Trabajo Práctico 3 - Clasificación CIFAR10

Tecnología Digital VI: Inteligencia Artificial

Federico Giorgi Gastón Loza Montaña Tomás Curzio

04/11/23

Configuración

Para este inciso del trabajo, utilizamos el código otorgado por la cátedra. Creamos nuestras cuentas de W&B así como un equipo, con el cual creamos un proyecto, para que todos los integrantes podamos acceder a los distintos gráficos, la metadata y la configuración de nuestros experimentos. Agregamos al código otorgado la siguiente linea para guardar la loss y accuracy tanto en `training` como en `validation`, luego de cada epoch, en W&B:

```
wandb.log({ "train_accuracy": train_accuracy, "val_accuracy": val_accuracy,
  "train_loss": running_loss, "val_loss": val_loss})
```

También, modificamos levemente el método `transform`, para incluir *data augmentation*, ya que se mencionó en clase que era un pre-procesamiento de datos habitual. En un principio lo agregamos a `transform`, lo cual nos dimos cuenta era un error, pues esa transformación también se aplica a `validation` y `test`, cuando nosotros solo la queremos en `training`. Para ello, modificamos el código de esta manera:

```
transform_train = transforms.Compose([
  transforms.RandomHorizontalFlip(p=0.5),
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Descargo el dataset CIFAR10, divido en training, validation, testing.
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
                                        transform=transform_train)
valset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
                                        transform=transform)
targets_ = trainset.targets
train_idx, val_idx = train_test_split(np.arange(len(targets_)), test_size=0.2,
                                        stratify=targets_)
```

```
train_sampler = torch.utils.data.SubsetRandomSampler(train_idx)
val_sampler = torch.utils.data.SubsetRandomSampler(val_idx)

trainloader = torch.utils.data.DataLoader(trainset, sampler=train_sampler,
                                          batch_size=batch_size, num_workers=2)
valloader = torch.utils.data.DataLoader(valset, sampler=val_sampler,
                                          batch_size=batch_size, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
                                          transform=transform)
```

Sin embargo, no vimos este error en un comienzo, y algunos de nuestros experimentos de los cuales se habla en este informe fueron corridos con ese error. Al haber cometido el mismo error en todos estos experimentos, consideramos que son comparables entre sí y que podemos utilizar esa información para tomar decisiones. De igual manera, re-hicimos varios de estos experimentos, y, afortunadamente, nos dieron muy similares, haciendonos ver que el error no fue un factor determinante en los resultados. A continuación se pueden ver algunos ejemplos.

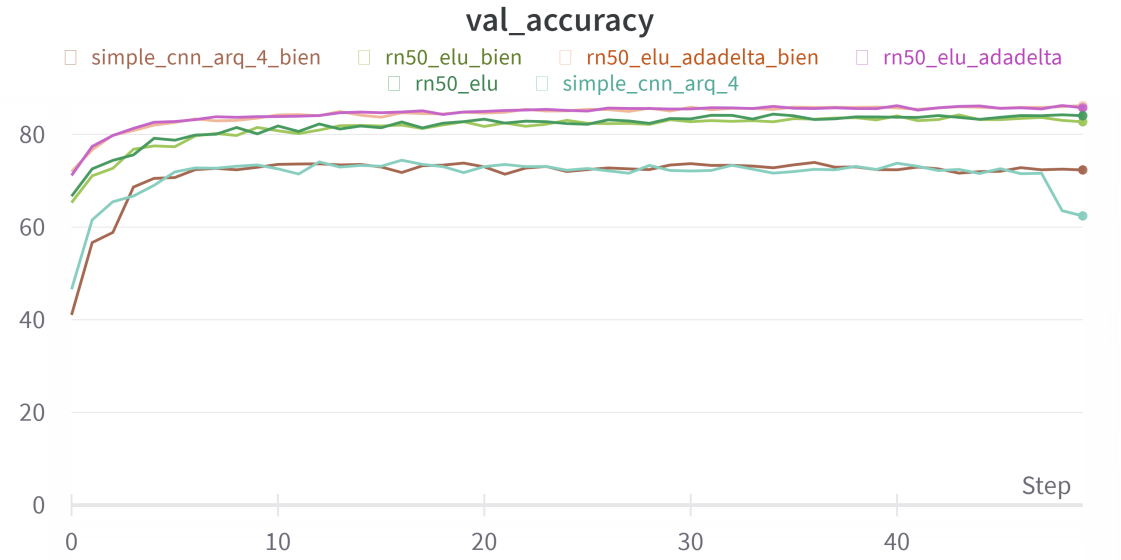

Figura 1: Diferencia muy pequeña con y sin transformación en validation (aquellas `_bien` son con el error corregido)

Arquitectura

Para comenzar a clasificar, mantuvimos los parámetros que se encontraban en el código otorgado, sólo cambiando `epochs` a 50, para darle una oportunidad a todas nuestras estructuras de entrenar un poco mas, ya que muchas veces con 10 epochs podíamos observar potencial de seguir creciendo y se cortaba. Esto nos deja con los siguientes parámetros para todos los experimentos mencionados en esta sección (notar que, si cambiamos algo más que la estructura, la mejora o desmejora se puede deber a esa otra cosa que se cambia en lugar a la estructura, que es lo que estamos intentando perfeccionar).

```
# Parametros
batch_size = 32
learning_rate = 0.02
momentum = .9
epochs = 50
```

Además, en todas los experimentos utilizamos la misma función de activación (ReLU), para asegurarnos de que en efecto lo único que cambiaba era la arquitectura, haciendolos comparables entre sí.

Para sentar una base de performance, corrimos el modelo que venía en el código `MLP`. No obtuvimos la mejor de las performances con un 53% de accuracy en validation en su mejor epoch, pero sirvió para entender donde estabamos parados. Luego, probamos distintas arquitecturas, entre ellas: - `relu_layer_div2`: Dividiendo entre 2 todo el tiempo la cantidad de features(**no es neuronas por capa??**), desde $32 \cdot 32 \cdot 3$ hasta llegar a las 10 features que deben salir de output (pues hay 10 clases a clasificar). - `relu_layer_div4`: Dividiendo entre 4, desde $32 \cdot 32 \cdot 3$ hasta 10. - `relu_div4_div2`: Intercalando divisiones entre 4 y 2, desde $32 \cdot 32 \cdot 3$ hasta 10. - `relu_6_layer_divrand`: Con 6 capas sin ningún criterio para la disminución de neuronas por capa. - `relu_4_layer_divrand`: Con 4 capas sin ningún criterio para la disminución de neuronas por capa.

Entre estas arquitecturas, varias de sus perfomances fueron similares, pero obtuvimos una arquitectura que fue mejor que las demás. Sin embargo la diferencia no fue mucha y la accuracy, si bien subió bastante, parecía llegar a un techo con este tipo de redes, o al menos no obtuvimos resultados mucho mejores que un 55% (esto, sin optimizar hiperparámetros). En las prácticas hemos hablado de que estos problemas de clasificación de imágenes se suelen resolver con redes convolucionales, por lo que decidimos pasar primero a este tipo de redes antes de seguir con los siguientes puntos de probar optimizadores, regularizadores y distintos hiperparámetros, pues consideramos que es lo mas razonable con el fin de obtener un mejor modelo de clasificación.

En las figuras dos y tres, se puede observar la accuracy y loss de tanto nuestros experimentos (dos de similares resultados) en cuanto a arquitecturas de redes densas se refiere. Como se puede ver en la figura cuatro, los mejores resultados fueron obtenidos dividiendo por 2 y dividiendo por 4 las features a medida que agregabamos capas, por lo cual utilizaremos esta idea en las layers fully conected luego de las convoluciones, en la siguiente sección de este informe.

Arquitectura CNN

Para experimentar arquitecturas de redes convulcionales, seguimos un procedimiento similar al realizado para las arquitecturas previas. Comenzamos corriendo al red convulcional provista (`default_conv`) para marcar ciero *benchmark*. A partir de allí, probamos las siguientes arquitecturas de CNNs:

- `simple_cnn`: Conformada por una primera capa convulcional que toma los 3 canales originales (RGB) y aplica 32 filtros de convolución con un kernel de tramaño 3x3 y un padding de 1. Luego una capa de agrupación (pooling) que reduce a la mitad el tamaño espacial de entrada, y una segunda capa convulcional que aplica 64 filtros con un kernel de 3x3 y padding. Finalmente termina con 2 capas *fully connected* para llegar a los 10 outputs.
- `simple_cnn_arq_2`: A las mismas capas de convulsion y pooling de la arquitectura anterior, le aplicamos capas *fully connected* dividiendo entre 2, desde $64 * 8 * 8$ a 10, pues fue la arquitectura que mejor perfemance anteriormente.
- `simple_cnn_arq_4`: al igual que la arquitectura anterior pero diviendo entre 4 las capas *fully connected*.

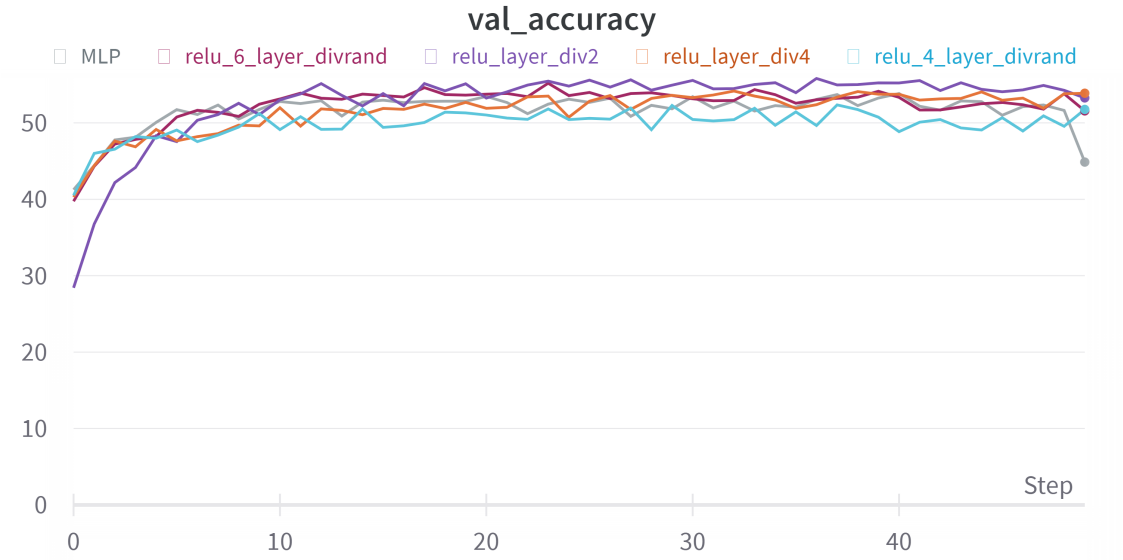

Figura 2: Experimentos arquitectura (accuracy)

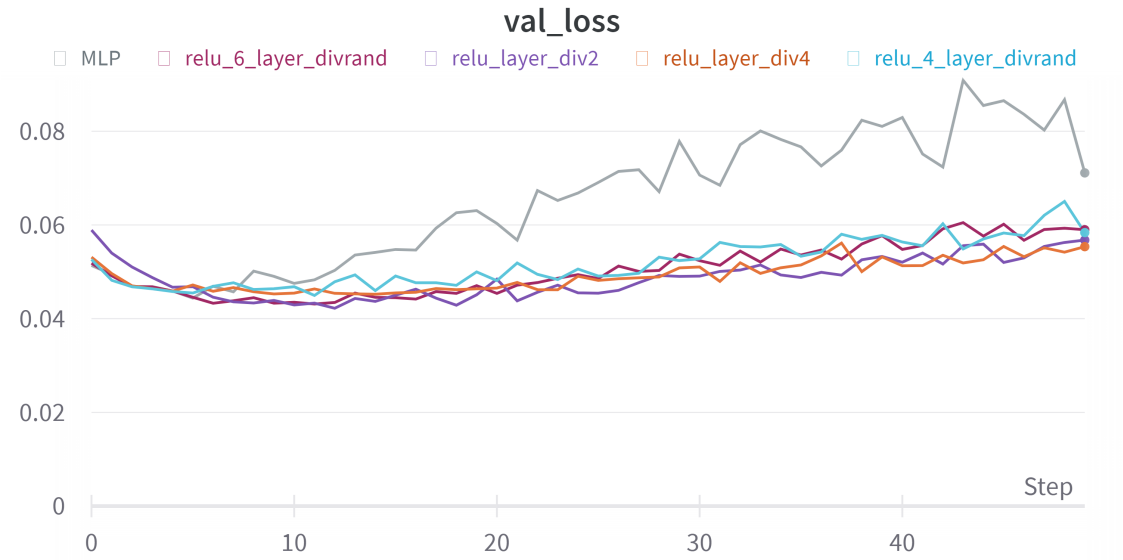

Figura 3: Experimentos arquitectura (loss)

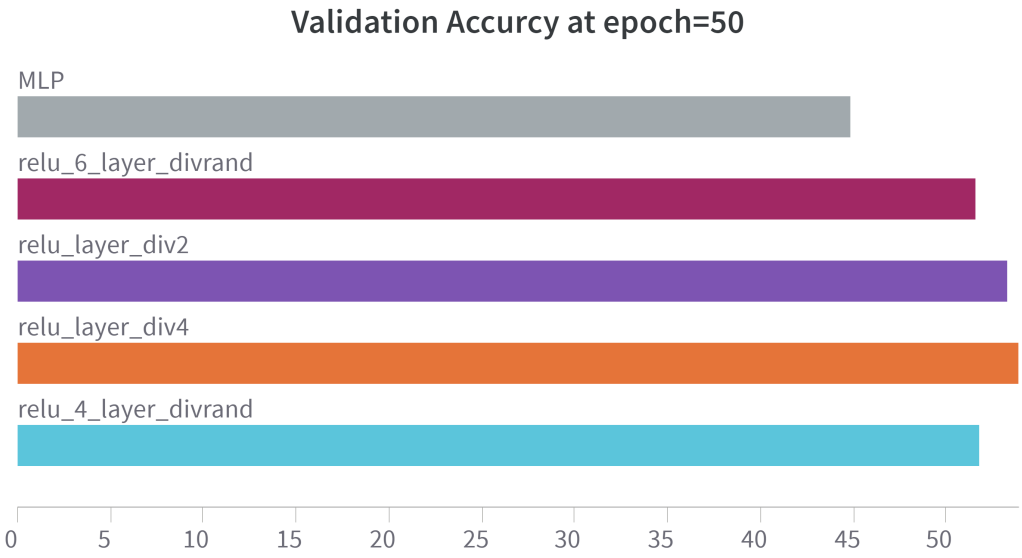

Figura 4: Experimentos arquitectura CNN

Si bien con estas arquitecturas de CNN logramos notables mejoras con respecto a las arquitecturas sin capas convulsionales (con accuracy por encima de 0.7), dado que se sugirió desde la cátedra probar arquitecturas famosas, probamos la Resnet. Para ello, importamos desde el sub-paquete `torchvision.models` el modelo pre entrenado `resnet50` de la siguiente manera:

```
import torchvision.models as models

class NetConv(nn.Module):
  def __init__(self):
    super().__init__()

    # Cargamos el modelo ResNet-50 con los pesos pre-entrenados
    self.resnet50 = models.resnet50(pretrained=True)

    # Y a continuación las capas fully-connected según nuestro mejores
    # resultados dividiendo entre 4
    num_features = self.resnet50.fc.in_features
      self.resnet50.fc = nn.Sequential(
        nn.Linear(num_features, 1024),
        nn.ReLU(),
        nn.Linear(1024, 256),
        nn.ReLU(),
        nn.Linear(256, 64),
        nn.ReLU(),
        nn.Linear(64, 16),
        nn.ReLU(),
        nn.Linear(16, 10)
      )
```

```
def forward(self, x):
  x = self.resnet50(x)
  return x
```

Si bien, cargamos el modelo con sus pesos pre-entrenados, por default los parámetros son importados como `param.requires_grad = True` lo que significa que serán modificados al entrenar con el dataset CIFAR10.

En las figuras 5, 6 y 7, podemos ver que nuestras CNNs con capas totalmente conectadas con división entre 4 y 2 superan a la simple y a la default, lo cuál parece tener sentido considerando nuestros previos experimentos. A su vez, es notable que la Resnet supera las otras redes ya que su accury supera el 80%.

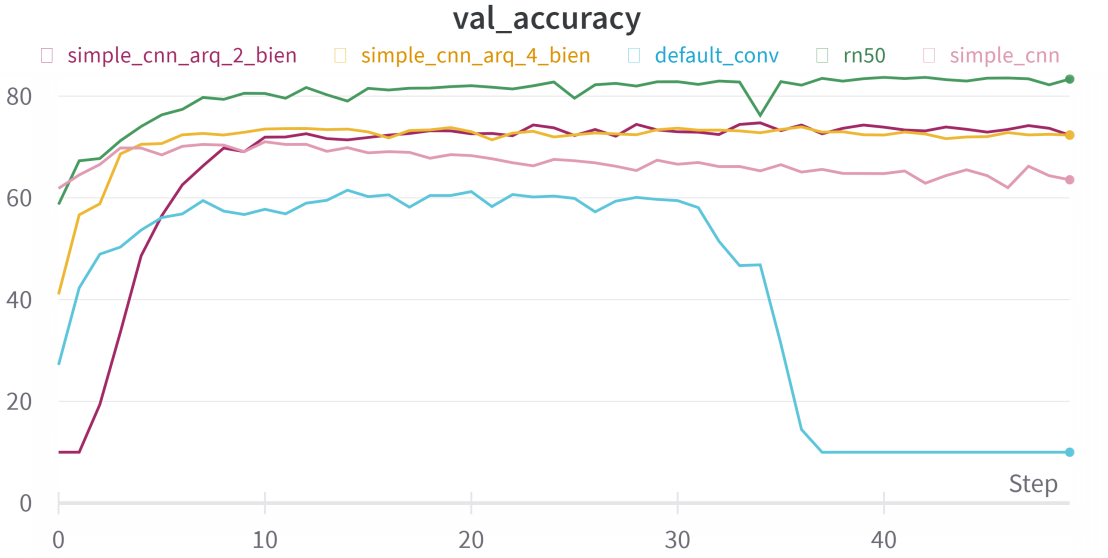

Figura 5: Experimentos arquitectura CNN (accuracy)

Funciones de activación

Optimizadores

Entrenamiento

Regularización

Evaluación final

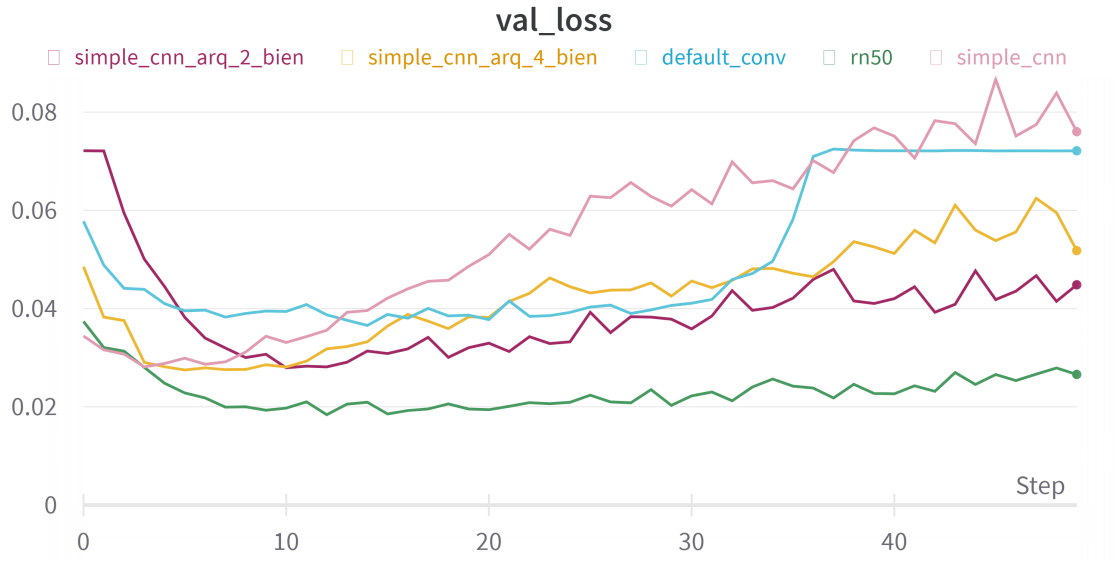

Figura 6: Experimentos arquitectura CNN (loss)

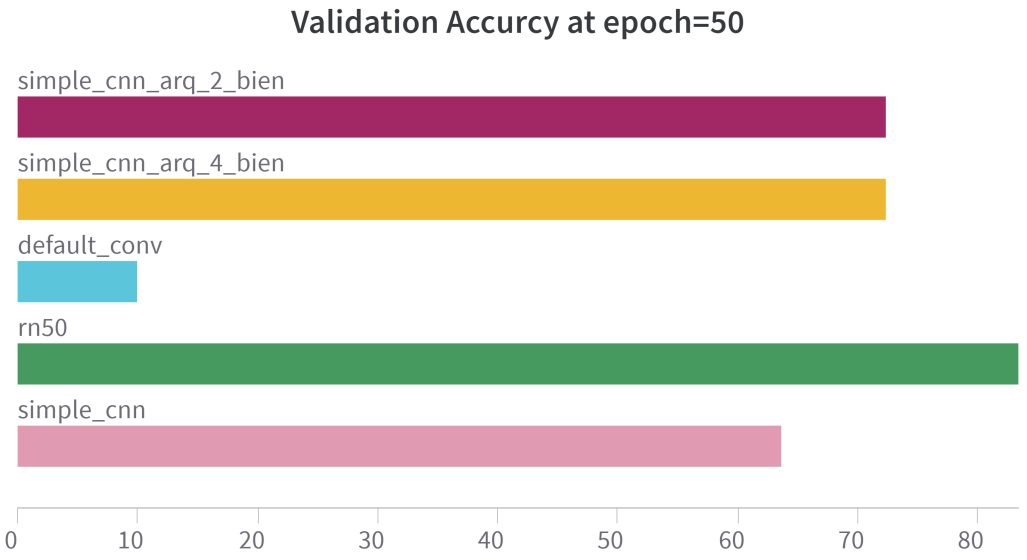

Figura 7: Experimentos arquitectura CNN