**Deep Reinforcement Learning(CS6482)**

**Prof. J. J. Collins**

Dept. of Computer Science & Information Systems

**Assignment-1**

**Implement a Convolutional Neural Network (CNN) using YOLO V1 architecture**

Student name: Muhammad Khubaib Akram (23098929)

Student name: Adeeb Naiyer (23136316)

Table of Content

1. Data Set

2. Pre-processing

3. The network structure and other hyperparameters

4. Loss function

5. The optimiser – (Adams, RMSprop)

6. Cross Fold Validation

7. Results – Mean average precision

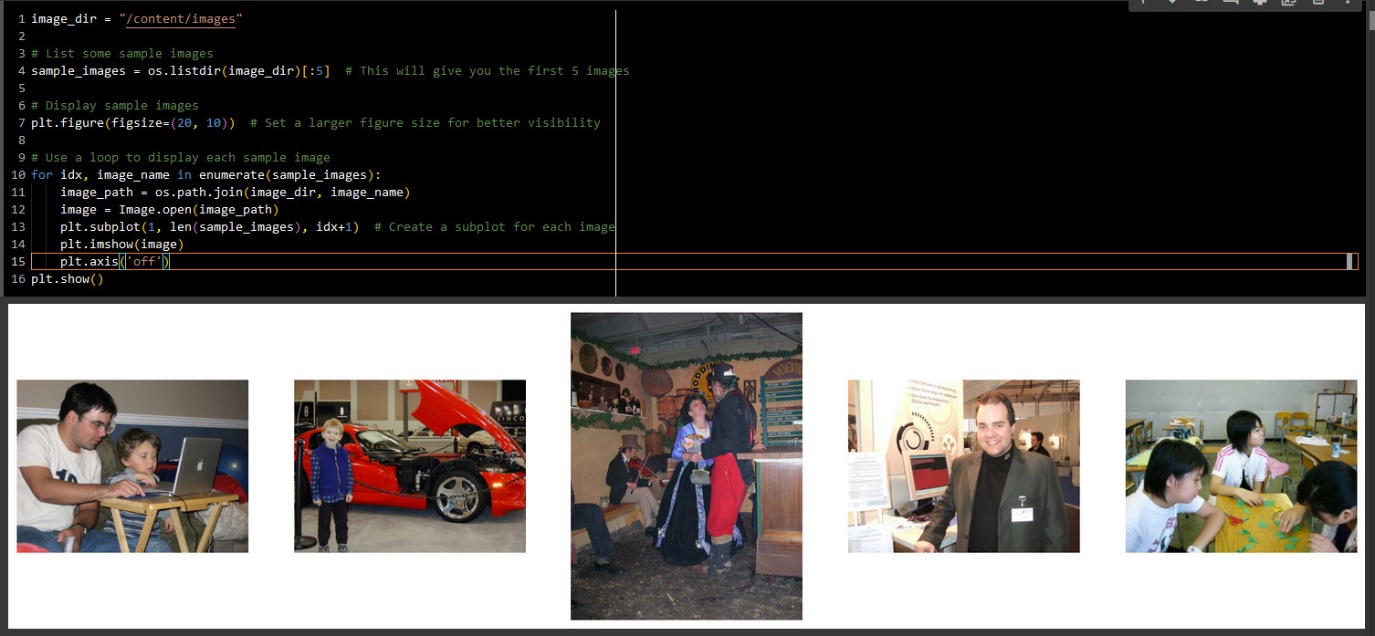
8. Evaluation of the results

9. Varying a parameters/hyperparameter(s) - overfitting and/or underfitting

10. References

1. **Data Set**

The PASCAL VOC (Visual Object Classes) dataset is a widely used benchmark in computer vision, specifically designed for object detection, segmentation, and classification tasks. It contains a diverse set of images across 20 object categories, including common objects such as cars, people, animals, and household items. Each image in the dataset is annotated with bounding boxes around objects and pixel-level segmentation masks where applicable, providing rich ground truth information for training and evaluating computer vision algorithms. The dataset has been instrumental in advancing research in object recognition and scene understanding, serving as the basis for numerous state-of-the-art models and techniques. Its widespread adoption and consistent updates over the years have established PASCAL VOC as a cornerstone dataset in the computer vision community, facilitating progress and innovation in various real-world applications, from autonomous driving to augmented reality.

****

**3. The network structure and other hyperparameters**

**3.1 Introduction to Convolutional Neural Networks (CNNs)**

Convolutional Neural Networks (CNNs) have emerged as a cornerstone in the field of computer vision, revolutionizing tasks such as image classification, object detection, and feature extraction. This report aims to elucidate the foundational concepts and significance of CNNs in the realm of visual recognition and understanding.

**3.2 Network architecture**

The YOLOv1 (You Only Look Once, version 1) architecture is an innovative approach for object detection that was introduced by Joseph Redmon et al. Unlike previous object detection systems which often use a two-step process (first generating potential bounding boxes, then classifying the contents of the boxes), YOLOv1 applies a single neural network to the full image, which directly predicts bounding boxes and class probabilities in one evaluation. Here is an in-depth explanation of its architecture:

Overall Architecture YOLOv1 uses a convolutional neural network (CNN) that divides the input image into an S x S grid. Each grid cell is responsible for predicting B bounding boxes and confidence scores for those boxes. Each bounding box consists of 5 predictions: x, y, w, h, and a confidence score. Alongside this, each grid cell also predicts C conditional class probabilities. Therefore, the network output is a tensor of shape (S, S, B\*5 + C).

**Network Design:** The architecture uses a sequence of layers consisting of convolutional layers followed by pooling layers, interspersed with occasional fully connected layers towards the end. Here's a closer look:

**Convolutional Layers:** These layers, typically arranged in increasing order of depth (number of filters), extract a hierarchy of features from the input image. They use various kernel sizes (like 7x7, 3x3, or 1x1) to capture different aspects of the input data. The 1x1 convolutional layers, in particular, help in dimensionality reduction and feature recombination.

**Pooling Layers:** Max pooling layers follow some of the convolutional layers to reduce spatial dimensions, thus gradually decreasing the height and width of the feature maps while retaining important information.

**Batch Normalization:** Though not originally included in the first YOLO paper, batch normalization can be applied after convolutional layers and before activation functions to help in stabilizing training and improving convergence.

Leaky ReLU Activation: Instead of using the traditional ReLU activation function, YOLOv1 applies the Leaky ReLU, which allows for a small gradient when the unit is not active

**A diagram of a dog and a bicycle

Description automatically generatedIdea of Yolo:**

[Picture reference](https://pjreddie.com/media/files/papers/yolo.pdf)

In YOLOv1, the input image is divided into a grid with dimensions S×S, where S is typically set to 7. If an object's center falls within a particular grid cell, that cell is tasked with detecting the object. Each grid cell predicts B bounding boxes, usually set to 2, along with confidence scores for these boxes. These confidence scores indicate the model's certainty about the presence of an object within the box (referred to as P(Objects)). Each bounding box is characterized by 5 predictions: x, y, w, h, and confidence. Here, (x, y) denote the coordinates of the box's center relative to the grid cell boundaries, while w and h represent the width and height of the box relative to the entire image. The confidence prediction reflects the Intersection Over Union (IOU) between the predicted box and any actual object box. Additionally, each grid cell predicts conditional probabilities for different object classes (denoted as P(Classi|Object)), with the total number of classes typically set to 20.

A diagram of a graph

Description automatically generatedBelow illustrate the output of the network:

[Picture reference](mailto:Refhttps://medium.com/@daksha.uchagaonkar/yolo-v1-working-explained-ec20750be682)

**The output size becomes: 7×7×(2×5+20)=1470**

A screenshot of a computer code

Description automatically generated**3.3 Architecture explanation:**

Each tuple or string in the list corresponds to a layer or a block of layers in the neural network, and their sequence represents the order in which they are applied in the model. Here's what each element means:

A tuple (kernel\_size, filters, stride, padding) represents a convolutional layer with the following parameters:

**Kernel\_size:** The size of the window to take a dot product with, usually a square (so a kernel size of 7 means a 7x7 window).

**Filters:** The number of filters (or kernels) to use, which is also the number of output channels this layer will produce.

**Stride:** The number of pixels to move the filter across the image; a stride of 2 means the filter jumps 2 pixels at a time as it slides across.

**Padding:** The number of pixels added to the outside of the input image to allow the kernel to slide across edge pixels.

A string "M" denotes a max-pooling layer, which typically halves the width and height of the input it processes. The common configuration for max-pooling in many CNNs is a 2x2 window with stride 2.

A list [(kernel\_size, filters, stride, padding), ... , n] represents a series of layers repeated n times. This is a convenient way to specify that a certain pattern of layers should be stacked together multiple times.

Non-Maximum Suppression (NMS) is a technique used in object detection to select the best bounding box when several overlapping bounding boxes are detected for the same object. This process helps reduce the number of redundant bounding boxes, keeping only the one with the highest confidence score while ensuring that different objects are not suppressed.

**3.4 Breaking down the architecture\_config:**

(7, 64, 2, 3): A convolutional layer with 64 filters of size 7x7, stride 2, and padding 3. "M": A max-pooling layer.

(3, 192, 1, 1): A convolutional layer with 192 filters of size 3x3, stride 1, and padding 1. "M": Another max-pooling layer.

(1, 128, 1, 0): A convolutional layer with 128 filters of size 1x1 (often used for dimension reduction), stride 1, and no padding.

(3, 256, 1, 1) followed by (1, 256, 1, 0) and (3, 512, 1, 1): A series of convolutional layers increasing in filter count. "M": Another max-pooling layer.

[(1, 256, 1, 0), (3, 512, 1, 1), 4]: A pattern of a 1x1 convolutional layer followed by a 3x3 convolutional layer, repeated 4 times.

(1, 512, 1, 0) and (3, 1024, 1, 1): Additional convolutional layers increasing the depth further. "M": Another max-pooling layer.

[(1, 512, 1, 0), (3, 1024, 1, 1), 2]: Similar to the previous pattern but repeated 2 times.

Finally, four 3x3 convolutional layers with a stride of 1 and padding of 1, all with 1024 filters. This series of layers presumably forms the backbone of the network, providing a rich feature representation of the input.

**3.5 Overview of CNN Layers and Blocks**

CNNs are characterized by a hierarchy of interconnected layers, each serving a specific function in feature extraction and representation:

**Conv2d**: These layers employ filters to extract features from input images, with the parameter count determined by filter size, input channels, and output channels.

**BatchNorm2d**: Used for normalizing activations, aiding in training stability and acceleration.

**LeakyReLU**: An activation function introducing non-linearity, crucial for enabling complex feature representations.

**MaxPool2d**: Reduces spatial dimensions, aiding in computation efficiency and mitigating overfitting.

**CNNBlock**: Custom blocks encapsulating convolution, batch normalization, and activation operations, facilitating modular network design.

**Flatten**: Reshapes input into a single dimension, preparing it for fully connected layers.

**Linear**: Fully connected layers that reduce input features to output features, crucial for high-level feature extraction.

**Dropout**: Regularization technique to prevent overfitting by randomly dropping units during training.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated**3.6 Summary** -with batch normalisation for YOLO v1Top of Form

**A screenshot of a computer program

Description automatically generated**

This is a summary of the YOLOv1 model architecture:

Darknet Backbone:

Input Shape: (3, Height, Width)

Output Shape: (1024, Height/32, Width/32)

Layers:

Convolutional Block (7x7 Convolution)

Input: 3 channels

Output: 64 channels

Kernel Size: (7, 7)

Stride: (2, 2)

Padding: (3, 3)

Activation: LeakyReLU (with negative slope of 0.1)

Max Pooling (2x2)

Kernel Size: (2, 2)

Stride: (2, 2)

Convolutional Block (3x3 Convolution)

Input: 64 channels

Output: 192 channels

Kernel Size: (3, 3)

Stride: (1, 1)

Padding: (1, 1)

Activation: LeakyReLU (with negative slope of 0.1)

Max Pooling (2x2)

Kernel Size: (2, 2)

Stride: (2, 2)

... (Multiple CNNBlocks continue, each followed by Max Pooling)

Fully Connected Layers (FCs):

Input Shape: (1024 \* Height/32 \* Width/32)

Output Shape: (1470)

Layers:

Flatten

Reshapes the input tensor into a 1-dimensional tensor

Linear (Fully Connected)

Input: 1024 \* Height/32 \* Width/32 features

Output: 496 features

Activation: LeakyReLU (with negative slope of 0.1)

Dropout

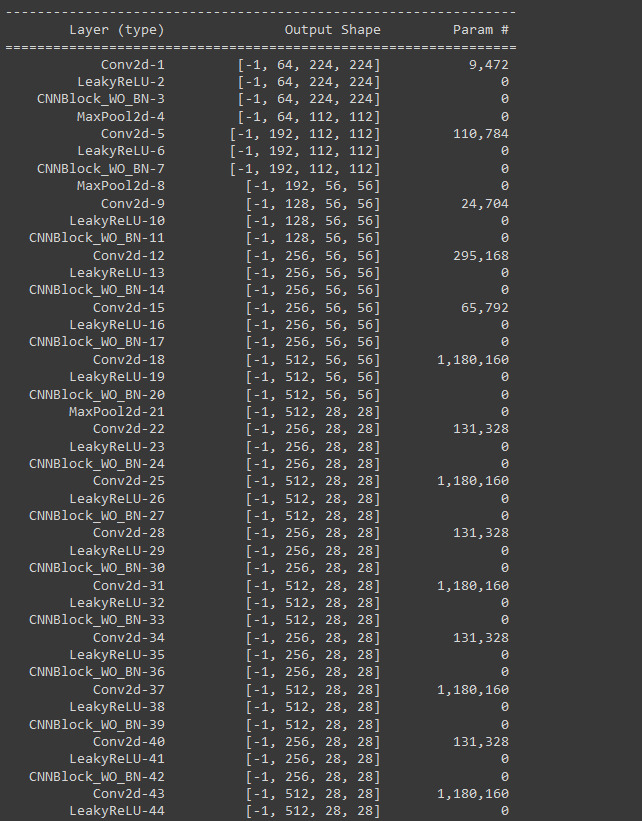
Dropout probability: 0.0 (no dropout applied)

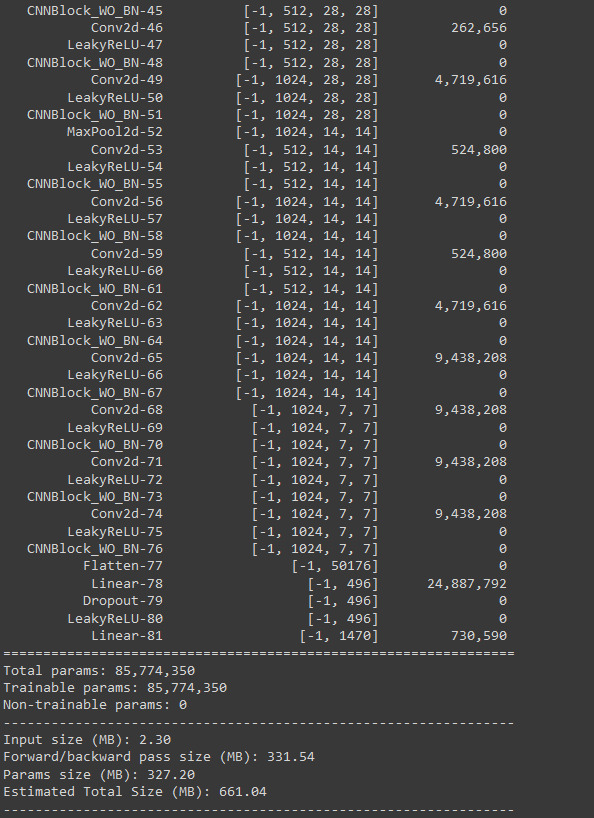
Linear (Fully Connected)

Input: 496 features

Output: 1470 features

This architecture represents the YOLOv1 model, consisting of a series of convolutional blocks (Darknet backbone) followed by fully connected layers. The model's output is a tensor with dimensions (Batch Size, 1470), which is later reshaped into predictions for bounding boxes and classes.

**Without batch normalisation for YOLO v1**

****

**Summary(without batch normalisation)**

The network consists of several convolutional layers followed by max-pooling layers and fully connected layers towards the end. Each convolutional layer uses a variant of convolution called Conv2d which is typical in CNN architectures for processing images.

Here are the details:

Convolutional Layers: The network includes multiple convolutional layers (Conv2d) with various kernel sizes, mostly 3x3 and 1x1, and different numbers of filters (from 64 up to 1024). The stride is typically set to 1, except in the first layer where it's 2, and padding is used to preserve spatial dimensions after convolution.

Activation Functions: After each convolutional operation, a LeakyReLU activation function is applied with a negative slope of 0.1. This is meant to introduce non-linearity into the model and allow it to learn more complex patterns.

Max Pooling Layers: Several max-pooling layers (MaxPool2d) with a kernel size of 2 and a stride of 2 are used to reduce the spatial dimensions of the feature maps and to provide an abstracted form of the features.

CNN Blocks without BN: Blocks named CNNBlock\_WO\_BN indicate that this particular version of the model does not include batch normalization, which is typically used to normalize the inputs of each layer to speed up training and improve convergence. Instead, it solely relies on the Conv2d and LeakyReLU layers for feature extraction.

Fully Connected Layers: At the end of the network, there are fully connected layers (Linear) that take the flattened output of the convolutional layers and transform them to the desired output size. The first fully connected layer reduces the features to 496 dimensions, and the second reduces further to 1470 dimensions, which is typical for the output of YOLOv1, where it predicts both class probabilities and bounding box coordinates.

Dropout: There is a dropout layer with a dropout rate of 0.6 before the last activation function. Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time.

This architecture is a simplified adaptation of the original YOLOv1, specifically omitting batch normalization. The absence of batch normalization means that this network may require more careful hyperparameter tuning, especially the learning rate, and could have slower convergence rates. It also may be more susceptible to issues such as vanishing or exploding gradients. Batch normalization often helps in stabilizing the training by normalizing the inputs to each layer, but here the model must rely solely on the LeakyReLU and weight initialization to maintain stable gradients.

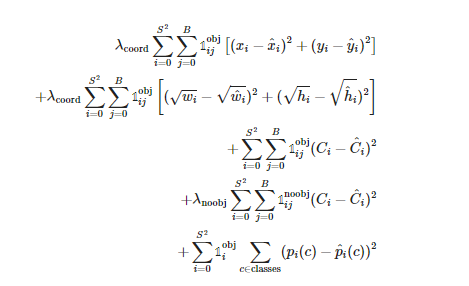
**4 Loss Function**

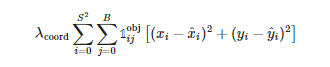
The loss function in YOLOv1 is like a guiding compass for the model to get better at spotting objects in pictures. It's made up of different parts, each focusing on a specific aspect of object detection.

First off, there's the "Localization Loss." This part helps the model learn how to accurately predict where objects are located in the image by comparing its guesses to the actual positions of the objects.

Then, there's the "Confidence Loss." This measures how confident the model is about its predictions. If it's sure about its guesses when it's right and unsure when it's wrong, that's a good thing. This helps the model learn to trust its predictions.

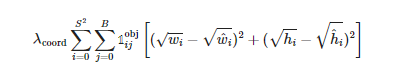
Lastly, there's the "Class Prediction Loss." This part deals with the model's ability to correctly identify what type of object it's looking at. It's like making sure the model knows the difference between a cat and a dog, for example.

All these parts work together to create the overall loss function. During training, the model tries to minimize this loss function, using techniques like adjusting weights or tweaking its predictions, to get better at spotting objects accurately in images. YOLO LOSS:

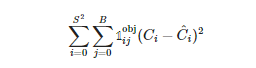
Box Coordinate Loss:

This term minimizes the squared error between the predicted (x\_i, y\_i) and the ground truth

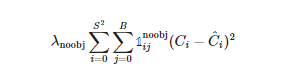
(x̂\_i, ŷ\_i) center coordinates of the bounding box for each cell i that contains an object. It is scaled by a factor λ\_coord to give it more weight relative to other components of the loss.

Box Dimension Loss:

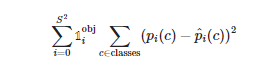
This term minimizes the squared error of the square root of the predicted bounding box width (w\_i) and height (h\_i) relative to the ground truth (ŵ\_i, ĥ\_i). This is done to reduce the impact of large boxes' deviations on the loss, and it is also scaled by λ\_coord.

Object Confidence Loss:

This loss penalizes the model for the difference between the predicted object confidence (C\_i) for each bounding box and the ground truth, which is 1 for boxes with objects. It is calculated only for the box with the highest IoU with the ground truth among the predictions for each cell.

**No Object Confidence Loss:**

This term minimizes the error between the predicted object confidence (C\_i) for each bounding box and the ground truth, which is 0 for boxes without objects. It is scaled by λ\_noobj because this prediction is typically less informative.

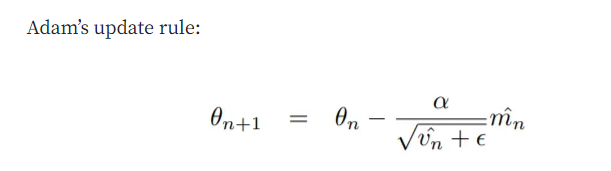
Class Prediction Loss:

This part of the loss function calculates the squared error of the predicted class probabilities (p\_i(c)) for each cell relative to the one-hot encoded ground truth class probabilities (p̂\_i(c)). This term only contributes to the loss for cells containing an object.

Overall, the YOLO loss function is a sum of these five terms, ensuring that the model is trained to accurately predict bounding box coordinates, dimensions, object confidence, and class probabilities.

**5. Optimiser**

**5.1 Adam Optimiser**

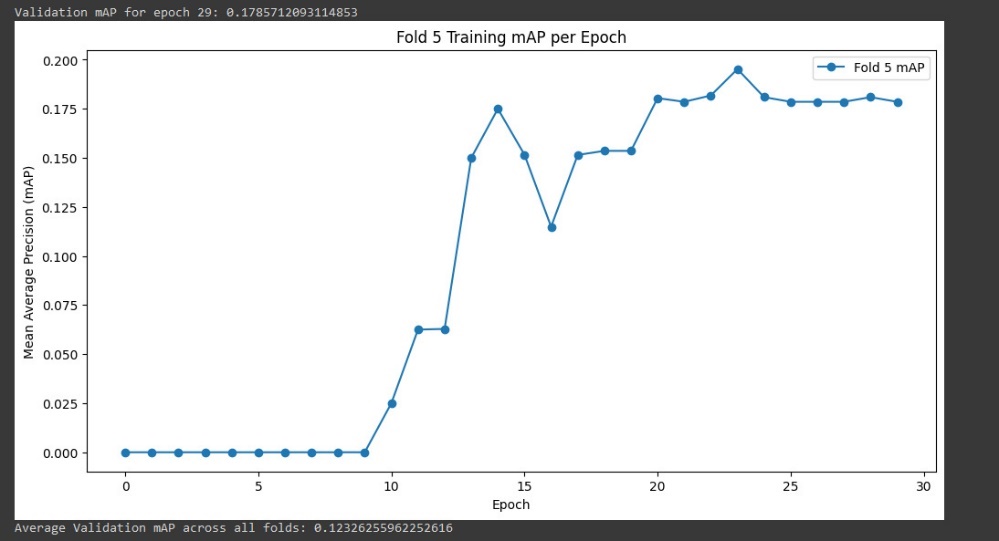
The Adam optimization algorithm is a technique utilized to improve the training efficiency and speed up the convergence of deep neural networks. Described in the paper "Adam: A Method for Stochastic Optimization," it extends the traditional gradient descent method by offering a dynamic adaptation of the learning rateIn standard gradient descent, the learning rate (α)

[Picture reference](https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be)

remains constant throughout training, requiring careful selection and manual tuning to find a suitable balance between convergence speed and stability. However, Adam addresses this challenge by adjusting the learning rate for each parameter (θ) based on its individual gradient history.

To explain how Adam functions, let's use an analogy of a father teaching his children, Chris and Sam, how to ride bikes. Chris pedals cautiously, while Sam is more daring and pedals faster. If the father were to push both bikes at the same speed, Chris might lag behind, and Sam could risk crashing.

Similarly, Adam continuously monitors the speed and acceleration of each parameter (similar to the children on bikes) and adapts its approach accordingly. If a parameter makes slow progress consistently (like Chris's bike), Adam provides a stronger push to speed up learning. Conversely, if a parameter progresses rapidly (like Sam's bike), Adam applies a gentle push to maintain stability.

By adjusting the learning rate for each parameter based on its gradient history, Adam ensures that the neural network learns efficiently as a whole, leading to smoother convergence and faster training. This adaptive strategy allows parameters to progress comfortably without compromising stability, much like Chris gaining speed comfortably while Sam avoids accidents, thereby optimizing the learning process.

The mAP values show an upward trend with less volatility compared to the RSEProp optimizer.

The mAP increases sharply up to around epoch 10, then continues to improve at a steadier rate.

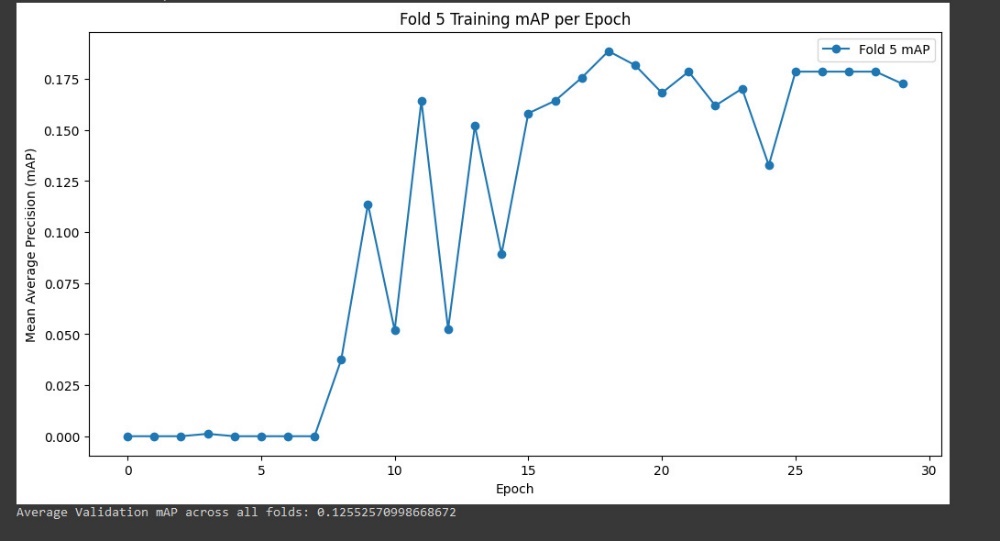
There's a significant dip around epoch 22, which could indicate a temporary learning setback, but the mAP recovers in subsequent epochs.

Average validation mAP across all folds is 0.1232, which is slightly lower than the RSEProp optimizer.

**5.2 RMSprop Optimiser**

A math equations on a white background

Description automatically generatedRMSprop is a method for optimizing neural networks created by Geoffrey Hinton, a pioneer in back-propagation. It addresses the challenge of vanishing or exploding gradients in complex functions like neural networks by employing a moving average of squared gradients to adjust the gradient's magnitude. This adjustment helps balance the step size, reducing it for large gradients to prevent explosion and increasing it for small gradients to prevent vanishing. Essentially, RMSprop adapts the learning rate during training, making it an adaptive technique rather than requiring a fixed learning rate as a hyperparameter.

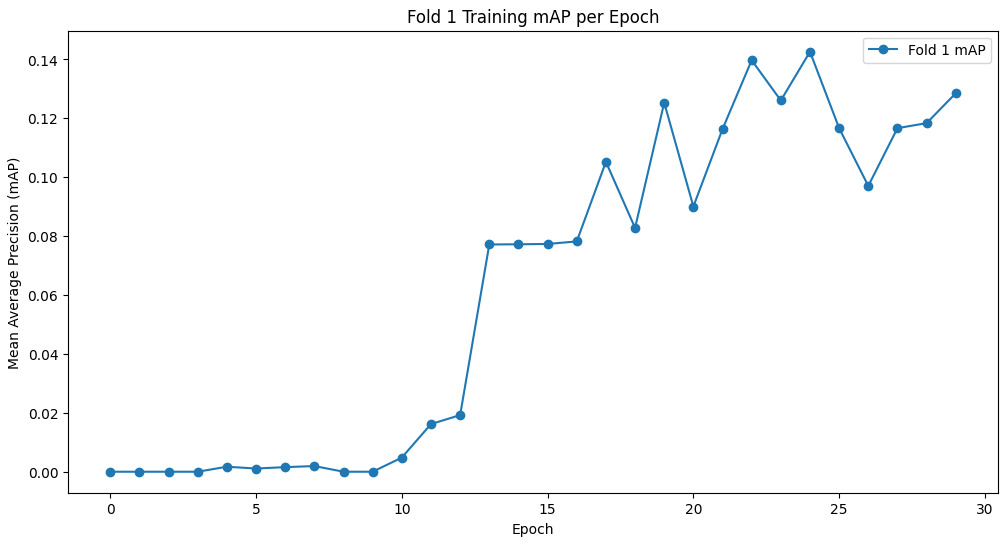
****[Picture reference](https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be)

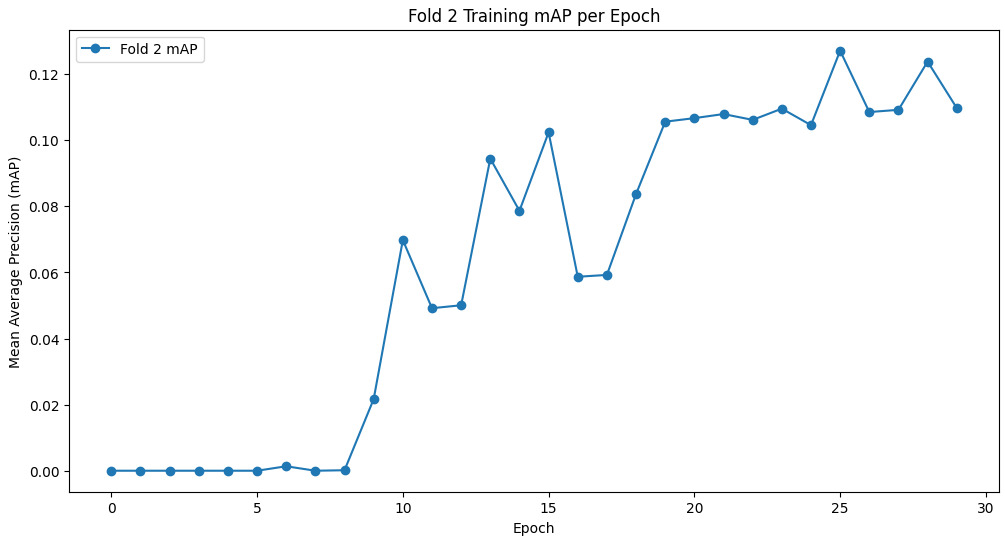
The mAP values are highly volatile, with significant peaks and troughs throughout the training process.

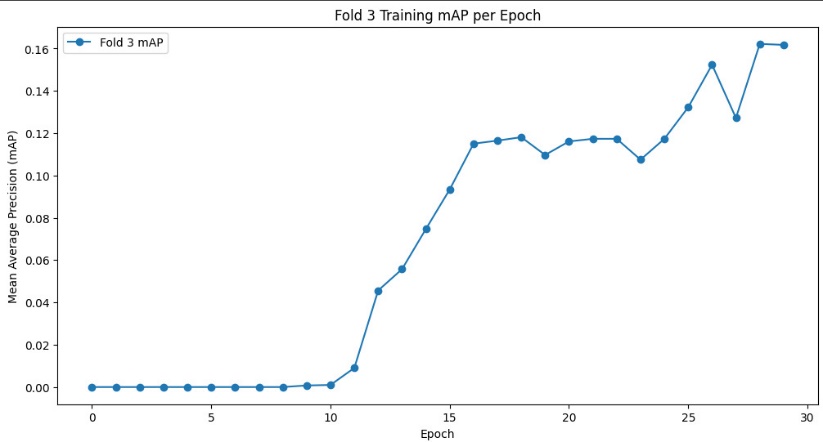
Despite the volatility, there's a general upward trend in mAP, suggesting some learning is taking place.

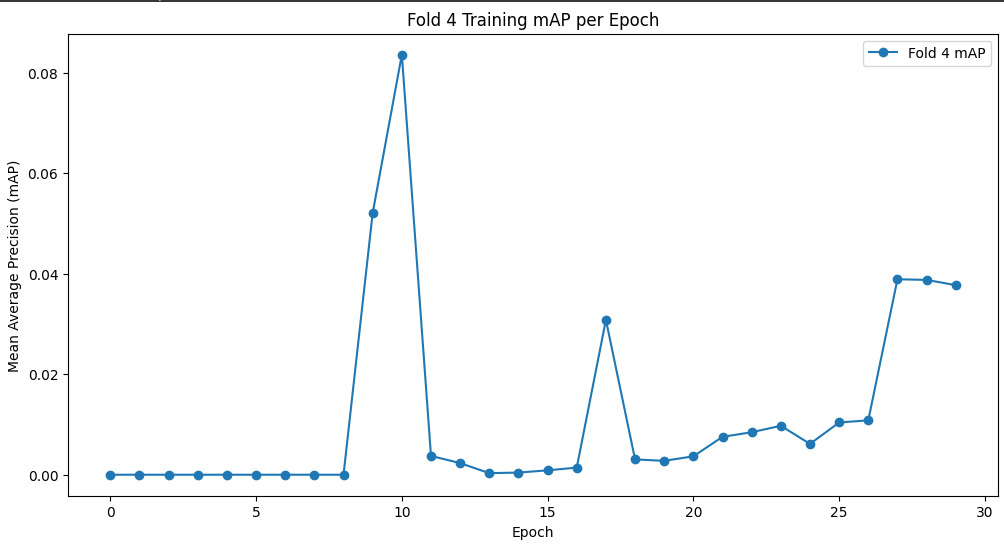
Average validation mAP across all folds is 0.1255, indicating the average performance across all the validation folds.

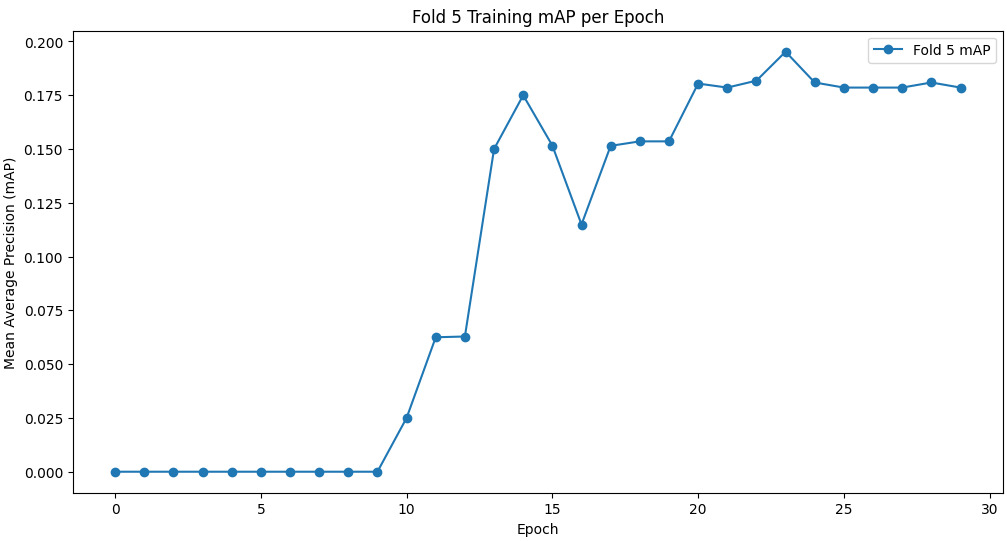
**6. K Fold Cross Validation**

K fold 1

K fold 2

K fold 3

K fold 4

K fold 5

Observations Across the Cross-validation Folds:

Fold 1 to Fold 5 mAP Graphs: The mAP across the five folds displays different learning patterns. Some folds show a steady increase in mAP over the epochs (Fold 3), while others have more variability and sudden changes in mAP (Fold 4).

Variability: The mAP in certain folds (particularly Fold 4) shows high variability, with significant peaks and drops. This could suggest that the data in these folds has features that the model finds challenging to learn consistently.

General Trend: Despite the variability, the general trend in most folds is an increase in mAP over time, indicating learning and improvement in the model’s ability to predict with precision.

Fold 5 as an Example: The mAP in Fold 5 shows a significant increase partway through training and then stabilizes, similar to the pattern seen in the non-cross-validation graph but with less pronounced growth.

**7. Results – accuracy and/or precision and/or recall**

Object detection models like YOLOv1 (You Only Look Once version 1), Mean Average Precision (mAP) is a commonly used metric to evaluate the model's accuracy. It combines two important concepts: precision and recall.

Precision measures the accuracy of the predictions made by the model, meaning out of all the detections the model made, how many were correct. Recall measures the model's ability to detect all possible instances in the data, meaning out of all the actual instances, how many the model detected.

Average Precision (AP) for a single class is calculated by plotting a Precision-Recall (PR) curve, which shows the trade-off between precision and recall for different thresholds. A threshold is the confidence level above which a prediction is considered positive. AP is the area under the PR curve.

In object detection, you often have multiple classes that you want to detect. To get a single metric that evaluates the overall performance across all these classes, you take the mean of the APs for each class, hence the term Mean Average Precision (mAP).

To summarize, in YOLOv1:

AP for a class = Area under the Precision-Recall curve for that class.

mAP = Mean of the APs across all classes.

This metric considers both the confidence of the detections and the correctness of the bounding boxes predicted by the model. A higher mAP value indicates a better performing model, both in terms of detecting objects accurately and in assigning the correct class to those detections.

**9. Impact of varying a parameters/hyperparameter(s)**

A graph of training and validation

Description automatically generated**Underfitting**

The graph illustrates the performance of a model over 10 training epochs, with the blue line representing the training loss and the orange line indicating the validation loss. While the training loss steadily decreases, suggesting the model is learning from the training data, the validation loss fluctuates without showing a clear downward trend. This inconsistency signals underfitting.

Underfitting happens when a model fails to grasp the underlying patterns in the data and instead memorizes specific examples from the training set. Consequently, the model performs well on the training data but struggles to generalize to new, unseen data.

**Regarding the factors contributing to underfitting:**

**Low Learning Rate (lr=1e-6):** A very low learning rate slows down training and might prevent the model from capturing the intricate patterns in the data. Updates to the model's weights during training may be too small to effectively capture the data's underlying relationships.

**Weight Decay (weight\_decay=0.1):** While weight decay helps prevent overfitting by penalizing large weight values, a high weight decay value (like 0.1) can also lead to underfitting. It pushes the model towards simpler solutions, potentially limiting its ability to learn complex features from the data.

**Limited Epochs (#epochs=10):** Although not directly linked to underfitting, restricting the number of training epochs to 10 may hinder the model's ability to fully explore the data and converge to an optimal solution. Especially with a low learning rate, more training epochs might be necessary for effective learning.

#underfitting

model = Yolov1\_WO\_BN(split\_size=7, num\_boxes=2, num\_classes=20).to(DEVICE)

optimizer = optim.Adam(model.parameters(), lr=1e-6, weight\_decay=0.1)

loss\_fn = YoloLoss().to(DEVICE)

train\_losses = []

valid\_losses = []

for epoch in range(10): # Fewer epochs to prevent the model from learning properly

train\_loss = train\_f(train\_loader, model, optimizer, loss\_fn)

valid\_loss = validate\_f(valid\_loader, model, loss\_fn)

train\_losses.append(train\_loss)

valid\_losses.append(valid\_loss)

print(f"Epoch {epoch+1}, Train Loss: {train\_loss}, Valid Loss: {valid\_loss}")

# Plotting training vs validation loss to demonstrate underfitting

plt.plot(train\_losses, label='Training Loss')

plt.plot(valid\_losses, label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Training vs Validation Loss Showing Underfitting')

plt.show()

**10. References**

<https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be>

<https://pjreddie.com/media/files/papers/yolo.pdf>