That graph data flow introduction and linear regression example is nice & all, but we still haven’t properly covered the main reason for *TensorFlow* existence:

**Artificial Neural Networks**, or **Deep Learning**.
That graph data flow introduction and linear regression example is nice & all, but we still haven’t properly covered the main reason for *TensorFlow* existence:

**Artificial Neural Networks**, or **Deep Learning**.

Neural Networks are mathematical models inspired by how human brain works, where

- the processing is done by **neurons**, and
- signals pass through the neuron only if they exceed a certain threshold.
Mathematical Model of a Neuron.

\[
\sum_{i=1}^{784} (w_i x_i) + b
\]
Artificial Neural Networks.
Linear Neuron Model: Limitations.

If we stick with a **Linear Neuron Model**, where each neuron simply outputs a weighted linear combination of its outputs, what type of function of original inputs $x_1, \ldots, x_n$ would we inevitably get?
If we stick with a **Linear Neuron Model**, where each neuron simply outputs a weighted **linear combination** of its outputs, what type of function of original inputs $x_1, \ldots, x_n$ would we inevitably get? **Linear function.**

And linear functions can only do all but so much when it comes to estimating **non-linearity**.
Introducing capability to estimate non-linear cases are activation functions. Activation function $f$ is being applied to the weighted linear combination of incoming signals, meaning that the function represented by the neuron becomes:

$$f\left(\sum_i w_i x_i + b\right)$$

instead of simply

$$\sum_i w_i x_i + b$$

which is what it was in case of linear neurons.
Now, a model of a neuron with a **ReLU activation function**: 

**Mathematical model**

\[ \sum_{i=1}^{784} (w_i x_i) + b \]

ReLU

Activation
Rotate it, say hi to *TensorFlow*.

Now let’s just rotate the images for models of neuron and artificial neural network models $90^\circ$ counter-clockwise. Reminds you of something?
Rotate it, say hi to *TensorFlow*. 

Now let’s just rotate the images for models of neuron and artificial neural network models 90° counter-clockwise. *Reminds you of something?*
Deep learning is a class of machine learning wherein learning happens on multiple levels of neuron networks.

Deep learning can work on large amounts of labeled and unlabeled data easily and efficiently.
**DL** is a vast subject and is an important concept for building AI. It is used in various applications, such as:

- Image or Object Recognition
- Voice recognition
- Handwriting detection
- Text classification
- Cancer detection
- Fraud detection
Trends on Different Deep Learning Libraries
Deep Learning example: Belgian Traffic Signs.

We will use *TensorFlow* to build a Neural Network that recognizes Belgian Traffic Signs.
Softmax Cross-Entropy Loss.

Here we have 62 distinct labels (road signs), hence 62 potential classes for the input image. We built our network, not counting the flattening layer, with two fully-connected hidden layers:

```python
hidden_layer_1 = tf.contrib.layers.fully_connected(inputs=images_flat,
                                                num_outputs=200,
                                                activation_fn=tf.nn.relu)

logits = tf.contrib.layers.fully_connected(inputs=hidden_layer_1,
                                            num_outputs=62,
                                            activation_fn=tf.nn.relu)
```

Both layers use ReLU activation function at each node:

Each input image, after going through all computational layers of our neural net, ends up being represented by 62 numbers $a_1, \ldots, a_{62}$. 
Softmax Cross-Entropy Loss.

https://deepnotes.io/softmax-crossentropy

Signal $a_i$ from output node $i$, $i = 1, \ldots, 62$, corresponds to how likely is that the input image describes traffic sign $i$. The higher - the likelier.

So, for example, if we got signals $(0, 0, 1.2, 0.2, \ldots, 0, 0)$ with node 3 yielding the highest value $\implies$ neural network classifies the input image as traffic sign 3.

In order to evaluate the quality of predicted classes given the actual labels, we use softmax cross-entropy loss function.
Softmax Cross-Entropy Loss.

Softmax Function.

Softmax function takes an $n$-dimensional vector of real numbers and transforms it into a vector of real numbers $\in (0, 1)$ which add up to 1.

$$p_i = \frac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}$$

It is a "soft" version of max function: Instead of selecting one maximum value, it breaks up the "whole" (1) into portions, with maximal element getting the largest portion of the distribution.

This property of softmax function that it outputs a probability distribution makes it suitable for probabilistic interpretation in classification tasks.

Cross entropy.

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.

$$H(y, p) = - \sum_{i} y_i \log(p_i)$$
TensorFlow also has an implementation of popular Python deep learning library named Keras.

It allows a seamless interface to

- Build layers for deep learning models (`model = keras.Sequential(...)`)  
- Compile the DL model with specified optimizer, loss function and evaluation metrics (`model.compile(optimizer, loss, metrics)`)  
- Training, or fitting, the DL model (`model.fit()`)  
- Evaluating accuracy of the DL model (`model.evaluate()`)  
- Make predictions according to the DL model (`model.predict()`).
## Building the DL model
```python
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

## Compiling the DL model
```python
model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

## Training the DL model.
```python
model.fit(train_images, train_labels, epochs=5)
```

## Evaluating the DL model.
```python
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

## Making predictions with the DL model.
```python
predictions = model.predict(test_images)
```
For a classification example, we will use a so-called Fashion MNIST dataset.

This dataset contains images of 10 different items of clothing, and the task is to train a neural network to recognize the items of clothing.
To cover a regression example - where we aim to predict the output of a continuous value (e.g. a price or temperature) - we will work with Boston House Pricing dataset.

It contains measurements of 14 variables on houses in Boston, and typically of utmost interest is to predict the house’s price according to other 13 variables.

See the corresponding notebook (#5) for details.
Shuffling data is critical in machine learning, especially if one talks about dividing data into training and testing subset.

But it could also be used **purely for training data**, when one wants to break it further into:

- subset of observations **actually used for training**,
- subset of observations used to calculate **validation metric**.
Another critical operation to perform is **feature normalization**. It helps in case we have **features measured in different units**.

**Example.** When performing customer segmentation analysis, one may collect variables:

- "transaction amount" that ranges between $100 and $10000, and
- "# of transactions", ranging from 0 to 30.

Here, a difference of 10$ in transaction amount is not even remotely as important as difference of 10 transactions.
Another critical operation to perform is feature normalization. It helps in case we have features measured in different units.

**Example.** When performing customer segmentation analysis, one may collect variables:

- "transaction amount" that ranges between $100 and $10000, and
- "#of transactions", ranging from 0 to 30.

Here, a difference of 10$ in transaction amount is not even remotely as important as difference of 10 transactions.

But if you were to plot the customers as dots on the 2D map, it won’t be geometrically reflected. Moving a dot 10$ up along $x$-axis or moving it 10 transaction along $y$-axis leads to the same distance this dot travels.
Normalizing Features: Centering and Scaling.

Another critical operation to perform is feature normalization. It helps in case we have features measured in different units.

**Example.** When performing customer segmentation analysis, one may collect variables:
- “transaction amount” that ranges between $100 and $10000, and
- “# of transactions”, ranging from 0 to 30.

Here, a difference of 10$ in transaction amount is not even remotely as important as difference of 10 transactions.

But if you were to plot the customers as dots on the 2D map, it won’t be geometrically reflected. Moving a dot 10$ up along $x$-axis or moving it 10 transaction along $y$-axis leads to the same distance this dot travels.

After you normalize though, this difference will be reflected geometrically. Those distances are critical for certain models, including neural networks.
To wrap it up today, I will just briefly talk about **TensorBoard** - a visualization tool provided by **TensorFlow**.

**TensorBoard** is a *flashlight* for Neural Net’s black box.

It allows one to:

- visualize the computation graph,
- display graphical performance summaries for the model,
- plenty other things (I’m pretty sure..)
To launch TensorBoard, we need to:

1. Write the graph sessions into a file:

   ```python
   # Launch the graph (session)
   with tf.Session() as sess:
       # This creates a 'logs' directory and
       # starts writing files with graph session results in there.
       train_writer = tf.summary.FileWriter(logdir='logs',
                                           graph=sess.graph)
   ```
To launch TensorBoard, we need to:

1. Write the graph sessions into a file:
   ```python
   # Launch the graph (session)
   with tf.Session() as sess:
       # This creates a 'logs' directory and
       # starts writing files with graph session results in there.
       train_writer = tf.summary.FileWriter(logdir='logs',
                                            graph=sess.graph)
   ```

2. Open Anaconda command prompt, proceed to the working directory (that contains the created logs directory) via a sequence of 'cd' calls.
To launch *TensorBoard*, we need to:

1. Write the graph sessions into a file:
   
   ```python
   # Launch the graph (session)
   with tf.Session() as sess:
     # This creates a 'logs' directory and
     # starts writing files with graph session results in there.
     train_writer = tf.summary.FileWriter(logdir='logs',
                                           graph=sess.graph)
   ```

2. Open *Anaconda command prompt*, proceed to the working directory (that contains the created logs directory) via a sequence of 'cd' calls.

3. Type in
   
   ```bash
   > tensorboard --logdir=logs
   ```
TensorBoard: Visualization for TensorFlow.

To launch **TensorBoard**, we need to:

1. Write the graph sessions into a file:
   ```
   # Launch the graph (session)
   with tf.Session() as sess:
       # This creates a 'logs' directory and
       # starts writing files with graph session results in there.
       train_writer = tf.summary.FileWriter(logdir='logs',
                                              graph=sess.graph)
   ```

2. Open **Anaconda command prompt**, proceed to the working directory (that contains the created logs directory) via a sequence of 'cd' calls.

3. Type in
   ```
   > tensorboard --logdir=logs
   ```

4. It gives you an address which you could copy-paste into browser
   ```
   TensorBoard 1.9.0 at http://usdandres-Vostro-3458:6006 (Press CTRL+C to quit)
   ```
   and voila - you’re in **TensorBoard**.
TensorBoard: Visualization for TensorFlow.

Main Graph

Auxiliary Nodes

Add

x

y
**TensorBoard**: Cleaning up the Visualization.

If for a built neural network, e.g. the one we made for Belgian Traffic Signs, we were to simply run *TensorBoard* via the `summary.FileWriter()`, then the graph we get is really dirty. How to clean it up?
TensorBoard: Cleaning up the Visualization.

To clean it up, one can use **variable scope** definitions. Those are done via `tf.name_scope()`, e.g.

```python
with tf.name_scope("Input"):
    x = tf.placeholder(dtype = tf.float32, shape = [None, 28, 28])
    y = tf.placeholder(dtype = tf.int32, shape = [None])
with tf.name_scope("Flat"):
    images_flat = tf.contrib.layers.flatten(x)
with tf.name_scope("Hidden"):
    hidden_layer = tf.contrib.layers.fully_connected(inputs=images_flat,
                                                   num_outputs=200,
                                                   activation_fn=tf.nn.relu)
...

with tf.name_scope("Train"):
    with tf.name_scope("Loss"):
        loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y, logits = logits))
        tf.summary.scalar('loss', loss)
..."
TensorBoard: Cleaning up the Visualization.

The resulting visualization might be far from crystal clear, but at the very least it is less intimidating.
To add visualizations that keep track of loss function value and accuracy progression, use `tf.summary.scalar`, and then `tf.summary.merge_all()` to merge all the summaries into one operation:

```python
with tf.name_scope("Loss"):
    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
                             logits = logits))
    tf.summary.scalar('loss', loss)
with tf.name_scope("Accuracy"):
    ...
    tf.summary.scalar('Accuracy', accuracy)

summary_op = tf.summary.merge_all()

tf.set_random_seed(1234)
sess = tf.Session()

train_writer = tf.summary.FileWriter(logdir='logs', graph=sess.graph)
sess.run(tf.global_variables_initializer())
```
When running the session, make sure to:

1. Run the `summary` operation (here - `summary_op`);
2. Add resulting summaries to file-writer (here - `train_writer`)

for i in range(201):
    _, summary = sess.run([train_op, summary_op], feed_dict={x: images28, y: labels})
    train_writer.add_summary(summary, i) # Write summary logs at each epoch