5.0 Training

Training a deep model like ResNet-50 would take days on a laptop like mine (MacBook Air M2. Color? Midnight blue by the way). This is because in calling the model, billions of parameters are involved, and we will need to update these parameters on every call, requiring an unfathomable amount of memory (if you are using my less naive implementation) and time.

5.1 The very far from right Training Script

My first training implementation looks like this:

```
import torch
from torch.utils.data import DataLoader
from tqdm import tqdm
import os

os.path.abspath(os.path.join(os.getcwd(), '..', 'dog_and_cat_classifier_cnn_from_scratch'))

from dog_and_cat_classifier_cnn_from_scratch.model import ResNet50
from dog_and_cat_classifier_cnn_from_scratch.data import CatAndDogDataset

# --- Hyperparameters ---
LEARNING_RATE = 0.01
NUM_EPOCHS = 50
BATCH_SIZE = 8
NUM_CLASSES = 2

# --- Setup ---
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Using device: {torch.cuda.get_device_name(0)} ")
else:
    device = torch.device("cpu")
    print("CUDA device not found. Using device: cpu ")

# Instantiate model
model = ResNet50(num_classes=NUM_CLASSES, lr=LEARNING_RATE).to(device)

criterion = model.loss
optimizer = model.configure_optimizers()

dataset = CatAndDogDataset(img_dir='../data/processed')
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

for epoch in range(NUM_EPOCHS):
    model.train()

    progress_bar = tqdm(dataloader, desc=f"Epoch {epoch+1}/{NUM_EPOCHS}", unit="batch", colour="green")

    # Loop over each batch in the dataloader
    for batch_idx, (images, labels) in enumerate(progress_bar):
        # Move data to the correct device
        images, labels = images.to(device), labels.to(device)

        # Zero the gradients from the previous step
        optimizer.zero_grad()

        # Forward pass: get predictions and calculate loss
        outputs = model(images)
        loss = criterion(outputs, labels)
```

```
        # Backward pass: compute gradients
        loss.backward()

        # Update model parameters
        optimizer.step()

        # Update the progress bar with the current loss
        # The .item() is needed to get the scalar value from the tensor
        progress_bar.set_postfix(loss=f'{loss.item():.4f}')

    print(f"\n Epoch {epoch+1} completed! Average loss: {loss.item():.4f}\n")

print(" Training finished! ")
```

See anything wrong with the code?

This was in `dog_and_cat_classifier_cnn_from_scratch/train.py`. Meaning that if I had finished training it, it would've finished with no way for me to recover the weights of the model. With just 3 epochs in, I realized my mistake and wrote another one with the help of AI.

It did not even have the Data Augmentation that we aimed to implement, nor L2 Regularization that both would help with overfitting.

5.2 The correct Training Script

Let's incorporate all the goodies in making a model converge and one where you actually have an output that's saved.

```
# modified_train.py  (drop-in replacement for your original training script)
import torch
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm
import gc
import os
import json
from datetime import datetime
from torchvision import transforms
from torch import nn
import math

os.path.abspath(os.path.join(os.getcwd(), '..', 'dog_and_cat_classifier_cnn_from_scratch'))

from dog_and_cat_classifier_cnn_from_scratch.model import ResNet50, Conv2D, LinearRegression
from dog_and_cat_classifier_cnn_from_scratch.data import CatAndDogDataset

# --- Hyperparameters ---
LEARNING_RATE = 0.1
NUM_EPOCHS = 80
BATCH_SIZE = 64
NUM_CLASSES = 2
VALIDATION_SPLIT = 0.2

def kaiming_init(module):
    """Apply Kaiming initialization to Conv2D and Linear layers"""
    if isinstance(module, Conv2D):
        # He normal initialization for conv layers
        nn.init.kaiming_normal_(module.w, mode='fan_out', nonlinearity='relu')
        if module.b is not None:
            nn.init.zeros_(module.b)
    elif isinstance(module, LinearRegression):
```

```
        nn.init.kaiming_normal_(module.w, mode='fan_out', nonlinearity='linear')
        if module.b is not None:
            nn.init.zeros_(module.b)

def log_weight_stats(model):
    """Print mean and std for each Conv2D and Linear layer"""
    print("  Weight Statistics:")
    for name, module in model.named_modules():
        if isinstance(module, (Conv2D, LinearRegression)):
            w_mean = module.w.mean().item()
            w_std = module.w.std().item()
            print(f" - {name}: mean={w_mean:.4f}, std={w_std:.4f}")

# --- Debug / instrumentation utilities ---
def tensor_stats(t: torch.Tensor):
    t = t.detach().cpu()
    return {
        'min': float(t.min().item()),
        'max': float(t.max().item()),
        'mean': float(t.mean().item()),
        'std': float(t.std().item())
    }

def dump_debug_info(images, labels, model, optimizer, prefix="DEBUG"):
    # Print basic batch info
    print(f"\n=== {prefix} DUMP ===")
    print("-> Input stats:")
    try:
        stats = tensor_stats(images)
        print(f"   min={stats['min']:.6g}, max={stats['max']:.6g}, mean={stats['mean']:.6g},
std={stats['std']:.6g}")
    except Exception as e:
        print("   could not compute input stats:", e)

    print("-> Label stats:")
    try:
        lstats = tensor_stats(labels.float())
        print(f"   min={lstats['min']:.6g}, max={lstats['max']:.6g}, mean={lstats['mean']:.6g},
std={lstats['std']:.6g}, dtype={labels.dtype}")
    except Exception as e:
        print("   could not compute label stats:", e)

    # Optimizer param-group LR info
    if optimizer is not None and hasattr(optimizer, 'param_groups'):
        for i, pg in enumerate(optimizer.param_groups):
            print(f"-> optimizer param_group[{i}] lr={pg.get('lr', 'N/A')}, weight_decay={pg.
 ↪get('weight_decay','N/A')}")
    else:
        print("-> optimizer: not a standard torch optimizer or missing (type=", type(optimizer), ")")

    # Parameter norms / max
    print("-> Top parameter stats (abs max) per named param (first 40 entries):")
    param_list = []
    for name, p in model.named_parameters():
        if p is None:
            continue
        try:
            abs_max = float(p.data.abs().max().cpu())
```

```
            mean = float(p.data.mean().cpu())
            param_list.append((abs_max, name, mean))
        except Exception:
            continue
    param_list = sorted(param_list, key=lambda x: x[0], reverse=True)
    for abs_max, name, mean in param_list[:40]:
        print(f"   {name:50s} abs_max={abs_max:.6g} mean={mean:.6g}")

    # BatchNorm running stats
    print("-> BatchNorm running stats (name, running_mean.max, running_var.max, eps):")
    for n, m in model.named_modules():
        if isinstance(m, nn.BatchNorm2d):
            try:
                rm = float(m.running_mean.abs().max().cpu())
                rv = float(m.running_var.max().cpu())
                print(f"   {n:40s} rm_max={rm:.6g} rv_max={rv:.6g} eps={m.eps}")
            except Exception:
                continue

    # If logits exploded, run forward hooks to find per-layer activation maxima (single forward, no grad)
    print("-> Running forward hooks to capture activation maxima (this is a single extra forward pass on
the same batch)...")
    activation_stats = {}
    hooks = []

    def make_hook(name):
        def hook(module, inp, out):
            try:
                t = out
                if isinstance(out, (list, tuple)):
                    t = out[0]
                # reduce to cpu scalar stats
                activation_stats[name] = (float(t.detach().abs().max().cpu()), float(t.detach().mean().
 ↪cpu()))
            except Exception as e:
                activation_stats[name] = ("ERR", str(e))
        return hook

    # register hooks on conv/bn/linear modules
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.BatchNorm2d, nn.ReLU, nn.Linear)):
            try:
                hooks.append(module.register_forward_hook(make_hook(name)))
            except Exception:
                pass

    # Run a single forward pass with no grad to capture activations
    try:
        model_cpu = model if next(model.parameters()).is_cuda == images.is_cuda else model.to(images.
 ↪device)
    except Exception:
        model_cpu = model
    was_training = model.training
    model.eval()
    with torch.no_grad():
        try:
            _ = model(images)
        except Exception as e:
```

```
            print("Forward during debug hook pass failed:", e)

    # Print top activation offenders
    items = []
    for k, v in activation_stats.items():
        try:
            if isinstance(v, tuple) and isinstance(v[0], float):
                items.append((v[0], k, v[1]))
        except Exception:
            continue
    items.sort(reverse=True, key=lambda x: x[0])
    print("Top activation offenders (max, name, mean):")
    for mx, name, mean in items[:40]:
        print(f"   {name:50s} max={mx:.6g} mean={mean:.6g}")

    # remove hooks
    for h in hooks:
        try:
            h.remove()
        except Exception:
            pass

    model.train(mode=was_training)
    print(f"=== {prefix} DUMP END ===\n")

# --- Setup ---
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Create directories for saving
os.makedirs('../models', exist_ok=True)
os.makedirs('../models/training_checkpoints', exist_ok=True)

def clear_memory():
    torch.cuda.empty_cache()
    gc.collect()

clear_memory()

# --- Automatic Model Loading ---
def find_latest_checkpoint():
    checkpoints = [f for f in os.listdir('../models/training_checkpoints') if f.endswith('.pth')]
    if not checkpoints:
        return None
    checkpoints.sort(key=lambda x: os.path.getmtime(os.path.join('../models/training_checkpoints', x)),
reverse=True)
    return os.path.join('../models/training_checkpoints', checkpoints[0])

def load_checkpoint(model, optimizer=None):
    checkpoint_path = find_latest_checkpoint()
    if checkpoint_path and os.path.exists(checkpoint_path):
        print(f" Loading checkpoint: {checkpoint_path}")
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint['model_state_dict'])
        start_epoch = checkpoint['epoch']
        best_val_loss = checkpoint['best_val_loss']
        training_history = checkpoint.get('training_history', [])
        if optimizer and 'optimizer_state_dict' in checkpoint:
            try:
```

```
                optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
            except Exception as e:
                print("Could not load optimizer state:", e)
        print(f"  Resuming from epoch {start_epoch}")
        return start_epoch, best_val_loss, training_history
    print("  No checkpoint found, starting fresh training")
    return 0, float('inf'), []

# --- Automatic Model Saving ---
def save_checkpoint(epoch, model, optimizer, best_val_loss, training_history, is_best=False):
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    checkpoint = {
        'epoch': epoch + 1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict() if optimizer is not None and hasattr(optimizer,
'state_dict') else None,
        'best_val_loss': best_val_loss,
        'training_history': training_history,
        'timestamp': timestamp,
        'hyperparameters': {
            'learning_rate': LEARNING_RATE,
            'batch_size': BATCH_SIZE,
            'num_epochs': NUM_EPOCHS
        }
    }
    checkpoint_path = f'../models/training_checkpoints/checkpoint_epoch_{epoch+1}_{timestamp}.pth'
    torch.save(checkpoint, checkpoint_path)
    if is_best:
        best_model_path = f'../models/best_model_{timestamp}.pth'
        torch.save(checkpoint, best_model_path)
        print(f"  Saved best model: {best_model_path}")
    history_path = f'../models/training_checkpoints/training_history.json'
    serializable_history = [{k: v.item() if isinstance(v, torch.Tensor) else v for k, v in d.items()} for
d in training_history]
    with open(history_path, 'w') as f:
        json.dump(serializable_history, f, indent=2)
    return checkpoint_path

def save_final_model(model, training_history, final_val_loss, final_val_acc):
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    final_checkpoint = {
        'model_state_dict': model.state_dict(),
        'final_val_loss': final_val_loss,
        'final_val_acc': final_val_acc,
        'training_history': training_history,
        'timestamp': timestamp,
        'hyperparameters': {
            'learning_rate': LEARNING_RATE,
            'batch_size': BATCH_SIZE,
            'num_epochs': NUM_EPOCHS
        }
    }
    final_path = f'../models/final_model_{timestamp}.pth'
    torch.save(final_checkpoint, final_path)
    print(f"  Saved final model: {final_path}")
    return final_path

def adjust_learning_rate(optimizer, epoch, base_lr=LEARNING_RATE, schedule=[20, 35, 50, 60], gamma=0.1):
```

```
    """
    Reduce learning rate at specific epochs.
    """
    lr = base_lr
    for milestone in schedule:
        if epoch >= milestone:
            lr *= gamma
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    return lr

# --- Instantiate Model ---
model = ResNet50(num_classes=NUM_CLASSES, lr=LEARNING_RATE, in_channels=3, dropout_rate=0.3).to(device)

# Ensure final classifier is initialized safely if present (small std, zero bias)
if hasattr(model, 'fc'):
    try:
        nn.init.normal_(model.fc.weight, std=0.01)
        if model.fc.bias is not None:
            nn.init.zeros_(model.fc.bias)
    except Exception:
        pass

# Log initial weight statistics
log_weight_stats(model)

# Use model's own optimizer and loss function (as you requested: use what you implemented)
optimizer = model.configure_optimizers()
criterion = model.loss

print(f" Using model's built-in optimizer and loss function (kept untouched)")

start_epoch, best_val_loss, training_history = load_checkpoint(model, optimizer)

# --- Dataset and DataLoaders ---
train_transform = transforms.Compose([
    transforms.Lambda(lambda x: x / 255.0),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1), scale=(0.9, 1.1)),
    transforms.RandomGrayscale(p=0.1),
    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]),
])

val_transform = transforms.Compose([
    transforms.Lambda(lambda x: x / 255.0),
    transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]),
])

train_dataset = CatAndDogDataset(img_dir='../data/processed', train=True, transform=train_transform)
val_dataset = CatAndDogDataset(img_dir='../data/processed', train=False, transform=val_transform)

dataset_size = len(train_dataset)
val_size = int(VALIDATION_SPLIT * dataset_size)
train_size = dataset_size - val_size

indices = torch.randperm(dataset_size).tolist()
train_indices = indices[:train_size]
```

```
val_indices = indices[train_size:]

train_subset = torch.utils.data.Subset(train_dataset, train_indices)
val_subset = torch.utils.data.Subset(val_dataset, val_indices)

print(f"  Training: {len(train_subset)} samples")
print(f"  Validation: {len(val_subset)} samples")

train_loader = DataLoader(train_subset,
                          batch_size=BATCH_SIZE,
                          shuffle=True,
                          pin_memory=True,
                          num_workers=4)

val_loader = DataLoader(val_subset,
                        batch_size=BATCH_SIZE * 2,
                        shuffle=False,
                        pin_memory=True,
                        num_workers=2)

print(f"\n  Starting training from epoch {start_epoch + 1}...")
for epoch in range(start_epoch, NUM_EPOCHS):
    # --- Adjust learning rate ---
    current_lr = adjust_learning_rate(optimizer, epoch)
    print(f"  Epoch {epoch+1} | Learning Rate: {current_lr:.8f}")

    # Print optimizer param groups for sanity
    for i, pg in enumerate(optimizer.param_groups):
        print(f"    param_group[{i}] lr={pg.get('lr', 'N/A')} weight_decay={pg.get('weight_decay','N/
 ↪A')}")

    model.train()
    train_loss = 0.0
    correct = 0
    total = 0

    progress_bar = tqdm(train_loader, desc=f"Epoch {epoch+1}/{NUM_EPOCHS}")

    for batch_idx, (images, labels) in enumerate(progress_bar):
        images, labels = images.to(device), labels.to(device)

        if batch_idx == 0:
            try:
                print(f"[epoch {epoch+1} batch {batch_idx}] input min/max/mean/std:",
                      float(images.min().item()), float(images.max().item()),
                      float(images.mean().item()), float(images.std().item()))
            except Exception:
                pass
            try:
                print(f"[epoch {epoch+1} batch {batch_idx}] labels min/max/dtype:",
                      float(labels.min().item()), float(labels.max().item()), labels.dtype)
            except Exception:
                pass

        optimizer.zero_grad()
        outputs = model(images)

        loss = criterion(outputs, labels)
        loss.backward()
```

```
        optimizer.step()

        pure_loss = float(loss.item())
        train_loss += pure_loss * images.size(0)

        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

        progress_bar.set_postfix({
            'loss': f'{pure_loss:.4f}',
            'acc': f'{100.*correct/total:.2f}%',
            'mem': f'{torch.cuda.memory_allocated()/1024**3:.2f}GB' if torch.cuda.is_available() else
'cpu',
        })

    # --- Validation ---
    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += float(loss.item()) * images.size(0)
            _, predicted = outputs.max(1)
            val_total += labels.size(0)
            val_correct += predicted.eq(labels).sum().item()

    avg_train_loss = train_loss / len(train_dataset)
    train_acc = 100. * correct / total if total > 0 else 0.0
    avg_val_loss = val_loss / len(val_dataset)
    val_acc = 100. * val_correct / val_total if val_total > 0 else 0.0

    epoch_stats = {
        'epoch': epoch + 1,
        'train_loss': avg_train_loss,
        'train_acc': train_acc,
        'val_loss': avg_val_loss,
        'val_acc': val_acc,
        'learning_rate': current_lr,
        'timestamp': datetime.now().isoformat(),
    }
    training_history.append(epoch_stats)

    print(f"\n  Epoch {epoch+1}:")
    print(f"    Train Loss: {avg_train_loss:.6f}, Train Acc: {train_acc:.2f}%")
    print(f"    Val Loss: {avg_val_loss:.6f}, Val Acc: {val_acc:.2f}%")

    # Log weight stats
    log_weight_stats(model)

    # Check best model
    is_best = avg_val_loss < best_val_loss
    if is_best:
```

```
        best_val_loss = avg_val_loss
        best_val_acc = val_acc
        print("    New best model!")

    # Save checkpoint
    checkpoint_path = save_checkpoint(epoch, model, optimizer, best_val_loss, training_history, is_best)
    print(f"    Checkpoint saved: {checkpoint_path}")

    clear_memory()

# --- Final Model Save ---
print(f"\n Training completed!")
print(f"   Best Validation Loss: {best_val_loss:.6f}")
print(f"   Best Validation Accuracy: {best_val_acc:.2f}%")
print(f"   Total Epochs Trained: {len(training_history)}")

final_path = save_final_model(model, training_history, best_val_loss, best_val_acc)
print(" Training finished! ")
print(f" Models saved in: models/")
print(f" Checkpoints saved in: models/training_checkpoints")
```

This new training script comes with all sort of useful features: - automatic checkpoints - automatic saving of best models - precise and surgical memory dumping - learning scheduler

The old script still had some problem with a batch size of 8, and so I went back to recheck all of my implementations and I found out that my ResNet layer were implemented incorrectly, resulting in an extra of tens of billion of parameters. At first, it took us a total of 8 excruciating hours to even get to the 50th epoch.

Once the correct model has actually been implemented, we increased the batch-sized to 64 and the epoch numbers to 70. Then, we trained it for a total of 2 hours on a single RTX 5090 and achieved a validation accuracy of 97.6%. (Wow!!).

Let's talk about the hyperparameters and what we did right, and what we could've done better. There are only three that matters in our case: 1. Batch size, 2. Epochs, and 3. Learning rate.

An optimal and faithful-to-design batch size would've been 128 to 256 size. I could not find a way to optimize our model and trainer to utilize less memory, leading to the design choice of just 64. Not to mention that the original paper had trained it on multiple GPUs, with a higher dedicated amount of ram, which had allowed for the choice of 256 batch size. Despite a reduction from the original batch size, our model managed to converged extremely well to a validation accuracy of >90% within 50 epochs.

We chose an epoch of 70 instead of 90-110 (the optimal and, again, faithful-to-design option) since we're working with just binary labels. Any more epochs would have leaded to overfitting, and the model would just memorize the data instead of generalizing to it.

A learning rate of 0.1 was first chosen. We also have a learning rate scheduler to scale down our learning rate by a factor of 0.1 in order to not miss the mark on finding the best answer. If we had kept our learning rate constant, then it would never reach a good validation accuracy since the model weight would overshoot the correct answer.

With all the hyperparameters and the tuning in between training, after just under 2 hours, we have achieved the ultimate Cat and Dog classifier!

5.3 Inference

Now comes the moment of truth. We will load the model and call `model.eval()` on it to activate inference phase. First we will load our model.

```
import torch
import os
os.path.abspath(os.path.join(os.getcwd(), '..', 'dog_and_cat_classifier_cnn_from_scratch'))

from dog_and_cat_classifier_cnn_from_scratch.model import ResNet50
from dog_and_cat_classifier_cnn_from_scratch.data import CatAndDogDataset
```

```
# Load model properly
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ResNet50(num_classes=2, lr=0.01, in_channels=3, dropout_rate=0.3)

model_path = "../models/final_model_20250915_092314.pth"
checkpoint = torch.load(model_path, map_location=device)
model.load_state_dict(checkpoint['model_state_dict'])
model.to(device)
model.eval()

print(" Model loaded successfully!")
```

```
 Model loaded successfully!
```

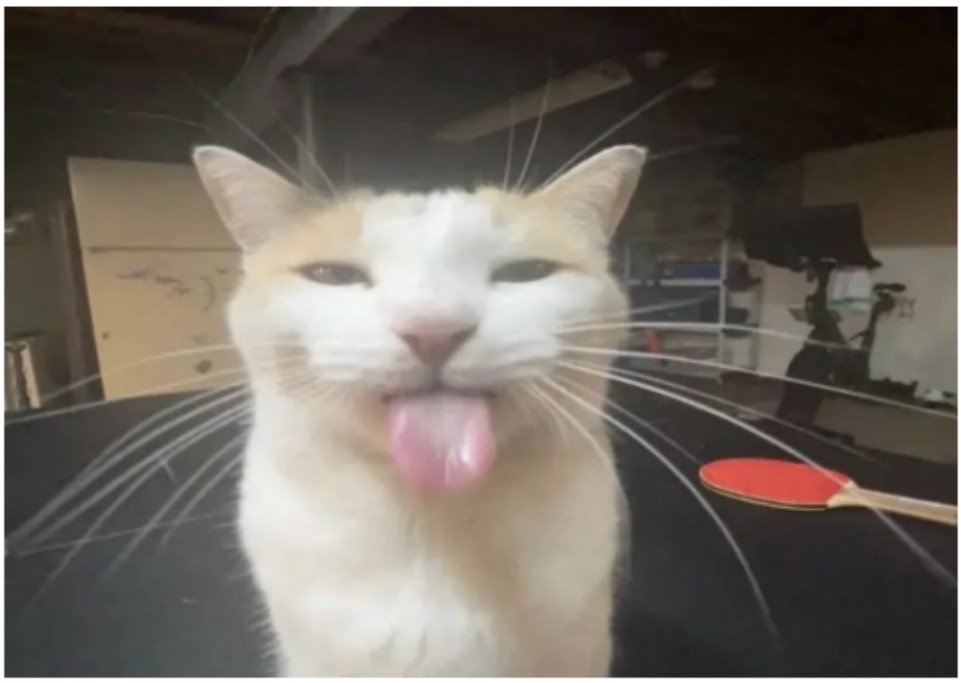
Let us test it on my most favorite cat image, the one with its tongue out, this image has haunted me over the past few days as of the time of writing this. Let's see what it would say?

Before I write any code, remember, our inference phase must have the same preprocessing pipeline as our training/validation dataset. All the normalization that we mentioned in the last notebook.

```
import numpy as np
import cv2
from urllib.request import urlopen
import matplotlib.pyplot as plt
from torchvision import transforms
import torch
from PIL import Image

img_url = "https://i.pinimg.com/736x/b6/83/78/b683788bf174e73ec3281c0f33cfecce.jpg"

resp = urlopen(img_url)
image_binary = np.asarray(bytearray(resp.read()), dtype="uint8")

image = cv2.imdecode(image_binary, cv2.IMREAD_COLOR)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

plt.imshow(image)
plt.axis('off')
plt.title('A cat sticking its tongue out')
plt.show()

print(f"Original image shape: {image.shape}")

image = cv2.imdecode(image_binary, cv2.IMREAD_COLOR)
image = cv2.resize(image, (224, 224), cv2.INTER_AREA)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

image = torch.from_numpy(image).permute(2, 0, 1).float()

preprocess = transforms.Compose([
    transforms.Lambda(lambda x: x / 255.0),  # Normalize to [0, 1]
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

def preprocess_image(image):
    # Use the exact same preprocessing as your training
    tensor = preprocess(image)
    return tensor.unsqueeze(0)
```

```python
# Preprocess the image
input_tensor = preprocess_image(image)

print(f"Final input shape: {input_tensor.shape}")
print(f"Input range: [{input_tensor.min():.3f}, {input_tensor.max():.3f}]")
print(f"Input mean: {input_tensor.mean():.3f}")

# Set model to evaluation mode
model.eval()

with torch.no_grad():
    # Model now outputs LOGITS (raw scores before softmax)
    logits = model(input_tensor)

    # Apply softmax to convert logits to probabilities
    probabilities = torch.softmax(logits, dim=1)

prediction = torch.argmax(logits).item()
confidence = probabilities[0][prediction].item() * 100

print(f"\n Prediction: {'CAT' if prediction == 0 else 'DOG'}")
print(f" Confidence: {confidence:.2f}%")
print(f" Output logits: {logits}")
print(f" Probabilities: {probabilities}")

# Show class-wise probabilities
class_names = ['CAT', 'DOG']
for i, (class_name, prob) in enumerate(zip(class_names, probabilities[0])):
    print(f"{class_name}: {prob.item() * 100:.2f}%")

if prediction == 0:
    print(" It is a cat!")
else:
    print(" It is a dog! (This might be wrong)")
```

A cat sticking its tongue out



```
Original image shape: (518, 736, 3)
Final input shape: torch.Size([1, 3, 224, 224])
Input range: [-1.948, 2.553]
Input mean: -0.250

 Prediction: CAT
 Confidence: 100.00%
 Output logits: tensor([[ 5.1016, -5.1318]])
 Probabilities: tensor([[9.9996e-01, 3.5951e-05]])
CAT: 100.00%
DOG: 0.00%
 It is a cat!
```

Woo-hoo! Our model is very confident that it is a cat. Too confident I would say, it could probably be a dog with a wig for all I know. We could see the output logits, and it makes perfect sense–a negative logit would yield very low probability, due to exponents, and a positive logit would yield high probability.