Deep Learning with Pokemon: A Fast-Track Guide to Building a Classification Demo

I. Introduction: Your Fast-Track Plan for a Pokemon Classifier Demo

The objective is clear: to develop a functional demonstration of a Pokemon image classification system for a presentation scheduled for the following day. This demanding timeline necessitates a strategic approach that prioritizes speed and feasibility above comprehensive model optimization or exhaustive dataset curation. This guide outlines a direct roadmap leveraging established deep learning techniques and readily available resources to achieve this goal efficiently.

The cornerstone of this rapid development strategy is **transfer learning**. Training complex deep learning models, particularly Convolutional Neural Networks (CNNs) for image tasks, typically requires vast amounts of labeled data and considerable computational time, often spanning days or weeks.[1] Attempting to train a Pokemon classifier from scratch within 24 hours is impractical given the scale of typical Pokemon image datasets (thousands of images, compared to millions used for foundational models like ImageNet).[1] Transfer learning provides a critical shortcut by utilizing models pre-trained on large, diverse datasets (e.g., ImageNet [1]). The underlying principle is that these models have already learned robust, general-purpose visual features—edges, textures, shapes—that are applicable to a wide range of image recognition tasks, including identifying Pokemon.[2] By leveraging these pre-learned features, the need to train a model from the ground up is eliminated, drastically reducing development time and data requirements.[7]

This guide will proceed systematically through the essential steps:

1. **Dataset Selection:** Evaluating options and recommending a suitable, pre-labeled dataset optimized for rapid deployment.
2. **Efficient Training:** Detailing the transfer learning process, focusing on feature extraction for speed, and recommending appropriate models and frameworks.
3. **Data Handling:** Outlining crucial preprocessing steps and pragmatic approaches to common challenges like class imbalance within the demo context.
4. **Streamlit Application Development:** Providing a blueprint for building the interactive user interface for the demo.
5. **Resource Pointers:** Consolidating links to key tutorials, code examples, and datasets.

Following this structured plan maximizes the chances of producing a working demonstration within the stringent time constraints.

II. Selecting Your Pokemon Dataset: Prioritizing Speed and Suitability

The foundation of any machine learning project is the dataset. For this time-sensitive task, selecting an appropriate dataset is paramount, balancing scope, quality, and accessibility.

A. Feasibility Check: Scraping vs. Existing Datasets

The prospect of creating a bespoke dataset by scraping images from the web or using APIs like PokeAPI [11] might seem appealing for ultimate customization.[12] However, this approach introduces significant overhead incompatible with a next-day deadline. The process involves several time-consuming stages:

- **Scraper Development:** Writing and debugging code to systematically gather images.
- **Data Acquisition:** Downloading potentially thousands of images, subject to network speed and source limitations.
- **Data Cleaning:** Filtering out irrelevant images, duplicates, or low-quality samples.
- **Labeling/Verification:** The most critical bottleneck – accurately assigning the correct Pokemon species label to each image, a task often requiring manual effort or complex verification logic.

Collectively, these steps typically require days, if not weeks, of dedicated effort. Therefore, constructing a custom dataset is entirely unrealistic under the current time pressure. The only viable strategy is to leverage **pre-existing, labeled Pokemon image datasets** readily available from platforms like Kaggle [2] and GitHub [4], which host numerous collections curated by the community.

B. Evaluating Candidate Datasets (Focus: Gen 1 Species Classification)

Referring back to the initial research summary (Table 1), several datasets warrant consideration, particularly those suitable for *species classification* with a manageable scope.

Analysis of available datasets reveals a common starting point for many Pokemon-related deep learning projects: the original 150 species from **Generation 1**.[2] This focus likely stems from a combination of factors beneficial for rapid development:

- **Manageable Scope:** Classifying 150 distinct categories is significantly less complex and computationally demanding than tackling the full Pokedex of nearly 900 species found in more comprehensive datasets.
- **Data Availability:** Several well-structured datasets specifically target this generation, offering thousands of labeled images.[2]
- **Nostalgic Appeal:** The familiarity and distinct characteristics of the first generation make it an engaging subject for introductory projects.

Key candidate datasets include:

- **"7,000 Labelled Pokémon" (Kaggle)** [2]**:** This dataset is frequently cited and used.[7] It contains over 7,000 images covering the 150 first-generation Pokemon, explicitly labeled for species classification. The images are generally centered and of reasonable resolution, suitable for training without excessive computational demands.[2] A noted characteristic is **class imbalance**, with image counts per Pokemon varying from roughly 20 to 70.[2] While imbalance is a common real-world data challenge, its presence must be acknowledged. Its focus on Gen 1 makes it highly suitable for a quick demo.
- **"PokemonClassification" (Kaggle)** [3]**:** Very similar to the above, this dataset also focuses on the 150 Gen 1 species, providing approximately 25 to 50 images per Pokemon. It is described as having high-quality, centered images with correct labels, making it another strong candidate for species classification.[3] It shares the Gen 1 focus and manageable scope advantages.
- **"The Complete Pokemon Images Data Set" (Kaggle):** This dataset is more comprehensive, covering Generations 1 through 8 (898 Pokemon). While offering broader coverage, the significantly larger number of classes (~900) increases the complexity of the classification task and potentially exacerbates class imbalance issues. Training a model to perform adequately across so many classes within a day is considerably more challenging. Labels are typically derived from image filenames.
- **GitHub Repositories:** Several GitHub repositories offer Pokemon images, sometimes structured specifically for frameworks like PyTorch [4] or focusing on high-quality official artwork.[15] While valuable, for immediate use in a standard classification task under time pressure, the Kaggle datasets often provide a more straightforward starting point due to their common use in tutorials and competitions, unless specific features like the PyTorch Dataset structure [4] are explicitly desired.
- **Other Task Datasets:** It's worth noting the existence of datasets tailored for different tasks, such as predicting Pokemon types [6], image generation [8], or multimodal learning combining images and text/stats.[9] These highlight the

breadth of possibilities but fall outside the scope of this rapid species classification project.

C. Recommendation and Rationale

Based on the requirements of speed, feasibility, and the specific task of species classification for a demo, the recommendation is to use either the **"7,000 Labelled Pokémon" dataset** [2] or the **"PokemonClassification" dataset** [3] from Kaggle.

Justification:

- **Ready Availability:** Both are hosted on Kaggle, easily downloadable with a free account.
- **Appropriate Labeling:** They are explicitly labeled for species classification.
- **Manageable Scope:** Focusing on Gen 1 (150 classes) significantly reduces model complexity and training time compared to full Pokedex datasets.
- **Sufficient Data:** ~4,000-7,000 images provide enough data for effective transfer learning, especially when combined with augmentation.
- **Community Validation:** Their use in other documented projects [7] suggests their suitability for this type of task.

D. Key Table: Recommended Pokemon Datasets for Rapid Classification

The following table summarizes the key characteristics of the recommended datasets:

Dataset Name	**Source (Kaggle Link)**	**Approx. Size / Scope**	**Key Features**	**Suitability Notes**
7,000 Labelled Pokémon	kaggle.com/datasets/lantian773030/pokemonclassification (Note: Link points to 'PokemonClassification', often associated/similar)	~7,000+ images, 150 Gen 1 species	Labeled Species, Centered Images, JPG format	Ideal for quick demo, manageable classes, known class imbalance [2], requires Kaggle account
PokemonClassification	kaggle.com/datasets/lantian773030/pokemonclassification	~4,000 images, 150 Gen 1 species	Labeled Species, High-quality, Centered	Excellent alternative, manageable classes, likely similar

			Images	imbalance, requires Kaggle account

(Note: Accessing Kaggle datasets requires a free Kaggle account.)

III. Training Your Classifier Efficiently: Leveraging Transfer Learning

With a suitable dataset selected, the next step is to train the classification model. Given the time constraint, transfer learning is the only practical approach. This involves adapting a pre-trained CNN.

A. Transfer Learning Strategies: Feature Extraction vs. Fine-Tuning

Two primary transfer learning strategies exist [6]:

1. **Feature Extraction:** This approach treats the pre-trained CNN (e.g., MobileNetV2, ResNet50) primarily as a fixed feature extractor.[1] The process involves:
   - Loading the pre-trained model without its final classification layer (the "head").
   - **Freezing** the weights of the remaining layers (the convolutional "base"). This prevents the pre-learned general visual features from being altered during training.[1]
   - Adding new, trainable classification layers on top of the frozen base. These layers (e.g., a pooling layer followed by one or more Dense/Linear layers) learn to map the extracted features to the specific Pokemon classes.[6]
   - Training only these newly added layers. Since only a small portion of the network is trained, this method is computationally efficient and converges quickly.[9]
2. **Fine-Tuning:** This strategy goes a step further by selectively **unfreezing** some of the top layers of the pre-trained base model.[2] These unfrozen layers are then trained alongside the new classification head, typically using a **very low learning rate**.[2] The goal is to slightly adjust the pre-trained features to better suit the nuances of the target dataset (Pokemon images).[6] Fine-tuning can potentially lead to higher accuracy but requires more careful implementation, longer training times, and risks "catastrophic forgetting" if the learning rate is not set appropriately.[2]

Recommendation for Demo: Due to the extreme time constraint ("demo tomorrow"),

Feature Extraction is the strongly recommended starting point. It offers the fastest path to a functional baseline model. It is simpler to implement and requires significantly less training time compared to fine-tuning.[1] Fine-tuning should only be considered as an optional enhancement if the feature extraction process yields results very quickly and spare time allows for further experimentation. The complexity and tuning required for effective fine-tuning [2] introduce risks incompatible with the deadline.

B. Choosing Your Tools: Model and Framework

Selecting the right pre-trained model and deep learning framework impacts efficiency and ease of implementation.

- **Pre-trained Model Selection:**
  - **MobileNetV2:** Developed by Google, MobileNetV2 is known for its computational efficiency, making it suitable for applications with limited resources, including potentially faster training.[4] It often achieves a good balance between speed and accuracy and has been used successfully for Pokemon classification.[13] It requires specific input preprocessing: typically 224x224 pixel images with pixel values scaled between -1 and +1.[4] Tutorials and examples specifically using MobileNetV2 for transfer learning are available.[4]
  - **ResNet50:** A highly influential, deeper architecture known for strong performance on image classification tasks.[1] It's a common choice for transfer learning and has been applied to Pokemon classification.[21] While powerful, it generally requires more computational resources for training and inference compared to MobileNetV2. Numerous tutorials exist.[1]
  - **Other Options:** Models like DenseNet201 have also been used in similar projects.[11]
  - **Recommendation: MobileNetV2** [4] is recommended as the primary choice due to its efficiency, which aligns well with the need for rapid training and potential deployment in a lightweight Streamlit app. ResNet50 [1] remains a viable alternative if computational resources are not a major constraint.
- **Framework Selection (TensorFlow/Keras vs. PyTorch):**
  - **TensorFlow/Keras:** Offers a high-level API (tf.keras) often favored for its straightforward syntax for standard model building and training.[3] Google provides comprehensive official tutorials on transfer learning [2], including specific examples with MobileNetV2.[4] Several Pokemon/Streamlit examples utilize TensorFlow.[11]
  - **PyTorch:** Widely used in the research community, known for its flexibility and

Pythonic feel. It also has excellent official transfer learning tutorials [1] and supporting resources.[8] Many Streamlit examples are built with PyTorch.[16]

- **Recommendation:** Both frameworks are fully capable of implementing the required transfer learning strategy. The choice should be guided by **user familiarity and the perceived clarity of available resources for *rapid adaptation***. Reviewing the core transfer learning tutorials for both frameworks (e.g., TensorFlow: [5]; PyTorch: [1]) and selecting the one that seems quickest to implement is advisable. The abundance of Streamlit examples for both platforms ensures support regardless of the choice.

C. Implementation Steps (Illustrative - Feature Extraction with Keras/MobileNetV2)

The following conceptual steps illustrate the feature extraction process using TensorFlow/Keras and MobileNetV2:

1. **Load Base Model:**
   Python
   ```python
   import tensorflow as tf

   IMG_SHAPE = (224, 224, 3) # Or size appropriate for chosen model
   base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                                  include_top=False, # Exclude classifier head
                                                  weights='imagenet') # Load pre-trained weights
   ```
   Loading the model without the top layer (include_top=False) is crucial.[4]
2. **Freeze Base Model Layers:**
   Python
   ```python
   base_model.trainable = False # Freeze the convolutional base
   ```
   This prevents updating the learned ImageNet features.[4]
3. **Add New Classifier Head:**
   Python
   ```python
   num_classes = 150 # For Gen 1 Pokemon

   model = tf.keras.Sequential([
     base_model,
     tf.keras.layers.GlobalAveragePooling2D(), # Pool features
     tf.keras.layers.Dropout(0.2), # Optional regularization
     tf.keras.layers.Dense(num_classes, activation='softmax') # Output layer for Pokemon
   classes
   ```

```
])
```

A GlobalAveragePooling2D layer reduces spatial dimensions, followed by a Dense layer with softmax activation for multi-class classification.[6] The number of output neurons must match the number of Pokemon classes.

4. **Compile Model:**
   Python
   ```python
   model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
             loss='categorical_crossentropy',
             metrics=['accuracy'])
   ```
   Use an appropriate optimizer like Adam [29] and categorical_crossentropy loss for multi-class classification. A standard learning rate like 0.001 is a reasonable starting point.[1]
5. **Prepare Data Loaders:** (Covered in next section, but conceptually involves using ImageDataGenerator [6]).
6. **Train the Model:**
   Python
   ```python
   # Assuming train_generator and validation_generator are set up
   epochs = 10 # Start with a small number for the demo
   history = model.fit(train_generator,
                   epochs=epochs,
                   validation_data=validation_generator)
   ```
   Train for a limited number of epochs initially (e.g., 5-10) to get a working model quickly.[9] Monitor validation accuracy.
7. **Save the Model:**
   Python
   ```python
   model.save('pokemon_classifier_mobilenetv2.h5') # Save the entire model
   ```
   Saving the trained model is essential for loading it into the Streamlit application.[1]

(Equivalent steps exist for PyTorch, involving loading models from torchvision.models, modifying the fc layer, freezing parameters via requires_grad=False, defining nn.CrossEntropyLoss and an optimizer, and implementing a training loop [1]).

IV. Essential Data Handling for Quick Results

Proper data handling is critical for transfer learning success. The pre-trained model expects inputs in a specific format, and addressing dataset challenges pragmatically is necessary for the demo.

A. Input Pipeline and Preprocessing

Pre-trained models like MobileNetV2 or ResNet50 have specific input requirements derived from their original training on ImageNet.[4] Failing to replicate these preprocessing steps will result in significantly degraded performance, as the model receives data in a format it doesn't understand.

- **Resizing:** Images from the dataset must be resized to the exact height and width expected by the model's input layer. Common sizes are 224x224 or 299x299 pixels.[4] Tools like Keras' ImageDataGenerator(target_size=...) or PyTorch's transforms.Resize and transforms.CenterCrop handle this.[1]
- **Normalization:** Pixel values need to be scaled to match the range used during the model's pre-training.
    - For MobileNetV2, this often means scaling pixels to the range [-1, 1] using its dedicated preprocess_input function.[4]
    - For other models like ResNet, it typically involves converting pixels to (by dividing by 255) and then normalizing using the ImageNet dataset's mean and standard deviation values.[19] PyTorch's transforms.ToTensor and transforms.Normalize are commonly used for this.[1]
- **Data Loaders/Generators:** Efficiently loading and preprocessing images during training is handled by data loaders. Keras' ImageDataGenerator.flow_from_directory automatically reads images from class-specific folders, applies preprocessing and augmentation, and yields batches.[6] PyTorch uses torchvision.datasets.ImageFolder combined with torch.utils.data.DataLoader to achieve the same.[1] These tools are essential for managing datasets that don't fit entirely in memory.

B. Addressing Challenges (Pragmatic Demo Solutions)

The recommended datasets exhibit **class imbalance** [2], meaning some Pokemon species have significantly more training images than others. This can bias the model, leading it to perform better on common Pokemon and worse on rarer ones. While sophisticated techniques exist to combat imbalance (e.g., weighted loss functions, class-aware sampling [18]), implementing and tuning them adds complexity unsuitable for the tight deadline. Pragmatic solutions for the demo include:

- **Mitigation 1: Basic Data Augmentation:** Applying simple, random transformations to the training images artificially increases dataset size and diversity.[2] This helps the model generalize better and can partially alleviate the effects of imbalance by creating more varied examples, especially for underrepresented classes. Suitable augmentations for Pokemon include:

  - Random Horizontal Flips (RandomFlip in Keras, transforms.RandomHorizontalFlip in PyTorch [8])
  - Slight Random Rotations (RandomRotation in Keras, transforms.RandomRotation in PyTorch [8])
  - Minor Zooms (RandomZoom in Keras) These can be easily integrated into the ImageDataGenerator or PyTorch transforms pipeline.[8] Avoid overly aggressive transformations that might obscure defining Pokemon features.
- **Mitigation 2: Focus on Overall Accuracy:** For the purpose of the demonstration, prioritize evaluating the model based on its **overall accuracy** across all 150 classes. This provides a single, easily understandable metric of performance. While more granular metrics like per-class precision, recall, and F1-score [2] offer a deeper understanding of model behavior (especially on imbalanced data), they might highlight poor performance on minority classes. Given the time constraint, achieving a reasonable overall accuracy for the demo is sufficient, even if performance isn't uniform across all Pokemon.

The necessity of matching the pre-trained model's preprocessing pipeline cannot be overstated.[4] For handling dataset challenges like imbalance within the demo context, the key is simplification. Basic augmentation is a low-effort way to improve robustness [2], and focusing on overall accuracy provides a practical evaluation metric for the presentation.[2]

V. Building Your Interactive Streamlit Demo Application

Streamlit is an excellent choice for rapidly creating interactive web applications for machine learning models, requiring only Python code.

A. Core Streamlit Components

Building the Pokemon classifier demo app involves several key Streamlit elements:

1. **Basic Setup:** Import Streamlit and set a title for the application.[22]
   Python
   ```python
   import streamlit as st
   st.title("Pokemon Species Classifier Demo")
   ```

2. **Model Loading:** Load the trained Keras (.h5) or PyTorch (.pth) model saved previously. Crucially, use Streamlit's caching (@st.cache_resource) to ensure the model (which can be large) is loaded only once when the app starts, not on every user interaction, significantly improving responsiveness.
   Python

```python
# Example for Keras/TensorFlow
import tensorflow as tf
@st.cache_resource
def load_my_model():
    model = tf.keras.models.load_model('pokemon_classifier_mobilenetv2.h5')
    return model
model = load_my_model()

# Example for PyTorch (requires model class definition)
# @st.cache_resource
# def load_pytorch_model():
#     # Assuming 'YourModelClass' is defined elsewhere
#     pytorch_model = YourModelClass(num_classes=150)
#     pytorch_model.load_state_dict(torch.load('pokemon_classifier_resnet.pth'))
#     pytorch_model.eval() # Set to evaluation mode
#     return pytorch_model
# model = load_pytorch_model()
```

3. **Image Upload:** Use st.file_uploader to create a widget allowing users to upload an image file.[22] Restrict accepted file types to common image formats.

```python
Python
from PIL import Image
import numpy as np

uploaded_file = st.file_uploader("Choose a Pokemon image...", type=["jpg", "png", "jpeg"])
```

4. **Image Preprocessing Function:** Define a function that takes the uploaded file, opens it using the Pillow (PIL) library, and applies the *exact same* resizing and normalization transformations used during model training. This consistency is critical.[4] The function should return the image data in the format expected by the model (e.g., a NumPy array or PyTorch tensor with the correct shape and value range).

```python
Python
# Example preprocessing function (adapt to match your training)
def preprocess_image(image_data, target_size=(224, 224)):
    img = Image.open(image_data).convert('RGB') # Ensure 3 channels
    img = img.resize(target_size)
    img_array = np.array(img)
    # Apply normalization specific to your model (e.g., MobileNetV2 [-1, 1])
    # Example: preprocessed_img =
tf.keras.applications.mobilenet_v2.preprocess_input(img_array)
    # Or for PyTorch: apply transforms.ToTensor() and transforms.Normalize()
    #... add normalization steps here...
    img_batch = np.expand_dims(img_array, axis=0) # Create batch dimension
    return img_batch # Or return preprocessed_img after normalization
```

5. **Prediction/Inference:** If a file is uploaded, call the preprocessing function and then feed the result to the loaded model to get predictions.[1]
   Python
```
if uploaded_file is not None:
    image = Image.open(uploaded_file)
    st.image(image, caption='Uploaded Image', use_column_width=True)
    st.write("Classifying...")

    processed_image = preprocess_image(uploaded_file)
    prediction = model.predict(processed_image) # Keras
    # or prediction = model(processed_image_tensor) # PyTorch
```

6. **Displaying Results:** Decode the model's output probabilities. For classification, this usually involves finding the index with the highest probability (np.argmax or torch.max). Map this index to the corresponding Pokemon name using a pre-defined list or dictionary of class names (ensure the order matches the training data). Display the prediction and optionally the confidence score.[22]
   Python
```
# Assuming 'class_names' is a list of Pokemon names in the correct order
# Load class names (e.g., from a file)
# with open('class_names.txt') as f:
#     class_names = [line.strip() for line in f.readlines()]

if uploaded_file is not None:
    #... (preprocessing and prediction code from above)...

    score = tf.nn.softmax(prediction) # Keras example for probabilities
    # or score = torch.softmax(prediction, dim=1) # PyTorch example

    predicted_class_index = np.argmax(score)
    predicted_pokemon = class_names[predicted_class_index]
    confidence = 100 * np.max(score)

    st.success(f"Prediction: {predicted_pokemon}")
    st.write(f"Confidence: {confidence:.2f}%")
```

B. Code Structure (app.py)

A typical app.py file for the Streamlit app would look like this:

Python

```python
import streamlit as st
from PIL import Image
import numpy as np
import tensorflow as tf # or import torch, torchvision.transforms as transforms

# --- Configuration ---
MODEL_PATH = 'pokemon_classifier_mobilenetv2.h5' # Or.pth file
CLASS_NAMES_PATH = 'class_names.txt' # File with one Pokemon name per line
IMAGE_SIZE = (224, 224) # Must match model's expected input size

# --- Model Loading ---
@st.cache_resource
def load_my_model(model_path):
    # Load Keras model
    model = tf.keras.models.load_model(model_path)
    # Or load PyTorch model (requires model class definition)
    # model = YourModelClass(...)
    # model.load_state_dict(torch.load(model_path))
    # model.eval()
    return model

model = load_my_model(MODEL_PATH)

# --- Load Class Names ---
@st.cache_data # Cache the class names list
def load_class_names(file_path):
    try:
        with open(file_path, 'r') as f:
            class_names = [line.strip() for line in f.readlines()]
        return class_names
    except FileNotFoundError:
        st.error(f"Error: Class names file not found at {file_path}")
        return None

class_names = load_class_names(CLASS_NAMES_PATH)

# --- Image Preprocessing ---
def preprocess_image(image_data, target_size):
    try:
        img = Image.open(image_data).convert('RGB') # Ensure 3 channels
        img = img.resize(target_size)
        img_array = np.array(img)

        #!!! CRITICAL: Apply the EXACT SAME normalization as during training!!!
        # Example for MobileNetV2 [-1, 1] scaling:
        preprocessed_img =
tf.keras.applications.mobilenet_v2.preprocess_input(img_array)
        # Example for typical PyTorch normalization:
        # transform = transforms.Compose()
        # preprocessed_img_tensor = transform(img)
        # preprocessed_img = preprocessed_img_tensor.numpy() # If model expects numpy

        # Add batch dimension
        img_batch = np.expand_dims(preprocessed_img, axis=0)
        return img_batch
    except Exception as e:
        st.error(f"Error preprocessing image: {e}")
        return None

# --- Streamlit App Interface ---
st.title("Pokemon Species Classifier Demo")
st.write("Upload an image of a Pokemon (Gen 1) to classify it.")

uploaded_file = st.file_uploader("Choose an image...", type=["jpg", "png", "jpeg"])

if uploaded_file is not None and class_names is not None:
    # Display uploaded image
    image = Image.open(uploaded_file)
    st.image(image, caption='Uploaded Image', use_column_width=True)

    # Preprocess and Predict
    st.write("Classifying...")
    processed_image = preprocess_image(uploaded_file, IMAGE_SIZE)

    if processed_image is not None:
        try:
            # Make prediction
            prediction = model.predict(processed_image) # Keras
            # or prediction = model(torch.from_numpy(processed_image).float()) # PyTorch

            # Get probabilities and predicted class
            # Keras (adjust if output is not logits)
            score = tf.nn.softmax(prediction).numpy()
            # PyTorch (adjust if output is not logits)
            # score = torch.softmax(prediction, dim=1).detach().numpy()

            predicted_class_index = np.argmax(score)
            predicted_pokemon = class_names[predicted_class_index]
            confidence = 100 * np.max(score)

            # Display result
            st.success(f"Prediction: {predicted_pokemon}")
            st.write(f"Confidence: {confidence:.2f}%")

        except Exception as e:
            st.error(f"Error during prediction: {e}")

elif class_names is None:
    st.warning("Could not load class names. Please check the file path.")

st.sidebar.info("This app uses a pre-trained model fine-tuned on Gen 1 Pokemon images.")
```

(Note: This is a template. The preprocess_image function and prediction decoding must be adapted precisely to the chosen model and framework.)

C. Finding Relevant Examples

Fortunately, several developers have shared projects combining Pokemon classification and Streamlit, providing invaluable references:

- **Highly Relevant Pokemon Examples:**
    - GunjanDhanuka/PokeDex_Classifier [11]: Uses TensorFlow/DenseNet201, Streamlit, and PokeAPI. A very complete example.
    - kyle1iao/Pokemon-Models [17]: Contains a classifier and Streamlit app.
    - imjeffhi4/pokemon-classifier [14]: Uses a Vision Transformer (ViT) but demonstrates deployment structure.
- **General Image Classification Streamlit Examples:**
    - TensorFlow: [26]
    - PyTorch: [22] (Especially [22] and [23] using ResNet).

Leveraging these existing projects is highly encouraged. Adapting their structure for model loading, preprocessing, and prediction within Streamlit can significantly accelerate development compared to starting from scratch. The existence of these public repositories demonstrates that the proposed project is feasible and follows a pattern successfully implemented by others.

D. (Optional Enhancement) Adding PokeAPI Details

If the core classification app is functional and time permits, consider enhancing the demo by integrating the PokeAPI. After predicting the Pokemon's name, the app could make an API call using this name to fetch and display additional details like its type(s), a brief description, or official sprites, similar to the approach in.[11] This adds an engaging layer to the demonstration. Libraries like requests in Python can be used for this.

VI. Key Code Snippets and Resource Pointers

This section consolidates critical resources and provides minimal code snippets for key steps.

- **Consolidated Links:**
    - **Recommended Dataset:** "PokemonClassification" / "7,000 Labelled Pokémon" on Kaggle: https://www.kaggle.com/datasets/lantian773030/pokemonclassification
    - **Core Transfer Learning Tutorials:**
        - TensorFlow/Keras: Official TL Guide [6], MobileNetV2 TL Example [5]
        - PyTorch: Official TL Tutorial [1], TL Guide [8]
    - **Key Streamlit Pokemon Examples:**
        - GunjanDhanuka/PokeDex_Classifier [11]: https://github.com/GunjanDhanuka/PokeDex_Classifier
        - kyle1iao/Pokemon-Models [17]: https://github.com/kyle1iao/Pokemon-Models

- **Key General Streamlit Image Classification Examples:**
  - PyTorch/ResNet: 23
  - TensorFlow: 31
- **PokeAPI:** https://pokeapi.co/ (for optional enhancement)

- **Illustrative Code Snippets (Conceptual):**
  - **Keras/TF: Load MobileNetV2 Base, Add Head, Compile**

```python
import tensorflow as tf
IMG_SHAPE = (224, 224, 3)
NUM_CLASSES = 150
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
include_top=False, weights='imagenet')
base_model.trainable = False # Freeze base
model = tf.keras.Sequential()
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
# model.summary() # To view structure
```

  - **PyTorch: Load ResNet Base, Modify Classifier, Define Loss/Optimizer**

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models

NUM_CLASSES = 150
model_ft = models.resnet18(weights='IMAGENET1K_V1') # Or resnet50, etc.
# Freeze all base layers
for param in model_ft.parameters():
    param.requires_grad = False
# Replace the final fully connected layer
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, NUM_CLASSES)
# Define loss and optimizer (only for the new layer)
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.Adam(model_ft.fc.parameters(), lr=0.001) # Train only the
new fc layer
# device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# model_ft = model_ft.to(device)
```

- **Streamlit: File Uploader & Basic Prediction Flow**

  ```python
  import streamlit as st
  from PIL import Image
  # Assume 'model', 'preprocess_image', 'class_names' are defined/loaded

  uploaded_file = st.file_uploader("Choose image...", type=["jpg", "png"])
  if uploaded_file is not None:
    st.image(Image.open(uploaded_file), caption='Uploaded Image.')
    st.write("Classifying...")
    processed = preprocess_image(uploaded_file) # Your preprocessing function
    prediction = model.predict(processed) # Or model(processed_tensor)
    #... decode prediction (argmax, map to class_names)...
    st.success(f"Prediction: {predicted_pokemon_name}")
  ```

- **requirements.txt (Example):**

  ```
  streamlit
  tensorflow # or torch and torchvision
  Pillow
  numpy
  requests # If using PokeAPI
  ```

VII. Conclusion: Demo Day Checklist

This guide has outlined a rapid development path for creating a Pokemon species classification demo within a 24-hour timeframe. The recommended strategy hinges on leveraging existing resources: selecting a manageable Gen 1 dataset from Kaggle [2], applying transfer learning via feature extraction with an efficient pre-trained model like MobileNetV2 [4], implementing essential preprocessing and basic data augmentation [8], and building the interactive interface using Streamlit, drawing inspiration from available examples.[11]

To ensure readiness for the demonstration, consider the following checklist:

- **[ ] Model Trained & Saved:** The transfer learning model has been trained (even for a few epochs) and saved to a file (.h5 or .pth).
- **[ ] Streamlit App (app.py):** The Streamlit script is complete, loads the model, handles image uploads, performs preprocessing *identical* to training, makes predictions, and displays results clearly.
- **[ ] Local Testing:** The Streamlit app runs correctly locally via streamlit run app.py.

- **[ ] Dependencies (requirements.txt):** A requirements.txt file lists all necessary Python packages.
- **[ ] Class Names:** A mechanism (e.g., a text file) is in place for the app to map prediction indices to Pokemon names correctly.
- **[ ] Sample Images:** Have several test images (ideally not from the training set) ready to use during the live demo.
- **[ ] Deployment (Optional but Recommended):** If possible, deploy the app using a service like Streamlit Community Cloud for easy sharing and access during the presentation. Several examples mention deployment.[11]
- **[ ] Talking Points:** Prepare a brief explanation of the project: the goal, the chosen dataset, the use of transfer learning, and how the Streamlit app works.

While the timeline is undeniably challenging, the outlined approach, focusing on established techniques and existing resources, makes creating a compelling and functional Pokemon classification demo achievable. Good luck with the presentation!

Works cited

1. Transfer Learning for Computer Vision Tutorial - PyTorch, accessed April 9, 2025, https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
2. Transfer learning & fine-tuning - Keras, accessed April 9, 2025, https://keras.io/guides/transfer_learning/
3. Basic classification: Classify images of clothing | TensorFlow Core, accessed April 9, 2025, https://www.tensorflow.org/tutorials/keras/classification
4. Finetuning TensorFlow/Keras Networks: Basics Using MobileNetV2 as an Example | by Alfred Weirich | Medium, accessed April 9, 2025, https://medium.com/@alfred.weirich/finetuning-tensorflow-keras-networks-basics-using-mobilenetv2-as-an-example-8274859dc232
5. How to Train MobileNetV2 On a Custom Dataset - Roboflow Blog, accessed April 9, 2025, https://blog.roboflow.com/how-to-train-mobilenetv2-on-a-custom-dataset/
6. Transfer learning and fine-tuning | TensorFlow Core, accessed April 9, 2025, https://www.tensorflow.org/tutorials/images/transfer_learning
7. Transfer Learning Guide: A Practical Tutorial With Examples for Images and Text in Keras, accessed April 9, 2025, https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras
8. PyTorch: Transfer Learning and Image Classification - PyImageSearch, accessed April 9, 2025, https://pyimagesearch.com/2021/10/11/pytorch-transfer-learning-and-image-classification/
9. PyTorch Tutorial 15 - Transfer Learning - YouTube, accessed April 9, 2025, https://www.youtube.com/watch?v=K0lWSB2QoIQ

10. tutorials/beginner_source/transfer_learning_tutorial.py at main · pytorch/tutorials - GitHub, accessed April 9, 2025, https://github.com/pytorch/tutorials/blob/master/beginner_source/transfer_learning_tutorial.py
11. GunjanDhanuka/PokeDex_Classifier: A web app that classifies an image into one of 150 Pokemon using Transfer Learning and Streamlit Framework - GitHub, accessed April 9, 2025, https://github.com/GunjanDhanuka/PokeDex_Classifier
12. How to load mobilenetv2 pretrained model from keras(EASY) - 7A - YouTube, accessed April 9, 2025, https://www.youtube.com/watch?v=traNQHEYzqE
13. Pokédex: Transfer Learning for Image Classification - Robert Aduviri, accessed April 9, 2025, https://robert-aduviri.github.io/portfolio/03-pokedex-image-classification/
14. imjeffhi4/pokemon-classifier - GitHub, accessed April 9, 2025, https://github.com/imjeffhi4/pokemon-classifier
15. MobileNet, MobileNetV2, and MobileNetV3 - Keras, accessed April 9, 2025, https://keras.io/api/applications/mobilenet/
16. Building a Pokémon Image Classifier using Deep Learning in PyTorch - Medium, accessed April 9, 2025, https://medium.com/@filipesampaiocampos/building-a-pok%C3%A9mon-image-classifier-using-deep-learning-in-pytorch-bfa2ce385994
17. kyle1iao/Pokemon-Models: Data Science, Machine Learning, and Deep Learning on Pokemon - GitHub, accessed April 9, 2025, https://github.com/kyle1iao/Pokemon-Models
18. deep convolutional neural network for Pokemon type classification using image data - GitHub, accessed April 9, 2025, https://github.com/rshnn/pokemon-types
19. Training a Classifier — PyTorch Tutorials 2.6.0+cu124 documentation, accessed April 9, 2025, https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
20. When fine-tuning a pre-trained Model, how does tensorflow know that the base_model has been changed? - Stack Overflow, accessed April 9, 2025, https://stackoverflow.com/questions/75365737/when-fine-tuning-a-pre-trained-model-how-does-tensorflow-know-that-the-base-mod
21. yoldemir/pokemon-classifier: Transfer learning example using ResNet50 - GitHub, accessed April 9, 2025, https://github.com/yoldemir/pokemon-classifier
22. Create an Image Classification Web App using PyTorch and Streamlit - Medium, accessed April 9, 2025, https://medium.com/data-science/create-an-image-classification-web-app-using-pytorch-and-streamlit-f043ddf00c24
23. Image-Classification-Web-App-using-PyTorch-and-Streamlit/streamlit_ui.py at main - GitHub, accessed April 9, 2025, https://github.com/denistanjingyu/Image-Classification-Web-App-using-PyTorch-and-Streamlit/blob/main/streamlit_ui.py
24. New gunjandhanuka/pokedex_classifier Streamlit App - Custom Components, accessed April 9, 2025, https://discuss.streamlit.io/t/new-gunjandhanuka-pokedex-classifier-streamlit-app/17560

25. Image classification | TensorFlow Core, accessed April 9, 2025, https://www.tensorflow.org/tutorials/images/classification
26. Image-Classification-Streamlit-TensorFlow/app.py at main - GitHub, accessed April 9, 2025, https://github.com/GSNCodes/Image-Classification-Streamlit-TensorFlow/blob/main/app.py
27. Image classification WEB APP with Python and Streamlit | Pneumonia classifier - YouTube, accessed April 9, 2025, https://www.youtube.com/watch?v=n_eMARPqBZI
28. Training and deploying an image classification model with streamlit. - GitHub, accessed April 9, 2025, https://github.com/pytholic/streamlit-image-classification
29. Use PyTorch to train your image classification model - Microsoft Learn, accessed April 9, 2025, https://learn.microsoft.com/en-us/windows/ai/windows-ml/tutorials/pytorch-train-model
30. How to make imaged based authentication? - Using Streamlit, accessed April 9, 2025, https://discuss.streamlit.io/t/how-to-make-imaged-based-authentication/86873
31. A basic web-app for image classification using Streamlit and Tensorflow - GitHub, accessed April 9, 2025, https://github.com/GSNCodes/Image-Classification-Streamlit-TensorFlow
32. Image Classification with Streamlit and InceptionV3 - GitHub, accessed April 9, 2025, https://github.com/rodrigoguedes09/Image-Classification-with-Streamlit
33. Deep-learning apps for image processing made easy: A step-by-step guide - Streamlit Blog, accessed April 9, 2025, https://blog.streamlit.io/deep-learning-apps-for-image-processing-made-easy-a-step-by-step-guide/
34. alvarobartt/tensorflow-serving-streamlit - GitHub, accessed April 9, 2025, https://github.com/alvarobartt/tensorflow-serving-streamlit
35. This project is an image classification application using PyTorch and Streamlit. The model is trained on the CIFAR-10 dataset and can classify images into 10 different categories - GitHub, accessed April 9, 2025, https://github.com/RaheesAhmed/Image-Classifier
36. denistanjingyu/Image-Classification-Web-App-using-PyTorch-and-Streamlit - GitHub, accessed April 9, 2025, https://github.com/denistanjingyu/Image-Classification-Web-App-using-PyTorch-and-Streamlit
37. damianboh/pokedex: Streamlit web app for searching Pokemon. Filter, chart and display Pokemon stats and search for Pokemon with similar stats. - GitHub, accessed April 9, 2025, https://github.com/damianboh/pokedex