Assignment 4 Computer Vision
Pets Expression Classification
Technical Report

Name	ID
Nadine Mohamed Tamish	8092
Hour Mohamed AbdElaziz	8119
Seif Maged Farouk	8387

Pets Expression Classification: Detailed Function Analysis

Overview

This report comprehensively analyzes Jupyter notebooks implementing a series of deep learning models for pet facial expression classification. The project involves training and evaluating five well-known convolutional neural network architectures such as: ResNet, MobileNet, VGG, InceptionV3, and DenseNet121 from scratch.

ResNet

Function Analysis

1. `class ResidualBlock(tf.keras.Model)`

- **Purpose:** Defines a residual block used in ResNet architectures to allow gradients to flow through skip connections, which helps mitigate the vanishing gradient problem.
- **Parameters:**
  - `filters`: Number of filters for the convolutional layers.
  - `stride`: Stride of the first convolution layer; used to reduce the spatial dimensions.
  - `downsample`: Optional downsampling layer applied to the identity connection to match the shape of the output.
- **Implementation Details:**
  - Contains two 3x3 convolutional layers with batch normalization and ReLU activation.
  - Optionally downsamples the identity input using a convolution if dimensions differ.
  - A dropout layer (with 30% drop rate) adds regularization after the second batch normalization.
  - The residual connection (`identity`) is added to the processed output before the final activation.

- **Why It's Used:** Residual connections enable the training of deeper networks by preserving gradient flow. This helps the network learn identity mappings more easily, improving convergence and overall accuracy.

2. `def make_layer(filters, blocks, stride=1)`

- **Purpose:** Creates a stack of residual blocks to form one layer of the ResNet architecture.
- **Parameters:**

    - `filters`: Number of filters for each residual block in the layer.
    - `blocks`: Number of residual blocks in this layer.
    - `stride`: Stride of the first block; used for downsampling.

- **Implementation Details:**
    - If the stride is not 1, a downsampling layer is created using a 1x1 convolution and batch normalization to match the dimensions.
    - Initializes the first `ResidualBlock` with the provided stride and optional downsample layer.
    - Adds subsequent `ResidualBlock` instances with stride=1 (no further downsampling).
- **Why It's Used:** This function modularizes layer construction, allowing the creation of variable-depth ResNet models by stacking blocks of residual units.

3. `class ResNet18(tf.keras.Model)`

- **Purpose:** Implements the full ResNet-18 model architecture using stacks of residual blocks.
- **Parameters:**
    - `num_classes`: Number of output classes for the classification task.
- **Implementation Details:**
    - Starts with a 7x7 convolution followed by batch normalization, ReLU, and 3x3 max pooling (standard ResNet stem).
    - Composed of four layers with increasing filter sizes (64, 128, 256, 512), each created by `make_layer`.
    - Applies global average pooling to reduce feature maps to a single vector per sample.
    - Ends with a fully connected layer for classification.

- **Why It's Used:** ResNet-18 is a deep convolutional neural network with skip connections, allowing efficient training of deeper models and achieving high accuracy on image classification tasks.

InceptionV3

Function Analysis

1. `conv_bn_relu(x, filters, kernel_size, strides=(1, 1), padding='same', weight_decay=0.00004)`

- **Purpose**: Applies a convolution followed by batch normalization and ReLU activation.

- **Parameters**:
  - `x`: Input tensor.
  - `filters`: Number of output channels.
  - `kernel_size`: Size of the convolutional filter.
  - `strides`: Stride of the convolution (default `(1,1)`).
  - `padding`: Padding type (`'same'` or `'valid'`).
  - `weight_decay`: L2 regularization factor (default `0.00004`).
- **Implementation Details**:
  - Uses `Conv2D` with no bias and L2 kernel regularization.
  - Applies `BatchNormalization` to normalize activations.
  - Uses `ReLU` for non-linearity.
- **Why It's Used**: Serves as the fundamental unit for convolutional feature extraction, normalization, and activation in Inception modules.

2. `inception_a(x)`

- **Purpose**: Implements the Inception-A module, which combines multiple convolutional branches.
- **Parameters**:
  - `x`: Input tensor.
- **Implementation Details**:
- Four branches:
  - 1x1 convolution.
  - 1x1 → 5x5 convolution.
  - 1x1 → 3x3 → 3x3 convolution.
  - 3x3 average pooling → 1x1 convolution.
- Outputs are concatenated along the channel axis.
- **Why It's Used**: Captures features at multiple receptive fields with minimal computational cost using factorized convolutions.

3. `reduction_a(x)`

- **Purpose**: Reduces spatial dimensions while increasing depth.
- **Parameters**:
  - `x`: Input tensor.
- **Implementation Details**:
  - Three branches:
    - 3x3 stride-2 convolution.
    - 1x1 → 3x3 → 3x3 stride-2 convolution.

        - 3x3 max pooling with stride 2.
    - Outputs are concatenated.

- **Why It's Used**: Performs downsampling to reduce feature map size and prepare for deeper layers while maintaining feature diversity.

4. `inception_b(x)`

- **Purpose**: Implements the Inception-B module with asymmetric convolutions.
- **Parameters**:
    - `x`: Input tensor.
- **Implementation Details**:
    - Four branches:
        - 1x1 convolution.
        - 1x1 → 1x7 → 7x1 convolution.
        - 1x1 → 7x1 → 1x7 → 7x1 → 1x7 convolution.
        - 3x3 average pooling → 1x1 convolution.
- All branches are concatenated.

- **Why It's Used**:
  Improves model efficiency by using asymmetric convolutions to expand the receptive field while reducing computation.

5. `reduction_b(x)`

- **Purpose**: Reduces spatial dimensions while increasing depth.
- **Parameters**:
    - `x`: Input tensor.
- **Implementation Details**:
    - Three branches:
        - 1x1 → 3x3 stride-2 convolution.
        - 1x1 → 1x7 → 7x1 → 3x3 stride-2 convolution.
        - 3x3 max pooling with stride 2.
- Outputs are concatenated.
- **Why It's Used**: Downsamples feature maps efficiently while capturing high-level spatial patterns.

6. `inception_c(x)`

- **Purpose**: Reduces spatial dimensions while increasing depth.
- **Parameters**:
    - `x`: Input tensor.
- **Implementation Details**:
    - Four branches:

        - 1x1 convolution.
        - 1x1 → parallel (1x3 and 3x1) convolutions.
        - 1x1 → 3x3 → parallel (1x3 and 3x1) convolutions.
        - 3x3 average pooling → 1x1 convolution.
- Concatenates all outputs.
- **Why It's Used**: Allows the model to learn a richer set of features by splitting filters across different convolutional shapes.

7. `build_inception_v3(input_shape=(299, 299, 3), num_classes=1000)`

- **Purpose**: Builds the complete InceptionV3 model from individual modules.
- **Parameters**:
    - `input_shape`: Shape of input images (default `(299, 299, 3)`).
    - `num_classes`: Number of output classes (default `1000`).
- **Implementation Details**:
    - **Stem**: Several convolutions and pooling layers to preprocess inputs.
    - **Body**:
        - 3x Inception-A modules.
        - 1x Reduction-A.
        - 4x Inception-B modules.
        - 1x Reduction-B.
        - 2x Inception-C modules.
    - **Final Layers**: Global average pooling, dropout, and dense softmax classification.
- **Why It's Used**: Implements the full InceptionV3 architecture for high-performance image classification with reduced computational cost through modular design.

DenseNet121

Function Analysis

1. `conv_block(x, growth_rate)`

- **Purpose**: Implements a single composite function used in DenseNet, applying bottleneck and convolutional operations.
- **Parameters**:
    - `x`: Input tensor.
    - `growth_rate`: Controls the number of filters added to the input tensor during the convolution.
- **Implementation Details**:
    - Applies Batch Normalization and ReLU activation.
    - Uses a `1x1` convolution (bottleneck layer) to reduce dimensionality and computation.
    - Follows with a `3x3` convolution to extract features.

    - Concatenates the output (`x1`) with the input (`x`) to implement dense connectivity.

- **Why It's Used**: This structure encourages feature reuse, reduces the number of parameters, and strengthens gradient flow by allowing the input to pass directly to subsequent layers.

2. `dense_block(x, num_layers, growth_rate)`

- **Purpose**: Stacks multiple `conv_block`s to form a dense block as described in the DenseNet architecture.
- **Parameters**:
    - `x`: Input tensor.
    - `num_layers`: Number of convolutional layers in the block.
    - `growth_rate`: Number of filters to add per layer.
- **Implementation Details**: Iteratively applies `conv_block`, each time expanding the number of feature maps.
- **Why It's Used**: This is the core component of DenseNet, where every layer receives input from all preceding layers, improving information flow and efficiency.

3. `transition_layer(x, compression=0.5)`

- **Purpose**: Reduces the number of feature maps and spatial dimensions between dense blocks.
- **Parameters**:
    - `x`: Input tensor.
    - `compression`: Compression factor for reducing the number of output channels (default 0.5).
- **Implementation Details**:
    - Applies Batch Normalization and ReLU.
    - Uses a `1x1` convolution to reduce feature maps.
    - Applies average pooling to downsample the feature maps spatially.
- **Why It's Used**:
  To prevent feature explosion and overfitting by compressing feature maps and reducing computational load between dense blocks.

4. `DenseNet121_Custom(input_shape=(224, 224, 3), num_classes=7, growth_rate=32)`

- **Purpose**: Constructs the full DenseNet-121 architecture using the previously defined building blocks.
- **Parameters**:
    - `input_shape`: Shape of input images (default `(224, 224, 3)`).
    - `num_classes`: Number of output classes for classification (default `7`).
    - `growth_rate`: Number of filters added per layer in dense blocks (default `32`).
- **Implementation Details**:

    - Begins with an initial `7x7` convolution and `3x3` max pooling.
    - Stacks 4 dense blocks with a configuration `[6, 12, 24, 16]` layers respectively (DenseNet-121 layout).
    - Inserts transition layers between dense blocks (except the last).
    - Ends with Batch Normalization, ReLU, global average pooling, and a dense softmax classifier.
- **Why It's Used**: Replicates the official DenseNet-121 architecture for image classification tasks, allowing deep supervision and efficient parameter usage through dense connectivity.

MobileNet Model

1. DepthwiseSeparableConv(Layer)

- **Purpose**: Implements the core building block of MobileNet, replacing standard convolution with a computationally efficient depthwise separable convolution.
- **Parameters**:
    - `out_channels`: Number of output feature maps/channels
    - `kernel_size`: Size of the convolution kernel (default 3)
    - `stride`: Stride value for downsampling (default 1)
- **Implementation Details**:
    - **Depthwise Convolution**: Applies a single 3x3 filter to each input channel independently using `DepthwiseConv2D`
    - **Pointwise Convolution**: Uses 1x1 convolution (`Conv2D`) to combine features across channels
    - **Normalization & Activation**: Each convolution is followed by `BatchNormalization` and `ReLU` activation
    - **Regularization**: Applies L2 regularization (1e-4) to prevent overfitting
    - **Weight Initialization**: Uses `he_normal` initialization optimal for ReLU activations
- **Why It's Used**: Reduces computational cost by ~8-9x compared to standard convolution while maintaining similar representational power. Separates spatial filtering from channel mixing for efficiency.

2. MobileNet_customed(tf.keras.Model)

- **Purpose**: Constructs the complete MobileNet architecture using depthwise separable convolutions for efficient image classification.
- **Parameters**:
    - `num_classes`: Number of output classes for classification (set to 4 for pet facial expressions)
- **Implementation Details**:
    - **Initial Convolution**: Standard 3x3 convolution with 32 filters and stride=2 for initial feature extraction

    - **Progressive Downsampling**: Series of depthwise separable blocks with increasing channels:
        - 32→64 (224×224), 64→128 (112×112), 128→256 (56×56), 256→512 (28×28), 512→1024 (14×14)
    - **Repeated Blocks**: 5 identical DepthwiseSeparableConv blocks with 512 filters for deep feature learning
    - **Classification Head**: `GlobalAveragePooling2D` followed by `Dense` layer with softmax activation
    - **Batch Normalization**: Applied after initial convolution for training stability
- **Why It's Used**: Provides efficient mobile-friendly architecture that balances accuracy with computational constraints, suitable for resource-limited environments.

Model Compilation Analysis

3. model.compile() Configuration

- **Purpose**: Configures the model for training by setting optimizer, loss function, and evaluation metrics.
- **Parameters**:
    - **Optimizer**: `Adam` with custom hyperparameters
        - `learning_rate=0.0001`: Conservative rate for stable convergence
        - `beta_1=0.9`: Momentum parameter for first moment estimates
        - `beta_2=0.999`: Decay rate for second moment estimates
        - `epsilon=1e-7`: Numerical stability constant
    - **Loss Function**: `categorical_crossentropy` for multi-class classification
    - **Metrics**: `accuracy` for performance evaluation
- **Implementation Details**:
    - Adam optimizer adapts learning rates per parameter using momentum and squared gradients
    - Categorical crossentropy computes cross-entropy between predicted probabilities and one-hot encoded labels
    - Lower learning rate prevents overshooting in complex architecture
- **Why It's Used**: Adam optimizer handles sparse gradients well and adapts to non-stationary objectives. Conservative learning rate ensures stable training for this deep architecture.

Training Process Analysis

4. model.fit() Multi-Phase Training

- **Purpose**: Trains the model using a multi-phase approach with data generators for augmented training data.
- **Parameters**:
    - `train_generator`: Augmented training data with transformations
    - `val_generator`: Validation data with only rescaling

    - `steps_per_epoch`: `train_generator.samples // batch_size` - ensures all samples seen per epoch
    - `validation_steps`: `val_generator.samples // batch_size` - validation batches per epoch
    - `epochs`: Multiple phases (30 → 10 → 10) for progressive training
- **Implementation Details**:
    - **Phase 1**: 30 epochs for initial feature learning and convergence
    - **Phase 2**: 10 epochs for fine-tuning and performance optimization
    - **Phase 3**: 10 epochs for final refinement and stability
    - **Data Augmentation**: Training uses rotation, shifts, zoom, and flips for robustness
    - **Validation**: Clean validation data without augmentation for unbiased evaluation
- **Why It's Used**: Multi-phase training allows monitoring convergence stages, prevents overfitting, and enables manual intervention. Data augmentation improves generalization by exposing model to varied transformations.

Architecture Flow Summary

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Key Design Principles:

- **Efficiency**: Depthwise separable convolutions reduce parameters by ~8x
- **Progressive Downsampling**: Gradual spatial reduction with channel expansion
- **Feature Reuse**: BatchNormalization and residual-like connections in DS blocks

- **Robust Training**: Multi-phase approach with comprehensive data augmentation

InceptionV3 Transfer Learning

create_inception_transfer_model(num_classes)

- **Purpose**: Creates a transfer learning model using pre-trained InceptionV3 as feature extractor with custom classification head for pet facial expression recognition.
- **Parameters**:
  - `num_classes`: Number of output classes for classification (4 for pet expressions)
- **Implementation Details**:
  - **Base Model Loading**: Uses `InceptionV3(weights='imagenet', include_top=False)` pre-trained on ImageNet
  - **Input Shape**: Configures for (299, 299, 3) - InceptionV3's required input dimensions
  - **Feature Freezing**: Sets `base_model.trainable = False` to freeze pre-trained weights initially
  - **Custom Classification Head**:
    - `GlobalAveragePooling2D()`: Reduces spatial dimensions from feature maps to 1D vector
    - `Dense(512, activation='relu')`: First fully connected layer with 512 neurons
    - `Dropout(0.5)`: High dropout rate to prevent overfitting
    - `Dense(256, activation='relu')`: Second FC layer with 256 neurons
    - `Dropout(0.3)`: Moderate dropout for regularization
    - `Dense(num_classes, activation='softmax')`: Final classification layer
  - **Model Creation**: Uses Functional API to connect base model with custom head
- **Why It's Used**: Leverages powerful ImageNet-trained features while adapting to specific pet expression classification task. Two-stage approach allows stable feature extraction followed by fine-tuning.

Training Configuration Analysis

2. Data Generator Setup

- **Purpose**: Configures data preprocessing and augmentation pipelines optimized for InceptionV3 architecture.
- **Parameters**:

    - `img_size = 299`: InceptionV3's required input size (different from typical 224×224)
    - `batch_size = 32`: Balanced batch size for memory efficiency and stable gradients
    - `target_size = (299, 299)`: Ensures all images match InceptionV3 input requirements
- **Implementation Details**:
    - **Training Augmentation**:
        - `rotation_range=30`: Random rotations up to ±30 degrees
        - `width_shift_range=0.2, height_shift_range=0.2`: Spatial translations
        - `shear_range=0.2, zoom_range=0.2`: Geometric transformations
        - `horizontal_flip=True`: Mirror augmentation (appropriate for facial expressions)
        - `fill_mode='nearest'`: Pixel filling strategy for transformed regions
    - **Validation/Test Processing**: Only rescaling (`rescale=1./255`) without augmentation
    - **Class Mode**: `categorical` for one-hot encoded multi-class labels
    - **Shuffling**: `shuffle=True` for training/validation, `shuffle=False` for test (consistent evaluation)
- **Why It's Used**: InceptionV3's larger input size (299×299) requires specific preprocessing. Augmentation improves generalization while maintaining evaluation consistency.

Two-Stage Training Analysis

3. Stage 1: Feature Extraction Training

- **Purpose**: Trains only the custom classification head while keeping pre-trained InceptionV3 features frozen.
- **Parameters**:
    - **Optimizer**: `Adam(learning_rate=0.001)` - Standard learning rate for new layers
    - **Loss Function**: `categorical_crossentropy` for multi-class classification
    - **Epochs**: 15 epochs for initial feature extraction learning
    - **Frozen Parameters**: Base model weights remain unchanged
- **Implementation Details**:
    - `base_model.trainable = False`: Prevents updating of pre-trained weights
    - Only classification head parameters are trainable (~1.3M vs 23M total parameters)
    - Higher learning rate suitable since only training new randomly initialized layers
    - Focuses on learning optimal combination of pre-trained features for new task

- **Why It's Used**: Stable initial training phase that adapts pre-trained features to new domain without destroying valuable ImageNet representations. Prevents catastrophic forgetting of pre-trained knowledge.

4. Stage 2: Fine-tuning

- **Purpose**: Unfreezes and fine-tunes upper layers of pre-trained model for task-specific feature adaptation.
- **Parameters**:
    - **Unfreezing Strategy**: `fine_tune_at = len(base_model.layers) // 2` - unfreezes top 50% of layers
    - **Optimizer**: `Adam(learning_rate=0.0001/10)` - Very low learning rate (0.00001)
    - **Epochs**: 10 epochs for careful fine-tuning
    - **Layer Selection**: Keeps lower layers frozen, unfreezes higher-level feature layers
- **Implementation Details**:

Selective Unfreezing:
python

```
for layer in base_model.layers[:fine_tune_at]:
```

    - `    layer.trainable = False`

    - **Dramatic Learning Rate Reduction**: 100x lower than Stage 1 to prevent destroying pre-trained features
    - **Progressive Adaptation**: Allows high-level features to adapt while preserving low-level representations
    - **Increased Trainable Parameters**: From ~1.3M to ~13M parameters
- **Why It's Used**: Fine-tuning allows adaptation of pre-trained features to specific task characteristics while maintaining stability through very low learning rates. Upper layers contain more task-specific features suitable for adaptation.

Architecture Flow Summary

Stage 1 - Feature Extraction (15 epochs): Input (299×299×3) → **Frozen InceptionV3** → Features → **Trainable Head** → 4 classes

- Learning Rate: 0.001
- Trainable Parameters: ~1.3M
- Focus: Learning optimal feature combination

Stage 2 - Fine-tuning (10 epochs): Input (299×299×3) → **Partially Frozen InceptionV3** → **Trainable Upper Layers** → **Head** → 4 classes

- Learning Rate: 0.00001 (100x lower)
- Trainable Parameters: ~13M

- Focus: Task-specific feature adaptation

Key Design Principles

Transfer Learning Strategy:

- **Feature Reuse**: Leverages powerful ImageNet representations for visual understanding
- **Domain Adaptation**: Custom head learns pet-specific facial expression patterns
- **Catastrophic Forgetting Prevention**: Frozen training prevents loss of pre-trained knowledge

Two-Stage Training Benefits:

- **Stability**: Initial frozen training establishes stable classification head
- **Gradual Adaptation**: Progressive unfreezing allows careful feature tuning
- **Learning Rate Management**: Different rates optimize new vs. pre-trained parameters
- **Overfitting Prevention**: Conservative approach reduces risk of poor generalization

Architecture Advantages:

- **Computational Efficiency**: Reuses expensive feature extraction computation
- **Data Efficiency**: Requires less training data compared to training from scratch
- **Performance**: Combines ImageNet's broad visual knowledge with task-specific learning
- **Flexibility**: Easy to adapt to different numbers of classes or similar vision tasks