```
%matplotlib inline
```

Training a Classifier

This is it. You have seen how to define neural networks, compute loss and make updates to the weigh

Now you might be thinking,

What about data?

Generally, when you have to deal with image, text, audio or video data, you can use standard python pa
Then you can convert this array into a `torch.*Tensor`.

- For images, packages such as Pillow, OpenCV are useful
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful

Specifically for vision, we have created a package called `torchvision`, that has data loaders for comm
MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.Da`

This provides a huge convenience and avoids writing boilerplate code.

For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat',
The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.

.. figure:: /_static/img/cifar10.png :alt: cifar10

cifar10

Training an image classifier

We will do the following steps in order:

1. Load and normalizing the CIFAR10 training and test datasets using `torchvision`
2. Define a Convolution Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data
6. Loading and normalizing CIFAR10 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Using `torchvision`, it's extremely easy to load CIFAR10.

```
import torch
import torchvision
```

```
import torchvision.transforms as transforms
```

The output of torchvision datasets are PILImage images of range [0, 1]. We transform them to Tensor

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-p
```

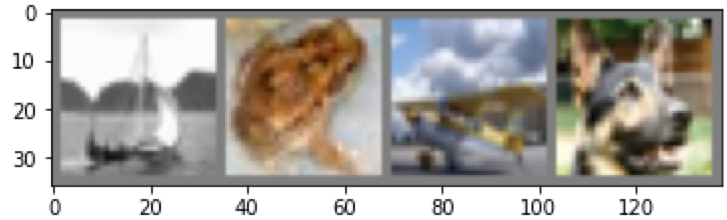
170500096/? [00:20<00:00, 68205847.04it/s]

```
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Let us show some of the training images, for fun.

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
ship  frog plane   dog
```



2. Define a Convolution Neural Network ^^ Copy the neural netw
   before and modify it to take 3-channel images (instead of 1-channel images as it was defined).

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

3. Define a Loss function and optimizer ^^ Let's use a Classifica
   momentum.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

4. Train the network ^^^^^^^^^^^^^^^^^^^^

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.153
[1,  4000] loss: 1.812
[1,  6000] loss: 1.656
[1,  8000] loss: 1.558
[1, 10000] loss: 1.507
[1, 12000] loss: 1.459
[2,  2000] loss: 1.404
[2,  4000] loss: 1.349
[2,  6000] loss: 1.324
[2,  8000] loss: 1.320
[2, 10000] loss: 1.300
[2, 12000] loss: 1.258
Finished Training
```

5. Test the network on the test data ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

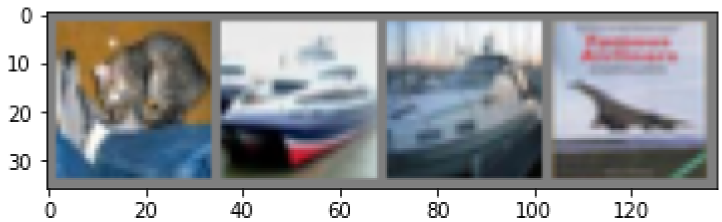
We have trained the network for 2 passes over the training dataset. But we need to check if the netwo

We will check this by predicting the class label that the neural network outputs, and checking it agains correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
GroundTruth:    cat  ship  ship plane
```



Okay, now let us see what the neural network thinks these examples above are:

```
outputs = net(images)
```

The outputs are energies for the 10 classes. Higher the energy for a class, the more the network think
So, let's get the index of the highest energy:

```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:    cat   car  ship plane
```

The results seem pretty good.

Let us look at how the network performs on the whole dataset.

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 56 %
```

That looks waaay better than chance, which is 10% accuracy (randomly picking a class out of 10 clas something.

Hmmm, what are the classes that performed well, and the classes that did not perform well:

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 63 %
Accuracy of   car : 72 %
Accuracy of  bird : 47 %
Accuracy of   cat : 36 %
Accuracy of  deer : 35 %
Accuracy of   dog : 36 %
Accuracy of  frog : 78 %
Accuracy of horse : 54 %
Accuracy of  ship : 70 %
Accuracy of truck : 65 %
```

Okay, so what next?

How do we run these neural networks on the GPU?

Training on GPU

Just like how you transfer a Tensor on to the GPU, you transfer the neural net onto the GPU.

Let's first define our device as the first visible cuda device if we have CUDA available:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assume that we are on a CUDA machine, then this should print a CUDA device:

print(device)
```

```
cpu
```

The rest of this section assumes that `device` is a CUDA device.

Then these methods will recursively go over all modules and convert their parameters and buffers to

.. code:: python

```
net.to(device)
```

Remember that you will have to send the inputs and targets at every step to the GPU too:

.. code:: python

```
    inputs, labels = inputs.to(device), labels.to(device)
```

Why dont I notice MASSIVE speedup compared to CPU? Because your network is realllly small.

Exercise: Try increasing the width of your network (argument 2 of the first `nn.Conv2d`, and argument be the same number), see what kind of speedup you get.

Goals achieved:

- Understanding PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

Training on multiple GPUs

If you want to see even more MASSIVE speedup using all of your GPUs, please check out :doc: `data_p`

Where do I go next?

- :doc: `Train neural nets to play video games </intermediate/reinforcement_q_learning>`
- `Train a state-of-the-art ResNet network on imagenet`_
- `Train a face generator using Generative Adversarial Networks`_
- `Train a word-level language model using Recurrent LSTM networks`_
- `More examples`_
- `More tutorials`_
- `Discuss PyTorch on the Forums`_
- `Chat with other users on Slack`_

Neural Network with Backpropagation In Python (from scratch)

[Reference]

The backpropagation algorithm is used in the classical feed-forward artificial neural network.

It is the technique still used to train large deep learning networks.

Below we implement the backpropagation algorithm for a neural network from scratch with Python.

After completing this tutorial, you will know:

1. Forward-propagate an input to calculate an output.
2. Back-propagate error and train a network.
3. Apply the backpropagation algorithm to a predictive modeling problem.

Description

This section provides a brief introduction to the Backpropagation Algorithm and the Wheat Seeds dat
code.

Backpropagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feed-forward networks
Networks.

Feed-forward neural networks are inspired by the information processing of one or more neural cells,
signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the s
connections of a cell's axon to other cell's dendrites.

The principle of the backpropagation approach is to model a given function by modifying internal wei
expected output signal. The system is trained using a supervised learning method, where the error be
expected output is presented to the system and used to modify its internal state.

Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-fo
a network structure to be defined of one or more layers where one layer is fully connected to the next
input layer, one hidden layer, and one output layer.

Backpropagation can be used for both classification and regression problems, but we will focus on cl

In classification problems, best results are achieved when the network has one neuron in the output la
class or binary classification problem with the class values of A and B. These expected outputs woul
vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is

Wheat Seeds Dataset

The seeds dataset involves the prediction of species given measurements seeds from different variet

There are 201 records and 7 numerical input variables. It is a classification problem with 3 output clas
value vary, so some data normalization may be required for use with algorithms that weight inputs lik

Below is a sample of the first 5 rows of the dataset.

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

Using the Zero Rule algorithm that predicts the most common class value, the baseline accuracy for t

You can learn more and download the seeds dataset from the [UCI Machine Learning Repository](#).

We downloaded the seeds dataset and place it into our current working directory with the filename se

The dataset is in tab-separated format, so we converted it to CSV using a text editor.

Step by step

The code is broken down into 6 parts:

1. Initialize Network.
2. Forward Propagate.
3. Back Propagate Error.
4. Train Network.
5. Predict.
6. Seeds Dataset Case Study.

These steps provide the foundation needed to implement the backpropagation algorithm from scratc problem.

1. Initialize Network

First, the creation of a new network ready for training.

Each neuron has a set of weights that need to be maintained. One weight for each input connection a will need to store additional properties for a neuron during training, therefore we will use a dictionary t properties by names such as 'weights' for the weights.

A network is organized into layers. The input layer is really just a row from our training dataset. The fir followed by the output layer that has one neuron for each class value.

We will organize layers as arrays of dictionaries and treat the whole network as an array of layers.

It is good practice to initialize the network weights to small random numbers. In this case, will we use

Below is a function named initialize_network() that creates a new neural network ready for training. It inputs, the number of neurons to have in the hidden layer and the number of outputs.

You can see that for the hidden layer we create n_hidden neurons and each neuron in the hidden layer input column in a dataset and an additional one for the bias.

You can also see that the output layer that connects to the hidden layer has n_outputs neurons, each that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

```
# 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_output
  network.append(output_layer)
  return network
```

Let's test out this function. Below is a complete example that creates a small network.

```
from random import seed
from random import random

# 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_output
  network.append(output_layer)
  return network

seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
  print(layer)
```

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381
```

Running the example, you can see that the code prints out each layer one by one. You can see the hid weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381, 0.6515929727
```

Now that we know how to create and initialize a network, we can use it to calculate an output.

2. Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer u

We call this forward-propagation.

It is the technique we will need to generate predictions during training that will need to be corrected, a network is trained to make predictions on new data.

We can break forward propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward Propagation.

2.1. Neuron Activation

The first step is to calculate the activation of one neuron given an input.

The input could be a row from our training dataset, as in the case of the hidden layer. It may also be th layer, in the case of the output layer.

Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.

```
activation = sum(weight_i * input_i) + bias
```

Where **weight** is a network weight, **input** is an input, **i** is the index of a weight or an input and **bias** is a multiply with (or you can think of the input as always being 1.0).

Below is an implementation of this in a function named activate(). We can see that the function assur list of weights. This helps here and later to make the code easier to read.

```
# 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
```

Next, we show how to use the neuron activation.

2.2. Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually i

Different transfer functions can be used. It is traditional to use the sigmoid activation function, but yo tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular w

The sigmoid activation function looks like an S shape, it's also called the logistic function. It can take between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (sl backpropagating error.

We can transfer an activation function using the sigmoid function as follows:

```
output = 1 / (1 + e^(-activation))
```

Where **e** is the base of the natural logarithms ([Euler's number]).

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

Now that we have the pieces, let's use them.

2.3. Forward Propagation

Forward propagating an input is straightforward.

We work through each layer of our network calculating the outputs for each neuron. All of the outputs neurons on the next layer.

Below is a function named forward_propagate() that implements the forward propagation for a row o network.

You can see that a neuron's output value is stored in the neuron with the name 'output'. You can also s in an array named new_inputs that becomes the array inputs and is used as inputs for the following la

The function returns the outputs from the last layer also called the output layer.

```
# 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
```

Let's put all of these pieces together and test out the forward propagation of our network.

We define our network inline with one hidden neuron that expects 2 input values and an output layer v

```
from math import exp

# 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

# test forward propagation
network = [[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
    [{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381,
row = [1, 0, None]
output = forward_propagate(network, row)
print(output)
```

```
[0.6629970129852887, 0.7253160725279748]
```

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. neurons, we get a list of two numbers as output.

The actual output values are just nonsense for now, but next, we will start to learn how to make the w

```
[0.6629970129852887, 0.7253160725279748]
```

3. Back Propagate Error

The backpropagation algorithm is named for the way in which weights are trained.

Error is calculated between the expected outputs and the outputs forward propagated from the netwo backward through the network from the output layer to the hidden layer, assigning blame for the error

The math for backpropagating error is rooted in calculus, but we will remain high level in this section a rather than why the calculations take this particular form.

This part is broken down into two sections.

1. Transfer Derivative.
2. Error Backpropagation.

3.1. Transfer Derivative

Given an output value from a neuron, we need to calculate it's slope.

We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

```
derivative = output * (1.0 - output)
```

Below is a function named transfer_derivative() that implements this equation.

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

Now, let’s see how this can be used.

3.2. Error Backpropagation

The first step is to calculate the error for each output neuron, this will give us our error signal (input) t network.

The error for a given neuron can be calculated as follows:

```
error = (expected - output) * transfer_derivative(output)
```

Where **expected** is the expected output value for the neuron, **output** is the output value for the neuron slope of the neuron's output value, as shown above.

This error calculation is used for neurons in the output layer. The expected value is the class value its more complicated.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in t traveling back along the weights of the output layer to the neurons in the hidden layer.

The back-propagated error signal is accumulated and then used to determine the error for the neuron

```
error = (weight_k * error_j) * transfer_derivative(output)
```

Where **error_j** is the error signal from the **j**th neuron in the output layer, **weight_k** is the weight that co
neuron and output is the output for the current neuron.

Below is a function named **backward_propagate_error()** that implements this procedure.

You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can s
iterated in reverse order, starting at the output and working backwards. This ensures that the neurons
calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name
implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the o

```
# 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'])
```

Let's put all of the pieces together and see how it works.

We define a fixed neural network with output values and backpropagate an expected output pattern. T

```
# 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'])

# test backpropagation of error
network = [[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.847433736937232
    [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095]}, {'
expected = [0, 1]
backward_propagate_error(network, expected)
for layer in network:
  print(layer)
```

```
[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.7
[{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095], 'd
```

Running the example prints the network after the backpropagation of error is complete. You can see t
in the neurons for the output layer and the hidden layer.

```
[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614
[{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095], 'delta': -0.1461
```

Now let's use the backpropagation of error to train the network.

4. Train Network

The network is trained using stochastic gradient descent.

This involves multiple iterations of exposing a training dataset to the network and for each row of dat
backpropagating the error and updating the network weights.

This part is broken down into two sections:

1. Update Weights.
2. Train Network.

4.1. Update Weights

Once errors are calculated for each neuron in the network via the back propagation method above, the

Network weights are updated as follows:

```
weight = weight + learning_rate * error * input
```

Where **weight** is a given weight, **learning_rate** is a parameter that you must specify, **error** is the error c procedure for the neuron and **input** is the input value that caused the error.

The same procedure can be used for updating the bias weight, except there is no input term, or input i

Learning rate controls how much to change the weight to correct for the error. For example, a value of amount that it possibly could be updated. Small learning rates are preferred that cause slower learnin iterations. This increases the likelihood of the network finding a good set of weights across all layers minimize error (called premature convergence).

Below is a function named **update_weights()** that updates the weights for a network given an input ro a forward and backward propagation have already been performed.

Note: Remember that the input for the output layer is a collection of outputs from the hidden layer.

```
# 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']
```

Now we know how to update network weights, let's see how we can do it repeatedly.

4.2. Train Network

The network is updated using stochastic gradient descent.

This involves first looping for a fixed number of epochs and within each epoch updating the network f

Because updates are made for each training pattern, this type of learning is called online learning. If e before updating the weights, this is called batch learning or batch gradient descent.

Below is a function that implements the training of an already initialized neural network with a given t of epochs and an expected number of output values.

The expected number of output values is used to transform class values in the training data into a on with one column for each class value to match the output of the network. This is required to calculate

You can also see that the sum squared error between the expected output and the network output is a
is helpful to create a trace of how much the network is learning and improving each epoch.

```
# 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):
    sum_error = 0
    for row in train:
      outputs = forward_propagate(network, row)
      expected = [0 for i in range(n_outputs)]
      expected[row[-1]] = 1
      sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
      backward_propagate_error(network, expected)
      update_weights(network, row, l_rate)
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
```

We now have all of the pieces to train the network. We can put together an example that includes ever
network initialization and train a network on a small dataset.

Below is a small contrived dataset that we can use to test out training our neural network.

```
X1                  X2                  Y
2.7810836           2.550537003         0
1.465489372         2.362125076         0
3.396561688         4.400293529         0
1.38807019          1.850220317         0
3.06407232          3.005305973         0
7.627531214         2.759262235         1
5.332441248         2.088626775         1
6.922596716         1.77106367          1
8.675418651         -0.242068655        1
7.673756466         3.508563011         1
```

Below is the complete example. We will use 2 neurons in the hidden layer. It is binary classification pr
neurons in the output layer. The network will be trained for 20 epochs with a learning rate of 0.5, whicl
few iterations.

```
from math import exp
from random import seed
from random import random

# 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_output
    network.append(output_layer)
    return network

  # 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):
```

```
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):
    sum_error = 0
    for row in train:
      outputs = forward_propagate(network, row)
      expected = [0 for i in range(n_outputs)]
      expected[row[-1]] = 1
      sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
      backward_propagate_error(network, expected)
      update_weights(network, row, l_rate)
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
  print(layer)
```

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.0
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.
```

Running the example first prints the sum squared error each training epoch. We can see a trend of thi

Once trained, the network is printed, showing the learned weights. Also still in the network are output could update our training function to delete these data if we wanted.


```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
```

```
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357
```

Once a network is trained, we need to use it to make predictions.

5. Predict

Making predictions with a trained neural network is easy enough.

We have already seen how to forward-propagate an input pattern to get an output. This is all we need the output values themselves directly as the probability of a pattern belonging to each output class.

It may be more useful to turn this output back into a crisp class prediction. We can do this by selectin probability. This is also called the [arg max function](#).

Below is a function named **predict()** that implements this procedure. It returns the index in the networ assumes that class values have been converted to integers starting at 0.

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

We can put this together with our code above for forward propagating input and with our small contri with an already-trained network. The example hardcodes a network trained from the previous step.

The complete example is listed below.

```
from math import exp

# 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

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

# Test making predictions with the network
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799]}, {'weigh
  [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]}, {'weights': [-
for row in dataset:
  prediction = predict(network, row)
  print('Expected=%d, Got=%d' % (row[-1], prediction))
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Running the example prints the expected output for each record in the training dataset, followed by th

It shows that the network achieves 100% accuracy on this small dataset.

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
```

```
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Now we are ready to apply our backpropagation algorithm to a real world dataset.

6. Wheat Seeds Dataset

This section applies the Backpropagation algorithm to the wheat seeds dataset.

The first step is to load the dataset and convert the loaded data to numbers that we can use in our ne helper function **load_csv()** to load the file, **str_column_to_float()** to convert string numbers to floats ar column to integer values.

Input values vary in scale and need to be normalized to the range of 0 and 1. It is generally good pract of the chosen transfer function, in this case, the sigmoid function that outputs values between 0 and ' **normalize_dataset()** helper functions were used to normalize the input values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that 201/5=40.2 use the helper functions **evaluate_algorithm()** to evaluate the algorithm with cross-validation and **acc** of predictions.

A new function named **back_propagation()** was developed to manage the application of the Backprop network, training it on the training dataset and then using the trained network to make predictions on

The complete example is listed below.

[3 Ways to Load CSV files into Colab]()

To upload from your local drive, start with the following code:

```
from google.colab import files
uploaded = files.upload()
```

It will prompt you to select a file. Click on "Choose Files" then select and upload the file. Wait for the fi the name of the file once Colab has uploaded it.

```
from google.colab import files
uploaded = files.upload()
```

Choose Files seeds_dataset.csv

```
%ls sample_data/
```

```
anscombe.json*                 mnist_test.csv
california_housing_test.csv    mnist_train_small.csv
california_housing_train.csv   README.md*
```

```
# 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])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# 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, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# 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:
```

```
    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_output
  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)
    predictions.append(prediction)
  return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
'''
First load the dataset using the code above

from google.colab import files
uploaded = files.upload()
'''
filename = 'seeds_dataset.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
```

```
n_folds = 5
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
Scores: [90.47619047619048, 92.85714285714286, 97.61904761904762, 92.85714285714286, 92.
Mean Accuracy: 93.333%
```

A network with 5 neurons in the hidden layer and 3 neurons in the output layer was constructed. The r
learning rate of 0.3. These parameters were found with a little trial and error, but you may be able to d

Running the example prints the average classification accuracy on each fold as well as the average pe

You can see that backpropagation and the chosen configuration achieved a mean classification accu
better than the Zero Rule algorithm that did slightly better than 28% accuracy.

```
Scores: [90.47619047619048, 92.85714285714286, 97.61904761904762, 92.85714285714286, 92.85714285714286
Mean Accuracy: 93.333%
```

Extensions

Next some extensions to the code above that we can explore.

- **Tune Algorithm Parameters**. Try larger or smaller networks trained for longer or shorter. See if w
  dataset.
- **Additional Methods**. Experiment with different weight initialization techniques (such as small ra
  functions (such as tanh and relu).
- **More Layers**. Add support for more hidden layers, trained in just the same way as the one hidde
  Change the network so that there is only one neuron in the output layer and that a real value is p
  practice on. A linear transfer function could be used for neurons in the output layer, or the outpu
  scaled to values between 0 and 1.
- **Batch Gradient Descent**. Change the training procedure from online to batch gradient descent a
  each epoch.

Conclusion

In the code above, we discovered how to implement the Multilayer perceptron algorithm from scratch

Specifically, we learned:

- How to forward propagate an input to calculate a network output.
- How to back propagate error and update network weights.

- How to apply the backpropagation algorithm to a real world dataset.