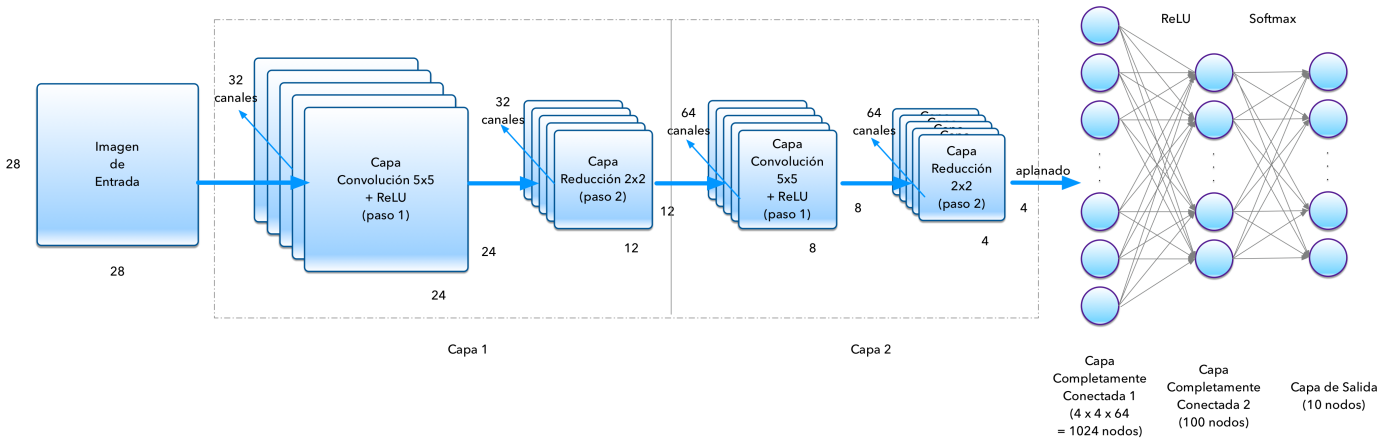
Clasificar dígitos escritos a mano MNIST usando una Red Neuronal Convolucional

Arquitectura de la Red Neuronal Convolucional



Como se puede observar, comenzamos con las imágenes de dígitos en escala de grises MNIST 28 × 28. Luego creamos 32, 5 × 5 filtros convolucionales/canales más activaciones de nodos ReLU. Después de esto, tenemos una altura y un ancho de 24. A continuación, realizamos una reducción aplicando una operación de reducción por valor máximo de 2 × 2 con una paso de 2, resultando en una altura y un ancho de 12. La capa segunda capa consiste en la misma estructura, pero ahora con 64 filtros/canales (altura y ancho de 8) y otra reducción por valor máximo con paso de 2 (altura y ancho de 4). A continuación, aplanamos la salida para obtener una capa completamente conectada con 1024 nodos (4 * 4 * 64), seguida de otra capa oculta de 100 nodos. Estas capas usarán activaciones de nodo ReLU. Finalmente, usamos una capa de clasificación de softmax para dar salida a las probabilidades de los 10 dígitos.

Formulas para calcular los tamaños de salida de las capas de convolución y *pooling* (agregación o reducción)

La formula que relaciona el tamaño de salida de la convolución con el tamaño de la entrada es:

$$W = \lfloor \frac{W_{prev} - f + 2 \times relleno}{paso} \rfloor + 1$$

$$f = \text{número de filtros usados en la convolución}$$

La formula que relaciona el tamaño de salida del *pooling* con el tamaño de la entrada es:

$$W = \lfloor \frac{W_{prev} - f}{paso} \rfloor + 1$$

$$f = \text{tamaño de la ventana de pooling}$$

Importar librerías

In [1]:
```
import torch, torchvision
from torchvision import datasets, transforms
from torch import nn, optim
from torch.nn import functional as F

import numpy as np
```

Cargar los datos

In [2]:
```
dispositivo = 'cuda' if torch.cuda.is_available() else 'cpu'
# Definir una transformación para normalizaz la data
transformacion = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize((0.5,), (0.5,)),
                                     ])

# Bajar y cargar la data de entrenamiento
datos_entrenamiento = datasets.MNIST('../datos/MNIST_data/', download=True, train=True, tra
datos_validacion = datasets.MNIST('../datos/MNIST_data/', download=True, train=False, trans
cargador_entrenamiento = torch.utils.data.DataLoader(datos_entrenamiento, batch_size=64, sh
cargador_validacion = torch.utils.data.DataLoader(datos_validacion, batch_size=64, shuffle=
```

Definir el modelo en PyTorch

In [3]:
```
class RedConvolucional(nn.Module):
    def __init__(self):
        super(RedConvolucional, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(1024, 100),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(100, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(-1, 1024)
        x = self.fc_layers(x)
        return x
```

Definir funciones de entrenamiento y de prueba

In [10]:
```python
def entrenar(modelo, dispositivo, cargador_entrenamiento, optimizador, perdida_fn, epoca):
    modelo.train()

    perdida_actual = 0
    for imagenes, etiquetas in cargador_entrenamiento:
        # Enviar datos al dispositivo
        imagenes, etiquetas = imagenes.to(dispositivo), etiquetas.to(dispositivo)

        # 1. Propagar hacia adelante los datos de entrenamiento usando el método forward()
        salida = modelo(imagenes)

        # 2. Calcule la pérdida (qué tan diferentes son las predicciones de nuestros modelo
        perdida = perdida_fn(salida.log(), etiquetas)

        # 3. Colocar a cero los gradientes del optimizador
        optimizador.zero_grad()

        # 4. Propagación hacia atrás
        perdida.backward()

        # 5. Realizar paso de optimización
        optimizador.step()

        perdida_actual += perdida

    print(f'Epoca: {epoca} | Perdida Entrenamiento: {perdida_actual/len(cargador_entrenamie

def validar(modelo, dispositivo, cargador_validacion, perdida_fn):
      ### Prueba

    # Colocar el modelo en modo evaluación
    modelo.eval()

    with torch.inference_mode():
        perdida_actual = 0
        for imagenes, etiquetas in cargador_validacion:
            # Enviar datos al dispositivo
            imagenes, etiquetas = imagenes.to(dispositivo), etiquetas.to(dispositivo)

            # 1. Propagar hacia adelante los datos de entrenamiento usando el método forwar
            salida = modelo(imagenes)
             # 2. Calcule la pérdida (qué tan diferentes son las predicciones de nuestros m
            perdida = perdida_fn(salida.log(), etiquetas)

            perdida_actual += perdida

        print(f'Epoca: {epoca} | Perdida Validación: {perdida_actual/len(cargador_validacio

def probar(modelo, dispositivo, cargador_validacion):
    modelo.eval()
    conteo_correcto, conteo_total = 0, 0
    for imagenes,etiquetas in cargador_validacion:
        # Enviar datos al dispositivo
        imagenes, etiquetas = imagenes.to(dispositivo), etiquetas.to(dispositivo)
        for i in range(len(etiquetas)):
            imagen = imagenes[i]
            # Apagar gradientes para acelerar esta parte
            with torch.inference_mode():
                logps = modelo(imagen)

            # Salida de la red son probabilidades logarítmicas,
```

```
            # se necesita tomar el exponente para obtener probabilidades
            ps = torch.exp(logps)
            probab = list(ps.numpy()[0])
            etiqueta_predicha = probab.index(max(probab))
            etiqueta_verdadera = etiquetas.numpy()[i]
            if(etiqueta_verdadera == etiqueta_predicha):
                conteo_correcto += 1
            conteo_total += 1

    print("Número de Imágenes Probadas =", conteo_total)
    print('Número de Imágenes Correctas =', conteo_correcto)
    print("\nExactitud del Modelo =", (conteo_correcto/conteo_total))
```

Instanciar modelo, definir la funcion de perdida y el optimizador

In [11]:
```
epocas = 10
modelo = RedConvolucional().to(dispositivo)
perdida_fn = nn.NLLLoss(reduction='mean')
optimizador = optim.SGD(modelo.parameters(), lr=0.01, momentum=0.5)
```

In [12]:
```
for epoca in range(1, epocas + 1):
    entrenar(modelo, dispositivo, cargador_entrenamiento, optimizador, perdida_fn, epoca)
    validar(modelo, dispositivo, cargador_validacion, perdida_fn)
    print('-----------------------------------------')

probar(modelo, dispositivo, cargador_validacion)
```

```
Epoca: 1 | Perdida Entrenamiento: 0.521513
Epoca: 1 | Perdida Validación: 0.163169
-----------------------------------------
Epoca: 2 | Perdida Entrenamiento: 0.165811
Epoca: 2 | Perdida Validación: 0.095199
-----------------------------------------
Epoca: 3 | Perdida Entrenamiento: 0.122940
Epoca: 3 | Perdida Validación: 0.076494
-----------------------------------------
Epoca: 4 | Perdida Entrenamiento: 0.101993
Epoca: 4 | Perdida Validación: 0.058770
-----------------------------------------
Epoca: 5 | Perdida Entrenamiento: 0.088693
Epoca: 5 | Perdida Validación: 0.048650
-----------------------------------------
Epoca: 6 | Perdida Entrenamiento: 0.079083
Epoca: 6 | Perdida Validación: 0.049575
-----------------------------------------
Epoca: 7 | Perdida Entrenamiento: 0.071980
Epoca: 7 | Perdida Validación: 0.042948
-----------------------------------------
Epoca: 8 | Perdida Entrenamiento: 0.066949
Epoca: 8 | Perdida Validación: 0.040246
-----------------------------------------
Epoca: 9 | Perdida Entrenamiento: 0.061834
Epoca: 9 | Perdida Validación: 0.037604
-----------------------------------------
Epoca: 10 | Perdida Entrenamiento: 0.058775
Epoca: 10 | Perdida Validación: 0.034641
-----------------------------------------
Número de Imágenes Probadas = 10000
Número de Imágenes Correctas = 9918

Exactitud del Modelo = 0.9918
```

Guardar el modelo

Estamos contentos con las predicciones de nuestros modelos, así que guardémoslo en un archivo para que pueda usarse más tarde.

In [13]:
```
from pathlib import Path

# 1. Crear un directorio para los modelos
MODELOS_PATH = Path('../modelos')
MODELOS_PATH.mkdir(parents=True, exist_ok=True)

# 2. Crear path para el modelo
MODELO_NOMBRE = "22-RNC_modelo.pth"
MODELO_DIRECCION = MODELOS_PATH / MODELO_NOMBRE

# 3. Guardar el state_dict del modelo
print(f"Guardando modelo en: {MODELO_DIRECCION}")
torch.save(obj=modelo.state_dict(), # guardando state_dict() solo guarda los parámetros del
           f=MODELO_DIRECCION)
```

```
Guardando modelo en: ..\modelos\22-RNC_modelo.pth
```

Cargar modelo y continuar entrenamiento

In [14]:
```
# Instanciar una nueva instancia del modelo
modelo = RedConvolucional()

# Cargar el state dict del modelo
modelo.load_state_dict(torch.load(MODELO_DIRECCION))

# Colocar el modelo en el dispositivo destino (si los datos estan en el GPU, el modelo tier
modelo.to(dispositivo)

print(f"Modelo cargado:\n{modelo}")
```

```
Modelo cargado:
RedConvolucional(
  (conv_layers): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): ReLU()
    (3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (4): Dropout(p=0.5, inplace=False)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): ReLU()
  )
  (fc_layers): Sequential(
    (0): Linear(in_features=1024, out_features=100, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=100, out_features=10, bias=True)
    (4): Softmax(dim=1)
  )
)
```

In [15]:
```
epocas = 5
for epoca in range(1, epocas + 1):
    entrenar(modelo, dispositivo, cargador_entrenamiento, optimizador, perdida_fn, epoca)
    validar(modelo, dispositivo, cargador_validacion, perdida_fn)
    print('-----------------------------------------')

probar(modelo, dispositivo, cargador_validacion)
```

```
Epoca: 1 | Perdida Entrenamiento: 0.051815
Epoca: 1 | Perdida Validación: 0.034446
-----------------------------------------
Epoca: 2 | Perdida Entrenamiento: 0.052512
Epoca: 2 | Perdida Validación: 0.034436
-----------------------------------------
Epoca: 3 | Perdida Entrenamiento: 0.054210
Epoca: 3 | Perdida Validación: 0.034523
-----------------------------------------
Epoca: 4 | Perdida Entrenamiento: 0.053778
Epoca: 4 | Perdida Validación: 0.034577
-----------------------------------------
Epoca: 5 | Perdida Entrenamiento: 0.054417
Epoca: 5 | Perdida Validación: 0.035247
-----------------------------------------
Número de Imágenes Probadas = 10000
Número de Imágenes Correctas = 9918

Exactitud del Modelo = 0.9918
```

In [ ]: