CODE

```
# Backprop on the Seeds Dataset
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp

# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
	minmax = list()
	stats = [[min(column), max(column)] for column in zip(*dataset)]
	return stats

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
		for i in range(len(row)-1):
			row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, *args):
	train = dataset[:180]
	test = dataset[180:]
	print("Train data")
	print(train)
```

```
	print("\n\nTest data")
	print(test)
	predicted = algorithm(train,test,*args)
	actual = [row[-1] for row in test]
	accuracy = accuracy_metric(actual, predicted)
	print("Actual\n",actual)
	print("predicted\n",predicted)
	return accuracy

# Calculate neuron activation for an input
def activate(weights, inputs):
	activation = weights[-1]
	for i in range(len(weights)-1):
		activation += weights[i] * inputs[i]
	return activation

# Transfer neuron activation
def transfer(activation):
	return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
	inputs = row
	for layer in network:
		new_inputs = []
		for neuron in layer:
			activation = activate(neuron['weights'], inputs)
			neuron['output'] = transfer(activation)
			new_inputs.append(neuron['output'])
		inputs = new_inputs
	return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
	return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
	for i in reversed(range(len(network))):
		layer = network[i]
		errors = list()
		if i != len(network)-1:
			for j in range(len(layer)):
				error = 0.0
				for neuron in network[i + 1]:
					error += (neuron['weights'][j] * neuron['delta'])
				errors.append(error)
		else:
			for j in range(len(layer)):
				neuron = layer[j]
				errors.append(expected[j] - neuron['output'])
		for j in range(len(layer)):
			neuron = layer[j]
			neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
	for i in range(len(network)):
		inputs = row[:-1]
		if i != 0:
			inputs = [neuron['output'] for neuron in network[i - 1]]
		for neuron in network[i]:
			for j in range(len(inputs)):
```

```
                    neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
                neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network

# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
    n_inputs = len(train[0]) - 1
    n_outputs = len(set([row[-1] for row in train]))
    network = initialize_network(n_inputs, n_hidden, n_outputs)
    train_network(network, train, l_rate, n_epoch, n_outputs)
    predictions = list()
    for row in test:
        prediction = predict(network, row[:-1])
        predictions.append(prediction)
    return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
filename = 'seeds.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)

# evaluate algorithm
l_rate = 0.5
n_epoch = 20
n_hidden = 5
accuracy = evaluate_algorithm(dataset, back_propagation, l_rate, n_epoch, n_hidden)
print('Accuracy: %.3f%%' % accuracy)
```

OUTPUT

Train data
[[0.4409820585457979, 0.5020661157024793, 0.570780399274047, 0.48648648648648646, 0.48610121168923714, 0.1893016422005273, 0.345150172328901 9, 0],
[0.40509915014164316, 0.44628099173553726, 0.6624319419237747, 0.3688063063063065, 0.5010691375623664, 0.03288301759221938, 0.21516494337764666, 0],
[0.3493862134088762, 0.3471074380165289, 0.8793103448275864, 0.22072072072072094, 0.50392017106201, 0.25145301590191005, 0.15066469719350079, 0],
[0.3068932955618508, 0.31611570247933873, 0.7931034482758617, 0.23930180180180172, 0.5338560228082679, 0.19424254638598865, 0.14081733136386, 0],
. . . .
[0.014164305949008532, 0.0661157024793389, 0.22504537205081615, 0.13851351351351326, 0.008553100498930866, 0.5118906759937069, 0.21861152141802068, 1],
[0.08404154863078381, 0.1322314049586778, 0.35571687840290406, 0.15822072072072058, 0.09123307198859591, 0.6645386105657336, 0.23781388478581966, 1],
[0.15297450424929188, 0.21900826446281002, 0.33756805807622525, 0.257882882882883, 0.18745545260156793, 0.11648831736207728, 0.3244707040866568, 1]]

Test data
[[0.07743153918791315, 0.11157024793388412, 0.43466424682395605, 0.10754504504504496, 0.10334996436208123, 0.5450467435540703, 0.15066469719350079, 1],
[0.1765816808309727, 0.2066115702479339, 0.5671506352087116, 0.18975225225225212, 0.27583749109052025, 0.5489474573847014, 0.30920728705071404, 1],
[0.15108593012275728, 0.19628099173553704, 0.451905626134301, 0.19200450450450463, 0.1988595866001424, 0.5320443641186338, 0.31462333825701644, 1],
[0.10009442870632677, 0.1363636363636364, 0.44827586206896564, 0.11768018018017999, 0.15680684248039922, 0.5778127397313708, 0.30329886755292945, 1],
[0.2171860245514637, 0.28099173553719, 0.41742286751361113, 0.33558558558558566, 0.2822523164647183, 0.7047159630212328, 0.39241752831117666, 1],
[0.11803588290840415, 0.16528925619834725, 0.3992740471869323, 0.15540540540540532, 0.1468282252316464, 0.3683444070264859, 0.25849335302806475, 1],
[0.16147308781869696, 0.19214876033057846, 0.5471869328493641, 0.19369369369369388, 0.2451888809693515, 0.6334629237150399, 0.26784835056622336, 1]]

('Actual\n', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
('predicted\n', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1])

Accuracy: 96.667%