Open in Colab

T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 5 Material

- Part 5.1: Part 5.1: Introduction to Regularization: Ridge and Lasso [Video] [Notebook]
- Part 5.2: Using K-Fold Cross Validation with Keras [Video] [Notebook]
- **Part 5.3: Using L1 and L2 Regularization with Keras to Decrease Overfitting** [Video] [Notebook]
- Part 5.4: Drop Out for Keras to Decrease Overfitting [Video] [Notebook]
- Part 5.5: Benchmarking Keras Deep Learning Regularization Techniques [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

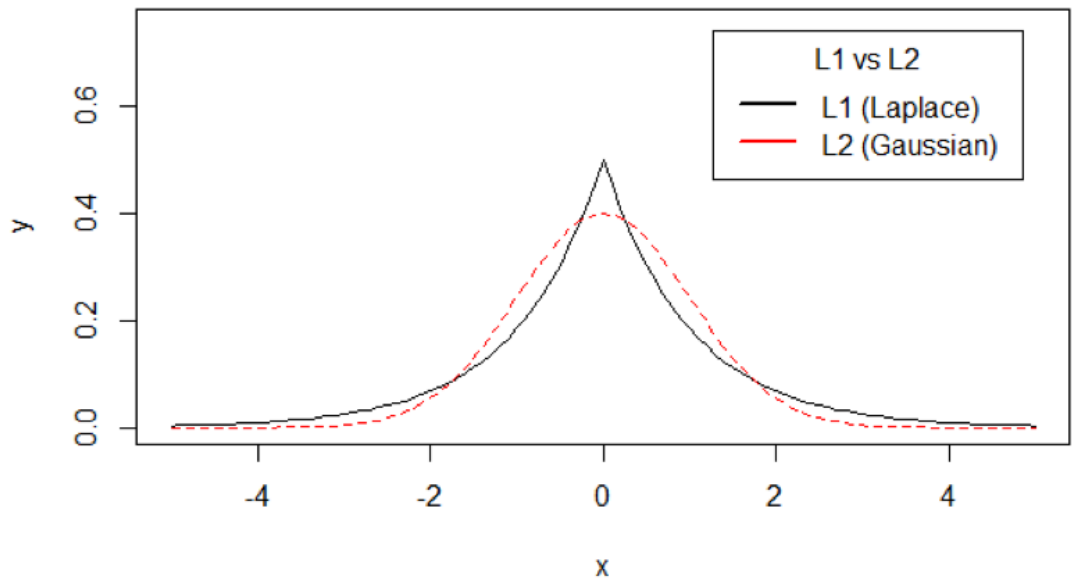
Part 5.3: L1 and L2 Regularization to Decrease Overfitting

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting [Cite:ng2004feature]. These algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases, the regularization algorithm is attached to the training algorithm by adding an objective.

These algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. You can add this penalty calculation to the calculated gradients for gradient-descent-based algorithms, such as backpropagation. The penalty is negatively combined with the objective score for objective-function-based training, such as simulated annealing.

Both L1 and L2 work differently in that they penalize the size of the weight. L2 will force the weights into a pattern similar to a Gaussian distribution; the L1 will force the weights into a pattern similar to a Laplace distribution, as demonstrated in Figure 5.L1L2.

Figure 5.L1L2: L1 vs L2



As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values. Keras allows l1/l2 to be directly added to your network.

In [2]:
```
import pandas as pd
from scipy.stats import zscore
```

```
# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We now create a Keras network with L1 regression.

In [3]:
```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers

# Cross-validate
kf = KFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
```

```
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    #kernel_regularizer=regularizers.l2(0.01),

    model = Sequential()
    # Hidden 1
    model.add(Dense(50, input_dim=x.shape[1],
            activation='relu',
             activity_regularizer=regularizers.l1(1e-4)))
    # Hidden 2
    model.add(Dense(25, activation='relu',
                    activity_regularizer=regularizers.l1(1e-4)))
     # Output
    model.add(Dense(y.shape[1],activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              verbose=0,epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    # raw probabilities to chosen class (highest probability)
    pred = np.argmax(pred,axis=1)
    oos_pred.append(pred)

    # Measure this fold's accuracy
    y_compare = np.argmax(y_test,axis=1) # For accuracy calculation
    score = metrics.accuracy_score(y_compare, pred)
    print(f"Fold score (accuracy): {score}")


# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
#oosDF.to_csv(filename_write,index=False)
```

```
Fold #1
Fold score (accuracy): 0.64
Fold #2
Fold score (accuracy): 0.6775
Fold #3
Fold score (accuracy): 0.6825
Fold #4
Fold score (accuracy): 0.6675
Fold #5
Fold score (accuracy): 0.645
Final score (accuracy): 0.6625
```

In [ ]: