instantly. This is a
at article and great code so I added the link to the collection
eural networks with python.

EPLY

dtdzung

JULY 17, 2017 AT 1:02 PM

great article. Thank you

REPLY

xxx

JULY 21, 2017 AT 10:21 AM

Asking questions are really fastidious thing if you are not understanding anything completely, but this piece of writing presents nice understanding yet.

REPLY

Lucian

JULY 28, 2017 AT 11:10 AM

Hi

Great tutorial, one of the (few..) best explained on the web.

I have a question: it is possible to give an image path to the model so it can recognize the content of the image (a number in this case) and print accuracy ?

edy have a Tensorflow model which predict given numbers
ed on MNIST) but it fails a bit. I would like to print the
uracy or, better, use a model like this with TF deeply
grated to predict these numbers.

nk you

EPLY

Andy ☆

JULY 28, 2017 AT 7:44 PM

Hi Lucian, thanks for the comment. I'm sorry, I'm not quite sure what you mean by image path? The code given here does predict the MNIST numbers and prints the accuracy. Are you asking whether there is a more accurate deep learning model to predict numbers and other image content? If so, there is – a convolutional neural network. Check out this post to learn how to implement in TensorFlow: Convolutional Neural Networks Tutorial in TensorFlow

I hope this helps

REPLY

John McDonald

AUGUST 10, 2017 AT 10:38 PM

Shouldn't
a=d*e in the 1st paragraph breakdown? Not a=d*c

REPLY

Andy ☆

AUGUST 11, 2017 AT 7:22 PM

Hi John, yes it should – thanks for picking this up. I've fixed it

REPLY

John McDonald ☆

AUGUST 12, 2017 AT 12:58 PM

No problem – I initially thought I might have missed a new way to break down functions!!

REPLY

Pasindu Tennage

AUGUST 17, 2017 AT 5:17 PM

nk you very much for posting this. Very informative. Keep up good work 🙂

EPLY

Leave a Reply

Your email address will not be published.

Comment

Name*

Email*

Website

POST COMMENT

contain Udemy affiliate
links