Using Object Segmentation to Analyze Nutrient Content of Meals

Project Abstract

This project explores machine learning techniques, using YOLOv8 instance segmentation model. The primary aim of this project is to provide users with an easy and convenient way to access detailed nutritional information about their meals without the need for manual input of the meal's contents. By simply snapping a photo of their meal, users can obtain valuable insights into the percentage of various macronutrients present on their plate, such as fiber, sugar, fat, protein, and carbohydrates.

Object segmentation is a technique used to detect the exact boundaries of an object, as opposed to traditional object detection, which only predicts a bounding box. The research investigates the complexity of the architecture used for instance segmentation, a network known as a convolutional neural network (CNN). In CNN, the process begins with pooling, which downsamples the image, followed by convolutional layers that extract features of the image using filters. These features are then upsampled to increase resolution. Fully connected layers analyze these features to look for high-level patterns and eventually output which class the output belongs in. The coordinates of the features that make up the object are kept track of, which allows for the segmentation mask, which is an outline of the object, to be generated. The parameters in the functions used to analyze features are optimized during training epochs (cycles). The model is trained on a large dataset of annotated and labeled images, where its parameters are changed slightly during each epoch.

Through rigorous experimentation and analysis, four distinct models were trained with varying numbers of epochs. The metrics of each model, segmentation loss, classification loss, focal loss, precision, and recall, were used to determine which of the 4 is the best-performing model. The 1st model was trained for 100 epochs, and the 2nd for 200 epochs. The third was trained for 100 epochs, stopped, and then trained for another 50 epochs (100+50 epochs). The fourth was trained for 100+100 epochs. Stopping halfway through training helps prevent overfitting. Results show that model 1 had an especially high classification loss, and that the model has low precision when analyzing partially hidden objects, indicating that this model is underfitting. Model 2 was overfit, given the large difference between training and validation losses, and the fact that validation losses were increasing instead of decreasing during the last 100 epochs. Models 3 and 4 were similar in performance, but model 3 had slightly lower classification loss and higher precision. So, model 3, trained for 100+50 epoch, was selected as the best-performing model.

The trained model is integrated with Edamam Nutrition Analysis API, which gives accurate and up-to-date nutritional information regarding various food items. The nutrient percentage in each food item, adjusted for its size relative to the total size of the meal, is summed to determine the overall percentage content for each nutrient in the meal. The final software created for this project can analyze an image of a meal, and then output the nutrient percentage content of each food item and the overall meal.

Not only does this project showcase the potential of machine learning in addressing nutritional awareness but also paves the way for future advancements in the field of convolutional neural networks.

Research

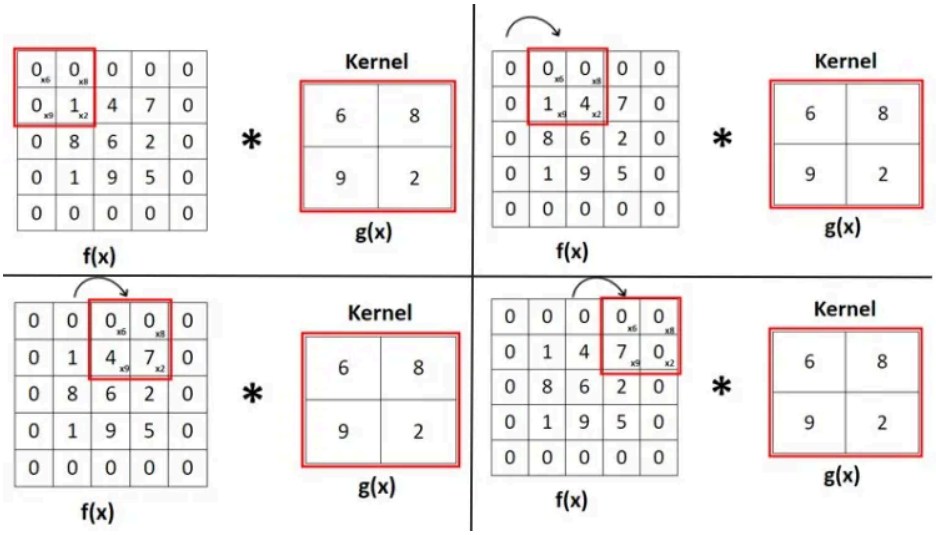
Summary

1. The convolutional layer extracts features like lines, textures, and color from the image
2. The fully connected layers analyze these features to find patterns and output the probability of each class being present in the image
3. The model is trained to optimize the parameters in the fully connected layers

Convolutional Layer

Feature Maps

- Feature maps: the result of applying a certain filter to the image, which usually highlights certain features of the image, such as curves/lines at different angles, textures, and colors. In the convolutional layer, many feature maps that highlight varying characteristics of the image are created, which are then fed into the fully connected layer for further processing.
- Filtering in feature maps: work by sliding a kernel across the image, and applying changes to every pixel in the image. The kernel in yolov8 is a 3x3, 5x5, or 7x7 matrix with weights (values in the matrix), with each index covering one pixel. The dot product is taken between the kernel and the pixels it covers. The result is used as the value for the center pixel that the kernel covers. The other pixels covered by the kernel stay the same. This process is repeated as the kernel slides across the image

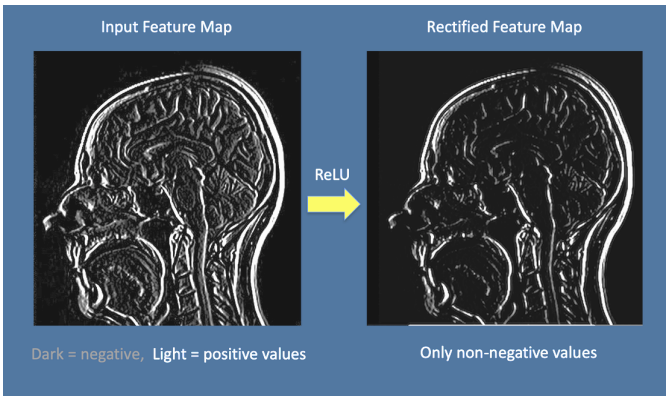


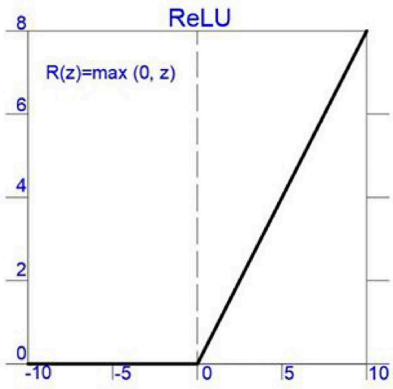
- Stride: the amount of pixels by which the kernel moves each time it shifts. A larger stride will increase computational efficiency/faster processing. A smaller stride can capture more details.
- Padding: due to the nature in which the kernel moves across the image, the perimeter of the original image will be cut down in the resulting image. A 10x10 image, processed by a 3x3 kernel, will result in an 8x8 image. This results in a loss of information. The padding adds extra pixels on the perimeter to prevent a reduction in dimensions. Replicate padding copies the edge pixels of the original image for the new pixels (inaccurate information on edges). Zero padding sets new pixels as 0 (lower computational efficiency)
- A few hundred feature maps are generated in the convolutional layer.

- All feature maps are flattened before being inputted into the fully connected layer. Flattening means reducing the spacial dimension of the image, from a 3D (height, width, color) to a 1D vector containing all these values.

Activation Functions

- Activation functions are applied to feature maps to highlight important characteristics and introduce non-linearity. Yolov8 uses the ReLU (rectified linear unit) activation function, which sets all negative inputs into the function to 0. The kernel's weights are trained to return all unimportant/background pixels as negative numbers, and those pixels will be set to black so only the important features are shown. This function is applied to feature maps to make them more effective, as important features will be more defined. The ReLU function is applied to the value of every pixel in the feature map.
- ReLU function: f(x) = max(0,x)





Upsampling and Pooling

Pooling

- Downsampling layers uses pooling methods to reduce the spatial dimensions of the image without removing any important features. Pooling is applied to images before creating feature maps. Pooling has many benefits. Reducing the image dimensions will increase computational efficiency, and reduce the noise in the image. It helps with extracting the higher level features, as most irrelevant small details will be "blurred"/made unnoticeable by the pooling functions, while more prominent features won't be. These features include edges, texture, and color)
- Max pooling: a 2x2 kernel slides across the image, takes the max value of the 4 pixels it covers and uses it as the value of the corresponding pixel in the resulting image
- Average pooling: same process as max pooling, but the 4 pixels are averaged instead of taking the largest value. Both pooling methods are effective

Upsampling

- After downsampling to generate feature maps, the maps are then upsampled to increase resolution. High resolution is needed in instance segmentation, as edges are more easily delineated (detected) when each region in the object is represented by more pixels
- Nearest-neighbor interpolation: adds new pixels to the image by duplicating the existing ones

- Bilinear interpolation: adds new pixels, where the value of the new pixels is calculated by averaging the value of the nearest pixels. Results in smoother transitions between color sections, it won't appear blocky and "pixelated" like the nearest-neighbor method.

Fully Connected Layers

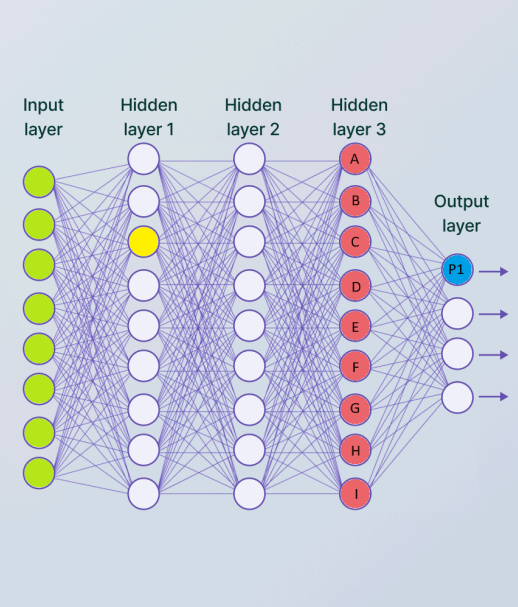
Input Layer

Feature maps are fed into the fully connected layers through the input layer.

- The fully connected layers have many layers of neurons. The input layer, multiple hidden layers, and an output layer. The input layer takes in feature map vectors as input, the hidden layers process the input to find patterns (high-level features), and the output layer returns the probability for each class (a class is a category in which the detection belongs, ie whether the object detected is an apple, an orange, or a banana; the fruits are examples of classes).
- The flattened vectors that represent feature maps are fed into the fully connected layer through the input layer. Each neuron in the input layer contains one pixel's value from one of many feature maps.
- A neuron is a computational unit that applies a function to inputs and returns an output.

Hidden Layers

The hidden layers analyze low-level features to recognize high-level features



Each neuron is connected to every neuron from the previous layer. Being connected means the first neuron passes its output to the second one. Each input layer neuron (green) represents one pixel's value in one feature map.

The feature maps pixel values in the input layer are of low-level features, such as line segments.

The neurons (yellow) in hidden layer 1 are connected to all the green input neurons. Each of these neurons represents different higher-level features, such as an edge or a simple shape. Various hidden layer neurons also represent the same feature but in different locations in the image. The output of these neurons is a single value known as their activation, representing how confident the neuron is that the feature it is associated with is present. Neurons in later hidden layers are associated with increasingly high-level features.

1. The feature map pixel values from each green neuron are passed to the yellow neuron
2. The yellow neuron processes the inputs with an activation function. The output of the function is called the activation level. The activation level represents the probability that the feature associated with the yellow neuron is present in the image. If the activation level is

high, the neuron activates, which means it passes the activation level value to the neurons in the next layer that it is connected to. If the activation level is low, the neuron will pass the value of 0 to the next layer. The activation level is calculated like this:

a. The weighted sum of all the inputs is taken.

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n + b$$

A weight factor (w) is multiplied by each input (x). The value of the weight represents how important the input layer neuron's low-level feature feature is in contributing to the hidden layer neuron's higher-level feature. For example, a pixel value from a feature map for curves (input layer lower-level feature) would contribute significantly to making up a circle (hidden layer higher-level feature), and the weight would be high for that input. However, suppose the input pixel value is instead from a feature map for straight lines, it will not contribute much to making up a circle, and the weight value would be low.

In other words, many low-level features are looked at to see if they are part of a higher-level feature. If the low-level feature *is* part of a higher-level feature, like a curve is to a circle, then the weight assigned to it is high. If there are *many* of these relevant low-level features, then their weighted sum will be high.

The bias term (b) offsets the result of the weighted sum, giving the neuron's activation some flexibility (leeway). Even if the weighted sum is low, the neuron might still activate, because a constant, positive term is added to it.

b. The weighted sum is a single value, and it is put through the ReLU activation function. Recall from the "pooling" section that ReLU sets all negative numbers to 0. This is why if the feature the hidden layer neuron is associated with is not present, and the weighted sum is a low value (negative), the neuron will pass the value of 0 onto the next neuron.

3. The value of the activation level of each neuron in the first hidden layer is passed to each neuron in the second hidden layer. The second hidden layer's neurons perform the process described above but use the activation levels from the previous layer's neurons instead of a pixel value as input.
4. Features associated with the neurons become increasingly high-level as more hidden layers are used (from line segments to parts of edges, to whole shapes). Yolov8 segmentation models usually have about a dozen hidden layers.

Output Layer

Analyze the high-level features in the last hidden layer to classify the detection.

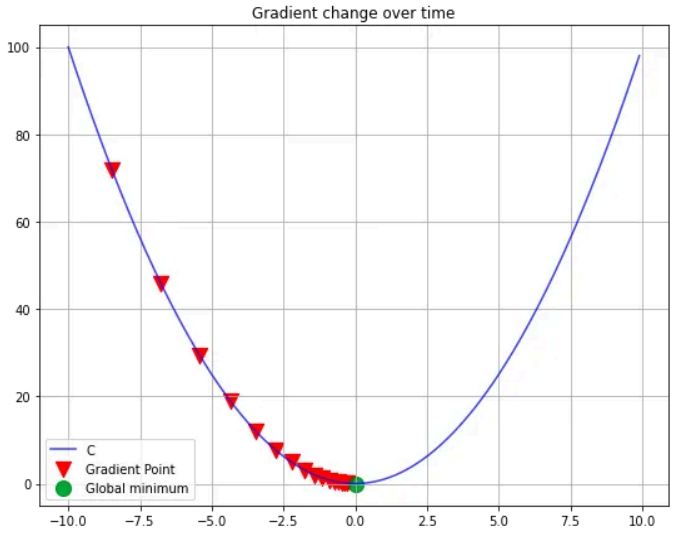
1. Each neuron in the last hidden layer is connected to each neuron in the output layer. The neurons in the output layer each represent a class. Each of the neurons will output the probability that the class it is associated with is present.
2. To calculate that probability, the same process described above is used, with the input to the output layer's neurons being the activation level of the neurons in the last hidden layer. For example, if the last hidden layer shows that an eye, a mouth, and a nose are likely to be present, then the probability of the class "face" being present will be high.

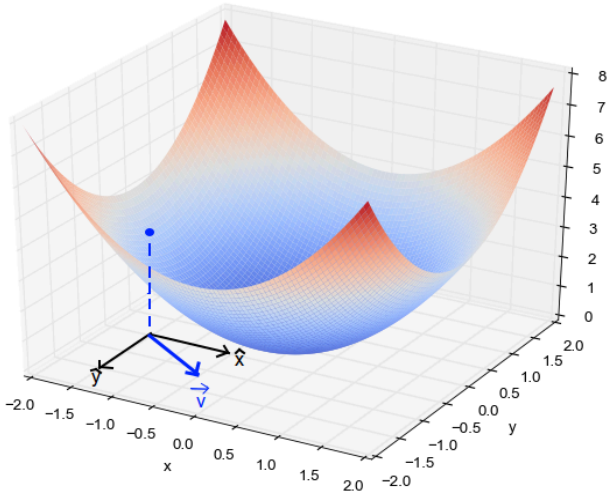
3. The softmax activation function is used instead of ReLU. While ReLU will make negative numbers 0, softmax squashes a list of numbers to a range between 0 to 1, while keeping the ratio between them. This method normalizes the outputs so that they represent the probability for each class.
4. The class with the highest probability is picked, and an outline of the object belonging to that class is generated (called a segmentation mask). This is possible because the model was keeping track of the coordinates of each feature that make up the object.

Training

The weights that are used in the fully connected layers and the convolutional layer are optimized during training.

- Weights are randomly generated at first. The weights and biases are adjusted by increments, with many cycles. These cycles are called epochs. This is what happens in each epoch:
- Backpropagation: The model, with the current weights, is used to analyze every image in a large dataset of annotated and labeled images. After the model runs, the loss is calculated based on what the classes and boundaries the model predicted, and what the actual values should be. Examples of loss include segmentation loss, classification loss, and focal loss, which are explained in depth later. The losses are passed through an optimizer, which calculates how each weight contributed to the loss. It calculates how much each weight and bias needs to be adjusted, and in which direction, to minimize loss. The optimizer uses a function called gradient descent. The following graphs show a visualization of minimizing the loss function.





Animation for gradient descent
https://miro.medium.com/v2/resize:fit:1400/format:webp/1*47skUygd3tWf3yB9A10QHg.gif

- The 2D plot shows the process of minimizing loss with one independent variable. The value of x is adjusted to minimize y. However, in a CNN model, there are multiple weights (independent variables) to adjust.

    - The 3D plot shows the function for minimizing loss with 2 independent variables, where both variables are taken into account when trying to minimize z. More independent variables will be impossible to visualize, as we cannot comprehend beyond 3D planes.
    - In each epoch, the weights and biases are adjusted slightly, and loss is decreased slightly.

- **Overfitting:** when a model becomes too accustomed to the training dataset. It starts memorizing the patterns in the training dataset instead of learning to detect those patterns anywhere. The dropout method is sometimes used to prevent overfitting. This method randomly shuts down neurons in the fully connected layers, which prevents the model from relying too much on one feature.

- **Underfitting:** underfitting is when the model cannot detect the patterns in any image that it is given. The solution is to train for more epochs.

Purpose

This project aims to create a prototype app that utilizes machine learning to allow the user to access nutritional information about their meals easily. Users can simply take a photo of their meal in order to know about its nutritional information, eliminating the need to manually input meal contents. This innovative approach aims to promote healthier eating habits, facilitate diet program management, and raise awareness about healthy eating.

Question

How would an AI estimate the percentage of different macronutrients that a meal contains by examining a picture of it?
How many epochs of training will optimize the model's performance?

Hypothesis

A yolov8 instance segmentation model could be trained to estimate the percentage of different macronutrients in a meal. This is because a segmentation model can be trained to detect certain food items in an image. It also has an advantage over regular object detection models as it can find the exact shape of the item detected in addition to drawing a bounding box around it. This allows for the size of the food item in proportion to the whole meal to be calculated. The nutrient percentage in each food item, adjusted for its size relative to the total size of the meal, can be summed to determine the overall percentage content for each nutrient in the meal.

Materials

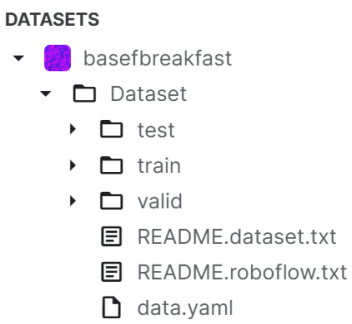
1. Install Visual Studio Code on a computer
    a. Install Python 3.11.3
    b. Install Ultralytics along with all required dependencies
2. Make an account with Kaggle and Roboflow (for annotating images)
3. Install Chrome Extension "Image Downloader" to gather images for the dataset from Google images

Constant Variables During Training

- Batch size: 16 (Batch size is the number of training images used in one iteration. Iterations run until all the training images are used, which is when 1 epoch is completed)
- Kernal dimensions 3x3, stride:1, padding type: replicate padding.
- Training patience: 100 (the number of epochs to continue after the validation loss starts flatlining/stops decreasing)
- Image size: 640. (All training images are resized to 640x640. This is a middle ground between high and low size. Low size increases training speed, large size increases attention to detail)
- Optimizer type: SGD (Stochastic gradient descent)
- Seed: 0 (seed for random number generator for starting point of weights, means all 4 models have same randomly initiated weights)
- When to disable mosaic augmentation: 10 epochs before finishing training (stabilizes model, makes model get used to regular and augmented images)
- Lr0: 0.01 (Initial learning rate, rate at which model adjusts weights during training, faster rate might result in overshooting)
- Dropout rate: 0 (dropout is when some input neurons are randomly omitted to prevent overfitting)

Procedure

1. Gathered a large number of images of common breakfast items, bacon, bananas, bagels, bread, and fried eggs, with 1000 instances for each class.
2. Created a dataset with these images
    a. Annotated all instances in Roboflow, by creating masks for all objects using SAM (segment anything model) and labeling them
    b. Applied augmentation steps to the annotated images, which include 10° vertical and horizontal shear, 0.5% noise, ±15° rotation, and 90° rotations clockwise and counterclockwise. Augmentation makes the dataset more diverse.
3. Downloaded dataset as a zipped/compressed folder, named it “BasefBreakfast”
4. Opened a notebook (coding platform) in Kaggle. Selected double T4 GPU as an accelerator.
5. Uploaded the zipped folder named “BasefBreakfast” to the Kaggle notebook. The file tree should look like this:



6. Edited the “data.yaml” file to look like this. This file is important when training the models, as it contains the paths to training and validation images.

data.yaml (158 B) /kagg

```
path: /kaggle/input/basefbreakfast
train: Dataset/train
val: Dataset/valid
test: Dataset/test

names:
  0: Bacon
  1: Bagel
  2: Banana
  3: Bread
  4: Egg
```

7. Trained 4 different yolov8 models on the dataset, each for a different number of epochs. All hyperparameters are kept at default during training. For the values of the hyperparameters, refer to this document: https://docs.ultralytics.com/usage/cfg/#train-settings
    a. Trained the 1st model for 100 epochs. Recorded performance indicators of the model, namely loss curves, mAP score, F1 score, and the confusion matrix.
    b. Trained the 2nd model for 200 consecutive epochs. Recorded the same performance indicators.

    c. Trained the 3rd model for 100 epochs, stopped the training, and then trained for another 50 epochs. Recorded the same performance indicators.
    d. Trained the 4th model for 100 epochs, stopped the training, and then trained for another 100 epochs. Recorded the same performance indicators.
    e. Compared the loss curves of the 4 models, and chose the one with the lowest segmentation loss and class loss as the best model.
    f. Download the weights of the best model.
8. In VS code, created a yolov8 segmentation model using the best weights from the last step.
9. Utilized this model to analyze percentages of macronutrients in meals (fiber, sugar, fat, protein, and carbohydrates:
    a. Took a picture of a meal consisting of bacon, fried eggs, bread, bagels, and/or bananas and inputted it into the yolov8 model
    b. The model returned the names of the food items detected and the segmentation boundaries of each of the items detected.
    c. Determined the percentage of macronutrients in each food item using the Edamam API
    d. Determined the percentage of macronutrients in the entire meal by taking into account the proportion of each food item in the entire meal, and the percentage of each macronutrient in that food item.

Results - Loss Curves and Other Metrics

Loss: “How sure the model is of itself that it predicted correctly, and whether or not it’s actually correct”
Segmentation loss:
Segmentation loss tells us how close the predicted segmentation mask is to the ground truth masks. It measures how effectively the model performs segmentation. This function, called **pixel-wise cross-entropy loss**, is used to calculate the loss for each pixel in the predicted mask (the probability that the pixel is predicted to be part of a certain class and whether it actually is part of that class).

$$-\sum_{i=1}^{C} y_i \log(\hat{y}_i) = -y_1 \log(\hat{y}_1) - y_2 \log(\hat{y}_2) - \ldots - y_C \log(\hat{y}_C)$$

Where

- C is the number of classes, in this case, 5
- i is the index of the class
- $\hat{y}_i$ represents the probability that the pixel belongs to class i (a number between 0-1)
- y_i represents the actual probability that the pixel belongs in class i. This is either 0 or 1, and this value is taken from the ground truth labels.

The logarithm function means that the higher the value of $\hat{y}_i$ is, the lower the output is. So, if $y_i = 1$ and $\hat{y}_i$ is high, then the loss is low. The loss of each pixel is summed to calculate the segmentation loss.

A simpler segmentation loss function that is applied along with pixel-wise cross-entropy loss is called the **IoU (intersection over union)**. This is the ratio between the overlap between the ground truth mask and the predicted mask (intersection) and the total area that is covered by both the ground truth mask and the predicted mask combined (union).

Classification loss:
Classification loss calculates the difference between the predicted label and the actual label for an object that was detected. Below is the function applied to a single instance that was detected

$$-\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where

- C is the number of classes, in this case, 5
- i is the index of the class
- $\hat{y}_i$ represents the probability that the object detected belongs to class i (a number between 0-1)
- y_i represents the actual probability that the pixel belongs in class i. This is either 0 or 1, and this value is taken from the ground truth labels.

If $y_i = 1$ and $\hat{y}_i$ (predicted probability) is high, the logarithm function will return a low number, meaning loss is low and the class was correctly predicted. If $y_i = 1$ and $\hat{y}_i$ is low, the logarithm will output a high number, meaning loss is high and that the model is predicting an incorrect class with high probability/confidence.

$$-\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{C} y_{n,i} \log(\hat{y}_{n,i})$$

This is the full function, where N is the total number of instances detected across multiple images (batch size). This function takes the sum of the loss value from each instance and then averages it. This final value is classification loss.

Distribution focal loss (dfl.):
Some instances might have low prediction confidence and high loss because they are cut off or blurry, or they are underrepresented in the dataset. Focal loss aims to have the model better recognize those hard cases.

Focal loss calculates classification loss as explained above, but will make low loss a lot lower, and high loss only slightly lower. When training a model, a function takes the loss values for each instance and adjusts the weights accordingly. Instances with higher loss contribute more to adjusting the weights. Focal loss makes high losses higher compared to low losses, which means the model will put more priority on adjusting weights to detect the instances with low confidence, as those have higher losses.

The function for distribution focal loss is the same as cross-entropy classification loss, but with a modulating factor that is multiplied by the original function.

$$\mathrm{FL}(p_t) = -(1 - p_t)^{\gamma} \log(p_t)$$

Where:

- FL stands for Focal Loss
- p_t represents the probability that the object detected belongs to class t (a number between 0-1)
- Gamma represents the magnitude by which it increases the loss for low-confidence classes, usually a value of 2

Precision (M):
Out of all the instances that were predicted, how many were correctly classified to the right class. Note precision doesn't take into account false negative instances.
(# true positives) / (# true positives + # false positives)

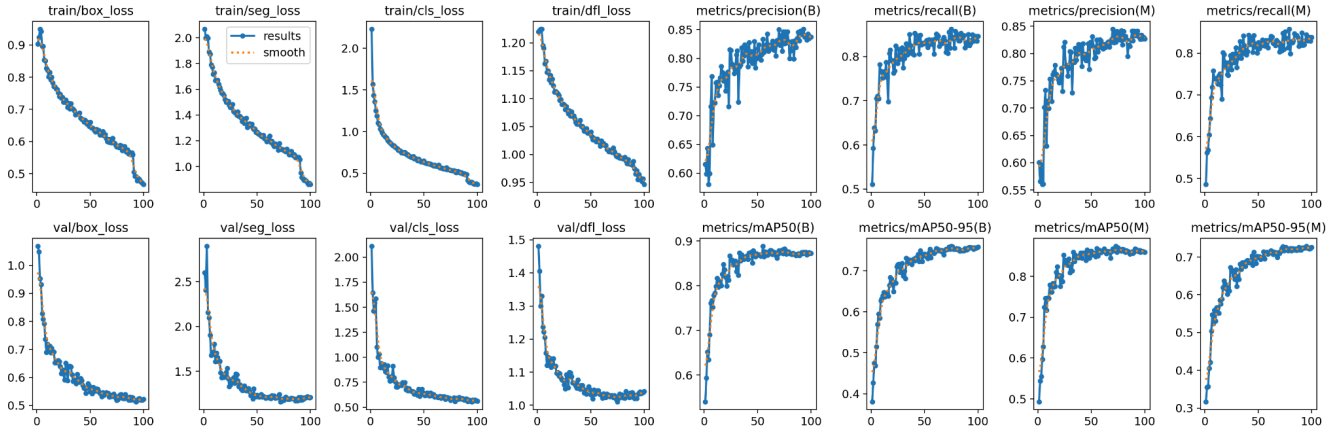
Recall (M):
Out of all the instances in the validation dataset (even the cases that the model failed to detect), how many were predicted (found) and classified correctly?
(# true positives) / (# true positives + #false negatives)

mAP50(M):
Precision for instances that have 0-50% overlap with another instance.

mAP50-95(M):
Precision for instances that have 50-95% overlap with another instance.

Note that a metric labeled (M) means it is an average of all classes, as opposed to metrics labeled (B), which applies to only a single class.

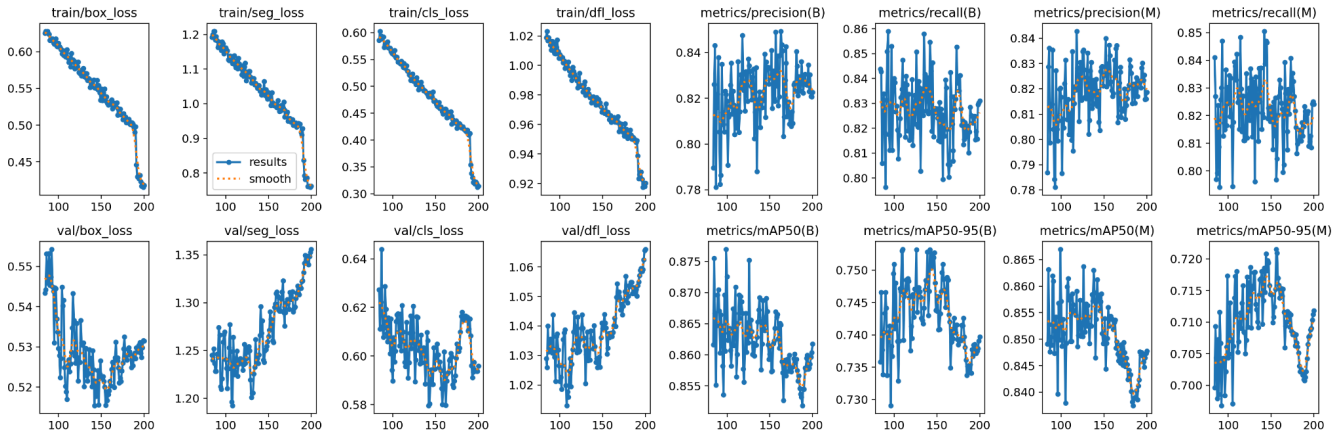
Figure 1: Training for 100 epochs, epochs 0-100



Notes:

- A steep decline in training losses during the last 10 epochs was due to albumentations shutting down during that time. Albumentarions is an augmentation Python library (that adds diversity to the dataset, by adding noise, rotations, shear, etc). Used to prevent overfitting. Albumentations shutting down makes training images much easier to detect. Performing better on unaugmented images might indicate too small of a dataset OR not enough epochs of training.
- Dfl especially high, and starting inclining at the end, should train for more epochs

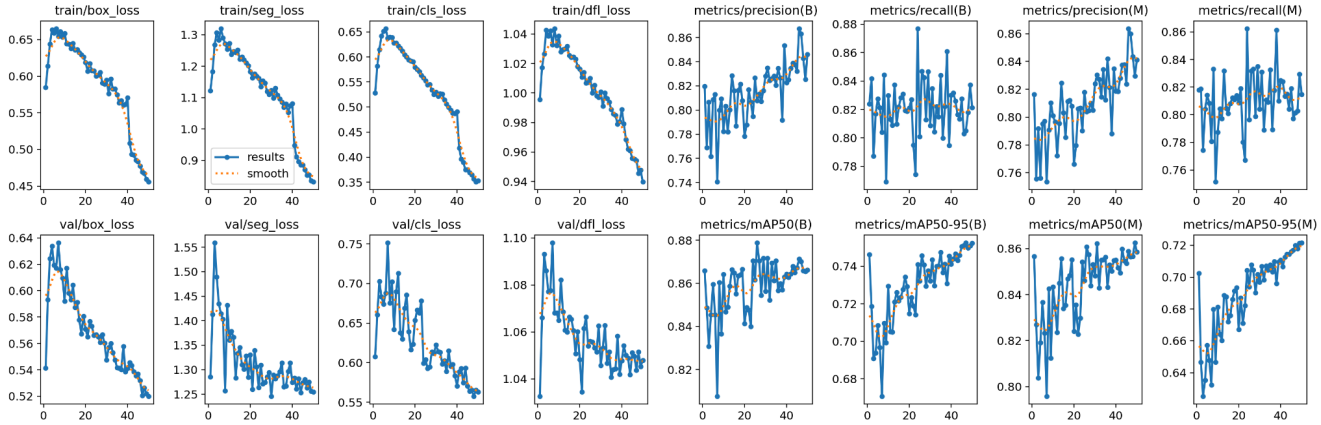
Figure 2: Training for 200 epochs, epochs 0-200



Notes:

- The model is overfitted (too accustomed to training images that it is unable to predict new images) indicated by decreasing training losses and rapidly increasing validation losses after 100 epochs
- Validation cls loss is not increasing as others, but has fluctuations, could also indicate overfitting
- Large fluctuations for recall rates specifically, within 0.8 to 0.85. It means the model is inconsistent with identifying all instances that are present in the image, which is also a result of overfitting, as this started to happen also at the 100 epochs mark

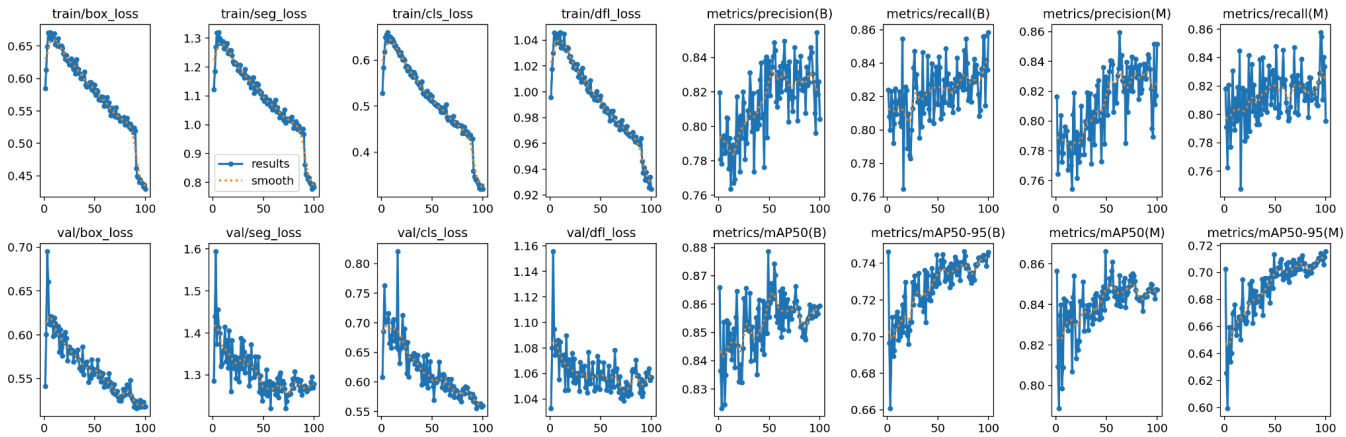
Figure 3: Trained for 100 epochs, ended, then trained another 50 epochs. Epochs 100-150



Notes:

- Trained for 100 epochs, stopped, loaded finished weights in a separate model, and trained that for 50 more epochs. During those last 50 matches. The weights will be decently accurate, but the training images will be completely new to the model, which prevents overfitting (the model becoming too attached/accustomed to the training data)
    - This method is effective, with loss much lower than continuous training for 150 epochs. Ie. cls loss 0.55 for this vs 0.62 for 150 epochs in a row
    - While loss decreased, precision and recall stayed much the same from the 100 epochs mark
- Fluctuations and increase in loss, in the beginning, were because the training function takes a few iterations to understand which direction (increase/decrease) to adjust each weight in order to make the model more effective. It does not yet know that, because information from the first 100 epochs training function does not carry over into the last, separate 50 epochs
- Graphs make fluctuations during the last 20 epochs look large for dfl loss, but they are really small and can be ignored. The range in which it fluctuates is 1.04 and 1.06.

Figure 4: Trained for 100 epochs, ended, then trained another 100 epochs. Epochs 100-200



Notes:

- Curves extremely similar compared to the 100+50 epochs model, basically same results
- Exact results in the table below, will help decide which is the best model

Choosing the Best Model

Table 1: Final loss values for each model

		seg loss	cls loss	dfl loss
Model 1	Training	0.8	0.3	0.95
	Val	1.2	0.6	1.05
Model 2	Training	0.7	0.3	0.92
	Val	1.35	0.59	1.07
Model 3	Training	0.8	0.35	0.94
	Val	1.25	0.55	1.05
Model 4	Training	0.78	0.3	0.92
	Val	1.3	0.56	1.06

Table 2: Other metrics for performances of each model

	Precision (M)	Recall (M)	mAP50 (M)	mAP50-95 (M)
Model 1	0.84	0.83	0.85	0.73
Model 2	0.8	0.81	0.85	0.71
Model 3	0.85	0.81	0.85	0.72
Model 4	0.8	0.81	0.85	0.72

Figure 5: Model 1 performance on image

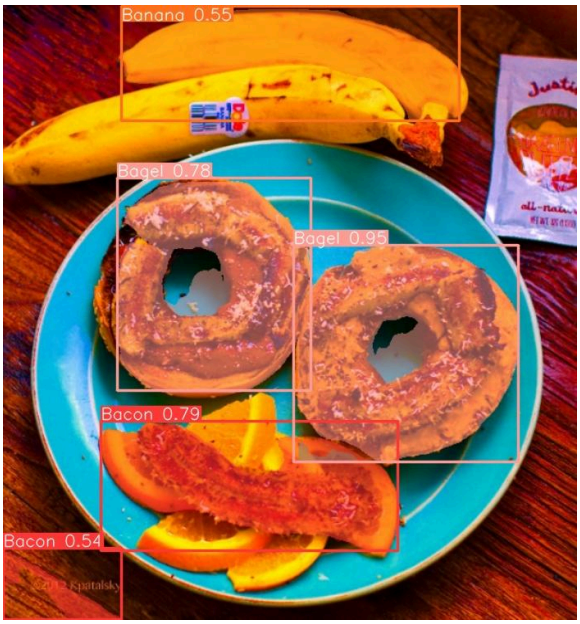


Figure 6: Model 2 performance on image

Figure 7: Model 3 performance on image

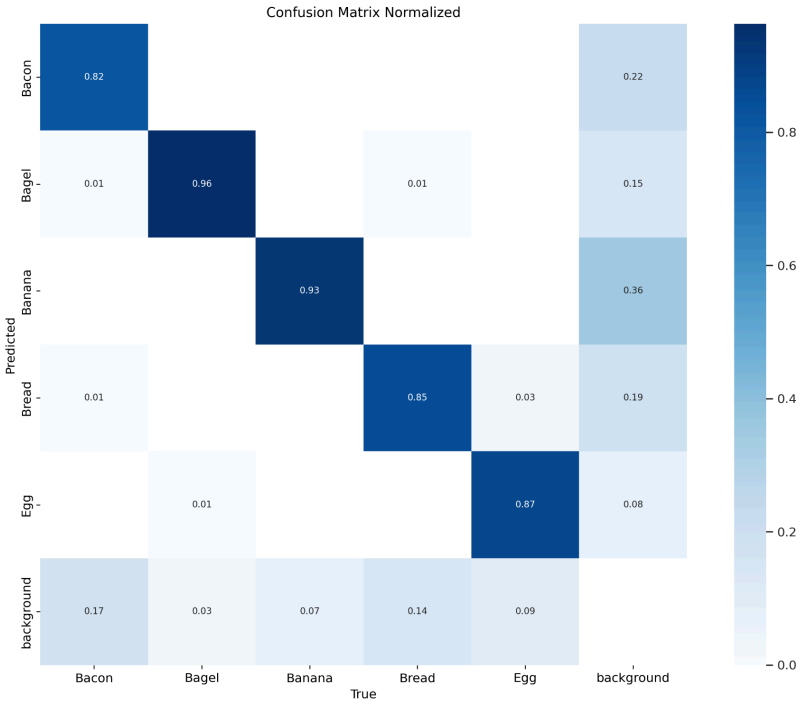
Figure 8: Model 4 performance on image

- Model 2 (200 epochs) is eliminated as it has the highest segmentation, classification, and distribution focal loss (dfl), and as can be observed from graphs, it is overfitting
- Minimizing classification loss is most important for this project, as different foods contain drastically varying amounts of nutrients. Distribution focal loss is also important, as it shows how well the model detects hard cases. In food detection, this indicates how well it detects foods that are blocked by other things on the plate. Even a 0.01 difference in loss could significantly impact performance.
    - Models 3 and 4 have a significantly lower classification and focal loss than other models, at 0.55 and 0.56 respectively for classification loss, and 1.05 and 1.06 for dfl. Model 3 has lower seg, cls, and focal loss than model 4
    - Model 3 also has a significantly higher precision than model 4 at 0.85 vs 0.8.
    - The segmentation mask of the objects is allowed to be slightly off from the actual masks, as this project only aims to *estimate* the nutrients. In fact, basing food item proportions on a 2D image already means the nutrient percentage calculations are an estimation. So, even though model 3 has a higher segmentation loss than model 1, it is picked for its lower classification loss.
- In the test image, all models except model 3 had false positives (models 1 and 4) and false negatives (models 1, 2, and 4)

<u>Model 3</u> (trained for 100 epochs, stopped, then trained for 50 epochs), is chosen as the best model.

Confusion Matrix and Potential Solution to Misidentification

Figure 9: Confusion matrix for model 3, normalized



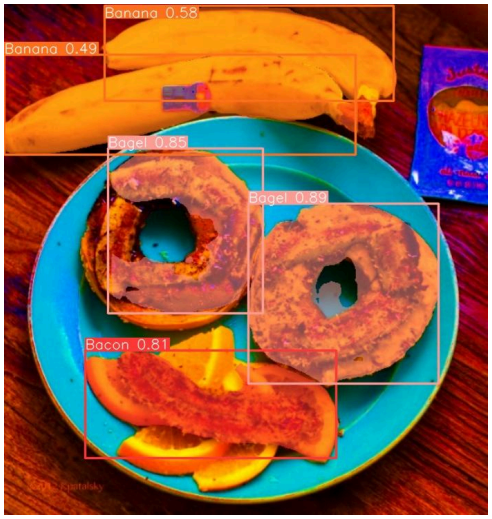
A confusion matrix shows the number of false positives, false negatives, and true positives for each class. This confusion matrix has been normalized, which is dividing each element by the sum of the elements in its corresponding row.

Bacon can be misidentified as bagel or bread (likely sides of bread), bagels misidentified for eggs, and vice versa (similar round shape). Since the similar shapes are confusing, color needs to be adjusted to prevent this confusion.

The red tones of the image were saturated to enhance that color, to help differentiate bacon from bread. Bacon (reddish brown) will become significantly more red, while the colors of bread and bagels (brown) won't be affected as much. Creates contrast. Saturation is performed on the image before it is inputted into the model.

Figure 10: Detection no red saturation

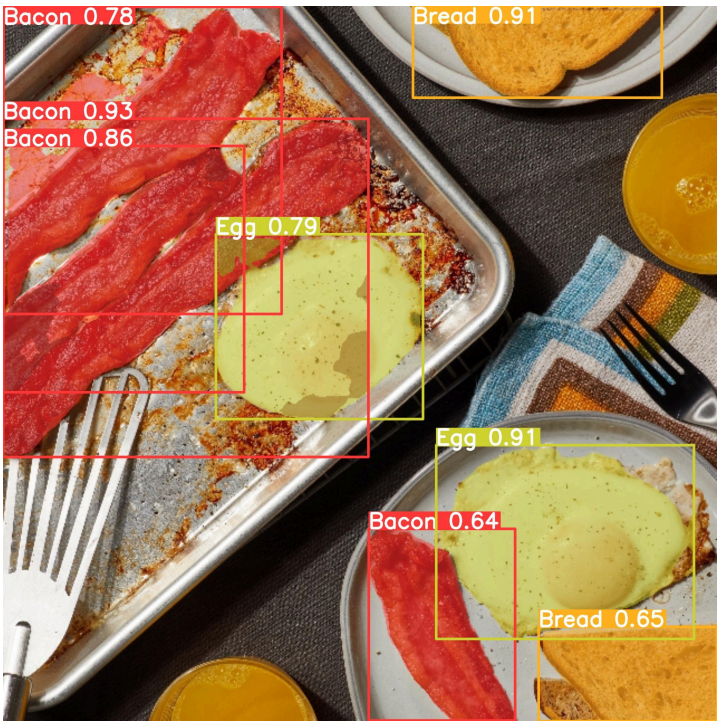
Figure 11: Detection with red saturation



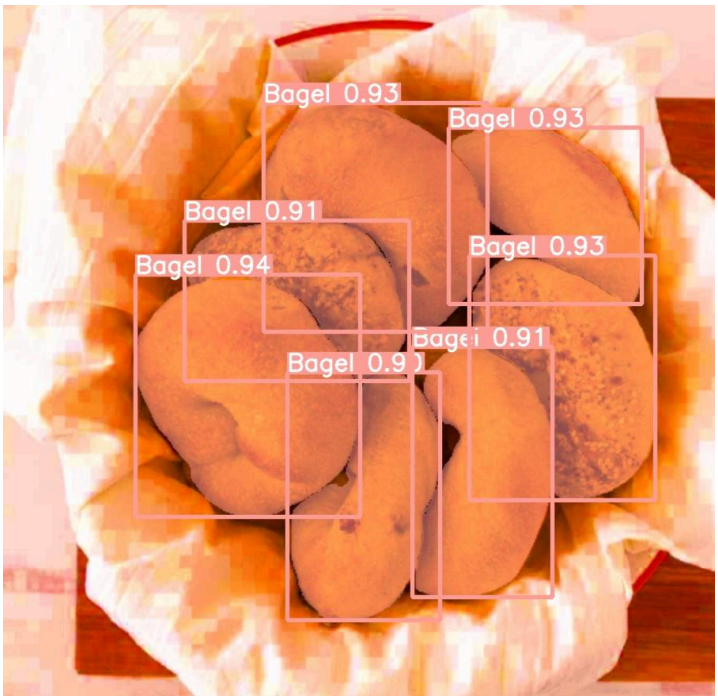
Red saturation tends to negatively impact segmentation effectiveness but makes the model better at classification. Only fixed the issue of distinguishing between bacon, bagel, and bread. DID NOT address bread vs egg, and bread vs bagels classification. A side effect is making eggs harder to detect (alter white color). A potential solution to investigate, but not completely viable. A better solution to the confusion issues will be to train a larger dataset.

Conclusion

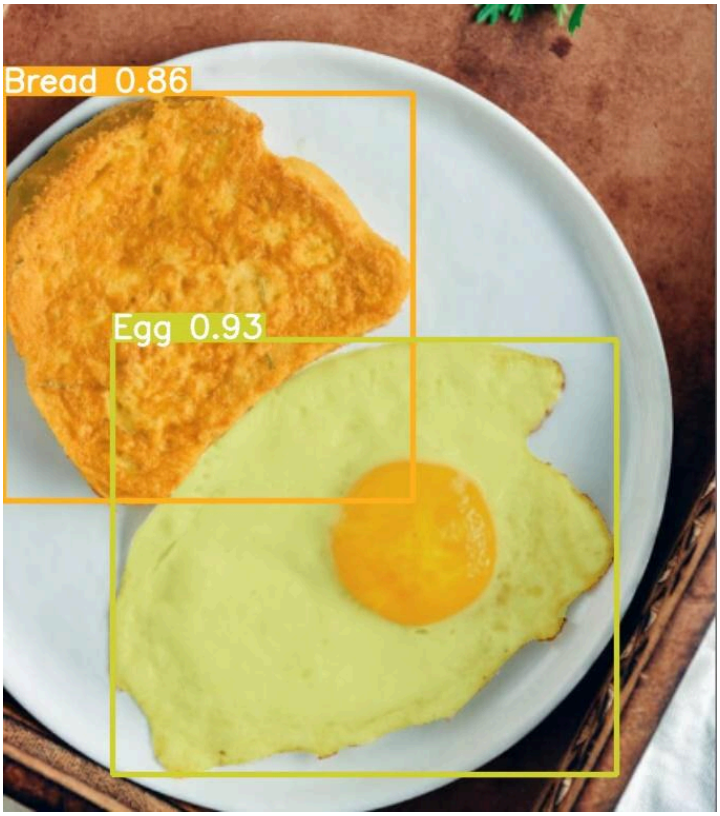
After analyzing the data above shows the model performs adequately at detecting, classifying, and creating segmented masks of breakfast food items. For future improvements, a larger dataset, preferably with 2000 instances of each food, should be used for training to further minimize the losses (and to prevent occasional false negatives). Below is an example of the prototype app in action:




```
Total Bacon
Protein: 13.7%
Fat: 37.1%
Carbohydrates: 0.0%
Sugar: 0.35%
Dietary fiber: 0.0%

Total Egg
Protein: 12.6%
Fat: 9.51%
Carbohydrates: 0.72%
Sugar: 0.37%
Dietary fiber: 0.0%

Total Bread
Protein: 10.7%
Fat: 4.53%
Carbohydrates: 47.5%
Sugar: 5.73%
Dietary fiber: 4.0%

Entire Meal
Total protein: 13.05% of your meal
Total fat: 27.33% of your meal
Total carbohydrates: 7.28% of your meal
Total sugar: 1.16% of your meal
Total dietary fiber: 0.6% of your meal
```



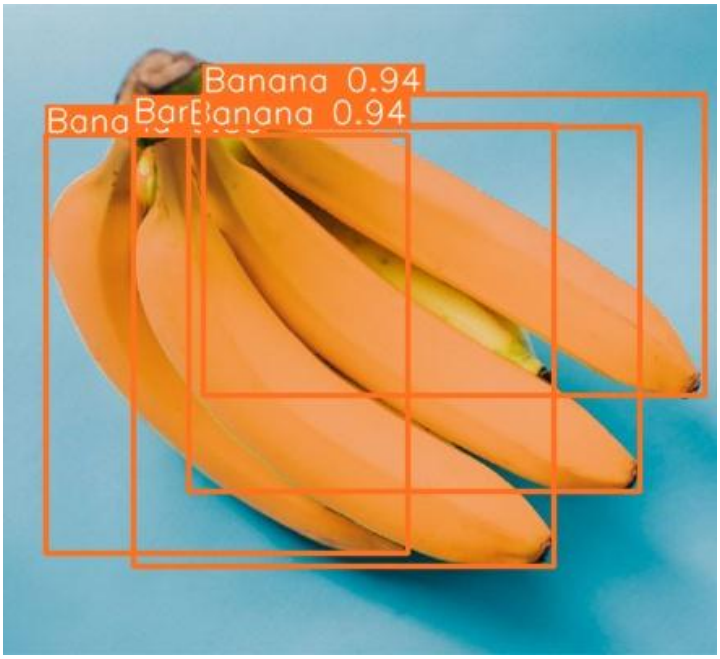

```
Total Bagel
Protein: 10.6%
Fat: 1.32%
Carbohydrates: 52.4%
Sugar: 8.43%
Dietary fiber: 1.6%

Entire Meal
Total protein: 10.6% of your me
Total fat: 1.32% of your meal
Total carbohydrates: 52.4% of y
Total sugar: 8.43% of your meal
Total dietary fiber: 1.6% of yo

PS C:\Daniel\Nutrition YOLO>
```



Total Egg
Protein: 12.6%
Fat: 9.51%
Carbohydrates: 0.72%
Sugar: 0.37%
Dietary fiber: 0.0%

Total Bread
Protein: 10.7%
Fat: 4.53%
Carbohydrates: 47.5%
Sugar: 5.73%
Dietary fiber: 4.0%

Entire Meal
Total protein: 11.8% of yo
Total fat: 7.43% of your m
Total carbohydrates: 20.3%
Total sugar: 2.61% of your
Total dietary fiber: 1.67%

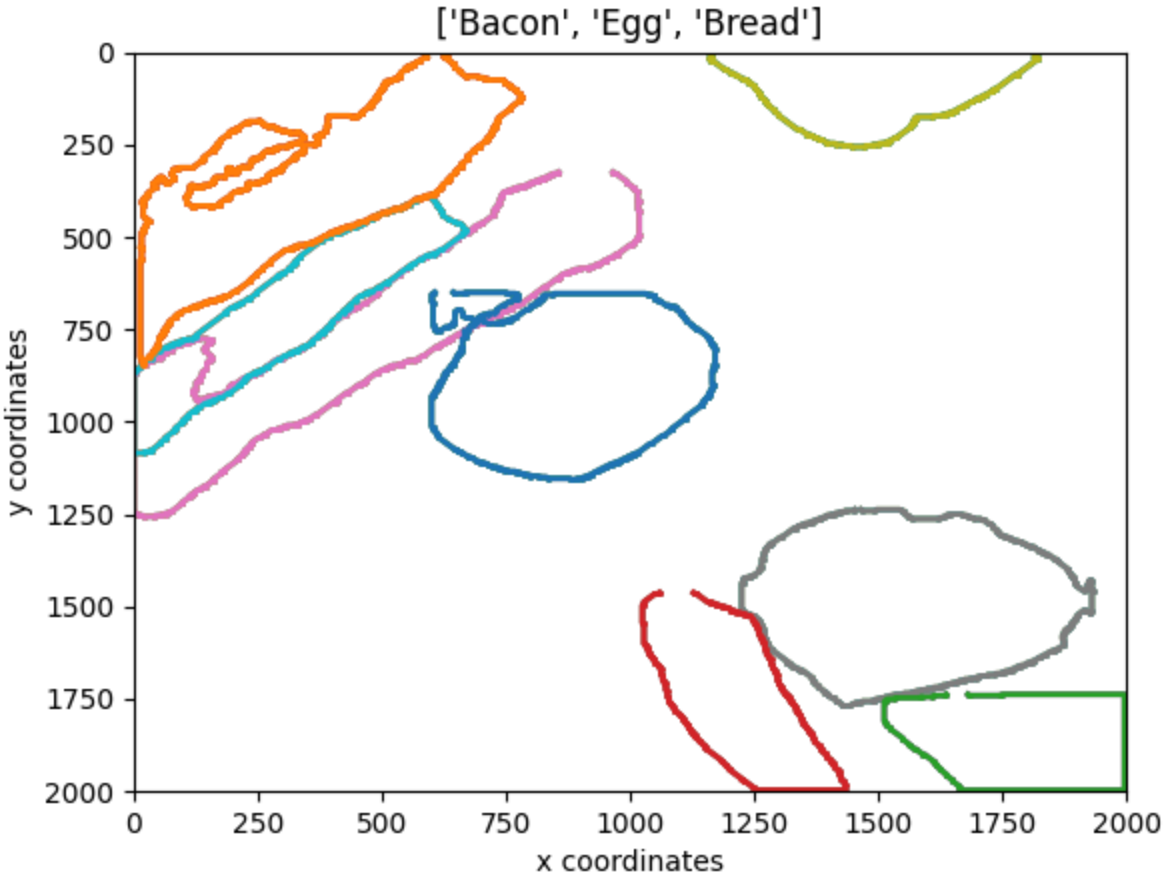


Total Banana
Protein: 1.09%
Fat: 0.33%
Carbohydrates: 22.8%
Sugar: 12.2%
Dietary fiber: 2.6%

Entire Meal
Total protein: 1.09% of
Total fat: 0.33% of you
Total carbohydrates: 22
Total sugar: 12.2% of y
Total dietary fiber: 2.

Extra Note: Area Calculations for Segmentation Masks

Figure 12: Visualization of the mask area calculations



1. After segmentation is performed, the model returns a list of coordinates for the mask (outline) of each item. These coordinates represent the location of the pixel in the image
2. The coordinates are plotted on the x and y axis, where the x and y axis are the length of the width and height of the image respectively. Each increment on the axis is 1 pixel.
3. The shoelace theorem is used to calculate the area of each item

$$A = \frac{1}{2}\left|(x_1y_2 + x_2y_3 + \cdots + x_ny_1) - (y_1x_2 + y_2x_3 + \cdots + y_nx_1)\right|$$

   Where:
   - n is the number of points on the outline of the shape
   - (x_1, x_2, x_3 … x_n) is a list containing the x coordinate values, listed in clockwise order
   - (y_1, y_2, y_3 … y_n) is a list containing the y coordinate values, listed in clockwise order

4. The area of the food item in proportion to the entire is calculated through (area of one food item) / (sum of area of all food items)

References

Anello, E. (2021, June 17). *Visualizing the Feature Maps and Filters by Convolutional Neural Networks*. Medium. Retrieved from https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e

Baheti, P. (2021, May 27). *Activation Functions in Neural Networks [12 Types & Use Cases]*. V7 Labs. Retrieved from https://www.v7labs.com/blog/neural-networks-activation-functions

Brownlee, J. (2020, August 2). *How to Calculate Precision, Recall, and F-Measure for Imbalanced Classification - MachineLearningMastery.com*. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/

Convolutional Neural Network. Learn Convolutional Neural Network from… | *by dshahid380*. (2019, February 24). Towards Data Science. Retrieved from https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529

Convolution and ReLU. Convolution is a mathematical operation… | *by Dhanush Kumar*. (2023, November 23). Medium. Retrieved from https://medium.com/@danushidk507/convolution-and-relu-fb69eb78dd0c

Edamam — Food Information API. Edamam is a food information API that… | *by World In Data*. (2023, October 30). Medium. Retrieved from https://medium.com/@worldindata/edamam-food-information-api-ffe3679357b

Flores, O. (2023, April 9). *Understanding the Concept of Weight Sharing in CNN and RNN*. Medium. Retrieved from https://medium.com/@tarek.tm/understanding-the-concept-of-weight-sharing-in-cnn-and-rnn-f48bd3f76d35

Germanov, A. (2023, May 4). *How to Detect Objects in Images Using the YOLOv8 Neural Network*. freeCodeCamp. Retrieved from https://www.freecodecamp.org/news/how-to-detect-objects-in-images-using-yolov8/

Gradient Descent vs. Backpropagation: What's the Difference? Analytics Vidhya. Retrieved from https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/

guide, s. (2019, July 12). *Convolutional Neural Network for Object Recognition and Detection*. Medium. Retrieved from https://medium.com/@ringlayer/convolutional-neural-network-for-object-recognition-and-detection-126a22af8975

Gurucharan, M. (2022, July 27). *Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network*. upGrad. Retrieved from https://www.upgrad.com/blog/basic-cnn-architecture/

Importance of Neural Network Bias and How to Add It. (n.d.). Turing. Retrieved from https://www.turing.com/kb/necessity-of-bias-in-neural-networks

Jain, S. (2023, April 21). *CNN | Introduction to Pooling Layer*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

Jiang, L. (2020, June 7). *A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam)*. Towards Data Science. Retrieved from https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

Menta, S. T. (2022, September 29). *Upsampling and Transposed Convolutions Layers | by Surya Teja Menta*. Medium. Retrieved from https://medium.com/@suryatejamenta/upsampler-and-convolution-transpose-layers-7e4346c05bb6

Rosebrock, A. (2021, May 14). *Convolutional Neural Networks (CNNs) and Layer Types*. PyImageSearch. Retrieved from https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/

7.3. Padding and Stride — Dive into Deep Learning 1.0.3 documentation. (n.d.). Dive into Deep Learning. Retrieved from https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html

Torres, F. (2023, October 16). *Convolutional Neural Network From Scratch | by Luís Fernando Torres | LatinXinAI*. Medium. Retrieved from https://medium.com/latinxinai/convolutional-neural-network-from-scratch-6b1c856e1c07

Unzueta, D. (2022, October 18). *Fully Connected Layer vs Convolutional Layer: Explained*. Built In. Retrieved March from https://builtin.com/machine-learning/fully-connected-layer

What Is a Convolutional Neural Network? | 3 things you need to know. (n.d.). MathWorks. Retrieved from https://www.mathworks.com/discovery/convolutional-neural-network.html

Zhou, V. (n.d.). *Machine Learning for Beginners: An Introduction to Neural Networks*. Towards Data Science. Retrieved from https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9