Chosen models

* **Model from scratch**

Building a model from scratch 4, 50 epochs

* # defining the data augmentation stage
* # loading libraries
* from tensorflow import keras
* from tensorflow.keras import layers
* import tensorflow as tf
* # Defining the data augmentation stage
* data\_augmentation = keras.Sequential(
* [
* layers.RandomFlip("horizontal"),
* layers.RandomRotation(0.1),
* layers.RandomZoom(0.2),
* layers.RandomContrast(0.1),
* layers.RandomBrightness(0.1)
* ]
* )
* # instantiating the convnet with data augmentation
* # loading libraries
* from tensorflow import keras
* from tensorflow.keras import layers
* # defining the model with data augmentation
* inputs = keras.Input(shape=(524, 524, 3)) # defining defining the input shape based on the image dimensions and the number of channels (3 for RGB)
* x = data\_augmentation(inputs) # augmenting the input
* x = layers.Rescaling(1./255)(x) # rescaling the input values to be between 0 and 1 since the pixel values are between 0 and 255
* x = layers.Conv2D(filters=32, kernel\_size=3, activation="relu")(x) # defining the convolutional layer
* x = layers.MaxPooling2D(pool\_size=2)(x) # defining the max pooling layer
* x = layers.Conv2D(filters=64, kernel\_size=3, activation="relu")(x) # defining the convolutional layer based on the previous layer
* x = layers.MaxPooling2D(pool\_size=2)(x) # defining the max pooling layer based on the previous layer
* x = layers.Conv2D(filters=128, kernel\_size=3, activation="relu")(x) # defining the convolutional layer based on the previous layer
* x = layers.MaxPooling2D(pool\_size=2)(x) # defining the max pooling layer based on the previous layer
* x = layers.Conv2D(filters=256, kernel\_size=3, activation="relu")(x) # defining the convolutional layer based on the previous layer
* x = layers.MaxPooling2D(pool\_size=2)(x) # defining the max pooling layer based on the previous layer
* x = layers.Conv2D(filters=256, kernel\_size=3, activation="relu")(x) # defining the convolutional layer based on the previous layer
* x = layers.Flatten()(x) # flattening the output
* x = layers.Dropout(0.5)(x) # adding dropout
* outputs = layers.Dense(9, activation="softmax")(x) # defining the output layer, specifying the number of classes to be 9 (the number of classes in the dataset), and using softmax since it is a multi-class, single-label classification problem
* model = keras.Model(inputs=inputs, outputs=outputs) # creating the model

I choose to run the different “from scratch” models presented in the book to gain an overview of the effect they each might have on the performance. This showed me, that when adding augmentation to the simple convnet model, I could gain a much higher validation accuracy at 0.6677 and much lower validation loss at 2.9550 compared to the other models. I was therefore certain to add data augmentation to the build from scratch model.

Furthermore, the simple convnet model performed well when adding batch normalization, with a validation accuracy at 0.5901 and validation loss at 4.9935. I therefore decided to add this to my build from scratch model as well.

Before making use of these layers, the model takes the input shape 524, 524, 3, which corresponds to the image size and number of colour channels in the images. It then takes the augmentation and rescaling layer, before moving on to the five stages of Conv2D, BatchNormalization, Activation, and MaxPooling2D layers, which will help with the large image size. Then, the model flattens the 3D outputs to 1D, before adding dropout, which should help in reducing the chance of overfitting. Lastly, it defines the output layer, where the number of units set to 9 represents the number of classes in the data, while the activation function is set to “softmax”, representing that this is a multiclass, single label problem.

When running the newly build model with these added layers, the performance was disappointing when comparing to when the model with these layers separately were run. Since my model with data augmentation performed the best, I decided to remove the batch normalization layer, and instead add more layers to the augmentation, in order to help the model generalize even better while avoiding overfitting. This presented a trained model with a validation accuracy at 0.7254 and validation loss at 1.1611, the best one so far. I decided not to add any other layers, since I did test the mini Xception like model presented in the book, that includes many more layers, which performed worse than my model build from scratch including fewer layers. The reasons for why a more complicated model doesn’t perform better than the more simple model could be due to the nature of the data, but it can’t be said for certain.

A table of numbers with black text

Description automatically generated with medium confidence

A graph of a training and validation accuracy

Description automatically generated A graph of a training and validation loss

Description automatically generated

Training and Validation Accuracy Plot

* Plot Description: This plot shows the accuracy of the from scratch model on both the training and validation datasets over 50 epochs.
* Observations:
  + The training accuracy increases steadily and reaches a high value, indicating that the model is learning and fitting the training data well.
  + The validation accuracy also increases initially but plateaus around epoch 40, indicating that the model’s performance on unseen data is not improving much after this point.
* Comments:
  + Overfitting: The divergence between training and validation accuracy in the latter epochs suggests overfitting. The model is becoming overly specialized to the training data and is not generalizing well to new data.
  + Data Augmentation: As mentioned in the document, adding data augmentation improved validation accuracy significantly. This implies that augmentation helped in reducing overfitting by making the model more robust to variations in the data.

Training and Validation Loss Plot

* Plot Description: This plot shows the loss values for both the training and validation datasets over 50 epochs.
* Observations:
  + The training loss decreases consistently, showing that the model is minimizing the error on the training set.
  + The validation loss decreases initially but starts to increase slightly after epoch 40, indicating overfitting.
* Comments:
  + Overfitting: The increase in validation loss while training loss continues to decrease is a classic sign of overfitting. The model is memorizing the training data rather than learning general features.
  + Dropout: The inclusion of dropout layers helped mitigate overfitting by preventing the model from becoming too reliant on any single feature during training.

A close-up of a list

Description automatically generated

* **Pretrained model**
  + Inception V3 with 524 size images running for 50 epochs

When choosing which pretrained model to run, it made sense to take inspiration from the article and their results. Since all of those pretrained models originally were trained on the same ImageNet data set, using any of them for waste classification would at first seem equally appropriate. The article presented the Inception V3 model as the one with the best performance amongst five different pretrained models, and I therefore chose this for the task.

A few notes for the final model:

* I used the Inception V3 pretrained model, which originally was trained on images with size 299x299. I chose to go with using the original image sizes of 524x524, since this provided the best performance.
* The article used batch size 32 for this model, and since this is a common size, and not too large or too small, I went with this as well.
* Unfreezing layers

1. Unfreezing Layers after a Specific Point (mixed9):
   * Leveraging Pretrained Knowledge: Inception V3, pre-trained on a large dataset like ImageNet, has learned rich feature representations in its lower and middle layers. These features are generally useful for a wide variety of image recognition tasks.
   * Adaptation to New Task: By unfreezing the layers from the "mixed9" block onwards, the model can adjust higher-level features more specific to the new dataset. The "mixed9" block is relatively deep within the network, where more complex and task-specific features are represented.
   * Efficient Training: Training the entire model from scratch can be computationally expensive and may require a large dataset. By freezing the earlier layers, you can reduce the computational load and the risk of overfitting, especially if the new dataset is smaller.
2. Keeping Batch Normalization Layers Non-Trainable:
   * Stability During Fine-Tuning: Batch normalization layers maintain internal statistics (mean and variance) that are crucial for stable training. When fine-tuning, changing these statistics dynamically (if these layers are trainable) can destabilize the training process, especially if the new dataset is smaller or significantly different in distribution.
   * Preserving Pretrained Performance: The statistics captured during pre-training are generally robust and help in maintaining the generalization capability of the model. By keeping batch normalization layers non-trainable, these beneficial properties are preserved, contributing to better performance and faster convergence.
3. Ensuring Adaptation with Stability:
   * Combination of Frozen and Trainable Layers: Freezing layers before "mixed9" ensures that the fundamental, general features learned from the original dataset remain intact. Unfreezing layers from "mixed9" onward allows the model to adapt the high-level, task-specific features, thus providing a balance between retaining valuable pretrained knowledge and learning new, task-specific patterns.
   * Selective Trainability: Making only specific layers trainable helps in fine-tuning the model more effectively without the risk of overfitting to the new dataset, which can happen if all layers are made trainable simultaneously.

* I set the learning rate low, both because it is what they do in the article, but also because…

1. Pre-Trained Weights Stability:
   * Fine-Tuning: When fine-tuning a model, the weights in the pre-trained layers already contain valuable information learned from a large dataset (like ImageNet). Using a smaller learning rate ensures that these weights are adjusted gently. This prevents significant alterations that could disrupt the learned features, leading to potential loss of the valuable knowledge encoded in these weights.
2. Avoiding Overfitting:
   * Gradual Adjustment: A smaller learning rate ensures that the model’s parameters are updated gradually. This helps in preventing overfitting to the new dataset, which might be smaller or less diverse than the dataset used for pre-training. It allows the model to adapt to new data without drastically overfitting to noise or small peculiarities in the new dataset.
3. Model Convergence:
   * Smooth Training Process: With a smaller learning rate, the training process becomes smoother and more stable. Sudden large updates (which can happen with a higher learning rate) might cause the model to diverge or oscillate around local minima. A smaller learning rate helps in achieving a stable and gradual convergence towards a new optimum that incorporates both the old and new knowledge effectively.
4. Risk of Catastrophic Forgetting:
   * Preservation of Learned Features: Using a large learning rate during fine-tuning can lead to catastrophic forgetting, where the model quickly loses the useful patterns learned from the initial dataset. A smaller learning rate mitigates this risk by making small, controlled updates to the weights.

* This model had validation accuracy at 0.9319 and validation loss at 0.3863. This is better than the model build from scratch, but I would definitely need to be aware of how both are performing with classification tasks, as this pretrained model might not have the quickest response time compared to the model build from scratch, which is essential when under a time limit.
* A table with numbers and symbols

  Description automatically generated
* A graph of a graph

  Description automatically generated with medium confidenceA graph of training and validation loss

  Description automatically generated

Training and Validation Accuracy Plot

* Plot Description: This plot shows the accuracy of the pretrained model on both training and validation datasets over a number of epochs.
* Observations:
  + The training accuracy starts high and increases slightly, which is typical for pretrained models as they already have learned weights.
  + The validation accuracy is relatively stable and shows minor improvements over epochs, suggesting good generalization from the start.
* Comments:
  + Overfitting: There is less evidence of overfitting compared to the from scratch model. The pretrained model shows a more stable validation accuracy, indicating that it generalizes well to unseen data.
  + Fine-Tuning: The stable performance suggests that fine-tuning was done appropriately. The pretrained model benefits from already learned features, and fine-tuning specific layers helped improve performance without significant overfitting.

Training and Validation Loss Plot

* Plot Description: This plot shows the loss values for both training and validation datasets over epochs.
* Observations:
  + The training loss starts low and decreases slightly, consistent with pretrained models that have already minimized loss on large datasets.
  + The validation loss remains relatively low and stable, indicating good generalization.
* Comments:
  + Overfitting: The pretrained model exhibits minimal overfitting as evidenced by the stable validation loss. This suggests that the pretrained features are effective and that the model is not overfitting to the training data.
  + Generalization: The low and stable validation loss reflects the pretrained model’s ability to generalize well to new data, which is a significant advantage of using pretrained models over from scratch models.

A screenshot of a computer

Description automatically generated

**Overall Comparison**

* From Scratch Model: This model showed initial overfitting issues, which were mitigated by using data augmentation and dropout. The final performance, while improved, was still subject to overfitting to some extent.
* Pretrained Model: This model demonstrated better initial performance with minimal overfitting. Fine-tuning allowed for maintaining good generalization, highlighting the efficiency of leveraging pretrained weights and features.