I. Project Overview

```
!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
Receiving objects: 100% (337/337), 14.51 MiB | 27.62 MiB/s, done.
Resolving deltas: 100% (82/82), done.
/content/workshop-chihuahua-vs-muffin/workshop-chihuahua-vs-muffin/workshop-chihuahua-vs-muffin/workshop-chihuahua-vs-muffin
'CNN_1 Chihuahua or Muffin.ipynb'   README.md   workshop_1.ipynb
 data                               resources   workshop_1_output.ipynb
```

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

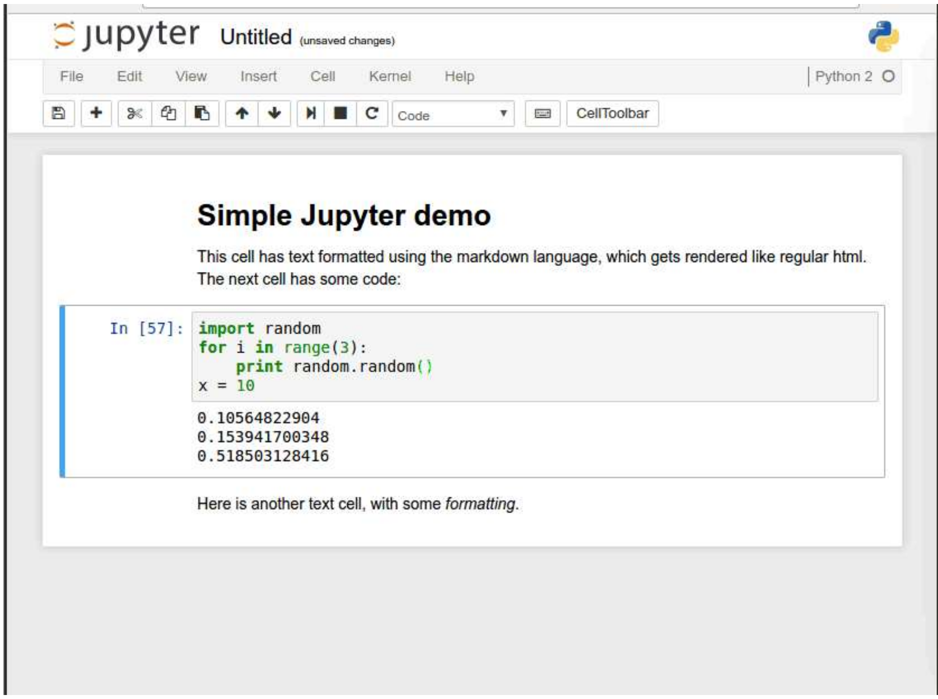
2) Load the data

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)

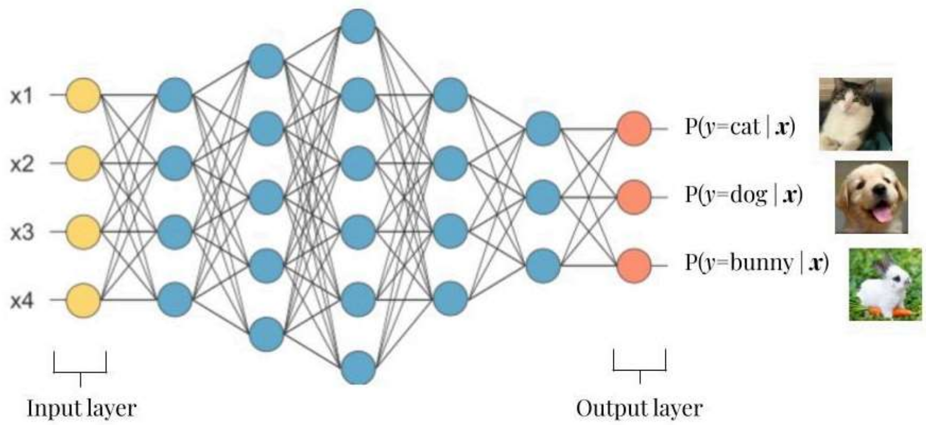
```
import matplotlib.pyplot as plt  # graphical library, to plot images
# special Jupyter notebook command to show plots inline instead of in a new window
%matplotlib inline
```

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:

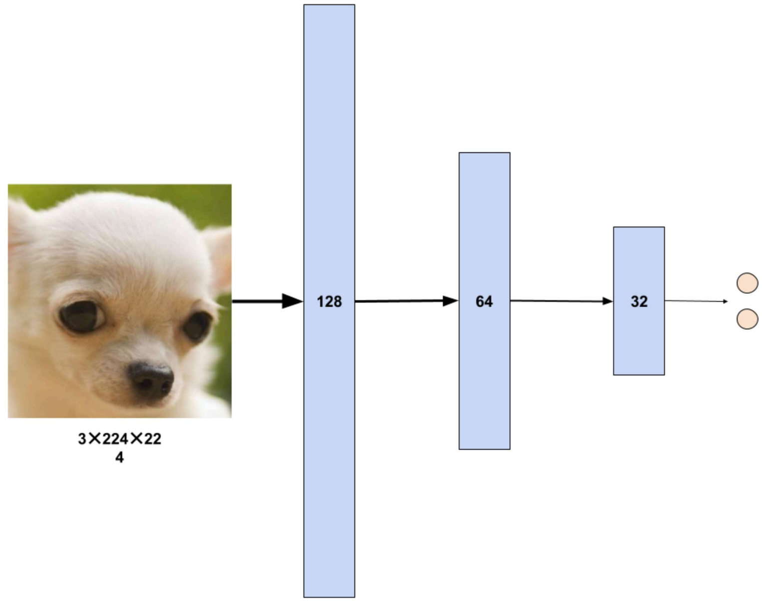
```
#define image height and width
input_height = 128
input_width = 128

# 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=49152, 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)
  )
```

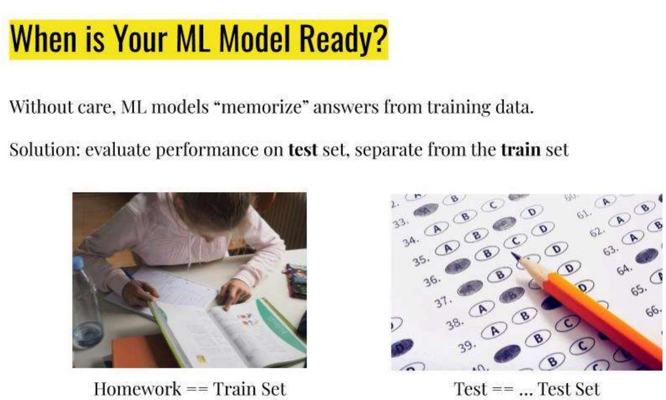


Essentially, our network looks like this:

(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.



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).

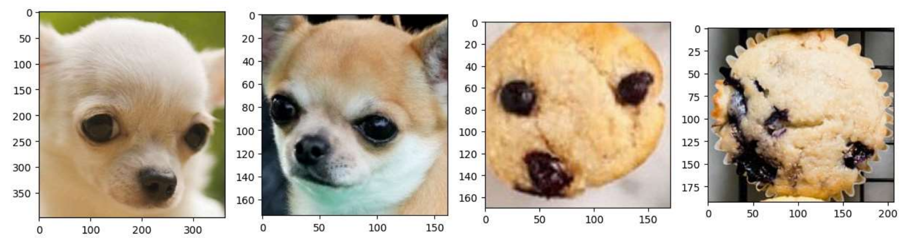
```
import os  # interact with the os. in our case, we want to view the file system

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']
```

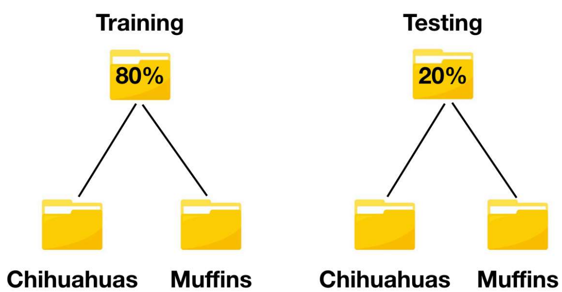
Let's also look at some of the images:

```
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 0x7f09742e14e0>
```

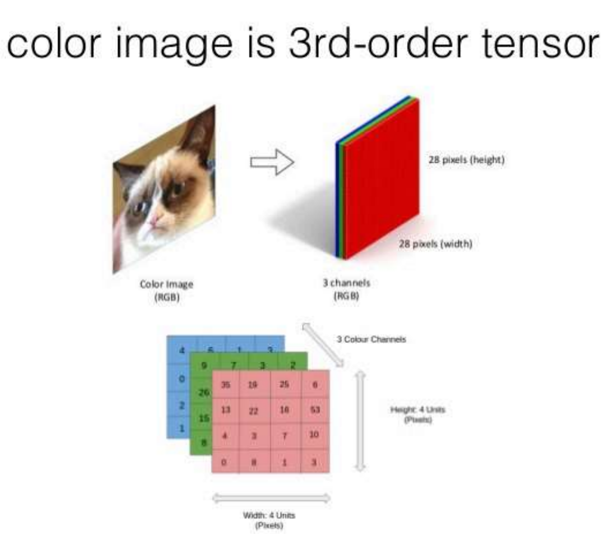


If you look in the data folder on your computer, there are 120 train images and 30 validation. So our data is split like this:



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.)



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=(128, 128), 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["train"])
```

```
==Train Dataset==
 Dataset ImageFolder
    Number of datapoints: 120
    Root location: data/train
    StandardTransform
Transform: Compose(
               Resize(size=(128, 128), 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: 120
    Root location: data/train
    StandardTransform
Transform: Compose(
               Resize(size=(128, 128), 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

dataloaders = {
    'train':
        torch.utils.data.DataLoader(
            image_datasets['train'],
            batch_size=8,
            shuffle=True,
            num_workers=4),
    'validation':
        torch.utils.data.DataLoader(
            image_datasets['validation'],
            batch_size=8,
            shuffle=False,
            num_workers=4)}

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

```
Train loader: <torch.utils.data.dataloader.DataLoader object at 0x7f09742238e0>
Validation loader: <torch.utils.data.dataloader.DataLoader object at 0x7f097444af20>
```

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).

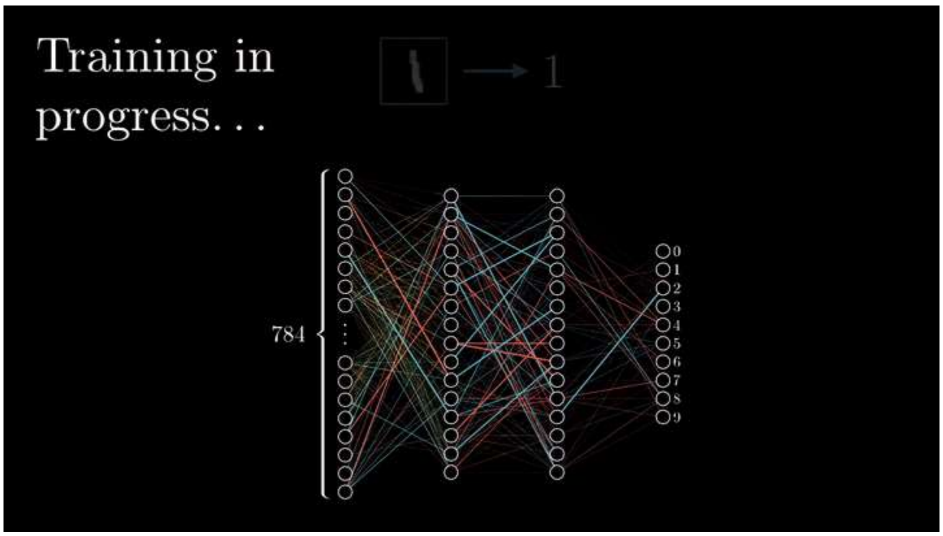
```
next(iter(dataloaders["train"]))
```

```
          [-0.1294, -0.1529, -0.1843,  ..., -0.0745, -0.0824, -0.0824],
          [-0.1294, -0.1529, -0.1843,  ..., -0.0824, -0.0824, -0.0824]],

         [[-0.3569, -0.3412, -0.3255,  ..., -0.6863, -0.6863, -0.6863],
          [-0.1529, -0.1373, -0.1216,  ..., -0.3882, -0.3804, -0.3804],
          [-0.1608, -0.1451, -0.1294,  ..., -0.3647, -0.3647, -0.3569],
          ...,
          [-0.1765, -0.2000, -0.2314,  ..., -0.1373, -0.1373, -0.1451],
          [-0.1765, -0.2000, -0.2314,  ..., -0.1451, -0.1529, -0.1529],
          [-0.1765, -0.2000, -0.2314,  ..., -0.1529, -0.1529, -0.1529]],

         [[-0.3647, -0.3490, -0.3333,  ..., -0.6941, -0.7176, -0.7176],
          [-0.1686, -0.1451, -0.1294,  ..., -0.4039, -0.4118, -0.4118],
          [-0.1765, -0.1529, -0.1373,  ..., -0.3961, -0.3961, -0.3961],
          ...,
          [-0.1765, -0.2000, -0.2314,  ..., -0.1294, -0.1294, -0.1373],
          [-0.1765, -0.2000, -0.2314,  ..., -0.1373, -0.1451, -0.1451],
          [-0.1765, -0.2000, -0.2314,  ..., -0.1451, -0.1451, -0.1451]]]]),
 tensor([1, 1, 1, 1, 0, 1, 1, 1])]
```

(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:
```

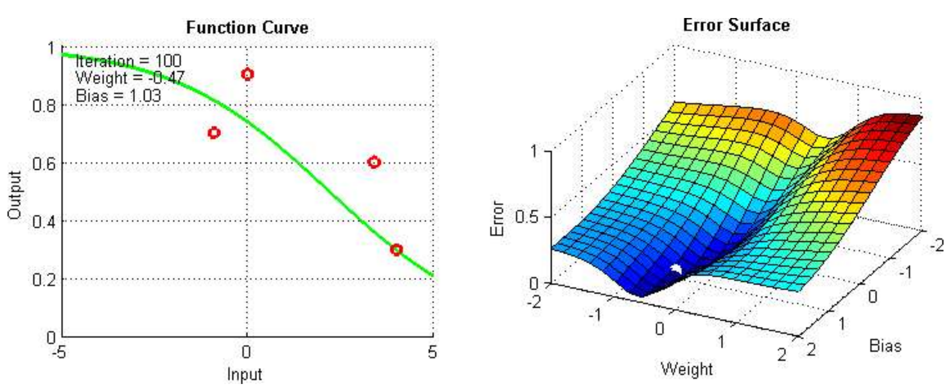
```
            model.eval()

        # 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)
```

```
<ipython-input-63-c9fe14cbe6c0>: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,  1.38s/epoch]
Epoch 1/3
----------------
<ipython-input-63-c9fe14cbe6c0>: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: 0.6416, Accuracy: 0.6833
validation error: 0.5566, Accuracy: 0.8667

Epoch 2/3
----------------
train error: 0.4829, Accuracy: 0.8833
validation error: 0.4537, Accuracy: 0.9000

Epoch 3/3
----------------
train error: 0.3835, Accuracy: 0.9500
validation error: 0.4301, Accuracy: 0.8667
```

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?