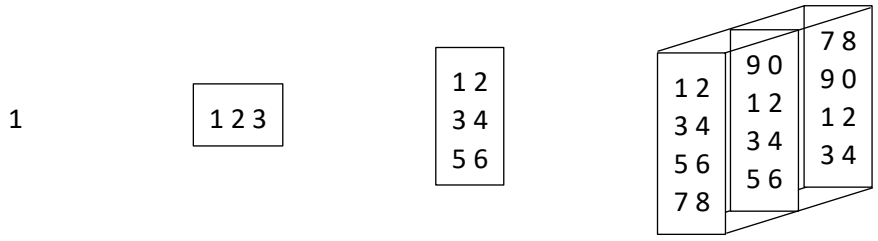
Topic 01- Introduction to Tensorflow

Tensorflow is a Python library (also available for other languages) for performing symbolic mathematical operations, which means that you create a graph structure that represents a number of connected mathematical operations that can be manipulated to do things like differentiation. It can also automatically evaluate the graph structure on the GPU for you, which makes it faster. Tensorflow makes use of NumPy and has similar operations to it as well so being familiar to NumPy is useful.

To install Tensorflow, follow the instructions here. These notes cover Tensorflow v1. Only install the GPU version if you have a modern NVIDIA graphic card with CUDA drivers installed. If you have a graphic card but it is not modern then you will need to pick the right version of Tensorflow according to your CUDA Compute Capability (a capability of at least 3.5 is required for anything above version 1.4, unless you build from source) and you need to choose the right CUDA driver version to install based on the Tensorflow version you install.

At the time of writing, Tensorflow v1 generates a lot of warnings to prepare us for writing Tensorflow v2 code. For this reason, we add warning suppression code at the top of each script. Feel free to comment out the second and fourth lines to see the warnings.

It is important to know what a tensor is: a structure of numbers organised as a regular grid with a certain number of dimensions. A 0-dimensional tensor is called a scalar, a 1-dimensional tensor is called a vector, a 2-dimensional tensor is called a matrix, a 3-dimensional tensor is called a 3-tensor.



Scalar	Vector	Matrix	3-tensor
0D number	1D list	2D rectangle	3D cuboid
shape: []	shape: [3]	shape: [3,2]	shape: [3,4,2]

Just like in NumPy, in Tensorflow, you specify a tensor by giving its shape and data type. The shape is a list of integers, with the empty list being for scalars (normal numbers), a list with a single number being a vector of the given number of elements, a list with two numbers being a matrix with the given number of rows and columns, etc. The datatype is usually either int32 for whole numbers and float32 for floating point numbers.

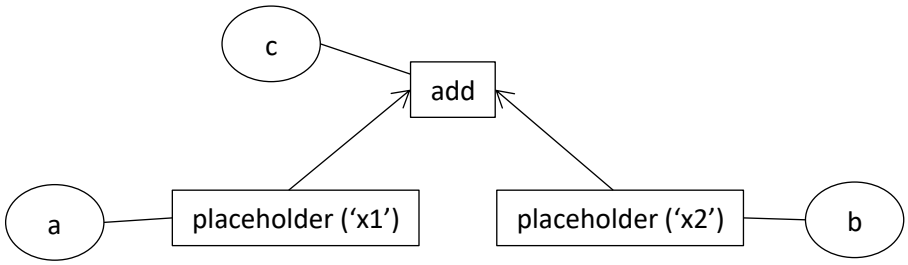
1. Graph and session

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/01_-_Graph_and_session.py

Every time you call almost any function in Tensorflow, nodes get added to a graph structure which acts as a mathematical expression. You can evaluate this expression by running the graph through a session object that takes in a set of inputs that are passed to the graph and returns a set of outputs from the

graph. Each graph node can be referenced by a Python variable to make it easier to say which node should connect to which other node.

In the given script, a graph is created that consists of two placeholder nodes and an operator node connecting them together. A placeholder is a node that represents an input in the graph. The operator node in this case is an addition operator which adds together the two inputs. Here is the resulting Tensorflow graph with nodes as rectangles and Python variables as ellipses.



Note that the Python variables are not part of the graph. For example, you can delete the Python variable without affecting the graph. The Python variables are only used for convenience.

Note also that the value returned by the session is always a NumPy array. You should technically also supply the inputs in the form of a NumPy array but Tensorflow conveniently converts whatever value you supply automatically.

You can visualise a Tensorflow graph using a program called Tensorboard that is downloaded when you install Tensorflow. After running the given script, the function 'tf.summary.FileWriter' will save a file with information about the graph in a file created in the same folder with the script. This file can be opened in TensorBoard by running 'tensorboard --logdir=<path to folder with script>' in the terminal and then opening the URL shown in the terminal (do this after running the program).

2. Tensor operations

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/02_-_Tensor_operations.py

There are several things you can do with a tensor in Tensorflow. Here are some useful operations:

- You can get the shape of a tensor using "tf.shape" (the same thing you set in the placeholder, but will be useful later).
- You can get a sub-tensor of a tensor by using indexing notation just like in NumPy. If "x" is a matrix then:
    - "x[0, 0]" gives you the first column of the first row (top left element)
    - "x[0, :]" gives you the whole first row
    - "x[:, 0]" gives you the whole first column
    - "x[:2, 1:3]" gives you the first two rows with the second and third columns

- You can concatenate two tensors together using “tf.concat”. The tensors to concatenate must be given in a list and you also need to specify along which axis you want to concatenate them (one on top of the other or side by side?).
- You can reshape a tensor using “tf.reshape” so that it has the same number of elements but arranged into a different shape such as a 2 by 3 matrix being rearranged into a six element vector.
- You can tile a tensor using “tf.tile” so that its rows or columns are replicated along their dimension (such as duplicating a matrix’s rows under each other). You need to provide a list of how many times to replicate each axis such as “[2, 1]” meaning that you want to replicate the first dimension (the rows) twice but leave the second dimension (the columns) as is.

3. Tensor expressions

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/03_-_Tensor_expressions.py

You can compute expressions that work on tensors of any dimension instead of scalars. This allows you to perform several computations concurrently which, if you’re using a GPU, will be faster than performing each computation separately. Adding a scalar to a vector will add the scalar to each element of the vector for example.

Note that the equation in the code is using an explicit conversion of Python constants into Tensorflow constants. We could instead let it happen implicitly (automatically) by writing ‘2*a + 1’ but there would still be a conversion.

4. Getting/setting intermediate nodes

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/04_-_Getting-setting_intermediate_nodes.py

You can set a value for any intermediate node, not just placeholders, and you can also get the value of any intermediate node, not just the final value in the expression.

5. Variably sized tensors

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/05_-_Variably_sized_tensors.py

Although tensors cannot change their size, there is no need to specify what their size is when creating them. Setting a dimension size to ‘None’ will allow you to use any number of elements in that dimension when passing a tensor as input. Any other expressions that depend on that tensor will change their size accordingly at run time. Note that this is why it is useful to use the “tf.shape” function. Also note that just because the dimension size is unspecified doesn’t mean that you can use rows with different amounts of columns for example as you must still supply a valid tensor.

6. Variables

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/06_-_Variables.py

It is usually the case in machine learning that we need to use a model with parameters that are adjusted and learned so that we get a desired behaviour, that is, how inputs are mapped into outputs. In Tensorflow these parameters are called variables. Variables are not passed in with the inputs but are stored within the session (not the graph). They are not the variables in the sense of Python's variables but are special nodes in the graph. There are also special nodes for setting variables to some value such as to a placeholder.

7. Initialisers

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/07_-_Initialisers.py

Variables must be initialised to some value before they can be used. Rather than setting each value separately, you can use a global initialiser node which initialises all values at once. Different variables can be initialised using different methods by setting their initializer parameter.

8. Variable names and scopes

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/08_-_Variable_names_and_scopes.py

Graph variables must be given unique names or you'll get an error. In order to avoid coming up with long names we can organize them into variable scopes which act like folders. A variable in one scope can have the same name as another variable in another scope.

It is possible to get a variable from a graph by querying its name using the 'get_tensor_by_name' function. Every variable name ends with a ':0' such as 'varname:0'. If the variable is in a scope then its name will be 'scopename/varname:0'. If the scope is nested within another scope then its name will be 'scopename1/scopename2/varname:0'.

9. Saving and restoring variables

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/09_-_Saving_and_restoring_variables.py

You can save and load variables for later use by using a 'saver' graph node. Remember that variable values are session specific, so closing a session will destroy the variable values. If you want to use the same variable values in a different session you will need to either manually set them or load them from a saved model. The saver node is useful for publishing or loading a trained model or even for saving your half-trained model's variables in order to resume training if the program crashes or the power goes out.

10. Polynomials

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/10_-_Polynomials.py

In machine learning we do not typically deal with expressions but with models. A model is a function that has learnable parameters which determine the behaviour of the function. The given script shows an example of a model that is a polynomial, where the parameters are the coefficients of the polynomial.

11. Encapsulating the graph

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/11_-_Encapsulating_the_graph.py

Up to now, we have been putting all the code of the graph in the same program scope as the rest of the code. This is not ideal, as later on it will become messy to avoid variable names from conflicting. One way to solve this problem is to put the graph inside a class. We can simply put the graph definition code in the constructor of the class. This class would represent the model and can encapsulate things like session handling, getting outputs from the model, saving its parameters, and so on.

Note that when separating the session from the rest of the code, we can no longer use a 'with' block for the session as session handling is going to be hidden away inside a function. Instead, we can call the 'close' function of the session when we are done. This is closer to what you would expect to use in an application that loads the model once during initialisation and then uses the model in several different locations.

Also note that as long as the session is created within the 'with' block of a graph, it is bound to that graph and will always work on that graph, even if other graphs are created later. In fact, you can make the constructor build the graph differently based on parameters it receives and then use several models at once. For example, you might have a number of coefficients parameter or just load variables from different files for each model.

12. Plotting with matplotlib

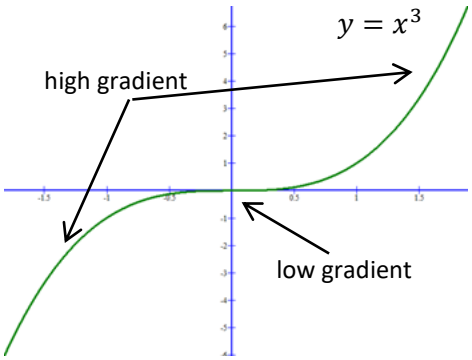
https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/12_-_Plotting_with_matplotlib.py

matplotlib is a very useful library for plotting and visualising numbers. In this script, we will be using it to plot a polynomial. We will be making extensive use of matplotlib throughout these notes.

13. Gradients

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/01_-_Introduction_to_Tensorflow/13_-_Gradients.py

The gradient is the rate of change in a functions output with respect to its input. For some points in the input, a function's output might increase or decrease by a lot whilst for others it might remain flat.



Gradients are very useful in machine learning as they help you find the lowest point in a function or even the most sensitive input in a function (a high gradient means that a small change in the input will give a big change in the output). Tensorflow has functions that automatically find the gradient of a graph. You pass in the input and get the gradient at that point in the graph expression. Not only that, but the gradient is computed analytically rather than numerically, that is, Tensorflow builds an expression that computes the gradient of your graph at any point and then adds the expression to your graph. This lets you find the gradient of the gradient which is useful for more complex forms of machine learning.

In Tensorflow, we can only find the gradient of a scalar. It can be with respect to any shape tensor but the node we are finding the gradient of must be a single number. For example, if you have a graph whose output is a vector, then you cannot find the gradient of the entire vector with respect to the input, although you can find the gradient of a single element in the vector or of the sum of elements in the vector. The gradient of an entire vector is called a Jacobian matrix, which is not possible in Tensorflow. If you do attempt to find the gradient of a non-scalar, the gradient function will find the gradient of the sum of all elements in the tensor.