Task 1 - Building a CNN from Scratch

The Oxford-IIIT Pet Dataset is a popular dataset for image classification and segmentation tasks, featuring 37 breeds of pets (dogs and cats) with various poses and backgrounds. It contains around 7,000 labeled images, and each image is annotated with the breed label.

In this analysis, we will dive deep into the dataset and address an image classification challenge. In the first part, we will focus on a binary classification task, distinguishing between dogs and cats. In the second part, we will tackle a multiclass classification task, where the goal will be to classify the images into one of the 37 different breeds.

Since the dataset contains mixed images, it's necessary to organize them by creating the appropriate directories to begin the analysis. The first step is to create the correct directories for training and validation. This should be done for both the Dog/Cat classification, where images starting with a lowercase letter represent dogs and images starting with an uppercase letter represent cats. Additionally, directories should be created for each breed type, which totals 37 breeds, extracted from the image filenames. The dataset will be split into 70% for training and 30% for validation, ensuring the model has sufficient data for both training and performance evaluation.

It is important to be cautious, as the dataset contains not only images but also other types of files. Specifically, there are files with extensions such as '.jpg' and '.mat', which should be properly filtered to ensure only image files are processed.

Additionally, there is an imbalance between the number of dog and cat images, with the number of dog images being nearly twice that of the cat images. This imbalance could negatively affect the model's performance, particularly during training. To address this, it is advisable to implement data augmentation and adjust for the class imbalance by calculating class weights, to increase the variance and fairness in the dataset.

Step 1 – Binary Classification

CNN Architecture and Development

The CNN may contain some numbers of convolutional modules and some fully connected layers. In the picture, the example contains two convolution modules (convolution + ReLU + pooling) for feature extraction, and two fully connected layers for classification.

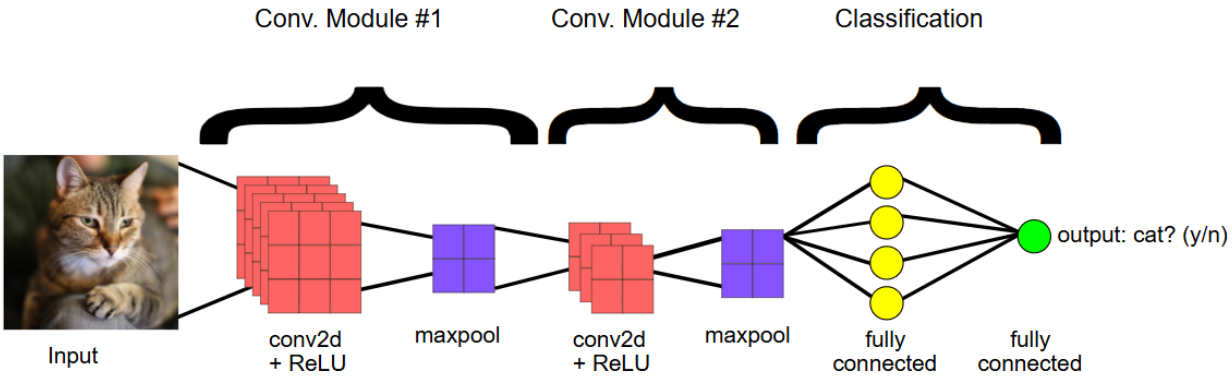


Figure 1 - Example of CNN simple structure

In the architecture implemented, we have three modules, each consisting of three layers:

- Convolution: this layer operates on 3x3 windows to extract local features from the input image.
- ReLU (Rectified Linear Unit): this activation function introduces non-linearity into the model, enabling it to learn more complex patterns.
- Max Pooling: this layer operates on 2x2 windows, reducing the spatial dimensions of the feature map while retaining the most important features.

```
# Our input feature map is 150x150x3: 150x150 for the image pixels, and 3 for
# the three color channels: R, G, and B
img_input = layers.Input(shape=(150, 150, 3))

# First convolution extracts 16 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(16, 3, activation='relu')(img_input)
x = layers.MaxPooling2D(2)(x)

# Second convolution extracts 32 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)

# Third convolution extracts 64 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
```

Figure 2 - CNN architecture implemented

```
# Flatten feature map to a 1-dim tensor so we can add fully connected layers
x = layers.Flatten()(x)

# Create a fully connected layer with ReLU activation and 512 hidden units
x = layers.Dense(512, activation='relu')(x)

# Create output layer with a single node and sigmoid activation
output = layers.Dense(1, activation='sigmoid')(x)

# Create model:
# input = input feature map
# output = input feature map + stacked convolution/maxpooling layers + fully
# connected layer + sigmoid output layer
model = Model(img_input, output)
```

Figure 3 - Fully connected layers implemented

At the top of the convolutional layers, two fully-connected layers are added. Since this is a binary classification problem, the final layer uses a Sigmoid activation function, which outputs a single scalar value between 0 and 1.

The next picture illustrates how the size of the feature map changes as it progresses through each layer of the network. Convolutional layers slightly reduce the size of the feature maps, depending on the padding applied. Meanwhile, each pooling layer typically reduces the size of the feature

map by half, helping to downsample and retain the most important features for further processing.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 150, 150, 3)	0
conv2d (Conv2D)	(None, 148, 148, 16)	448
max_pooling2d (MaxPooling2D)	(None, 74, 74, 16)	0
conv2d_1 (Conv2D)	(None, 72, 72, 32)	4,640
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 32)	0
conv2d_2 (Conv2D)	(None, 34, 34, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 64)	0
flatten (Flatten)	(None, 18496)	0
dense (Dense)	(None, 512)	9,470,464
dense_1 (Dense)	(None, 1)	513

Total params: 9,494,561 (36.22 MB)
Trainable params: 9,494,561 (36.22 MB)
Non-trainable params: 0 (0.00 B)

Figure 4 - Model architecture

The model takes in images of size 150x150 pixels with three color channels (RGB) as input. The first layer of the network is a convolutional layer (Conv2D) that applies convolution operations to extract features from the input image. The first convolutional layer uses 16 filters, the second uses 32 filters, and the third uses 64 filters. As we progress deeper into the network, the number of filters increases to capture more complex features from the images.

Following each convolutional layer, there are max-pooling layers (MaxPooling2D) that reduce the spatial dimensions (height and width) of the feature maps. Pooling helps to reduce the computational cost and also prevents overfitting by keeping only the most important features of the image.

Once the features have been extracted through the convolution and pooling layers, the output is flattened into a 1D vector using a Flatten layer, which allows it to be passed to the fully connected layers.

The model ends with two fully connected layers (Dense). The first fully connected layer has 512 units, and the final layer has one unit with a Sigmoid activation function. This last layer outputs a scalar value between 0 and 1, representing the probability that the image belongs to a specific class, making it suitable for binary classification.

The model contains 9,494,561 parameters, which include weights and biases that the model will adjust during training in order to improve its performance. All these parameters are trainable,

meaning the model will modify them as it learns from the training data. The architecture of the model is designed to progressively reduce the image dimensions while increasing the depth of the network to capture more complex patterns. Finally, fully connected layers are used to classify the features extracted from the image.

Model configuration

In this section, we are configuring the specifications for training the Convolutional Neural Network (CNN). The model is set up for a binary classification task, where we aim to classify images into two classes, so the loss function chosen is binary_crossentropy. This loss function is ideal for binary classification problems when the output is a single value between 0 and 1, as it is in our case with a sigmoid activation function at the output layer.

We use the RMSprop optimizer with a learning rate of 0.001. RMSprop is a variant of stochastic gradient descent (SGD) that adjusts the learning rate automatically during training, making it more efficient and often more stable than standard SGD. It is particularly well-suited for deep learning tasks where the gradients can vary widely. Other optimizers like Adam and Adagrad also adjust the learning rate in a similar manner and would work well in this case as well.

Finally, we specify that during training, we will monitor the classification accuracy metric. This helps us track how well the model is performing in terms of correctly classifying the images.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['accuracy'])
```

Figure 5 - Model compile

To summarize, we are configuring the model with binary_crossentropy as the loss function, using the RMSprop optimizer with a learning rate of 0.001, and tracking accuracy to monitor the performance during training.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size=(150, 150),  # All images will be resized to 150x150
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

# Flow validation images in batches of 20 using val_datagen generator
validation_generator = val_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

Figure 6 - Data generator

In order to set up data generators to load images from our source folders and convert them into float32 tensors, which will be fed into the neural network along with their labels. We will use one generator for training images and another for validation images. Each generator will provide batches of 20 images, resized to 150x150 pixels, with their binary labels.

To prepare the data, we will normalize the images by scaling the pixel values from the original range of [0, 255] to [0, 1]. This step is important because normalized data helps the neural network perform better.

Preventing Overfitting

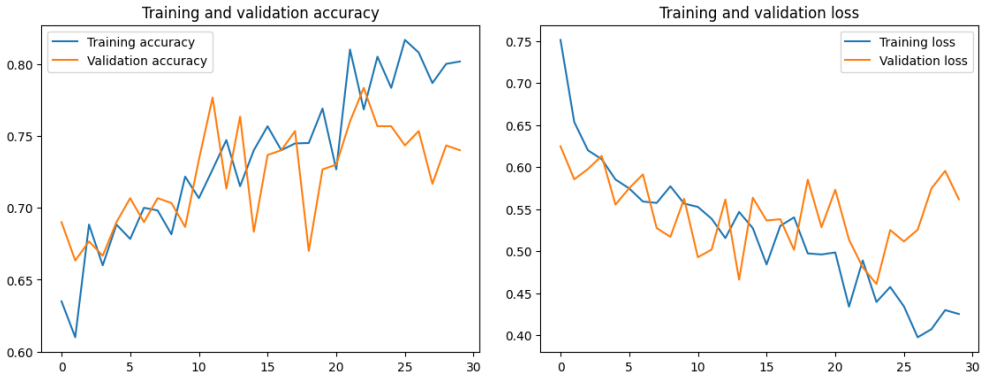
A major concern when training any machine learning model, including convolutional neural networks (CNNs), is overfitting. Overfitting occurs when a model becomes overly tailored to the training data, failing to generalize well to new, unseen examples. To mitigate this issue, several effective techniques can be applied during the development of a CNN:

- **Data Augmentation**: This technique artificially increases the diversity and quantity of training examples by applying random transformations to the existing images. These transformations can include rotations, scaling, translations, and flips, creating new variants of the original images. Data augmentation is particularly beneficial when the training dataset is relatively small, as it helps the model learn more robust features that can generalize better to new data.
- **Dropout Regularization**: Dropout is a technique where, during each training step, random units (neurons) are "dropped" or temporarily removed from the network. This prevents the model from becoming too reliant on specific neurons, promoting a more generalized and robust learning process. Dropout helps reduce overfitting by forcing the model to learn redundant representations across different neurons, ensuring that it does not memorize the training data.
- **L2 Regularization (Weight Decay)**: L2 regularization, also known as weight decay, penalizes large weights by adding an additional term to the loss function that is proportional to the sum of the squared weights. This discourages overly complex models with large weight values, promoting simpler models that generalize better to unseen data. By controlling the magnitude of the weights, L2 regularization reduces the risk of overfitting.
- **Batch Normalization**: Batch Normalization normalizes the activations of each layer by adjusting and scaling them during training. This stabilizes and accelerates learning, making the network less sensitive to the initial weight values and improving generalization. Additionally, Batch Normalization acts as a regularizer by introducing a small amount of noise during training, reducing the model’s dependence on specific neuron activations and helping prevent overfitting.
- **ReduceLROnPlateau**: This adaptive learning rate scheduling technique reduces the learning rate when the model's performance (e.g., validation loss) stops improving for a defined number of epochs. By lowering the learning rate at appropriate times, the model avoids converging too quickly to a suboptimal solution and instead fine-tunes the learned features for better generalization. ReduceLROnPlateau is particularly useful in combination with other regularization methods, ensuring a smoother training process.

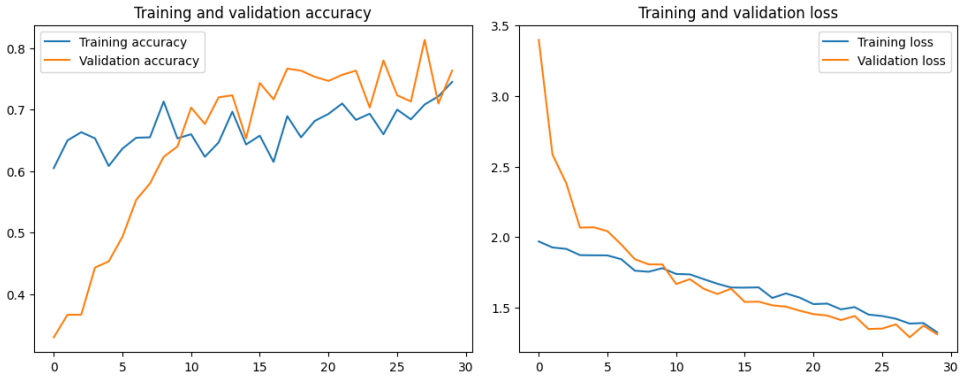
By integrating these techniques, CNNs can achieve better generalization, reducing the likelihood of overfitting while maintaining strong predictive performance on unseen data.

Evaluating Accuracy and Loss for the Model in Binary Classification

In the development of a machine learning model, the evaluation phase is crucial for understanding how well the model is learning and generalizing the data. Accuracy tells us how often the model makes correct predictions, while loss quantifies the difference between the model's predictions and the actual values. By observing the evolution of these metrics over time, we can identify if the model is improving or overfitting.



In this case, in the images above, the model was affected by overfitting over time, getting divergence. By applying techniques such as adjusting class weights to address the dataset imbalance, where there are twice as many dog images as cat images, it is possible to improve the model's performance and overcome this issue.



The observed convergence of accuracy and the consistent decrease in loss over epochs indicate that the implemented regularization techniques—Data Augmentation, Dropout, L2 Regularization, Batch Normalization, and ReduceLROnPlateau—are effectively mitigating overfitting and enhancing the model’s generalization capabilities. This suggests that the model is learning meaningful patterns rather than memorizing the training data, thereby improving its performance on unseen samples.

Step 2 – Multiclass Classification

Once the directories for training and validation are set up with 70% of the data allocated for training and 30% for validation, we can proceed with the dataset preparation.

For this multiclass classification task, the output layer of the model must have one neuron per class (37 neurons in total), and the activation function should be softmax to ensure that the output represents a probability distribution across the classes.

Next, we need to adjust the model's compilation parameters. For multiclass classification, the categorical_crossentropy loss function should be used, and we will monitor categorical_accuracy to assess the model's performance.

The images will be processed using the ImageDataGenerator class, which will rescale the pixel values to a range between 0 and 1 (by dividing them by 255). Two separate generators will be created: one for training and one for validation. These generators will handle image resizing and ensure that labels are encoded as one-hot vectors.

To further improve the model and reduce overfitting, techniques such as data augmentation and dropout regularization should be implemented. Data augmentation will increase the diversity of the training set by applying random transformations to the images, while dropout will help prevent overfitting by randomly dropping a fraction of the neurons during training.

These strategies will help ensure that the model generalizes better to new, unseen data.

```
# Data Augmentation for the training set
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,  # Rotate images randomly up to 30 degrees
    width_shift_range=0.2,  # Translate horizontally by up to 20% of the width
    height_shift_range=0.2,  # Translate vertically by up to 20% of the height
    shear_range=0.2,  # Apply shearing transformations
    zoom_range=0.2,  # Apply random zoom
    horizontal_flip=True,  # Flip images horizontally
    fill_mode='nearest'  # Fill in missing pixels
)
```

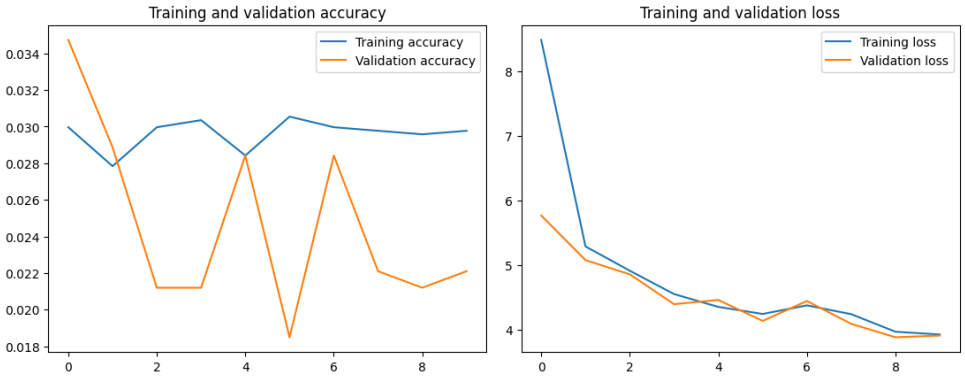
Figure 7 - Data augmentation in multiclass classification

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Dropout(0.3),  # Dropout layer to prevent overfitting

    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Dropout(0.3),

    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.4),  # Higher dropout before the final layer
    Dense(37, activation='softmax')  # 37 classes, softmax activation
])
```

Figure 8 - Dropout in multiclass classification

Evaluating Accuracy and Loss for the Model in Multiclass Classification

In this scenario, the poor performance arises from the small size of the dataset and a large number of categories (37 breeds). The model's low training accuracy (~30%) and unstable validation accuracy suggest it's underfitting, meaning it's not learning enough from the data, even if improvement techniques are in place. Each breed folder created does not contain a sufficient number of samples to effectively train the model. Even with techniques such as data augmentation or dropout, no significant improvements are achieved. This is why Task 2 is introduced in the next chapter, where the main idea is to use a pre-trained model on a large dataset and apply it to the current dataset.



The advantage of leveraging a pre-trained model is that it has already learned generalizable features from a much larger and diverse dataset. These features can be transferred to the current problem, allowing the model to adapt more efficiently, even with a smaller dataset. Fine-tuning the pre-trained model on the specific task at hand can lead to better performance and improved accuracy, as it capitalizes on the knowledge gained from the large dataset, while also tailoring the model to the unique characteristics of the new data.

Task 2 - Transfer Learning with Pretrained Models

In this task, the objective is to work with the dataset under analysis (https://www.kaggle.com/datasets/tanlikesmath/the-oxfordiiit-pet-dataset/data), which contains images of 37 different breeds, by implementing a pre-trained model that has been trained on a different and larger dataset.

During the training process, it is essential to optimize hyperparameters to enhance the model's performance. This involves adjusting the learning rate, batch size, and other configurations to achieve the highest possible accuracy in classification.

Feature Extraction Using a pre-trained Model

The selected pre-trained model is Inception V3, a widely used Convolutional Neural Network (CNN) architecture for image classification tasks. It has been pre-trained on ImageNet, a large-scale dataset containing 1.4 million images across 1,000 classes. ImageNet includes a diverse range of categorized images, enabling the model to learn general and universal features that can be applied to classification tasks on new datasets.

The goal is to extract intermediate representations (features). These representations can be highly informative, even if the task is quite different from the one the original model was trained on. This demonstrates the adaptability of convolutional neural networks. The pre-trained model provides weights trained on a large dataset of images. Although the pre-trained model is not specifically trained on the target dataset, the weights can still serve as a useful starting point.

The first step is to import the InceptionV3 model from TensorFlow's Keras applications module and initialize it without the top classification layers.

```
from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'
pre_trained_model = InceptionV3(
    input_shape=(150, 150, 3), include_top=False, weights=None)
pre_trained_model.load_weights(local_weights_file)
```

Figure 9 - Import the InceptionV3 model

Setting include_top=False removes the fully connected (top) layers of the model, allowing it to function solely as a feature extractor. It means making the model non-trainable, the weights can not be updated during training.

```
for layer in pre_trained_model.layers:
  layer.trainable = False
```

Figure 10 - Model non-trainable

The next step is to choose which intermediate layer of Inception V3 to use for feature extraction. A common approach is to use the output from the last layer before the Flatten operation, known as the "bottleneck layer." The rationale behind this is that the subsequent fully connected layers are highly specialized for the original task the network was trained on, and as a result, the features learned by these layers may not be very useful for a new task. In contrast, the bottleneck features tend to retain more general characteristics.

The selected layer for feature extraction is mixed7. Although it is not the actual bottleneck of the network, it is chosen to preserve a sufficiently large 7×7 feature map, ensuring a richer representation of spatial features. In contrast, utilizing the bottleneck layer would yield a 3×3 feature map, which may be too small to capture adequate information for effective feature extraction.

In summary, the output of a neural network layer represents the extracted features from the input data. In the case of the mixed7 layer in Inception V3, the output shape is (None, 7, 7, 768). Here, None denotes the batch size, which can vary dynamically, while 7×7 specifies the spatial dimensions (height and width) of the feature map. The value 768 corresponds to the number of channels, representing the depth of the feature representation. This output encapsulates the learned features up to the mixed7 layer, making it suitable for downstream tasks such as feature extraction for other applications.

Comparing this model architecture between Inception v3's mixed7 layer and CNN developed from scratch, the main differences lie in the network structure, the complexity of the layers, and the output shape.

The model type developed from Scratch is a simpler convolutional neural network (CNN) with basic Conv2D layers and pooling operations, while Inception v3 is a much more complex network with multiple branches and operations within each layer.

The Layer Output Shape from the simple convolutional and pooling layers decreases progressively as the network extracts higher-level features, ultimately flattening to a large vector (None, 18496) after the Flatten layer. This is then passed through dense layers.

In Inception v3 the output of mixed7 has a shape of (None, 7, 7, 768). This means the feature map is still spatial (7x7), and the number of channels is 768. The Flatten operation would eventually occur after this point in Inception v3, but you're extracting features from the convolutional layers rather than passing through fully connected layers.

In total, the simple CNN model has roughly 9,494,561 parameters, with the majority in the dense layers (9,470,464 parameters), which indicates that it is a relatively simpler model with fewer feature extraction layers compared to Inception v3, which has millions of parameters spread across its various layers.

In essence, Inception v3's mixed7 layer provides more generalizable features that can be useful for a wide variety of tasks, thanks to its complex architecture and pre-training on a large dataset like ImageNet.

```
# Add classification layers
last_layer = pre_trained_model.get_layer('mixed7')
last_output = last_layer.output

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu', kernel_regularizer=l1_l2(l1=0.001, l2=0.01))(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.7)(x)
x = layers.Dense(37, activation='softmax')(x)


model = Model(pre_trained_model.input, x)

# Compile the model
model.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(learning_rate=1e-4),
    metrics=['categorical_accuracy']
)
```

Figure 11 - Model configuration following a fully connected classifier on top of the last output

The next step, as shown above, is to adapt the features extracted from the pretrained InceptionV3 model to our specific classification task. Since the original model was trained on a different dataset, we add custom fully connected layers to learn new representations suitable for our 37-class classification problem. The Flatten layer converts the extracted feature maps into a one-dimensional vector, which is then passed through a dense layer with 1,024 neurons and ReLU activation to capture complex patterns. A Dropout layer is applied to reduce overfitting, as well as regularization L1 and L2.

For multiclass classification, the output layer must have one neuron per class (37 neurons in this case) and use the softmax activation function. This ensures that the model produces a probability distribution over all possible classes, where the sum of probabilities equals 1. One-hot encoding is used for the labels, where each class is represented as a vector with a 1 at the index of the class and 0s elsewhere. This is essential when using categorical_crossentropy as the loss function.

With regards to the optimizer, Adam is often preferred due to its combination of momentum and adaptive learning rates, which helps it converge faster and more effectively. It adjusts the learning rate for each parameter individually, making it more robust and less sensitive to hyperparameter tuning.

Leveraging Pretrained Models on the Dataset in analysis

In this case, preparing the dataset for analysis involves several important steps to ensure it aligns with the requirements of the pre-trained model. Once the dataset is downloaded, since it contains mixed images, it's necessary to create the correct directories for training and validation for each breed type, which in total amounts to 37 breeds. The dataset will be split into 70% for training and 30% for validation, ensuring the model has sufficient data to train and evaluate performance effectively.

Since the sample size is not large, data augmentation will be applied to the training set. Data augmentation involves applying random transformations to the images, such as rotations, flips, zooms, or shifts, to artificially increase the size and variety of the training data. This technique helps prevent overfitting by introducing more variability into the training data, allowing the model to generalize better on unseen data. By generating new variations of the images, data augmentation improves the robustness and performance of the model, especially when working with smaller datasets.

Improving Accuracy with Fine-Tuning

Fine-tuning the last layers of the pretrained model to adapt it for extracting task-specific features is an effective approach to improving accuracy. This involves freezing the initial layers, which capture general features, and training only the final layers to tailor them to the new dataset.

In the feature-extraction, we added two classification layers on top of an Inception V3 layer. The weights of the pretrained network were not updated during training. One way to increase performance even further is to "fine-tune" the weights of the top layers of the pretrained model alongside the training of the top-level classifier.

Initially, we freeze the layers of the pre-trained model and train only the newly added classifier on top. This prevents the gradients from overwriting the pre-trained weights, as the new classifier has random weights.

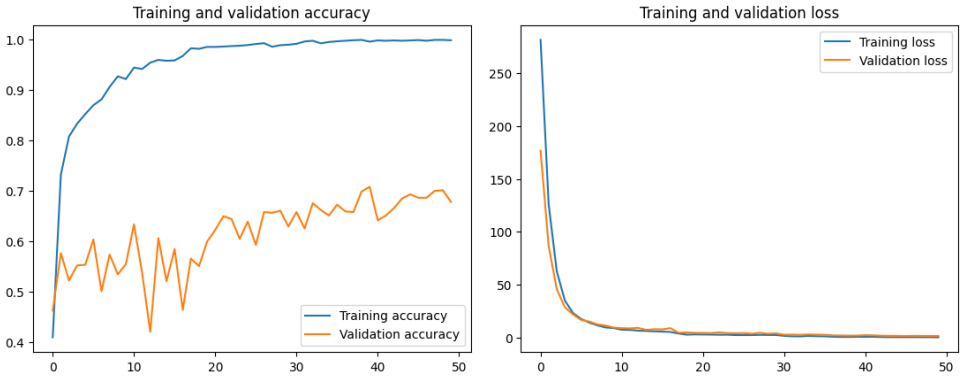
Once the top classifier is trained and performing well, we begin fine-tuning the higher layers of the pre-trained model. Fine-tuning only the top layers is recommended because they contain features more specific to the dataset the model was originally trained on. The lower layers learn general features that can be transferred across datasets, while the upper layers are more specialized to the original dataset.

We unfreeze all layers from the mixed7 module onward. After unfreezing, we will recompile the model and continue training to adapt the top layers to the new dataset while preserving the general knowledge of the lower layers.

Conclusion and Evaluating Results

In this model, after several attempts, some key techniques were employed to enhance performance and mitigate overfitting, leading to improved convergence and higher accuracy.

1. **Data Augmentation**: To increase the diversity of the training data and reduce overfitting, data augmentation was applied. This technique involved generating additional variations of the training images by applying transformations such as rotation, flipping, scaling, and shifting. By artificially expanding the training set, the model was exposed to a broader range of input variations, allowing it to generalize better to unseen data.
2. **Regularization Techniques**: Regularization methods were utilized to prevent overfitting and improve model generalization:
    - **Dropout**: A dropout rate of 70% was applied to the fully connected layer, randomly disabling a fraction of the neurons during training. This helps prevent the model from becoming too reliant on specific features, encouraging it to learn more robust and generalized patterns.
    - **Regularization**: L1 and L2 regularization (weight decay) was added to the dense layers, penalizing large weights and discouraging the model from overfitting to noise in the data.
    - **Batch Normalization**: This technique was employed after dense layers to normalize the activations, speeding up training and improving stability. It also helps in reducing internal covariate shifts, ensuring the model's training dynamics remain steady.
3. **Pre-trained Model**: By leveraging the power of a pre-trained InceptionV3 model, we benefited from transfer learning. The pre-trained model provided a solid feature extraction backbone, which was fine-tuned for the multiclass classification task. This allowed the model to learn relevant features from the data efficiently, accelerating convergence and improving the final performance.
4. **Learning Rate Adjustment**: The implementation of the **ReduceLROnPlateau** callback helped optimize the learning process. By reducing the learning rate when the validation loss plateaued, the model was able to refine its parameters more effectively in later epochs, preventing overshooting and ensuring smoother convergence.



With these improvements, the model demonstrated progress in both training and validation accuracy. While the current test accuracy exceeds **80%**, running the machine for 60 minutes with 50 epochs. Nevertheless, the training accuracy exhibits a rapid increase in the initial epochs. This indicates that the model learns the training data effectively, achieving high accuracy. Conversely, the validation accuracy fluctuates significantly and remains considerably lower, around 0.7. This discrepancy between training and validation accuracy suggests that the model is likely overfitting.