Clone the Entire Repository in Colab

```
!git clone https://github.com/patitimoner/workshop-chihuahua-vs-muffin.git

%cd workshop-chihuahua-vs-muffin

!ls
```

```
Cloning into 'workshop-chihuahua-vs-muffin'...
remote: Enumerating objects: 337, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 337 (delta 1), reused 4 (delta 1), pack-reused 330 (from 1)
Receiving objects: 100% (337/337), 14.51 MiB | 10.92 MiB/s, done.
Resolving deltas: 100% (82/82), done.
/content/workshop-chihuahua-vs-muffin
'CNN_1 Chihuahua or Muffin.ipynb'   README.md   workshop_1.ipynb
 data                               resources   workshop_1_output.ipynb
```

I. Project Overview

In this project, we'll build a neural network classifier that determines: **MUFFIN... or CHIHUAHUA!**

This is what we'll cover in the tutorial:

1) Build the neural network

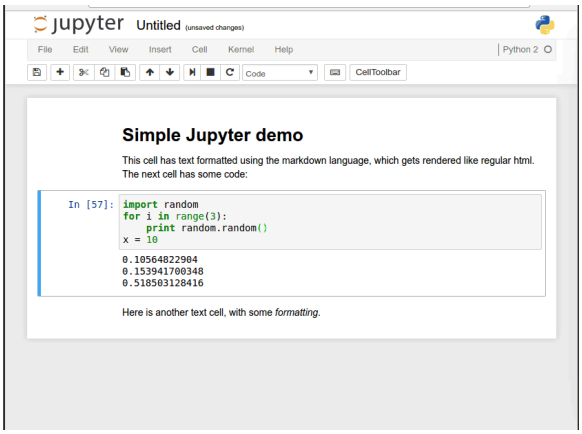
2) Load the data

3) Train the model on the data

4) Visualize the results

Remember: This is an INTERACTIVE Notebook!

You should run and play with the code as you go to see how it works. Select a cell and **press shift-enter to execute code.**



II. Deep Learning Tutorial

Let's get to the fun stuff!

Generic Python imports (select the below cell and press shift-enter to execute it)

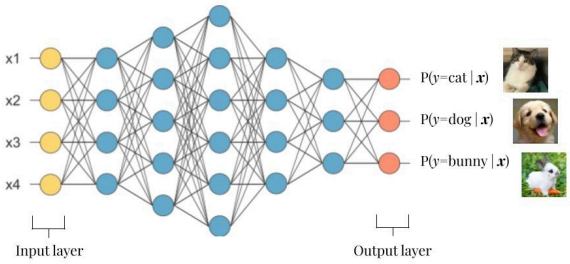
```
import matplotlib.pyplot as plt  # graphical library, to plot images
# special Jupyter notebook command to show plots inline instead of in a new window
%matplotlib inline
```

Deep learning imports

```
import torch                                        # PyTorch deep learning framework
from torchvision import datasets, models, transforms    # extension to PyTorch for dataset management
import torch.nn as nn                               # neural networks module of PyTorch, to let us define neural network layers
from torch.nn import functional as F                # special functions
import torch.optim as optim                         # optimizers
```

(1) Build our Neural Network

Recall from the lesson that a neural network generally looks like this. Input is on the left, output is on the right. The number of output neurons correspond to the number of classes.



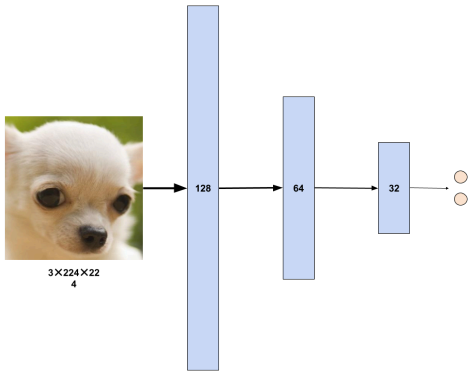
So let's define a similar architecture for our 2-class muffin-vs-chihuahua classifier:

```
#define image height and width
input_height = 224
input_width = 224

# Extends PyTorch's neural network baseclass
class MySkynet(nn.Module):
    """
    A very basic neural network.
    """
    def __init__(self, input_dim=(3, input_height, input_width)):
        """
        Constructs a neural network.

        input_dim: a tuple that represents "channel x height x width" dimensions of the input
        """
        super().__init__()
        # the total number of RGB pixels in an image is the tensor's volume
        num_in_features = input_dim[0] * input_dim[1] * input_dim[2]
        # input layer
        self.layer_0 = nn.Linear(num_in_features, 128)
        # hidden layers
        self.layer_1 = nn.Linear(128, 64)
        self.layer_2= nn.Linear(64, 32)
        # output layer, output size of 2 for chihuahua and muffin
        self.layer_3= nn.Linear(32, 2)

    def forward(self, x):
        """
        Define the forward pass through our network.
        """
        batch_size = x.shape[0]
        # convert our RGB tensor into one long vector
        x = x.view(batch_size, -1)

        # pass through our layers
        x = F.relu(self.layer_0(x))
        x = F.relu(self.layer_1(x))
        x = F.relu(self.layer_2(x))
        x = F.relu(self.layer_3(x))

        # convert the raw output to probability predictions
        x = F.softmax(x, dim=1)

        return x
```

Now that we've defined the network above, let's initialize it. If available, we'll place the network on the GPU; if not, it goes on the CPU.

```
 # cuda:0 means the first cuda device found
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = MySkynet().to(device)                      # load our simple neural network
model
```

```
MySkynet(
  (layer_0): Linear(in_features=150528, out_features=128, bias=True)
  (layer_1): Linear(in_features=128, out_features=64, bias=True)
  (layer_2): Linear(in_features=64, out_features=32, bias=True)
  (layer_3): Linear(in_features=32, out_features=2, bias=True)
)
```



Essentially, our network looks like this:

(2) Data and Data Loading

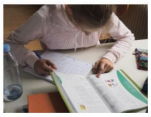
Separate "train" and "test" datasets

Recall from the below slide, we should make two separate datasets to train and test our model. That way, we know our model learns more than rote memorization.

When is Your ML Model Ready?

Without care, ML models "memorize" answers from training data.

Solution: evaluate performance on **test** set, separate from the **train** set



Homework == Train Set

Test == ... Test Set

Inspect our data

Let's look in our data folder to see what's there. As you can see, the folder is **split into "train" for training**, and **"validation" for testing** (to validate our model).

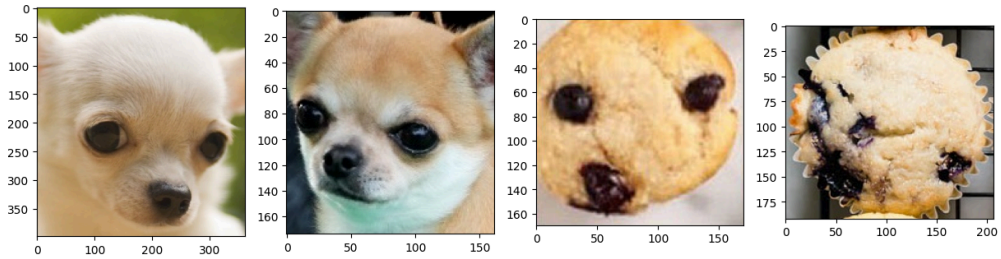
```
import os  # interact with the os. in our case, we want to view the file system

print("Data contents:", os.listdir("data"))
print("Train contents:", os.listdir("data/train"))
print("Validation contents:", os.listdir("data/validation"))
```

```
Data contents: ['train', 'validation']
Train contents: ['chihuahua', 'muffin']
Validation contents: ['chihuahua', 'muffin']
```

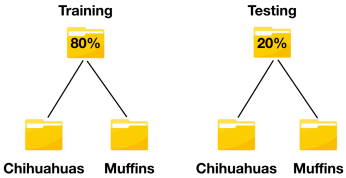
Let's also look at some of the images:

```
from PIL import Image  # import our image opening tool

_, ax = plt.subplots(1, 4, figsize=(15,60))  # to show 4 images side by side, make a "1 row x 4 column" axes
ax[0].imshow(Image.open("data/train/chihuahua/4.jpg"))  # show the chihuahua in the first column
ax[1].imshow(Image.open("data/train/chihuahua/5.jpg"))  # show the chihuahua in the second column
ax[2].imshow(Image.open("data/train/muffin/131.jpg"))   # show the muffin in the third column
ax[3].imshow(Image.open("data/train/muffin/107.jpg"))   # show the muffin in the fourth column
```

```
<matplotlib.image.AxesImage at 0x7d3d0c5c5fd0>
```



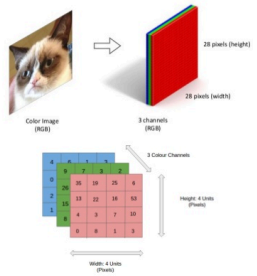
If you look in the data folder on your computer, there are 120 train images and 30 validation. So our data is split like this:



Load our data

That's great that we have data! But we have to load all the images and convert them into a form that our neural network understands. Specifically, PyTorch works with **Tensor** objects. (A tensor is just a multidimensional matrix, i.e. an N-d array.)

color image is 3rd-order tensor



To easily convert our image data into tensors, we use the help of a "dataloader." The dataloader packages data into convenient boxes for our model to use. You can think of it like one person passing boxes (tensors) to another.

First, we define some "transforms" to convert images to tensors. We must do so for both our train and validation datasets.

For more information about transforms, check out the link here: https://pytorch.org/docs/stable/torchvision/transforms.html

```
normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                 std=[0.5, 0.5, 0.5])

# transforms for our training data
train_transforms = transforms.Compose([
    # resize to resnet input size
    transforms.Resize((input_height,input_width)),
    # transform image to PyTorch tensor object
    transforms.ToTensor(),
    normalize
])

# these validation transforms are exactly the same as our train transforms
validation_transforms = transforms.Compose([
    transforms.Resize((input_height,input_width)),
    transforms.ToTensor(),
    normalize
])

print("Train transforms:", train_transforms)
```

```
Train transforms: Compose(
    Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=True)
    ToTensor()
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
)
```

Second, we create the datasets, by passing the transforms into the ImageFolder constructor.

These just represent the folders that hold the images.

```
# insert respective transforms to replace ?
image_datasets = {
    'train':
        datasets.ImageFolder('data/train', train_transforms),
    'validation':
        datasets.ImageFolder('data/validation', validation_transforms)}


print("==Train Dataset==\n", image_datasets["train"])
print()
print("==Validation Dataset==\n", image_datasets["validation"])
```

```
==Train Dataset==
 Dataset ImageFolder
    Number of datapoints: 120
    Root location: data/train
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=True)
               ToTensor()
               Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
           )

==Validation Dataset==
 Dataset ImageFolder
    Number of datapoints: 30
    Root location: data/validation
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=True)
               ToTensor()
               Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
           )
```

And finally, form dataloaders from the datasets:

```
# define batch size, number of images to load in at once
batch_size = 32

dataloaders = {
    'train':
        torch.utils.data.DataLoader(
            image_datasets['train'],
            batch_size=batch_size,
            shuffle=True,
            num_workers=4),
    'validation':
        torch.utils.data.DataLoader(
            image_datasets['validation'],
            batch_size=batch_size,
            shuffle=False,
            num_workers=4)}

print("Train loader:", dataloaders["train"])
print("Validation loader:", dataloaders["validation"])
```

```
Train loader: <torch.utils.data.dataloader.DataLoader object at 0x7d3d0c64e120>
Validation loader: <torch.utils.data.dataloader.DataLoader object at 0x7d3d07b4f800>
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which
  warnings.warn(
```

We can see a dataloader outputs 2 things: a BIG tensor to represent an image, and a vector to represent the labels (0 or 1).

```
next(iter(dataloaders["train"]))
```

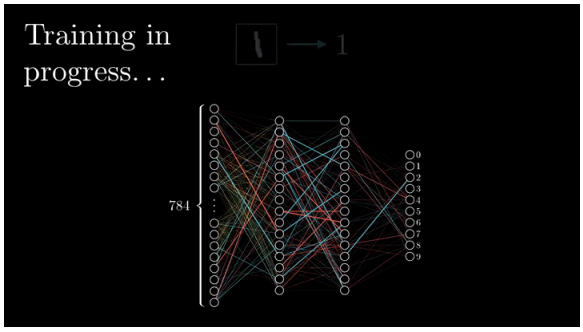
```
[tensor([[[[-0.1686, -0.1216, -0.0980,  ..., -0.4039, -0.4039, -0.4118],
           [-0.1294, -0.0824, -0.0510,  ..., -0.3804, -0.3804, -0.3882],
           [-0.1216, -0.0745, -0.0431,  ..., -0.3647, -0.3647, -0.3725],
           ...,
           [ 0.2549,  0.2784,  0.2549,  ..., -0.3569, -0.3412, -0.3333],
           [ 0.1843,  0.2314,  0.2314,  ..., -0.3569, -0.3490, -0.3412],
           [ 0.1608,  0.2157,  0.2078,  ..., -0.3804, -0.3725, -0.3647]],
```

```
tensor([[[[-0.1765, -0.1294, -0.1059,  ..., -0.3412, -0.3412, -0.3490],
          [-0.1294, -0.0824, -0.0588,  ..., -0.3412, -0.3412, -0.3490],
          [-0.1216, -0.0745, -0.0431,  ..., -0.3412, -0.3412, -0.3490],
          ...,
          [-0.1922, -0.1529, -0.1294,  ..., -0.4902, -0.4824, -0.4667],
          [-0.2078, -0.1529, -0.1294,  ..., -0.4667, -0.4588, -0.4510],
          [-0.2078, -0.1529, -0.1294,  ..., -0.4902, -0.4824, -0.4745]],

         [[-0.0510, -0.0039,  0.0196,  ..., -0.2549, -0.2549, -0.2627],
          [ 0.0431,  0.0980,  0.1294,  ..., -0.1216, -0.1216, -0.1294],
          [ 0.1216,  0.1686,  0.2078,  ..., -0.0196, -0.0196, -0.0275],
          ...,
          [-0.4118, -0.2941, -0.2000,  ..., -0.4745, -0.4588, -0.4510],
          [-0.4510, -0.3098, -0.2078,  ..., -0.4667, -0.4588, -0.4510],
          [-0.4431, -0.3020, -0.2000,  ..., -0.4902, -0.4824, -0.4745]]],


        [[[ 0.1843,  0.2235,  0.2784,  ...,  0.1686,  0.1843,  0.1922],
          [ 0.2000,  0.2235,  0.2549,  ...,  0.1451,  0.1765,  0.1922],
          [ 0.2078,  0.2157,  0.2314,  ...,  0.1451,  0.1765,  0.1922],
          ...,
          [-0.2157, -0.2314, -0.2471,  ..., -0.6235, -0.6157, -0.6078],
          [-0.2392, -0.2627, -0.2941,  ..., -0.6235, -0.6157, -0.6078],
          [-0.2941, -0.3176, -0.3569,  ..., -0.6157, -0.6078, -0.6000]],

         [[-0.0039,  0.0353,  0.0902,  ..., -0.0745, -0.0588, -0.0510],
          [ 0.0118,  0.0353,  0.0667,  ..., -0.0980, -0.0667, -0.0510],
          [ 0.0118,  0.0196,  0.0353,  ..., -0.1059, -0.0745, -0.0588],
          ...,
          [ 0.0980,  0.0824,  0.0667,  ..., -0.6235, -0.6157, -0.6078],
          [ 0.0745,  0.0510,  0.0196,  ..., -0.6235, -0.6157, -0.6078],
          [ 0.0196, -0.0039, -0.0431,  ..., -0.6157, -0.6078, -0.6000]],

         [[-0.0824, -0.0431,  0.0118,  ..., -0.2314, -0.2157, -0.2078],
          [-0.0667, -0.0431, -0.0118,  ..., -0.2549, -0.2235, -0.2078],
          [-0.0667, -0.0510, -0.0431,  ..., -0.2549, -0.2314, -0.2078],
          ...,
          [-0.6627, -0.6784, -0.6941,  ..., -0.6235, -0.6157, -0.6078],
          [-0.6863, -0.7098, -0.7412,  ..., -0.6235, -0.6157, -0.6078],
          [-0.7412, -0.7647, -0.8039,  ..., -0.6157, -0.6078, -0.6000]]],


        [[[ 0.4588,  0.4588,  0.4510,  ...,  0.5843,  0.6157,  0.6157],
          [ 0.4588,  0.4588,  0.4510,  ...,  0.5843,  0.6078,  0.6157],
          [ 0.4667,  0.4588,  0.4510,  ...,  0.6000,  0.6314,  0.6392],
          ...,
          [-0.4510, -0.4510, -0.4510,  ..., -0.5059, -0.5059, -0.4980],
          [-0.4588, -0.4588, -0.4588,  ..., -0.5059, -0.5059, -0.5059],
          [-0.4745, -0.4745, -0.4745,  ..., -0.4980, -0.4980, -0.5059]],
```

(4) Train the model!

Hurray! We've built a neural network and have data to give it. Now we **repeatedly iterate over the data to train the model.**

Every time the network gets a new example, it looks something like this. Note the **forward pass** and the corresponding **backward pass**.



Define the train loop

We want the network to learn from every example in our training dataset. However, the best performance comes from more practice.
Therefore, we **run through our dataset for multiple *epochs*.**

After each epoch, we'll check how our model performs on the validation set to monitor its progress.

```
from tqdm import tnrange, tqdm_notebook # import progress bars to show train progress

def train_model(model, dataloaders, loss_function, optimizer, num_epochs):
    """
    Trains a model using the given loss function and optimizer, for a certain number of epochs.

    model: a PyTorch neural network
    loss_function: a mathematical function that compares predictions and labels to return an error
    num_epochs: the number of times to run through the full training dataset
    """
    # train for n epochs. an epoch is a full iteration through our dataset
    for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
        # print a header
        print('Epoch {}/{}'.format(epoch+1, num_epochs))
        print('----------------')

        # first train over the dataset and update weights; at the end, calculate our validation performance
        for phase in ['train', 'validation']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            # keep track of the overall loss and accuracy for this batch
            running_loss = 0.0
            running_corrects = 0

            # iterate through the inputs and labels in our dataloader
            # (the tqdm_notebook part is to display a progress bar)
            for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
                # move inputs and labels to appropriate device (GPU or CPU)
                inputs = inputs.to(device)
                labels = labels.to(device)

                # FORWARD PASS
                outputs = model(inputs)
                # compute the error of the model's predictions
                loss = loss_function(outputs, labels)

                if phase == 'train':
                    # BACKWARD PASS
```

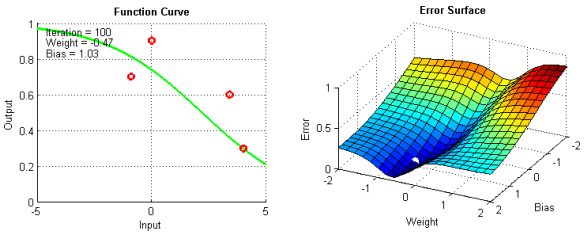
```
                optimizer.zero_grad()  # clear the previous gradients
                loss.backward()        # backpropagate the current error gradients
                optimizer.step()       # update the weights (i.e. do the learning)

            # track our accumulated loss
            running_loss += loss.item() * inputs.size(0)
            # track number of correct to compute accuracy
            _, preds = torch.max(outputs, 1)
            running_corrects += torch.sum(preds == labels.data)

        # print our progress
        epoch_loss = running_loss / len(image_datasets[phase])
        epoch_acc = running_corrects.double() / len(image_datasets[phase])
        print(f'{phase} error: {epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}')

    print()
```

Loss function and optimizer

One last thing: we must define a function that gives feedback for how well the model performs. This is the **loss**, or "error" **function**, that compares model predictions to the true labels.

Once we calculate the error, we also need to define how the model should react to that feedback. **The optimizer determines how the network learns from feedback.**

```
loss_function = nn.CrossEntropyLoss()             # the most common error function in deep learning
optimizer = optim.SGD(model.parameters(), lr=0.1)  # Stochastic Gradient Descent, with a learning rate of 0.1
```



Run training

Let's put everything together and TRAIN OUR MODEL! =D

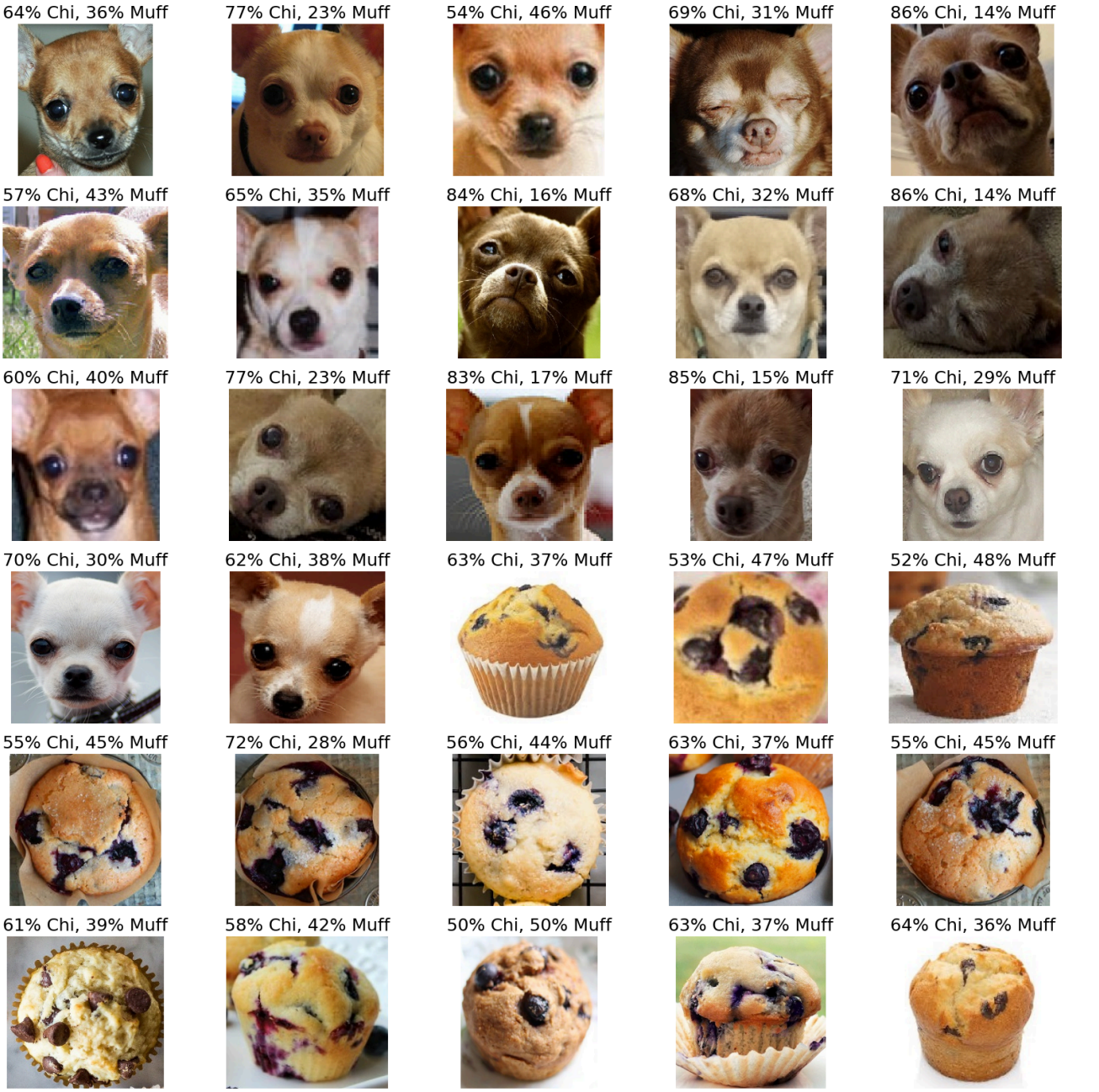
```
train_model(model, dataloaders, loss_function, optimizer, num_epochs=3)
```

```
/tmp/ipython-input-963760017.py:12: TqdmDeprecationWarning: Please use `tqdm.notebook.trange` instead of `tqdm.tnrange`
  for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
Total progress: 100%                                        3/3 [00:04<00:00,  1.43s/epoch]
Epoch 1/3
----------
/tmp/ipython-input-963760017.py:30: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
train error: 0.6851, Accuracy: 0.5417
validation error: 0.6702, Accuracy: 0.5667

Epoch 2/3
----------
train error: 0.6645, Accuracy: 0.5417
validation error: 0.6466, Accuracy: 0.5667

Epoch 3/3
----------
train error: 0.6362, Accuracy: 0.5417
validation error: 0.6278, Accuracy: 0.5667
```

Examine model performance

How do we examine our model's predictions? Let's visualize what the model thinks on the validation set.

```
from glob import glob
from math import floor

# get all the images from our validation sets
validation_img_paths = glob("data/validation/**/*.jpg", recursive=True)
images = [Image.open(img_path) for img_path in validation_img_paths]

# put all the images together to run through our model
validation_batch = torch.stack( [validation_transforms(img).to(device) for img in images])
pred_logits_tensor = model(validation_batch)
pred_probs = pred_logits_tensor.cpu().data.numpy()

# show the probabilities for each picture
fig, axs = plt.subplots(6, 5, figsize=(20, 20))
for i, img in enumerate(images):
    ax = axs[floor(i/5)][i % 5]
    ax.axis('off')
    ax.set_title("{:.0f}% Chi, {:.0f}% Muff".format(100*pred_probs[i,0], 100*pred_probs[i,1]), fontsize=18)
    ax.imshow(img)
```



Consider: How accurate was your model? How confident were its predictions? Does it make clear-cut decisions?

Congratulations! You've successfully trained a neural network!

III. Can You Do Better?

Now that we've shown you how to train a neural network, can you improve the validation accuracy by tweaking the parameters? **We challenge you to reach 100% accuracy!**

Some parameters to play with:

- Number of epochs
- The learning rate "lr" parameter in the optimizer
- The type of optimizer (https://pytorch.org/docs/stable/optim.html)
- Number of layers and layer dimensions
- Image size
- Data augmentation transforms (https://pytorch.org/docs/stable/torchvision/transforms.html)

IV. Personal Experiment (Challenges)

Improve the validation accuracy of the neural network to 100% by experimenting with different hyperparameters, network architecture, image size, and data augmentation techniques.

1. Experiment with hyperparameters

Modify the number of epochs, learning rate, and optimizer type to see if they improve accuracy. This code block modifies the `train_model` function call to increase the number of epochs and change the optimizer to Adam with a smaller learning rate to see if it improves the validation accuracy.

```
optimizer = optim.Adam(model.parameters(), lr=0.001) # Change optimizer to Adam with a smaller learning rate
train_model(model, dataloaders, loss_function, optimizer, num_epochs=10) # Increase the number of epochs
```

```
/tmp/ipython-input-963760017.py:12: TqdmDeprecationWarning: Please use `tqdm.notebook.trange` instead of `tqdm.tnrange`
  for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
Total progress: 100%                                                    10/10 [00:24<00:00,  2.55s/epoch]
Epoch 1/10
----------
/tmp/ipython-input-963760017.py:30: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
train error: 0.6058, Accuracy: 0.6083
validation error: 0.5552, Accuracy: 0.8000

Epoch 2/10
----------
train error: 0.4791, Accuracy: 0.9000
validation error: 0.5703, Accuracy: 0.7667

Epoch 3/10
----------
train error: 0.4319, Accuracy: 0.9250
validation error: 0.5144, Accuracy: 0.8333

Epoch 4/10
----------
train error: 0.4032, Accuracy: 0.9333
validation error: 0.5707, Accuracy: 0.7333

Epoch 5/10
----------
train error: 0.4141, Accuracy: 0.8917
validation error: 0.4391, Accuracy: 0.8667

Epoch 6/10
----------
train error: 0.3787, Accuracy: 0.9333
validation error: 0.4138, Accuracy: 0.9000

Epoch 7/10
----------
train error: 0.3784, Accuracy: 0.9333
validation error: 0.4136, Accuracy: 0.9000

Epoch 8/10
----------
train error: 0.3633, Accuracy: 0.9500
validation error: 0.4402, Accuracy: 0.8667

Epoch 9/10
----------
train error: 0.3679, Accuracy: 0.9417
validation error: 0.4455, Accuracy: 0.8667

Epoch 10/10
----------
train error: 0.3632, Accuracy: 0.9500
validation error: 0.4315, Accuracy: 0.8667
```

The validation accuracy did not reach 100% with the previous changes. I will try using a smaller learning rate with the Adam optimizer and increase the number of epochs further to see if it improves the accuracy.

```
optimizer = optim.Adam(model.parameters(), lr=0.0001) # Further decrease the learning rate
train_model(model, dataloaders, loss_function, optimizer, num_epochs=20) # Increase the number of epochs
```

```
/tmp/ipython-input-963760017.py:12: TqdmDeprecationWarning: Please use `tqdm.notebook.trange` instead of `tqdm.tnrange`
  for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
```

Total progress: 100% 20/20 [00:51<00:00, 2.49s/epoch]

```
Epoch 1/20
----------
/tmp/ipython-input-963760017.py:30: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
train error: 0.3550, Accuracy: 0.9583
validation error: 0.4459, Accuracy: 0.8667

Epoch 2/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4463, Accuracy: 0.8667

Epoch 3/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4463, Accuracy: 0.8667

Epoch 4/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4462, Accuracy: 0.8667

Epoch 5/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 6/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 7/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 8/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 9/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 10/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 11/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 12/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 13/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 14/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4460, Accuracy: 0.8667

Epoch 15/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 16/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 17/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 18/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 19/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667

Epoch 20/20
----------
train error: 0.3549, Accuracy: 0.9583
validation error: 0.4461, Accuracy: 0.8667
```

The validation accuracy is still not reaching 100%. I will try a different optimizer, Adagrad, with a slightly larger learning rate and keep the number of epochs at 20 to see if this combination performs better.

```
optimizer = optim.Adagrad(model.parameters(), lr=0.01) # Change optimizer to Adagrad
train_model(model, dataloaders, loss_function, optimizer, num_epochs=20) # Keep the number of epochs at 20
```

```
/tmp/ipython-input-963760017.py:12: TqdmDeprecationWarning: Please use `tqdm.notebook.trange` instead of `tqdm.tnrange`
  for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
Total progress: 100%                                                20/20 [00:39<00:00,  2.11s/epoch]
Epoch 1/20
----------
/tmp/ipython-input-963760017.py:30: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
train error: 0.4939, Accuracy: 0.8167
validation error: 0.7133, Accuracy: 0.6000

Epoch 2/20
----------
train error: 0.6971, Accuracy: 0.6167
validation error: 0.5782, Accuracy: 0.7333

Epoch 3/20
----------
train error: 0.5716, Accuracy: 0.7417
validation error: 0.5799, Accuracy: 0.7333

Epoch 4/20
----------
train error: 0.6170, Accuracy: 0.6917
validation error: 0.6466, Accuracy: 0.6667

Epoch 5/20
----------
train error: 0.6466, Accuracy: 0.6667
validation error: 0.6451, Accuracy: 0.6667

Epoch 6/20
----------
train error: 0.6383, Accuracy: 0.6750
validation error: 0.6449, Accuracy: 0.6667

Epoch 7/20
----------
train error: 0.6382, Accuracy: 0.6750
validation error: 0.6133, Accuracy: 0.7000

Epoch 8/20
----------
train error: 0.5878, Accuracy: 0.7250
validation error: 0.6456, Accuracy: 0.6667

Epoch 9/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6449, Accuracy: 0.6667

Epoch 10/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6442, Accuracy: 0.6667

Epoch 11/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6439, Accuracy: 0.6667

Epoch 12/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6436, Accuracy: 0.6667

Epoch 13/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6434, Accuracy: 0.6667

Epoch 14/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6431, Accuracy: 0.6667

Epoch 15/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6427, Accuracy: 0.6667

Epoch 16/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6422, Accuracy: 0.6667

Epoch 17/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6414, Accuracy: 0.6667

Epoch 18/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6399, Accuracy: 0.6667

Epoch 19/20
----------
train error: 0.5966, Accuracy: 0.7167
validation error: 0.6409, Accuracy: 0.6667

Epoch 20/20
----------
train error: 0.5883, Accuracy: 0.7250
validation error: 0.6400, Accuracy: 0.6667
```

2. Modify network architecture

Adjust the number of layers and layer dimensions in the `MySkynet` class. This code block modifies the `MySkynet` class to adjust the number of layers and layer dimensions. Specifically, it increases the dimensions of the existing layers and adds a new hidden layer. It then instantiates a new model with this modified architecture and redefines the optimizer to use the parameters of this new model instance. The output shows the updated architecture of the `MySkynet` model.

```
# define image height and width
input_height = 224
input_width = 224

# Extends PyTorch's neural network baseclass
class MySkynet(nn.Module):
    """
    A modified neural network with different layer dimensions.
    """
    def __init__(self, input_dim=(3, input_height, input_width)):
        """
        Constructs a neural network.

        input_dim: a tuple that represents "channel x height x width" dimensions of the input
        """
```

```
        super().__init__()
        # the total number of RGB pixels in an image is the tensor's volume
        num_in_features = input_dim[0] * input_dim[1] * input_dim[2]
        # input layer
        self.layer_0 = nn.Linear(num_in_features, 256) # Increased dimension
        # hidden layers
        self.layer_1 = nn.Linear(256, 128) # Increased dimension
        self.layer_2= nn.Linear(128, 64) # Increased dimension
        self.layer_3= nn.Linear(64, 32) # Added a new layer
        # output layer, output size of 2 for chihuahua and muffin
        self.layer_4= nn.Linear(32, 2) # Adjusted layer index

    def forward(self, x):
        """
        Define the forward pass through our network.
        """
        batch_size = x.shape[0]
        # convert our RGB tensor into one long vector
        x = x.view(batch_size, -1)

        # pass through our layers
        x = F.relu(self.layer_0(x))
        x = F.relu(self.layer_1(x))
        x = F.relu(self.layer_2(x))
        x = F.relu(self.layer_3(x)) # Pass through the new layer
        x = F.relu(self.layer_4(x)) # Pass through the adjusted output layer

        # convert the raw output to probability predictions
        x = F.softmax(x, dim=1)

        return x

# Instantiate a new model with the modified architecture
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = MySkynet().to(device)

# Redefine the optimizer to use the parameters of the new model instance
optimizer = optim.SGD(model.parameters(), lr=0.1)

print(model)
```

```
MySkynet(
  (layer_0): Linear(in_features=150528, out_features=256, bias=True)
  (layer_1): Linear(in_features=256, out_features=128, bias=True)
  (layer_2): Linear(in_features=128, out_features=64, bias=True)
  (layer_3): Linear(in_features=64, out_features=32, bias=True)
  (layer_4): Linear(in_features=32, out_features=2, bias=True)
)
```

3. Experiment with image size and data augmentation

Change the input image size and add more data augmentation transforms to the dataloaders. This code block modifies the input image size and adds more data augmentation transforms to the training dataloader. Specifically, it changes the `input_height` and `input_width` variables to 256, redefines the `MySkynet` class to accommodate the new input size, and adds `transforms.RandomHorizontalFlip` and `transforms.ColorJitter` to the `train_transforms`. It then recreates the `image_datasets` and `dataloaders` with the updated transforms and input size. The output will show the modified train and validation transforms, the new input dimensions, and the updated model architecture.

```
# Modify image height and width
input_height = 256
input_width = 256

# Redefine the network with the new input size
class MySkynet(nn.Module):
    """
    A modified neural network with different layer dimensions and input size.
    """
    def __init__(self, input_dim=(3, input_height, input_width)):
        """
        Constructs a neural network.

        input_dim: a tuple that represents "channel x height x width" dimensions of the input
        """
        super().__init__()
        # the total number of RGB pixels in an image is the tensor's volume
        num_in_features = input_dim[0] * input_dim[1] * input_dim[2]
        # input layer
        self.layer_0 = nn.Linear(num_in_features, 256) # Increased dimension
        # hidden layers
        self.layer_1 = nn.Linear(256, 128) # Increased dimension
        self.layer_2= nn.Linear(128, 64) # Increased dimension
        self.layer_3= nn.Linear(64, 32) # Added a new layer
        # output layer, output size of 2 for chihuahua and muffin
        self.layer_4= nn.Linear(32, 2) # Adjusted layer index

    def forward(self, x):
        """
        Define the forward pass through our network.
        """
        batch_size = x.shape[0]
        # convert our RGB tensor into one long vector
        x = x.view(batch_size, -1)

        # pass through our layers
        x = F.relu(self.layer_0(x))
        x = F.relu(self.layer_1(x))
        x = F.relu(self.layer_2(x))
        x = F.relu(self.layer_3(x)) # Pass through the new layer
        x = F.relu(self.layer_4(x)) # Pass through the adjusted output layer


        # convert the raw output to probability predictions
        x = F.softmax(x, dim=1)

        return x

# Instantiate a new model with the modified architecture and input size
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = MySkynet().to(device)

# Redefine the optimizer to use the parameters of the new model instance
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Add more transforms to the training data
train_transforms = transforms.Compose([
    transforms.Resize((input_height,input_width)),
    transforms.RandomHorizontalFlip(p=0.5),  # Add random horizontal flip
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1), # Add color jitter
    transforms.ToTensor(),
    normalize
])
```

```
# These validation transforms remain the same
validation_transforms = transforms.Compose([
    transforms.Resize((input_height,input_width)),
    transforms.ToTensor(),
    normalize
])

# Recreate the datasets
image_datasets = {
    'train':
        datasets.ImageFolder('data/train', train_transforms),
    'validation':
        datasets.ImageFolder('data/validation', validation_transforms)}

# Recreate the dataloaders
batch_size = 32

dataloaders = {
    'train':
        torch.utils.data.DataLoader(
            image_datasets['train'],
            batch_size=batch_size,
            shuffle=True,
            num_workers=4),
    'validation':
        torch.utils.data.DataLoader(
            image_datasets['validation'],
            batch_size=batch_size,
            shuffle=False,
            num_workers=4)}

print("Train transforms:", train_transforms)
print("Validation transforms:", validation_transforms)
print("New input height:", input_height)
print("New input width:", input_width)
print("New model architecture:", model)
```

```
Train transforms: Compose(
    Resize(size=(256, 256), interpolation=bilinear, max_size=None, antialias=True)
    RandomHorizontalFlip(p=0.5)
    ColorJitter(brightness=(0.8, 1.2), contrast=(0.8, 1.2), saturation=(0.8, 1.2), hue=(-0.1, 0.1))
    ToTensor()
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
)
Validation transforms: Compose(
    Resize(size=(256, 256), interpolation=bilinear, max_size=None, antialias=True)
    ToTensor()
    Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
)
New input height: 256
New input width: 256
New model architecture: MySkynet(
  (layer_0): Linear(in_features=196608, out_features=256, bias=True)
  (layer_1): Linear(in_features=256, out_features=128, bias=True)
  (layer_2): Linear(in_features=128, out_features=64, bias=True)
  (layer_3): Linear(in_features=64, out_features=32, bias=True)
  (layer_4): Linear(in_features=32, out_features=2, bias=True)
)
```

4. Retrain and evaluate

Train the model with the modified parameters and evaluate its performance on the validation set. This code block trains the model with the modified parameters and evaluates its performance on the validation set by calling the `train_model` function with the updated `model`, `dataloaders`, `loss_function`, and `optimizer` for 20 epochs. The output will show the training and validation loss and accuracy for each epoch.

```
train_model(model, dataloaders, loss_function, optimizer, num_epochs=20)
```

```
/tmp/ipython-input-963760017.py:12: TqdmDeprecationWarning: Please use `tqdm.notebook.trange` instead of `tqdm.tnrange`
  for epoch in tnrange(num_epochs, desc="Total progress", unit="epoch"):
Total progress: 100%                                                20/20 [01:09<00:00,  3.48s/epoch]
Epoch 1/20
----------
/tmp/ipython-input-963760017.py:30: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for inputs, labels in tqdm_notebook(dataloaders[phase], desc=phase, unit="batch", leave=False):
train error: 0.6885, Accuracy: 0.6750
validation error: 0.6740, Accuracy: 0.7667

Epoch 2/20
----------
train error: 0.6660, Accuracy: 0.8083
validation error: 0.6428, Accuracy: 0.7000

Epoch 3/20
----------
train error: 0.6465, Accuracy: 0.7667
validation error: 0.6052, Accuracy: 0.9000

Epoch 4/20
----------
train error: 0.5779, Accuracy: 0.8917
validation error: 0.5223, Accuracy: 0.9333

Epoch 5/20
----------
train error: 0.5229, Accuracy: 0.8667
validation error: 0.4855, Accuracy: 0.8667

Epoch 6/20
----------
train error: 0.4575, Accuracy: 0.9167
validation error: 0.4758, Accuracy: 0.8667

Epoch 7/20
----------
train error: 0.4448, Accuracy: 0.8917
validation error: 0.4414, Accuracy: 0.8667

Epoch 8/20
----------
train error: 0.4116, Accuracy: 0.9250
validation error: 0.4239, Accuracy: 0.8667

Epoch 9/20
----------
train error: 0.3944, Accuracy: 0.9250
validation error: 0.4032, Accuracy: 0.9333

Epoch 10/20
----------
train error: 0.4318, Accuracy: 0.8917
validation error: 0.3943, Accuracy: 0.9333

Epoch 11/20
----------
train error: 0.4134, Accuracy: 0.9083
validation error: 0.4897, Accuracy: 0.8000

Epoch 12/20
----------
train error: 0.4354, Accuracy: 0.8667
validation error: 0.3751, Accuracy: 0.9333

Epoch 13/20
----------
train error: 0.3694, Accuracy: 0.9500
validation error: 0.3610, Accuracy: 0.9667

Epoch 14/20
----------
train error: 0.3740, Accuracy: 0.9500
validation error: 0.4517, Accuracy: 0.8333

Epoch 15/20
----------
train error: 0.3781, Accuracy: 0.9417
validation error: 0.3953, Accuracy: 0.9000

Epoch 16/20
----------
train error: 0.3559, Accuracy: 0.9583
validation error: 0.4527, Accuracy: 0.8333

Epoch 17/20
----------
train error: 0.3577, Accuracy: 0.9667
validation error: 0.4651, Accuracy: 0.8333

Epoch 18/20
----------
train error: 0.3547, Accuracy: 0.9667
validation error: 0.3961, Accuracy: 0.9333

Epoch 19/20
----------
train error: 0.3515, Accuracy: 0.9667
validation error: 0.5020, Accuracy: 0.7667

Epoch 20/20
----------
train error: 0.3373, Accuracy: 0.9833
validation error: 0.4007, Accuracy: 0.9333
```

5. Repeat and iterate

Continue experimenting with different combinations of parameters until the desired accuracy is reached or the performance plateaus. This code block continues the experimentation by increasing the number of epochs and changing the learning rate to see if further training with the current architecture and data augmentation improves the validation accuracy. Specifically, it keeps the current network architecture and data augmentation techniques in place. It then defines a new optimizer, changing it to `optim.SGD` with a decreased learning rate of `0.01`. Finally, it calls the `train_model` function with the current `model`, `dataloaders`, and `loss_function`, using the newly defined `optimizer` and increasing the number of training epochs to `30`. The output will display the training and validation loss and accuracy for each of the 30 epochs, allowing for an assessment of the model's performance with these updated parameters.

```
# Keep the current architecture and data augmentation, but increase epochs and decrease learning rate
optimizer = optim.SGD(model.parameters(), lr=0.01) # Decreased learning rate
train_model(model, dataloaders, loss_function, optimizer, num_epochs=30) # Increased number of epochs
```