```
import warnings
warnings.filterwarnings('ignore')
```

3 Layer Feed Forward Network Implementation

```
class FeedForwardNet(object):
  """
  A Simple 3 Layer Feed Forward Neural Network
  """
  def __init__(self, n_input=None, n_hidden=None, n_output=None, h_activation='tanh'):
    self.n_input = n_input
    self.n_hidden = n_hidden
    self.n_output = n_output

    self.W_h = 2*np.random.randn(self.n_input, self.n_hidden)-1 # Initial weights betwe
    self.b_h = np.zeros(self.n_hidden)  # Initial bias between Input and Hidden Layer
    self.W_o = 2*np.random.randn(self.n_hidden, self.n_output)-1 # Initial weights betw
    self.b_o = np.zeros(self.n_output)# Initial weights between Hidden and Output Layer

    # Computed during forward pass
    self.z_h = None # Hidden layer linear output
    self.a_h = None # Hidden layer activation
    self.z_o = None # Final layer linear output
    self.a_o = None # Final layer activation

    #Computed during backward pass
    self.dW_h = None # Hidden Layer Weight Gradients
    self.db_h = None # Hidden Layer Bias Gradients
    self.dW_o = None # Output Layer Weight Gradients
    self.db_o = None # Output Layer Bias Gradients

    self.h_activation = h_activation
    self.history = []

  def predict(self, X):
    probs = self.forward(X)
    return np.argmax(probs[0], axis=1)

  def forward(self, X):
    self.z_h = X.dot(self.W_h) + self.b_h # Hidden Layer Output
    self.a_h = self.hidden_layer_activation(self.z_h) # Hidden Layer Activations
    self.z_o = self.a_h.dot(self.W_o) + self.b_o # Final Layer Output
    self.a_o = self.softmax(self.z_o) # Final Layer Activations

    probs = self.a_o
    return probs

  def backprop(self, X, y):
    probs = self.a_o
    dL_o = self.cross_entropy_derivative(probs, y)
    self.dW_o = (self.a_h.T).dot(dL_o)
    self.db_o = np.sum(dL_o, axis=0)

    dL_h = dL_o.dot(self.W_o.T) * self.hidden_layer_activation_derivative(self.z_h)
    self.dW_h = np.dot(X.T, dL_h)
    self.db_h = np.sum(dL_h, axis=0)


  def softmax(self, x):
    scores = np.exp(x - np.max(x)) # For numerical stability
    probs = scores / np.sum(scores, axis=1, keepdims=True)
    return probs

  def hidden_layer_activation(self, x):
    if self.h_activation == 'relu':
      return self.relu(x)
    elif self.h_activation == 'tanh':
      return self.tanh(x)
```

```
    elif self.h_activation == 'sigmoid':
      return self.sigmoid(x)
    else:
      raise NotImplementedError

  def hidden_layer_activation_derivative(self, x):
    if self.h_activation == 'relu':
      return self.relu_derivative(x)
    elif self.h_activation == 'tanh':
      return self.tanh_derivative(x)
    elif self.h_activation == 'sigmoid':
      return self.sigmoid_derivative(x)
    else:
      raise NotImplementedError

  def tanh(self, x):
    return np.tanh(x)

  def tanh_derivative(self, x):
    return (1 - np.power(self.tanh(x), 2))

  def sigmoid(self, x):
    return 1. / (1 + np.exp(-x))

  def sigmoid_derivative(self, x):
    return x * (1. - x)

  def relu(self, x):
    return x * (x > 0)

  def relu_derivative(self, x):
    return 1 * (x > 0)

  def cross_entropy_loss(self, probs, y):
    num_of_examples = y.shape[0]
    log_likelihood = -np.log(probs[range(num_of_examples),y])
    loss = np.sum(log_likelihood) / num_of_examples
    return loss

  def cross_entropy_derivative(self, probs, y):
    num_of_examples = y.shape[0]
    probs[range(num_of_examples),y] -= 1
    return probs

  def train(self, X_train, y_train, learning_rate=0.01, epochs=1000, verbose=0):

    # Vanilla Gradient Descent Update
    for i in range(epochs):

        # Forward Propagation
        probs = self.forward(X_train)
        loss = self.cross_entropy_loss(probs,y_train)

        # Backward Propagation
        self.backprop(X_train, y_train)

        # Add regularization terms (b1 and b2 don't have regularization terms)
        self.dW_o += 0.1 * self.W_o
        self.dW_h += 0.1 * self.W_h

        # Gradient Descent Parameter Updates
        self.W_o += -learning_rate * self.dW_o
        self.b_o += -learning_rate * self.db_o
        self.W_h += -learning_rate * self.dW_h
        self.b_h += -learning_rate * self.db_h

        # Print loss
        if verbose==0 and i % 1000 == 0:
          print("Loss after epoch {} {}".format(i, loss))
        self.history.append(loss)
```

Data Preparation

```
# Download the datasets
import pandas as pd
import numpy as np
uris = [
    'https://archive.ics.uci.edu/ml/machine-learning-databases/dermatology/dermatology.
    'https://archive.ics.uci.edu/ml/machine-learning-databases/pendigits/pendigits.tra
dermatology_dataset, pendigit_dataset = [pd.read_csv(uri, header=None) for uri in uris]
```

```
#Dermatology Dataset Exploration
print('Dermatology Dataset Shape : {}'.format(dermatology_dataset.shape))
print(dermatology_dataset.head(5))

# Remove rows with missing values
dermatology_dataset.iloc[:,33] = pd.to_numeric(dermatology_dataset.iloc[:,33], errors=
dermatology_dataset = dermatology_dataset.dropna()
print('Dermatology Dataset Shape After Cleanup : {}'.format(dermatology_dataset.shape))

# Filter out the data for 3 classes from the dataset
dm_class_labels = {1:'psoriasis',2:'seboreic dermatitis', 3:'lichen planus'}
dermatology_dataset = dermatology_dataset.loc[dermatology_dataset.iloc[:,34].isin(dm_cl

X_dm, y_dm = dermatology_dataset.iloc[:,0:dermatology_dataset.shape[1]-1], dermatology_
X_dm, y_dm = X_dm.as_matrix() , y_dm.as_matrix()

# Normalise the features for faster convergence
#X_dm = (X_dm - np.mean(X_dm, axis=0)) / np.std(X_dm, axis=0)
# Convert labels to 0,1,2 for easier processing
y_dm -= 1

print('Features Shape: {}'.format(X_dm.shape))
print('Labels Shape: {}'.format(y_dm.shape))
```

```
Dermatology Dataset Shape : (366, 35)
   0  1  2  3  4  5  6  7  8  9 ... 25 26 27 28 29 30 31 3
0  2  2  0  3  0  0  0  0  1  0 ...  0  0  3  0  0  0  1
1  3  3  3  2  1  0  0  0  1  1 ...  0  0  0  0  0  0  1
2  2  1  2  3  1  3  0  3  0  0 ...  0  2  3  2  0  0  2
3  2  2  2  0  0  0  0  0  3  2 ...  3  0  0  0  0  0  3
4  2  3  2  2  2  2  0  2  0  0 ...  2  3  2  3  0  0  2

   33 34
0  55  2
1   8  1
2  26  3
3  40  1
4  45  3

[5 rows x 35 columns]
Dermatology Dataset Shape After Cleanup : (358, 35)
Features Shape: (242, 34)
Labels Shape: (242,)
```

```
#Pen Digits Dataset Exploration
print('Pen Digit Dataset Shape : {}'.format(pendigit_dataset.shape))
print(pendigit_dataset.head(5))

# Filter out the data for 4 digits from the dataset
```

```
pendigit_dataset = pendigit_dataset.loc[pendigit_dataset.iloc[:,16].isin([0,1,2,3])]

X_pd, y_pd = pendigit_dataset.iloc[:,0:16], pendigit_dataset.iloc[:,16]
X_pd, y_pd = X_pd.as_matrix() , y_pd.as_matrix()

# Normalise the features for faster convergence
#X_pd = (X_pd - np.mean(X_pd, axis=0)) / np.std(X_pd, axis=0)

print('Features Shape: {}'.format(X_pd.shape))
print('Labels Shape: {}'.format(y_pd.shape))
```

```
Pen Digit Dataset Shape : (7494, 17)
    0    1   2    3    4    5    6    7   8   9   10  11   12  13   14  15
0  47  100  27   81   57   37   26    0   0  23   56  53  100  90   40  98
1   0   89  27  100   42   75   29   45  15  15   37   0   69   2  100   6
2   0   57  31   68   72   90  100  100  76  75   50  51   28  25   16   0
3   0  100   7   92    5   68   19   45  86  34  100  45   74  23   67   0
4   0   67  49   83  100  100   81   80  60  60   40  40   33  20   47   0
Features Shape: (3058, 16)
Labels Shape: (3058,)
```

```
def generate_k_folds(dataset, k):
  """
  Returns a list of folds, where each fold is a tuple like (training_set,
  test_set), where each set is a tuple like (examples, classes)
  """
  folds=[]
  n=dataset[0].shape[0]
  fold_size = n//k
  # Divide the data into k equal subsections, keep k-1 section for training
  # and 1 for testing, repeat k times to generate folds
  for i in range(k):
    indices = [j for j in range(n)]
    if i == k-1:
      fold_size = n - i*fold_size
    test_idx = indices[i*fold_size:i*fold_size+fold_size],
    training_idx = indices[0:i*fold_size] + indices[i*fold_size+fold_size:]

    examples=dataset[0]
    classes=dataset[1]
    training_set_examples=examples[training_idx,:]
    training_set_classes=np.array(classes)[training_idx]
    training_set=(training_set_examples,training_set_classes)
    test_set_examples=examples[test_idx,:]
    test_set_classes=np.array(classes)[test_idx]
    test_set=(test_set_examples,test_set_classes)
    fold =(training_set,test_set)
    folds.append(fold)
  return folds
```

```
def k_fold_cross_validation_accuracy(folds, epochs, learning_rate, n_hidden, h_activat
  """Trains the model and returns its k-fold cross validation accuracy for specified pa
  scores = []
  for i, fold in enumerate(folds):
    train, valid = fold
    X_valid, y_valid = valid
    X_train, y_train = train

    n_classes = len(set(y_train))
    model = FeedForwardNet(n_input=X_train.shape[1], \
                           n_hidden=n_hidden, \
                           n_output=n_classes, \
                           h_activation=h_activation )
    model.train(X_train,
                y_train,
                epochs=epochs,
                learning_rate=learning_rate,
                verbose=1)
```

```
    y_pred = model.predict(X_valid)
    accuracy = np.mean(y_pred == y_valid)
    scores.append(accuracy)

  k_fold_cross_validation_accuracy = 0
  if len(scores) > 0:
    k_fold_cross_validation_accuracy = sum(scores)/len(scores)
  return k_fold_cross_validation_accuracy
```

Experiments

Setup

1) No of hidden units: [1, 3, 5, 10, 50, 100]

2) Activations:

- **tanh**

  It squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered.

- **relu**

  It computes the function f(x)=max(0,x). In other words, the activation is simply thresholded at zero (see image above on the left). It has been found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

4) Epochs: 20000

  Instead of choosing threshold values as stopping criteria, I have chosen number of epochs as a stopping criteria. A fairly large number of epochs would be helpful in seeing overfitting patterns and would ensure we are not treating a local minima an arbitrary stopping criteria, hence leading to better generalisation.

5) Learning Rate: 0.001

Compute 5-folds cross validation accuray for pen digits datasets

```
ıidden_units = [1,3,5,10,50,100]
ıctivations = ['tanh', 'relu']
ɪpochs = 10000
.earning_rate = 0.0001
ɔd_folds = generate_k_folds([X_pd, y_pd], 2)
ɔd_results = []
ɔrint(['Number of hidden units', 'Hidden Layer Activation', '5-fold CV accuracy'])
ʻor h_activation in activations:
  for n_hidden in hidden_units:
    accuracy = k_fold_cross_validation_accuracy(pd_folds, epochs, learning_rate, n_hidde
    result = [n_hidden, h_activation, accuracy]
    pd_results.append(result)
    print(result)
```

```
['Number of hidden units', 'Hidden Layer Activation', '5-fold CV accuracy']
[1, 'tanh', 0.2511445389143231]
[3, 'tanh', 0.4941137998691955 7]
[5, 'tanh', 0.3826030085022890 7]
[10, 'tanh', 0.600719424460431 7]
[50, 'tanh', 0.877697841726618 7]
[100, 'tanh', 0.719097449313276 7]
[1, 'relu', 0.2553956834532374]
[3, 'relu', 0.2547416612164813 4]
[5, 'relu', 0.2547416612164813 4]
```

Compute 5-folds cross validation accuray for dermatology datasets

```
! Compute 5-folds cross validation accuray for dermatology datasets
lm_folds = generate_k_folds([X_dm, y_dm], 2)
ιidden_units = [1,3,5,10,50,100]
ιctivations = ['tanh','relu']
ːpochs = 20000
.earning_rate = 0.001

lm_results = []
ɔrint(['Number of hidden units', 'Hidden Layer Activation', '5-fold CV accuracy'])
ːor h_activation in activations:
  for n_hidden in hidden_units:
    accuracy = k_fold_cross_validation_accuracy(dm_folds, epochs, learning_rate, n_hidde
    result = [n_hidden, h_activation, accuracy]
    dm_results.append(result)
    print(result)
```

```
['Number of hidden units', 'Hidden Layer Activation', '5-fold CV accuracy']
[1, 'tanh', 0.5371900826446281]
[3, 'tanh', 0.6487603305785123]
[5, 'tanh', 0.8471074380165289]
[10, 'tanh', 0.9958677685950413]
[50, 'tanh', 0.9917355371900827]
[100, 'tanh', 0.9917355371900827]
[1, 'relu', 0.2272727272727272 7]
[3, 'relu', 0.2272727272727272 7]
[5, 'relu', 0.2272727272727272 7]
[10, 'relu', 0.4173553719008264 4]
[50, 'relu', 0.4586776859504132 5]
[100, 'relu', 0.4586776859504132 5]
```

Observations

- As we can observe from accuracy tables of both trained datasets, increasing the number of hidden units results in more complex models. When the dataset itself is complex i.e have a large number of distinct features, higher dimensional hidden units help in modeling more complex behaviours and result in better accuracy. On the other hand, in simpler datasets, higher dimensional hidden units are prone to overfitting . It lead to memorisation of the training set which performs poorly on the test set.
- For both the datasets, tanh activation function performs better than the relu activation with the given learning rates. It appears a large gradient flowing through the ReLU neuron is causing the weights to update in such a way that the neuron is never activating from that datapoint again. Due to this, the gradient flowing through the unit will is forever be zero from that point on, resulting in consistent loss for large learning rates. Tuning the learning rates and setting it to a lesser values seems to be resolving the issue.
- Training is much faster for relu than tanh activation function