Detección de Rostros Reales y Falsos con Redes Neuronales Convolucionales

Jaime Ballesteros Domínguez
jaime.ballesteros101@alu.ulpgc.es

José Gabriel Reyes Rodríguez
jose.reyes121@alu.ulpgc.es

15 de enero de 2024

Resumen

En este trabajo se pedía desarrollar un modelo de clasificación basado en una red neuronal convolucional capaz de distinguir imágenes reales de falsas. Para conseguir un buen rendimiento de dicho clasificador, se probó con distintas funciones de pérdida y optimización, con diferentes batch sizes y se experimento con las funciones de entrenamiento y evaluación. Principalmente podemos concluir que la creación de una red neuronal es extremadamente delicado, debido a los retos de compatibilidad que presentan los distintos modelos, funciones de pérdida, optimización y otros. En esta tarea se ha podido comprobar como ciertos modelos pueden dar resultados notables con el clasificador adecuado o pésimos si no se elige el correcto.

1. Introducción

En la era digital actual, la creciente proliferación de imágenes falsificadas, especialmente en plataformas de redes sociales, plantea desafíos significativos para la autenticidad de la información. En este trabajo se nos propone abordar este problema mediante el desarrollo de un modelo de clasificación basado en Convolutional Neural Networks (CNN). El objetivo principal es distinguir entre imágenes de rostros auténticas y aquellas alteradas por expertos en falsificación, lo cual es una tarea difícil dada la diversidad de técnicas empleadas por los falsificadores. Se dispone de un conjunto de datos que incluye imágenes clasificadas según su dificultad de detección (fácil, medio y difícil), lo que permitirá evaluar la eficacia del modelo en diferentes escenarios.

2. Metodología

En esta sección se explicarán los cuadros de código de esta red neuronal, obviando aquellos dedicados a la importación o descarga de datos.

```
data_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225])
])

full_dataset = datasets.ImageFolder(root=DATASET,
transform=data_transforms)

train_size = int(0.7 * len(full_dataset))
valid_size = int(0.2 * len(full_dataset))
test_size = len(full_dataset) - train_size - valid_size

train_dataset, valid_dataset, test_dataset = random_split(
    full_dataset, [train_size, valid_size, test_size])

batch_size = 32

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

Como hemos podido ver en la sección de arriba, esta trata todo el preparado de datos. Como podemos ver en los distintos transforms, estos aplican una serie de cambios a las imágenes del dataset como puede ser su redimensionamiento o girarla horizontalmente de manera aleatoria. Después de esto, se divide el dataset en tres partes; entrenamiento, validación y test.

```
#pretrained = models.vgg16(pretrained=True)
pretrained = models.resnet18(pretrained=True)

for i, param in enumerate(pretrained.parameters()):
    print(i, param.shape)
    if i < 14:
        param.requires_grad = False
```

En este apartado, el bucle for itera sobre todos los parámetros del modelo e imprime su índice y forma. Esto puede ser útil para comprender

la estructura del modelo y decidir qué capas congelar. En nuestro caso se decidió congelar los primeros 14 bloques de parámetros, lo que significa que estos no se actualizarán durante el entrenamiento.

```
pretrained.classifier = None
pretrained.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(512 * 7 * 7, 2048),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(2048, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 2),
)
```

Aquí, la primera línea elimina las capas del clasificador predefinido, mientras que las siguientes definen el nuevo clasificador. Dado que este problema se ha definido como uno de clasificación lineal entre verdadero y falso, la última capa contendrá únicamente dos salidas.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
pretrained.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(pretrained.parameters(),
lr=0.0001, weight_decay=0.01)
#optimizer = optim.SGD(pretrained.parameters(),
lr=0.001, momentum=0.9)
#optimizer  = optim.AdamW(pretrained.parameters(),
lr=0.0001, weight_decay=0.01)
#optimizer = optim.RMSprop(pretrained.parameters(),
lr=0.00005, momentum=0.9)
#optimizer = optim.Adagrad(pretrained.parameters(),
lr=0.005, lr_decay=0.01, weight_decay=0.01)
```

En esta sección tenemos una de las partes más importantes del código, siendo esta la selección de las funciones de pérdida y optimización. En el caso de la función de pérdida, la única que nos proporcionó datos coherentes fue CrossEntropyLoss, ya que BCELoss presentaba una precisión mínima o muy cambiante. Entre las funciones de optimización que se usaron en este trabajo, las que mejores resultados fueron Adam, SGD, AdamW, RMSprop y Adagrad.

```
def train_model(model, criterion, optimizer, train_loader):
```

```
    model.train()
    running_loss = 0.0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
    epoch_loss = running_loss / len(train_dataset)


    return epoch_loss

def evaluate_model(model, data_loader):
    model.eval()
    correct = 0
    total = 0

    for inputs, labels in data_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')
    return accuracy
```

Seguidamente, se añaden las funciones de entrenamiento (train-model) y evaluación (evaluate-model). El objetivo principal de la función train-model es entrenar un modelo utilizando un conjunto de entrenamiento. Cuando se ejecuta, cambia el modelo al modo de entrenamiento, itera sobre el conjunto de entrenamiento en el DataLoader, calcula la pérdida con la función definida, realiza retropropagación y actualiza los parámetros del modelo en el optimizador. Finalmente, se devuelve la pérdida promedio por época. Esto se calcula dividiendo la pérdida acumulada total por el tamaño del conjunto de entrenamiento.

La función evaluate-model, por otro lado, utiliza el conjunto de datos de evaluación para evaluar la precisión del modelo. Durante el desarrollo, cambia su modelo al modo de evaluación y usa un DataLoader para iterar sobre el conjunto de datos de evaluación, calcular las predicciones del modelo y compararlas con las etiquetas reales para determinar la precisión del modelo. Luego imprime la precisión determinada en el conjunto de datos de evaluación y devuelve este valor. Estas características desem-

peñan un papel clave en la capacitación y evaluación efectiva de modelos de aprendizaje profundo.

```
num_epochs = 20
loss_list = []
accuracy_list = []
for epoch in range(num_epochs):
    train_loader = DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset,
    batch_size=batch_size, shuffle=True)
    epoch_loss = train_model(pretrained, criterion,
    optimizer, train_loader)
    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}')
    accuracy = evaluate_model(pretrained, valid_loader)

    loss_list.append(epoch_loss)
    accuracy_list.append(accuracy)
```

Posteriormente, este fragmento de código se encarga de entrenar y evaluar el modelo seleccionado durante una serie de épocas. En cada época, los conjuntos de entrenamiento y validación se cargan en lotes. Después, el modelo se entrena con el conjunto de entrenamiento utilizando la función train-model y se genera la pérdida promedio de cata iteración. Luego, la precisión del modelo se evalúa con respecto al conjunto de validación utilizando evaluate-model y se imprime en cada época. Finalmente, dichos valores de pérdida y precisión se guardan en listas que utilizaremos en el siguiente paso.

```
plt.figure(figsize=(10,5))
plt.title("Loss vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Loss")
plt.plot(range(1, num_epochs + 1), loss_list, label="Training Loss")
plt.grid(True)

plt.ylim(0, 1)

plt.show()

plt.figure(figsize=(10,5))
plt.title("Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Accuracy")
plt.plot(range(1, num_epochs + 1), accuracy_list, label="Validation Accuracy")
plt.grid(True)

plt.ylim(50, 80)
```

```
plt.show()
```

Para ir concluyendo, en este paso simplemente se imprimirá la progresión del error y de la precisión en cada época.

```
evaluate_model(pretrained, test_loader)
```

Finalmente, se evaluará la precisión del modelo ya entrenado y evaluado en el conjunto de test.

3. Experimentos

Como ya se ha comentado en varias ocasiones, a la hora de encontrar la mejor combinación, se realizaron varias pruebas con diferentes modelos preentrenados, funciones de pérdida y de optimización. Debido a que la función de pérdida BCELoss presentaba una serie de dificultades y tasas de precisión bajas, se decidió descartarla. Por lo tanto, los siguientes resultados obtenidos se han logrado con la función de pérdida CrossEntropyLoss:

VGG16		
Optimizador	**Métrica 1**	**Métrica 2**
Adam	70.73 %	68.29 %
SGD	67.80 %	64.39 %
AdamW	62.93 %	63.41 %
RMSprop	67.32 %	68.29 %
Adagrad	66.83 %	67.32 %
RESNET18		
Optimizador	**Métrica 1**	**Métrica 2**
Adam	74.15 %	70.24 %
SGD	67.32 %	70.24 %
AdamW	67.32 %	70.73 %
RMSprop	67.80 %	67.32 %
Adagrad	73.17 %	65.85 %

Como se puede apreciar, la función de optimización Adam se presenta como la más adecuada en términos de precisión en ambos modelos. También se puede ver que el modelo Resnet-18 presenta mejores resultados en 4 de las 5 funciones.

3.1. VGG16 vs Resnet-18 (Adam)

Para visualizar un poco más las diferencias entre ambos modelos, a continuación se mostrarán las gráficas de pérdida y precisión de la función de optimización Adam, al ser esta la que presenta los mejores resultados.

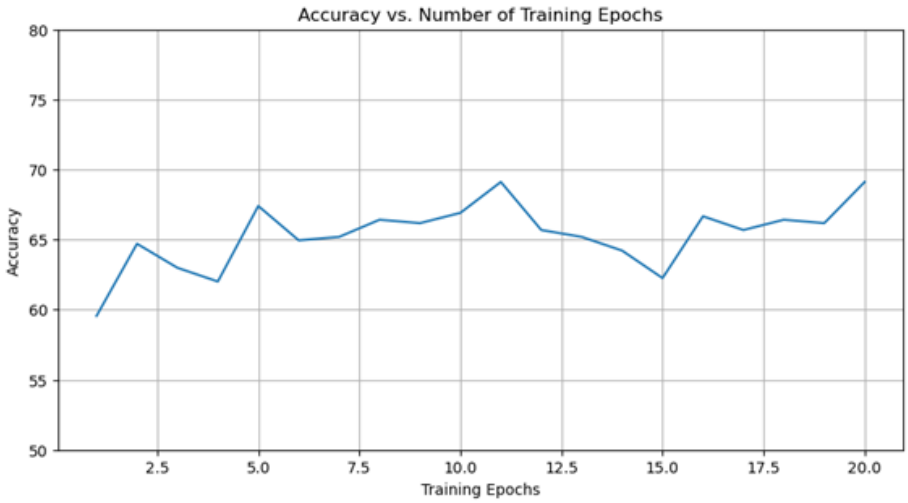

Figura 1: Precisión VGG16.

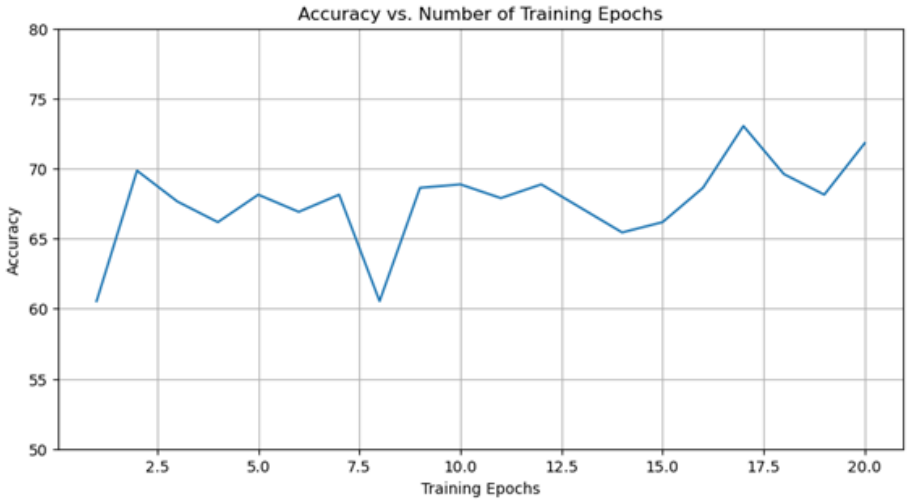

Figura 2: Precisión Resnet-18.

Si se miran ambas gráficas con detenimiento, podremos ver cómo la precisión obtenida en VGG16 ronda más el 65 %, con algunos acercamientos al 70 %, mientras que la precisión con Resnet-18 presenta unos valores bastante más cercanos al 70 %. También se puede apreciar cómo en la segunda gráfica que incluso llegan o sobrepasan dicho porcentaje.

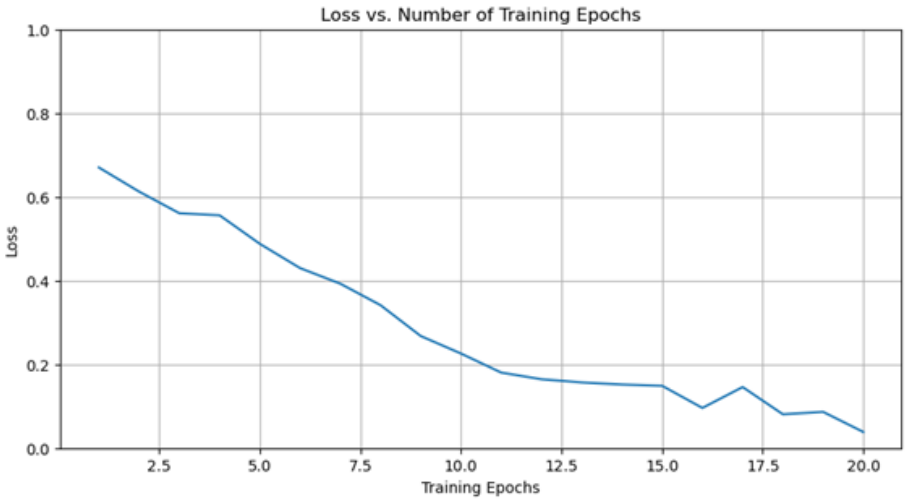

Figura 3: Precisión Resnet-18.

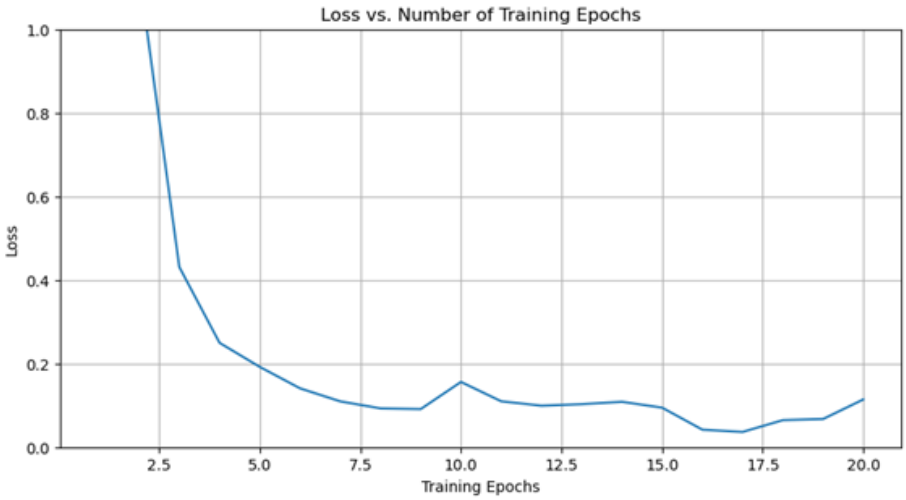

Figura 4: Precisión Resnet-18.

Sin embargo, donde más se pueden apreciar las diferencias entre modelos es en la pérdida. En este caso podemos apreciar un descenso constante y casi ininterrumpido en la pérdida del modelo VGG16, a la vez que se mantiene en valores menores al 1. En contraposición a este decrecimiento constante, la pérdida en Resnet-18 presenta una brusca caída que parte

de valores superiores al 1, pero que a las pocas iteraciones ya se asienta en valores inferiores al 0,2.

4. Conclusiones

Como hemos podido comprobar en los anteriores apartados, y a falta de evaluar otros modelos preentrenados, sería recomendable la utilización del modelo Resnet-18. A rasgos generales, este modelo presenta una capacidad de aprendizaje mayor y más rápida que su contraparte de VGG16. Por último, entre las distintas funciones de optimización estudiadas, Adam presenta los resultados más prometedores, ya sea por su rápida convergencia o a lo mejor por ser la que tiene los hiperparámetros más adecuados.

De cara a trabajos futuros podríamos mejorarlos usando conjuntos de datos distintos a los ya presentados, para ver cómo se adapta la red a nueva información. También, se podrían probar otros modelos distintos para comprobar si alguno mejora a los que hemos usado. Finalmente, cabe destacar que los hiperparámetros seguramente tengan más capacidad de afinarse para mejorar la precisión de nuestra red.