

Tensorflow and Keras

CSE 4311 – Neural Networks and Deep Learning

Vassilis Athitsos

Computer Science and Engineering Department

University of Texas at Arlington

Pros and Cons of Using a Library

- Many machine learning methods are already implemented in programming libraries.
- A common question is: which library is the “best”.
- The answer depends a lot on the context.
- When studying and trying to understand a method, sometimes the answer is “no library”.
 - It is important to do some implementations from scratch.
 - In such an implementation, every single detail must be explicitly addressed.
 - Oftentimes, implementations expose issues that we have not yet fully understood, and that we need to revisit.
- Using a library can boost productivity, minimize coding time.
- Library code can also be optimized to run very efficiently.

Semester Plan

- Assignment 3 is about implementation from scratch.
- There are many people using neural networks daily who have never done such an implementation.
- It is not an easy assignment, but it helps demystify the training process.
- Subsequent assignments are about neural network variations.
 - Implementing those variations from scratch has little added value over assignment 3.
 - Implementing them using a library allows us to implement more methods during the semester.
- So, starting with assignment 4, we will be using the Keras library.

Keras, Tensorflow, PyTorch

- Tensorflow is a Python programming library for implementing neural network.
- Keras is an additional library, that is implemented on top of Tensorflow.
- Keras provides a more simple, high-level interface than Tensorflow.
- Tensorflow allows you to do more things “under the hood”, if you do not want to simply implement standard “off-the-shelf” methods.
- PyTorch is an alternative to Tensorflow and Keras for implementing neural networks, that is also commonly used.
 - We will not be covering PyTorch this semester, but hopefully it should be easier to learn once you are familiar with Keras.

Installing Keras on Your Computer

- Install Anaconda
- Open Anaconda Prompt or Anaconda Power Shell
- Type:

```
conda install -c anaconda keras
```

- You can use the Spyder editor (comes with Anaconda) to write code, test and debug.
- Some people recommend that you create an Anaconda “environment” and install Keras within the environment.
 - That is useful for computers where Keras is installed multiple times, possibly by different users, and Keras is combined in multiple ways with different other packages.
 - For this class, if you use your own computer, this is not needed.

Running Keras Online

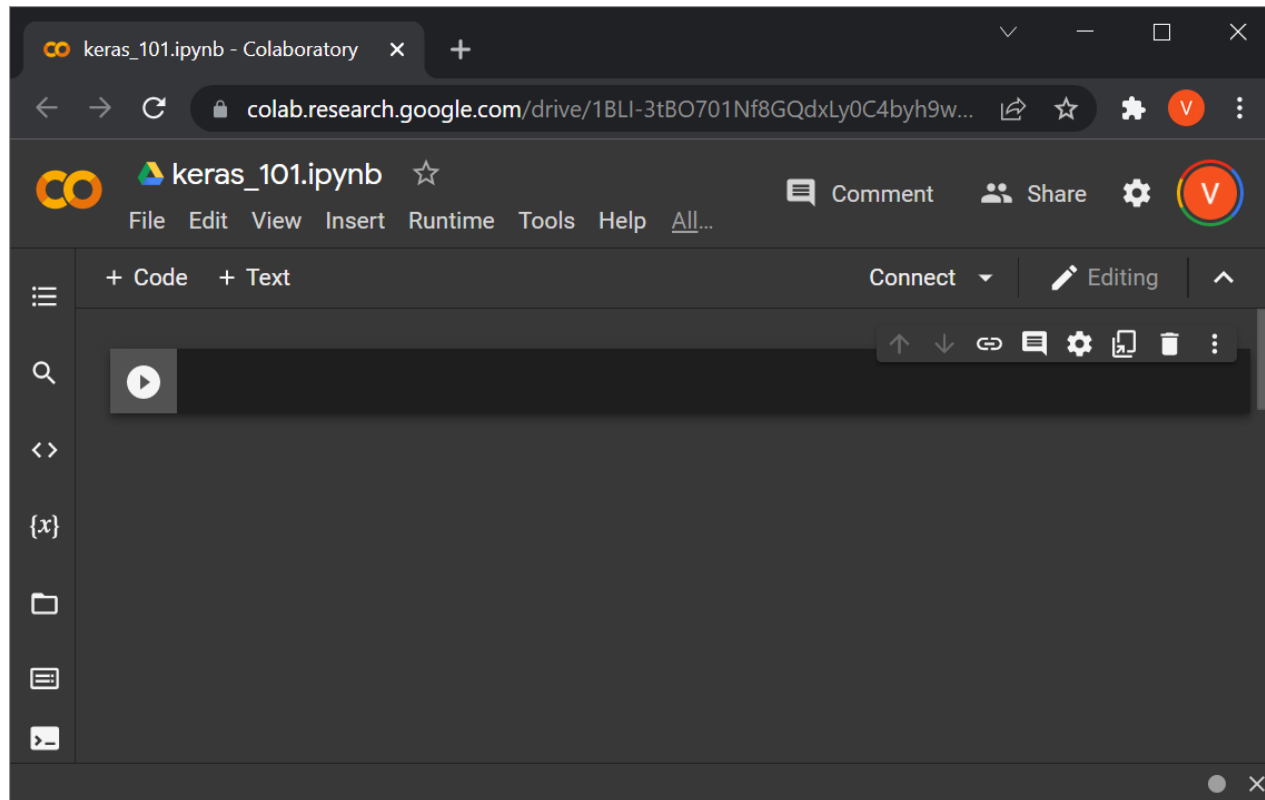
- Google Colab is an online environment where you can write and run Python and Keras code.

<https://colab.research.google.com/>

- Keras is already installed, so this is very convenient.
- It is a useful job skill to get familiar with Colab.
 - Easy to pick up, if you do everything else you need to do in this class.
- Given that there are many different Python libraries with many different versions, that may not be backward compatible, using Colab ensures that we can run your code.
- Most of the times, the code I write on Anaconda runs on Colab with no problems.

Colab Basics

- Go to Google Colab, sign in with your Google Account.
- Go to File->New notebook
- Give a name to your notebook, like keras_101.ipynb.



If Using Colab, Some Prep

- On Google Drive, I created a top-level folder called cse4392.
- In that folder, I copied:
 - uci_data.py
 - This is posted on the website, under this lecture, and contains some helper code.
 - The uci_datasets directory with all the text files.
- Then, on Google Colab:
 - Create and run a cell with this code:

```
from google.colab import drive
drive.mount('/content/drive')
```

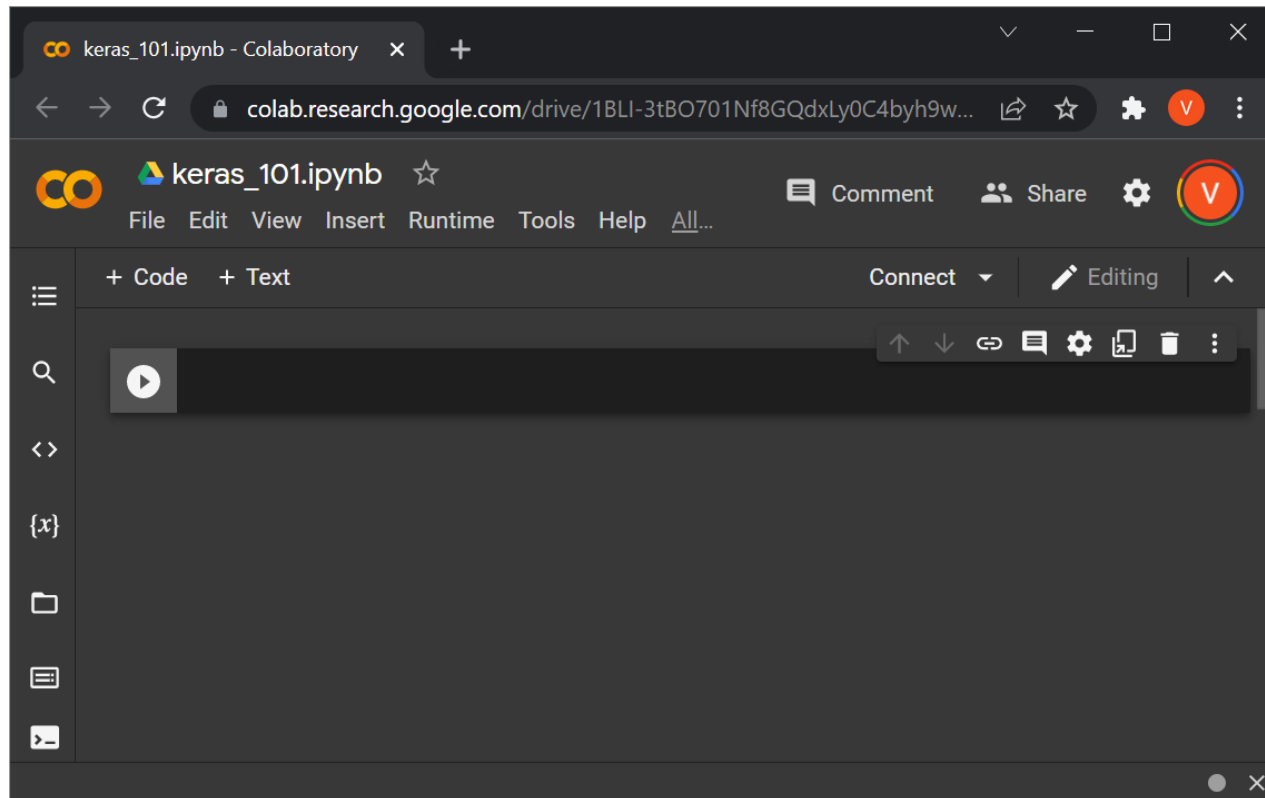
- Create and run another cell with this code:

```
cd /content/drive/MyDrive/cse4392
```

- The next slides show these steps in more detail.

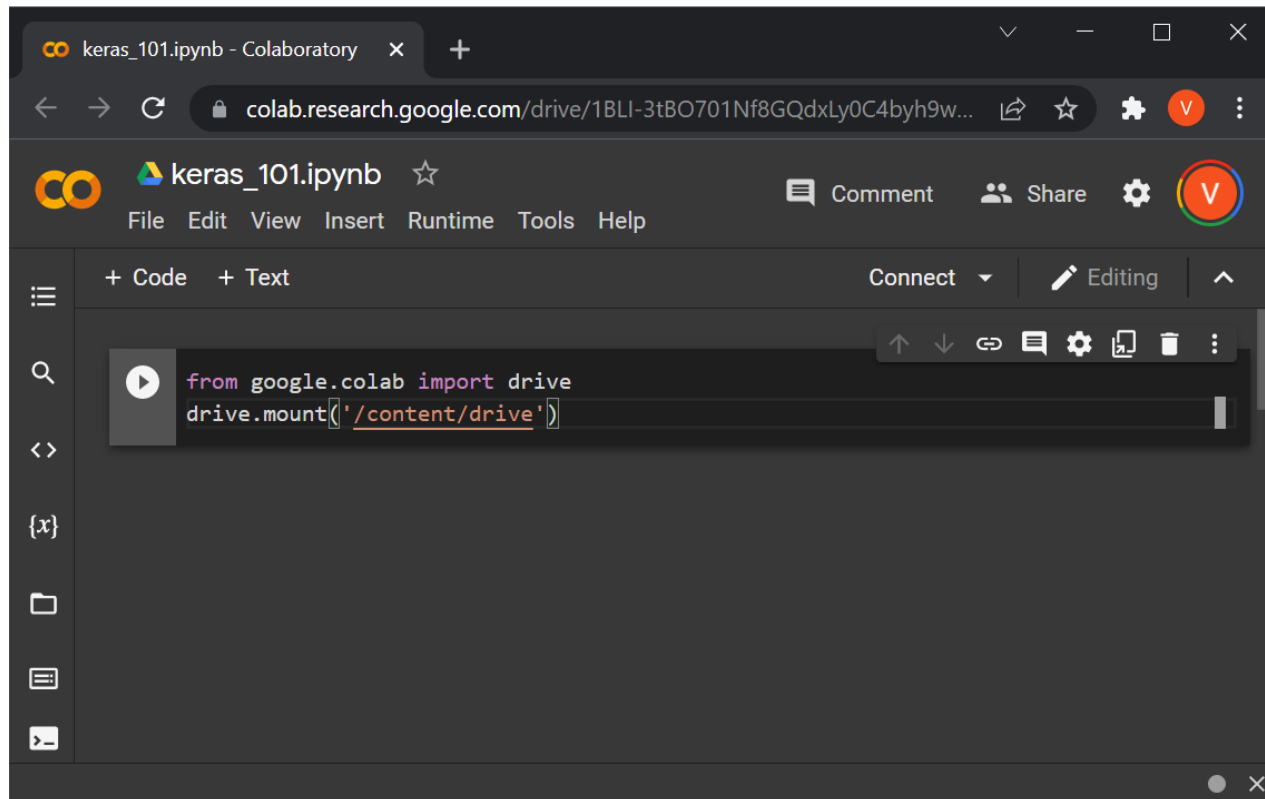
Code Cells in Colab

- To create a new code cell in Colab, click the “+ Code” button.
- To run the code on that cell, click the “play” button to the left of the cell, or press SHIFT+ENTER.



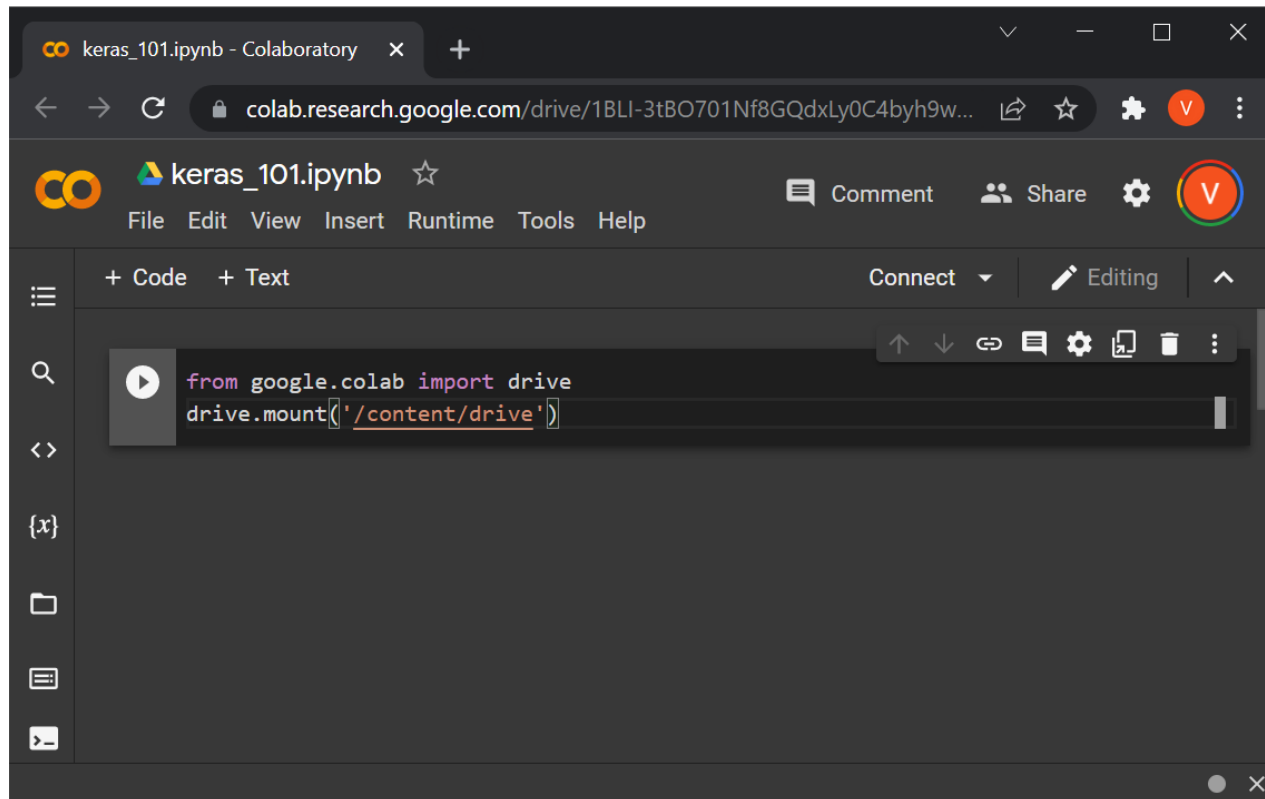
Pasting Code to Colab

- Pasting code to Colab from Powerpoint does not work for me.
- First I paste from Powerpoint to a Google Drive text file.
- Then I copy-paste from the Google Drive file to the Colab notebook.



Mounting Google Drive

- Put in the code that you see, and run the cell.
- You may get this prompt “Permit this notebook to access your Google Drive files?” If so, click “Connect to Google Drive”. You get a couple more prompts to connect to your account and give permissions to Google Drive.



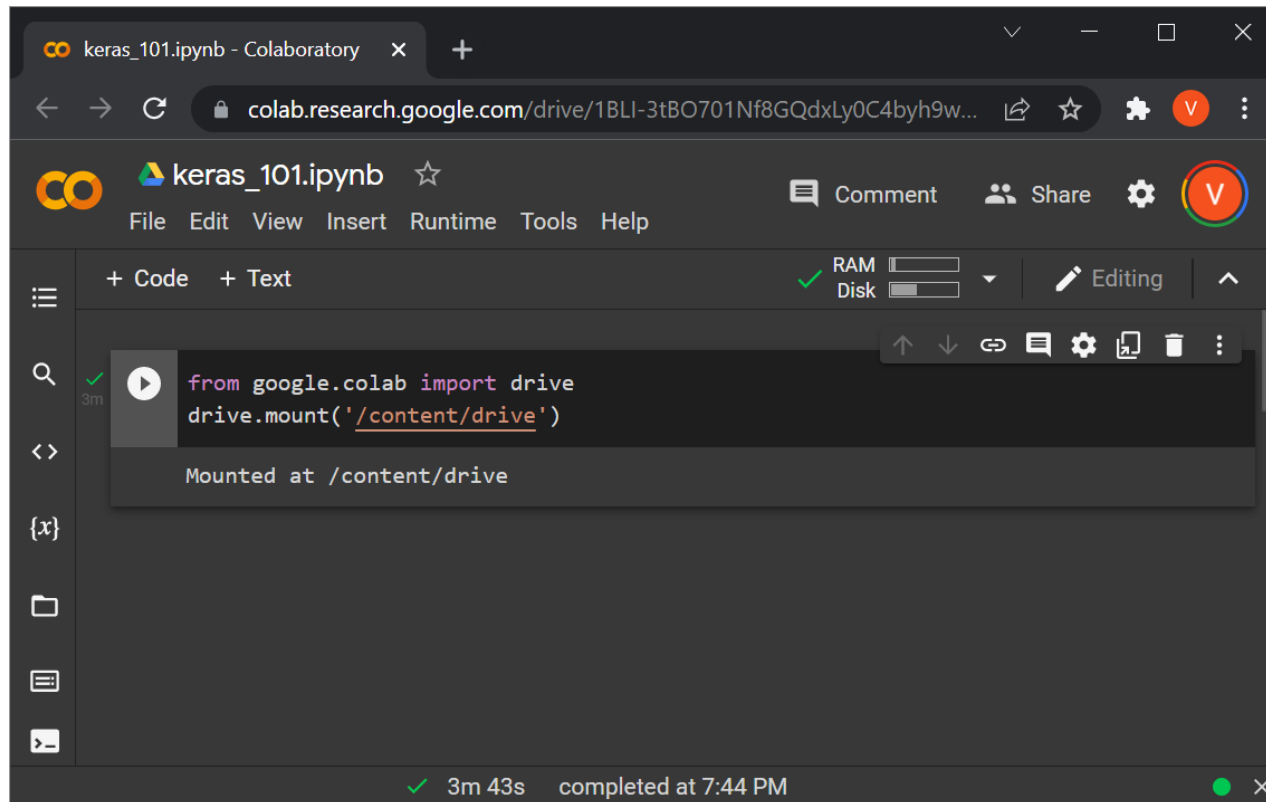
The screenshot shows a Google Colaboratory notebook titled 'keras_101.ipynb'. The browser address bar displays 'colab.research.google.com/drive/1BLI-3tBO701Nf8GQdxLy0C4byh9w...'. The notebook interface includes a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar, there are buttons for '+ Code' and '+ Text', and a 'Connect' dropdown menu. The main code cell contains the following Python code:

```
from google.colab import drive
drive.mount('/content/drive')
```

The code cell is in 'Editing' mode, as indicated by the 'Editing' button in the top right corner of the code editor. The left sidebar shows a file explorer with a folder icon and a list of files.

Mounting Google Drive

- Once you deal with all the prompts, you get a confirmation that the drive is mounted.



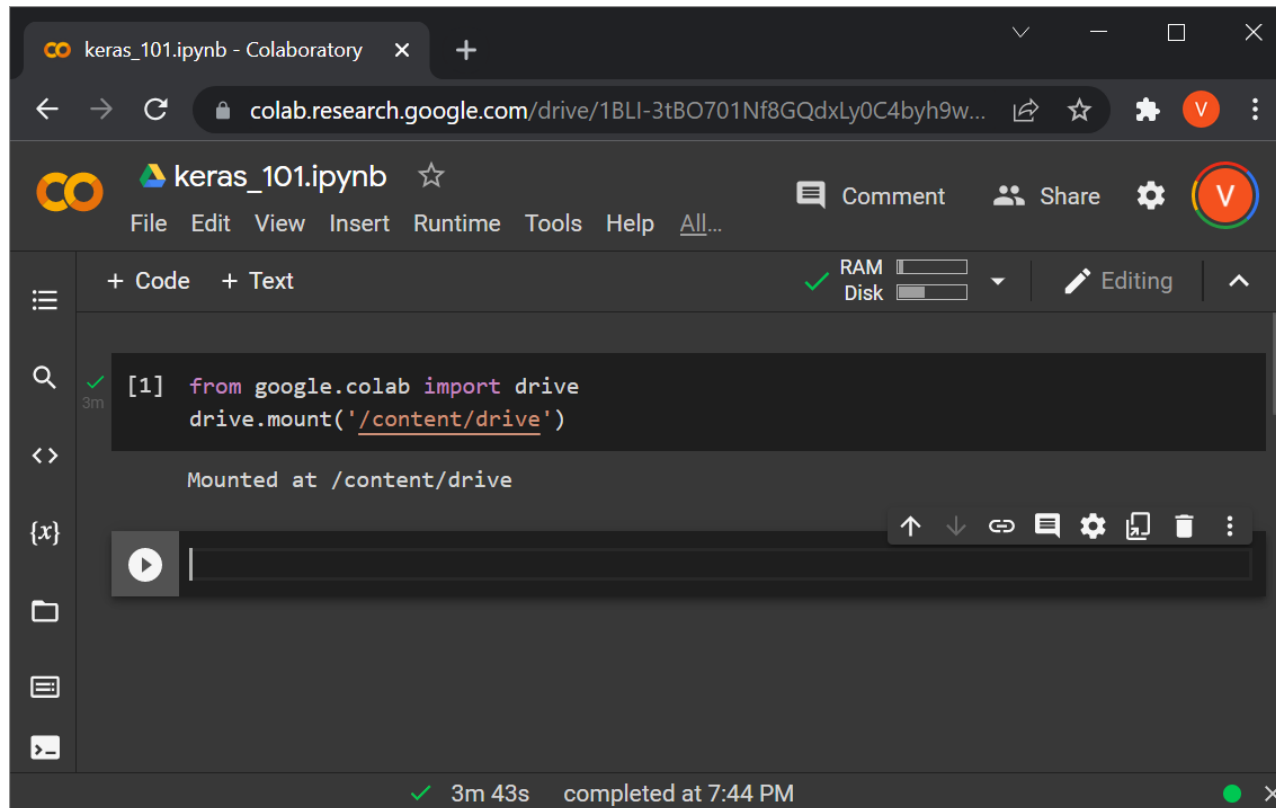
The screenshot shows the Google Colaboratory web interface. The browser tab is titled 'keras_101.ipynb - Colaboratory'. The address bar shows the URL 'colab.research.google.com/drive/1BLI-3tBO701Nf8GQdxLy0C4byh9w...'. The interface includes a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar, there are buttons for '+ Code' and '+ Text'. On the right side of the toolbar, there are indicators for 'RAM' and 'Disk' usage, a 'Comment' button, a 'Share' button, and a user profile icon. The main code editor area contains the following Python code:

```
from google.colab import drive
drive.mount('/content/drive')
```

Below the code, a confirmation message is displayed: 'Mounted at /content/drive'. The status bar at the bottom of the interface shows a green checkmark, the text '3m 43s', and 'completed at 7:44 PM'.

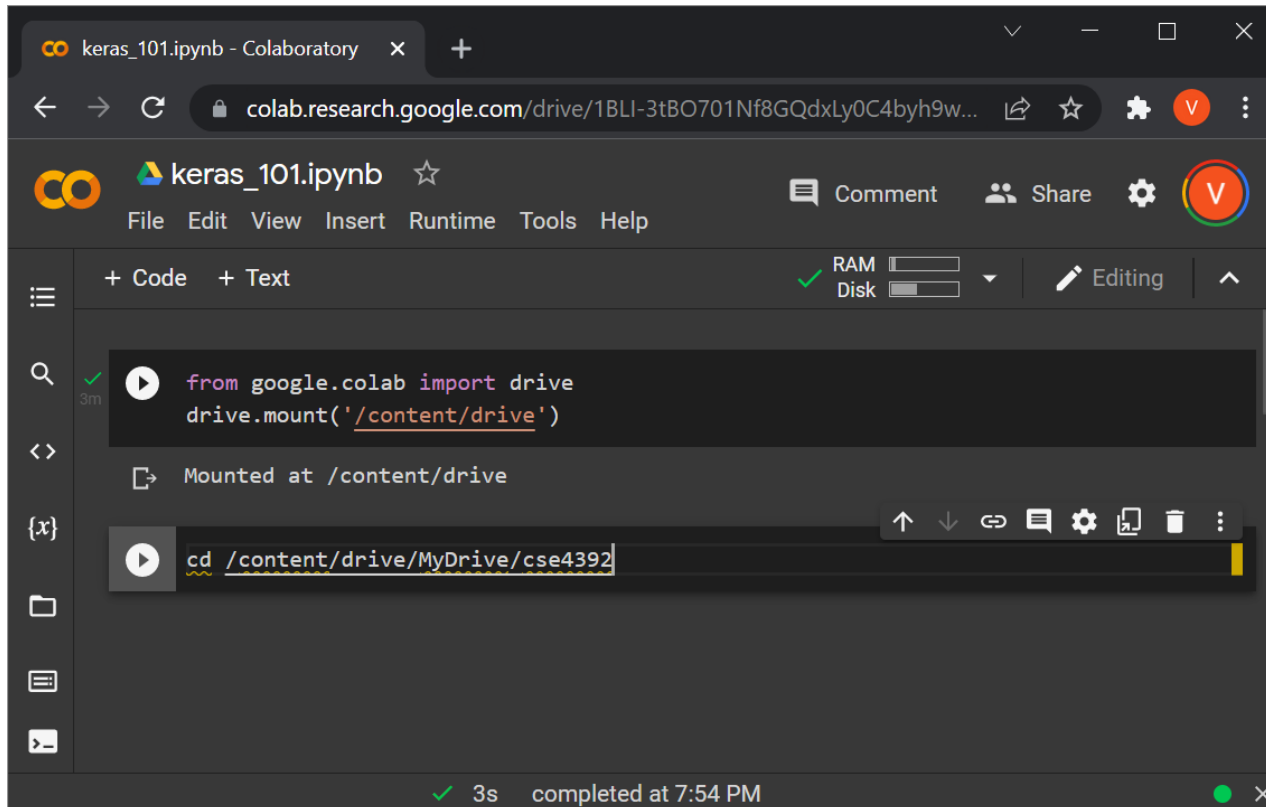
Mounting Google Drive

- Click on “+ Code” again, to add a new cell.



Mounting Google Drive

- In the new cell, type (or paste) the code that you see.



The screenshot shows a Google Colaboratory notebook interface. The browser address bar displays the URL `colab.research.google.com/drive/1BLI-3tBO701Nf8GQdxLy0C4byh9w...`. The notebook title is `keras_101.ipynb`. The menu bar includes `File`, `Edit`, `View`, `Insert`, `Runtime`, `Tools`, and `Help`. The toolbar shows `+ Code` and `+ Text` buttons, along with RAM and Disk usage indicators. The code editor contains two cells. The first cell, executed 3 minutes ago, contains the code `from google.colab import drive` and `drive.mount('/content/drive')`, with a status message `Mounted at /content/drive`. The second cell contains the command `cd /content/drive/MyDrive/cse4392`. The status bar at the bottom indicates the notebook is completed at 7:54 PM.

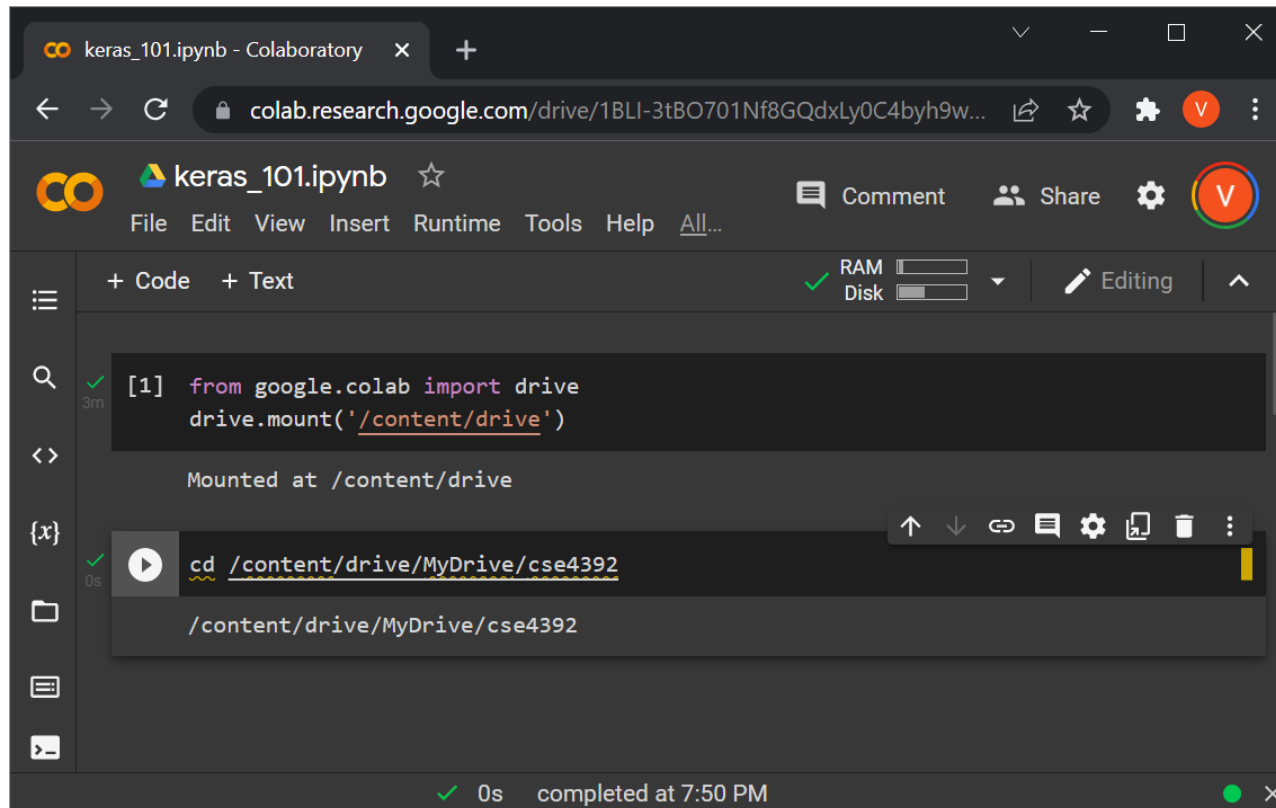
```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
cd /content/drive/MyDrive/cse4392
```

Mounting Google Drive

- Execute the new cell.
- You get a confirmation that the working directory has changed.



The screenshot shows the Google Colaboratory web interface. The browser tab is titled 'keras_101.ipynb - Colaboratory'. The address bar shows the URL 'colab.research.google.com/drive/1BLI-3tBO701Nf8GQdxLy0C4byh9w...'. The Colab logo and file name 'keras_101.ipynb' are in the top left. A menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. On the right, there are buttons for 'Comment', 'Share', and a settings icon. Below the menu bar, there are tabs for '+ Code' and '+ Text'. A status bar shows 'RAM' and 'Disk' usage with progress bars, and a 'V' icon in a red circle. The main code area contains two cells. The first cell is a code cell with the following code:

```
[1] from google.colab import drive
drive.mount('/content/drive')
```

 Below the code, it says 'Mounted at /content/drive'. The second cell is a code cell with the following code:

```
cd /content/drive/MyDrive/cse4392
```

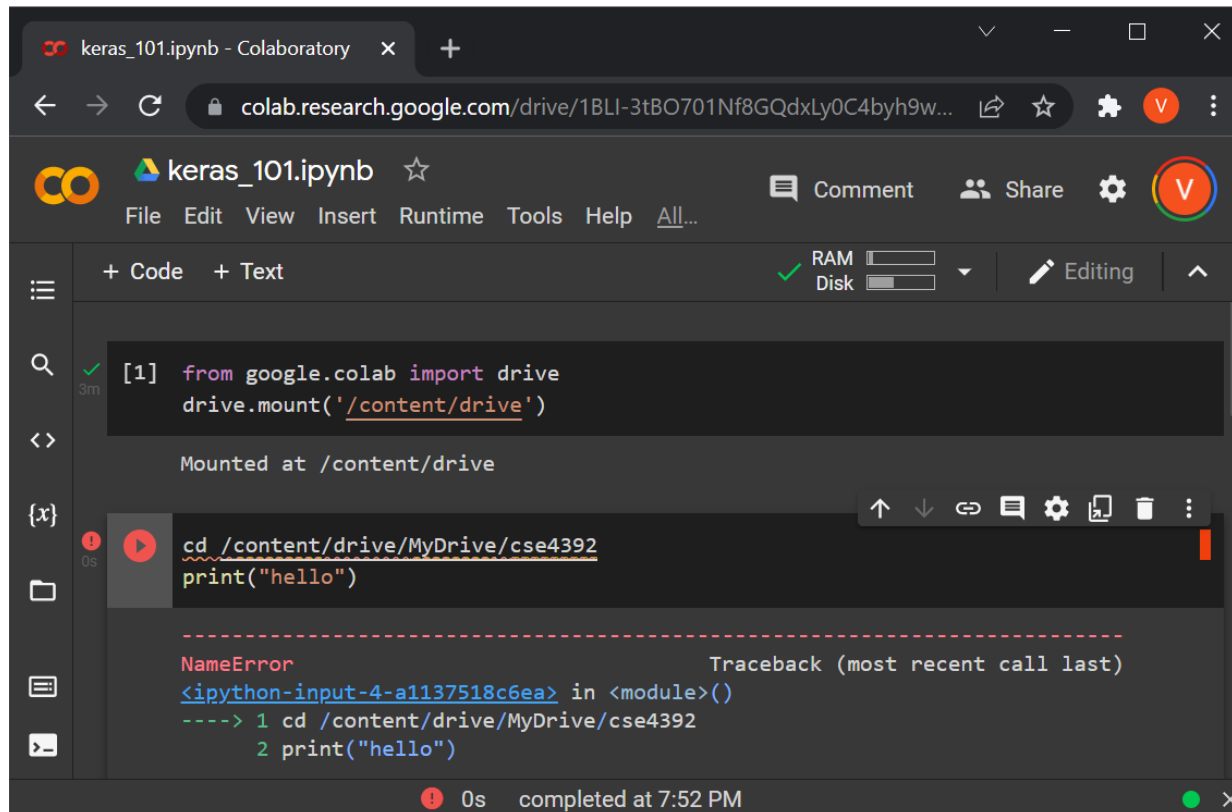
 Below the code, it shows the current directory:

```
/content/drive/MyDrive/cse4392
```

. The bottom status bar shows a green checkmark, '0s', and 'completed at 7:50 PM'.

Mounting Google Drive

- For some weird reason, the `cd` command only works (for me) when it is the ONLY line in the cell.
- Here is how it fails otherwise:



The screenshot shows a Google Colaboratory notebook interface. The top bar indicates the notebook is named 'keras_101.ipynb'. The left sidebar shows a file explorer with a folder icon. The main area contains two code cells. The first cell, marked with a green checkmark and '3m', contains the code `from google.colab import drive` and `drive.mount('/content/drive')`. Below the code, it says 'Mounted at /content/drive'. The second cell, marked with a red exclamation mark and '0s', contains the code `cd /content/drive/MyDrive/cse4392` and `print("hello")`. Below the code, a traceback is shown for a `NameError`. The traceback indicates the error occurred in the second line of the cell: `2 print("hello")`. The bottom status bar shows '0s' and 'completed at 7:52 PM'.

```
[1] from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

cd /content/drive/MyDrive/cse4392
print("hello")

-----
NameError                                Traceback (most recent call last)
<ipython-input-4-a1137518c6ea> in <module>()
----> 1 cd /content/drive/MyDrive/cse4392
      2 print("hello")
```


Continuing with Rest of Code

- Now you can create new cells and put in whatever code you like.

The screenshot shows a Google Colaboratory notebook titled 'keras_101.ipynb'. The interface includes a top navigation bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help' menus. Below the navigation bar, there are tabs for '+ Code' and '+ Text'. The notebook contains three code cells:

```
[1] from google.colab import drive
    drive.mount('/content/drive')

Mounted at /content/drive
```

```
[5] cd /content/drive/MyDrive/cse4392

/content/drive/MyDrive/cse4392
```

```
import tensorflow as tf
import numpy as np
print(tf.__version__)
from uci_data import *
```

The output of the third cell is '2.7.0'. The status bar at the bottom indicates '3s completed at 7:54 PM'.

Keras: A First Example

```
import tensorflow as tf
import numpy as np
from uci_data import *

(training_set, test_set) = read_uci1("uci_datasets", "pendigits")
(training_inputs, training_labels) = training_set
(test_inputs, test_labels) = test_set

input_shape = training_inputs[0].shape
number_of_classes = np.max([np.max(training_labels), np.max(test_labels)]) + 1

model = tf.keras.Sequential([
    tf.keras.Input(shape = input_shape),
    tf.keras.layers.Dense(number_of_classes, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])

model.fit(training_inputs, training_labels, epochs=10)

test_loss, test_acc = model.evaluate(test_inputs, test_labels, verbose=0)
print('\nTest accuracy: %.2f%%' % (test_acc * 100))
```

- This is a small Keras example.
- It trains and tests a neural network on the pendigits dataset.
- In the next slides we will see what each part does.

Imports

```
import tensorflow as tf  
import numpy as np  
from uci_data import *
```

- We will have to import tensorflow and numpy for pretty much all our code from now on.
- uci_data.py is my code, with some auxiliary functions.
 - It is posted on the class website, under this lecture.

Loading a Dataset

```
(training_set, test_set) = read_uci1("uci_datasets", "pendigits")  
(training_inputs, training_labels) = training_set  
(test_inputs, test_labels) = test_set
```

- No tensorflow or Keras code is here yet.
- We are just loading the pendigits dataset, using my `read_uci1` helper function, defined at `uci_data.py`.
- `training_inputs` is a numpy array
 - `training_inputs.shape` gives the dimensions of the array, which is `(7494, 16)`
 - 7494 rows (number of training inputs)
 - 16 columns (number of attributes)

Loading a Dataset

```
(training_set, test_set) = read_uci1("uci_datasets", "pendigits")  
(training_inputs, training_labels) = training_set  
(test_inputs, test_labels) = test_set
```

- training_labels is a 7494x1 2D array.
 - It could also be a 1D array of 7494 elements, I tested that and it works.
- My code makes sure that class labels have been mapped to consecutive integers starting at 0.
 - The Keras code later assumes that this mapping has been done.

Loading a Dataset

```
(training_set, test_set) = read_uci1("uci_datasets", "pendigits")  
(training_inputs, training_labels) = training_set  
(test_inputs, test_labels) = test_set
```

- test_inputs is a 3498x16 2D array.
 - 3498 rows, which is the number of test objects.
 - 16 columns, which is the number of attributes.
 - Obviously, the number of attributes in the test inputs should match the number of attributes in the training inputs.
- test_labels is a 3498x1 2D array.

Verifying the Array Sizes

- A (very) common source of bugs in Tensorflow and Keras is wrong array dimensions and sizes. For example:
 - Using an $M \times N$ matrix where $N \times M$ is expected.
 - Using an $M \times 1$ 2D matrix where a 1D matrix of size M is expected.
- It is useful to keep track of what array dimensions, and sizes for those dimensions, are expected.
- When testing and debugging, you can use code like this:

```
print(training_inputs.shape)
print(training_labels.shape)
print(test_inputs.shape)
print(test_labels.shape)
```

Output:

```
(7494, 16)
(7494, 1)
(3498, 16)
(3498, 1)
```

Parameters for the Network

```
input_shape = training_inputs[0].shape
number_of_classes = np.max([np.max(training_labels),
                             np.max(test_labels)]) + 1
```

- Variable `input_shape` is the dimensionality of the data.
 - It will be the number of units on the input layer.
- Variable `number_of_classes` is the number of classes.

```
print("input_shape = ", input_shape)
print("number_of_classes = ", number_of_classes)
```

Output:

```
input_shape = (16,)
number_of_classes = 10
```


Creating a 2-Layer Network

```
model = tf.keras.Sequential([  
    tf.keras.Input(shape = input_shape),  
    tf.keras.layers.Dense(number_of_classes, activation='sigmoid')  
])
```

- Here is our first piece of Keras code.
- We are creating a two-layer network.
- Function **tf.keras.Sequential** creates a network that is a sequence of layers.
 - That is the only type of networks we know so far.
- The argument is a list, where every element specifies a layer.

Creating a 2-Layer Network

```
model = tf.keras.Sequential([  
    tf.keras.Input(shape = input_shape),  
    tf.keras.layers.Dense(number_of_classes, activation='sigmoid')  
])
```

- **tf.keras.Input**(shape = input_shape) creates an input layer, and input_shape specifies the number of input units.
- Soon, when we discuss convolutional neural networks, we will see that input_shape can be a list of 2 or 3 numbers, because our inputs will be 2D or 3D arrays representing images.
 - For now, input_shape will be a list of a single number, because our inputs are simple vectors.

Creating a 2-Layer Network

```
model = tf.keras.Sequential([  
    tf.keras.Input(shape = input_shape),  
    tf.keras.layers.Dense(number_of_classes, activation='sigmoid')  
])
```

- **tf.keras.layers.Dense(number_of_classes, activation='sigmoid')** creates a fully connected layer.
 - This is the last layer we specify, so this will be the output layer.
 - We want one output unit for each class, so this is specified by passing `number_of_classes` as first argument.
 - The activation parameter specifies that we will use the sigmoid activation function
 - We will see more options for activation functions in later lectures.

Creating a 2-Layer Network

```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),  
              metrics=['accuracy'])
```

- **model.compile** needs to be called before we train the network.
 - This is one of the Keras-specific “rituals” that we just have to learn and follow.
- The arguments we pass to **model.compile** specify some important hyperparameters of the network.
 - `optimizer='adam'` specifies one specific variant of gradient descent, that we will describe later.

Loss Function

```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),  
              metrics=['accuracy'])
```

- **loss=tf.keras.losses.SparseCategoricalCrossentropy()** specifies that we will use a loss function called “Cross Entropy”.
 - The loss parameter specifies the loss function (which we can also call the “error” function).
 - The cross-entropy loss function is an alternative to the sum-of-squared differences that we have seen in the backpropagation slides.
 - We will study cross entropy, and other loss functions, in a few lectures.

Loss Function

```
model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),  
              metrics=['accuracy'])
```

- `metrics=['accuracy']` makes no difference in training.
 - It simply specifies that we would like to see the classification accuracy on the training set printed after each training round.
 - In general, it is useful to print out some metrics after each training round, to see how much progress is being made.

Loss Function

```
model.fit(training_inputs, training_labels, epochs=10)
```

- **model.fit** is the function that actually trains the network.
- We specify the training inputs and training labels.
 - Obviously, their dimensions must match the input units and output units that we have already specified.
 - All class labels in `training_labels` should be non-negative integers that are smaller than the number of output units.
 - Our previous lines of code made sure of that.
- Parameter `epochs` specifies the number of training rounds.
 - “Epochs” is the term commonly used to specify the number of training rounds in backpropagation.

Output During Training

Epoch 1/10 235/235 [=====] - 0s 955us/step -
loss: 43.5318 - accuracy: 0.1715

Epoch 2/10 235/235 [=====] - 0s 1ms/step - loss:
15.4048 - accuracy: 0.3979

Epoch 3/10 235/235 [=====] - 0s 983us/step -
loss: 8.0585 - accuracy: 0.5701

...

- For every epoch, we see various statistics. Most important for now:
 - loss shows the value of the loss function after each epoch.
 - accuracy shows the classification accuracy on the training set after each epoch.

Testing the Network

```
test_loss, test_acc = model.evaluate(test_inputs, test_labels, verbose=0)
print('\nTest accuracy: %.2f%%' % (test_acc * 100))
```

- **model.evaluate** applies the network on a test set.
- `test_inputs` and `test_labels` specify the test set.
- The `verbose` parameter specifies how much extra output we want (none in our case).
- Output:

Test accuracy: 96.28%

Some Benchmarks

- I tried different setups for running code that trains (10 epochs) and tests a 2-layer network on the pendigits dataset.
- Running the training and test code (**model.fit()** and **model.evaluate()**) on Colab took about 5 seconds.
- The same code, running locally on my computer (with Spyder/Anaconda), took less than two seconds.
- My solution for the backpropagation assignment (which uses numpy, but not Tensorflow or Keras) took about 5 seconds running on my computer.

Side Note, Regarding Assignment 3

- If you use the Keras code from these slides as your solution for assignment 3, it will not be correct.
- Various specs given in the assignment (and our backpropagation slides) do not match what is done in the Keras code we used.
- This is great, because I do NOT want you to use Keras for the assignment.
 - The whole point is to implement backpropagation from scratch.
- At the same time, in terms of computations, both the Keras version and our assignment should be similar.
- The differences in time is due:
 - Optimizations in Keras, making the code run faster.
 - The shared nature of Colab, which makes runtimes less predictable.

Some Benchmarks

- Another experiment: a 4-layer network for the pendigits dataset, with 50 nodes in each hidden unit.
 - Still training for 10 epochs.
- Running the training and test code (**model.fit()** and **model.evaluate()**) on Colab took about 11 seconds.
- The same code, running locally on my computer (with Spyder/Anaconda), took about 3 seconds.
- My solution for the backpropagation assignment (which uses numpy, but not Tensorflow or Keras) took about 8 seconds running on my computer.

GPU vs. CPU Processing

- Note: computers with graphics cards should run the code several times faster than computers without such cards.
 - The computer I used here did NOT have a graphics card.
 - I will try to add some results with a graphics card in the next few days.
- A nice feature of Tensorflow and Keras is that they automatically exploit GPUs, if your system has them.
 - Your code remains the same.

Code for a 4-Layer Network

```
model = tf.keras.Sequential([  
    tf.keras.Input(shape = input_shape),  
    tf.keras.layers.Dense(50, activation='sigmoid'),  
    tf.keras.layers.Dense(50, activation='sigmoid'),  
    tf.keras.layers.Dense(number_of_classes, activation='sigmoid')  
])
```

- To create the 4-layer network used in the previous timing experiments, just call **tf.keras.Sequential()** as shown above.
- It is very straightforward.
 - Each hidden layer has 50 units, is fully connected, and uses the sigmoid activation function.

Evaluating the Network on an Input

- Suppose that we have already trained a 2-layer model on the pendigits dataset, as shown in the previous slides.
- Here we see how to (and how not to) apply the model to a new input vector.
- To do that, we use the **predict()** method.

```
test_index = 10 # nothing special about this value
```

```
input_vector = test_inputs[test_index,:]
```

```
nn_output = model.predict(input_vector)
```

- We would hope that nn_output contains the output of the model (which is a vector of as many dimensions as the number of classes).
- The above is an example of how NOT to do it. We get an error.

Evaluating the Network on an Input

```
test_index = 10 # nothing special about this value
input_vector = test_inputs[test_index,:]
nn_output = model.predict(input_vector)
```

- The code above gives an error.
 - This is a good example of often frustrating errors we get because of incompatible array shapes.
- Problem: `input_vector` is a 1D vector. The `predict()` method wants a 2D vector where each row is an input.
- Fix:

```
test_index = 10 # nothing special about this value
input_vector = test_inputs[test_index,:]
input_vector = np.reshape(input_vector, (1, 16))
nn_output = model.predict(input_vector)
```


Evaluating the Network on an Input

- Fix:

```
test_index = 10 # nothing special about this value
```

```
input_vector = test_inputs[test_index,:]
```

```
input_vector = np.reshape(input_vector, (1, 16))
```

```
nn_output = model.predict(input_vector)
```

- The **numpy.reshape()** method is used to convert input vector from a 1D array of 16 elements to a 1x16 2D array.
- Now **model.predict()** is happy.
- nn_output is a 1x10 2D array.
- If we want to convert nn_output to a 1D array we do:

```
nn_output = nn_output.flatten()
```

From Output to Class Prediction

- As we know, to convert the output to a class prediction, we need to find the argmax.
 - In other words, we find the output unit that gave the highest response.

```
test_index = 10 # nothing special about this value
```

```
input_vector = test_inputs[test_index,:]
```

```
input_vector = np.reshape(input_vector, (1, 16))
```

```
nn_output = model.predict(input_vector)
```

```
nn_output = nn_output.flatten()
```

```
predicted_class = np.argmax(nn_output)
```

```
actual_class = test_labels[test_index]
```

```
print("predicted class label = %d\nactual class label = %d\n" %  
      (predicted_class, actual_class))
```

Output:

predicted class label = 0
actual class label = 7

From Output to Class Prediction

- As we know, to convert the output to a class prediction, we need to find the argmax.
 - In other words, we find the output unit that gave the highest response.

```
test_index = 10 # nothing special about this value
```

```
input_vector = test_inputs[test_index,:]
```

```
input_vector = np.reshape(input_vector, (1, 16))
```

```
nn_output = model.predict(input_vector)
```

```
nn_output = nn_output.flatten()
```

```
predicted_class = np.argmax(nn_output)
```

```
actual_class = test_labels[test_index]
```

```
print("predicted class label = %d\nactual class label = %d\n" %  
      (predicted_class, actual_class))
```

Note the use of **argmax**, it returns the **position** of the maximum value in the array.

Classification Ties

- It can happen that two or more output units tie for the highest output value.
- Your code should be able to detect that.
- It so happens that our 2-layer network produces lots of ties.
 - In a few slides we will see what causes that and how to fix it.
 - For `test_index = 10`, six units tie for the highest value.

```
(indices,) = np.nonzero(nn_output == nn_output[predicted_class])  
print("indices =", indices)
```

Output:

```
indices = [0 3 4 5 6 7]
```

Classification Ties

- It can happen that two or more output units tie for the highest output value.
- Your code should be able to detect that.
- It so happens that our 2-layer network produces lots of ties.
 - In a few slides we will see what causes that and how to fix it.
 - For `test_index = 10`, six units tie for the highest value.

```
(indices,) = np.nonzero(nn_output == nn_output[predicted_class])  
print("indices =", indices)
```

- Numpy trick 1: the expression `my_array == number` returns a boolean numpy array, where the value at position `i` is `True` iff `my_array[i] == number`.

Classification Ties

- It can happen that two or more output units tie for the highest output value.
- Your code should be able to detect that.
- It so happens that our 2-layer network produces lots of ties.
 - In a few slides we will see what causes that and how to fix it.
 - For `test_index = 10`, six units tie for the highest value.

```
(indices,) = np.nonzero(nn_output == nn_output[predicted_class])  
print("indices =", indices)
```

- Numpy trick 2: function call `np.nonzero(my_array)` returns an array of all indices in `my_array` that are not zero.
 - For a boolean array, values of True are non-zero.

Classification Ties

- It can happen that two or more output units tie for the highest output value.
- Your code should be able to detect that.
- It so happens that our 2-layer network produces lots of ties.
 - In a few slides we will see what causes that and how to fix it.
 - For `test_index = 10`, six units tie for the highest value.

```
(indices,) = np.nonzero(nn_output == nn_output[predicted_class])  
print("indices =", indices)
```

- The best way to become familiar with these tricks (and other numpy tricks we'll be introducing) is to try them out yourselves, and make sure you understand what they do.

Classification Accuracy and Ties

- In that example, how accurate was the classification?
- We cannot say 100% accurate, because we did not get a single correct answer.
- We cannot say 100% wrong, because the correct class label, “7”, was one of six class labels that tied for highest value.
- In such cases, it is fair to say that the result was 16.67% accurate.
 - This is 100% divided by the number of classes that tied.
- Obviously, if there are ties but the correct class label is not included in those ties, then the result is 0% accurate.
 - Which means, it is 100% wrong.

Code for Measuring Accuracy

```
predicted_class = np.argmax(nn_output)
actual_class = test_labels[test_index]
print("predicted: %d\nactual: %d\n" % (predicted_class, actual_class))
```

```
(indices,) = np.nonzero(nn_output == nn_output[pred])
print("indices =", indices)
number_of_ties = np.prod(indices.shape)
```

```
if (nn_output[actual_class] == nn_output[predicted_class]):
    accuracy = 1.0 / number_of_ties
else:
    accuracy = 0
```

```
print("accuracy = %.4f" % (accuracy))
```

Output:

```
indices = [0 3 4 5 6 7]
accuracy = 0.1667
```

Normalizing Input Values

- The reason for that many ties is numerical.
- If we print `nn_output` for our example, we see that six values are numerically equal to 1.
- This can happen, if the input to the sigmoid function is too high.
- To avoid such numerical issues, it is good to normalize the input values.
- Without such normalization, input values can be arbitrarily high or low.
- The backpropagation assignment specifies that you should divide all input vectors by the “MAXIMUM ABSOLUTE value over all attributes over all training objects”.
 - Then, all attribute values are between -1 and 1.

Normalizing Input Values

- This code does the required normalization, feel free to use in your assignment:

```
(training_set, test_set) = read_uci1("uci_datasets", "pendigits")
(training_inputs, training_labels) = training_set
(test_inputs, test_labels) = test_set
max_value = np.max(np.abs(training_inputs))
training_inputs = training_inputs / max_value
test_inputs = test_inputs / max_value
```

- Normalizing like that before training the model, my code found no classification ties in any of the 3498 test objects of the pendigit dataset.

Normalizing Input Values

- Normalizing like that, we get no ties in any of the 3498 test objects of the pendigit dataset.

Recap

- We have seen how to use Keras to train and test multi-layer networks.
 - So far we have only used fully-connected layers.
 - So far we have only used the sigmoid activation function.
- Topics for the next few lectures: some additional options for our networks.
 - Different activation functions.
 - Different optimization choices (all variants of gradient descent).
 - Working with images.
 - Convolutional neural networks for images.