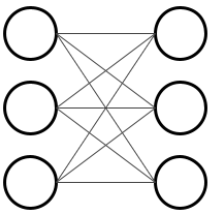
# Lecture 7: CNNs in Practice

## Contents





**DSCI 572**

Supervised Learning II

## Lecture Learning Objectives

- Load image data using `torchvision.datasets.ImageFolder()` to train a network in PyTorch
- Explain what “data augmentation” is and why we might want to do it
- Be able to save and re-load a PyTorch model
- Tune the hyperparameters of a PyTorch model using Ax

- Describe what transfer learning is and the different flavours of it: “out-of-the-box”, “feature extractor”, “fine tuning”

# Imports

```
import json
import numpy as np
import pandas as pd
from collections import OrderedDict
import torch
import torchvision
from torch import nn, optim
from torchvision import transforms, models, datasets
from torchsummary import summary
from PIL import Image
from matplotlib import pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
```

```
import sys, os
sys.path.append(os.path.join(os.path.abspath(".."), "code"))
from plotting import *
from set_seed import *
DATA_DIR = DATA_DIR = os.path.join(os.path.abspath(".."), "data/")
```

```
set_seed(1)
```

```
# !conda install -y memory_profiler
```

```
import memory_profiler
```

```
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
```

# Recap

A Convolutional Neural Network (CNN) is a specialized type of neural network architecture predominantly used for image classification tasks.

### The typical structure of a CNNs

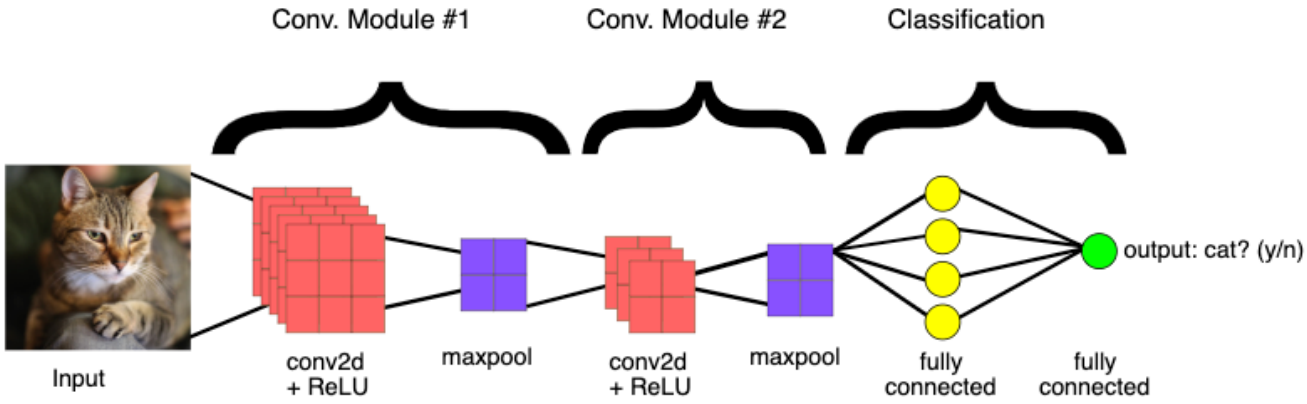
- Several convolutional and pooling layers:
  - **Convolutional layers** (referred to as Conv2d layers in PyTorch) use filters or kernels to capture spatial hierarchies and patterns within the data.
  - **Activation functions** are employed to introduce non-linearities into the model.
  - **Pooling layers** are used to reduce the spatial dimensions of the feature maps, helping in decreasing the complexity of the model.
  - **Dropout layers (optional)** can be incorporated for regularization in order to prevent overfitting.
- Followed by a fully connected (dense) layer that leads to the output.
  - For binary classification: Use Sigmoid activation if using `BCELoss`

### Loss function

- Multiclass classification: `CrossEntropyLoss`
- Binary classification: `BCEWithLogitsLoss` loss or use Sigmoid if using `BCELoss`
- For regression: No activation and MSELoss

### Parameters of the model and parameter sharing

- The kernels (or filters) used in convolutional layers, along with their corresponding biases, and the weights (and biases) in the fully connected layers after flattening, constitute the parameters of the CNN model.
- CNNs use of the same convolutional filters across the entire input, allowing the network to detect features regardless of their spatial location and significantly reducing the number of parameters. This is known as **parameter sharing**.




Source

# 1. Datasets, Dataloaders, and Transforms

## 1.1 Preparing Data

- `torch` and `torchvision` provide out-of-the-box functionality for loading in lots of different kinds of data.
- The way you create a dataloader depends on the data you have (i.e., do you have numpy arrays, tensors, images, or something else?) and the PyTorch docs can help you out.
- Loading data into PyTorch is usually a two-step process:
  1. Create a `dataset` (this is your raw data)
  2. Create a `dataloader` (this will help you batch your data)
- Working with CNNs and images, you'll mostly be using `torchvision.datasets.ImageFolder()` (docs).

- For example, consider the training dataset I have in the current directory at `lectures/data/animal_faces`:

```
├── train
│   ├── cat
│   │   ├── flickr_cat_000002.jpg
│   │   ├── flickr_cat_000003.jpg
│   │   ├── flickr_cat_000004.jpg
│   │   ├── flickr_cat_000005.jpg
│   │   ├── ...
│   ├── dog
│   │   ├── flickr_dog_000002.jpg
│   │   ├── flickr_dog_000003.jpg
│   │   ├── flickr_dog_000004.jpg
│   │   ├── flickr_dog_000005.jpg
│   │   ├── ...
│   └── wild
│       ├── flickr_wild_000002.jpg
│       ├── flickr_wild_000003.jpg
│       ├── flickr_wild_000005.jpg
│       ├── flickr_wild_000006.jpg
│       ├── ...
└── valid
│   ├── cat
│   │   ├── flickr_cat_000008.jpg
│   │   ├── flickr_cat_000011.jpg
│   │   ├── ...
│   ├── dog
│   │   ├── flickr_dog_000043.jpg
│   │   ├── flickr_dog_000045.jpg
│   │   ├── ...
│   └── wild
│       ├── flickr_wild_000004.jpg
│       ├── flickr_wild_000012.jpg
│       ├── ...
```

```
TRAIN_DIR = DATA_DIR + "animal_faces/train/"

mem = memory_profiler.memory_usage()[0]

train_dataset = torchvision.datasets.ImageFolder(root=TRAIN_DIR)

print(f"Memory consumed: {memory_profiler.memory_usage()[0] - mem:.0f} MB")
```

```
Memory consumed: 0 MB
```

- Notice how our memory usage is the same, we aren't loading anything in yet, just making PyTorch aware of what kind of data we have and where it is.

- We can now check various information about our `train_dataset`:

```
print(f"Classes: {train_dataset.classes}")
print(f"Class count: {train_dataset.targets.count(0)}, {train_dataset.targets.
print(f"Samples:",len(train_dataset))
print(f"First sample: {train_dataset.samples[0]}")
```

```
Classes: ['cat', 'dog', 'wild']
Class count: 50, 50
Samples: 150
First sample: ('/Users/kvarada/MDS/2024-25/572/DSCI_572_sup-learn-2_students/l
```

- Now, we could start working with this dataset directly
- For example, here's the first sample:

```
gen = iter(train_dataset)
```

```
img, target = next(gen)
print(f"Class: {train_dataset.classes[target]}")
img
```

```
Class: cat
```

- But often we want to apply some **pre-processing** to our data
- For example, `ImageFolder` loads our data using the `PIL` package, but we need tensors.

```
print(f"Image data type: {type(img)}")
print(f"     Image size: {img.size}")
```

```
Image data type: <class 'PIL.Image.Image'>
     Image size: (512, 512)
```

- Any pre-processing we wish to apply to our images is done using `torchvision.transforms`
- There are a lot of transformation options here, but we'll explore some more later. For now, we'll `Resize()` our images and convert them `ToTensor()`

- We use `transforms.Compose()` to chain multiple transformations together:

```
IMAGE_SIZE = (64, 64)

data_transforms = transforms.Compose([
    transforms.Resize(IMAGE_SIZE),
    transforms.ToTensor()
])

train_dataset = torchvision.datasets.ImageFolder(root=TRAIN_DIR, transform=dat

img, target = next(iter(train_dataset))
print(f"Image data type: {type(img)}")
print(f"     Image size: {img.shape}")
```

```
Image data type: <class 'torch.Tensor'>
     Image size: torch.Size([3, 64, 64])
```

- Great, but there's one more issue: we want to work with **batches** of data, because most of the time, we won't be able to fit an entire dataset into RAM at once (especially when it comes to image data)
- This is where PyTorch's `dataloader` comes in
- It allows us to specify how we want to batch our data:

```
BATCH_SIZE = 64

mem = memory_profiler.memory_usage()[0]

train_loader = torch.utils.data.DataLoader(
    train_dataset,          # our raw data
    batch_size=BATCH_SIZE,  # the size of batches we want the dataloader to re
    shuffle=True,           # shuffle our data before batching
    drop_last=False         # don't drop the last batch even if it's smaller t
)

print(f"Memory consumed: {memory_profiler.memory_usage()[0] - mem:.0f} MB")
```

```
Memory consumed: 0 MB
```

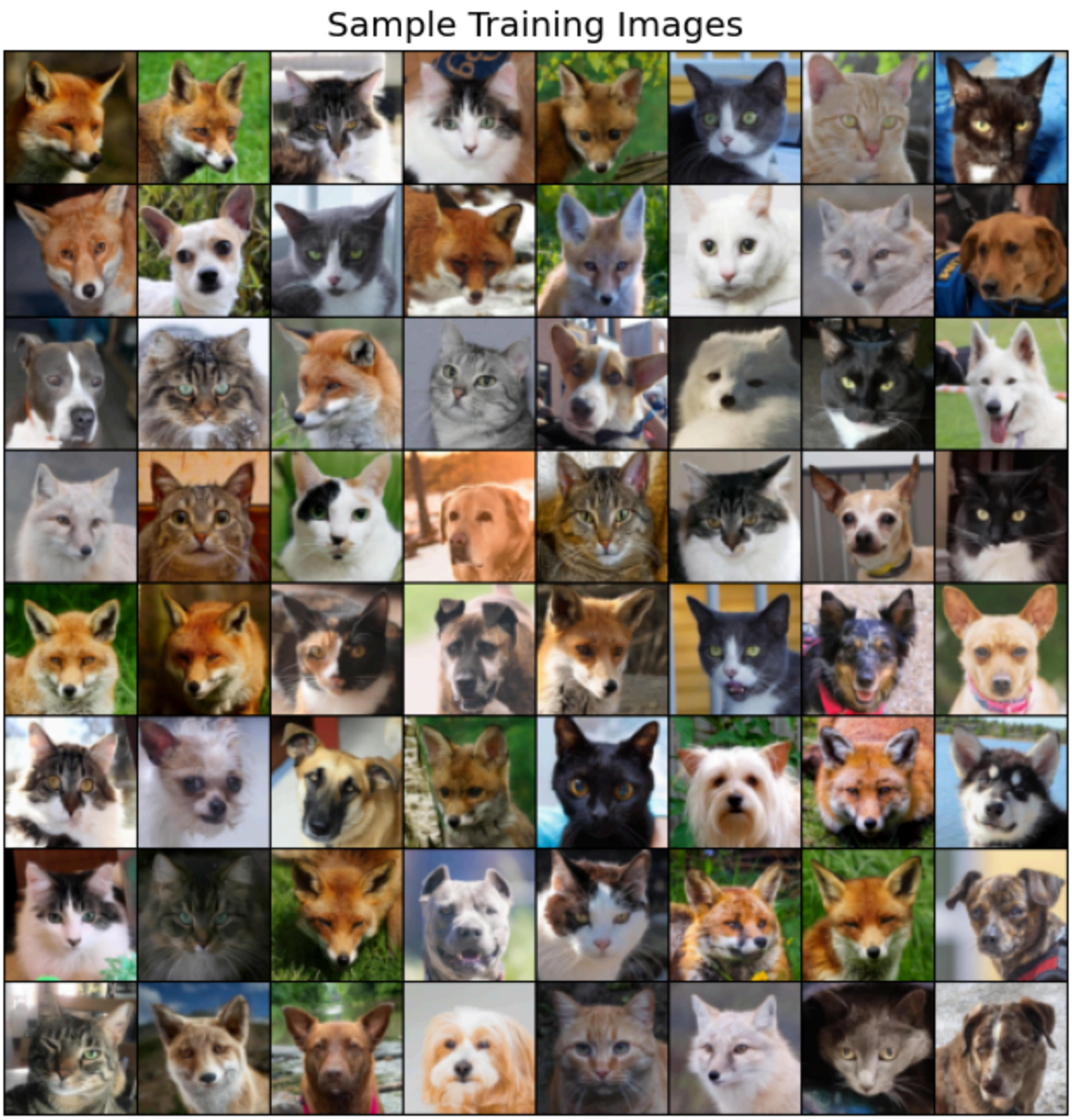
- Once again, we aren't loading anything yet, we just prepared the loader

- We can now query the loader to return a batch of data (this will consume memory):

```
mem = memory_profiler.memory_usage()[0]

imgs, targets = next(iter(train_loader))

print(f"       # of batches: {len(train_loader)}")
print(f"    Image data type: {type(imgs)}")
print(f"   Image batch size: {imgs.shape}")  # dimensions are (batch size, ima
print(f"  Target batch size: {targets.shape}")
print(f"       Batch memory: {memory_profiler.memory_usage()[0] - mem:.2f} MB"
```

```
      # of batches: 3
    Image data type: <class 'torch.Tensor'>
   Image batch size: torch.Size([64, 3, 64, 64])
  Target batch size: torch.Size([64])
       Batch memory: 16.59 MB
```

```
# Plot samples
sample_batch = next(iter(train_loader))
plt.figure(figsize=(10, 8)); plt.axis("off"); plt.title("Sample Training Image
plt.imshow(np.transpose(torchvision.utils.make_grid(sample_batch[0], padding=1
```




## 1.2 Saving and Loading PyTorch Models

- After training a model for several epochs and investing significant computational resources, it's important to save the model so you don't have to retrain it every time you want to make predictions. (Making predictions is often referred to as "inference".)

- Fortunately, PyTorch makes it easy to save and load models. You can refer to the official PyTorch documentation for detailed guidance.
- By convention, PyTorch models are typically saved with `.pt` or `.pth` file extensions.
- Depending on your needs, you may choose to save either the entire model or just the model parameters:
  - **Saving model parameters:** Use `model.state_dict()` to save only the model's weights and biases.
  - **Checkpointing during training:** For long-running training, it's useful to save more than just the model's `state_dict`. Checkpoints typically include the model's `state_dict`, the optimizer's `state_dict`, and other variables like the current epoch number, allowing you to resume training seamlessly.
  - **Saving the entire model:** Use `torch.save(model, filepath)` to serialize the entire model object, including its class definition, hyperparameters, and `state_dict`, using Python's `pickle` module.
  - Check out the documentation for more details.

```
# Example: Save model
PATH = "models/model.pt"
torch.save(model.state_dict(), PATH)     # save model at PATH

# Example: Load model
model = MyModelClass()                   # create an instance of the model
model.load_state_dict(torch.load(PATH))  # load model from PATH
```

- If you're using the model for **inference** (not training), make sure to switch to eval mode: `model.eval()`

# 1.3 Data Augmentation

Data augmentation serves two main purposes:

- to make your CNN more robust to variations like scaling or rotation, and
- to increase the diversity of your training data, either by generating variations dynamically or expanding the dataset size explicitly.

- By applying data augmentation, we can expose the CNN to variations in the images, like rotations and flips, enabling it to learn and predict more effectively under these conditions. This helps the model generalize better to unseen data.
- Common image augmentations include:
  - rotation/flipping
  - cropping
  - adding noise
  - You can view others in the PyTorch docs
- You can apply data augmentation using the transform argument in `torchvision.datasets.ImageFolder()`, just like we did earlier.
- Adding variation to the training data is generally a good practice to improve the robustness of your model.
- **Check out Appendix D for an example of data augmentation.**

## 1.4 (Optional) Batch Normalization

- Earlier in the course, we saw how normalizing the inputs to our neural network can help our optimization (by making sure the scale of one feature doesn't overwhelm others)
- But what about the hidden layers of our network? They also have data flowing into them and parameters to optimize, can we normalize them too to make optimization better?
- Batch normalization is the normalization of data in hidden layers
- It is usually applied before (and sometimes after) the activation function of a hidden layer:

$$z^* = \frac{z - \mu}{\sqrt{\sigma^2 + \eta}} \times \gamma + \beta$$

- Where:
  - $z$ = the output of your hidden layers before/after the activation function
  - $\mu = \frac{1}{n}\sum_{i=1}^{n} z_i$ (i.e., the mean of $z$)
  - $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(z_i - \mu)^2$ (i.e, the variance of $z$)
  - $\gamma$ and $\beta$ are parameters that are learned through backpropagation
- Batch normalization can help stabilize and speed up optimization, make your network more invariant to changes in the training distribution, and often has a slight regularization effect
- Even though we have some guiding intuitions about the mechanics of batch normalization, we still don't have a widely accepted theory for how *exactly* this technique works the way it does
  - "*Somewhat shockingly, however, despite its prominence, we still have a poor understanding of what the effectiveness of BatchNorm is stemming from.*" (Santurkar et al., 2018)
  - Read more about this issue here

# 2. Hyperparameter Tuning

- With neural networks we potentially have a lot of hyperparameters to tune:
  - Number of layers
  - Number of nodes in each layer
  - Activation functions
  - Regularization
  - Initialization (starting weights)
  - Optimization hyperparameters (learning rate, momentum, weight decay)
  - etc.
- When the number of hyperparameters are large and models are expensive to compute, brute force strategies such as **grid search** perform poorly. In these cases, we need to

make our approach smarter. For example, we can use an optimization algorithm instead of blindly evaluating our model for all values of hyperparameters.

- There are many packages out there that make neural network hyperparameter tuning fast and easy using such approaches
  - Ax
  - Raytune
  - Neptune
  - Optuna
- We'll be using Ax, created by Facebook (just like PyTorch)
- Ax uses Bayesian optimization to tune the hyperparameters of our model (how? see here)

```
# !conda install -y ax-platform
```

- Below, I've adapted a tutorial from their docs:

```
from ax.service.managed_loop import optimize
from ax.plot.contour import plot_contour
from ax.plot.trace import optimization_trace_single_method
from ax.utils.notebook.plotting import render, init_notebook_plotting
from ax.utils.tutorials.cnn_utils import train, evaluate
```

### Data

In this section, we'll work with the popular CIFAR-10 dataset. As noted in Appendix D, we achieved a validation accuracy of around 0.70 on this dataset without any hyperparameter optimization. Now, let's explore how we can improve this performance by tuning the hyperparameters.

### Data loaders

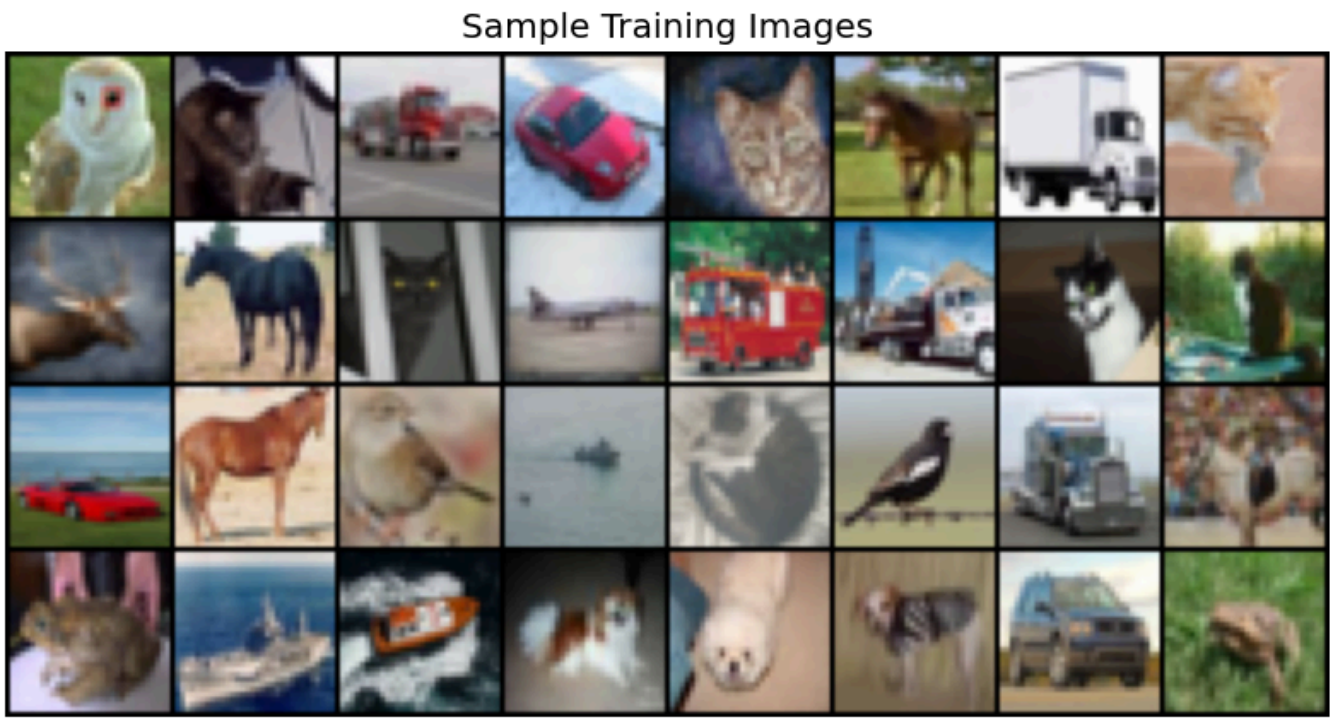
```

```

```
from torchvision import datasets, transforms

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

train_dataset = datasets.CIFAR10(root='./data', train=True, transform=transfor
valid_dataset = datasets.CIFAR10(root='./data', train=False, transform=transfo

IMAGE_SIZE = 32
BATCH_SIZE = 32
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE
validloader = torch.utils.data.DataLoader(valid_dataset, batch_size=BATCH_SIZE
```

```
Files already downloaded and verified
```

```
Files already downloaded and verified
```

```
# Plot samples
sample_batch = next(iter(trainloader))
plt.figure(figsize=(10, 8)); plt.axis("off"); plt.title("Sample Training Image
plt.imshow(np.transpose(torchvision.utils.make_grid(sample_batch[0], padding=1
```



```
class CIFAR10_CNN(nn.Module):
    def __init__(self):
        super(CIFAR10_CNN, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        return self.model(x)
```

```
# Instantiate the model
cifar10_model = CIFAR10_CNN().to(device)
summary(cifar10_model, (3, 32, 32));
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 10]                  --
|    └─Conv2d: 2-1                       [-1, 32, 32, 32]          896
|    └─ReLU: 2-2                         [-1, 32, 32, 32]          --
|    └─MaxPool2d: 2-3                    [-1, 32, 16, 16]          --
|    └─Conv2d: 2-4                       [-1, 64, 16, 16]          18,496
|    └─ReLU: 2-5                         [-1, 64, 16, 16]          --
|    └─MaxPool2d: 2-6                    [-1, 64, 8, 8]            --
|    └─Conv2d: 2-7                       [-1, 128, 8, 8]           73,856
|    └─ReLU: 2-8                         [-1, 128, 8, 8]           --
|    └─MaxPool2d: 2-9                    [-1, 128, 4, 4]           --
|    └─Flatten: 2-10                     [-1, 2048]                --
|    └─Linear: 2-11                      [-1, 512]                 1,049,088
|    └─ReLU: 2-12                        [-1, 512]                 --
|    └─Linear: 2-13                      [-1, 10]                  5,130
==========================================================================================
Total params: 1,147,466
Trainable params: 1,147,466
Non-trainable params: 0
Total mult-adds (M): 12.52
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.44
Params size (MB): 4.38
Estimated Total Size (MB): 4.83
==========================================================================================
```

### Training loop

- Now, we need a training function
- This function will be re-run multiple times throughout the hyperparameter optimization process, as we wish to train the model on different hyperparameter configurations
- The argument `hyperparameters` is a dictionary containing the hyperparameters we wish to tune:

```
def train(model, trainloader, hyperparameters, epochs=10):
    """Training wrapper for PyTorch network."""

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(),
                           lr=hyperparameters.get("lr", 0.001),
                           betas=(hyperparameters.get("beta1", 0.9), 0.999))
    model.train()
    for epoch in range(epochs):
        for X, y in trainloader:
            if device.type in ['cuda', 'mps']:
                X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            y_hat = model(X)
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()

    return model
```

### Evaluation

- We also need an `evaluate()` function that reports how well our model is doing on some validation data
- This will also be called multiple times during the hyperparameter optimization
- We have already defined such a function before in this lecture, but here is its definition again:

```
def evaluate(model, validloader):
    """Validation wrapper for PyTorch network."""

    model.eval()
    accuracy = 0

    with torch.no_grad():  # this stops pytorch doing computational graph stuf
        for X, y in validloader:
            if device.type in ['cuda', 'mps']:
                X, y = X.to(device), y.to(device)
            y_hat = model(X)
            _, y_hat_labels = torch.softmax(y_hat, dim=1).topk(1, dim=1)
            accuracy += (y_hat_labels.squeeze() == y).type(torch.float32).mean
    accuracy /= len(validloader)  # avg accuracy

    return accuracy
```

- Let's make sure our evaluation function is working:

```
model = CIFAR10_CNN().to(device)
evaluate(model, validloader)
```

```
0.08027156549520767
```

- The accuracy is bad right now because we haven't trained our model yet
- We then have a wrapper function that puts everything together
- Basically each iteration of hyperparameter optimization (i.e., each time we try a new set of hyperparameters), this function is executed. It trains the model using the given hyperparameters, and then evaluates the model's performance.

```
def train_evaluate(parameterization):
    model = CIFAR10_CNN().to(device)
    model = train(model, trainloader, hyperparameters=parameterization)
    return evaluate(model, validloader)
```

- Finally, we use `optimize()` to run Bayesian optimization on a hyperparameter dictionary
- I ran this on a GPU and have included the output below:

```
# best_parameters, values, experiment, model = optimize(
#     parameters=[
#         {"name": "lr", "type": "range", "bounds": [1e-6, 0.4], "log_scale": 
#         {"name": "beta1", "type": "range", "bounds": [0.2, 0.999], "value_ty
#     ],
#     evaluation_function=train_evaluate,
#     objective_name='accuracy',
#     total_trials = 8
# )
```

```
best_parameters = {'lr': 0.00031221646054819564, 'beta1': 0.9989999999999999}
```

```
best_parameters.get("beta1")
```

```
0.9989999999999999
```

```

```

```
# Use the best parameters from Ax to train the final model

# best_model = train(model, trainloader, best_parameters)
```

```
# Save model
PATH = "../models/optimized_CIFAR10-cnn.pt"
# torch.save(best_model, PATH)
```

```
# Load saved model

best_model = torch.load(PATH)
```

```
/var/folders/b3/g26r0dcx4b35vf3nk31216hc0000gr/T/ipykernel_74836/3311455064.py

You are using `torch.load` with `weights_only=False` (the current default valu
```

```
evaluate(best_model.to('mps'), validloader)
```

```
0.7478035143769968
```

```
# render(plot_contour(model=model, param_x='lr', param_y='beta1', metric_name=
```

`Ax` is highly flexible and can optimize any hyperparameter, including network architecture-related parameters (e.g., number of Conv2D layers, number of output channels in the first Conv2D layer, kernel size for the first Conv2D layer). However, when optimizing architecture-related parameters, you must dynamically build the model in your train and evaluate function based on the parameter configurations suggested by `Ax`. This dynamic model-building process can become computationally intensive, especially for large-scale networks or datasets.

# 3. (Optional) Explaining CNNs

- CNNs and neural networks in general are primarily used for **prediction** (i.e., we want the best prediction performance, and we might not care how we get it)
- However interpreting why a model makes certain predictions can be useful
- Interpreting neural networks is an active area of research and it is difficult to do
- There are a few main options:
  - SHAP
  - Grad-CAM
  - Captum
- Captum is a library for specifically interpreting PyTorch models. It's quite new still but has some great functionality!
- Captum contains a variety of state-of-the-art algorithms for interpreting model predictions, see the docs here

```
# !conda install -y captum
```

# 4. Transfer Learning

- In practice, very few people train an entire CNN from scratch because it requires a large dataset, powerful computers, and a huge amount of human effort to train the model.
- Instead, a common practice is to download a pre-trained model and fine tune it for your task. This is called **transfer learning**.
- Transfer learning is one of the most common techniques used in the context of computer vision and natural language processing.
- It refers to using a model already trained on one task as a starting point for learning to perform another task
- There are many famous deep learning architectures out there that have been very successful across a wide range of problems, e.g.: AlexNet, VGG, ResNet, Inception, MobileNet, etc.

- Many of these models have been pre-trained on famous datasets like ImageNet [1, 2]

## ImageNet

- ImageNet is an image dataset that became a very popular benchmark in the field ~10 years ago.
- Currently contains ~14 million labelled images with ~21,841 categories
- There are various versions with different number of images and classes
  - ILSVRC, a popular annual competition in computer vision, uses a smaller subset of ImageNet. This subset consists of about 1.2 million training images, 50,000 validation images, and 150,000 testing images across 1,000 categories.
- Wikipedia article on ImageNet
- Here are some example classes.

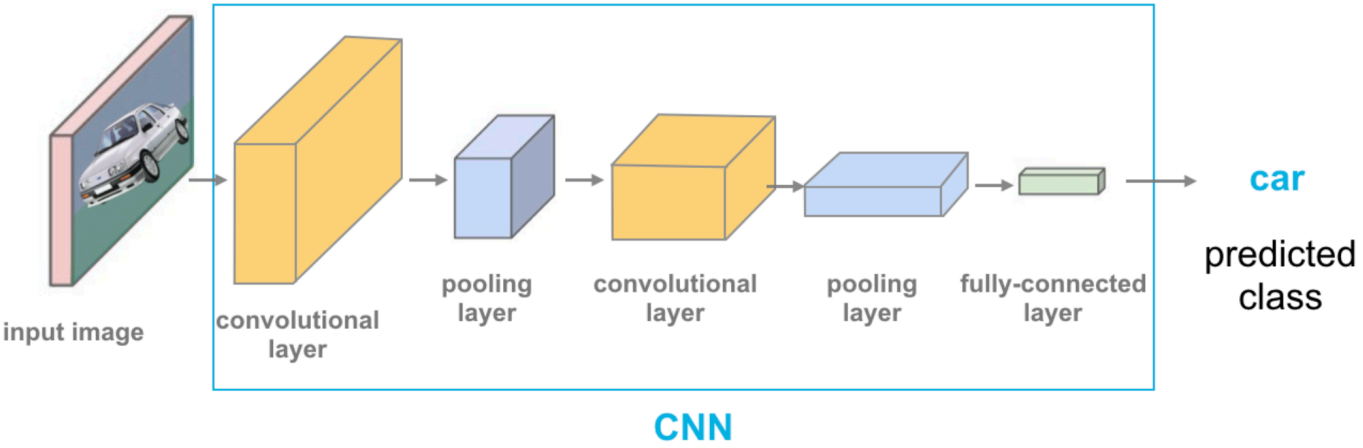
```
with open(DATA_DIR + "imagenet_classes.txt") as f:
    classes = [line.strip() for line in f.readlines()]
classes[100:110]
```

```
['black swan, Cygnus atratus',
 'tusker',
 'echidna, spiny anteater, anteater',
 'platypus, duckbill, duckbilled platypus, duck-billed platypus, Ornithorhynchu
 'wallaby, brush kangaroo',
 'koala, koala bear, kangaroo bear, native bear, Phascolarctos cinereus',
 'wombat',
 'jellyfish',
 'sea anemone, anemone',
 'brain coral']
```

- The idea of transfer learning is instead of developing a machine learning model from scratch, you use these available pre-trained models for your tasks either directly or by fine tuning them.
- There are three common ways to use transfer learning in computer vision
  1. Using pre-trained models out-of-the-box
  2. Using pre-trained models as feature extractor and training your own model with these features
  3. Starting with weights of pre-trained models and fine-tuning the weights for your task.
- We will explore the first two approaches.

# 4.1 Using pre-trained models out-of-the-box

- Let's first try one of these models and apply it to our own problem right out of the box



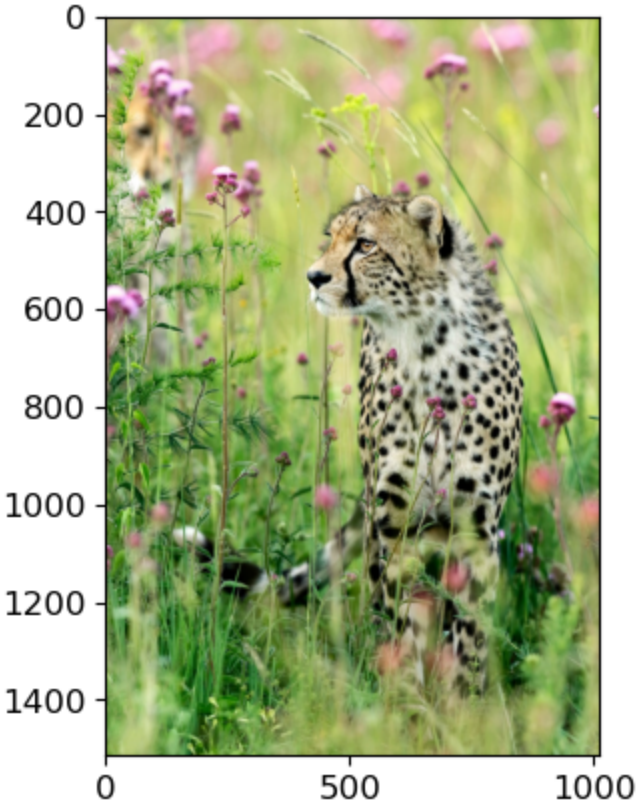

Source: https://cezannec.github.io/Convolutional_Neural_Networks/

- We can easily download famous models using the `torchvision.models` module. All models are available with pre-trained weights (based on ImageNet's 224 x 224 images)
- Remember this example I showed you in the intro video of DSCI 571?
  - We used a pre-trained model vgg16 which is trained on the ImageNet data.
  - We preprocess the given image.
  - We get prediction from this pre-trained model on a given image along with prediction probabilities.
  - For a given image, this model will spit out one of the 1000 classes from ImageNet.

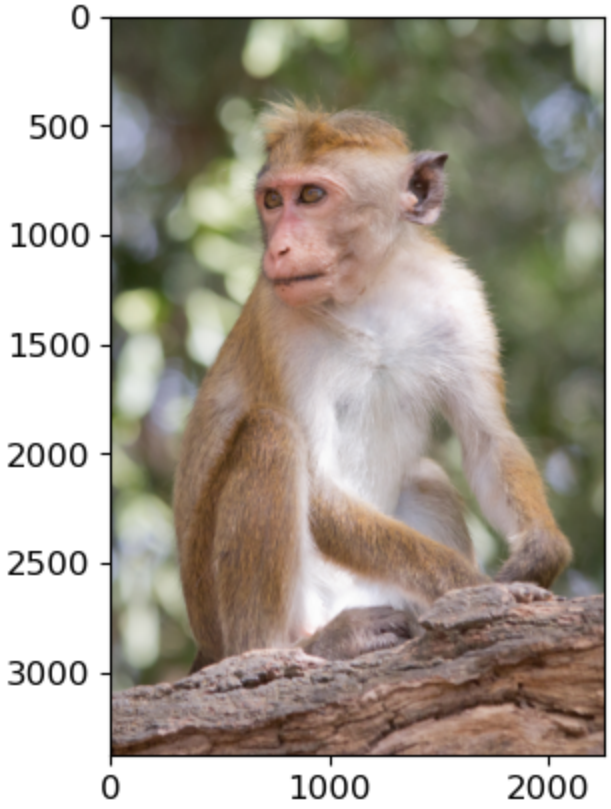
```
import torch
from PIL import Image
from torchvision import transforms
from torchvision.models import vgg16

# Load the VGG16 model pre-trained on ImageNet data
vgg16 = models.vgg16(weights='DEFAULT')
vgg16.eval()  # Set the model to evaluation mode

# Define a function to classify an image
def classify_image(image_path, class_labels_file, topn=4):
    # Load and preprocess the image
    img = Image.open(image_path)
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
    ])
    img = preprocess(img).unsqueeze(0)  # Add a batch dimension

    # Perform inference
    with torch.no_grad():
        output = vgg16(img)

    # Load the class labels from the text file
    with open(class_labels_file, 'r') as f:
        class_labels = [line.strip() for line in f.readlines()]

    # Get class probabilities
    _, indices = torch.sort(output, descending=True)
    probabilities = torch.nn.functional.softmax(output, dim=1)

    # Create a dataframe with topn class labels and their corresponding probab
    d = {'Class': [class_labels[idx] for idx in indices[0][:topn]],
         'Probability score': [np.round(probabilities[0, idx].item(),3) for id
    df = pd.DataFrame(d, columns = ['Class','Probability score'])
    return df
```

```
import glob

import matplotlib.pyplot as plt
from PIL import Image

# Predict topn labels and their associated probabilities for unseen images
images = glob.glob(DATA_DIR + "test_images/*.*")
class_labels_file = DATA_DIR + "/imagenet_classes.txt"
for img_path in images:
    img = Image.open(img_path).convert("RGB")
    img.load()
    plt.imshow(img)
    plt.show()
    df = classify_image(img_path, class_labels_file)
    print(df.to_string(index=False))
    print("------------------------------------------------------------")
```

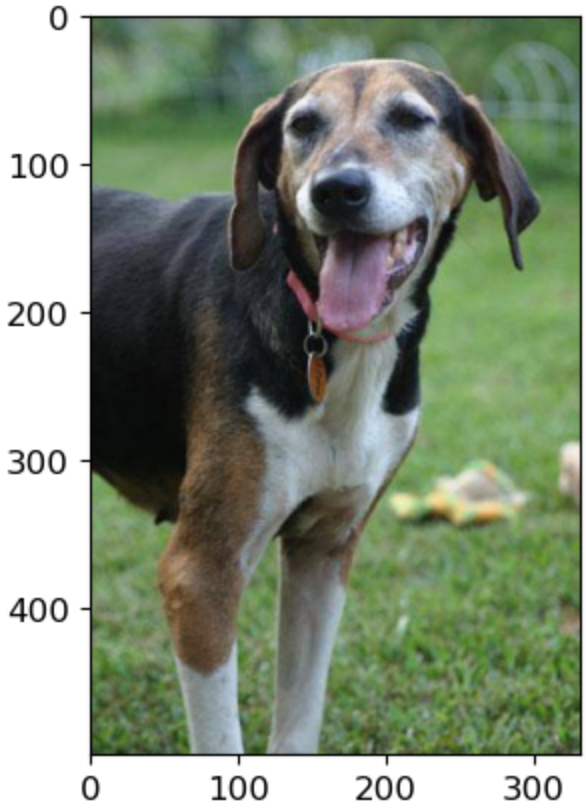

```
                      Class  Probability score
                  tiger cat              0.636
              tabby, tabby cat              0.174
Pembroke, Pembroke Welsh corgi              0.081
               lynx, catamount              0.011
------------------------------------------------------------
```




```
                                    Class  Probability score
         cheetah, chetah, Acinonyx jubatus                0.994
                  leopard, Panthera pardus                0.005
jaguar, panther, Panthera onca, Felis onca                0.001
       snow leopard, ounce, Panthera uncia                0.000
------------------------------------------------------------------
```




| Class | Probability score |
|---|---|
| macaque | 0.885 |
| patas, hussar monkey, Erythrocebus patas | 0.062 |
| proboscis monkey, Nasalis larvatus | 0.015 |
| titi, titi monkey | 0.010 |

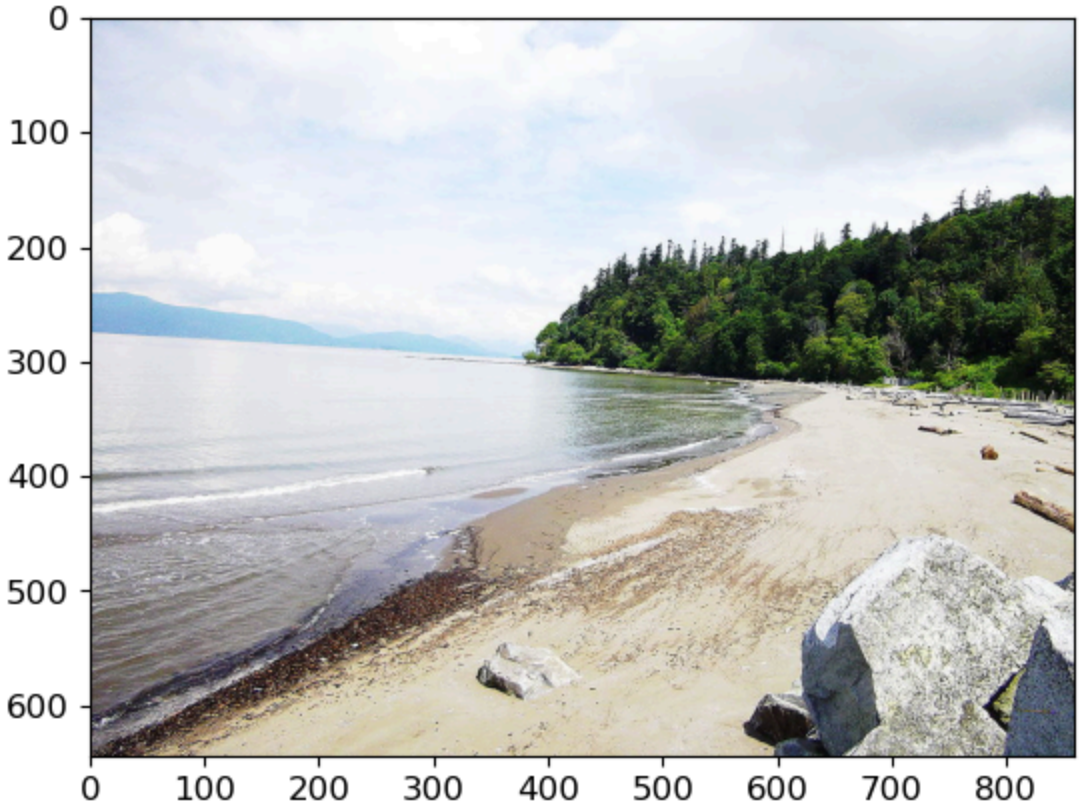



```
                         Class  Probability score
Walker hound, Walker foxhound                0.582
             English foxhound                0.144
                       beagle                0.068
                   EntleBucher               0.059
------------------------------------------------------------
```

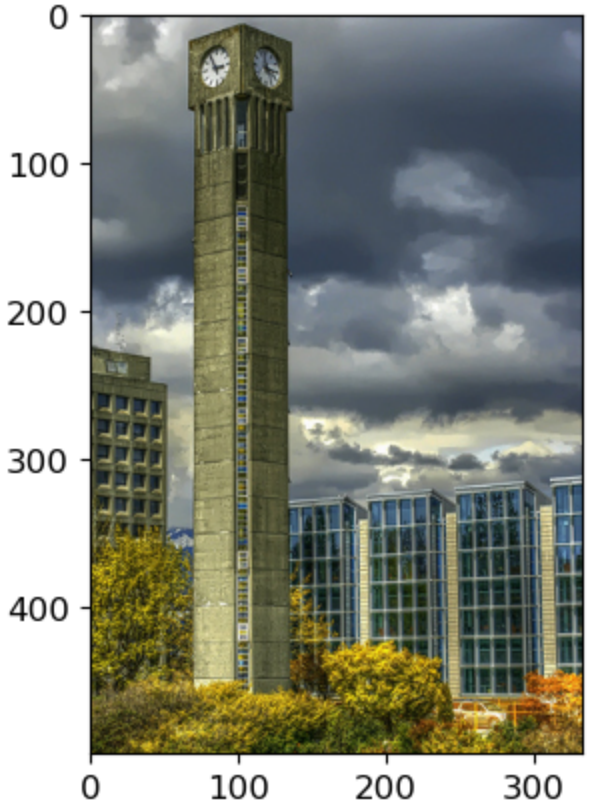
- We got these predictions without "doing the ML ourselves".
- We are using **pre-trained** `vgg16` model which is available in `torchvision`.
- `torchvision` has many such pre-trained models available that have been very successful across a wide range of tasks: AlexNet, VGG, ResNet, Inception, MobileNet, etc.
- Many of these models have been pre-trained on famous datasets like **ImageNet**.
- So if we use them out-of-the-box, they will give us one of the ImageNet classes as classification.

Let's try it out on some very different images from the training set.

```
images = glob.glob(DATA_DIR + "UBC_img/*.*")

for img_path in images:
    img = Image.open(img_path).convert("RGB")
    img.load()
    plt.imshow(img)
    plt.show()
    df = classify_image(img_path, class_labels_file)
    print(df.to_string(index=False))
    print("-----------------------------------------------------------------")
```



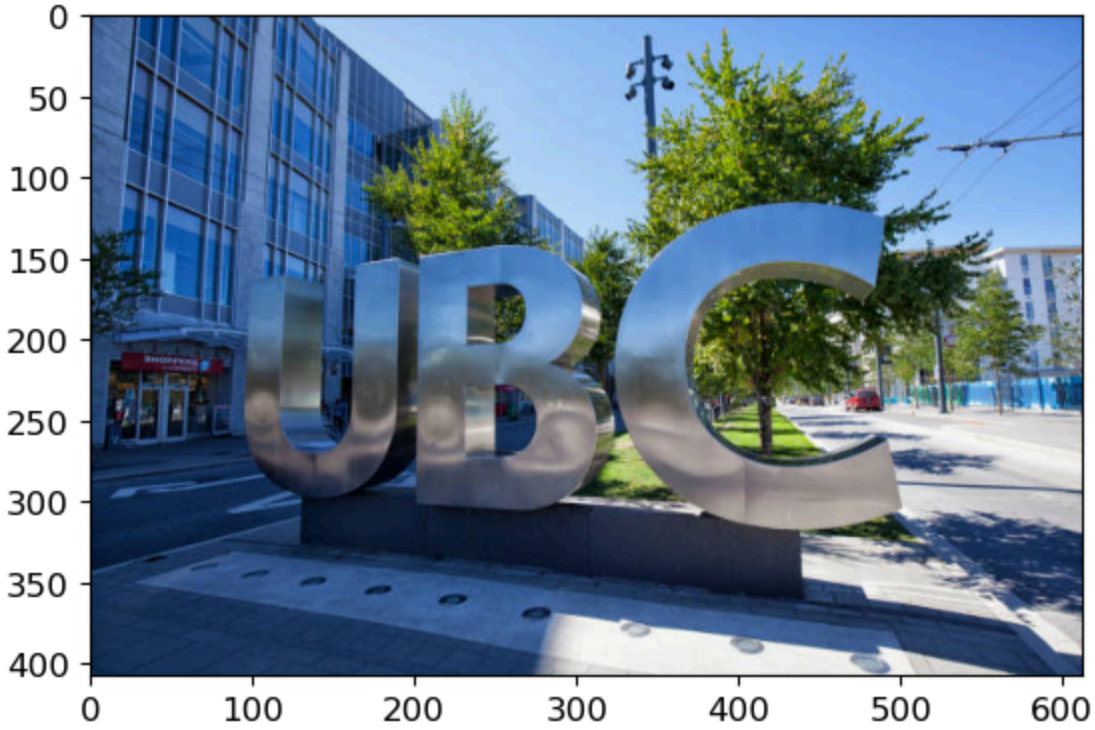

```
                                                Class  Probability score
                                     sandbar, sand bar              0.681
                   seashore, coast, seacoast, sea-coast              0.101
                                   lakeside, lakeshore              0.058
breakwater, groin, groyne, mole, bulwark, seawall, jetty              0.053
-----------------------------------------------------------------
```




```
                                Class  Probability score
                             obelisk                0.598
          pedestal, plinth, footstall                0.071
                   bell cote, bell cot                0.042
beacon, lighthouse, beacon light, pharos                0.022
-------------------------------------------------------------
```

```
                                    Class  Probability score
                                      fig              0.590
                                pomegranate              0.300
grocery store, grocery, food market, market              0.027
                                   banana              0.018
------------------------------------------------------------
```


```
                      Class  Probability score
                totem pole              0.995
                       pole              0.005
pedestal, plinth, footstall              0.000
                cowboy boot              0.000
------------------------------------------------------------
```




```
                                  Class  Probability score
                                   wing              0.084
                             toilet seat             0.081
                              car mirror             0.069
binoculars, field glasses, opera glasses             0.062
------------------------------------------------------------
```

```
              Class  Probability score
            sundial              0.219
           fountain              0.179
     patio, terrace              0.138
lakeside, lakeshore              0.131
-----------------------------------------------------------
```

- It's not doing very well here because ImageNet doesn't have proper classes for these images.
- Here we are using pre-trained models out-of-the-box.
- Can we use pre-trained models for our own classification problem with our classes?
- Yes!! There are two ways to benefit from transfer learning:
    1. Use a pre-trained network as a "**feature extractor**" and add new layers to it for your own task
    2. Same as the previous one, but also "**fine-tune**" the weights of the pre-trained network using your own data

## 4.2 Using pre-trained models as feature extractors

- In this method, we use a pre-trained model as a "feature extractor" which creates useful features for us that we can use to train some other model
- We have two options here:
    1. Add some extra layers to the pre-trained network to suit our particular task
    2. Pass training data through the network and save the output to use as features for training some other model

# 4.2.1 Approach 1: Adding extra layers to the pre-trained network

- Let's do approach 1 first. Let's adapt `DenseNet` to predict classes in our animal faces dataset. I'm going to load the model, and then "freeze" all of its parameters (we don't want to update them)

**(Optional)**

- DenseNet paper
- This architecture connects each layer to every other layer in a feed-forward fashion.
- The dense connectivity ensures maximum information flow between layers in the network. This connectivity leads to significant improvements in efficiency and effectiveness in learning, as each layer has access to all preceding layers' feature maps.

For this part we'll use the animal faces dataset. For sanity, let's load the dataset again.

```
# Attribution: [Code from PyTorch docs](https://pytorch.org/tutorials/beginner

IMAGE_SIZE = 224
BATCH_SIZE = 32

data_transforms = {
    "train": transforms.Compose(
        [
            transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
        ]
    ),
    "valid": transforms.Compose(
        [
            transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
        ]
    ),
}
data_dir = DATA_DIR + "animal_faces"
image_datasets = {
    x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x])
    for x in ["train", "valid"]
}
dataloaders = {
    x: torch.utils.data.DataLoader(
        image_datasets[x], batch_size=BATCH_SIZE, shuffle=True, num_workers=4
    )
    for x in ["train", "valid"]
}
dataset_sizes = {x: len(image_datasets[x]) for x in ["train", "valid"]}
class_names = image_datasets["train"].classes

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

```
# Get a batch of training data
inputs, classes = next(iter(dataloaders["train"]))
```

```
plt.figure(figsize=(10, 8)); plt.axis("off"); plt.title("Sample Training Image
plt.imshow(np.transpose(torchvision.utils.make_grid(inputs, padding=1, normali
```


```
densenet = models.densenet121(weights='DenseNet121_Weights.DEFAULT')

for param in densenet.parameters():  # Freeze parameters so we don't update th
    param.requires_grad = False
```

- The layers can be accessed using the `.named_children()` method, the last one is the classification layer, a fully-connected layer outputting 1000 values (one for each ImageNet class):

```
list(densenet.named_children())[-1]
```

```
('classifier', Linear(in_features=1024, out_features=1000, bias=True))
```

```
densenet.classifier
```

```
Linear(in_features=1024, out_features=1000, bias=True)
```

- We are going to do 3-class classification, so let's replace this layer with our own layers:

```

```

```
new_layers = nn.Sequential(
    nn.Linear(1024, 500),
    nn.ReLU(),
    nn.Linear(500, 3)
)
densenet.classifier = new_layers
```

- Let's check that the last layer of our model is updated:

```
densenet.classifier
```

```
Sequential(
  (0): Linear(in_features=1024, out_features=500, bias=True)
  (1): ReLU()
  (2): Linear(in_features=500, out_features=3, bias=True)
)
```

- Now we need to train our new layers:

```
def trainer(model, criterion, optimizer, trainloader, validloader, device, epo
    """Simple training wrapper for PyTorch network."""

    train_loss, valid_loss, valid_accuracy = [], [], []
    for epoch in range(epochs):  # for each epoch
        train_batch_loss = 0
        valid_batch_loss = 0
        valid_batch_acc = 0

        # Training
        model.train()
        for X, y in trainloader:
            if device.type in ['cuda', 'mps']:
                X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            y_hat = model(X)
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()
            train_batch_loss += loss.item()
        train_loss.append(train_batch_loss / len(trainloader))

        # Validation
        model.eval()

        with torch.no_grad():  # this stops pytorch doing computational graph
            for X, y in validloader:
                if device.type in ['cuda', 'mps']:
                    X, y = X.to(device), y.to(device)
                y_hat = model(X)
                _, y_hat_labels = torch.softmax(y_hat, dim=1).topk(1, dim=1)
                loss = criterion(y_hat, y)
                valid_batch_loss += loss.item()
                valid_batch_acc += (y_hat_labels.squeeze() == y).type(torch.fl
        valid_loss.append(valid_batch_loss / len(validloader))
        valid_accuracy.append(valid_batch_acc / len(validloader))  # accuracy



        # Print progress
        if verbose:
            print(f"Epoch {epoch + 1}:",
                  f"Train Loss: {train_loss[-1]:.3f}.",
                  f"Valid Loss: {valid_loss[-1]:.3f}.",
                  f"Valid Accuracy: {valid_accuracy[-1]:.2f}.")

    results = {"train_loss": train_loss,
               "valid_loss": valid_loss,
               "valid_accuracy": valid_accuracy}
    return results
```

```
# We have a big model so this will take some time to run! If you have a GPU, t
densenet.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(densenet.parameters(), lr=2e-3)
# results = trainer(densenet, criterion, optimizer, dataloaders['train'], data
```

```
Epoch 1: Train Loss: 1.456. Valid Loss: 0.756. Valid Accuracy: 0.61.
Epoch 2: Train Loss: 0.756. Valid Loss: 0.486. Valid Accuracy: 0.83.
Epoch 3: Train Loss: 0.289. Valid Loss: 0.354. Valid Accuracy: 0.93.
Epoch 4: Train Loss: 0.153. Valid Loss: 0.281. Valid Accuracy: 0.89.
Epoch 5: Train Loss: 0.090. Valid Loss: 0.324. Valid Accuracy: 0.85.
Epoch 6: Train Loss: 0.041. Valid Loss: 0.608. Valid Accuracy: 0.81.
Epoch 7: Train Loss: 0.054. Valid Loss: 0.576. Valid Accuracy: 0.85.
Epoch 8: Train Loss: 0.016. Valid Loss: 0.437. Valid Accuracy: 0.86.
Epoch 9: Train Loss: 0.031. Valid Loss: 0.350. Valid Accuracy: 0.87.
Epoch 10: Train Loss: 0.006. Valid Loss: 0.249. Valid Accuracy: 0.89.
```

- We leveraged the power of `DenseNet` to get a good model, **without training any convolutional layers ourselves**.

## 4.2.2 Approach 2: Using extracted features in other models

- Now, you can use pre-trained model as arbitrary feature extractors, you don't have to add on layers, you can just extract the output values of the network (well, you can extract values from any layer you like) and use those values as "features" to train another model
- Below, I'm going to pass all my data through the network and save the outputs:

```
def get_features(model, train_loader, valid_loader):
    """
    Extract features from both training and validation datasets using the prov

    This function passes data through a given neural network model to extract
    to work with datasets loaded using PyTorch's DataLoader. The function oper
    that gradients are not required, optimizing memory and computation for inf
    """

    # Disable gradient computation for efficiency during inference
    with torch.no_grad():
        # Initialize empty tensors for training features and labels
        Z_train = torch.empty((0, 1024))  # Assuming each feature vector has 1
        y_train = torch.empty((0))

        # Initialize empty tensors for validation features and labels
        Z_valid = torch.empty((0, 1024))
        y_valid = torch.empty((0))

        # Process training data
        for X, y in train_loader:
            # Extract features and concatenate them to the corresponding tenso
            Z_train = torch.cat((Z_train, model(X)), dim=0)
            y_train = torch.cat((y_train, y))

        # Process validation data
        for X, y in valid_loader:
            # Extract features and concatenate them to the corresponding tenso
            Z_valid = torch.cat((Z_valid, model(X)), dim=0)
            y_valid = torch.cat((y_valid, y))

    # Return the feature and label tensors
    return Z_train.detach().numpy(), y_train.detach().numpy(), Z_valid.detach(
```

```
densenet = models.densenet121(weights="DenseNet121_Weights.IMAGENET1K_V1")
for param in densenet.parameters():  # Freeze parameters so we don't update th
    param.requires_grad = False
densenet.classifier = nn.Identity()  # remove that last "classification" layer
Z_train, y_train, Z_valid, y_valid = get_features(
    densenet, dataloaders["train"], dataloaders["valid"]
)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[50], line 5
      3     param.requires_grad = False
      4 densenet.classifier = nn.Identity()  # remove that last "classificatio
----> 5 Z_train, y_train, Z_valid, y_valid = get_features(
      6     densenet, dataloaders["train"], dataloaders["valid"]
      7 )

Cell In[49], line 21, in get_features(model, train_loader, valid_loader)
     18 y_valid = torch.empty((0))
     20 # Process training data
---> 21 for X, y in train_loader:
     22     # Extract features and concatenate them to the corresponding tenso
     23     Z_train = torch.cat((Z_train, model(X)), dim=0)
     24     y_train = torch.cat((y_train, y))

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torch/utils/data/data
    698 if self._sampler_iter is None:
    699     # TODO(https://github.com/pytorch/pytorch/issues/76750)
    700     self._reset()  # type: ignore[call-arg]
--> 701 data = self._next_data()
    702 self._num_yielded += 1
    703 if (
    704     self._dataset_kind == _DatasetKind.Iterable
    705     and self._IterableDataset_len_called is not None
    706     and self._num_yielded > self._IterableDataset_len_called
    707 ):

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torch/utils/data/data
   1434 else:
   1435     # no valid `self._rcvd_idx` is found (i.e., didn't break)
   1436     if not self._persistent_workers:
-> 1437         self._shutdown_workers()
   1438     raise StopIteration
   1440 # Now `self._rcvd_idx` is the batch index we want to fetch
   1441
   1442 # Check if the next sample has already been generated

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torch/utils/data/data
   1563         self._mark_worker_as_unavailable(worker_id, shutdown=True)
   1564 for w in self._workers:
   1565     # We should be able to join here, but in case anything went
   1566     # wrong, we set a timeout and if the workers fail to join,
   1567     # they are killed in the `finally` block.
-> 1568     w.join(timeout=_utils.MP_STATUS_CHECK_INTERVAL)
   1569 for q in self._index_queues:
   1570     q.cancel_join_thread()

File ~/miniforge3/envs/jbook/lib/python3.12/multiprocessing/process.py:149, in
    147 assert self._parent_pid == os.getpid(), 'can only join a child process
    148 assert self._popen is not None, 'can only join a started process'
--> 149 res = self._popen.wait(timeout)
    150 if res is not None:
    151     _children.discard(self)
```

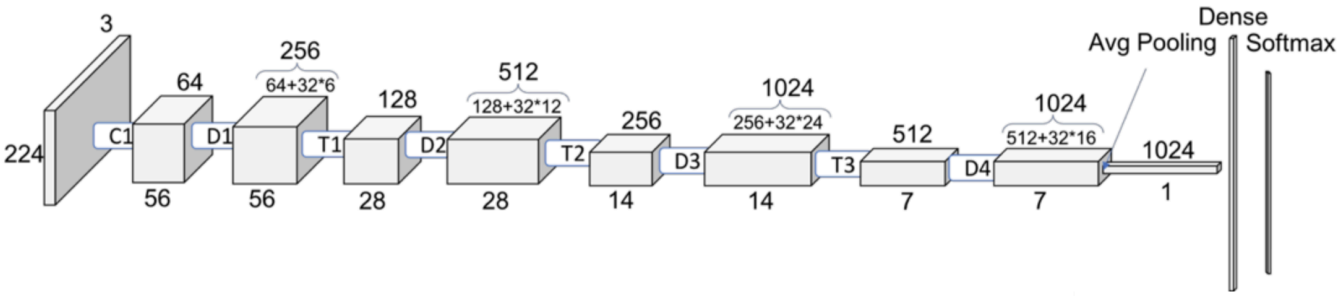
```
File ~/miniforge3/envs/jbook/lib/python3.12/multiprocessing/popen_fork.py:40, 
     38 if timeout is not None:
     39     from multiprocessing.connection import wait
---> 40     if not wait([self.sentinel], timeout):
     41         return None
     42 # This shouldn't block if wait() returned successfully.

File ~/miniforge3/envs/jbook/lib/python3.12/multiprocessing/connection.py:1136
   1133     deadline = time.monotonic() + timeout
   1135 while True:
-> 1136     ready = selector.select(timeout)
   1137     if ready:
   1138         return [key.fileobj for (key, events) in ready]

File ~/miniforge3/envs/jbook/lib/python3.12/selectors.py:415, in _PollLikeSele
    413 ready = []
    414 try:
--> 415     fd_event_list = self._selector.poll(timeout)
    416 except InterruptedError:
    417     return ready

KeyboardInterrupt:
```

Now we have extracted feature vectors for all examples. What's the shape of these features?

```
Z_train.shape
```

```
(150, 1024)
```

The size of each feature vector is 1024 because the size of the last layer in densenet architecture is 1024.




Source: https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a

Let's examine the feature vectors.

```
pd.DataFrame(Z_train)
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **0** | 0.000313 | 0.005374 | 0.002826 | 0.001279 | 0.122242 | 0.442731 | 0.000435 |
| **1** | 0.000241 | 0.003134 | 0.003507 | 0.002049 | 0.156063 | 1.389513 | 0.000719 |
| **2** | 0.000365 | 0.004651 | 0.003649 | 0.001773 | 0.135300 | 0.560930 | 0.000426 |
| **3** | 0.000427 | 0.004292 | 0.001752 | 0.002217 | 0.110419 | 0.751406 | 0.000813 |
| **4** | 0.000274 | 0.004262 | 0.002464 | 0.001161 | 0.096465 | 0.183448 | 0.000617 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **145** | 0.000315 | 0.004802 | 0.003487 | 0.001302 | 0.135401 | 0.203762 | 0.000417 |
| **146** | 0.000229 | 0.002126 | 0.005788 | 0.000987 | 0.150055 | 0.306430 | 0.000683 |
| **147** | 0.000537 | 0.004665 | 0.002712 | 0.001891 | 0.067649 | 0.686347 | 0.000659 |
| **148** | 0.000443 | 0.007026 | 0.002466 | 0.004005 | 0.045698 | 0.854127 | 0.000202 |
| **149** | 0.000232 | 0.005873 | 0.004485 | 0.000534 | 0.169225 | 0.349557 | 0.000477 |

150 rows × 1024 columns

The features are hard to interpret but they have some important information about the images which can be useful for classification.

Let's try out logistic regression on these extracted features.

```
Z_train
```

```
array([[3.1345180e-04, 5.3741559e-03, 2.8259738e-03, ..., 1.5153911e+00,
        2.3207568e-01, 3.1438375e+00],
       [2.4096968e-04, 3.1335356e-03, 3.5071352e-03, ..., 6.5486968e-01,
        1.1149722e+00, 6.8205917e-01],
       [3.6489207e-04, 4.6512899e-03, 3.6485465e-03, ..., 2.2344625e+00,
        1.6992384e+00, 1.4832728e+00],
       ...,
       [5.3744350e-04, 4.6654441e-03, 2.7122677e-03, ..., 2.1755195e+00,
        2.4243596e+00, 1.6216286e+00],
       [4.4266929e-04, 7.0261215e-03, 2.4659948e-03, ..., 2.1108343e-01,
        4.3030614e-01, 1.1860528e+00],
       [2.3162669e-04, 5.8726245e-03, 4.4846330e-03, ..., 1.2377231e-01,
        5.7571673e-01, 2.5890307e+00]], shape=(150, 1024), dtype=float32)
```

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
pipe.fit(Z_train, y_train)
print(f"Train accuracy: {pipe.score(Z_train, y_train) * 100:.2f}%")
print(f"Valid accuracy: {pipe.score(Z_valid, y_valid) * 100:.2f}%")
```

```
Train accuracy: 100.00%
Valid accuracy: 89.33%
```

- So what did we just do:
    1. We passed all images through DenseNet and saved all the output values. DenseNet outputs 1024 values per input. We had 150 images of animal faces, so we extracted a tensor of shape `(150, 1024)` from DenseNet.
    2. We now have a dataset of 1024 features and 150 examples for a multiclass classification problem
    3. We used this data to train a logistic regression model.

# 4.3 Using pre-trained models for fine tuning

- Alright, this is the final and most common workflow of transfer learning

- Above, we stacked some extra layers onto `DenseNet` and just trained those layer (we "froze" all of `DenseNet`'s weights)
- But we can also "fine tune" `DenseNet`'s weights if we like, to make it more suited to our data
- We can choose to "fine tune" all of `DenseNet`'s ~8 million parameters, or just some of them
- To do this, we use the same workflow as above, but we unfreeze the layers we wish to "fine-tune".
- This is computationally intensive. Don't try it on your laptop.

```
# Load (but don't freeze) the model
densenet = models.densenet121(weights='DenseNet121_Weights.DEFAULT')

# Replace classification layer
new_layers = nn.Sequential(
    nn.Linear(1024, 500),
    nn.ReLU(),
    nn.Linear(500, 3)
)

densenet.classifier = new_layers

# Move to GPU if available
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

densenet.to(device);
```

```
# Train the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(densenet.parameters(), lr=2e-3)
# results = trainer(densenet, criterion, optimizer, dataloaders['train'], data
```

```
Epoch 1: Train Accuracy: 0.79 Valid Accuracy: 0.61
Epoch 2: Train Accuracy: 0.95 Valid Accuracy: 0.97
Epoch 3: Train Accuracy: 0.98 Valid Accuracy: 0.98
```

- By far our best results yet
- You could also choose to fine-tune just some layers. For example, below I'll freeze everything but the last two layers:

```
# Freeze all but the last two layers
for layer in densenet.features[:-2]:
    for param in layer.parameters():
        param.requires_grad = False

# Now re-train...
```

- Fine-tuning more layers typically leads to better accuracy, but we have to pay the cost of training.

## 4.4 Transfer Learning Summary

1. Use a pre-trained model as a "feature extractor" (good if you want to adapt a pre-trained model for a specific problem)
2. Fine-tune a pre-trained model (same as 2 but generally yields better results, although at more computational cost)

# 5. Lecture Highlights

1. PyTorch makes data loading easy with `dataset` and `dataloader`.
2. Hyperparameter tuning is hard. Don't do grid-search when the cost of computations is very high. Use smarter methods such as Ax.
3. Transfer learning is a very powerful method to adapt successful models and datasets to your own problem.