

Homework 2: Backpropagation

Submission Instructions: Before the deadline, export the completed notebook to PDF and upload it to GradeScope. The PDF should clearly show your code, and the result of running the code. Check the PDF to ensure that it is readable, the font-size is not small, and no information is cut-off. There will be no make-ups or extensions for corrupted/damaged/unreadable PDFs.

Names of Collaborators:

In this problem set, you will experiment with neural networks for image recognition. We will start with a toy neural network, where you will build up the pieces to implement backpropagation. Then, we will switch to industrial strength neural networks that are already trained, and use them for image recognition.

In [27]:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from IPython import display
```

Problem 1: Neural Network Layers

We are going to implement a simple neural network that uses fully connected layers and the ReLU activation function. In this problem, we first implement the individual layers, then later on you will chain them together to create a neural network.

The ReLU layer should follow the implementation discussed in lecture. The fully connected layer is just a linear layer without a bias term. It is called a fully connected layer because every input "neuron" is connected to every output "neuron" through a weighted summation.

The input to the forward pass of each layer will always be an $N \times B$ matrix, where N is the number of dimensions for an input vector, and B is the number of vectors in the batch. Since neural networks need to perform gradient descent over many millions of iterations, we need to make each step very efficient. On most hardware, it is much faster to batch operations together. We will batch the forward pass to operate on B vectors of dimension N at once, and we will do a similar batching for the backwards pass too.

Implement the following methods:

- `forwards()` for ReLU
- `backwards()` for ReLU
- `forwards()` for FullyConnectedLayer
- `backwards()` for FullyConnectedLayer
- `backwards_param()` for FullyConnectedLayer
- `update_param()` for FullyConnectedLayer

In [28]:

```

class ReLU(object):
    def __repr__(self):
        return f"Relu"

    def forwards(self, x):
        return np.where(x > 0, x, 0)

    def backwards(self, x, grad_output):
        assert x.shape == grad_output.shape

        # Return the gradient with respect to the input.
        return np.where(x > 0, grad_output, 0)

    def backwards_param(self, x, grad_output):
        return None # We return none because there are no learnable parameters.

class FullyConnectedLayer(object):
    def __init__(self, in_dim, out_dim):
        self.out_dim = out_dim # During forward pass, the resulting output vector will have out_dim dimensions
        self.in_dim = (
            in_dim # During forward pass, each input vector will have in_dim dimensions
        )

        # Create an initial guess for the parameters w by sampling from a Gaussian
        # distribution.
        self.w = np.random.normal(
            0, math.sqrt(2 / (in_dim + out_dim)), [out_dim, in_dim]
        )

    def __repr__(self):
        return f"FullyConnectedLayer({self.in_dim} , {self.out_dim})"

    def forwards(self, x):
        # Computes the forward pass of a linear layer (which is also called
        # fully connected). The input x will be a matrix that has the shape:
        # (in_dim)x(batch_size). The output should be a matrix that has the
        # shape (out_dim)x(batch_size). Note: in this implementation, there is
        # no bias term in order to keep it simple.
        assert x.shape[0] == self.in_dim
        # Return the result of the forwards pass.
        return self.w @ x

    def backwards(self, x, grad_output):
        assert grad_output.shape[0] == self.out_dim
        # Return the gradient with respect to the input.
        return self.w.T @ grad_output

    def backwards_param(self, x, grad_output):
        assert grad_output.shape[0] == self.out_dim
        # Return the gradient with respect to the parameters.
        return grad_output @ x.T

    def update_param(self, grad):
        # Given the gradient with respect to the parameters, perform a gradient step.
        # This function should modify self.w based on grad. You should implement
        # the basic version of gradient descent. The function does not return anything.
        self.w -= 0.1 * grad

```

The below code can be useful to test the forward pass of these functions. Feel free to design your own test cases too!

In [29]:

```

def test_equality(name, actual, expected):
    result = (np.abs(actual-expected) < 1e-7).all()
    if result:
        print("OK\t", name)
    else:
        print("FAIL\t", name)
        print("Actual:")
        print(actual)
        print("Expected:")
        print(expected)

test_input = np.array([[10., -5., 3., 0., 2., -1.]])
expected_output = np.array([[10., 0., 3., 0., 2., 0.]])
actual_output = ReLU().forwards(test_input)
test_equality("ReLU Forward 1", actual_output, expected_output)

test_input = np.array([[10., -5., 3., 0., 2., -1.], [3., 2., 1., 0., -1., -2.]])
expected_output = np.array([[10., 0., 3., 0., 2., 0.], [3., 2., 1., 0., 0., 0.]])
actual_output = ReLU().forwards(test_input)
test_equality("ReLU Forward Batch", actual_output, expected_output)

layer = FullyConnectedLayer(6,2)
layer.w[0, :] = 1
layer.w[1, :] = 0
test_input = np.array([[10, -5, 3, 0, 2, -1]]).T
expected_output = np.array([[test_input.sum(), 0]]).T
actual_output = layer.forwards(test_input)
test_equality("Fully Connected Forward 1", actual_output, expected_output)

layer = FullyConnectedLayer(6,2)
layer.w[0, :] = -1
layer.w[1, :] = 2
test_input = np.array([[10, -5, 3, 0, 2, -1]]).T
expected_output = np.array([[ -test_input.sum(), 2*test_input.sum()]]).T
actual_output = layer.forwards(test_input)
test_equality("Fully Connected Forward 2", actual_output, expected_output)

layer = FullyConnectedLayer(3,2)
layer.w[0, :] = 1
layer.w[1, :] = .5
test_input = np.array([[1,2,3],[-4,-5,-6]]).T
expected_output = np.array([[test_input[:, 0].sum(), 0.5*test_input[:, 0].sum()],
                             [test_input[:, 1].sum(), 0.5*test_input[:, 1].sum()]]).T
actual_output = layer.forwards(test_input)
test_equality("Fully Connected Forward Batch", actual_output, expected_output)

print('Done.')

```

```

OK ReLU Forward 1
OK ReLU Forward Batch
OK Fully Connected Forward 1
OK Fully Connected Forward 2
OK Fully Connected Forward Batch
Done.

```

We can also check whether the gradient of the output is implemented correctly. The `test_gradient_output` function will approximate the gradient with finite differencing, and compare it against the analytical gradient. If the gradients are calculated correctly, the two should be nearly equal.

In [30]:

```

def test_gradient_output(name, layer, x, epsilon=1e-7):
    grad_approx = np.zeros(x.shape)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i][j] += epsilon
            fxph = layer.forwards(x)
            x[i][j] -= 2 * epsilon
            fxmh = layer.forwards(x)
            x[i][j] += epsilon
            grad_approx[i][j] = (fxph - fxmh).sum() / (2 * epsilon)
    grad_backprop = layer.backwards(x, np.ones(layer.forwards(x).shape))

    numerator = np.linalg.norm(grad_backprop - grad_approx)
    denominator = np.linalg.norm(grad_backprop) + np.linalg.norm(grad_approx)
    difference = numerator / (denominator)
    if difference < 1e-7:
        print("OK\t", name)
    else:
        print("FAIL\t", name, "Difference={}".format(difference))

layer = ReLU()
test_input = np.array([[3., -4, 2]]).T
test_gradient_output("ReLU Output Gradient", layer, test_input)

layer = ReLU()
test_input = np.array([[6, -1, -3], [4, 0.1, 2]]).T
test_gradient_output("ReLU Output Gradient Batch", layer, test_input)

layer = FullyConnectedLayer(1,1)
layer.w[:] = 2.
test_input = np.array([[3.]]).T
test_gradient_output("Fully Connected Output Gradient 1", layer, test_input)

layer = FullyConnectedLayer(6,2)
test_input = np.array([[-1.,2.,-3.,4.,5.,6.]]).T
test_gradient_output("Fully Connected Output Gradient 2", layer, test_input)

layer = FullyConnectedLayer(100,100)
test_input = np.random.randn(100,1)
test_gradient_output("Fully Connected Output Gradient 3", layer, test_input)

layer = FullyConnectedLayer(100,100)
test_input = np.random.randn(100,50)
test_gradient_output("Fully Connected Output Gradient Batch", layer, test_input)

print('Done.')
OK ReLU Output Gradient
OK ReLU Output Gradient Batch
OK Fully Connected Output Gradient 1
OK Fully Connected Output Gradient 2
OK Fully Connected Output Gradient 3
OK Fully Connected Output Gradient Batch
Done.
The below will check that the gradient of the parameters is correct:

In [31]:

```

```

def test_gradient_param(name, layer, x, epsilon=1e-7):
    grad_approx = np.zeros(layer.w.shape)
    for i in range(layer.w.shape[0]):
        for j in range(layer.w.shape[1]):
            layer.w[i][j] += epsilon
            fxph = layer.forwards(x)
            layer.w[i][j] -= 2 * epsilon
            fxmh = layer.forwards(x)
            layer.w[i][j] += epsilon
            grad_approx[i][j] = (fxph - fxmh).sum() / (2 * epsilon)
    grad_backprop = layer.backwards_param(x, np.ones(layer.forwards(x).shape))
    numerator = np.linalg.norm(grad_backprop - grad_approx)
    denominator = np.linalg.norm(grad_backprop) + np.linalg.norm(grad_approx)
    difference = numerator / (denominator + epsilon)
    if difference < 1e-7:
        print("OK\t", name)
    else:
        print("FAIL\t", name, "Difference={}".format(difference))

layer = FullyConnectedLayer(1,1)
test_input = np.array([[1]]).T
test_gradient_param("Fully Connected Params Gradient 1", layer, test_input)

layer = FullyConnectedLayer(6,2)
test_input = np.array([[-1,2,-3,4,5,6]]).T
test_gradient_param("Fully Connected Params Gradient 2", layer, test_input)

layer = FullyConnectedLayer(100,100)
test_input = np.random.randn(100,1)
test_gradient_param("Fully Connected Params Gradient 3", layer, test_input)

layer = FullyConnectedLayer(100,100)
test_input = np.random.randn(100,50)
test_gradient_param("Fully Connected Params Gradient Batch", layer, test_input)

print('Done.')
OK Fully Connected Params Gradient 1
OK Fully Connected Params Gradient 2
OK Fully Connected Params Gradient 3
OK Fully Connected Params Gradient Batch
Done.

```

Problem 2: Loss Function

In order to train the parameters of the neural network, we need to implement a loss function that compares the prediction to the target. Implement the **squared error loss function**: $\text{Loss}(x,y) = \frac{1}{n} \|x - y\|_2^2$ where n is the batch size. The function should return both the loss and the gradient. It is standard practice to also divide the loss function by the batch size, which normalizes the loss value against the batch size.

In [32]:

```

def euclidean_loss(prediction, target):
    assert prediction.shape == target.shape
    # TODO: Implement a function that computes:
    # - the loss (scalar)
    # - the gradient of the loss with respect to the prediction (tensor)
    # The function should return these two values as a tuple.
    loss = np.sum((target - prediction) ** 2) / target.shape[1]
    grad = (prediction - target) / target.shape[0]
    return loss, grad

```

Let's test that the loss is implemented correctly and the gradients are accurate:

In [33]:

```
def test_gradient_loss(name, x, target, epsilon=1e-7):
    grad_approx = np.zeros(x.shape)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i][j] += epsilon
            fxph, _ = euclidean_loss(x, target)
            x[i][j] -= 2 * epsilon
            fxmh, _ = euclidean_loss(x, target)
            x[i][j] += epsilon
            grad_approx[i][j] = (fxph - fxmh).sum() / (2 * epsilon)
    _, grad_exact = euclidean_loss(x, target)
    numerator = np.linalg.norm(grad_exact - grad_approx)
    denominator = np.linalg.norm(grad_exact) + np.linalg.norm(grad_approx)
    difference = numerator / (denominator + epsilon)
    if difference < 1e-7:
        print("OK\t", name)
    else:
        print("FAIL\t", name, "Difference={}".format(difference))
```

```
test_a = np.array([[ -1.,  5, -2, 0],[4.3, -10, 7.8, 8.4]])
test_b = np.array([[ 1., -3, -2, 2],[8.4, 7.8, -10, 4.3]])
loss, _ = euclidean_loss(test_a, test_a)
test_equality("Euclidean Loss 1", loss, 0)
loss, _ = euclidean_loss(test_a, test_b)
test_equality("Euclidean Loss 2", loss, 739.3/4.)
```

```
test_gradient_loss("Euclidean Loss Gradient 1", test_a, test_a)
test_gradient_loss("Euclidean Loss Gradient 2", test_a, test_b)
```

```
print('Done.')
```

```
OK Euclidean Loss 1
OK Euclidean Loss 2
OK Euclidean Loss Gradient 1
OK Euclidean Loss Gradient 2
Done.
```

Problem 3: Neural Network

In this section, we will combine the layers, the gradients, and the loss function together to train a neural network. We will optimize the parameters of all the layers with gradient descent where the gradients are calculated with back-propagation.

We have implemented most of the class for you. Implement the rest of the **backwards()** method.

In [34]:

```

class NeuralNetwork(object):
    def __init__(self):
        self.layers: List[Union[ReLU, FullyConnectedLayer]] = []
        self.inputs = []
        self.grad_params = []

    def add(self, layer):
        self.layers.append(layer)

    def forwards(self, x):
        self.inputs = []
        out = x
        for layer in self.layers:
            self.inputs.append(out)
            out = layer.forwards(out)
        return out

    def backwards(self, grad_output):
        assert len(self.inputs) == len(self.layers)
        self.grad_params = [] # store gradients of the parameters for each layer

        n = len(self.layers) - 1
        for layerid in range(n, -1, -1):
            x = self.inputs[layerid]
            grad_params = self.layers[layerid].backwards_param(x, grad_output)
            grad_output = self.layers[layerid].backwards(x, grad_output)

            if grad_params is not None: # store grad_params for update_param() below
            self.grad_params.append((layerid, grad_params))
        return grad_output

    def update_param(self, step_size=0.001):
        for layerid, grad_param in self.grad_params:
            self.layers[layerid].update_param(grad_param * step_size)

```

We can use the below code block to train the neural network. The code below first creates a 2 layer network, which consists of two fully connected layers, and trains it for many thousands of iterations with gradient descent to minimize the loss function.

The function `sample_batch()` automatically creates training data for us. Given a two dimensional input $\mathbf{x} \in \mathbb{R}^2$, the `sample_batch()` specifies the target output to be $|x_0| - |x_1|$ where x_0 is the first dimension of \mathbf{x} , x_1 is the second dimension, and $||$ is the absolute value operation.

Experiment with a few different options. In our implementation, we are able to minimize the loss value to 0.0005. Turn in a PDF that contains the loss curve. For full credit, the loss curve should get near zero.

In [35]:

```

nn = NeuralNetwork()
nn.add(FullyConnectedLayer(2,50))
nn.add(ReLU())
nn.add(FullyConnectedLayer(50,1))

def sample_batch(batch_size):
    x = np.random.randn(2, batch_size)
    y = np.expand_dims(np.abs(x[0, :]) - np.abs(x[1, :]), axis=0)
    return x, y

loss_values = []

for iter in range(100001):
    input_data, target = sample_batch(100)

    output = nn.forwards(input_data)
    loss, grad_loss = euclidean_loss(output, target)
    nn.backwards(grad_loss)
    nn.update_param(step_size=0.0001)

    if iter % 10000 == 0:
        print(f"Iter {iter} Loss={loss}")

    loss_values.append(loss)

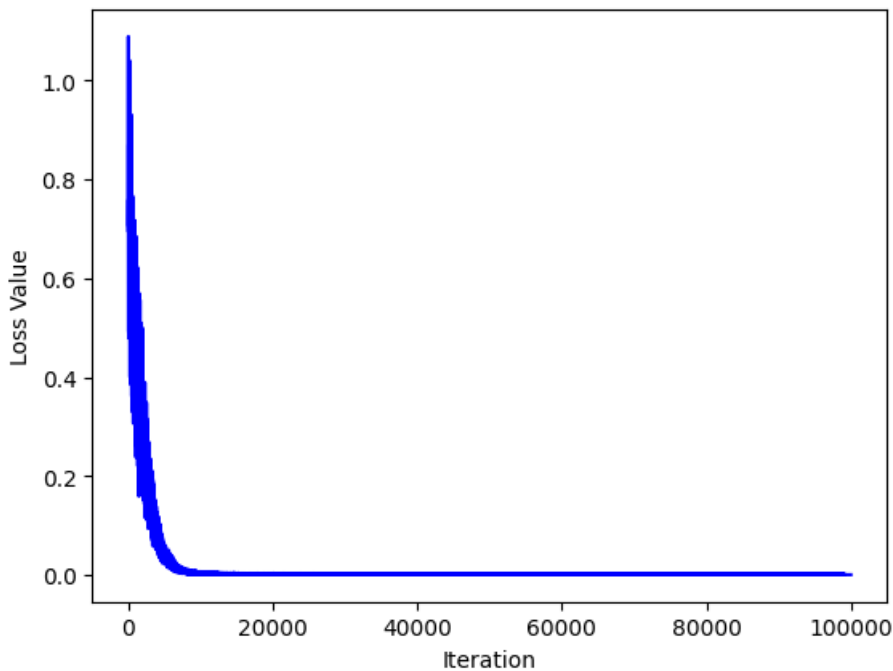
plt.clf()
plt.plot(loss_values, color='blue')
plt.ylabel('Loss Value')
plt.xlabel('Iteration')
plt.show()

```

```

Iter0 Loss=0.7082914860900575
Iter10000 Loss=0.0036078600747076624
Iter20000 Loss=0.0021450140066445484
Iter30000 Loss=0.001334279347269829
Iter40000 Loss=0.0009417137558780359
Iter50000 Loss=0.00045881005291657355
Iter60000 Loss=0.0007415672213500783
Iter70000 Loss=0.00039677213095259007
Iter80000 Loss=0.0005210901048367791
Iter90000 Loss=0.00028276059026094636
Iter100000 Loss=0.00019361251279542173

```



trying higher learning rate of 0.1

In []:


```

nn = NeuralNetwork()
nn.add(FullyConnectedLayer(2,50))
nn.add(ReLU())
nn.add(FullyConnectedLayer(50,1))
loss_values = []

for iter in range(100001):
    input_data, target = sample_batch(100)

    output = nn.forwards(input_data)
    loss, grad_loss = euclidean_loss(output, target)
    nn.backwards(grad_loss)
    nn.update_param(step_size=0.1)

    if iter % 10000 == 0:
        print(f"Iter {iter} Loss={loss}")

    loss_values.append(loss)

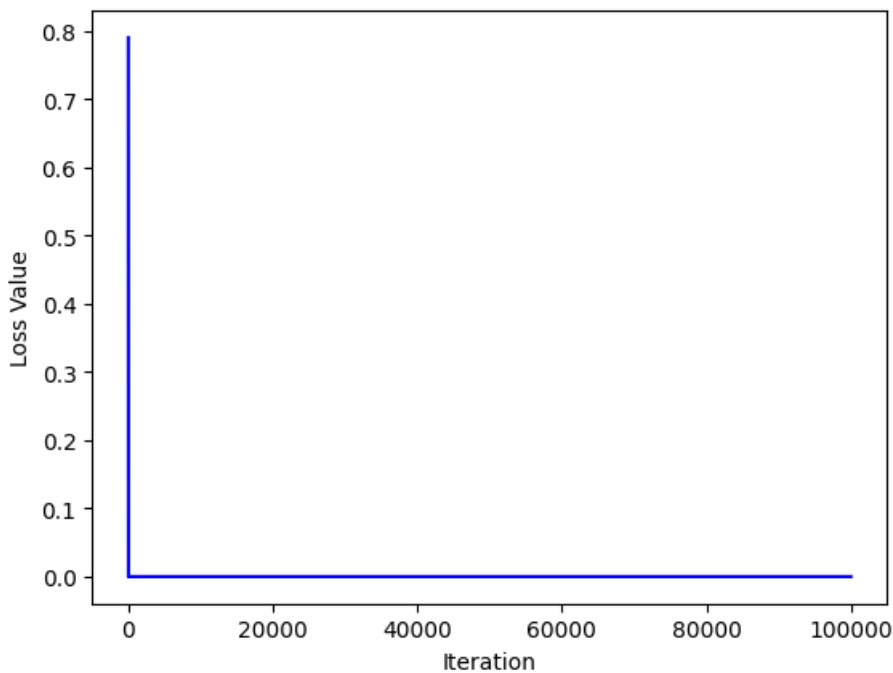
plt.clf()
plt.plot(loss_values, color='blue')
plt.ylabel('Loss Value')
plt.xlabel('Iteration')
plt.show()

```

```

Iter0 Loss=0.7907201820335065
Iter10000 Loss=1.0268025711438869e-07
Iter20000 Loss=1.0046271551130232e-08
Iter30000 Loss=3.844871392997895e-09
Iter40000 Loss=1.0177570891952546e-08
Iter50000 Loss=2.928515965049331e-09
Iter60000 Loss=3.047484641190921e-09
Iter70000 Loss=6.850561553493556e-09
Iter80000 Loss=1.2877289364665583e-09
Iter90000 Loss=7.159382313084575e-10
Iter100000 Loss=2.2210809438209946e-09

```



trying a higher batch size of 500

In []:

```

nn = NeuralNetwork()
nn.add(FullyConnectedLayer(2,50))
nn.add(ReLU())
nn.add(FullyConnectedLayer(50,1))
loss_values = []

for iter in range(100001):
    input_data, target = sample_batch(500)

    output = nn.forwards(input_data)
    loss, grad_loss = euclidean_loss(output, target)
    nn.backwards(grad_loss)
    nn.update_param(step_size=0.0001)

    if iter % 10000 == 0:
        print(f"Iter {iter} Loss={loss}")

    loss_values.append(loss)

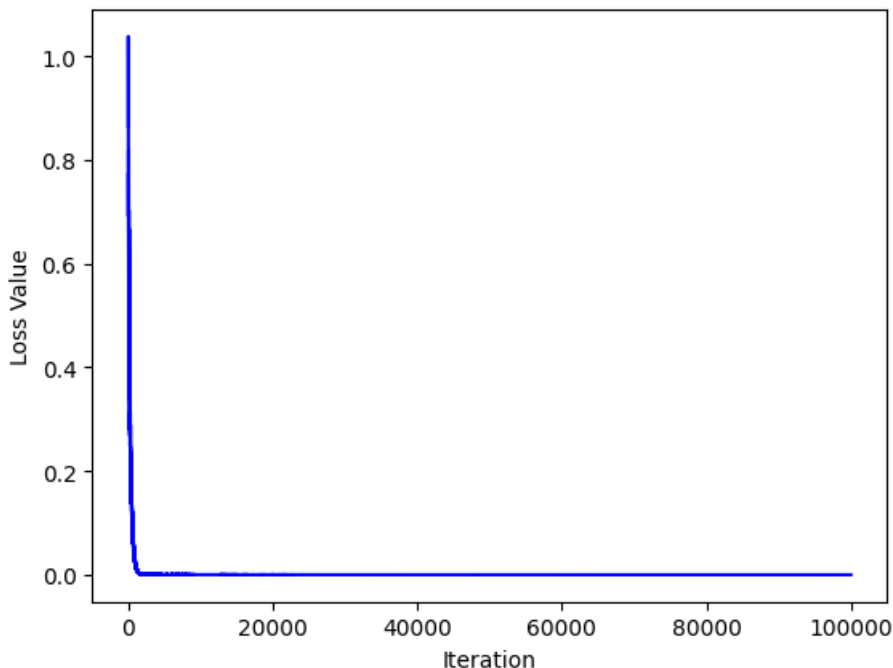
plt.clf()
plt.plot(loss_values, color='blue')
plt.ylabel('Loss Value')
plt.xlabel('Iteration')
plt.show()

```

```

Iter0 Loss=0.8655398635759874
Iter10000 Loss=0.00017126454702954204
Iter20000 Loss=0.00010109409700320243
Iter30000 Loss=4.732625013692646e-05
Iter40000 Loss=4.3264417811621744e-05
Iter50000 Loss=3.3305454136701525e-05
Iter60000 Loss=3.0385399247939892e-05
Iter70000 Loss=2.8171309698177166e-05
Iter80000 Loss=1.8989443338281665e-05
Iter90000 Loss=1.807640204009675e-05
Iter100000 Loss=1.469246868119795e-05

```



change the number of units in a multi neural network

In []:

```

nn = NeuralNetwork()
nn.add(FullyConnectedLayer(2,100))
nn.add(ReLU())
nn.add(FullyConnectedLayer(100,1))
loss_values = []

for iter in range(100001):
    input_data, target = sample_batch(100)

    output = nn.forwards(input_data)
    loss, grad_loss = euclidean_loss(output, target)
    nn.backwards(grad_loss)
    nn.update_param(step_size=0.0001)

    if iter % 10000 == 0:
        print(f"Iter {iter} Loss={loss}")

    loss_values.append(loss)

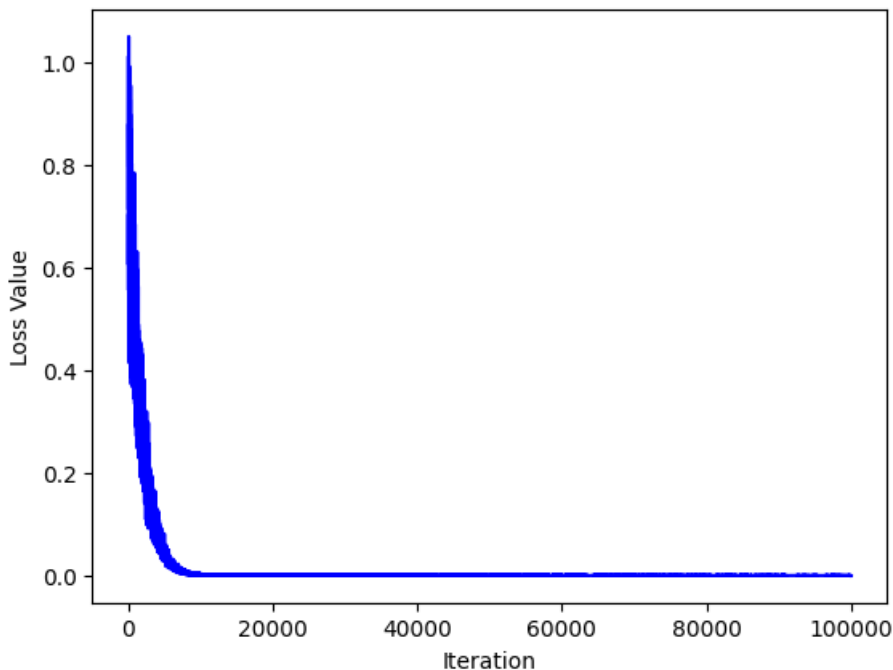
plt.clf()
plt.plot(loss_values, color='blue')
plt.ylabel('Loss Value')
plt.xlabel('Iteration')
plt.show()

```

```

Iter0 Loss=0.7154484227201688
Iter10000 Loss=0.001196942023190794
Iter20000 Loss=0.000601335439797646
Iter30000 Loss=0.0004094249845747489
Iter40000 Loss=0.0002886185220218139
Iter50000 Loss=0.00017593630128772019
Iter60000 Loss=0.00019877468561019926
Iter70000 Loss=0.00030798525057494414
Iter80000 Loss=0.00014969029949589758
Iter90000 Loss=0.0001651530483853054
Iter100000 Loss=0.0001690590667919935

```



Changing the sample batch function to work with a single layer network

In [38]:

```
def sample_batch(batch_size, ndim):
    x = np.random.randn(ndim, batch_size)
    y = np.expand_dims(np.abs(x[0, :]) - np.abs(x[1, :]), axis=0)
    return x, y
```

```
ndim = 100
nn = NeuralNetwork()
nn.add(FullyConnectedLayer(ndim,1))
loss_values = []
```

```
for iter in range(100001):
    input_data, target = sample_batch(100, ndim)

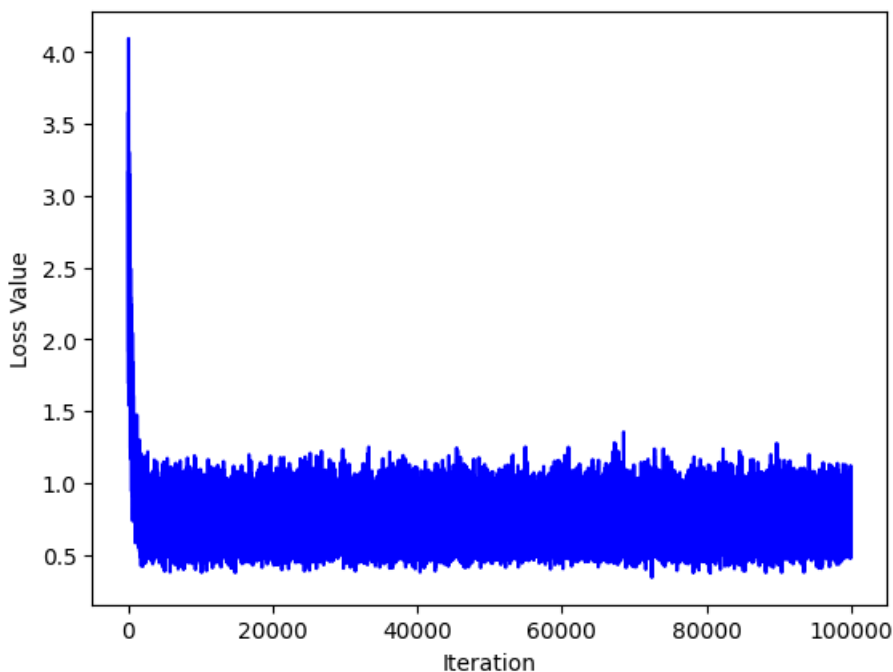
    output = nn.forwards(input_data)
    loss, grad_loss = euclidean_loss(output, target)
    nn.backwards(grad_loss)
    nn.update_param(step_size=0.0001)
```

```
if iter % 10000 == 0:
    print(f"Iter{iter} Loss={loss}")
```

```
loss_values.append(loss)
```

```
plt.clf()
plt.plot(loss_values, color='blue')
plt.ylabel('Loss Value')
plt.xlabel('Iteration')
plt.show()
```

```
Iter0 Loss=3.160277474409223
Iter10000 Loss=0.6575972439550288
Iter20000 Loss=0.8685373679213783
Iter30000 Loss=0.7821842357640033
Iter40000 Loss=0.7751007776253336
Iter50000 Loss=0.5888398364135108
Iter60000 Loss=0.7284124954493223
Iter70000 Loss=0.8828332466245673
Iter80000 Loss=0.5615422862639246
Iter90000 Loss=0.9227500465903773
Iter100000 Loss=0.8714722787742099
```



Short Answer Question: The above neural network uses a two-layer network. For the above `sample_batch()` function, will it work with a single layer? Why or why not?

Your Answer: Using a single layer that takes in dimensional input dim (ndim) data as input and outputs a 1 dimensional data, we can use the sample batch function. However, the loss value seems to flucuate around 1 regardless of number of iterations and hence it won't work effectively for a single layer as the data isn't linearly separable due to the absense of a hidden layer.

Short Answer Question: The loss curve usually goes down each iteration, but it is not guaranteed to go down. Give two reasons why.

Your Answer: The weight update is given by $w_{t+1} = w_t - k \frac{\partial L}{\partial w_t}$ where k is the learning rate. Looking at this equation, we can see that a high learning rate will cause the weight update to be high. Hence this causes the weight to overshoot and miss the point where the loss is minimum and hence cause loss curve to increase.

- Another reason is that neural networks are complex architectures. There could be a local minima or saddle points. If we reach a point where the loss reaches the local minima, there is no way for the loss to decrease when we keep updating the weights. It will be stuck at that local minimum and not decrease any further


Large Neural Networks

The neural network we have implemented above is fairly simple, but it illustrates the core concepts. The most advanced neural networks today are heavily engineered to efficiently work on GPUs and carefully tuned to train in a reasonable amount of time.

Before you continue, we need to do two things: attach a GPU to the Google Colab, and install some packages:

- To attach a GPU, click on the **Runtime** menu, then **Change runtime type**, and select **GPU** as a hardware accelerator. The notebook will automatically attach to the new runtime.
- We also need to install a few packages. Run the code below to automatically install them.

```
In [ ]:
!pip --quiet install ftfy regex tqdm
!pip --quiet install git+https://github.com/openai/CLIP.git
```



```
Preparing metadata (setup.py) ... done
Building wheel for clip (setup.py) ... done
```

Let's load some libraries. This will load PyTorch, which is a state-of-the-art library for deep learning. PyTorch is a very advanced version of the above simple neural network library that we built. We will also import `clip` which is a very large neural network built in PyTorch for natural images. Finally, we will also detect if there is GPU / CUDA support on your machine, and use it if possible. But if you don't have a GPU, this code will still work for you, and may just be a bit slower.

```
In [ ]:
import torch
import clip
from PIL import Image
device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
print(f'Using device: {device}')
```

Using device: cpu

If you have successfully attached a GPU to this runtime, it should output `Using device: cuda` above. If it says CPU instead, the code is not running on a GPU.

Now, let's load an image for us to operate on.

```
In [ ]:
!wget -qN https://www.cs.columbia.edu/~vondrick/class/coms4732/hw2/Lion.jpg
im = Image.open("Lion.jpg")
plt.imshow(im)
plt.show()
```



Let's load the already trained neural network, and print the neural network architecture. As you will see, this neural network will be huge consisting of hundreds of layers and different operations. However, the principles are the same as the toy neural network above. Each layer consists of a `forwards()` function, a `backwards()` function, and as `backwards_param()` if there are learnable parameters. They are all chained together with back-propagation.

In []:

```
model, preprocess = clip.load("ViT-B/32", device=device)
print(model)
```

100%  338M/338M [00:03<00:00, 98.2MiB/s]

```
CLIP(
  (visual): VisionTransformer(
    (conv1): Conv2d(3, 768, kernel_size=(32, 32), stride=(32, 32), bias=False)
    (ln_pre): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (transformer): Transformer(
      (resblocks): Sequential(
        (0): ResidualAttentionBlock(
          (attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
          )
          (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (mlp): Sequential(
            (c_fc): Linear(in_features=768, out_features=3072, bias=True)
            (gelu): QuickGELU()
            (c_proj): Linear(in_features=3072, out_features=768, bias=True)
          )
          (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
        (1): ResidualAttentionBlock(
          (attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
          )
          (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (mlp): Sequential(
            (c_fc): Linear(in_features=768, out_features=3072, bias=True)
            (gelu): QuickGELU()
            (c_proj): Linear(in_features=3072, out_features=768, bias=True)
          )
          (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
        (2): ResidualAttentionBlock(
          (attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
          )
          (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (mlp): Sequential(
            (c_fc): Linear(in_features=768, out_features=3072, bias=True)
            (gelu): QuickGELU()
            (c_proj): Linear(in_features=3072, out_features=768, bias=True)
          )
          (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
        (3): ResidualAttentionBlock(
          (attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
          )
          (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (mlp): Sequential(
            (c_fc): Linear(in_features=768, out_features=3072, bias=True)
```

```

        (gelu): QuickGELU()
        (c_proj): Linear(in_features=3072, out_features=768, bias=True)
    )
    (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(4): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(5): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(6): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(7): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(8): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(9): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
  (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(10): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=768, out_features=3072, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=3072, out_features=768, bias=True)
  )
)

```

```

        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
    (11): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
      )
      (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=768, out_features=3072, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=3072, out_features=768, bias=True)
      )
      (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
  )
  (ln_post): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
(transformer): Transformer(
  (resblocks): Sequential(
    (0): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
      )
      (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (1): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
      )
      (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (2): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
      )
      (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (3): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
      )
      (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (4): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)
        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
      )
      (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    )
    (5): ResidualAttentionBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
      )
      (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (mlp): Sequential(
        (c_fc): Linear(in_features=512, out_features=2048, bias=True)

```



```

        (gelu): QuickGELU()
        (c_proj): Linear(in_features=2048, out_features=512, bias=True)
    )
    (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(6): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(7): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(8): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(9): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(10): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
(11): ResidualAttentionBlock(
  (attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
  )
  (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (mlp): Sequential(
    (c_fc): Linear(in_features=512, out_features=2048, bias=True)
    (gelu): QuickGELU()
    (c_proj): Linear(in_features=2048, out_features=512, bias=True)
  )
  (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)
)
(token_embedding): Embedding(49408, 512)
(ln_final): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
)

```

This neural network is able to associate images with natural language text. This is called a **multi-modal** method, which is a topic that we will cover in the later lectures in the course. However, for now, it is simple enough to use directly. Let's define a few categorical labels, and see which one the neural network picks for this image. (You can read more about this network [here](#)).

In []:

```

def classify(im, labels):
    image = preprocess(im).unsqueeze(0).to(device)
    text = clip.tokenize(labels).to(device)

    with torch.no_grad():
        image_features = model.encode_image(image)
        text_features = model.encode_text(text)

        logits_per_image, logits_per_text = model(image, text)
        probs = logits_per_image.softmax(dim=-1).cpu().numpy()

    for i, label in enumerate(labels):
        print(f" {labels[i]} = {probs[0,i]*100}%")

labels = ["a truck", "a dog", "a cat", "a lion"]

classify(im, labels)

a truck = 0.0012693605640379246%
a dog = 0.0068436682340689%
a cat = 0.0896868237759918%
a lion = 99.90220069885254%
On our machine, this neural network is very confident that the image is a lion. Its second choice would be a cat, which makes sense!

```

Problem 4: Say Cheese

Let's experiment with the neural network a bit more. Can you find interesting cases where it works? and interesting cases where it does not?

Use the below code to take picture with your webcam.

In []:

```
#@title Take a webcam picture {display-mode: "form"}
```

```
from IPython.display import display, Javascript
from google.colab.output import eval_js
from base64 import b64decode
```

```
def take_photo(filename='photo.jpg', quality=0.8):
    js = Javascript("""
    async function takePhoto(quality) {
      const div = document.createElement('div');
      const capture = document.createElement('button');
      capture.textContent = 'Capture';
      div.appendChild(capture);

      const video = document.createElement('video');
      video.style.display = 'block';
      const stream = await navigator.mediaDevices.getUserMedia({video: true});

      document.body.appendChild(div);
      div.appendChild(video);
      video.srcObject = stream;
      await video.play();

      // Resize the output to fit the video element.
      google.colab.output.setIframeHeight(document.documentElement.scrollHeight, true);

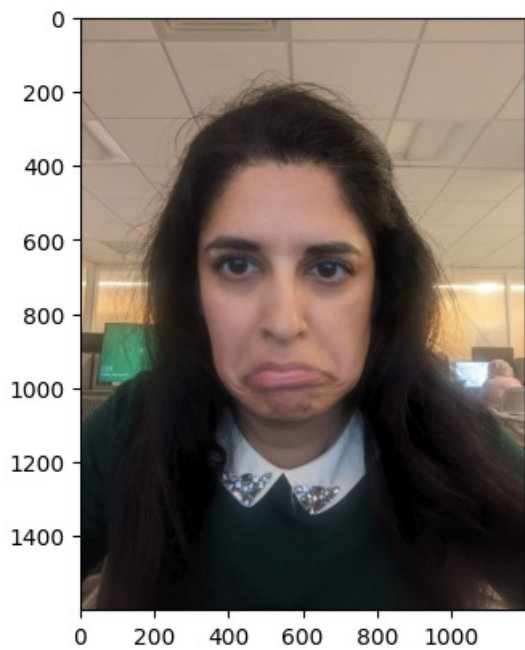
      // Wait for Capture to be clicked.
      await new Promise((resolve) => capture.onclick = resolve);

      const canvas = document.createElement('canvas');
      canvas.width = video.videoWidth;
      canvas.height = video.videoHeight;
      canvas.getContext('2d').drawImage(video, 0, 0);
      stream.getVideoTracks()[0].stop();
      div.remove();
      return canvas.toDataURL('image/jpeg', quality);
    }
    """)
    display(js)
    data = eval_js('takePhoto({}).format(quality)')
    binary = b64decode(data.split(',')[1])
    with open(filename, 'wb') as f:
        f.write(binary)
    return filename
In [ ]:
#take_photo('frowning_premu.jpg')
```

```
im = Image.open('frowning_premu.jpg')
plt.imshow(im)
plt.show()
```

```
labels = ["a person smiling", "a person frowning"]
```

```
classify(im, labels)
```



```
a person smiling = 0.8363667875528336%  
a person frowning = 99.1636335849762%
```

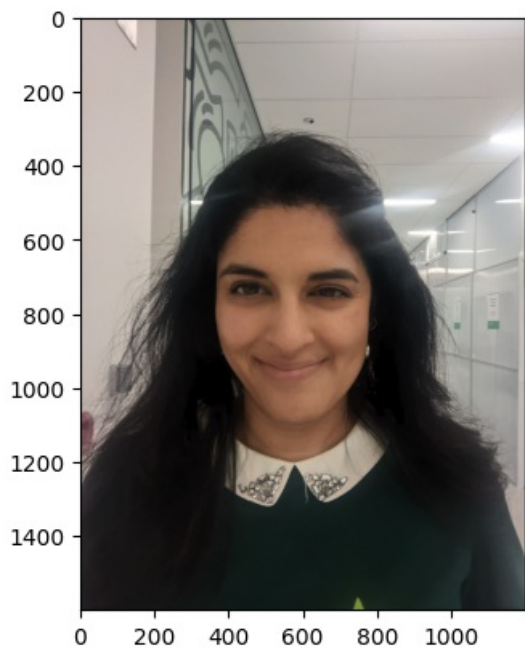
```
In [ ]:
```

```
#take_photo('frowning_premu.jpg')
```

```
im = Image.open('smiling.jpg')  
plt.imshow(im)  
plt.show()
```

```
labels = ["a person smiling", "a person frowning"]
```

```
classify(im, labels)
```



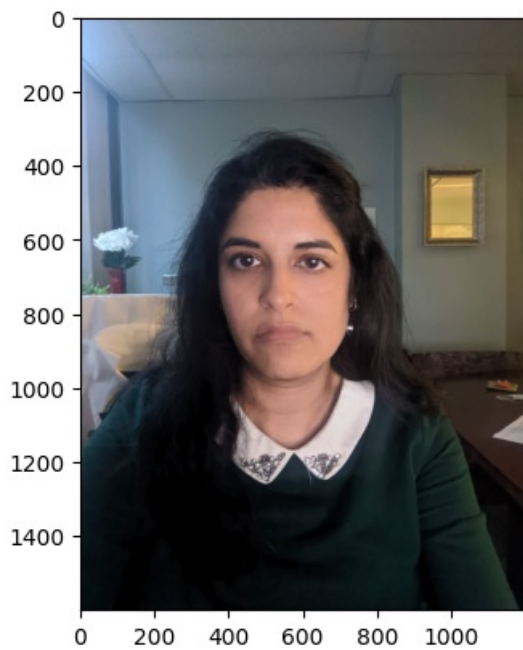
```
a person smiling = 99.4135856628418%  
a person frowning = 0.5864170845597982%
```

```
In [ ]:
```

```
im = Image.open('normal.jpg')  
plt.imshow(im)  
plt.show()
```

```
labels = ["a person smiling", "a person frowning"]
```

```
classify(im, labels)
```



a person smiling = 53.61819863319397%
a person frowning = 46.38180136680603%

Short Answer: Neural networks still have many failure modes. Experiment with different pictures from your webcam and different labels. What types of failures do you see?

Your Answer:

```
In [ ]:
im = Image.open('crying_prema.jpg')
plt.imshow(im)
plt.show()
```

labels = ["a person smiling", "a person frowning", "a person crying", "a person angry", "a person looks silly"]

classify(im, labels)



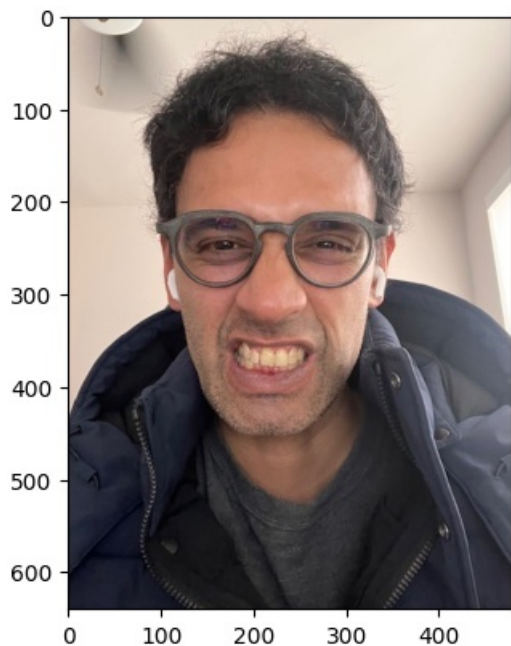
a person smiling = 12.184490263462067%
a person frowning = 35.860806703567505%
a person crying = 32.06746280193329%
a person angry = 8.135668933391571%
a person looks silly = 11.751574277877808%

In []:

```
im = Image.open('angry_raju.jpg')
plt.imshow(im)
plt.show()
```

```
labels = ["a person smiling", "a person frowning", "a person crying", "a person angry", "a person looks silly"]
```

```
classify(im, labels)
```



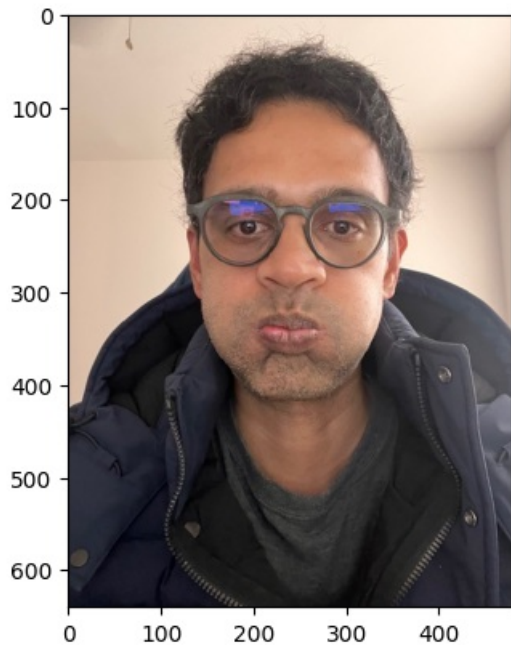
```
a person smiling = 52.6441752910614%
a person frowning = 3.525272011756897%
a person crying = 10.89218482375145%
a person angry = 15.074065327644348%
a person looks silly = 17.864297330379486%
```

```
In [ ]:
```

```
im = Image.open('silly_raju.jpg')
plt.imshow(im)
plt.show()
```

```
labels = ["a person smiling", "a person frowning", "a person crying", "a person angry", "a person looks silly"]
```

```
classify(im, labels)
```



```
a person smiling = 5.433856695890427%
a person frowning = 26.078608632087708%
a person crying = 6.304382532835007%
a person angry = 21.04077786207199%
a person looks silly = 41.14236831665039%
```

As we can see, neural network is not able to make a distinction between an angry and smiling person however it does accurately classify the person as silly even though its assigning a significant probability between angry and frowning

In []:

In []: