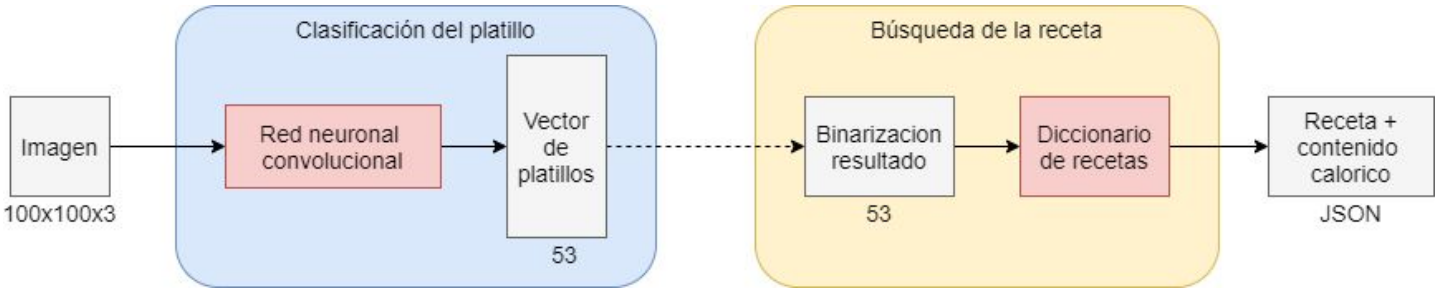
Alumno: Ramos Diaz Enrique

Reporte Proyecto Final: Comida Mexicana

1. Etapa Conceptual

El objetivo de este proyecto es **identificar platillos típicos de la gastronomía mexicana por medio de una clasificación a partir de una imágen o fotografía** del mismo utilizando una red neuronal convolucional, y además brindar su receta más tradicional **junto al contenido calórico de cada ingrediente** que la conforman por medio de la consulta en un diccionario, utilizando el resultado obtenido por la red como identificador.

En la siguiente imágen se describe el diseño conceptual del proyecto:



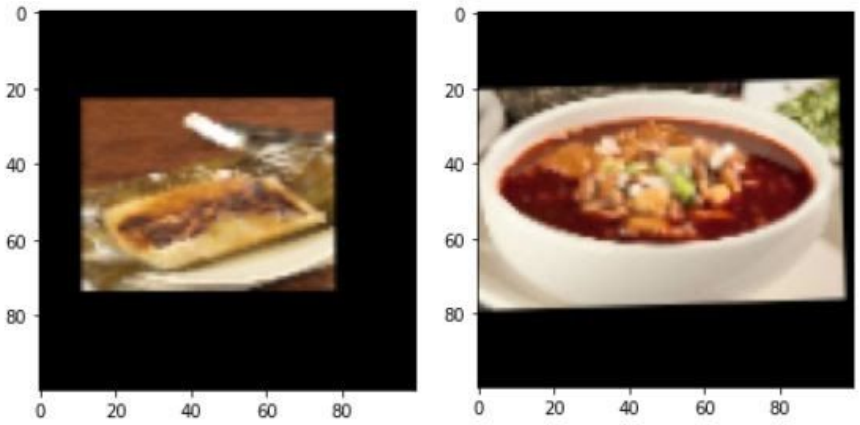
El diccionario de recetas posee un total de 53 platillos disponibles:

1. Birria
2. Burrito Arrachera
3. Camarones Empanizados Al Coco
4. Chicharron Preparado
5. Chongo Zamorano
6. Coctel Camaron
7. Elote Preparado
8. Enchiladas Huastecas
9. Enchiladas Verdes
10. Gordita Chicharrón
11. Guacamole
12. Huevos Rancheros
13. Milanesa
14. Mixiote
15. Mojarra Mojo Ajo
16. Mole De Olla
17. Mole Poblano
18. Molletes
19. Pambazos
20. Picadillo
21. Pico Gallo
22. Pozol
23. Pozole Verde
24. Quesadillas De Pollo
25. Sopes
26. Tacos Al Pastor
27. Tacos Dorados
28. Tamales Oaxaqueño
29. Tampiqueña Res
30. Tlacoyos
31. Torta Ahogada
32. Tostadas
33. Ajo Comino
34. Arroz Rojo
35. Chicharrón En Salsa Verde
36. Chilaquiles Verdes
37. Chiles En Nogada
38. Chiles Rellenos
39. Cochinita Pibil
40. Enfrijoladas
41. Esquites
42. Frijoles Olla
43. Huaraches
44. Huevos a la mexicana
45. Pan De Muerto
46. Pastel azteca
47. Pechuga Asada
48. Pescado Veracruzana
49. Pozole Rojo
50. Romeritos
51. Tinga De Pollo
52. Tlayudas
53. Verduras salteadas

1.1 Datos de entrada

Las entradas de la red son imágenes a color (RGB, 3 canales) de 100x100 en formato JPEG, normalizadas (divididas entre 255) y como arreglos de numpy de tipo float32.

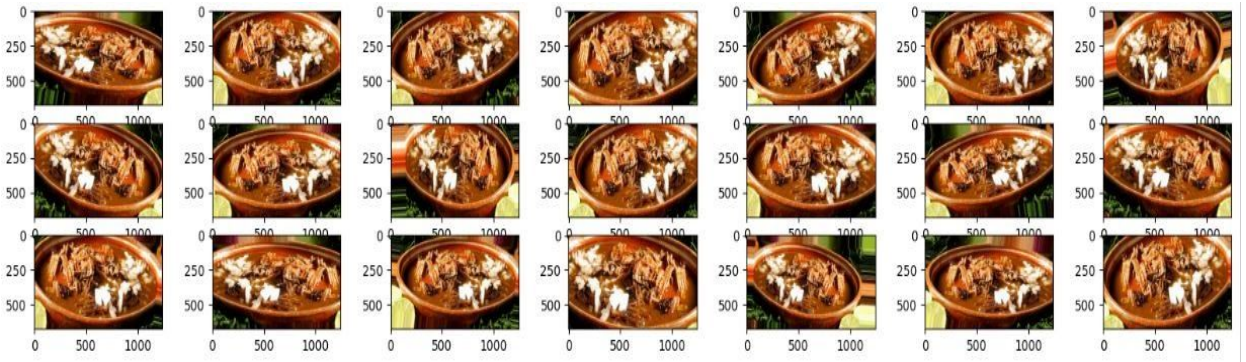
Esto último porque así lo requiere la implementación de Tensorflow. Así lucen estas imágenes:



Se decidió dejar las imágenes en RGB porque **en la comida el color es la característica más importante** para distinguir entre distintos tipos de esta, incluso más que la forma o la textura, por eso mismo se omitieron filtros como escala de grises (que pudiesen confundir a la red y tener malos resultados) o bordes (que no brindan suficiente información para la clasificación).
En otros trabajos similares de detección de alimentos simples, como frutas o verduras, se utilizan dimensiones cercanas a 200x200; para este trabajo se decidió dejarla en **100x100**, para lograr un **equilibrio entre tamaño en memoria y calidad de la misma.**

1.2 Ajustes en el preprocesamiento

En el procesamiento se aplicó **data augmentation**, pues se considera que 50 imágenes no son suficientes para que la red logre aprender correctamente durante el entrenamiento, además de reducir el tiempo y esfuerzo del mismo.
Utilizando métodos de Keras, se generaron **21 variantes de cada imágen**, teniendo así un total de **1100 imágenes por clase, en total 58300**. Estas variaciones se muestran enseguida:



Posteriormente, se evitó estirar o redimensionar las imágenes para lograr las dimensiones deseadas.

Se establecieron dimensiones “umbrales”, siendo estas de 500x500 pixeles. Así, a aquellas imágenes que fueran de menor tamaño se les agregaron a los **bordes píxeles en color negro para rellenar la imagen**; por el contrario, aquellas que fueran de mayor tamaño, fueron **recortadas**. Luego, fueron reescaladas a 100x100. La **normalización de las imágenes** es para reducir el tamaño en memoria de estas, y para aplicar data augmentation este es un requisito, además de brindarle a la red **valores numéricos cercanos (de 0 a 1) y así facilitar el aprendizaje.**

Para las etiquetas de cada imagen, es decir, su platillo, se crearon **arreglos numpy con dichas clases codificadas en OneHot**, teniendo un **vector de 53 posiciones** para cada imágen.

Finalmente, fueron **divididas en 2 conjuntos**: uno para entrenamiento y otro para pruebas, con random state constante (de modo que esta división siempre sea igual en cada ejecución) y con una relación **80% - 20%.**

1.3 Datos de salida

La salida de la red es un vector de tamaño 53, en donde cada posición representa un platillo disponible del diccionario de recetas presentado.
La posición con el valor más alto corresponde al platillo identificado por la red para dicha imágen. (El orden de los platillos respecto a las posiciones del arreglo es el mismo que el enlistado arriba).

2. Arquitecturas de red preliminares

2.1 Propuesta 1: Basándose en la red de los Simpsons = 8 mil características

La primera propuesta para la arquitectura de la red se basó en el **problema de clasificación de personajes de los Simpsons**, en donde en la última capa convolucional (es decir, la que fue aplanada para ser parte del clasificador) fue de tamaño 8x8x128 = **8192 características.**

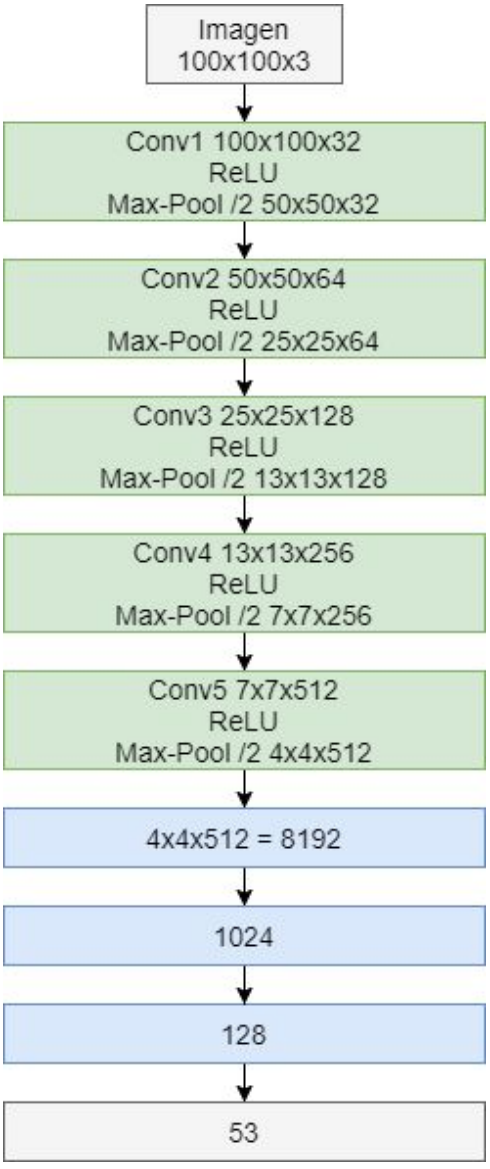
Entonces, esta primera propuesta se hizo con la intención de llegar a aproximadamente 8 mil características para ingresar al clasificador:

- Al ser la imagen de entrada de tamaño 100x100, para el extractor de características se decidió implementar **5 capas convolucionales.**
- Tamaño de kernel 3x3 y stride de 1.
- Al final de cada una de estas capas se aplica la función de activación **ReLU.**
- Cada capa convolucional tiene su respectivo **max-pooling** con un stride de 2x2, de modo que al final se tiene una capa de tamaño 4x4x512 = 8192.
- Para el clasificador se utilizan **2 capas ocultas directamente conectadas con ReLU.** Además de **1 capa de salida con 53 neuronas.**
- **Optimizador Adam** con un **learning rate de 0.001**, intentando lograr un equilibrio entre eficiencia y velocidad de convergencia.

Alumno: Ramos Diaz Enrique

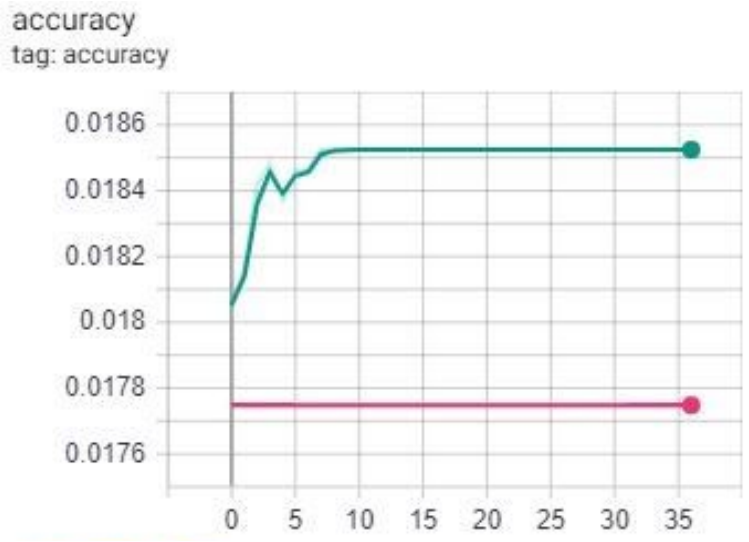
- Función de error: **categorical cross entropy**, que sirve para problemas de clasificación con muchas clases.
- Accuracy: **categorical accuracy.**

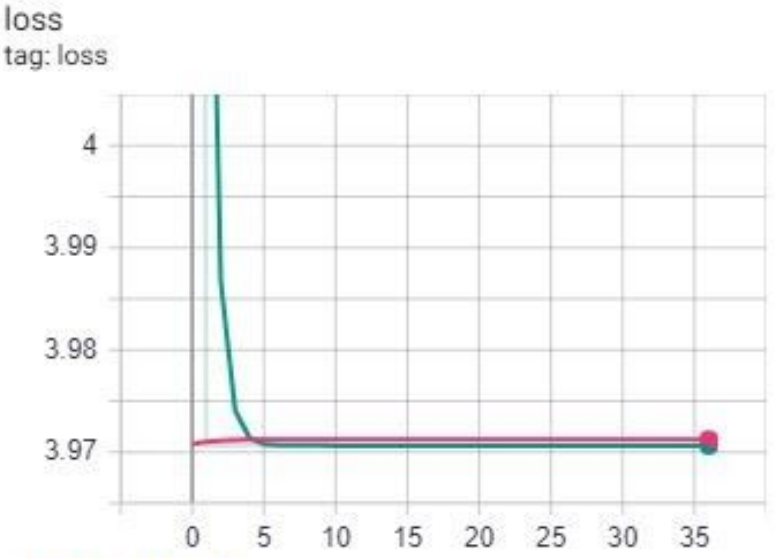
NOTA: Cabe mencionar que en la inicialización de los pesos de todas las arquitecturas, estos fueron multiplicados por 0.01 para acercarlos lo más posible a cero, y así lograr una convergencia más rápida durante los entrenamientos.



Se decidió utilizar **tamaños de batch pequeños**, pues benefician al aprendizaje de la red durante el entrenamiento.

Al entrenar con 35 épocas para tener una perspectiva de cómo se irá comportando el entrenamiento, y utilizando un **batch de tamaño 53, la precisión se mantenía constante en 1.83% y la pérdida se quedaba estancada en 3.98%**, tanto en entrenamiento (curva verde) como en prueba (curva rosa), teniendo el problema de **underfitting** con este diseño.

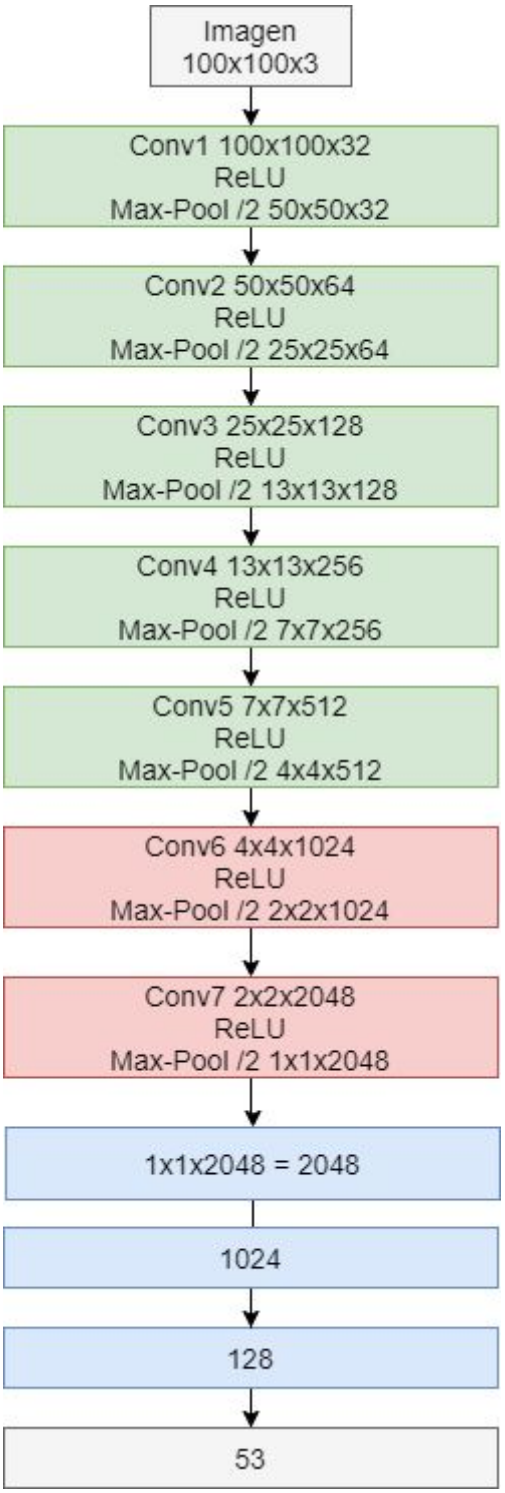




Debido a los bajos resultados, esta arquitectura queda descartada, pero sirve como base para la siguiente propuesta.

2.2 Propuesta 2: Añadiendo capas hasta llegar a un tamaño de 1x1

Probablemente la arquitectura anterior necesita de más capas convolucionales para lograr aprender correctamente durante en entrenamiento, entonces se decidió **agregar 2 capas convolucionales extra junto a sus respectivas capas max-pooling,** de modo que al final del extractor de características las dimensiones de la imagen queden de 1x1, con **2048 filtros** o características en ellas.



Para evitar que la precisión y el error se queden estancados durante el entrenamiento, se decidió modificar el **learning rate del optimizador a 0.0001.** El resto de la red, así como las funciones de error y accuracy se dejaron igual. **Se hizo una prueba de entrenamiento con 35 épocas, utilizando también batches de tamaño 53 y se logró llegar a una precisión de aproximadamente 30%.**

2.4 Red seleccionada

Debido a lo prometedor que se comportaba esta arquitectura a las 35 épocas, se decidió reiniciar el entrenamiento, pero esta vez con **80 épocas** para ver si la precisión aumentaba o si se llegaba a estancar en algún momento.

Se pudo notar que en las primeras 20 épocas, los resultados parecen no ser tan buenos, ya que a duras penas la red lograba superar el 5% de precisión y el error estaba estancado en aproximadamente un 4% en entrenamiento.

Pero una vez superadas las 30-40 épocas, la precisión empezaba a subir poco a poco hasta un 40-50% y el error a disminuir hasta un 1%. **Al llegar a las 80 épocas, siempre se obtienen precisiones por encima del 98% en entrenamiento**, mientras que el error ya se encontraba muy bajo, siendo esta la arquitectura seleccionada para cumplir con el objetivo de este proyecto. Más adelante se hablaran e interpretaran estos resultados.

Entonces, la arquitectura seleccionada queda de la siguiente forma:

- **7 capas convolucionales**
- Tamaño de kernel 3x3 con stride de 1
- Función de activación **ReLU** en cada capa convolucional
- Para cada capa convolucional hay una **capa max-pooling** con un stride de 2
- **3 capas directamente conectadas con ReLU**
- **1 capa de salida con 53 neuronas**, sin función de activación
- 46640 imagenes para entrenamiento, 11660 para pruebas
- Batch de tamaño 53
- **Categorical Cross Entropy** como función de pérdida
- **Optimizador Adam con learning rate de 0.0001**
- Entrenado durante **80 épocas**
- **Accuracy: categorical accuracy**

Alumno: Ramos Diaz Enrique

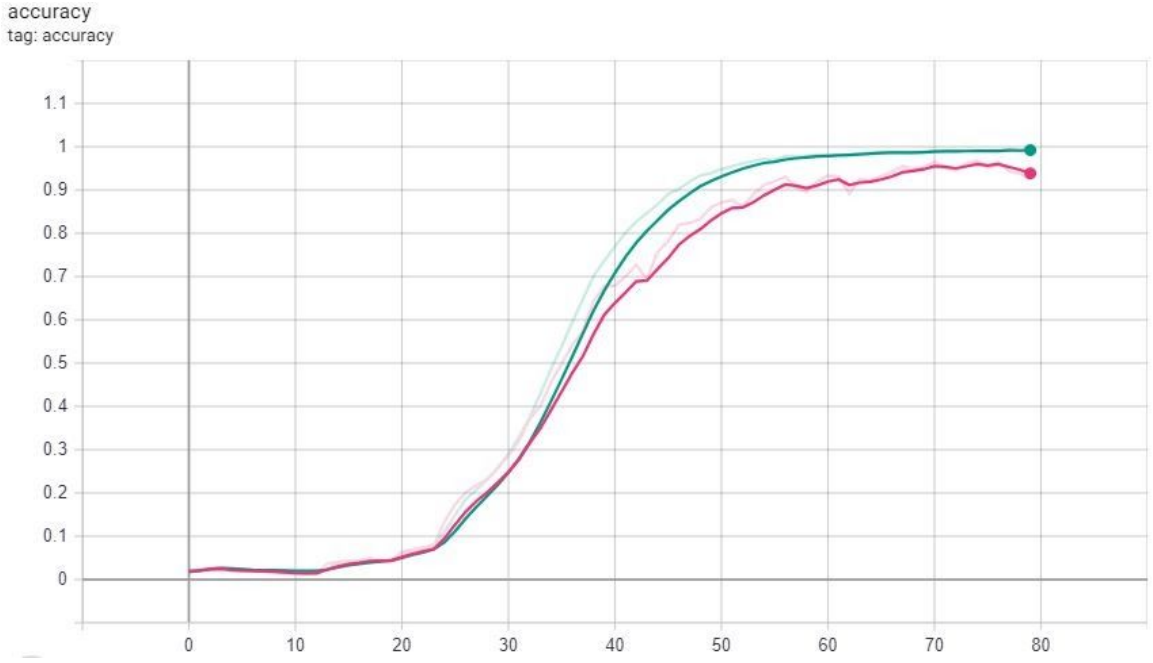
3. Validación de la red seleccionada

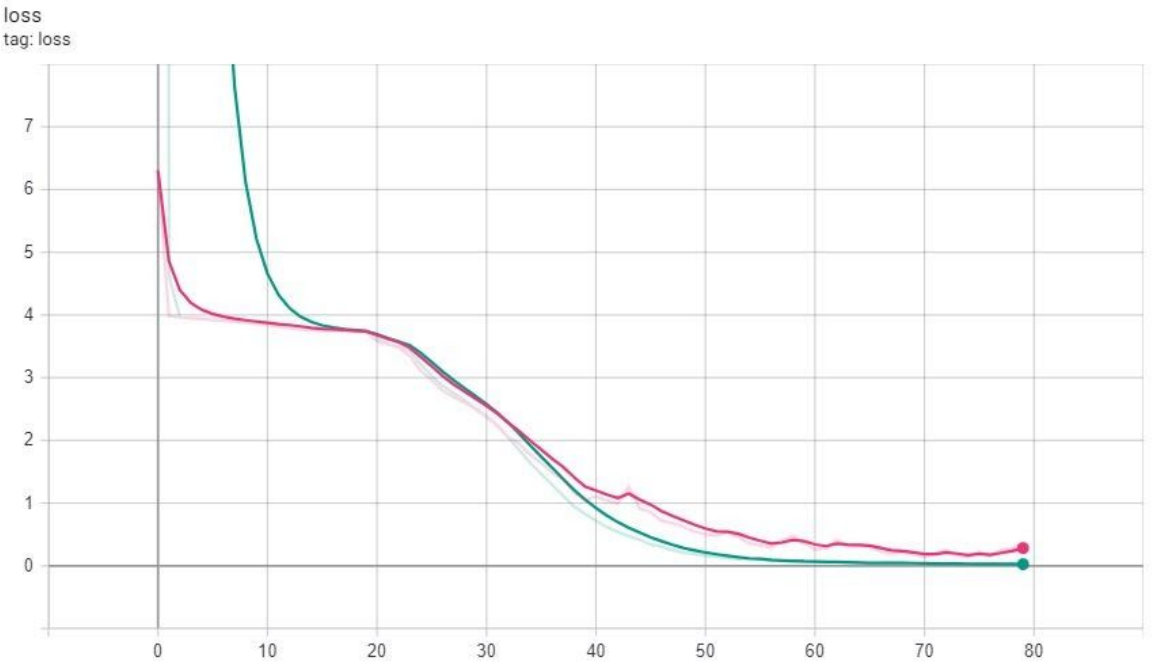
Tal vez la primera propuesta necesito de más épocas para mostrar su verdadero potencial, y también el learning rate influyó mucho en su bajo desempeño.

De todos modos, la seleccionada fue la propuesta 2, de 7 capas convolucionales.

A continuación se muestran las curvas de 2 distintos entrenamientos con 80 épocas:

Entrenamiento 1:

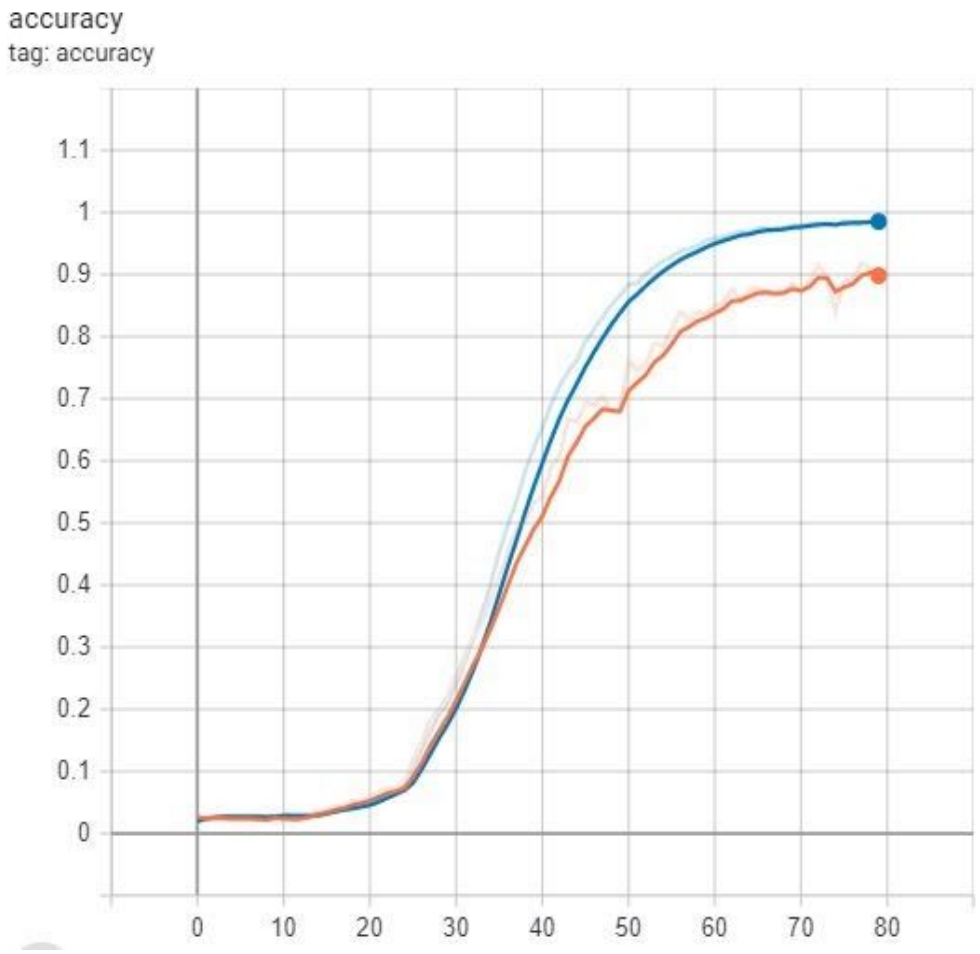


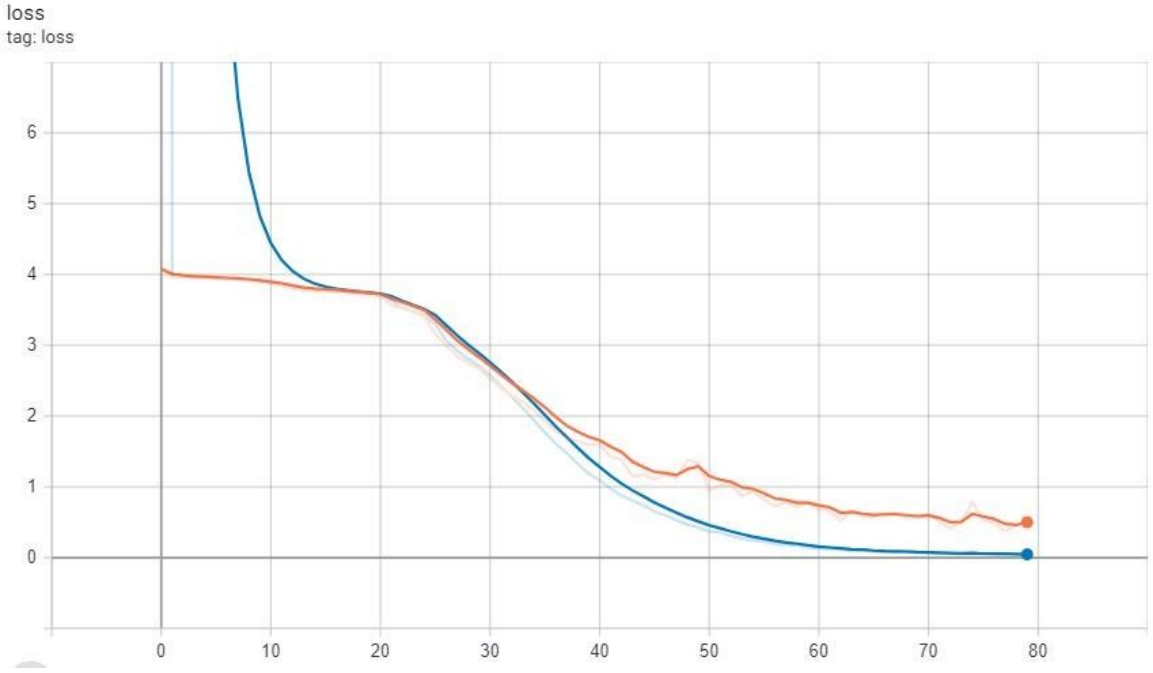


La curva verde son los resultados de los datos de entrenamiento, y la curva rosa son de los datos de prueba.

Entrenamiento 2:

Alumno: Ramos Diaz Enrique





La curva azul son los resultados de los datos de entrenamiento, y la curva naranja son de los datos de prueba.

Como se puede notar en las curvas de precisión, en el **entrenamiento 1** se logra llegar a un **99% de precisión en entrenamiento y 96% en pruebas.**

Por otro lado, en el **entrenamiento 2 se llega a un 98% en entrenamiento y a un 91% en pruebas.**

Esta diferencia ocurre debido a los **pesos generados aleatoriamente** en cada entrenamiento, por lo que se decide utilizar los del primer entrenamiento.

Sin embargo, estos resultados obtenidos pueden ser un poco engañosos, ya que se utilizó data augmentation, y las variaciones generadas de cada imágen no son muy distintas entre sí, pudiendo ser consideradas incluso como **imágenes repetidas o calcadas.**

La presencia de overfitting es fácilmente refutada ya que tanto el error de prueba como el de entrenamiento siempre fueron disminuyendo en cada generación, teniendo ambos un valor muy pequeño al final del entrenamiento.

Así, podemos concluir que la red logró aprender correctamente sin memorizar patrones, y además es apta para cumplir con el objetivo principal del proyecto planteado, por medio del siguiente módulo.

4. Módulo Extra: Búsqueda de la receta

Para satisfacer en su totalidad el objetivo de este sistema, no basta con tener una red neuronal con buenos resultados, ya que hay que darles un significado útil a estos para el usuario los pueda entender fácilmente.

Para esto, primero se debe de **convertir el vector de salida de la red de 53 posiciones a OneHot**; es decir, asignarle un 1 a la posición con el valor más alto, y 0 al resto de las posiciones (puede que el vector de salida de la red ya tenga este formato, pero hay que asegurarlo).

Previamente se debe tener **cargado un diccionario de Python que posea los 53 platillos**, cuya **clave** corresponde a un **identificador OneHot de 53 posiciones**, en la que la posición en donde se encuentre el 1 corresponde al platillo (según el orden presentado anteriormente), y para el **valor** sería la **receta de este platillo en formato JSON**.

Así, simplemente **se consulta utilizando el vector de salida de la red neuronal ya en OneHot como identificador en este diccionario de recetas**, para recuperar esta y ser mostrada al usuario. Esta receta incluye las cantidades y kilocalorías de cada ingrediente, normalizadas según el *Sistema Mexicano de Alimentos Equivalentes.*