P. Mishra, *Explainable AI Recipes*
https://doi.org/10.1007/978-1-4842-9029-3_7

7. Explainability for Deep Learning Models

Pradeepta Mishra[1]
(1) Bangalore, Karnataka, India

Deep learning models are becoming the backbone of artificial intelligence implementations. At the same time, it is super important to build the explainability layers to explain the predictions and output of the deep learning model. To build trust for the deep learning model outcome, we need to explain the results or output. At a high level, a deep learning layer involves more than one hidden layer, whereas a neural network layer has three layers: the input layer, the hidden layer, and the output layer. There are different variants of neural network models such as single hidden layer neural network model, multiple hidden layer neural networks, feed-forward neural networks, and backpropagation neural networks. Depending upon the structure of the neural network model, there are three popular structures: recurrent neural networks, which are mostly used for sequential information processing, such as audio processing, text classification, etc.; deep neural networks, which are used for building extremely deep networks; and finally, convolutional neural network models, which are used for image classification.

Deep SHAP is a framework to derive the SHAP values from a deep learning model developed using TensorFlow, Keras, or PyTorch. If we compare the machine learning models with deep learning models, the deep learning models are too difficult to explain to anyone. In this chapter, we will provide recipes for explaining the components of a deep learning model.

Recipe 7-1. Explain MNIST Images Using a Gradient Explainer Based on Keras

Problem

You want to explain a Keras-based deep learning model using SHAP.

Solution

We are using a sample image dataset called MNIST. We can first train a convolutional neural network using Keras from the TensorFlow pipeline. Then we can use the gradient explainer module from the SHAP library to build the explainer object. The explainer object can be used to create SHAP values, and further, using SHAP values, we can get more visibility into image classification tasks and individual class prediction and corresponding probability values.

How It Works

Let's take a look at the following example:

```
import TensorFlow as tf
from TensorFlow.keras import Input
from TensorFlow.keras.layers import Flatten, Dense, Dropout, Conv2D
import warnings
warnings.filterwarnings("ignore")
# load the MNIST data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

There are two inputs: one for generating explanations using a feedforward neural network layer and another using the convolutional neural network layer. This is to compare the two inputs that can be explained by the SHAP library in different ways.

```
# define our model
input1 = Input(shape=(28,28,1))
input2 = Input(shape=(28,28,1))
input2c = Conv2D(32, kernel_size=(3, 3), activation='relu')(input2)
joint = tf.keras.layers.concatenate([Flatten()(input1), Flatten()(input2c)])
out = Dense(10, activation='softmax')(Dropout(0.2)(Dense(128, activation='relu')(joint)))
model = tf.keras.models.Model(inputs = [input1, input2], outputs=out)
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 28, 28, 1)]  0           []

 input_1 (InputLayer)           [(None, 28, 28, 1)]  0           []

 conv2d (Conv2D)                (None, 26, 26, 32)   320         ['input_2[0][0]']

 flatten (Flatten)              (None, 784)          0           ['input_1[0][0]']

 flatten_1 (Flatten)            (None, 21632)        0           ['conv2d[0][0]']

 concatenate (Concatenate)      (None, 22416)        0           ['flatten[0][0]',
                                                                  'flatten_1[0][0]']

 dense_1 (Dense)                (None, 128)          2869376     ['concatenate[0][0]']

 dropout (Dropout)              (None, 128)          0           ['dense_1[0][0]']

 dense (Dense)                  (None, 10)           1290        ['dropout[0][0]']

==================================================================================================
Total params: 2,870,986
Trainable params: 2,870,986
Non-trainable params: 0
__________________________________________________________________________________________________
```

Compile the model using the Adam optimizer, with sparse categorical cross entropy and accuracy. We can choose different types of optimizers to achieve the best accuracy.

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

As the next step, we can train the model. An epoch of 3 has been selected due to processing constraints, but the epoch size can be increased based on the time availability and the computational power of the machines.

```
# fit the model
model.fit([x_train, x_train], y_train, epochs=3)
```

Once the model is created, in the next step we can install the SHAP library and create a gradient explainer object either using the same training dataset or using the test dataset.

```
pip install shap
import shap
# since we have two inputs we pass a list of inputs to the explainer
explainer = shap.GradientExplainer(model, [x_train, x_train])
# we explain the model's predictions on the first three samples of the test set
shap_values = explainer.shap_values([x_test[:3], x_test[:3]])
# since the model has 10 outputs we get a list of 10 explanations (one for each output)
print(len(shap_values))
```

The two inputs were explained previously. There are two set of SHAP values, one corresponding to the feedforward layer and another relating to the convolutional neural network layer. See Figure 7-1 and Figure 7-2.

```
# since the model has 2 inputs we get a list of 2 explanations (one for each input) for each ou
print(len(shap_values[0]))
# here we plot the explanations for all classes for the first input (this is the feed forward i
shap.image_plot([shap_values[i][0] for i in range(10)], x_test[:3])
```

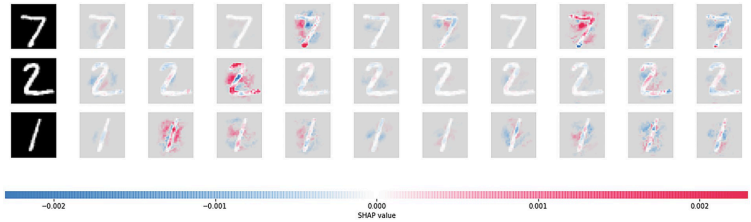

Figure 7-1 SHAP value for three samples with positive and negative weights

```
# here we plot the explanations for all classes for the second input (this is the conv-net input
shap.image_plot([shap_values[i][1] for i in range(10)], x_test[:3])
```

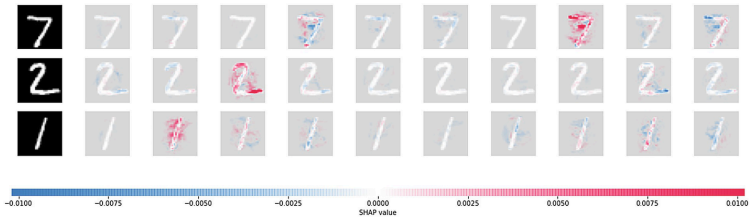

Figure 7-2 SHAP value for the second input versus all classes

```
# get the variance of our estimates
shap_values, shap_values_var = explainer.shap_values([x_test[:3], x_test[:3]], return_variances=
```

To explain the feedforward way of weight distribution and attribution of classes, we need to estimate the variances; hence, we need to get the SHAP values of variances. See Figure 7-3.

```
# here we plot the explanations for all classes for the first input (this is the feed forward i
shap.image_plot([shap_values_var[i][0] for i in range(10)], x_test[:3])
```

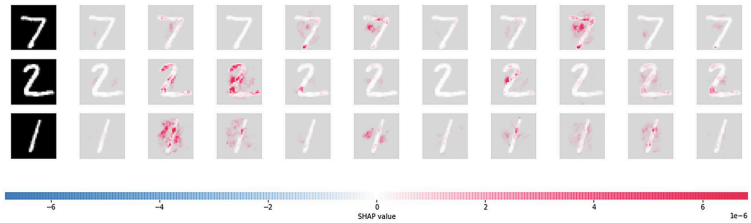

Figure 7-3 Feedforward input explanations for all classes

Recipe 7-2. Use Kernel Explainer–Based SHAP Values from a Keras Model

Problem

You want to explain the kernel-based explainer for a structured data problem for binary classification, while training with a deep learning model from Keras.

Solution

We will use the census income dataset, which is available in the SHAP library; develop a neural network model; and then use the trained model object to apply the kernel explainer. The kernel SHAP method is defined as a special weighted linear regression to compute the importance of each feature in a deep learning model.

How It Works

Let's take a look at the following example:

```
from sklearn.model_selection import train_test_split
from keras.layers import Input, Dense, Flatten, Concatenate, concatenate, Dropout, Lambda
from keras.models import Model
from keras.layers.embeddings import Embedding
from tqdm import tqdm
import shap
# print the JS visualization code to the notebook
#shap.initjs()
```

If the machine supports JS visualization, then please remove the comment and run the previous script. See Figure 7-4.

```
X,y = shap.datasets.adult()
X_display,y_display = shap.datasets.adult(display=True)
# normalize data (this is important for model convergence)
dtypes = list(zip(X.dtypes.index, map(str, X.dtypes)))
for k,dtype in dtypes:
    if dtype == "float32":
        X[k] -= X[k].mean()
        X[k] /= X[k].std()
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2, random_state=7)
# build model
input_els = []
encoded_els = []
for k,dtype in dtypes:
    input_els.append(Input(shape=(1,)))
    if dtype == "int8":
        e = Flatten()(Embedding(X_train[k].max()+1, 1)(input_els[-1]))
    else:
        e = input_els[-1]
    encoded_els.append(e)
encoded_els = concatenate(encoded_els)
layer1 = Dropout(0.5)(Dense(100, activation="relu")(encoded_els))
out = Dense(1)(layer1)
# train model
clf = Model(inputs=input_els, outputs=[out])
clf.compile(optimizer="adam", loss='binary_crossentropy')
clf.fit(
    [X_train[k].values for k,t in dtypes],
```

```
    y_train,
    epochs=5,
    batch_size=512,
    shuffle=True,
    validation_data=([X_valid[k].values for k,t in dtypes], y_valid)
)
def f(X):
    return clf.predict([X[:,i] for i in range(X.shape[1])]).flatten()
# print the JS visualization code to the notebook
shap.initjs()
explainer = shap.KernelExplainer(f, X.iloc[:50,:])
shap_values = explainer.shap_values(X.iloc[299,:], nsamples=500)
```

To generate the SHAP values, we need to use the kernel explainer function from the SHAP library.

```
shap_values50 = explainer.shap_values(X.iloc[280:285,:], nsamples=500)
shap_values
import warnings
warnings.filterwarnings("ignore")
# summarize the effects of all the features
shap_values50 = explainer.shap_values(X.iloc[280:781,:], nsamples=500)
shap.summary_plot(shap_values50)
```

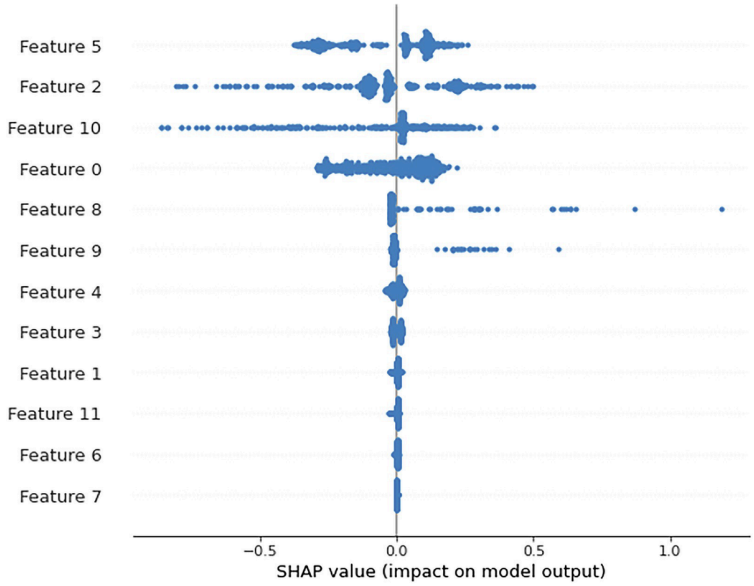

Figure 7-4 SHAP values feature importance

Recipe 7-3. Explain a PyTorch-Based Deep Learning Model

Problem

You want to explain a deep learning model developed using PyTorch.

Solution

We are using a tool called Captum, which acts as a platform. Different kinds of explainability methods are embedded into Captum that help to further elaborate on how a decision has been made. A typical neural network model interpretation can be done to understand the feature importance, dominant layer identification, and dominant neuron identification. Captum provides three attribution algorithms that help in achieving

three things: primary attribution, layer attribution, and neuron attribution.

How It Works

The following syntax explains how to install the library:

```
conda install captum -c pytorch
```

or

```
pip install captum
```

The primary attribution layer provides integrated gradients, gradient shapely additive explanations (SHAP), saliency, etc., to interpret the model in a more effective way. We can use sample data as titanic survival prediction dataset, which is a common dataset that is used for machine learning examples or tutorials every developer can quickly relate to it without much introduction.

```
# Initial imports
import numpy as np
import torch
from captum.attr import IntegratedGradients
from captum.attr import LayerConductance
from captum.attr import NeuronConductance
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
import pandas as pd
dataset_path = "https://raw.githubusercontent.com/pradmishra1/PublicDatasets/main/titanic.csv"
titanic_data = pd.read_csv(dataset_path)
del titanic_data['Unnamed: 0']
del titanic_data['PassengerId']
titanic_data = pd.concat([titanic_data,
                          pd.get_dummies(titanic_data['Sex']),
                          pd.get_dummies(titanic_data['Embarked'],prefix="embark"),
                          pd.get_dummies(titanic_data['Pclass'],prefix="class")], axis=1)
titanic_data["Age"] = titanic_data["Age"].fillna(titanic_data["Age"].mean())
titanic_data["Fare"] = titanic_data["Fare"].fillna(titanic_data["Fare"].mean())
titanic_data = titanic_data.drop(['Name','Ticket','Cabin','Sex','Embarked','Pclass'], axis=1)
# Set random seed for reproducibility.
np.random.seed(707)
# Convert features and labels to numpy arrays.
labels = titanic_data["Survived"].to_numpy()
titanic_data = titanic_data.drop(['Survived'], axis=1)
feature_names = list(titanic_data.columns)
data = titanic_data.to_numpy()
# Separate training and test sets using
train_indices = np.random.choice(len(labels), int(0.7*len(labels)), replace=False)
test_indices = list(set(range(len(labels))) - set(train_indices))
train_features = data[train_indices]
train_labels = labels[train_indices]
test_features = data[test_indices]
test_labels = labels[test_indices]
train_features.shape
(623, 12)
```

Now that the train and test datasets are ready, we can start writing the code for the model development using PyTorch.

```
Import torch
import torch.nn as nn
torch.manual_seed(1)  # Set seed for reproducibility.
Class TitanicSimpleNNModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(12, 12)
        self.sigmoid1 = nn.Sigmoid()
        self.linear2 = nn.Linear(12, 8)
        self.sigmoid2 = nn.Sigmoid()
        self.linear3 = nn.Linear(8, 2)
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        lin1_out = self.linear1(x)
        sigmoid_out1 = self.sigmoid1(lin1_out)
        sigmoid_out2 = self.sigmoid2(self.linear2(sigmoid_out1))
        return self.softmax(self.linear3(sigmoid_out2))
net = TitanicSimpleNNModel()
criterion = nn.CrossEntropyLoss()
num_epochs = 200
optimizer = torch.optim.Adam(net.parameters(), lr=0.1)
input_tensor = torch.from_numpy(train_features).type(torch.FloatTensor)
label_tensor = torch.from_numpy(train_labels)
```

The deep learning model configuration is done, so we can proceed with running epochs or iterations to reduce the errors.

```
For epoch in range(num_epochs):
  output = net(input_tensor)
  loss = criterion(output, label_tensor)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
  if epoch % 20 == 0:
        print ('Epoch {}/{} => Loss: {:.2f}'.format(epoch+1, num_epochs, loss.item()))
torch.save(net.state_dict(), '/model.pt')
Epoch 1/200 => Loss: 0.70
Epoch 21/200 => Loss: 0.55
Epoch 41/200 => Loss: 0.50
Epoch 61/200 => Loss: 0.49
Epoch 81/200 => Loss: 0.48
Epoch 101/200 => Loss: 0.49
Epoch 121/200 => Loss: 0.47
Epoch 141/200 => Loss: 0.47
Epoch 161/200 => Loss: 0.47
Epoch 181/200 => Loss: 0.47
out_probs = net(input_tensor).detach().numpy()
out_classes = np.argmax(out_probs, axis=1)
print("Train Accuracy:", sum(out_classes == train_labels) / len(train_labels))
Train Accuracy: 0.8523274478330658
test_input_tensor = torch.from_numpy(test_features).type(torch.FloatTensor)
out_probs = net(test_input_tensor).detach().numpy()
out_classes = np.argmax(out_probs, axis=1)
print("Test Accuracy:", sum(out_classes == test_labels) / len(test_labels))
Test Accuracy: 0.832089552238806
```

The integrated gradient is extracted from the neural network model; this can be done using the `attribute` function.

```
ig = IntegratedGradients(net)
test_input_tensor.requires_grad_()
attr, delta = ig.attribute(test_input_tensor,target=1, return_convergence_delta=True)
attr = attr.detach().numpy()
```

```
np.round(attr,2)
array([[-0.7 , 0.09, -0. , ..., 0. , 0. , -0.33], [-2.78, -0. , -0. , ..., 0. , 0. , -1.82], [-0
```

The `attr` object contains the feature importance of the input features from the model.

```
importances = np.mean(attr, axis=0)
for i in range(len(feature_names)):
        print(feature_names[i], ": ", '%.3f'%(importances[i]))
Age :  -0.574
SibSp :  -0.010
Parch :  -0.026
Fare :  0.278
female :  0.101
male :  -0.460
embark_C :  0.042
embark_Q :  0.005
embark_S :  -0.021
class_1 :  0.067
class_2 :  0.090
class_3 :  -0.144
```

The `LayerConductance` helps us compute the neuron importance and combines the neuron activation by taking the partial derivative of the neuron with respect to the input and output. The conductance layer builds on the integrated gradients by looking at the flow of IG attribution.

```
cond = LayerConductance(net, net.sigmoid1)
cond_vals = cond.attribute(test_input_tensor,target=1)
cond_vals = cond_vals.detach().numpy()
Average_Neuron_Importances = np.mean(cond_vals, axis=0)
Average_Neuron_Importances
array([ 0.03051018, -0.23244175, 0.04743345, 0.02102091, -0.08071412, -0.09040915, -0.13398956,
neuron_cond = NeuronConductance(net, net.sigmoid1)
neuron_cond_vals_10 = neuron_cond.attribute(test_input_tensor, neuron_selector=10, target=1)
neuron_cond_vals_0 = neuron_cond.attribute(test_input_tensor, neuron_selector=0, target=1)
# Average Feature Importances for Neuron 0
nn0 = neuron_cond_vals_0.mean(dim=0).detach().numpy()
np.round(nn0,3)
array([ 0.008, 0. , 0. , 0.028, 0. , -0.004, -0. , 0. , -0.001, -0. , 0. , -0. ], dtype=float32)
```

The average feature importance for neuron 0 can be replicated to any number of neurons by using a threshold. If the weight threshold exceeds a certain level, then the neuron attribution and average feature importance for that neuron can be derived.

Conclusion

In this chapter, we looked two frameworks, SHAP and Captum, to explain a deep learning model developed either using Keras or using PyTorch. The more we parse the information using these libraries and take a smaller chunk of data, the more visibility we will get into how the model works, how the model makes prediction, and how the model makes an attribution to a local instance.

To review, this book started with explaining linear supervised models for both regression and classification tasks, then explained nonlinear decision tree–based models, and then covered the ensemble models such as bagging, boosting, and stacking. Finally, we ended the book with explain-

ing the times-series model, natural language processing–based text classification, and deep neural network–based models.