

## Shapley Values Tutorial

Welcome, during this ungraded lab you are going to be working with SHAP (SHapley Additive exPlanations). This procedure is derived from game theory and aims to understand (or explain) the output of any machine learning model. In particular you will:

1. Train a simple CNN on the fashion mnist dataset.
2. Compute the Shapley values for examples of each class.
3. Visualize these values and derive information from them.


To learn more about Shapley Values visit the official [SHAP repo](#).

Let's get started!

### Imports

Now import all necessary dependencies:

```
1 import shap
2 import numpy as np
3 import tensorflow as tf
4 from tensorflow import keras
5 import matplotlib.pyplot as plt
```

 /home/bikas/miniconda3/envs/xai/lib/python3.6/site-packages/tqdm/auto.py:22: TqdmWarning: IPProgress not found. Please update jupyter and notebook from .autonotebook import tqdm as notebook\_tqdm

### Train a CNN model

For this lab you will use the [fashion MNIST](#) dataset. Load it and pre-process the data before feeding it into the model:

```
1 # Download the dataset
2 (x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
3
4 # Reshape and normalize data
5 x_train = x_train.reshape(60000, 28, 28, 1).astype("float32") / 255
6 x_test = x_test.reshape(10000, 28, 28, 1).astype("float32") / 255
```

For the CNN model you will use a simple architecture composed of a single convolutional and maxpooling layers pair connected to a fully connected layer with 256 units and the output layer with 10 units since there are 10 categories.

Define the model using Keras' [Functional API](#):

```

1 # Define the model architecture using the functional API
2 inputs = keras.Input(shape=(28, 28, 1))
3 x = keras.layers.Conv2D(32, (3, 3), activation='relu')(inputs)
4 x = keras.layers.MaxPooling2D((2, 2))(x)
5 x = keras.layers.Flatten()(x)
6 x = keras.layers.Dense(256, activation='relu')(x)
7 outputs = keras.layers.Dense(10, activation='softmax')(x)
8
9 # Create the model with the corresponding inputs and outputs
10 model = keras.Model(inputs=inputs, outputs=outputs, name="CNN")
11
12 # Compile the model
13 model.compile(
14     loss=tf.keras.losses.SparseCategoricalCrossentropy(),
15     optimizer=keras.optimizers.Adam(),
16     metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
17 )
18
19 # Train it!
20 model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

```

Epoch 1/5  
1875/1875 [=====] - 11s 6ms/step - loss: 0.3721 - sparse\_categorical\_accuracy: 0.8668 - val\_loss: 0.2996 - val  
Epoch 2/5  
1875/1875 [=====] - 10s 5ms/step - loss: 0.2502 - sparse\_categorical\_accuracy: 0.9093 - val\_loss: 0.2570 - val  
Epoch 3/5  
1875/1875 [=====] - 10s 6ms/step - loss: 0.2063 - sparse\_categorical\_accuracy: 0.9241 - val\_loss: 0.2583 - val  
Epoch 4/5  
1875/1875 [=====] - 10s 5ms/step - loss: 0.1695 - sparse\_categorical\_accuracy: 0.9370 - val\_loss: 0.2557 - val  
Epoch 5/5  
1875/1875 [=====] - 10s 5ms/step - loss: 0.1399 - sparse\_categorical\_accuracy: 0.9483 - val\_loss: 0.2521 - val  
<tensorflow.python.keras.callbacks.History at 0x76d2e89cf780>

Judging the accuracy metrics looks like the model is overfitting. However, it is achieving a >90% accuracy on the test set so its performance is adequate for the purposes of this lab.

## ✓ Explaining the outputs

You know that the model is correctly classifying around 90% of the images in the test set. But how is it doing it? What pixels are being used to determine if an image belongs to a particular class?

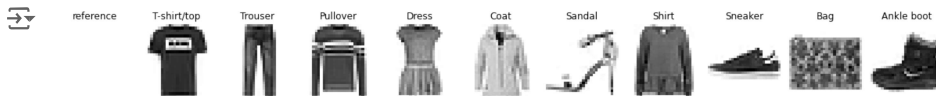
To answer these questions you can use SHAP values.

Before doing so, check how each one of the categories looks like:

```

1 # Name each one of the classes
2 class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
3               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
4
5 # Save an example for each category in a dict
6 images_dict = dict()
7 for i, l in enumerate(y_train):
8     if len(images_dict)==10:
9         break
10 if l not in images_dict.keys():
11     images_dict[l] = x_train[i].reshape((28, 28))
12
13 # Function to plot images
14 def plot_categories(images):
15     fig, axes = plt.subplots(1, 11, figsize=(16, 15))
16     axes = axes.flatten()
17
18     # Plot an empty canvas
19     ax = axes[0]
20     dummy_array = np.array([[[0, 0, 0, 0]]], dtype='uint8')
21     ax.set_title("reference")
22     ax.set_axis_off()
23     ax.imshow(dummy_array, interpolation='nearest')
24
25     # Plot an image for every category
26     for k,v in images.items():
27         ax = axes[k+1]
28         ax.imshow(v, cmap=plt.cm.binary)
29         ax.set_title(f"{class_names[k]}")
30         ax.set_axis_off()
31
32     plt.tight_layout()
33     plt.show()
34
35
36 # Use the function to plot
37 plot_categories(images_dict)

```



Now you know how the items in each one of the categories looks like.

You might wonder what the empty image at the left is for. You will see shortly why it is important.

## DeepExplainer

To compute shap values for the model you just trained you will use the `DeepExplainer` class from the `shap` library.

To instantiate this class you need to pass in a model along with training examples. Notice that not all of the training examples are passed in but only a fraction of them.

This is done because the computations done by the `DeepExplainer` object are very intensive on the RAM and you might run out of it.

```

1 # Take a random sample of 5000 training images
2 background = x_train[np.random.choice(x_train.shape[0], 5000, replace=False)]
3
4 # Use DeepExplainer to explain predictions of the model
5 e = shap.DeepExplainer(model, background)
6
7 # Compute shap values
8 # shap_values = e.shap_values(x_test[1:5])

```

 keras is no longer supported, please use tf.keras instead.  
Your TensorFlow version is newer than 2.4.0 and so graph support has been removed in eager mode. See PR #1483 for discussion.

Now you can use the `DeepExplainer` instance to compute Shap values for images on the test set.

So you can properly visualize these values for each class, create an array that contains one element of each class from the test set:

```
1 # Save an example of each class from the test set
2 x_test_dict = dict()
3 for i, l in enumerate(y_test):
4     if len(x_test_dict)==10:
5         break
6     if l not in x_test_dict.keys():
7         x_test_dict[l] = x_test[i]
8
9 # Convert to list preserving order of classes
10 x_test_each_class = [x_test_dict[i] for i in sorted(x_test_dict)]
11
12 # Convert to tensor
13 x_test_each_class = np.asarray(x_test_each_class)
14
15 # Print shape of tensor
16 print(f"x_test_each_class tensor has shape: {x_test_each_class.shape}")

x_test_each_class tensor has shape: (10, 28, 28, 1)
```

Before computing the shap values, make sure that the model is able to correctly classify each one of the examples you just picked:

```
1 # Compute predictions
2 predictions = model.predict(x_test_each_class)
3
4 # Apply argmax to get predicted class
5 np.argmax(predictions, axis=1)

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Since the test examples are ordered according to the class number and the predictions array is also ordered, the model was able to correctly classify each one of these images.

## Visualizing Shap Values

Now that you have an example of each class, compute the Shap values for each example:

```
1 # Compute shap values using DeepExplainer instance
2 shap_values = e.shap_values(x_test_each_class)

`tf.keras.backend.set_learning_phase` is deprecated and will be removed after 2020-10-11. To update it, simply pass a True/False value
```

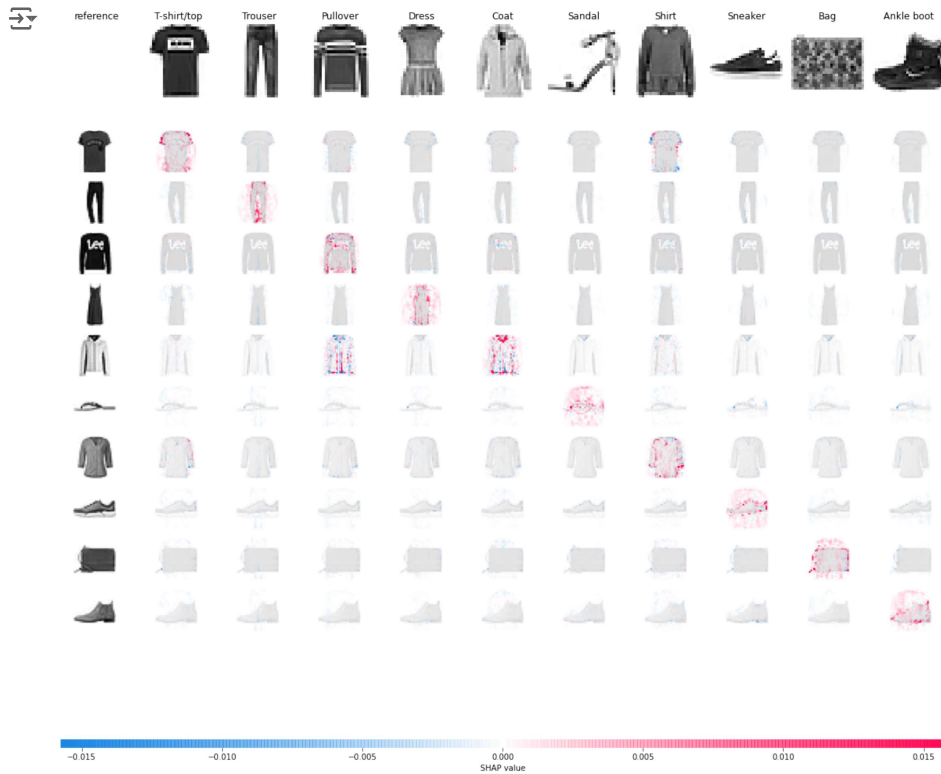
Now take a look at the computed shap values. To understand the next illustration have these points in mind:

- Positive shap values are denoted by red color and they represent the pixels that contributed to classifying that image as that particular class.
- Negative shap values are denoted by blue color and they represent the pixels that contributed to NOT classify that image as that particular class.
- Each row contains each one of the test images you computed the shap values for.
- Each column represents the ordered categories that the model could choose from. Notice that `shap.image_plot` just makes a copy of the classified image, but you can use the `plot_categories` function you created earlier to show an example of that class for reference.

```

1 # Plot reference column
2 plot_categories(images_dict)
3
4 # Print an empty line to separate the two plots
5 print()
6
7 # Plot shap values
8 shap.image_plot(shap_values, -x_test_each_class)

```



Now take some time to understand what the plot is showing you. Since the model is able to correctly classify each one of these 10 images, it makes sense that the shapley values along the diagonal are the most prevalent. Specially positive values since that is the class the model (correctly) predicted.

What else can you derive from this plot? Try focusing on one example. For instance focus on the **coat** which is the fifth class. Looks like the model also had "reasons" to classify it as **pullover** or a **shirt**. This can be concluded from the presence of positive shap values for these classes.

Let's take a look at the tensor of predictions to double check if this was the case:

```

1 # Save the probability of belonging to each class for the fifth element of the set
2 coat_probs = predictions[4]
3
4 # Order the probabilities in ascending order
5 coat_args = np.argsort(coat_probs)
6
7 # Reverse the list and get the top 3 probabilities
8 top_coat_args = coat_args[::-1][:3]
9
10 # Print (ordered) top 3 classes
11 for i in list(top_coat_args):
12     print(class_names[i])

```

```

↗ Coat
Pullover
Shirt

```

Indeed the model selected these 3 classes as the most probable ones for the **coat** image. This makes sense since these objects are similar to each other.

Now look at the **t-shirt** which is the first class. This object is very similar to the **pullover** but without the long sleeves. It is not a surprise that white pixels in the area where the long sleeves are present will yield high shap values for classifying as a **t-shirt**. In the same way, white pixels in this area will yield negative shap values for classifying as a **pullover** since the model will expect these pixels to be colored if the item was indeed a **pullover**.

You can get a lot of insight repeating this process for all the classes. What other conclusions can you arrive at?

## ✓ Multi Class Hand Written Digit Classifications

### Importing Library

```

1 import keras
2 import numpy as np
3 from keras import layers
4 from tensorflow.keras.utils import to_categorical
5 import shap
6 import tensorflow

```

### Model / data parameters

```

1 num_classes = 10
2 input_shape = (28, 28, 1)

```

### Load the data and split it between train and test sets

```

1 (x_train, y_train), (x_test, y_test) = tensorflow.keras.datasets.mnist.load_data()

1 # Scale images to the [0, 1] range
2 x_train = x_train.astype("float32") / 255
3 x_test = x_test.astype("float32") / 255
4 # Make sure images have shape (28, 28, 1)
5 x_train = np.expand_dims(x_train, -1)
6 x_test = np.expand_dims(x_test, -1)
7 print("x_train shape:", x_train.shape)
8 print(x_train.shape[0], "train samples")
9 print(x_test.shape[0], "test samples")

```

```

↗ x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples

```

convert class vectors to binary class matrices

```

1 y_train = to_categorical(y_train, num_classes)
2 y_test = to_categorical(y_test, num_classes)
3
4 batch_size = 128
5 epochs = 10


```

## Model

```

1 model = keras.Sequential(
2     [
3         layers.Input(shape=input_shape),
4         layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
5         layers.MaxPooling2D(pool_size=(2, 2)),
6         layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
7         layers.MaxPooling2D(pool_size=(2, 2)),
8         layers.Flatten(),
9         layers.Dropout(0.5),
10        layers.Dense(num_classes, activation="softmax"),
11    ]
12 )
13 model.summary()
14 model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

```

 WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential model. `keras.Input` is intended to be used with the `keras.models.Sequential` class.  
Model: "sequential\_3"


Layer (type)	Output Shape	Param #
=====		
conv2d_6 (Conv2D)	(None, 26, 26, 32)	320
=====		
max_pooling2d_6 (MaxPooling2D)	(None, 13, 13, 32)	0
=====		
conv2d_7 (Conv2D)	(None, 11, 11, 64)	18496
=====		
max_pooling2d_7 (MaxPooling2D)	(None, 5, 5, 64)	0
=====		
flatten_3 (Flatten)	(None, 1600)	0
=====		
dropout_3 (Dropout)	(None, 1600)	0
=====		
dense_3 (Dense)	(None, 10)	16010
=====		
Total params: 34,826		
Trainable params: 34,826		
Non-trainable params: 0		

## Train the Model

```

1 model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
2
3 score = model.evaluate(x_test, y_test, verbose=0)
4 print("Test loss:", score[0])
5 print("Test accuracy:", score[1])

```

 Epoch 1/10  
422/422 [=====] - 8s 18ms/step - loss: 0.7602 - accuracy: 0.7665 - val\_loss: 0.0838 - val\_accuracy: 0.9782  
Epoch 2/10  
422/422 [=====] - 8s 18ms/step - loss: 0.1193 - accuracy: 0.9640 - val\_loss: 0.0590 - val\_accuracy: 0.9840  
Epoch 3/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0883 - accuracy: 0.9737 - val\_loss: 0.0480 - val\_accuracy: 0.9883  
Epoch 4/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0728 - accuracy: 0.9778 - val\_loss: 0.0432 - val\_accuracy: 0.9877  
Epoch 5/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0632 - accuracy: 0.9811 - val\_loss: 0.0414 - val\_accuracy: 0.9882  
Epoch 6/10  
422/422 [=====] - 8s 18ms/step - loss: 0.0571 - accuracy: 0.9813 - val\_loss: 0.0363 - val\_accuracy: 0.9898  
Epoch 7/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0505 - accuracy: 0.9843 - val\_loss: 0.0353 - val\_accuracy: 0.9902  
Epoch 8/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0498 - accuracy: 0.9844 - val\_loss: 0.0326 - val\_accuracy: 0.9907  
Epoch 9/10  
422/422 [=====] - 8s 19ms/step - loss: 0.0408 - accuracy: 0.9877 - val\_loss: 0.0322 - val\_accuracy: 0.9910  
Epoch 10/10

```
422/422 [=====] - 8s 19ms/step - loss: 0.0399 - accuracy: 0.9879 - val_loss: 0.0294 - val_accuracy: 0.9915
Test loss: 0.02999413199722767
Test accuracy: 0.9900000095367432
```

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Assuming x_train and y_train are already defined
5 class_names = ['0', '1', '2', '3', '4',
6               '5', '6', '7', '8', '9']
7
8 # Save an example for each category in a dict
9 images_dict = dict()
10 for i, l in enumerate(y_train):
11     l = np.argmax(l) # Convert one-hot encoded label to scalar
12     if len(images_dict) == 10:
13         break
14     if l not in images_dict.keys():
15         images_dict[l] = x_train[i].reshape((28, 28))
16
17 # Function to plot images
18 def plot_categories(images):
19     fig, axes = plt.subplots(1, 11, figsize=(16, 15))
20     axes = axes.flatten()
21
22     # Plot an empty canvas
23     ax = axes[0]
24     dummy_array = np.array([[0, 0, 0, 0]], dtype='uint8')
25     ax.set_title("reference")
26     ax.set_axis_off()
27     ax.imshow(dummy_array, interpolation='nearest')
28
29     # Plot an image for every category
30     for k, v in images.items():
31         ax = axes[k + 1]
32         ax.imshow(v, cmap=plt.cm.binary)
33         ax.set_title(f"{class_names[k]}")
34         ax.set_axis_off()
35
36     plt.tight_layout()
37     plt.show()
38
39 # Use the function to plot
40 plot_categories(images_dict)
41
```

## Explainable AI

```
1 # select a set of background examples to take an expectation over
2 background = x_train[np.random.choice(x_train.shape[0], 100, replace=False)]
3
4 # explain predictions of the model on three images
5 e = shap.DeepExplainer(model, background)
6 # ...or pass tensors directly
7 # e = shap.DeepExplainer((model.layers[0].input, model.layers[-1].output), background)
8 shap_values = e.shap_values(x_test[0:50])
```



```

1 # Save an example of each class from the test set
2 x_test_dict = dict()
3 for i, l in enumerate(y_test):
4     l = np.argmax(l) # Convert one-hot encoded label to scalar
5     if len(x_test_dict) == 10:
6         break
7     if l not in x_test_dict.keys():
8         x_test_dict[l] = x_test[i]
9
10 # Compute predictions
11 predictions = model.predict(x_test_each_class)
12
13 # Apply argmax to get predicted class
14 np.argmax(predictions, axis=1)
15
16 # Use the function to plot
17 plot_categories(images_dict)
18
19 # plot the feature attributions
20 shap.image_plot(shap_values, -x_test[0:5])

```

