Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- **Part 4.1: Encoding a Feature Vector for Keras Deep Learning** [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [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 4.1: Encoding a Feature Vector for Keras Deep Learning

Neural networks can accept many types of data. We will begin with tabular data, where there are well-defined rows and columns. This data is what you would typically see in Microsoft Excel. Neural networks require numeric input. This numeric form is called a feature vector. Each input neurons receive one feature (or column) from this vector. Each row of training data typically becomes one vector. This section will see how to encode the following tabular data into a feature vector. You can see an example of tabular data below.

In [2]:
```
import pandas as pd

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

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

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 5)

display(df)
```

	id	job	area	income	...	pop_dense	retail_dense	crime	product
0	1	vv	c	50876.0	...	0.885827	0.492126	0.071100	b
1	2	kd	c	60369.0	...	0.874016	0.342520	0.400809	c
...	...	...	...	...	...	...	...	...	...
1998	1999	qp	c	67949.0	...	0.909449	0.598425	0.117803	c
1999	2000	pe	c	61467.0	...	0.925197	0.539370	0.451973	c

2000 rows × 14 columns

You can make the following observations from the above data:

- The target column is the column that you seek to predict. There are several candidates here. However, we will initially use the column "product". This field specifies what product someone bought.
- There is an ID column. You should exclude his column because it contains no information useful for prediction.
- Many of these fields are numeric and might not require further processing.
- The income column does have some missing values.
- There are categorical values: job, area, and product.

To begin with, we will convert the job code into dummy variables.

```
In [3]: pd.set_option('display.max_columns', 7)
        pd.set_option('display.max_rows', 5)

        dummies = pd.get_dummies(df['job'],prefix="job")
        print(dummies.shape)

        pd.set_option('display.max_columns', 9)
        pd.set_option('display.max_rows', 10)

        display(dummies)
```

```
(2000, 33)
```

	job_11	job_al	job_am	job_ax	...	job_rn	job_sa	job_vv	job_zz
0	0	0	0	0	...	0	0	1	0
1	0	0	0	0	...	0	0	0	0
2	0	0	0	0	...	0	0	0	0
3	1	0	0	0	...	0	0	0	0
4	0	0	0	0	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	0	0	0	0	...	0	0	1	0
1996	0	0	0	0	...	0	0	0	0
1997	0	0	0	0	...	0	0	0	0
1998	0	0	0	0	...	0	0	0	0
1999	0	0	0	0	...	0	0	0	0

2000 rows × 33 columns

Because there are 33 different job codes, there are 33 dummy variables. We also specified a prefix because the job codes (such as "ax") are not that meaningful by themselves. Something such as "job_ax" also tells us the origin of this field.

Next, we must merge these dummies back into the main data frame. We also drop the original "job" field, as the dummies now represent it.

```
In [4]: pd.set_option('display.max_columns', 7)
        pd.set_option('display.max_rows', 5)

        df = pd.concat([df,dummies],axis=1)
        df.drop('job', axis=1, inplace=True)

        pd.set_option('display.max_columns', 9)
        pd.set_option('display.max_rows', 10)
```

```
display(df)
```

	id	area	income	aspect	...	job_rn	job_sa	job_vv	job_zz
0	1	c	50876.0	13.100000	...	0	0	1	0
1	2	c	60369.0	18.625000	...	0	0	0	0
2	3	c	55126.0	34.766667	...	0	0	0	0
3	4	c	51690.0	15.808333	...	0	0	0	0
4	5	d	28347.0	40.941667	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	1996	c	51017.0	38.233333	...	0	0	1	0
1996	1997	d	26576.0	33.358333	...	0	0	0	0
1997	1998	d	28595.0	39.425000	...	0	0	0	0
1998	1999	c	67949.0	5.733333	...	0	0	0	0
1999	2000	c	61467.0	16.891667	...	0	0	0	0

2000 rows × 46 columns

We also introduce dummy variables for the area column.

```
In [5]: pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

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

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
display(df)
```

	id	income	aspect	subscriptions	...	area_a	area_b	area_c	area_
0	1	50876.0	13.100000	1	...	0	0	1	
1	2	60369.0	18.625000	2	...	0	0	1	
2	3	55126.0	34.766667	1	...	0	0	1	
3	4	51690.0	15.808333	1	...	0	0	1	
4	5	28347.0	40.941667	3	...	0	0	0	
...	...	...	...	...	...	...	...	...	
1995	1996	51017.0	38.233333	1	...	0	0	1	
1996	1997	26576.0	33.358333	2	...	0	0	0	
1997	1998	28595.0	39.425000	3	...	0	0	0	
1998	1999	67949.0	5.733333	0	...	0	0	1	
1999	2000	61467.0	16.891667	0	...	0	0	1	

2000 rows × 49 columns

The last remaining transformation is to fill in missing income values.

In [6]:
```
med = df['income'].median()
df['income'] = df['income'].fillna(med)
```

There are more advanced ways of filling in missing values, but they require more analysis. The idea would be to see if another field might hint at what the income was. For example, it might be beneficial to calculate a median income for each area or job category. This technique is something to keep in mind for the class Kaggle competition.

At this point, the Pandas dataframe is ready to be converted to Numpy for neural network training. We need to know a list of the columns that will make up *x* (the predictors or inputs) and *y* (the target).

The complete list of columns is:

In [7]:
```
print(list(df.columns))
```

```
['id', 'income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'di
st_unhealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'product', 'job_
11', 'job_al', 'job_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de',
'job_dz', 'job_e2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_k
l', 'job_kp', 'job_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'j
ob_pe', 'job_po', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_s
a', 'job_vv', 'job_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

This data includes both the target and predictors. We need a list with the target removed. We also remove **id** because it is not useful for prediction.

In [8]:
```
x_columns = df.columns.drop('product').drop('id')
print(list(x_columns))
```

```
['income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'dist_unh
ealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'job_11', 'job_al', 'j
ob_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de', 'job_dz', 'job_e
2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_kl', 'job_kp', 'j
ob_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'job_pe', 'job_p
o', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_sa', 'job_vv', 'j
ob_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

Generate X and Y for a Classification Neural Network

We can now generate *x* and *y*. Note that this is how we generate y for a classification problem. Regression would not use dummies and would encode the numeric value of the target.

In [9]:
```
# 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 can display the *x* and *y* matrices.

In [10]:
```
print(x)
print(y)
```

```
[[5.08760000e+04 1.31000000e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [6.03690000e+04 1.86250000e+01 2.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [5.51260000e+04 3.47666667e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 ...
 [2.85950000e+04 3.94250000e+01 3.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 1.00000000e+00]
 [6.79490000e+04 5.73333333e+00 0.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [6.14670000e+04 1.68916667e+01 0.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]]
[[0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 1 0]
 [0 0 1 ... 0 0 0]
 [0 0 1 ... 0 0 0]]
```

The x and y values are now ready for a neural network. Make sure that you construct the neural network for a classification problem. Specifically,

- Classification neural networks have an output neuron count equal to the number of classes.
- Classification neural networks should use **categorical_crossentropy** and a **softmax** activation function on the output layer.

Generate X and Y for a Regression Neural Network

The program generates the *x* values the say way for a regression neural network. However, *y* does not use dummies. Make sure to replace **income** with your actual target.

In [11]:
```
y = df['income'].values
```

Module 4 Assignment

You can find the first assignment here: assignment 4

In [ ]:
```
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- **Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC** [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [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: using Google CoLab
```

Part 4.2: Multiclass Classification with ROC and AUC

The output of modern neural networks can be of many different forms. However, classically, neural network output has typically been one of the following:

- **Binary Classification** - Classification between two possibilities (positive and negative). Common in medical testing, does the person has the disease (positive) or not (negative).
- **Classification** - Classification between more than 2. The iris dataset (3-way classification).
- **Regression** - Numeric prediction. How many MPG does a car get? (covered in next video)

We will look at some visualizations for all three in this section.

It is important to evaluate the false positives and negatives in the results produced by a neural network. We will now look at assessing error for both classification and regression neural networks.

Binary Classification and ROC Charts

Binary classification occurs when a neural network must choose between two options: true/false, yes/no, correct/incorrect, or buy/sell. To see how to use binary classification, we will consider a classification system for a credit card company. This system will either "issue a credit card" or "decline a credit card." This classification system must decide how to respond to a new potential customer.

When you have only two classes that you can consider, the objective function's score is the number of false-positive predictions versus the number of false negatives. False negatives and false positives are both types of errors, and it is essential to understand the difference. For the previous example, issuing a credit card would be positive. A false positive occurs when a model decides to issue a credit card to someone who will not make payments as agreed. A false negative happens when a model denies a credit card to someone who would have made payments as agreed.

Because only two options exist, we can choose the mistake that is the more serious type of error, a false positive or a false negative. For most banks issuing credit cards, a false positive is worse than a false negative. Declining a potentially good credit card holder is better than accepting a credit card holder who would cause the bank to undertake expensive collection activities.

Consider the following program that uses the wcbreast_wdbc dataset to classify if a breast tumor is cancerous (malignant) or not (benign).

```
In [2]: import pandas as pd

        df = pd.read_csv(
            "https://data.heatonresearch.com/data/t81-558/wcbreast_wdbc.csv",
            na_values=['NA','?'])

        pd.set_option('display.max_columns', 5)
        pd.set_option('display.max_rows', 5)

        display(df)
```

	id	diagnosis	...	worst_symmetry	worst_fractal_dimension
0	842302	M	...	0.4601	0.11890
1	842517	M	...	0.2750	0.08902
...	...	...	...	...	...
567	927241	M	...	0.4087	0.12400
568	92751	B	...	0.2871	0.07039

569 rows × 32 columns

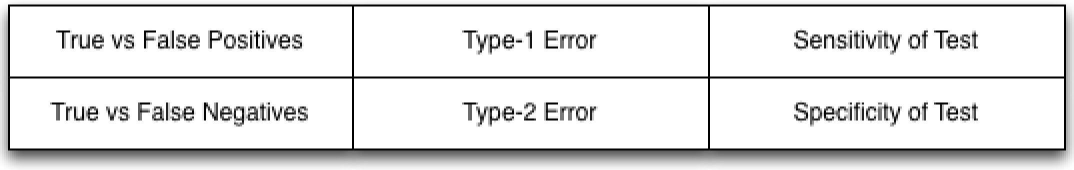
ROC curves can be a bit confusing. However, they are prevalent in analytics. It is essential to know how to read them. Even their name is confusing. Do not worry about their name; the receiver operating characteristic curve (ROC) comes from electrical engineering (EE).

Binary classification is common in medical testing. Often you want to diagnose if someone has a disease. This diagnosis can lead to two types of errors, known as false positives and false negatives:

- **False Positive** - Your test (neural network) indicated that the patient had the disease; however, the patient did not.
- **False Negative** - Your test (neural network) indicated that the patient did not have the disease; however, the patient did have the disease.
- **True Positive** - Your test (neural network) correctly identified that the patient had the disease.
- **True Negative** - Your test (neural network) correctly identified that the patient did not have the disease.

Figure 4.ETYP shows you these types of errors.

Figure 4.ETYP: Type of Error

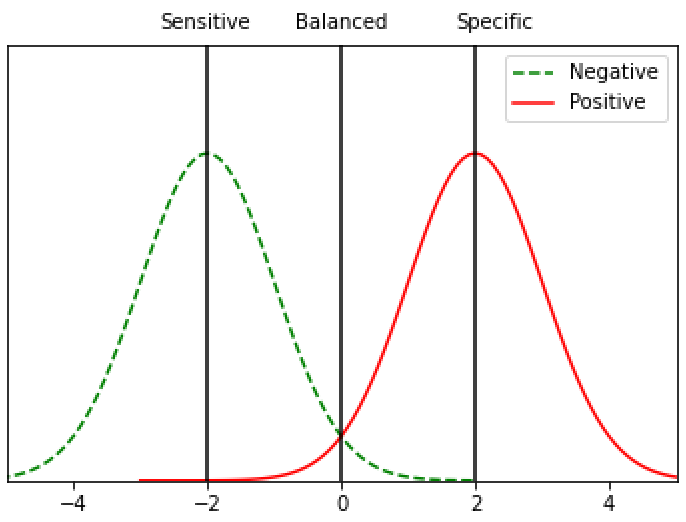


True vs False Positives	Type-1 Error	Sensitivity of Test
True vs False Negatives	Type-2 Error	Specificity of Test

Neural networks classify in terms of the probability of it being positive. However, at what possibility do you give a positive result? Is the cutoff 50%? 90%? Where you set, this cutoff is called the threshold. Anything above the cutoff is positive; anything below is negative. Setting this cutoff allows the model to be more sensitive or specific:

More info on Sensitivity vs. Specificity: Khan Academy

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
import math

mu1 = -2
mu2 = 2
variance = 1
sigma = math.sqrt(variance)
x1 = np.linspace(mu1 - 5*sigma, mu1 + 4*sigma, 100)
x2 = np.linspace(mu2 - 5*sigma, mu2 + 4*sigma, 100)
plt.plot(x1, stats.norm.pdf(x1, mu1, sigma)/1,color="green",
         linestyle='dashed')
plt.plot(x2, stats.norm.pdf(x2, mu2, sigma)/1,color="red")
plt.axvline(x=-2,color="black")
plt.axvline(x=0,color="black")
plt.axvline(x=+2,color="black")
plt.text(-2.7,0.55,"Sensitive")
plt.text(-0.7,0.55,"Balanced")
plt.text(1.7,0.55,"Specific")
plt.ylim([0,0.53])
plt.xlim([-5,5])
plt.legend(['Negative','Positive'])
plt.yticks([])
plt.show()
```



We will now train a neural network for the Wisconsin breast cancer dataset. We begin by preprocessing the data. Because we have all numeric data, we compute a z-score for each column.

In [4]:
```
from scipy.stats import zscore

x_columns = df.columns.drop('diagnosis').drop('id')
for col in x_columns:
    df[col] = zscore(df[col])

# Convert to numpy - Regression
x = df[x_columns].values
y = df['diagnosis'].map({'M':1,"B":0}).values # Binary classification,
                                              # M is 1 and B is 0
```

We can now define two functions. The first function plots a confusion matrix. The second function plots a ROC chart.

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# Plot a confusion matrix.
# cm is the confusion matrix, names are the names of the classes.
def plot_confusion_matrix(cm, names, title='Confusion matrix',
                            cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(names))
    plt.xticks(tick_marks, names, rotation=45)
    plt.yticks(tick_marks, names)
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```python
# Plot an ROC. pred - the predictions, y - the expected output.
def plot_roc(pred,y):
    fpr, tpr, _ = roc_curve(y, pred)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC)')
    plt.legend(loc="lower right")
    plt.show()
```

ROC Chart Example

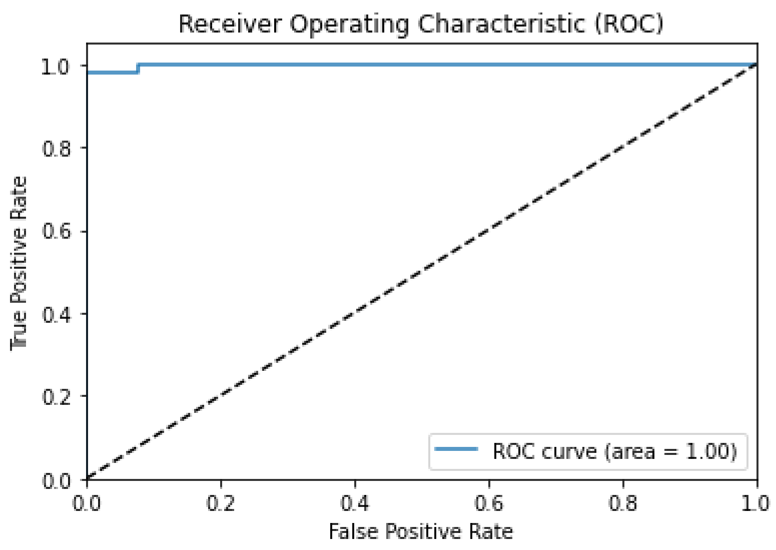
The following code demonstrates how to implement a ROC chart in Python.

In [6]:
```python
# Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
                kernel_initializer='random_normal'))
model.add(Dense(50,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(25,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(1,activation='sigmoid',kernel_initializer='random_normal'))
model.compile(loss='binary_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics =['accuracy'])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
    patience=5, verbose=1, mode='auto', restore_best_weights=True)

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

```
Epoch 1/1000
14/14 - 2s - loss: 0.6850 - accuracy: 0.8427 - val_loss: 0.6673 - val_accura
cy: 0.9441 - 2s/epoch - 164ms/step
Epoch 2/1000
14/14 - 0s - loss: 0.6261 - accuracy: 0.9319 - val_loss: 0.5393 - val_accura
cy: 0.9720 - 130ms/epoch - 9ms/step
Epoch 3/1000
14/14 - 0s - loss: 0.4265 - accuracy: 0.9413 - val_loss: 0.2536 - val_accura
cy: 0.9720 - 148ms/epoch - 11ms/step
Epoch 4/1000
14/14 - 0s - loss: 0.2112 - accuracy: 0.9437 - val_loss: 0.1067 - val_accura
cy: 0.9720 - 145ms/epoch - 10ms/step
Epoch 5/1000
14/14 - 0s - loss: 0.1171 - accuracy: 0.9624 - val_loss: 0.0644 - val_accura
cy: 0.9790 - 110ms/epoch - 8ms/step
Epoch 6/1000
14/14 - 0s - loss: 0.0852 - accuracy: 0.9789 - val_loss: 0.0552 - val_accura
cy: 0.9860 - 114ms/epoch - 8ms/step
Epoch 7/1000
14/14 - 0s - loss: 0.0744 - accuracy: 0.9789 - val_loss: 0.0541 - val_accura
cy: 0.9860 - 115ms/epoch - 8ms/step
Epoch 8/1000
14/14 - 0s - loss: 0.0662 - accuracy: 0.9812 - val_loss: 0.0473 - val_accura
cy: 0.9930 - 154ms/epoch - 11ms/step
Epoch 9/1000
14/14 - 0s - loss: 0.0602 - accuracy: 0.9812 - val_loss: 0.0493 - val_accura
cy: 0.9860 - 90ms/epoch - 6ms/step
Epoch 10/1000
14/14 - 0s - loss: 0.0548 - accuracy: 0.9859 - val_loss: 0.0468 - val_accura
cy: 0.9860 - 225ms/epoch - 16ms/step
Epoch 11/1000
14/14 - 0s - loss: 0.0491 - accuracy: 0.9836 - val_loss: 0.0484 - val_accura
cy: 0.9860 - 133ms/epoch - 10ms/step
Epoch 12/1000
14/14 - 0s - loss: 0.0458 - accuracy: 0.9836 - val_loss: 0.0486 - val_accura
cy: 0.9860 - 119ms/epoch - 8ms/step
Epoch 13/1000
Restoring model weights from the end of the best epoch: 8.
14/14 - 0s - loss: 0.0417 - accuracy: 0.9883 - val_loss: 0.0477 - val_accura
cy: 0.9860 - 124ms/epoch - 9ms/step
Epoch 13: early stopping
```

```
Out[6]: <keras.callbacks.History at 0x7f6a8aee46d0>
```

```
In [7]: pred = model.predict(x_test)
        plot_roc(pred,y_test)
```



Multiclass Classification Error Metrics

If you want to predict more than one outcome, you will need more than one output neuron. Because a single neuron can predict two results, a neural network with two output neurons is somewhat rare. If there are three or more outcomes, there will be three or more output neurons. The following sections will examine several metrics for evaluating classification error. We will assess the following classification neural network.

In [8]:
```
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
```

In [9]:
```
# Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
                kernel_initializer='random_normal'))
model.add(Dense(50,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(25,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(y.shape[1],activation='softmax',
                kernel_initializer='random_normal'))
model.compile(loss='categorical_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics =['accuracy'])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

```
Epoch 1/1000
47/47 - 2s - loss: 1.4456 - accuracy: 0.4753 - val_loss: 1.1348 - val_accura
cy: 0.4980 - 2s/epoch - 45ms/step
Epoch 2/1000
47/47 - 0s - loss: 1.1477 - accuracy: 0.4720 - val_loss: 1.0890 - val_accura
cy: 0.4980 - 249ms/epoch - 5ms/step
Epoch 3/1000
47/47 - 0s - loss: 1.0847 - accuracy: 0.5040 - val_loss: 1.0205 - val_accura
cy: 0.5380 - 482ms/epoch - 10ms/step
Epoch 4/1000
47/47 - 0s - loss: 0.9608 - accuracy: 0.5920 - val_loss: 0.9546 - val_accura
cy: 0.5740 - 309ms/epoch - 7ms/step
Epoch 5/1000
47/47 - 0s - loss: 0.8508 - accuracy: 0.6480 - val_loss: 0.8616 - val_accura
cy: 0.6600 - 290ms/epoch - 6ms/step
Epoch 6/1000
47/47 - 0s - loss: 0.7942 - accuracy: 0.6660 - val_loss: 0.8018 - val_accura
cy: 0.6900 - 298ms/epoch - 6ms/step
Epoch 7/1000
47/47 - 0s - loss: 0.7581 - accuracy: 0.6927 - val_loss: 0.8044 - val_accura
cy: 0.6740 - 271ms/epoch - 6ms/step
Epoch 8/1000
47/47 - 0s - loss: 0.7434 - accuracy: 0.6893 - val_loss: 0.7885 - val_accura
cy: 0.6660 - 246ms/epoch - 5ms/step
Epoch 9/1000
47/47 - 0s - loss: 0.7522 - accuracy: 0.6867 - val_loss: 0.7835 - val_accura
cy: 0.6720 - 281ms/epoch - 6ms/step
Epoch 10/1000
47/47 - 0s - loss: 0.7158 - accuracy: 0.6987 - val_loss: 0.7727 - val_accura
cy: 0.6840 - 327ms/epoch - 7ms/step
Epoch 11/1000
47/47 - 0s - loss: 0.7129 - accuracy: 0.6887 - val_loss: 0.7966 - val_accura
cy: 0.6820 - 231ms/epoch - 5ms/step
Epoch 12/1000
47/47 - 0s - loss: 0.7105 - accuracy: 0.6947 - val_loss: 0.7700 - val_accura
cy: 0.6620 - 239ms/epoch - 5ms/step
Epoch 13/1000
47/47 - 0s - loss: 0.7119 - accuracy: 0.6940 - val_loss: 0.7680 - val_accura
cy: 0.6700 - 254ms/epoch - 5ms/step
Epoch 14/1000
47/47 - 0s - loss: 0.6934 - accuracy: 0.7047 - val_loss: 0.7743 - val_accura
cy: 0.6600 - 289ms/epoch - 6ms/step
Epoch 15/1000
47/47 - 0s - loss: 0.6904 - accuracy: 0.7093 - val_loss: 0.7564 - val_accura
cy: 0.6860 - 266ms/epoch - 6ms/step
Epoch 16/1000
47/47 - 0s - loss: 0.6837 - accuracy: 0.7007 - val_loss: 0.7423 - val_accura
cy: 0.7000 - 297ms/epoch - 6ms/step
Epoch 17/1000
47/47 - 0s - loss: 0.6783 - accuracy: 0.7120 - val_loss: 0.7519 - val_accura
cy: 0.6840 - 258ms/epoch - 5ms/step
Epoch 18/1000
47/47 - 0s - loss: 0.6665 - accuracy: 0.7153 - val_loss: 0.7582 - val_accura
cy: 0.6660 - 259ms/epoch - 6ms/step
Epoch 19/1000
47/47 - 0s - loss: 0.6702 - accuracy: 0.7000 - val_loss: 0.7504 - val_accura
```

```
cy: 0.6880 - 271ms/epoch - 6ms/step
Epoch 20/1000
47/47 - 0s - loss: 0.6624 - accuracy: 0.7147 - val_loss: 0.7527 - val_accura
cy: 0.6800 - 328ms/epoch - 7ms/step
Epoch 21/1000
Restoring model weights from the end of the best epoch: 16.
47/47 - 1s - loss: 0.6558 - accuracy: 0.7160 - val_loss: 0.7653 - val_accura
cy: 0.6720 - 527ms/epoch - 11ms/step
Epoch 21: early stopping
```

Out[9]:
```
<keras.callbacks.History at 0x7f6a8ad5db50>
```

Calculate Classification Accuracy

Accuracy is the number of rows where the neural network correctly predicted the target class. Accuracy is only used for classification, not regression.

$$accuracy = \frac{c}{N}$$

Where c is the number correct and N is the size of the evaluated set (training or validation). Higher accuracy numbers are desired.

As we just saw, by default, Keras will return the percent probability for each class. We can change these prediction probabilities into the actual iris predicted with **argmax**.

In [10]:
```
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
# raw probabilities to chosen class (highest probability)
```

Now that we have the actual iris flower predicted, we can calculate the percent accuracy (how many were correctly classified).

In [11]:
```
from sklearn import metrics

y_compare = np.argmax(y_test,axis=1)
score = metrics.accuracy_score(y_compare, pred)
print("Accuracy score: {}".format(score))
```

```
Accuracy score: 0.7
```

Calculate Classification Log Loss

Accuracy is like a final exam with no partial credit. However, neural networks can predict a probability of each of the target classes. Neural networks will give high probabilities to predictions that are more likely. Log loss is an error metric that penalizes confidence in wrong answers. Lower log loss values are desired.

The following code shows the output of predict_proba:

```
In [12]: from IPython.display import display

         # Don't display numpy in scientific notation
         np.set_printoptions(precision=4)
         np.set_printoptions(suppress=True)

         # Generate predictions
         pred = model.predict(x_test)

         print("Numpy array of predictions")
         display(pred[0:5])

         print("As percent probability")
         print(pred[0]*100)

         score = metrics.log_loss(y_test, pred)
         print("Log loss score: {}".format(score))

         # raw probabilities to chosen class (highest probability)
         pred = np.argmax(pred,axis=1)
```

```
Numpy array of predictions
array([[0.    , 0.1201, 0.7286, 0.1494, 0.0018, 0.    , 0.    ],
       [0.    , 0.6962, 0.3016, 0.0001, 0.0022, 0.    , 0.    ],
       [0.    , 0.7234, 0.2708, 0.0003, 0.0053, 0.0001, 0.    ],
       [0.    , 0.3836, 0.6039, 0.0086, 0.0039, 0.    , 0.    ],
       [0.    , 0.0609, 0.6303, 0.3079, 0.001 , 0.    , 0.    ]],
      dtype=float32)
As percent probability
[ 0.0001 12.0143 72.8578 14.9446  0.1823  0.0009  0.0001]
Log loss score: 0.7423401429280638
```

Log loss is calculated as follows:

$$\text{log loss} = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i))$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable y-hat is the neural network’s prediction, and the variable y is the known correct answer. In this case, y will always be 0 or 1. The training data have no probabilities. The neural network classifies it either into one class (1) or the other (0).

The variable N represents the number of elements in the training set the number of questions in the test. We divide by N because this process is customary for an average. We also begin the equation with a negative because the log function is always negative over the domain 0 to 1. This negation allows a positive score for the training to minimize.

You will notice two terms are separated by the addition (+). Each contains a log function. Because y will be either 0 or 1, then one of these two terms will cancel out to 0. If y is 0, then the first term will reduce to 0. If y is 1, then the second term will be 0.

If your prediction for the first class of a two-class prediction is y-hat, then your prediction for the second class is 1 minus y-hat. Essentially, if your prediction for class A is 70% (0.7), then your prediction for class B is 30% (0.3). Your score will increase by the log of your prediction for the correct class. If the neural network had predicted 1.0 for class A, and the correct answer was A, your score would increase by log (1), which is 0. For log loss, we seek a low score, so a correct answer results in 0. Some of these log values for a neural network's probability estimate for the correct class:

- -log(1.0) = 0
- -log(0.95) = 0.02
- -log(0.9) = 0.05
- -log(0.8) = 0.1
- -log(0.5) = 0.3
- -log(0.1) = 1
- -log(0.01) = 2
- -log(1.0e-12) = 12
- -log(0.0) = negative infinity

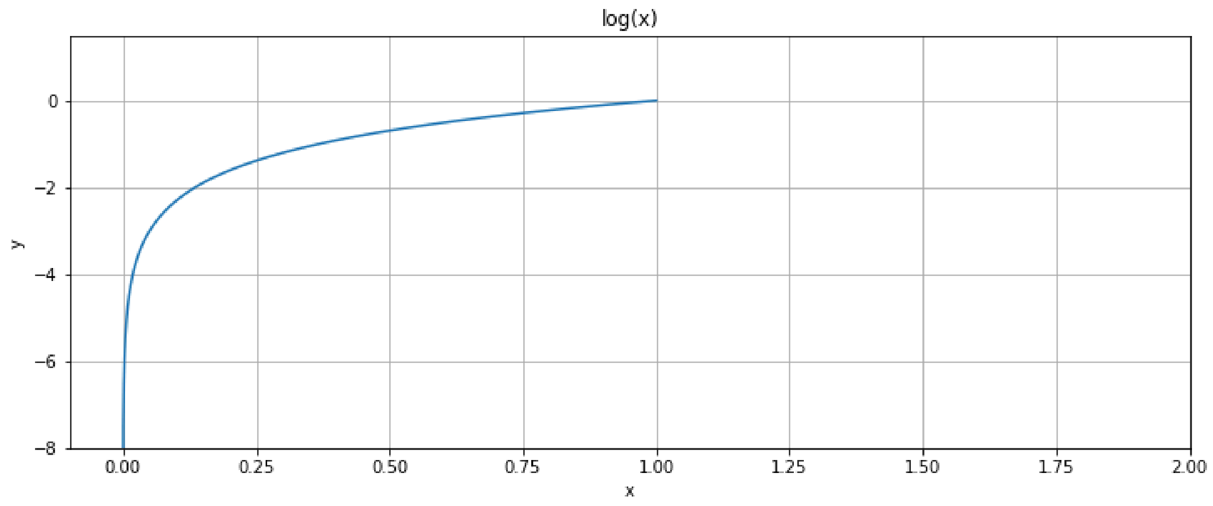
As you can see, giving a low confidence to the correct answer affects the score the most. Because log (0) is negative infinity, we typically impose a minimum value. Of course, the above log values are for a single training set element. We will average the log values for the entire training set.

The log function is useful to penalizing wrong answers. The following code demonstrates the utility of the log function:

```
In [13]: %matplotlib inline
from matplotlib.pyplot import figure, show
from numpy import arange, sin, pi

#t = arange(1e-5, 5.0, 0.00001)
#t = arange(1.0, 5.0, 0.00001) # computer scientists
t = arange(0.0, 1.0, 0.00001)  # data     scientists

fig = figure(1,figsize=(12, 10))

ax1 = fig.add_subplot(211)
ax1.plot(t, np.log(t))
ax1.grid(True)
ax1.set_ylim((-8, 1.5))
ax1.set_xlim((-0.1, 2))
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('log(x)')
```

```
show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:12: RuntimeWarn
ing: divide by zero encountered in log
  if sys.path[0] == '':
```

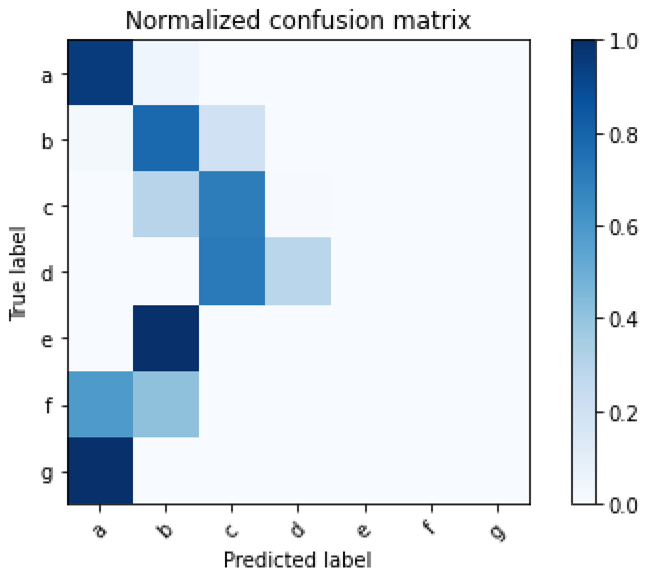


Confusion Matrix

A confusion matrix shows which predicted classes are often confused for the other classes. The vertical axis (y) represents the true labels and the horizontal axis (x) represents the predicted labels. When the true label and predicted label are the same, the highest values occur down the diagonal extending from the upper left to the lower right. The other values, outside the diagonal, represent incorrect predictions. For example, in the confusion matrix below, the value in row 2, column 1 shows how often the predicted value A occurred when it should have been B.

In [14]:
```
import numpy as np
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

# Compute confusion matrix
cm = confusion_matrix(y_compare, pred)
np.set_printoptions(precision=2)

# Normalize the confusion matrix by row (i.e by the number of samples
# in each class)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
print('Normalized confusion matrix')
print(cm_normalized)
plt.figure()
plot_confusion_matrix(cm_normalized, products,
        title='Normalized confusion matrix')

plt.show()
```

```
Normalized confusion matrix
[[0.95 0.05 0.   0.   0.   0.   0.  ]
 [0.02 0.78 0.2  0.   0.   0.   0.  ]
 [0.   0.29 0.7  0.01 0.   0.   0.  ]
 [0.   0.   0.71 0.29 0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.  ]
 [0.59 0.41 0.   0.   0.   0.   0.  ]
 [1.   0.   0.   0.   0.   0.   0.  ]]
```



```
In [14]:
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- **Part 4.3: Keras Regression for Deep Neural Networks with RMSE** [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [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 4.3: Keras Regression for Deep Neural Networks with RMSE

We evaluate regression results differently than classification. Consider the following code that trains a neural network for regression on the data set **jh-simple-dataset.csv**. We begin by preparing the data set.

In [2]:
```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# 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)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', 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['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)
```

Next, we create a neural network to fit the data we just loaded.

In [3]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

```
Train on 1500 samples, validate on 500 samples
Epoch 1/1000
1500/1500 - 1s - loss: 1905.4454 - val_loss: 1628.1341
Epoch 2/1000
1500/1500 - 0s - loss: 1331.4213 - val_loss: 889.0575
Epoch 3/1000
1500/1500 - 0s - loss: 554.8426 - val_loss: 303.7261
Epoch 4/1000
1500/1500 - 0s - loss: 276.2087 - val_loss: 241.2495
Epoch 5/1000
1500/1500 - 0s - loss: 232.2832 - val_loss: 208.2143
Epoch 6/1000
1500/1500 - 0s - loss: 198.5331 - val_loss: 179.5262
Epoch 7/1000
1500/1500 - 0s - loss: 169.0791 - val_loss: 154.5270
Epoch 8/1000
1500/1500 - 0s - loss: 144.1286 - val_loss: 132.8691
Epoch 9/1000
1500/1500 - 0s - loss: 122.9873 - val_loss: 115.0928
Epoch 10/1000
1500/1500 - 0s - loss: 104.7249 - val_loss: 98.7375
Epoch 11/1000
1500/1500 - 0s - loss: 89.8292 - val_loss: 86.2749
Epoch 12/1000
1500/1500 - 0s - loss: 77.3071 - val_loss: 75.0022
Epoch 13/1000
1500/1500 - 0s - loss: 67.0604 - val_loss: 66.1396
Epoch 14/1000
1500/1500 - 0s - loss: 58.9584 - val_loss: 58.4367
Epoch 15/1000
1500/1500 - 0s - loss: 51.2491 - val_loss: 52.7136
Epoch 16/1000
1500/1500 - 0s - loss: 45.1765 - val_loss: 46.5179
Epoch 17/1000
1500/1500 - 0s - loss: 39.8843 - val_loss: 41.3721
Epoch 18/1000
1500/1500 - 0s - loss: 35.1468 - val_loss: 37.2132
Epoch 19/1000
1500/1500 - 0s - loss: 31.1755 - val_loss: 33.0697
Epoch 20/1000
1500/1500 - 0s - loss: 27.6307 - val_loss: 30.3131
Epoch 21/1000
1500/1500 - 0s - loss: 24.8457 - val_loss: 26.9474
Epoch 22/1000
1500/1500 - 0s - loss: 22.4056 - val_loss: 24.3656
Epoch 23/1000
1500/1500 - 0s - loss: 20.3071 - val_loss: 22.1642
Epoch 24/1000
1500/1500 - 0s - loss: 18.5446 - val_loss: 20.4782
Epoch 25/1000
1500/1500 - 0s - loss: 17.1571 - val_loss: 18.8670
Epoch 26/1000
1500/1500 - 0s - loss: 15.9407 - val_loss: 17.6862
Epoch 27/1000
1500/1500 - 0s - loss: 14.9866 - val_loss: 16.5275
Epoch 28/1000
```

```
1500/1500 - 0s - loss: 14.1251 - val_loss: 15.6342
Epoch 29/1000
1500/1500 - 0s - loss: 13.4655 - val_loss: 14.8625
Epoch 30/1000
1500/1500 - 0s - loss: 12.8994 - val_loss: 14.2826
Epoch 31/1000
1500/1500 - 0s - loss: 12.5566 - val_loss: 13.6121
Epoch 32/1000
1500/1500 - 0s - loss: 12.0077 - val_loss: 13.3087
Epoch 33/1000
1500/1500 - 0s - loss: 11.5357 - val_loss: 12.6593
Epoch 34/1000
1500/1500 - 0s - loss: 11.2365 - val_loss: 12.1849
Epoch 35/1000
1500/1500 - 0s - loss: 10.8074 - val_loss: 11.9388
Epoch 36/1000
1500/1500 - 0s - loss: 10.5593 - val_loss: 11.4006
Epoch 37/1000
1500/1500 - 0s - loss: 10.2093 - val_loss: 10.9751
Epoch 38/1000
1500/1500 - 0s - loss: 9.8386 - val_loss: 10.8651
Epoch 39/1000
1500/1500 - 0s - loss: 9.5938 - val_loss: 10.5728
Epoch 40/1000
1500/1500 - 0s - loss: 9.1488 - val_loss: 9.8661
Epoch 41/1000
1500/1500 - 0s - loss: 8.8920 - val_loss: 9.5228
Epoch 42/1000
1500/1500 - 0s - loss: 8.5156 - val_loss: 9.1506
Epoch 43/1000
1500/1500 - 0s - loss: 8.2628 - val_loss: 8.9486
Epoch 44/1000
1500/1500 - 0s - loss: 7.9219 - val_loss: 8.5034
Epoch 45/1000
1500/1500 - 0s - loss: 7.7077 - val_loss: 8.0760
Epoch 46/1000
1500/1500 - 0s - loss: 7.3165 - val_loss: 7.6620
Epoch 47/1000
1500/1500 - 0s - loss: 7.0259 - val_loss: 7.4933
Epoch 48/1000
1500/1500 - 0s - loss: 6.7422 - val_loss: 7.0583
Epoch 49/1000
1500/1500 - 0s - loss: 6.5163 - val_loss: 6.8024
Epoch 50/1000
1500/1500 - 0s - loss: 6.2633 - val_loss: 7.3045
Epoch 51/1000
1500/1500 - 0s - loss: 6.0029 - val_loss: 6.2712
Epoch 52/1000
1500/1500 - 0s - loss: 5.6791 - val_loss: 5.9342
Epoch 53/1000
1500/1500 - 0s - loss: 5.4798 - val_loss: 6.0110
Epoch 54/1000
1500/1500 - 0s - loss: 5.2115 - val_loss: 5.3928
Epoch 55/1000
1500/1500 - 0s - loss: 4.9592 - val_loss: 5.2215
Epoch 56/1000
```

```
1500/1500 - 0s - loss: 4.7189 - val_loss: 5.0103
Epoch 57/1000
1500/1500 - 0s - loss: 4.4683 - val_loss: 4.7098
Epoch 58/1000
1500/1500 - 0s - loss: 4.2650 - val_loss: 4.5259
Epoch 59/1000
1500/1500 - 0s - loss: 4.0953 - val_loss: 4.4263
Epoch 60/1000
1500/1500 - 0s - loss: 3.8027 - val_loss: 4.1103
Epoch 61/1000
1500/1500 - 0s - loss: 3.5759 - val_loss: 3.7770
Epoch 62/1000
1500/1500 - 0s - loss: 3.3755 - val_loss: 3.5737
Epoch 63/1000
1500/1500 - 0s - loss: 3.1781 - val_loss: 3.4833
Epoch 64/1000
1500/1500 - 0s - loss: 3.0001 - val_loss: 3.2246
Epoch 65/1000
1500/1500 - 0s - loss: 2.7691 - val_loss: 3.1021
Epoch 66/1000
1500/1500 - 0s - loss: 2.6227 - val_loss: 2.8215
Epoch 67/1000
1500/1500 - 0s - loss: 2.4682 - val_loss: 2.7528
Epoch 68/1000
1500/1500 - 0s - loss: 2.3243 - val_loss: 2.5394
Epoch 69/1000
1500/1500 - 0s - loss: 2.1664 - val_loss: 2.3886
Epoch 70/1000
1500/1500 - 0s - loss: 2.0377 - val_loss: 2.2536
Epoch 71/1000
1500/1500 - 0s - loss: 1.8845 - val_loss: 2.2354
Epoch 72/1000
1500/1500 - 0s - loss: 1.7931 - val_loss: 2.0831
Epoch 73/1000
1500/1500 - 0s - loss: 1.6889 - val_loss: 1.8866
Epoch 74/1000
1500/1500 - 0s - loss: 1.5820 - val_loss: 1.7964
Epoch 75/1000
1500/1500 - 0s - loss: 1.5085 - val_loss: 1.7138
Epoch 76/1000
1500/1500 - 0s - loss: 1.4159 - val_loss: 1.6468
Epoch 77/1000
1500/1500 - 0s - loss: 1.3606 - val_loss: 1.5906
Epoch 78/1000
1500/1500 - 0s - loss: 1.2652 - val_loss: 1.5063
Epoch 79/1000
1500/1500 - 0s - loss: 1.1937 - val_loss: 1.4506
Epoch 80/1000
1500/1500 - 0s - loss: 1.1180 - val_loss: 1.4817
Epoch 81/1000
1500/1500 - 0s - loss: 1.1412 - val_loss: 1.2800
Epoch 82/1000
1500/1500 - 0s - loss: 1.0385 - val_loss: 1.2412
Epoch 83/1000
1500/1500 - 0s - loss: 0.9846 - val_loss: 1.1891
Epoch 84/1000
```

```
1500/1500 - 0s - loss: 0.9937 - val_loss: 1.1322
Epoch 85/1000
1500/1500 - 0s - loss: 0.8915 - val_loss: 1.0847
Epoch 86/1000
1500/1500 - 0s - loss: 0.8562 - val_loss: 1.1110
Epoch 87/1000
1500/1500 - 0s - loss: 0.8468 - val_loss: 1.0686
Epoch 88/1000
1500/1500 - 0s - loss: 0.7947 - val_loss: 0.9805
Epoch 89/1000
1500/1500 - 0s - loss: 0.7807 - val_loss: 0.9463
Epoch 90/1000
1500/1500 - 0s - loss: 0.7502 - val_loss: 0.9965
Epoch 91/1000
1500/1500 - 0s - loss: 0.7529 - val_loss: 0.9532
Epoch 92/1000
1500/1500 - 0s - loss: 0.6857 - val_loss: 0.8712
Epoch 93/1000
1500/1500 - 0s - loss: 0.6717 - val_loss: 0.8498
Epoch 94/1000
1500/1500 - 0s - loss: 0.6869 - val_loss: 0.8518
Epoch 95/1000
1500/1500 - 0s - loss: 0.6626 - val_loss: 0.8275
Epoch 96/1000
1500/1500 - 0s - loss: 0.6308 - val_loss: 0.7850
Epoch 97/1000
1500/1500 - 0s - loss: 0.6056 - val_loss: 0.7708
Epoch 98/1000
1500/1500 - 0s - loss: 0.5991 - val_loss: 0.7643
Epoch 99/1000
1500/1500 - 0s - loss: 0.6102 - val_loss: 0.8104
Epoch 100/1000
1500/1500 - 0s - loss: 0.5647 - val_loss: 0.7227
Epoch 101/1000
1500/1500 - 0s - loss: 0.5474 - val_loss: 0.7107
Epoch 102/1000
1500/1500 - 0s - loss: 0.5395 - val_loss: 0.6847
Epoch 103/1000
1500/1500 - 0s - loss: 0.5350 - val_loss: 0.7383
Epoch 104/1000
1500/1500 - 0s - loss: 0.5551 - val_loss: 0.6698
Epoch 105/1000
1500/1500 - 0s - loss: 0.5032 - val_loss: 0.6520
Epoch 106/1000
1500/1500 - 0s - loss: 0.5418 - val_loss: 0.7518
Epoch 107/1000
1500/1500 - 0s - loss: 0.4949 - val_loss: 0.6307
Epoch 108/1000
1500/1500 - 0s - loss: 0.5166 - val_loss: 0.6741
Epoch 109/1000
1500/1500 - 0s - loss: 0.4992 - val_loss: 0.6195
Epoch 110/1000
1500/1500 - 0s - loss: 0.4610 - val_loss: 0.6268
Epoch 111/1000
1500/1500 - 0s - loss: 0.4554 - val_loss: 0.5956
Epoch 112/1000
```

```
1500/1500 - 0s - loss: 0.4704 - val_loss: 0.5977
Epoch 113/1000
1500/1500 - 0s - loss: 0.4687 - val_loss: 0.5736
Epoch 114/1000
1500/1500 - 0s - loss: 0.4497 - val_loss: 0.5817
Epoch 115/1000
1500/1500 - 0s - loss: 0.4326 - val_loss: 0.5833
Epoch 116/1000
1500/1500 - 0s - loss: 0.4181 - val_loss: 0.5738
Epoch 117/1000
1500/1500 - 0s - loss: 0.4252 - val_loss: 0.5688
Epoch 118/1000
1500/1500 - 0s - loss: 0.4675 - val_loss: 0.5680
Epoch 119/1000
1500/1500 - 0s - loss: 0.4328 - val_loss: 0.5463
Epoch 120/1000
1500/1500 - 0s - loss: 0.4091 - val_loss: 0.5912
Epoch 121/1000
1500/1500 - 0s - loss: 0.4047 - val_loss: 0.5459
Epoch 122/1000
1500/1500 - 0s - loss: 0.4456 - val_loss: 0.5509
Epoch 123/1000
1500/1500 - 0s - loss: 0.4081 - val_loss: 0.5540
Epoch 124/1000
Restoring model weights from the end of the best epoch.
1500/1500 - 0s - loss: 0.4353 - val_loss: 0.5538
Epoch 00124: early stopping
```

```
Out[3]: <tensorflow.python.keras.callbacks.History at 0x1a40e6b0d0>
```

Mean Square Error

The mean square error (MSE) is the sum of the squared differences between the prediction ($\hat{y}$) and the expected (y). MSE values are not of a particular unit. If an MSE value has decreased for a model, that is good. However, beyond this, there is not much more you can determine. We seek to achieve low MSE values. The following equation demonstrates how to calculate MSE.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}_i - y_i\right)^2$$

The following code calculates the MSE on the predictions from the neural network.

```
In [4]: from sklearn import metrics

        # Predict
        pred = model.predict(x_test)

        # Measure MSE error.
        score = metrics.mean_squared_error(pred,y_test)
        print("Final score (MSE): {}".format(score))
```

```
Final score (MSE): 0.5463447829677607
```

Root Mean Square Error

The root mean square (RMSE) is essentially the square root of the MSE. Because of this, the RMSE error is in the same units as the training data outcome. We desire Low RMSE values. The following equation calculates RMSE.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2}$$

In [5]:
```python
import numpy as np

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 0.7391513938076291
```

Lift Chart

We often visualize the results of regression with a lift chart. To generate a lift chart, perform the following activities:

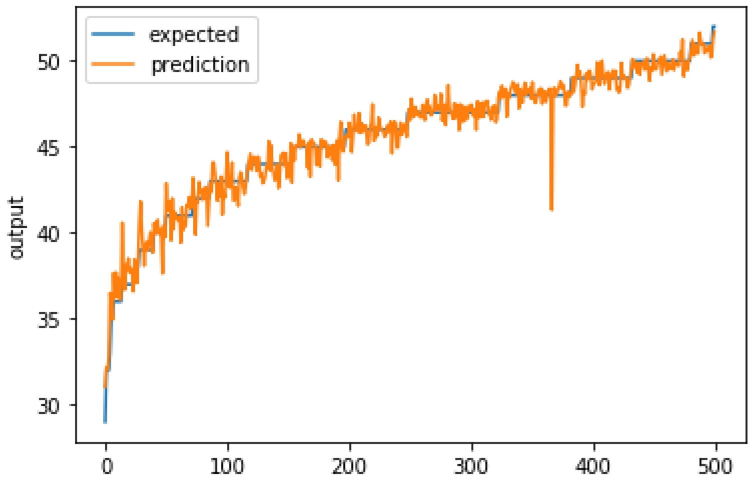
- Sort the data by expected output and plot these values.
- For every point on the x-axis, plot that same data point's predicted value in another color.
- The x-axis is just 0 to 100% of the dataset. The expected always starts low and ends high.
- The y-axis is ranged according to the values predicted.

You can interpret the lift chart as follows:

- The expected and predict lines should be close. Notice where one is above the other.
- The below chart is the most accurate for lower ages.

In [7]:
```python
# Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()
```

```
# Plot the chart
chart_regression(pred.flatten(),y_test)
```



```
In [ ]:
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- **Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training** [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [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 4.4: Training Neural Networks

Backpropagation [Cite:rumelhart1986learning] is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Researchers have extended classic backpropagation and modified to give rise to many different training algorithms. This section will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means of determining a neural network's weights during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ, theta) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

We repeat this process for every iteration(t). The training algorithm determines how we calculate the weight change. Classic backpropagation calculates a gradient (∇, nabla) for every weight in the neural network for the neural network's error function (J). We scale the gradient by a learning rate (η, eta).

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})$$

The learning rate is an important concept for backpropagation training. Setting the learning rate can be complex:

- Too low a learning rate will usually converge to a reasonable solution; however, the process will be prolonged.
- Too high of a learning rate will either fail outright or converge to a higher error than a better learning rate.

Common values for learning rate are: 0.1, 0.01, 0.001, etc.

Backpropagation is a gradient descent type, and many texts will use these two terms interchangeably. Gradient descent refers to calculating a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will indicate how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the expected output. We can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

The gradient is the partial derivative of each weight in the neural network concerning the error function. Each weight has a gradient that is the slope of the error function. Weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

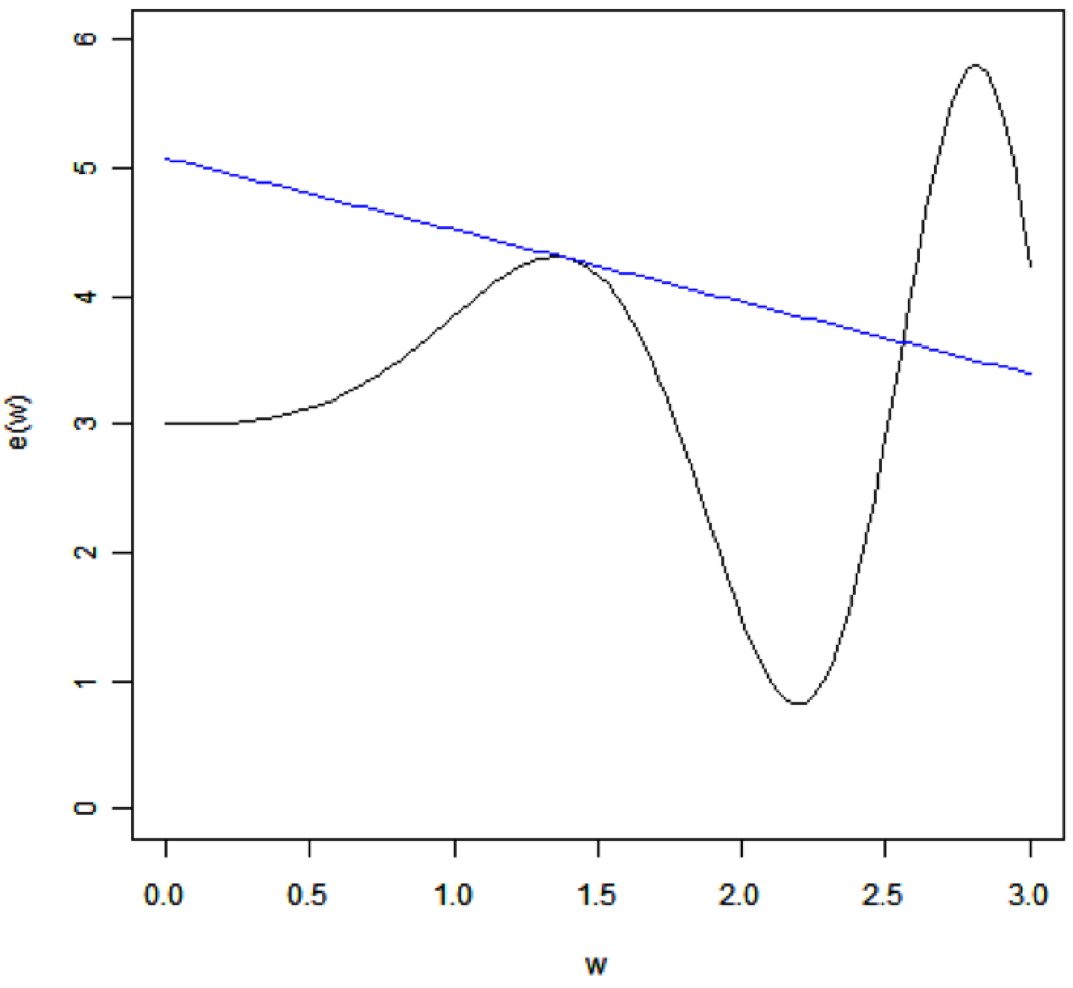
- Zero gradient - The weight does not contribute to the neural network's error.
- Negative gradient - The algorithm should increase the weight to lower error.
- Positive gradient - The algorithm should decrease the weight to lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process. First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had infinite computation resources, we would try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some shortcuts to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 4.DRV is a graph that demonstrates the error for a single weight:

Figure 4.DRV: Derivative



Looking at this chart, you can easily see that the optimal weight is where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The straight line barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error. The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight.

Derivatives are one of the most fundamental concepts in calculus. For this book, you need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to adjust the weight for a lower error. Using our working definition of the gradient, we will show how to calculate it.

Momentum Backpropagation

Momentum adds another term to the calculation of v_t:

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1}) + \lambda v_{t-1}$$

Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η, eta) and momentum (λ, lambda). Momentum adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount(v_t).

This technique has the effect of adding additional force behind the direction a weight is moving. Figure 4.MTM shows how this might allow the weight to escape local minima.

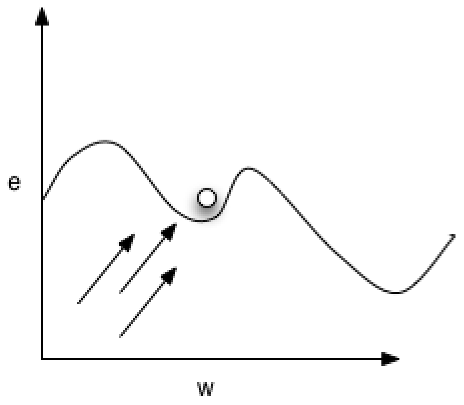


Figure 4.MTM: Momentum

A typical value for momentum is 0.9.

Batch and Online Backpropagation

How often should the weights of a neural network be updated? We can calculate gradients for a training set element. These gradients can also be summed together into batches, and the weights updated once per batch.

- **Online Training** - Update the weights based on gradients calculated from a single training set element.
- **Batch Training** - Update the weights based on the sum of the gradients over all training set elements.
- **Batch Size** - Update the weights based on the sum of some batch size of training set elements.

- **Mini-Batch Training** - The same as batch size, but with minimal batch size. Mini-batches are very popular, often in the 32-64 element range.

Because the batch size is smaller than the full training set size, it may take several batches to make it completely through the training set.

- **Step/Iteration** - The number of processed batches.
- **Epoch** - The number of times the algorithm processed the complete training set.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is currently one of the most popular neural network training algorithms. It works very similarly to Batch/Mini-Batch training, except that the batches are made up of a random set of training elements.

This technique leads to a very irregular convergence in error during training, as shown in Figure 4.SGD.

Figure 4.SGD: SGD Error

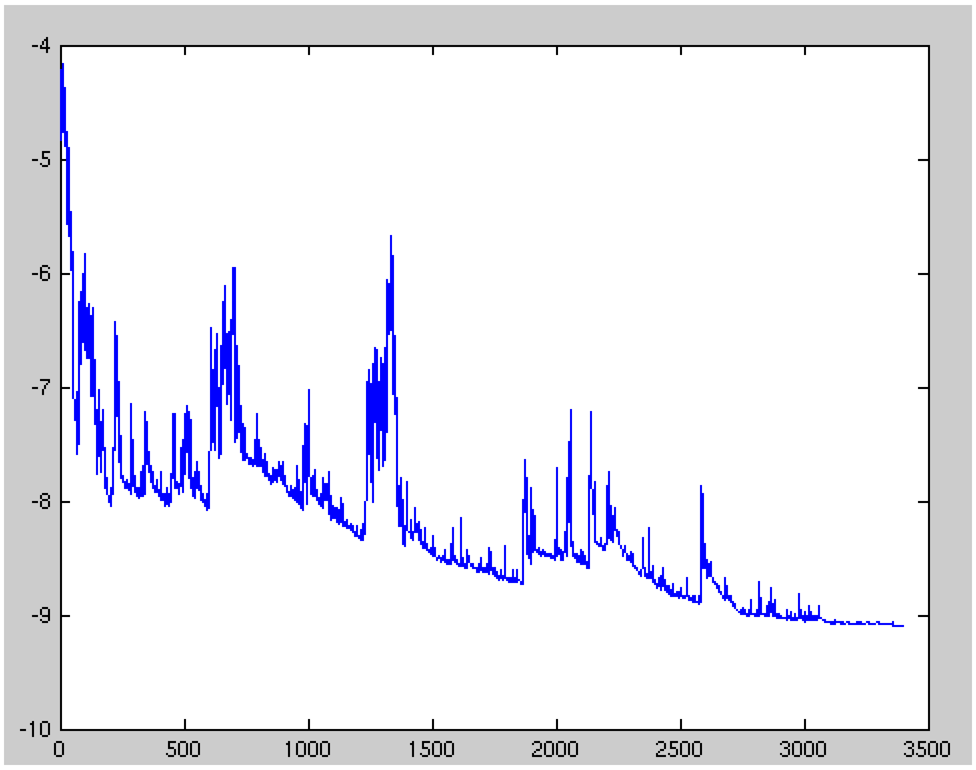


Image from Wikipedia

Because the neural network is trained on a random sample of the complete training set each time, the error does not make a smooth transition downward. However, the error usually does go down.

Advantages to SGD include:

- Computationally efficient. Each training step can be relatively fast, even with a huge training set.
- Decreases overfitting by focusing on only a portion of the training set each step.

Other Techniques

One problem with simple backpropagation training algorithms is that they are susceptible to learning rate and momentum. This technique is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima yet small enough not to destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often helpful to automatically decrease the learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Other training techniques:

- **Resilient Propagation** - Use only the magnitude of the gradient and allow each neuron to learn at its rate. There is no need for learning rate/momentum; however, it only works in full batch mode.
- **Nesterov accelerated gradient** - Helps mitigate the risk of choosing a bad mini-batch.
- **Adagrad** - Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta** - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods** - Non-gradient methods can *sometimes* be useful, though rarely outperform gradient-based backpropagation methods. These include: simulated annealing, genetic algorithms, particle swarm optimization, Nelder Mead, and many more.

ADAM Update

ADAM is the first training algorithm you should try. It is very effective. Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates. [Cite:kingma2014adam] Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

The values m_t and v_t are estimates of the gradients' first moment (the mean) and the second moment (the uncentered variance). However, they will be strongly biased towards zero in the initial training cycles. The first moment’s bias is corrected as follows.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

Similarly, the second moment is also corrected:

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \eta} \hat{m}_t$$

Adam is very tolerant to initial learning rate (\alpha) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1, 0.999 for β_2, and 10-8 for η.

Methods Compared

The following image shows how each of these algorithms train. It is animated, so it is not displayed in the printed book, but can be accessed from here: https://bit.ly/3kykkbn.

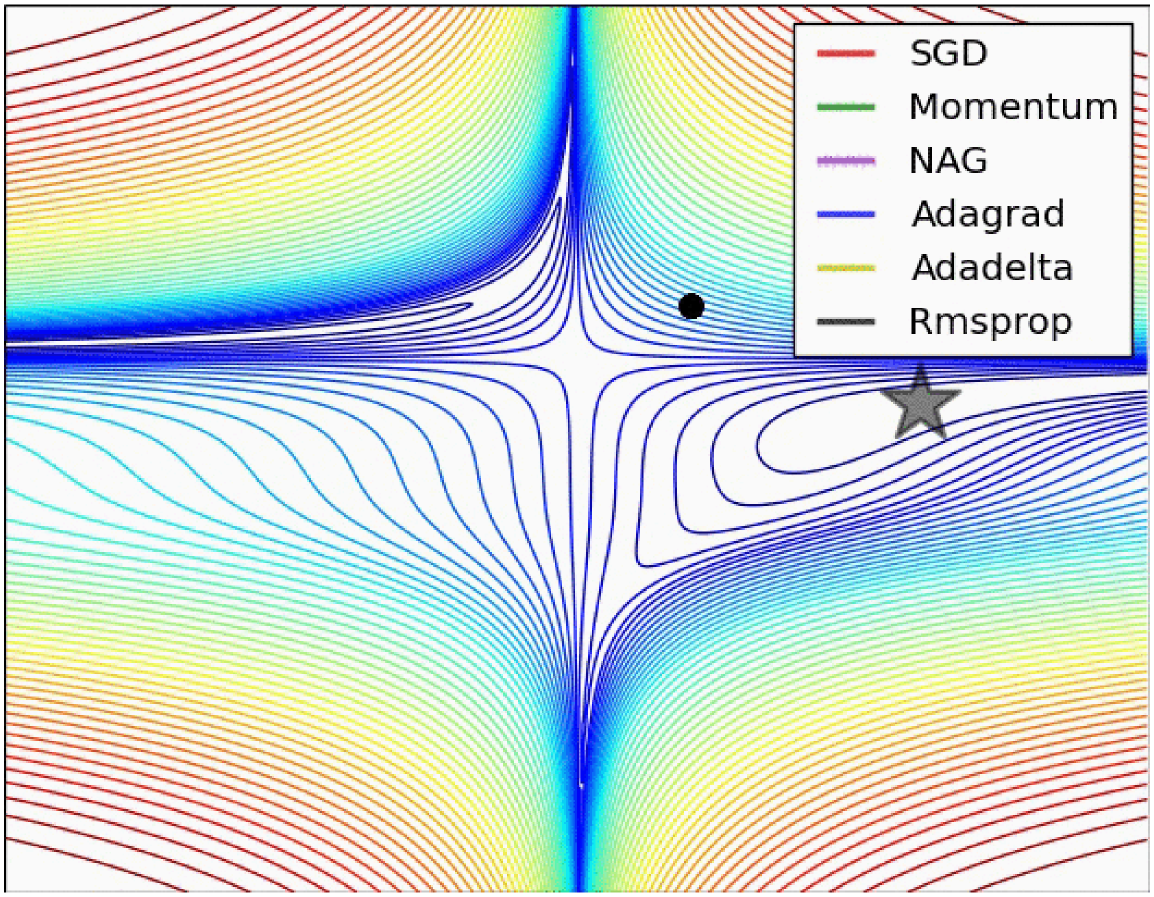


Image credits: Alec Radford

Specifying the Update Rule in Keras

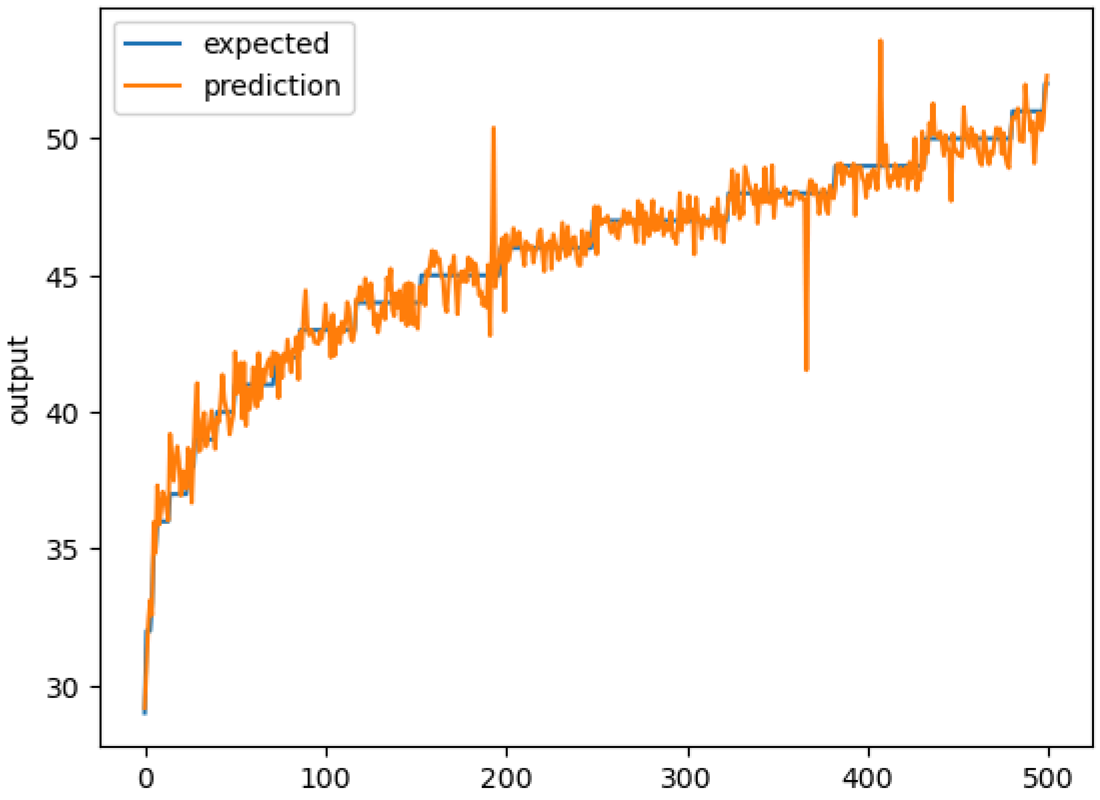
TensorFlow allows the update rule to be set to one of:

- Adagrad
- **Adam**
- Ftrl
- Momentum
- RMSProp
- **SGD**

In [2]:
```
%matplotlib inline

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt


# Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', 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['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam') # Modify here
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
```

```python
# Plot the chart
pred = model.predict(x_test)
chart_regression(pred.flatten(),y_test)
```

```
2024-02-14 00:07:14.691344: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2024-02-14 00:07:16.219137: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2024-02-14 00:07:16.221467: I tensorflow/core/common_runtime/process_util.c
c:146] Creating new thread pool with default inter op setting: 2. Tune using
inter_op_parallelism_threads for best performance.
Restoring model weights from the end of the best epoch: 155.
Epoch 160: early stopping
16/16 [==============================] - 0s 2ms/step
```



```
In [ ]:
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- **Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch** [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 4.5: Error Calculation from Scratch

We will now look at how to calculate RMSE and logloss by hand. RMSE is typically used for regression. We begin by calculating RMSE with libraries.

```
In [2]: from sklearn import metrics
        import numpy as np

        predicted = [1.1,1.9,3.4,4.2,4.3]
        expected = [1,2,3,4,5]

        score_mse = metrics.mean_squared_error(predicted,expected)
        score_rmse = np.sqrt(score_mse)
        print("Score (MSE): {}".format(score_mse))
        print("Score (RMSE): {}".format(score_rmse))
```

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

We can also calculate without libraries.

```
In [3]: score_mse = ((predicted[0]-expected[0])**2 + (predicted[1]-expected[1])**2
        + (predicted[2]-expected[2])**2 + (predicted[3]-expected[3])**2
        + (predicted[4]-expected[4])**2)/len(predicted)
        score_rmse = np.sqrt(score_mse)

        print("Score (MSE): {}".format(score_mse))
        print("Score (RMSE): {}".format(score_rmse))
```

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

Classification

We will now look at how to calculate a logloss by hand. For this, we look at a binary prediction. The predicted is some number between 0-1 that indicates the probability true (1). The expected is always 0 or 1. Therefore, a prediction of 1.0 is completely correct if the expected is 1 and completely wrong if the expected is 0.

```
In [4]: from sklearn import metrics

        expected = [1,1,0,0,0]
        predicted = [0.9,0.99,0.1,0.05,0.06]

        print(metrics.log_loss(expected,predicted))
```

```
0.06678801305495843
```

Now we attempt to calculate the same logloss manually.

```
In [5]: import numpy as np

        score_logloss = (np.log(1.0-np.abs(expected[0]-predicted[0]))+\
        np.log(1.0-np.abs(expected[1]-predicted[1]))+\
np.log(1.0-np.abs(expected[2]-predicted[2]))+\
np.log(1.0-np.abs(expected[3]-predicted[3]))+\
np.log(1.0-np.abs(expected[4]-predicted[4])))\
*(-1/len(predicted))

print(f'Score Logloss {score_logloss}')
```

```
Score Logloss 0.06678801305495843
```

```
In [ ]:
```