Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Clasificación de imágenes mediante CNN

Informe de Tarea 5

Alumno: José Rubio
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: Juan Pablo Cáceres B.
Javier Smith D.
Hans Starke D.
José Villagrán E.

Fecha de realización: 17 de diciembre de 2020
Fecha de entrega: 17 de diciembre de 2020
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Códigos

1. Introducción.

La automatización, junto con el desarrollo de maquinas que realizan una gran cantidad de tareas inteligentes corresponde a uno de los desarrollo más importantes de la actualidad. Hoy en día se tienen una gran cantidad de dispositivos que realizan actividades que antes solo podían ser hechas por seres humanos, como predicciones, detecciones fenómenos, recuperación de información, **clasificaciones**, etc. Por lo que algunas tareas ya pasaron a segundo plano para que sean desarrolladas por máquinas, debido a que estos sistemas tienen un mejor rendimiento que un humano.

Uno de los modelos mas interesantes que el *machine learning* ha desarrollado corresponde a los **clasificadores**, la cual como su nombre lo indica permite clasificar una gran cantidad de datos en etiquetas (o *labels*) según las características con las que cuenten estas. El potencial que tiene esta herramienta es prácticamente ilimitado, se tienen clasificadores de plantas, animales, tipos de persona, deportes, juegos, música, etc. Además de la disposición de varias librerías que facilitan el trabajo al momento de programar uno de estos sistemas, donde algunas de estas librerías favorecen el desarrollo de arquitecturas sumamente complejas para el uso de *deep learning* para construir estos clasificadores.

Dentro de los clasificadores más complejos de desarrollar se encuentran las **Redes Neuronales Convolucionales o CNN** (Convolutional Neural Networks), estas redes permiten desarrollar sistemas inteligentes a partir de imágenes, lo que los convierte en los ojos de la Inteligencia Artificial, su misma arquitectura esta inspirada en la estructura biológica de los ojos humanos por lo que la cantidad de datos que procesan estos datos es mucho más grande que los sistemas configurados mediante *machine learning*. Dada esta característica se necesita una mayor cantidad de información para poder entrenar estas redes de forma correcta, ademas de un hardware más potente debido a la tremenda cantidad de parámetros a entrenar y datos necesarios.

Dado esto, en este informe se mostrará el desarrollo de 2 CNN, las cuales fueron programas utilizando la librería `Pytorch`, la cual permite programar de forma sencilla arquitecturas complejas que usan *deep learning*. Para el entrenamiento de estas se uso la base de datos **CIFAR-10**, este consiste en un dataset de 60000 imágenes de 32x32 pixeles, las cuales pertenecen a 10 clases distribuidas equitativamente. El tamaño de este dataset comprimido es de unos 166 MB, por lo que el entrenamiento utilizando esta base de datos no es viable si es procesado por una CPU debido a la enrome cantidad de cálculos necesarios. Debido a esto el sistema sera entrenado en la GPU, la cual permite desarrollar una gran cantidad de cálculos en paralelo, lo que reduce en gran medida el tiempo de entrenamiento.

Teniendo esto en mente, este archivo contendrá la siguiente información:

- Extracción de la base de datos **CIFAR-10** y su almacenamiento en la intefaz de Colab.
- Configuración de los objetos para procesar los batches de la base de datos.
- Métodos de programación de las arquitecturas solicitas, se programaron 2 CNN similares, sin embargo una de ellas es mas compleja debido a las capas convolucionales y lineales extras que se le añadieron.
- Entrenamiento de las redes en la GPU, donde se configuró un registro que mostró el rendimiento de la red mientras estuvo entrenando.
- Análisis de los sistemas programados viendo el comportamiento de la función de costos y los *accuracies* de las redes, ademas de sus matrices de confusión.

2. Desarrollo.

En esta sección se mostrarán los algoritmos utilizados para el desarrollo de los clasificadores.

2.1. Extracción y Almacenamiento de CIFAR-10.

El tamaño de esta base de datos es de 166 MB, por ende fue necesario extraer este dataset mediante el siguiente comando:

```
!wget -c https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

Código 1: Extraccíon y almacenamiento del dataset CIFAR-10.

Con la base descargada, entonces se cargaron los archivos al colab mediante la ejecución del siguiente comando:

```
!tar -xvzf cifar-10-python.tar.gz
```

Código 2: Carga de los batches de la base de datos CIFAR-10.

Donde al ejecutar esta base de datos se cargan a colab los archivos `batches.meta`, `data_batch_1`, `data_batch_2`, `data_batch_3`, `data_batch_4`, `data_batch_5`, `test_batch`, `readme.html`. Si bien se tienen todos los datos de este dataset, para efecttos de esta tarea solo usaremos 3 batches: `data_batch_1`, `data_batch_2` y `test_batch` para el desarrollo del conjunto de entrenamiento, validación y prueba respectivamente.

2.2. Acceso a los datasets de entrenamiento, validación y prueba.

Para acceder a los batches cargados se usó la siguiente función auxiliar:

```
def unpickle(file):
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

Código 3: Función unpickle para cargar los batches

Con el uso de esta función se programó la siguiente clase para poder configurar de forma correcta los datasets necesarios.

```
from torch.utils.data import Dataset

class CIFAR10Train(Dataset):
  def __init__(self, path, scale=True):
    # Constructor, debe leer el archivo data_batch_1 dentro de la carpeta
    # indicada (este archivo se usará para el set de entrenamiento)
    self.data_batch_1 = unpickle(path + '/data_batch_1')
    self.scale = scale

```

```
10
11  def ___len___(self):
12    # Debe retornar el número de imágenes en el dataset de entrenamiento
13    return len(self.data_batch_1[b'data'])
14
15  def ___getitem___(self, index):
16    # Debe retornar un par label, image
17    # Donde label es una etiqueta, e image es un arreglo de 3x32x32
18    # index es un número (o lista de números) que indica cuáles imágenes
19    # y labels se deben retornar
20    labels = []
21    images = []
22    try:
23      for i in index:
24        label = self.data_batch_1[b'labels'][i]
25        if self.scale:
26          image = 2*(np.array(self.data_batch_1[b'data'][i]))/255 - 1 # Al obtener los datos, tambien
      ↪ se aplica un escalamiento lineal.
27        else:
28          image = np.array(self.data_batch_1[b'data'][i])
29        image = image.reshape(3, 32, 32)
30        labels.append(label)
31        images.append(image)
32    except:
33      labels = self.data_batch_1[b'labels'][index]
34      if self.scale:
35        image = 2*(np.array(self.data_batch_1[b'data'][index]))/255 - 1 # Se aplica el mismo
      ↪ escalamiento.
36      else:
37        image = np.array(self.data_batch_1[b'data'][index])
38      images = image.reshape(3, 32, 32)
39
40    return labels, images
```

Código 4: Implementación de los dataset para pytorch

Este objeto carga el archivo `data_batch_1` donde permite se extrae el largo y se obtienen las etiquetas y datos de las imágenes. Estos datos vinieron en un arreglo unidimensional de 3072 valores, dado esto se hizo una conversión de estos arreglos para colocarlos en arreglos con dimensiones de 32x32 en 3 canales, es decir las dimensiones se cambiaron a 3x32x32 (línea 29 y 38).

Los datos de los arreglos inicialmente vinieron en un rango de 0 a 255, por ende se configuro un parámetro `scale`, el cual al activarlo escalaba las imágenes al rango [-1, 1]. Estas características fueron configuradas para que este objeto pudiese cargar los datos del conjunto de entrenamiento de forma correcta a pytorch, de la misma forma se programaron las clases `CIFAR10Val` y `CIFAR10Test` para cargar de la misma forma los datos del conjunto de validación y prueba respectivamente (estos bloques son sumamente similares, solo que cargan los archivos `data_batch_2` y `test_batch`, la implementación de estas clases están en los Anexos).

2.3. Arquitecturas de las Redes.

Para este trabajo se solicitaron las siguientes redes:

Arquitectura G (Grande)	Arquitectura P (Pequeña)
Convolución con 64 filtros de 3x3 + ReLU + BN	Convolución con 64 filtros de 3x3 + ReLU + BN
Convolución con 64 filtros de 3x3 + ReLU + BN	Max pooling
Max pooling	Convolución con 128 filtros de 3x3 + ReLU + BN
Convolución con 128 filtros de 3x3 + ReLU + BN	Max pooling
Convolución con 128 filtros de 3x3 + ReLU + BN	Convolución con 256 filtros de 3x3 + ReLU + BN
Max pooling	Max pooling
Convolución con 256 filtros de 3x3 + ReLU + BN	Convolución con 512 filtros de 3x3 + ReLU + BN
Convolución con 256 filtros de 3x3 + ReLU + BN	Max pooling
Max pooling	Fully connected con 128 unidades + ReLU + BN
Convolución con 512 filtros de 3x3 + ReLU + BN	Fully connected con 10 unidades
Convolución con 512 filtros de 3x3 + ReLU + BN	
Max pooling	
Fully connected con 128 unidades + ReLU + BN	
Fully connected con 256 unidades + ReLU + BN	
Fully connected con 512 unidades + ReLU + BN	
Fully connected con 1024 unidades + ReLU + BN	
Fully connected con 10 unidades	

Figura 1: Arquitecturas de redes solicitadas para la tarea.

Dado esto, entonces se programaron los siguientes objetos.

2.3.1. Red Grande.

Usando los módulos disponibles de `pytorch` se programó la arquitectura grande, el código utilizado se aprecia en el siguiente bloque:

```
class MyNetBig(nn.Module):
  def __init__(self, nclasses):
    super(MyNetBig, self).__init__()

    # Inicializadores de las capas necesarias para la arquitectura grande.

    self.nclasses = nclasses
    self.conv1 = nn.Conv2d(3, 64, (3, 3), stride=1, padding=1)
    self.conv2 = nn.Conv2d(64, 64, (3, 3), stride=1, padding=1)
    self.conv3 = nn.Conv2d(64, 128, (3, 3), stride=1, padding=1)
```

```
    self.conv4 = nn.Conv2d(128, 128, (3, 3), stride=1, padding=1)
    self.conv5 = nn.Conv2d(128, 256, (3, 3), stride=1, padding=1)
    self.conv6 = nn.Conv2d(256, 256, (3, 3), stride=1, padding=1)
    self.conv7 = nn.Conv2d(256, 512, (3, 3), stride=1, padding=1)
    self.conv8 = nn.Conv2d(512, 512, (3, 3), stride=1, padding=1)
    self.MaxPool = nn.MaxPool2d((3, 3), stride=2, padding=1)
    self.BN1 = nn.BatchNorm2d(64)
    self.BN2 = nn.BatchNorm2d(128)
    self.BN3 = nn.BatchNorm2d(256)
    self.BN4 = nn.BatchNorm2d(512)
    self.BN1d1 = nn.BatchNorm1d(128)
    self.BN1d2 = nn.BatchNorm1d(256)
    self.BN1d3 = nn.BatchNorm1d(512)
    self.BN1d4 = nn.BatchNorm1d(1024)
    self.flatten = nn.Flatten(1, 3)
    self.fc_1 = nn.Linear(2048, 128)
    self.fc_2 = nn.Linear(128, 256)
    self.fc_3 = nn.Linear(256, 512)
    self.fc_4 = nn.Linear(512, 1024)
    self.fc_out = nn.Linear(1024, nclasses)

  # Pasada hacia adelante.

  def forward(self, x):
    x = self.BN1(F.relu(self.conv1(x)))
    x = self.BN1(F.relu(self.conv2(x)))
    x = self.MaxPool(x)
    x = self.BN2(F.relu(self.conv3(x)))
    x = self.BN2(F.relu(self.conv4(x)))
    x = self.MaxPool(x)
    x = self.BN3(F.relu(self.conv5(x)))
    x = self.BN3(F.relu(self.conv6(x)))
    x = self.MaxPool(x)
    x = self.BN4(F.relu(self.conv7(x)))
    x = self.BN4(F.relu(self.conv8(x)))
    x = self.MaxPool(x)
    x = self.flatten(x)
    x = self.BN1d1(F.relu(self.fc_1(x)))
    x = self.BN1d2(F.relu(self.fc_2(x)))
    x = self.BN1d3(F.relu(self.fc_3(x)))
    x = self.BN1d4(F.relu(self.fc_4(x)))
    x_out = self.fc_out(x)
    return x_out
```

Código 5: Arquitectura en pytorch para la CNN grande.

Esta estructura recibe tensores de 4 dimensiones, y para que el *forward* no tenga problemas de dimensionalidad las imágenes recibidas deben tener dimensiones 32x32. Cada convolución y capa Lineal tiene función de activación RELU y un Batch Normalization, excepto la última capa la cual no posee una función de salida, esto se configuró de esta forma debido a que las funciones de costos cuentan con las funciones de salida para esta capa.

2.3.2. Red Pequeña.

De forma análoga, la implementación de la red pequeña siguió la siguiente estructura en pytorch.

```
class MyNetSmall(nn.Module):
  def ___init___(self, nclasses):
    super(MyNetSmall, self).___init___()

    # Inicializadores de las capas necesarias para la arquitectura grande.

    self.nclasses = nclasses
    self.conv1 = nn.Conv2d(3, 64, (3, 3), stride=1, padding=1)
    self.conv3 = nn.Conv2d(64, 128, (3, 3), stride=1, padding=1)
    self.conv5 = nn.Conv2d(128, 256, (3, 3), stride=1, padding=1)
    self.conv7 = nn.Conv2d(256, 512, (3, 3), stride=1, padding=1)
    self.MaxPool = nn.MaxPool2d((3, 3), stride=2, padding=1)
    self.BN1 = nn.BatchNorm2d(64)
    self.BN2 = nn.BatchNorm2d(128)
    self.BN3 = nn.BatchNorm2d(256)
    self.BN4 = nn.BatchNorm2d(512)
    self.BN1d1 = nn.BatchNorm1d(128)
    self.flatten = nn.Flatten(1, 3)
    self.fc_1 = nn.Linear(2048, 128)
    self.fc_out = nn.Linear(128, nclasses)

  # Pasada hacia adelante.

  def forward(self, x):
    x = self.BN1(F.relu(self.conv1(x)))
    x = self.MaxPool(x)
    x = self.BN2(F.relu(self.conv3(x)))
    x = self.MaxPool(x)
    x = self.BN3(F.relu(self.conv5(x)))
    x = self.MaxPool(x)
    x = self.BN4(F.relu(self.conv7(x)))
    x = self.MaxPool(x)
    x = self.flatten(x)
    x = self.BN1d1(F.relu(self.fc_1(x)))
    x_out = self.fc_out(x)
    return x_out
```

Código 6: Arquitectura en pytorch para la CNN pequeña.

Como su nombre lo indica, esta arquitectura es más pequeña que la anterior, reduciendo las capas convolucionales y las capas lineales de la clasificación. Al igual que la arquitectura anterior, este sistema recibe imágenes de 32x32 en 3 canales, además tambien se aplican RELU's y Batch Normalization a cada capa convolucional y oculta, excepto las capas de Pooling ya que la única finalidad de estas capas es reducir a la mitad el tamaño de las imágenes procesadas. También se conserva la característica de no aplicarle una función de salida a la capa de salida, este trabajo es realizado por la función de pérdidas al momento de realizar el entrenamiento.

2.4. Entrenamiento de las Redes.

2.4.1. Configuración de Datasets.

Con las arquitecturas programadas, entonces lo siguiente fue configurar el entrenamiento de las redes y su validación. Para ello en primer se configuraron los datasets usando el objeto mostrado en el punto 2.2 de la siguiente forma:

```
from torch.utils.data import DataLoader, random_split

# Para poder reportar
import sys

dataset_train = CIFAR10Train('cifar-10-batches-py', scale=False)
dataset_val = CIFAR10Val('cifar-10-batches-py', scale=False)
dataset_test = CIFAR10Test('cifar-10-batches-py',scale=False)

batch_size = 32

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True, num_workers=4,
    ↪ pin_memory=True)
val_loader = DataLoader(dataset_val, batch_size=batch_size, shuffle=True, num_workers=4,
    ↪ pin_memory=True)
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=True, num_workers=4,
    ↪ pin_memory=True)

total_train = train_loader.dataset.___len___()
total_val = val_loader.dataset.___len___()
```

Código 7: Configuración de los datasets de entrenamiento, validación y prueba

El paso más importante de este código consiste en el uso de la función `DataLoader` del módulo `torch.utils.data` la cual permite extraer y convertir todas las imágenes de los datasets seleccionados en los tensores necesarios para el entrenamiento, validación y prueba.

2.4.2. Entrenamiento.

Con los datos configurados correctamente, entonces se programó el entrenamiento utilizando el siguiente código:

```
# Selccionando y configurando la red

net = MyNetBig(10)
net.cuda()

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
epochs = 20

train_acc, val_acc, train_loss, val_loss = [], [], [], []
train_time, valid_time = [], []
```

```
12
13 for epoch in range(epochs):
14   t_i = time.time()
15   # El entrenamiento de los parametros se realizaran mediante .train()
16   net.train()
17
18   # Inicializando el valor para las métricas.
19   running_loss, running_acc = 0.0, 0.0
20
21   for i, data in enumerate(train_loader, 0):
22     labels = data[0].cuda()
23     inputs = data[1].float().cuda()
24     optimizer.zero_grad()
25     outputs = net(inputs)
26     loss = criterion(outputs, labels)
27     loss.backward()
28     optimizer.step()
29
30     items = (i+1) * batch_size
31     running_loss += loss.item()
32
33     Y_pred = F.softmax(outputs, dim=1)
34     max_prob, max_idx = torch.max(Y_pred, dim=1)
35     running_acc += torch.sum(max_idx == labels).item()
36     info = f'\rEpoch:{epoch+1}({items}/{total_train}), '
37     info += f'Loss:{running_loss/(i+1):02.5f}, '
38     info += f'Train Acc:{running_acc/items*100:02.1f} %'
39     sys.stdout.write(info)
40
41   train_acc.append(running_acc)
42   train_loss.append(running_loss)
43   t_f = time.time()
44   epoch_train_time = round(t_f - t_i, 2)
45   # net.eval()
46   tv_i = time.time()
47   with torch.no_grad():
48     running_acc = 0.0
49     preds = np.zeros(10)
50     targets = np.array([])
51     for i, data in enumerate(val_loader, 0):
52       labels = data[0].cuda()
53       inputs = data[1].float().cuda()
54       Y_pred = net(inputs)
55       preds = np.vstack([preds, Y_pred.cpu().numpy()], )
56       targets = np.append(targets, labels.cpu().numpy())
57       Y_pred = F.softmax(Y_pred, dim=1)
58       max_prob, max_idx = torch.max(Y_pred, dim=1)
59       running_acc += torch.sum(max_idx == labels).item()
60     info = f', Val Acc:{running_acc/total_val*100:02.2f} %.\n'
61     sys.stdout.write(info)
62     preds = torch.tensor(np.delete(preds, 0, 0))
```

```
    targets = torch.tensor(targets)
    val_loss_epoch = criterion(preds, targets.type(torch.LongTensor))
    val_loss.append(val_loss_epoch)
    val_acc.append(running_acc)
    tv_f = time.time()
    epoch_valid_time = round(tv_f - tv_i, 2)
  train_time.append(epoch_train_time)
  valid_time.append(epoch_valid_time)
  print('Epoch train time: ', epoch_train_time, '[s], Valid time: ', round(epoch_valid_time, 2), '[s]')

train_total = np.sum(np.array(train_time))
valid_total = np.sum(np.array(valid_time))
print('Train Total time: ', train_total, '[s], Valid Total time: ', valid_total, '[s]')
```

Código 8: Entrenamiento de la CNN grande mediante los módulos y funciones de pytorch.

Como se contaba con una gran cantidad de imágenes y parámetros de las redes, entonces lo que se hizo en primer lugar fue llevar la red a la memoria de la GPU mediante el comando `.cuda()`. Tras esto se seleccionaron el criterio y optimizador para realizar el proceso, donde se usó **Entropía Cruzada y ADAM** con tasa de aprendizaje igual a 0.001 respectivamente. Como se deseó obtener el rendimiento de las redes a medida que estas se entrenaban, entonces se crearon 3 listas vacías, donde estas fuesen almacenando los valores de *accuracy y loss* para cada época. Se escogió 20 épocas para el entrenamiento

Durante el entrenamiento, se aseguraron que los parámetros de la red se entrenarán mediante el comando `.train()` y se inicializaron los valores del coste y el *accuracy* para cada época, tras esto se hizo el procesamiento de los batches del sistema, donde cada tensor generado se llevó a la GPU utilizando `.cuda()` nuevamente, tras esto se realizó una pasada hacia adelante y luego se evaluó la pérdida usando las predicciones producto de la pasada hacia adelante de la red. Con esto se realizó la pasada hacia atrás (*backpropagation*) optimizando los parámetros.

Luego como se quiso analizar el rendimiento de la red mientras se entrenaba, entonces se evaluó el batch para obtener las predicciones y se compararon con las etiquetas reales, esto entregó los costes y los *accuracies* del conjunto del entrenamiento mientras el entrenamiento se realizaba, finalizando este proceso se agregó el valor final de los costos y *accuracy* a los arreglos iniciales y luego se evaluaron todos los batches con la red entrenada a la época actual para obtener el *accuracy* del conjunto de validación, cuyo valor también fue agregado a los arreglos iniciales.

El entrenamiento mostrado en este bloque se realizó para la red Grande, sin embargo este mismo sistema fue utilizado para entrenar la red pequeña (la implementación de esta se encuentra en los anexos).

Arquitectura	Cantidad de parámetros
Red Grande	5.655.498
Red Pequeña	1.816.714

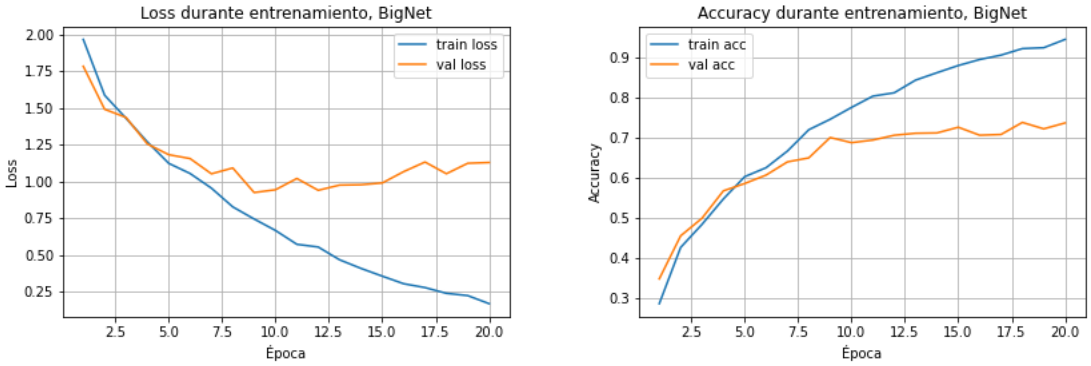
Tabla 1: Cantidad de parámetros de las arquitecturas programadas.

3. Resultados y Análisis.

Aplicando los bloques anteriores se obtuvieron los siguientes resultados:

3.1. Red Grande.

Las curvas de costos y accuracy obtenidas para esta red fueron las siguientes:



(a) (b)

Figura 2: Curvas de costo y accuracy obtenidas durante el entrenamiento de la red.

La matriz de confusión obtenida para este sistema fue el siguiente:

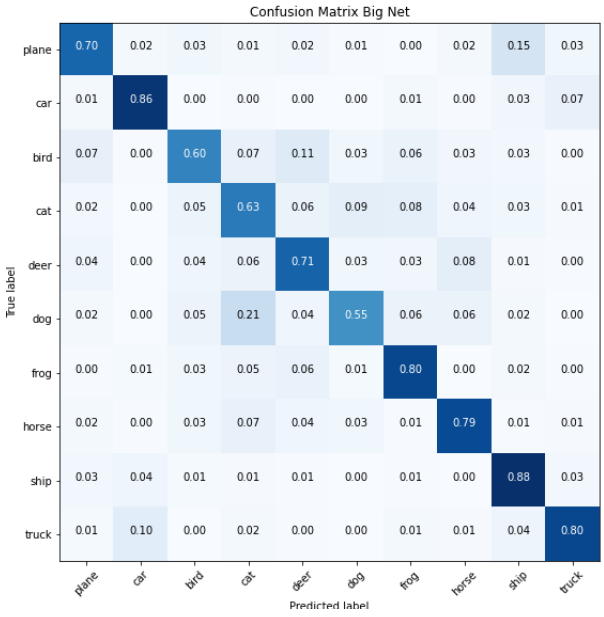
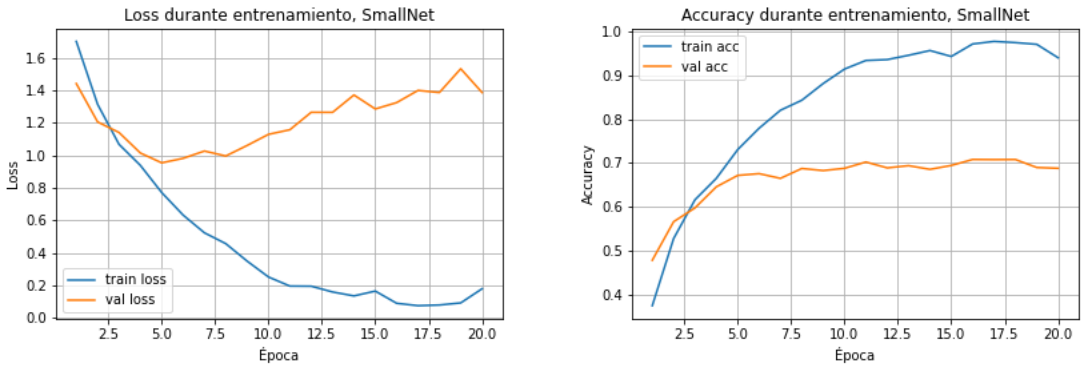


Figura 3: Matriz de confusión normalizada para la red Grande. Esta información se obtuvo al evaluar el conjunto de Prueba.

3.2. Red Pequeña.

Las curvas de costos y accuracy obtenidas par la red pequeña fueron las siguientes:



(a) (b)

Figura 4: Curvas de costo y accuracy obtenidas durante el entrenamiento de la red pequeña.

El comportamiento de esta red puede mostrarse también en la siguiente matriz de confusión:

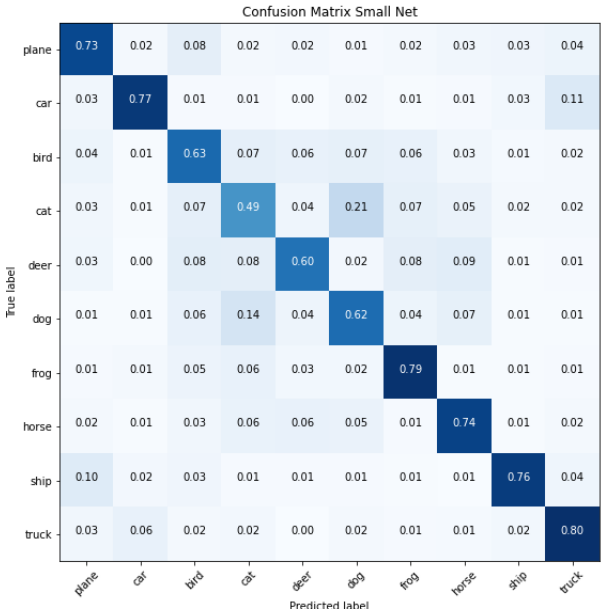


Figura 5: Matriz de confusión normalizada para la red pequeña.

Al igual que en el caso anterior, esta matriz fue generada al evaluar los datos en el conjunto de prueba. Cabe destacar que debido a la menor cantidad de parámetros de esta red, esta se pudo entrenar más rápido, necesitando 4.5 segundos por época en comparación a los 7 segundos por época de la red grande.

3.3. Análisis

Los valores mostrados en la tabla 2 indican que el sistema que tuvo un mejor desempeño fue la arquitectura grande, a pesar de que necesitó más tiempo para poder completar su entrenamiento, esto tambien es apoyado debido a la mayor estabilidad que tuvo su *loss* de validación al momento de estar entrenando, sin llegar a aumentar.

A pesar de esto la red pequeña pudo realizar la misma tarea sin la necesidad de capas convolucionales extras, alcanzando un rendimiento similar a la red grande. Sin embargo esta reducción de la arquitectura fue a coste a una reducción en el accuracy de este sistema, ya que en las pruebas realizadas se apreció que la Red Grande **siempre** tuvo un mejor rendimiento que la Red Pequeña.

Arquitectura	Test Accuracy	Tiempo de Entrenamiento
Red Grande	72.9 %	116.72 [s]
Red Pequeña	69.23 %	60.31 [s]

Tabla 2: Accuracy del conjunto de prueba para las arquitecturas diseñadas. Épocas: 20.

Si bien el entrenamiento de la red pequeña tomó casi la mitad del tiempo que la red grande, la curva de costos de esta red indica una alza de estos valores a medida que avanzaban las épocas, sin embargo el accuracy obtenidos obtenidos para el conjunto de validación muestran que no se tiene un *overfitting* al final de las épocas, por lo que el comportamiento de los costos de validación para ambas redes no fue implementado de forma precisa, aunque conserva la propiedad de ser mayor a los costos de entrenamiento para la mayor cantidad de épocas.

Los sistemas configurados tuvieron un rendimiento máximo de 0.71 en el conjunto de validación aproximadamente, donde las figuras indican que este límite pudo ser alcanzado más rápido por la red pequeña (figuras 2b y 4b). Esto indicaría que la esta red fue mas eficiente, sin embargo la red grande pudo lograr un accuracy un poco más alto a medida que avanzaba el entrenamiento, a pesar de que tardó un poco mas en converger a este valor. La razón de esta tardanza es simple: una mayor cantidad de parámetros entrenables (tabla 1), donde a cambio de esto se pudo lograr un valor un poco más alto en la clasificación de la base de datos.

Los sistemas fueron implementados y entrenados correctamente, dando como resultado las matrices de confusión representadas en las figuras 3 y 5 para la red grande y pequeña respectivamente. En ambas figuras se aprecia que pudo acertar en la mayor parte de imágenes del conjunto de prueba, donde la red grande tuvo facilidades para catalogar las imágenes que perteneciesen a autos, mientras que la red pequeña pudo reconocer mejor los camiones. Se tienen que ambas redes pudieron reconocer mejor los vehículos que el resto de imágenes, mientras que los animales presentaron dificultades para poder clasificarse correctamente, en parte es debido a que dentro del dataset se tienen imágenes muy similares que están dentro de las clases de animales (caballos, ciervos, perros, gatos). Mientras que algunos vehículos pueden ser mejor diferidos debido a que las características de estos vehículos son mas diferentes entre ellos.

4. Conclusión.

La clasificación de imágenes es uno de los mas grandes avances en visión computacional que se han logrado, por lo que el conocimiento adquirido sobre la implementación de estas redes correspondió a un conocimiento adquirido sumamente valioso. Las redes fueron implementadas de forma exitosa usando la librería `pytorch`, si bien se tenía conocimiento de esta librería antes de desarrollar esta actividad, el uso de algunas funcionalidades como `with torch.no_grad()`: correspondió a un comando muy útil al momento de realizar el entrenamiento de las redes solicitadas.

El uso de la GPU fue fundamental al momento de desarrollar este trabajo, debido a la gran cantidad de datos y parámetros entrenables, donde lo más destacable de este proceso fue el traslado de la memoria de los tensores, ya que los arreglos numpy para obtener los costos de validación no pueden ser obtenidos desde la memoria GPU, por lo que se pasaron a la CPU. Este paso fue implementado correctamente, sin embargo puede que en este mismo proceso se encuentre el error, ya que los costes de validación durante el entrenamiento presentaron una alza en comparación a los costes iniciales. Este resultado fue incorrecto ya que el *overfitting* no existió, esto se pudo comprobar en los accuracies del conjunto de validación de ambas redes. Esto indica que es necesario comprender mejor las funciones y módulos que ofrece `pytorch`, siendo este un conocimiento fundamental por adquirir debido a la enorme gama de posibilidades que ofrece esta librería para desarrollar sistemas de deep learning.

Las redes implementadas alcanzaron un buen accuracy a pesar de entrenarse solo con un batch del dataset, por lo que las arquitecturas fueron diseñadas para procesar de forma correcta este dataset. Se realizó el experimento con todos los datos, donde el sistema alcanzó un accuracy del 90 %, este puede corresponde a una red buena y muy potente, sin embargo esta lejos de los sistemas mas eficaces para clasificar esta base de datos como la DenseNet, ResNet, AlexNet o GoogLeNet. Aunque estos sistemas no son mucho mas complejos de los implementados en esta tarea, por lo que con el conocimiento adquirido en este trabajo se pueden implementar estas redes sin mucho esfuerzo para obtener un rendimiento excepcional en esta base de datos. A pesar de esto, en trabajos futuros de este mismo rubro, se espera implementar redes con arquitecturas más simples que puedan obtener una gran precisión, ademas de corregir algunas implementaciones para obtener los valores de los costes del conjunto de validación correctos, para poder entregar un análisis más preciso.

Referencias

[1] *Illustrated: 10 CNN Architectures* Raimi Karim.

[2] *Deep Residual Networks*, Facebook AI Research, Kaiming He, July 16.

[3] *A guide to convolution aritmethic for deep learning* Vicent Dumolin, Francesco Visin, Université de Montreal, Politécnico de Milano, March 24 2016.

[4] *Deep Learning*, PPT del curso *EL7008-1 Procesamiento Avanzado de Imágenes*, Departamento de Ingeniería Eléctrica, Universidad de Chile. Semestre Primavera 2020.

[5] *Going deeper with convolutions*, Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Google AI Research. September 17 2014.

5. Anexos

5.1. Implementación de datasets.

5.1.1. Conjunto de validación.

```
class CIFAR10Val(Dataset):
  def __init__(self, path, scale=True):
    # Constructor, debe leer el archivo data_batch_1 dentro de la carpeta
    # indicada (este archivo se usará para el set de entrenamiento)
    self.data_batch_2 = unpickle(path + '/data_batch_2')
    self.scale = scale


  def __len__(self):
    # Debe retornar el número de imágenes en el dataset de entrenamiento
    return len(self.data_batch_2[b'data'])

  def __getitem__(self, index):
    # Debe retornar un par label, image
    # Donde label es una etiqueta, e image es un arreglo de 3x32x32
    # index es un número (o lista de números) que indica cuáles imágenes
    # y labels se deben retornar
    labels = []
    images = []
    try:
      for i in index:
        label = self.data_batch_2[b'labels'][i]
        if self.scale:
          image = 2*(np.array(self.data_batch_2[b'data'][i]))/255 - 1 # Al obtener los datos, tambien
      ↪ se aplica un escalamiento lineal.
        else:
          image = np.array(self.data_batch_2[b'data'][i])
        image = image.reshape(3, 32, 32)
        labels.append(label)
        images.append(image)
    except:
      labels = self.data_batch_2[b'labels'][index]
      if self.scale:
        image = 2*(np.array(self.data_batch_2[b'data'][index]))/255 - 1 # Se aplica el mismo
      ↪ escalamiento.
      else:
        image =  np.array(self.data_batch_2[b'data'][index])
      images = image.reshape(3, 32, 32)

    return labels, images
```

Código 9: Clase para cargar los datos del conjunto de validación.

5.1.2. Conjunto de prueba.

```
class CIFAR10Test(Dataset):
  def __init__(self, path, scale=True):
    # Constructor, debe leer el archivo data_batch_1 dentro de la carpeta
    # indicada (este archivo se usará para el set de entrenamiento)
    self.data_batch_t = unpickle(path + '/test_batch')
    self.scale = scale

  def __len__(self):
    # Debe retornar el número de imágenes en el dataset de entrenamiento
    return len(self.data_batch_t[b'data'])

  def __getitem__(self, index):
    # Debe retornar un par label, image
    # Donde label es una etiqueta, e image es un arreglo de 3x32x32
    # index es un número (o lista de números) que indica cuáles imágenes
    # y labels se deben retornar
    labels = []
    images = []
    try:
      for i in index:
        label = self.data_batch_t[b'labels'][i]
        if self.scale:
          image = 2*(np.array(self.data_batch_t[b'data'][i]))/255 - 1 # Al obtener los datos, tambien
      ↪ se aplica un escalamiento lineal.
        else:
          image = np.array(self.data_batch_t[b'data'][i])
        image = image.reshape(3, 32, 32)
        labels.append(label)
        images.append(image)
    except:
      labels = self.data_batch_t[b'labels'][index]
      if self.scale:
        image = 2*(np.array(self.data_batch_t[b'data'][index]))/255 - 1 # Se aplica el mismo
      ↪ escalamiento.
      else:
        image =  np.array(self.data_batch_t[b'data'][index])
      images = image.reshape(3, 32, 32)

    return labels, images
```

Código 10: Clase para carfar los datos del conjunto de prueba.

5.2. Entrenamiento de red pequeña.

```
# Selccionando y configurando la red

net_s = MyNetSmall(10)
net_s.cuda()

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net_s.parameters(), lr=1e-3)
epochs = 20

train_acc_small, val_acc_small, train_loss_small, val_loss_small = [], [], [], []
train_time_small, valid_time_small = [], []

for epoch in range(epochs):
  t_i = time.time()
  # El entrenamiento de los parametros se realizaran mediante .train()
  net_s.train()

  # Inicializando el valor para las métricas.
  running_loss, running_acc = 0.0, 0.0

  for i, data in enumerate(train_loader, 0):
    labels = data[0].cuda()
    inputs = data[1].float().cuda()
    optimizer.zero_grad()
    outputs = net_s(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    items = (i+1) * batch_size
    running_loss += loss.item()

    Y_pred = F.softmax(outputs, dim=1)
    max_prob, max_idx = torch.max(Y_pred, dim=1)
    running_acc += torch.sum(max_idx == labels).item()
    info = f'\rEpoch:{epoch+1}({items}/{total_train}), '
    info += f'Loss:{running_loss/(i+1):02.5f}, '
    info += f'Train Acc:{running_acc/items*100:02.1f}%'
    sys.stdout.write(info)

  train_acc_small.append(running_acc)
  train_loss_small.append(running_loss)
  t_f = time.time()
  epoch_train_time = round(t_f - t_i, 2)
  # net.eval()
  tv_i = time.time()
  with torch.no_grad():
    running_acc = 0.0
```

```
    preds = np.zeros(10)
    targets = np.array([])
    for i, data in enumerate(val_loader, 0):
      labels = data[0].cuda()
      inputs = data[1].float().cuda()
      Y_pred = net_s(inputs)
      preds = np.vstack([preds, Y_pred.cpu().numpy()])
      targets = np.append(targets, labels.cpu().numpy())
      Y_pred = F.softmax(Y_pred, dim=1)
      max_prob, max_idx = torch.max(Y_pred, dim=1)
      running_acc += torch.sum(max_idx == labels).item()
    info = f', Val Acc:{running_acc/total_val*100:02.2f} %.\n'
    sys.stdout.write(info)
    preds = torch.tensor(np.delete(preds, 0, 0))
    targets = torch.tensor(targets)
    val_loss_epoch = criterion(preds, targets.type(torch.LongTensor))
    val_loss_small.append(val_loss_epoch)
    val_acc_small.append(running_acc)
    tv_f = time.time()
    epoch_valid_time = round(tv_f - tv_i, 2)
  train_time_small.append(epoch_train_time)
  valid_time_small.append(epoch_valid_time)
  print('Epoch train time: ', epoch_train_time, '[s], Valid time: ', epoch_valid_time, '[s]')

train_total = np.round(np.sum(np.array(train_time_small)), 2)
valid_total = np.round(np.sum(np.array(valid_time_small)), 2)
print('Train Total time: ', train_total, '[s], Valid Total time: ', valid_total, '[s]')
```

Código 11: Bloque para entrenar la red pequeña.

5.3. Evaluación de redes.

5.3.1. Gráficos de métricas Red Grande.

```
Epochs = range(1, epochs+1)

plt.plot(Epochs, np.array(train_loss)/313, label='train loss')
plt.plot(Epochs, val_loss, label='val loss')
plt.title('Loss durante entrenamiento, BigNet')
plt.xlabel('Época')
plt.ylabel('Loss')
plt.grid(True)
# plt.savefig('Big_loss.png')
plt.show()

plt.plot(Epochs, np.array(train_acc)/10000, label='train acc')
plt.plot(Epochs, np.array(val_acc)/10000, label='val acc')
plt.title('Accuracy durante entrenamiento, BigNet')
plt.xlabel('Época')
plt.ylabel('Accuracy')
```

```
plt.grid(True)
plt.legend()
# plt.savefig('Big_accuracy.png')
plt.show()
```

Código 12: Loss y Accuracies para entrenamiento y validación Red Grande

5.3.2. Gráficos de métricas Red Pequeña

```
plt.plot(Epochs, np.array(train_loss_small)/313, label='train loss')
plt.plot(Epochs, val_loss_small, label='val loss')
plt.title('Loss durante entrenamiento, SmallNet')
plt.xlabel('Época')
plt.ylabel('Loss')
plt.grid(True)
# plt.savefig('Small_loss.png')
plt.show()

plt.plot(Epochs, np.array(train_acc_small)/10000, label='train acc')
plt.plot(Epochs, np.array(val_acc_small)/10000, label='val acc')
plt.title('Accuracy durante entrenamiento, SmallNet')
plt.xlabel('Época')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()
# plt.savefig('Small_accuracy.png')
plt.show()
```

Código 13: Loss y Accuracies para entrenamiento y validación Red Pequeña

5.4. Matrices de confusión.

5.4.1. Función generadora de matrices de confusión.

```
def plot_confusion_matrix(cm, classes, normalize=True, title='Confusion matrix', cmap=plt.cm.
    ↪ Blues, fig_size=(7,7)):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    # print(cm)
    plt.figure(figsize=fig_size)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    # plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
```

```
16
17     fmt = '.2f' if normalize else 'd'
18     thresh = cm.max() / 2.
19     for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
20         plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white" if cm[i, j] >
       ↪ thresh else "black")
21
22     plt.tight_layout()
23     plt.ylabel('True label')
24     plt.xlabel('Predicted label')
25
26
27
28 classes = ('plane', 'car', 'bird', 'cat',
29           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Código 14: Función auxiliar que plotea una matriz de confusión.

5.4.2. Red Grande.

```
1  net.cuda()
2
3  preds_big, targets = np.array([]), np.array([])
4
5  with torch.no_grad():
6    running_acc = 0.0
7    for i, data in enumerate(val_loader, 0):
8      labels = data[0].cuda()
9      inputs = data[1].float().cuda()
10     Y_pred = net(inputs)
11     Y_pred = F.softmax(Y_pred, dim=1)
12     max_prob, max_idx = torch.max(Y_pred, dim=1)
13     preds_big = np.append(preds_big, max_idx.cpu().numpy())
14     targets = np.append(targets, labels.cpu().numpy())
15     running_acc += torch.sum(max_idx == labels).item()
16   info = f'Test Accuracy: {running_acc/total_val*100:02.2f} %.\n'
17   sys.stdout.write(info)
18
19 cm_big = confusion_matrix(targets, preds_big)
20
21 plot_confusion_matrix(cm_big, classes, title='Confusion Matrix Big Net' ,fig_size=(8,8))
22 # plt.savefig('CM_BigNet.png')
```

Código 15: Matriz de confusión red grande.

5.4.3. Red Pequeña

```
1  net_s.cuda()
2
3  preds_small, targets = np.array([]), np.array([])
4
```

```
with torch.no_grad():
  running_acc = 0.0
  for i, data in enumerate(val_loader, 0):
    labels = data[0].cuda()
    inputs = data[1].float().cuda()
    Y_pred = net_s(inputs)
    Y_pred = F.softmax(Y_pred, dim=1)
    max_prob, max_idx = torch.max(Y_pred, dim=1)
    preds_small = np.append(preds_small, max_idx.cpu().numpy())
    targets = np.append(targets, labels.cpu().numpy())
    # print(max_idx.size())
    running_acc += torch.sum(max_idx == labels).item()
  info = f'Test Accuracy: {running_acc/total_val*100:02.2f} %.\n'
  sys.stdout.write(info)

cm_small = confusion_matrix(targets, preds_small)
plot_confusion_matrix(cm_small, classes, title='Confusion Matrix Small Net', fig_size=(8,8))
plt.savefig('CM_SmallNet.png')
```

Código 16: Matriz de confusión red pequeña.