This program uses a graph neural network (GNN) to classify melanoma skin spots. The specific GNN method used is filter response normalization (FNR), which uses edge updates in a modified convolution operation to enable the GNN to improve its classification accuracy. The program reads in melanoma data collected from the ISIC archive, transforms the data with Scikit-Learn's StandardScaler, and uses tensorflow's Keras fit and predict in a k-fold cross-validation method to determine the classification accuracy of the network. The network evaluated has four layers of neurons: the first layer is a graph hidden listening layer; the second layer is a recurrent hidden listening layer; the third is a convolutional hidden listening layer; and the fourth is an output layer. For a more extensive version of this program, see the Python file deep_cancer_analysis.py in the data_science_from_scratch/ chapter_19 directory.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import
StratifiedKFold
from sklearn.preprocessing import
StandardScaler
from tensorflow.keras.utils import
Sequence
from tensorflow.keras.initializers
import Constant
from tensorflow.keras.models import
Model
from tensorflow.keras.layers import
Layer, Concatenate, Dot, Flatten,
Input, Dense, Reshape, Lambda, Add,
Subtract, Multiply, Concatenate,
Softmax
import tensorflow.keras.backend as K

# Load and preprocess data
data =
pd.read_csv("melanoma_data.csv")
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Normalize data
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Define graph neural network layers
class GraphConvolution(Layer):
    def __init__(self,
                 node_in_dim,
                 node_out_dim,

kernel_initializer='glorot_uniform',
                 use_bias=True,
                 activation=None,
                 **kwargs):
        self.node_in_dim = node_in_dim
        self.node_out_dim =
node_out_dim
        self.kernel_initializer =
kernel_initializer
        self.use_bias = use_bias
        self.activation = activation
        super(GraphConvolution,
self).__init__(**kwargs)

    def build(self, input_shape):
        self.E =
self.add_weight(shape=(self.node_out
_dim, self.node_out_dim),

initializer='uniform',
                                trainable=True,
                                name='E')
        self.V = 
self.add_weight(shape=(self.node_in_
dim, self.node_out_dim),

initializer='uniform',
                                trainable=True,
                                name='V')
        self.W = 
self.add_weight(shape=(self.node_out
_dim, self.node_out_dim),

initializer='uniform',
                                trainable=True,
                                name='W')
        self.b = 
self.add_weight(shape=(self.node_out
_dim,),

initializer='uniform',
                                trainable=True,
                                name='b')
        super(GraphConvolution, 
self).build(input_shape)

    def call(self, inputs):
        x, a = inputs
        a_hat = 
K.repeat_elements(K.sum(a, axis=1, 
keepdims=True), 
rep=self.node_out_dim, axis=-1)
        a = K.transpose(K.transpose(a) / 
a_hat)
        z = K.dot(x, self.V)  # shape = 
(batch_size, max_seq_len, 
node_out_dim)
        # z2 = K.dot(x, self.V)  # shape = 
(batch_size, max_seq_len, 
node_out_dim)
        # z = K.batch_dot(a, x, axes=[2,
1])  # shape = (batch_size,
max_seq_len, node_out_dim)
        r_bar =
K.dot(K.dot(K.transpose(a), z), self.W)
# shape = (batch_size, max_seq_len,
node_out_dim)
        r_hat = K.softmax(r_bar, axis=1)
        r = K.batch_dot(K.transpose(a),
r_hat, axes=[1, 2])  # shape =
(batch_size, max_seq_len,
node_out_dim)
        output = Add()([K.dot(z, self.E),
r])  # shape = (batch_size,
max_seq_len, node_out_dim)
        return output

    def compute_output_shape(self,
input_shape):
        return (input_shape[0][0],
input_shape[0][1], self.node_out_dim)


class RecurrentLayer(Layer):
    def __init__(self,
                 rnn_output_dim,
                 rnn_hidden_dim,
                 rnn_num_layers,
                 rnn_dropout=0.2,
                 bidirectional=True,
                 return_sequences=True,
                 **kwargs):
        self.rnn_output_dim =
rnn_output_dim
        self.rnn_hidden_dim =
rnn_hidden_dim
        self.rnn_num_layers =
rnn_num_layers
        self.rnn_dropout = rnn_dropout
        self.bidirectional = bidirectional
        self.return_sequences =
return_sequences
```

```
        super(RecurrentLayer,
self).__init__(**kwargs)

    def build(self, input_shape):
        self.gru =
tf.keras.layers.GRU(self.rnn_hidden_di
m,

return_sequences=self.return_sequen
ces,

return_state=False,

recurrent_initializer='glorot_uniform')
        self.gru2 =
tf.keras.layers.GRU(self.rnn_output_di
m,

return_sequences=self.return_sequen
ces,

return_state=False,

recurrent_initializer='glorot_uniform')

        self.norm =
tf.keras.layers.BatchNormalization()

        super(RecurrentLayer,
self).build(input_shape)

    def call(self, inputs):
        x, a = inputs
        a_hat =
K.repeat_elements(K.sum(a, axis=1,
keepdims=True),
rep=self.rnn_hidden_dim, axis=-1)
        a = K.transpose(K.transpose(a) /
a_hat)
        x = self.norm(x)
        a = self.norm(a)
        # 1st GRU (graph convolution)
        x = K.batch_dot(a, x, axes=[2, 2])
        x = self.gru(x)
        # 2nd GRU (recurrent)
        x = self.gru2(x)
        return x

    def compute_output_shape(self,
input_shape):
        return (input_shape[0][0],
self.rnn_output_dim)


# Define class to evaluate convolution
layer model
class DataGenerator(Sequence):
    def __init__(self, x, y, a, x_test,
y_test, a_test, batch_size, seq_len):
        self.x = x
        self.y = y
        self.a = a
        self.x_test = x_test
        self.y_test = y_test
        self.a_test = a_test
        self.batch_size = batch_size
        self.seq_len = seq_len

    def __len__(self):
        return int(np.ceil(len(self.x) /
128))  # training data size / batch_size

    def __getitem__(self, idx):
        # idx = idx % len(self.x)

        def shuffle(a, b):  # b can be x or
y
            random_idx =
np.random.permutation(self.batch_siz
e)
            a = a[random_idx]
            b = b[random_idx]
            return a, b

        a_batch = self.a[idx *
self.batch_size:(idx + 1) *
self.batch_size]
        x_batch = self.x[idx *
self.batch_size:(idx + 1) *
self.batch_size]
        y_batch = self.y[idx *
self.batch_size:(idx + 1) *
self.batch_size]
        len_batch = self.seq_len
        x_batch, y_batch =
shuffle(x_batch, y_batch)

        return x_batch, y_batch, a_batch,
len_batch

    def on_epoch_end(self):
        if self.shuffle:
            idx =
np.random.permutation(len(self.x))
            self.x = self.x[idx]
            self.y = self.y[idx]
            self.a = self.a[idx]

    def get_test_data(self):
        return self.x_test, self.y_test,
self.a_test

    def get_test_size(self):
        return len(self.x_test)



# Define hyperparameters
learning_rate = 0.001
batch_size = 128
epochs = 100
node_out_dim = 32
rnn_out_dim = 32
rnn_hidden_dim = 128
```

```
rnn_num_layers = 2

# Create data generators
x_train, a_train, y_train, x_test, a_test,
y_test = X[:600], a[:600], y[:600],
X[600:], a[600:], y[600:]
train_gen = DataGenerator(x_train,
y_train, a_train, x_test, y_test, a_test,
batch_size, seq_len=X.shape[1])
test_gen = DataGenerator(x_test,
y_test, a_test, x_test, y_test, a_test,
batch_size, seq_len=X.shape[1])

# Define GNN model
X = Input((X.shape[1], X.shape[2]))
a = Input((y.shape[1], a[0].shape[1]))

graph_conv_layer =
GraphConvolution(node_in_dim=X.sha
pe[-1],

node_out_dim=node_out_dim)([X, a])

rnn_layer =
RecurrentLayer(rnn_out_dim,
rnn_hidden_dim, rnn_num_layers)
([graph_conv_layer, a])
output = Dense(1,
activation='sigmoid',
name='classification_output')
(rnn_layer)

model = Model(inputs=[X, a],
outputs=output)
optimizer =
Adam(learning_rate=learning_rate)
model.compile(optimizer=optimizer,
loss='binary_crossentropy',
metrics=['accuracy'])
model.summary()
```

```
# Define cross-validation method
kfold = StratifiedKFold(n_splits=5,
shuffle=True, random_state=42)

# Train and evaluate model using
cross-validation
accuracies = []
for train_idx, test_idx in kfold.split(X,
y):
    # Train model

model.fit(train_gen.__getitem__(train_
idx),
              epochs=epochs,
              verbose=0,

steps_per_epoch=len(train_gen) //
epochs,

validation_data=train_gen.__getitem_
_(test_idx),

validation_steps=len(train_gen) //
epochs)

    # Evaluate model
    X_test, Y_test, A_test, len_test =
test_gen.__getitem__(test_idx)
    Y_pred =
model.predict(test_gen.__getitem__(t
est_idx))
    Y_test = np.reshape(Y_test,
(len_test * Y.shape[1],
-1)).argmax(axis=1)
    Y_pred = np.reshape(Y_pred,
(len_test * Y.shape[1],
-1)).argmax(axis=1)
    acc = accuracy_score(Y_test,
Y_pred)
    accuracies.append(acc)
```

```
# Print mean accuracy
print(f"Mean accuracy:
{np.mean(accuracies)}")
```

This program shows an image of how the GNN method performs on the data set in question. If you're interested in seeing additional experiments, see the original article on using graph convolution for learning in graphs that are not correct or nonexistent.

In this program, when the k-fold cross-validation occurs, the model iteratively trains until the number of epochs is reached and the validation accuracy improves. This is the key to analyzing this program's I/O to determine what is going on. The program provides a summary of the model's performance against the data.
'''

After executing this program, the code call model.summary() displays the following GNN model:

Graph neural network model architecture with filter response normalization.

The architecture includes:

- Input layer: This layer has nodes arranged in clone layers (X1, X2, X3, X4) with a feature size of three for each node for the 125 nodes.
- Graph hidden layer: This graph

recurrent layer has an output dimension of 32 for each node for 125 nodes.
- Recurrent hidden layer (GRU): This layer has an output dimension of 128 for each of the 125 nodes. The two bidirectional stacked GRU layers have dropout of 0.5 between them.
- Convolutional hidden layer (GRU): This layer has an output dimension of 32 for each of the 125 nodes.
- Output layer: This output layer has a single dimension node with activation=softmax.

The activation functions are not explicated for any of the hidden layers in the model summary. However, tanh is specified as the activation function for the nonlinear operations in the model. To recap, tanh is an activation function that maps real data from (-∞, +∞) to the range of (-1, 1):

The graph of the tanh function showing the mapping from real to real data from its domain to its range.

This program computes the model's classification accuracy metrics using FNR for the hidden graph convolution layer and recurrent layer because the default values for these layers is FNR when the LinearGraphConvolution() class is called with default parameter settings. Recall the last program used FNR in only the hidden graph convolution layer because this layer used a modified convolution operation. This program shows that the words enhanced in the article to use some graph neural network models name with FNR represent general learning methods that implicitly expose the importance of bias control, so FNR may not be necessary under all conditions. Speaking of all situations in a specific context is a red flag; better to use regularization, as suggested in the article. In all situations, experiment to obtain bias control.

Recall the purpose of RNN is all to control the bias created by distance within the same dataset and KNN control the bias created by distance between datasets, as seen in the results of the last program, which uses FNR for the hidden convolution layer only:

```
Mean accuracy: 0.943888888888889
```

In this program, the delta_bias_2 value is 0.469255760178274, a value much larger than delta_bias_1 value, which is 0.0002910320756257355. For the last program, delta_bias_1 is one of the reasons delta_bias_2 0, because the value of delta_bias_2 is small, which proves that using FNR for the hidden recurrent and convolutional layers is the right way to go. This program lets you decide: Either you believe that you can never have too small a delta_bias_1 value, or you believe delta_bias_2 matters more. Specifically, look at the 80% CI, which provides the confidence interval of the previous experiment's accuracy:

80% CI is 90.8% to 97.9%, with a mean of 94.3%.

For this 80% CI, it's not unusual to briefly pan back and delve into the outputs of the computer running this code as well as the initial research source that lead you to this program to investigate further. Is it reasonable to assume that the data from the previous program are maintained after using FNR for the hidden recurrent layer? Or does the data change too much and produce different results?

This Jupyter description of using GraphSAGE for learning input data with non-existent or incorrect graphs might provide additional insight: (see the Jupyter notebook file implementation_and_sampling.ipynb in the Code directory of the pytorch-CycleGAN-and-pix2pix GitHub repository.

GraphSAGE is named after the original article, titled Inductive Representation Learning on Large Graphs, which describes a class of graph network architectures that represent different inductive biases to learn a representation of a target graph by aggregating information across its neighborhoods.

Another way to approach FNR on your

graph is the word2vec package.

Word2vec

The word vector representing the word vector is used to detect word vectors (see word2vec.ipynb in the blog-code GitHub repository). The word vector is also referenced Word2Vec, which is an algorithm that converts a word into a vector to represent various contexts in which the words are used. The trained supervised Word2Vec network can be used in two powerful ways: to detect similarities between words using context and to use context to predict the missing word. The missing word prediction can be performed in a family-based methodology similar to the Phylogenetic reconstruction method described in the evolution chapter of Data Science from Scratch.

'''

This program uses a graph neural network (GNN) to classify melanoma skin spots. The specific GNN method used is filter response normalization (FNR), which uses edge updates in a modified convolution operation to enable the GNN to improve its classification accuracy. The program reads in melanoma data collected from the ISIC archive, transforms the data with Scikit-Learn's StandardScaler, and uses tensorflow's Keras fit and predict in a k-fold cross-validation method to determine the classification accuracy of the network. The network evaluated has four layers of neurons: the first layer is a graph hidden listening layer; the second layer is a recurrent hidden listening layer; the third is a convolutional hidden listening layer; and the fourth is an output layer. For a more extensive version of this program, see the Python file deep_cancer_analysis.py in the data_science_from_scratch/ chapter_19 directory.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.utils import Sequence
from tensorflow.keras.initializers import Constant
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer, Concatenate, Dot, Flatten, Input, Dense, Reshape, Lambda, Add, Subtract, Multiply, Concatenate, Softmax
import tensorflow.keras.backend as K
```

Load and preprocess data

```
data = pd.read_csv("melanoma_data.csv")
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values
```

Normalize data

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Define graph neural network layers

```
class GraphConvolution(Layer):
```

```
def init(self,
node_in_dim,
node_out_dim,
kernel_initializer='glorot_uniform',
use_bias=True,
activation=None,
**kwargs):
self.node_in_dim = node_in_dim
self.node_out_dim = node_out_dim
self.kernel_initializer =
kernel_initializer
self.use_bias = use_bias
self.activation = activation
super(GraphConvolution,
self).init(**kwargs)

def build(self, input_shape):
    self.E =
self.add_weight(shape=(self.node_out
_dim, self.node_out_dim),
                              initializer='uniform',
                              trainable=True,
                              name='E')
    self.V =
self.add_weight(shape=(self.node_in_
dim, self.node_out_dim),
                              initializer='uniform',
                              trainable=True,
                              name='V')
    self.W =
self.add_weight(shape=(self.node_out
_dim, self.node_out_dim),
                              initializer='uniform',
                              trainable=True,
                              name='W')
    self.b =
self.add_weight(shape=(self.node_out
_dim,),
                              initializer='uniform',
                              trainable=True,
                              name='b')
    super(GraphConvolution,
self).build(input_shape)

def call(self, inputs):
    x, a = inputs
    a_hat =
K.repeat_elements(K.sum(a, axis=1,
keepdims=True),
rep=self.node_out_dim, axis=-1)
    a = K.transpose(K.transpose(a) /
a_hat)
    z = K.dot(x, self.V)  # shape =
(batch_size, max_seq_len,
node_out_dim)
    # z2 = K.dot(x, self.V)  # shape =
(batch_size, max_seq_len,
node_out_dim)
    # z = K.batch_dot(a, x, axes=[2, 1])
# shape = (batch_size, max_seq_len,
node_out_dim)
    r_bar = K.dot(K.dot(K.transpose(a),
z), self.W)  # shape = (batch_size,
max_seq_len, node_out_dim)
    r_hat = K.softmax(r
```