SuperDataScience

TENSORFLOW 2.0

PRACTICAL ADVANCED

Master Tensorflow 2.0, Google's most powerful Machine Learning Library, with 6 advanced practical projects covering Generative Adversarial Networks (GANs), DeepDream, AutoEncoders, LSTM Recurrent Neural Networks (RNNs), Tensorboard, Transfer Learning with TF Hub and TF Serving

By Dr. Ryan Ahmed, Ph.D., MBA
SuperDataScience Team

1. WHAT'S NEW IN TENSORFLOW 2.0 COMPARED TO TF 1.0?

- TensorFlow now enables eager execution by default which means that operations can be evaluated immediately.
- Eager execution means that we can now interact with TF 2.0 line by line in Google Colab or Jupyter notebook without the need to define a graph and run sessions and all the complexity that came with TensorFlow 1.0.

1.1 EAGER EXECUTION IS ENABLED BY DEFAULT IN TF 2.0

A. Adding two variables in TF 1.0 was a headache (luckily, not anymore)!

 # Install and import TensorFlow 1.0

```
!pip install tensorflow-gpu==1.13.01
import tensorflow as tf
```

 # First we have a "construction phase" where we build a graph

```
>> x = tf.Variable(3)
>> y = tf.Variable(5)
```

```
# Tensorflow created a graph but did not execute the graph yet so a session is needed to run the graph

    >> z = tf.add(x,y)

# “Execution phase” where we run a session and this makes it super difficult to debug and develop models

    >> with tf.Session() as sess:
    >> sess.run(tf.global_variables_initializer())
    # initialize all variables
    >> z = sess.run(z) # run the session
    >> print("The sum of x and y is:", z) # we now get the expected answer, i.e: 8
```

B. Adding two variables in TF 2.0 is easier than ever!

```
# Install and import TensorFlow 2.0

    >> !pip install tensorflow-gpu==2.0.0.alpha0
    >> import tensorflow as tf

# TensorFlow 2.0 still works with graphs but enable eager execution by default

# Let's add the same variables together

    >> x = tf.Variable(3)
    >> y = tf.Variable(5)
    >> z = tf.add(x,y) # immediate answer!
    >> print("The sum of x and y is:", z)
    # we get the answer immediately!
```

1.2 KERAS IS THE DEFAULT API (NOW EASIER THAN EVER TO TRAIN AND DEBUG MODELS)

- The second important feature in TF 2.0 is the use of keras as the high level API by default

- Keras is extremely easy to work with since Keras syntax is very pythonic

- Let's build a mini artificial neural network that can classify fashion images using keras API using couple of lines of code.

 # Import TensorFlow and load dataset

```
>> import tensorflow as tf
>> fashion_mnist = tf.keras.datasets.fashion_mnist
>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```

 # Build, compile and fit the model to training data

```
>> model = tf.keras.Sequential([
   tf.keras.layers.Flatten(input_shape=(28, 28)),
   tf.keras.layers.Dense(128, activation=tf.nn.relu),
   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
>> model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
>> model.fit(train_images, train_labels, epochs=5)
```

1.3 EASILY LAUNCH TENSORBOARD

- Tensorboard enable us to track the network progress such as accuracy and loss throughout various epochs along with the graph showing various layers of the network.

- TensorBoard provides a built-in performance dashboard that can be used to track device placement and help minimize bottlenecks during model execution and training.

- Here's how to launch Tensorboard:

```
>> %load_ext tensorboard
>> fashion_mnist = tf.keras.datasets.fashion_mnist

>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

>> model = tf.keras.Sequential([
   tf.keras.layers.Flatten(input_shape=(28, 28)),
   tf.keras.layers.Dense(128, activation=tf.nn.relu),
   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

>> model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

>> log_dir="logs/fit/" +
datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
>> tensorboard_callback =
tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)
>> model.fit(train_images, train_labels, epochs=5, callbacks =
[tensorboard_callback])
>> %tensorboard --logdir logs/fit
```

1.4 DISTRIBUTED STRATEGY

- Tensorflow enables distributed strategy which allows developers to develop the model once and then decide how they want to run it later; over multiple GPUs or TPUs.

- This will dramatically improve the computational efficiency with just two additional lines of code

```
>> !pip install tensorflow-gpu==2.0.0.alpha0
>> fashion_mnist = tf.keras.datasets.fashion_mnist
```

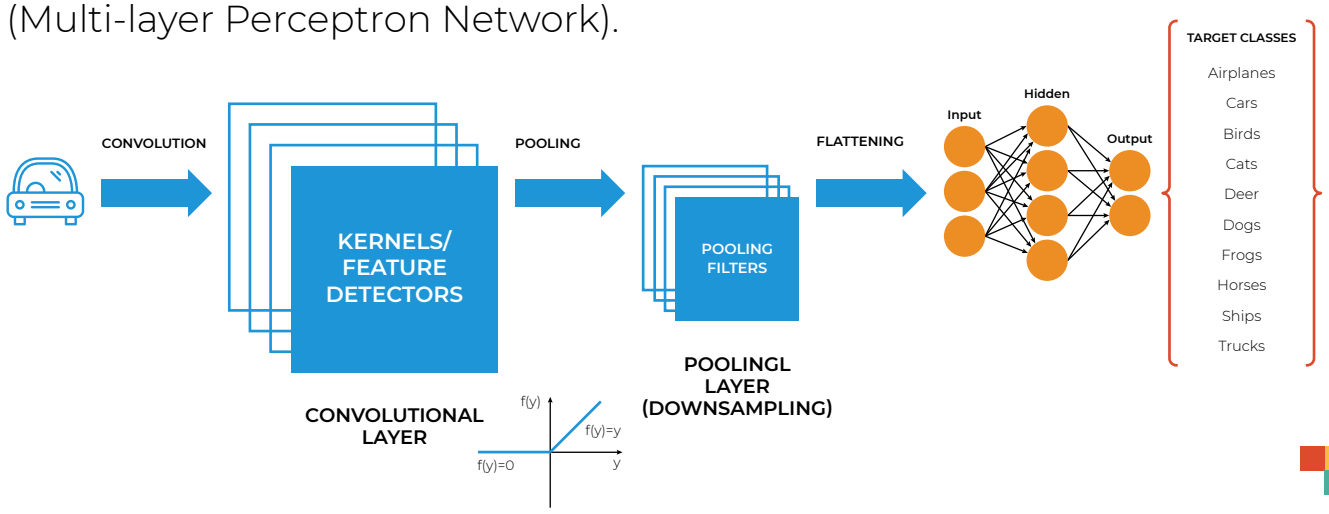
```
>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
>> strategy = tf.distribute.MirroredStrategy()

>> with strategy.scope():
>>  model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
    ])

  >> model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
```

2. HOW TO BUILD A CONVOLUTIONAL NEURAL NETWORK (CNN) IN TF 2.0?

2.1 CONCEPT

CNNs are a type of deep neural networks that are commonly used for image classification.

CNNs are formed of (1) Convolutional Layers (Kernels and feature detectors), (2) Activation Functions (RELU), (3) Pooling Layers (Max Pooling or Average Pooling), and (4) Fully Connected Layers (Multi-layer Perceptron Network).



2.2 BUILD A DEEP CONVOLUTIONAL NEURAL NETWORKS IN TF 2.0:

```
>> cnn = tf.keras.Sequential()

>> cnn.add(tf.keras.layers.Conv2D(32, (3,3), activation = 'relu',
input_shape = (32,32,3)))
>> cnn.add(tf.keras.layers.Conv2D(32, (3,3), activation = 'relu'))
>> cnn.add(tf.keras.layers.MaxPooling2D(2,2))
>> cnn.add(tf.keras.layers.Dropout(0.3))

>> cnn.add(tf.keras.layers.Conv2D(64, (3,3), activation = 'relu'))
>> cnn.add(tf.keras.layers.Conv2D(64, (3,3), activation = 'relu'))
>> cnn.add(tf.keras.layers.MaxPooling2D(2,2))
>> cnn.add(tf.keras.layers.Dropout(0.3))

>> cnn.add(tf.keras.layers.Flatten())

>> cnn.add(tf.keras.layers.Dense(1024, activation = 'relu'))
>> cnn.add(tf.keras.layers.Dropout(0.3))

>> cnn.add(tf.keras.layers.Dense(1024, activation = 'relu'))

>> cnn.add(tf.keras.layers.Dense(10, activation = 'softmax'))
>> cnn.summary()
```

3. TRANSFER LEARNING

3.1 CONCEPT

- Transfer learning is a machine learning technique in which a network that has been trained to perform a specific task is being reused (repurposed) as a starting point for another similar task.

- Transfer learning is widely used since starting from a pre-trained models can dramatically reduce the computational time required if training is performed from scratch.

- In transfer learning, **a base (reference)** Artificial Neural Network on a base dataset and function is being trained. Then, this trained network weights are then **repurposed in a second ANN** to be trained on a new dataset and function.

- Transfer Learning Strategy #1 Steps:

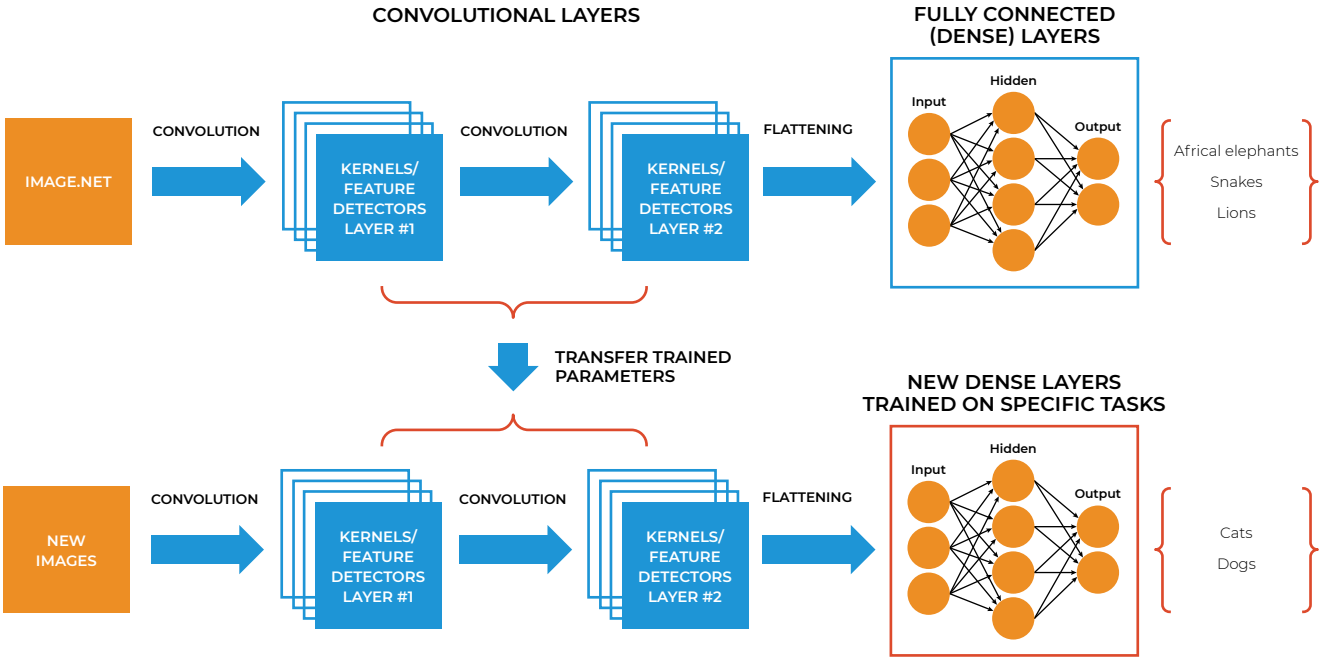
    1. Freeze the trained CNN network weights from the first layers.

    2. Only train the newly added dense layers (with randomly initialized weights).

- Transfer Learning Strategy #2 Steps:

    1. Initialize the CNN network with the pre-trained weights

    2. Retrain the entire CNN network while setting the learning rate to be very small, this is critical to ensure that you do not aggressively change the trained weights.



3.2 APPLY TRANSFER LEARNING IN TF 2.0

Download the pre-trained model (MobileNet) using tensorflow 2.0 Hub

```
>> MobileNet_feature_extractor_url =
"https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vect
or/2" #@param {type:"string"}
>> MobileNet_feature_extractor_layer =
hub.KerasLayer(MobileNet_feature_extractor_url,
input_shape=(224, 224, 3))

>> MobileNet_feature_extractor_layer.trainable = False
# Freeze the layers
```

Create a new model consisting of pre-trained model (MobileNet) with a classifier

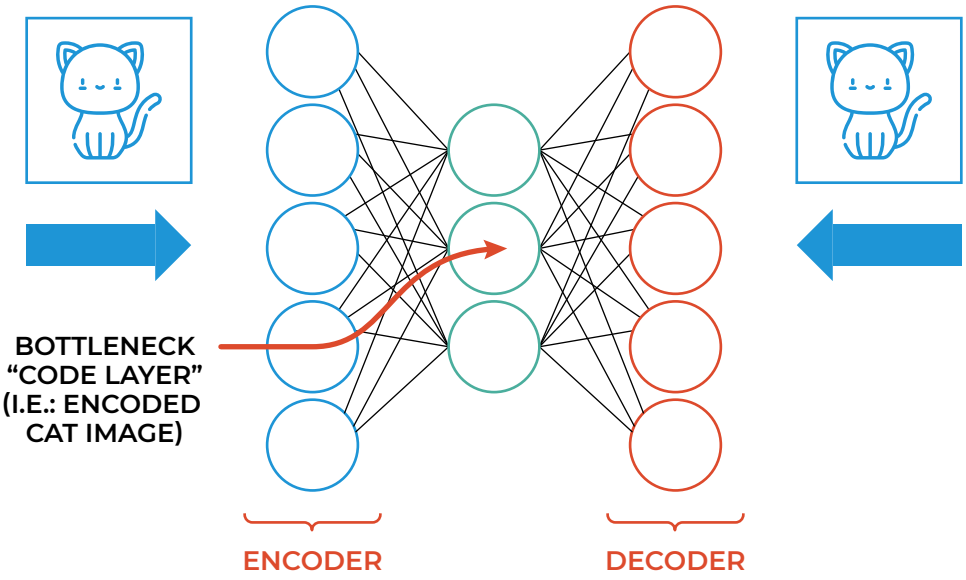
```
>> model = tf.keras.Sequential([
  MobileNet_feature_extractor_layer,
  tf.keras.layers.Dense(flowers_data.num_classes,
activation='softmax')
])

>> model.compile(optimizer=tf.keras.optimizers.Adam(),
loss='categorical_crossentropy', metrics=['accuracy'])

>> history = model.fit_generator(flowers_data, epochs=5)
```

4. AUTOENCODERS

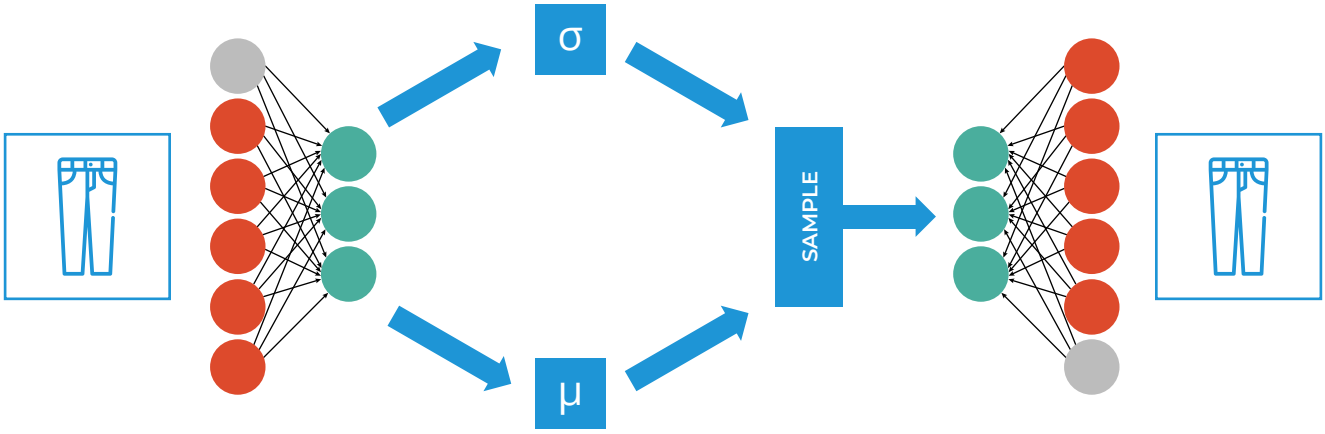
4.1 CONCEPT

- Auto encoders are a type of Artificial Neural Networks that are used to perform a task of data encoding (representation learning).

- Auto encoders use the same input data as an input and output to the model.
- Auto encoders work by adding a bottleneck in the network which forces the network to create a compressed (encoded) version of the original input.
- Auto encoders work well if correlations exists between input data (performs poorly if the all input data is independent).



- Variational Auto Encoders (VARs) have a continuous latent space by default which make them super powerful in generating new images.
- In VARs, the encoder does not generate a vector of size n but it generates two vectors instead as follows: (1) Vector mean μ and (2) standard deviations σ
- Then the decoder can start sampling from this distribution



4.2 AUTOENCODERS IN TF 2.0

```
>> autoencoder = tf.keras.models.Sequential()
```

Let's build the encoder CNN

```
>> autoencoder.add(tf.keras.layers.Conv2D(16, (3,3), strides=1,
padding="same", input_shape=(28, 28, 1)))
>> autoencoder.add(tf.keras.layers.MaxPooling2D((2,2),
padding="same"))

>> autoencoder.add(tf.keras.layers.Conv2D(8, (3,3), strides=1,
padding="same"))
>> autoencoder.add(tf.keras.layers.MaxPooling2D((2,2),
padding="same"))
```

Encoded image (code layer)

```
>> autoencoder.add(tf.keras.layers.Conv2D(8, (3,3), strides=1,
padding="same"))
```

Let's build the decoder CNN

```
>> autoencoder.add(tf.keras.layers.UpSampling2D((2, 2)))
>> autoencoder.add(tf.keras.layers.Conv2DTranspose(8,(3,3),
strides=1, padding="same"))

>> autoencoder.add(tf.keras.layers.UpSampling2D((2, 2)))
```

```
>> autoencoder.add(tf.keras.layers.Conv2DTranspose(1, (3,3),
strides=1, activation='sigmoid', padding="same"))
```

```
# Compile and fit the model
```

```
>> autoencoder.compile(loss='binary_crossentropy',
optimizer=tf.keras.optimizers.Adam(lr=0.001))
>> autoencoder.summary()

>> autoencoder.fit(X_train_noisy.reshape(-1, 28, 28, 1),
X_train.reshape(-1, 28, 28, 1), epochs=10,
        batch_size=200)
```

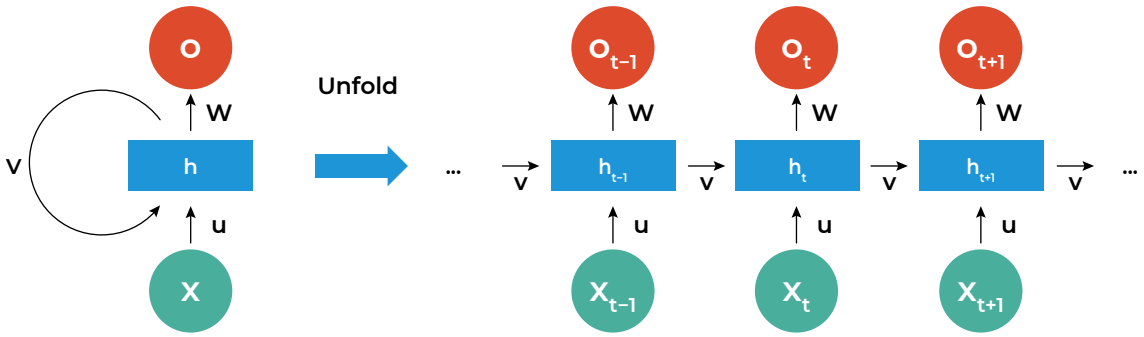
5. RECURRENT NEURAL NETWORKS (RNNS)

5.1 RNN CONCEPT

- RNNs are a type of ANN designed to take temporal dimension into consideration by having a memory (internal state) (feedback loop).
- A RNN contains a temporal loop in which the hidden layer not only gives an output but it feeds itself as well.
- RNN can recall what happened in the previous time stamp so it works great with sequence of text.
- A RNN accepts an input x and generate an output o.
- The output o does not depend on the input x alone, however, it depends on the entire history of the inputs that have been fed to the network in previous time steps.
- Two equations that govern the RNN are as follows:

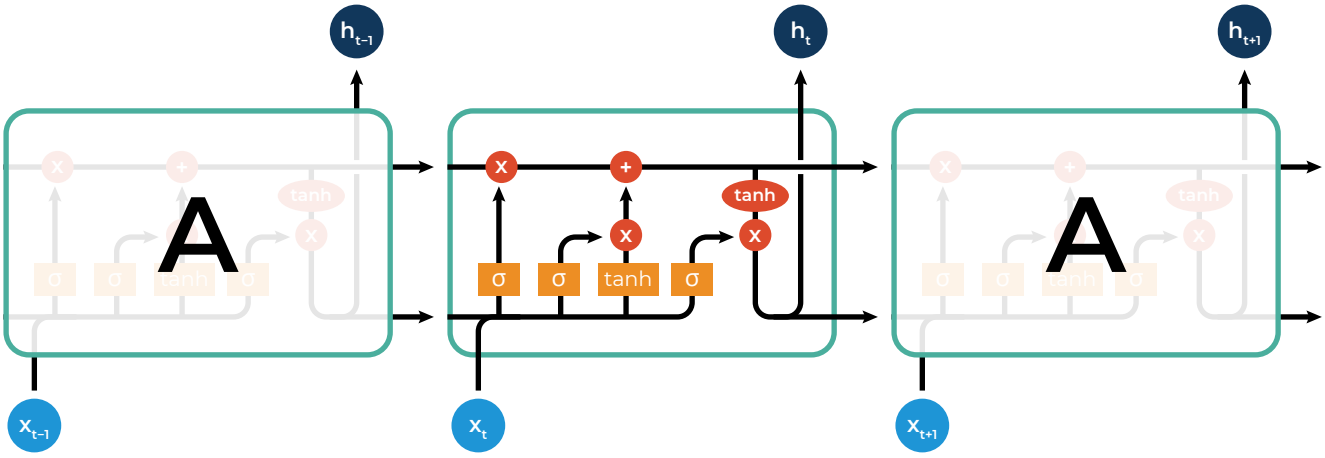
Internal state update: $h_t = \tanh(X_t * U + h_{t-1} * V)$

Output update: $o_t = \text{softmax}(W * h_t)$

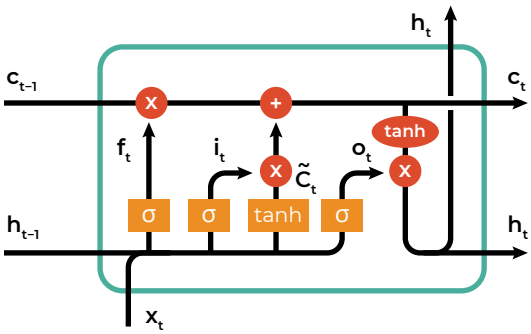


5.2 LONG SHORT TERM MEMORY (LSTM) CONCEPT

- LSTM networks work better compared to vanilla RNN since they overcome vanishing gradient problem.
- LSTM networks are type of RNN that are designed to remember long term dependencies by default.
- LSTM can remember and recall information for a prolonged period of time.



- LSTM equations:



$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$\tilde{C}_t = \tanh(W_c.[h_{t-1}, x_t] + b_c$$
$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

5.3 LSTM IN TF 2.0

```
>> model = tf.keras.Sequential()
>> model.add(tf.keras.layers.Embedding(vocab_size,
embed_size, input_shape=(X_train.shape[1],))
# Add an embedding layer
>> model.add(tf.keras.layers.LSTM(units=128, activation='tanh'))
# add an LSTM network
>> model.add(tf.keras.layers.Dense(units=1,
activation='sigmoid'))
>> model.compile(optimizer='rmsprop',
loss='binary_crossentropy', metrics=['accuracy'])
>> model.summary()
```

6. DEEPDREAM

6.1 CONCEPT (PLEASE REFER TO COURSE MATERIAL TO ACCES THE GOOGLE COLAB CODE)

- LSTM networks work better compared to vanilla RNN since they overcome vanishing gradient problem.
- The algorithm works by creating dream-like effect.
- Deep Dream Steps:

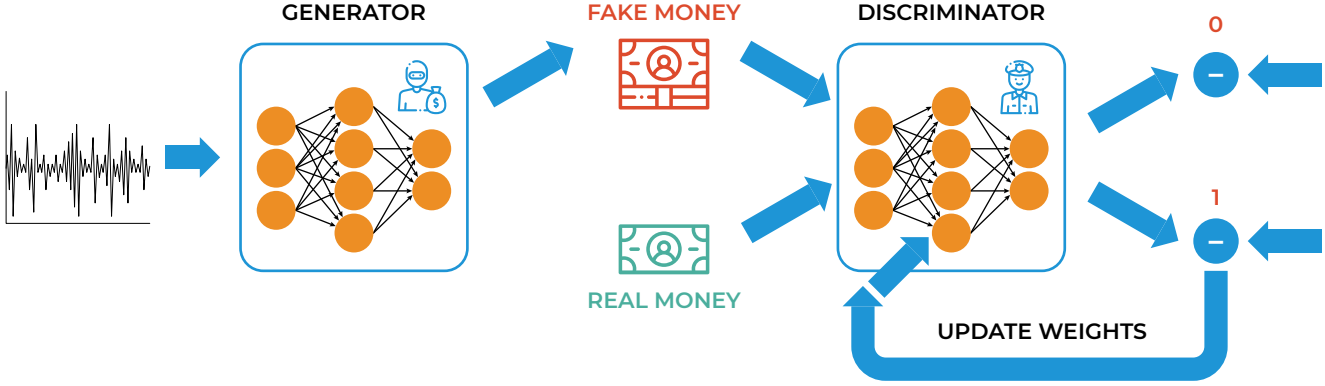
    1. Forward an image through a trained ANN, CNN, ResNet..etc
    2. Select a layer of choice (first layers capture edges and deep layers capture full shapes such as faces)
    3. Calculate the activations (output) coming out from the layer of interest.
    4. Calculate gradient of loss (activations) with respect to the pixels of the input image.

5. Modify the image to increase these activations, and thus enhance the patterns seen by the network resulting in trippy hallucinated image!

7. GENERATIVE ADVERSARIAL NETWORKS (GANS)

6.1 GANS CONCEPT (PLEASE REFER TO COURSE MATERIAL TO ACCES THE GOOGLE COLAB CODE)

- GANs work by having a generator network (counterfeiter) who is being trained to create fake dollars that are indistinguishable from the real ones (generated by the bank).
- The discriminator network (police) is being trained to determine if the money is real or fake.
- The counterfeiter is trying to fool the police by pretending that he generated a real dollar bill.
- But, the discriminator will detect the fake money and provide feedback to the generator on why does he think that the money is fake.
- Overtime, the generator will become expert in generating new money that are indistinguishable from the real ones and the discriminator will fail to tell the difference.



8. TENSORFLOW SERVING

8.1 SAVE THE TRAINED MODEL

```
>> import tempfile # Obtain a temporary storage directory
>> MODEL_DIR = tempfile.gettempdir()
>> version = 1 # specify the model version, choose #1 for now

# Let's join the temp model directory with our chosen version
  number

>> export_path = os.path.join(MODEL_DIR, str(version))
>> print('export_path = {}\n'.format(export_path))


# Save the model using simple_save

>> if os.path.isdir(export_path):
  print('\nAlready saved a model, cleaning up\n')
  !rm -r {export_path}

>> tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

8.2 ADD TENSORFLOW-MODEL-SERVER PACKAGE TO OUR LIST OF PACKAGES

```
>> !echo "deb
http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server >> tensorflow-model-server-universal"
| tee /etc/apt/sources.list.d/tensorflow-serving.list && \
curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflo
w-serving.release.pub.gpg | apt-key add -
>> !apt update
```

10.3 INSTALL TENSORFLOW MODEL SERVER:

```
>> !apt-get install tensorflow-model-server
```

10.4 RUN TENSORFLOW SERVING

```
>> os.environ["MODEL_DIR"] = MODEL_DIR
>> %%bash --bg
>> nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=fashion_model \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1

>> !tail server.log
```

10.5 START MAKING REQUESTS IN TENSORFLOW SERVING

Let's create a JSON object and make 3 inference requests

```
>> data = json.dumps({"signature_name": "serving_default",
"instances": test_images[0:10].tolist()})
>> print('Data: {} ... {}'.format(data[:50], data[len(data)-52:]))
>> !pip install -q requests
>> import requests
>> headers = {"content-type": "application/json"}
>> json_response =
requests.post('http://localhost:8501/v1/models/fashion_model:pr
edict', data=data, headers=headers)
>> predictions = json.loads(json_response.text)['predictions']
>> show(0, 'The model thought this was a {} (class {}), and it was
actually a {} (class {})'.format(
  class_names[np.argmax(predictions[0])], test_labels[0],
class_names[np.argmax(predictions[0])], test_labels[0]))
```