
Neural Networks with PyTorch Lightning model

PyTorch Lightning is a lightweight wrapper for organizing PyTorch code in a structured and modular way. It provides a high-level interface that automates much of the boilerplate code typically required for training PyTorch models, allowing researchers and practitioners to focus more on experimenting with models and less on engineering tasks.

Here's why PyTorch Lightning is important:

1. **Structured Code**: PyTorch Lightning encourages writing organized, structured code by separating concerns into distinct components like data handling, model definition, training, and evaluation. This makes the codebase cleaner, more readable, and easier to maintain.

2. **Reproducibility**: By abstracting away much of the training loop and providing standard hooks for various stages of training, PyTorch Lightning helps ensure reproducibility across different experiments and environments. This is crucial for research and development, where reproducibility is a fundamental aspect.

3. **Reduced Boilerplate**: PyTorch Lightning eliminates a significant amount of boilerplate code that would otherwise be necessary for tasks like setting up training loops, handling distributed training, and logging metrics. This allows practitioners to focus more on designing experiments and less on writing infrastructure code.

4. **Flexibility**: Despite its high-level abstractions, PyTorch Lightning remains highly flexible, allowing users to customize and extend functionality as needed. It provides hooks and callbacks for incorporating custom logic at various stages of the training process.

5. **Community and Ecosystem**: PyTorch Lightning has a growing community of users and contributors, which means access to a wealth of resources, tutorials, and pre-built components. This fosters collaboration and accelerates development by leveraging shared knowledge and best practices.

In [ ]:
```
 #%%capture
%pip install lightning
%pip install torchvision torchmetrics lightning
```

In [35]:
```
import os, torch
import torch.nn as nn
import torch.utils.data as data
import torchvision as tv
```

```
from torch.utils.data import random_split, DataLoader, Subset
from torchmetrics import Accuracy
import lightning as L
```

In [36]:
```
if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

print("You are using", device)
```

```
You are using mps
```

Checking GPU information

In [ ]:
```
!nvidia-smi #only works for device = "gpu"!!!!!!!
```

In [ ]:
```
!nvidia-smi -L
```

In [37]:
```
import torch

# Detectar el dispositivo disponible
if torch.cuda.is_available():
    # Sistema con GPU NVIDIA (CUDA)
    num_gpus = torch.cuda.device_count()
    os.environ["CUDA_VISIBLE_DEVICES"] = ",".join(str(x) for x in range(n
    device = "cuda"
    print(f"✓ GPU CUDA detectada: {num_gpus} GPU(s) disponible(s)")
    print(f"  Dispositivo: {torch.cuda.get_device_name(0)}")
    print(f"  CUDA_VISIBLE_DEVICES = {os.environ['CUDA_VISIBLE_DEVICES']}
elif torch.backends.mps.is_available():
    # Sistema macOS con Apple Silicon (MPS/MPU)
    device = "mps"
    print("✓ MPU (Metal Performance Shaders) detectado")
    print("  Dispositivo: Apple Silicon GPU")
    print("  Aceleración de hardware habilitada para macOS")
else:
    # Fallback a CPU
    device = "cpu"
    print("⚠ No se detectó GPU/MPU - usando CPU")

print(f"Dispositivo seleccionado: {device}")
```

```
✓ MPU (Metal Performance Shaders) detectado
  Dispositivo: Apple Silicon GPU
  Aceleración de hardware habilitada para macOS
Dispositivo seleccionado: mps
```

Custom PyTorch Lightning Data Module for Dataset Handling

This code defines a custom data module `MyDataModule` using PyTorch Lightning.

Let's break down what each part does:

1. **Initialization**:
   - The `__init__` method initializes the data module with parameters such as `batch_size`, `transform`, `num_classes`, and `input_shape`.
   - It sets up a transformation pipeline using `torchvision.transforms.Compose`. This pipeline converts input data into tensors and then normalizes them.
2. **Data Preparation**:
   - The `prepare_data` method is responsible for any one-time preparation needed for the data.
   - In this case, it downloads the FashionMNIST dataset if not already downloaded and applies the transformation pipeline defined earlier.
3. **Data Setup**:
   - The `setup` method prepares datasets for training, validation, testing, or prediction based on the stage specified.
   - For the 'fit' stage (training), it splits the training dataset into training and validation sets using `random_split` from `torch.utils.data`.
   - For the 'test' stage, it prepares the test dataset.
   - For the 'predict' stage, it sets up a subset of the test dataset containing the first 5 elements, just for demonstration purposes.
4. **Data Loaders**:
   - The `train_dataloader`, `val_dataloader`, `test_dataloader`, and `predict_dataloader` methods return PyTorch `DataLoader` instances for training, validation, testing, and prediction, respectively.
   - These `DataLoader` objects are responsible for batching, shuffling, and loading the data.
5. **Instantiating the Data Module**:
   - An instance of `MyDataModule` is created with a specified `batch_size` of 32.
   - `prepare_data` is called to ensure data is prepared before setup.
   - `setup` is called without specifying a stage, which means it will prepare data for all stages (`'fit'`, `'test'`, and `'predict'`).

By utilizing this custom data module, you can easily organize your data loading pipeline, making it more modular and reusable within PyTorch Lightning training loops.

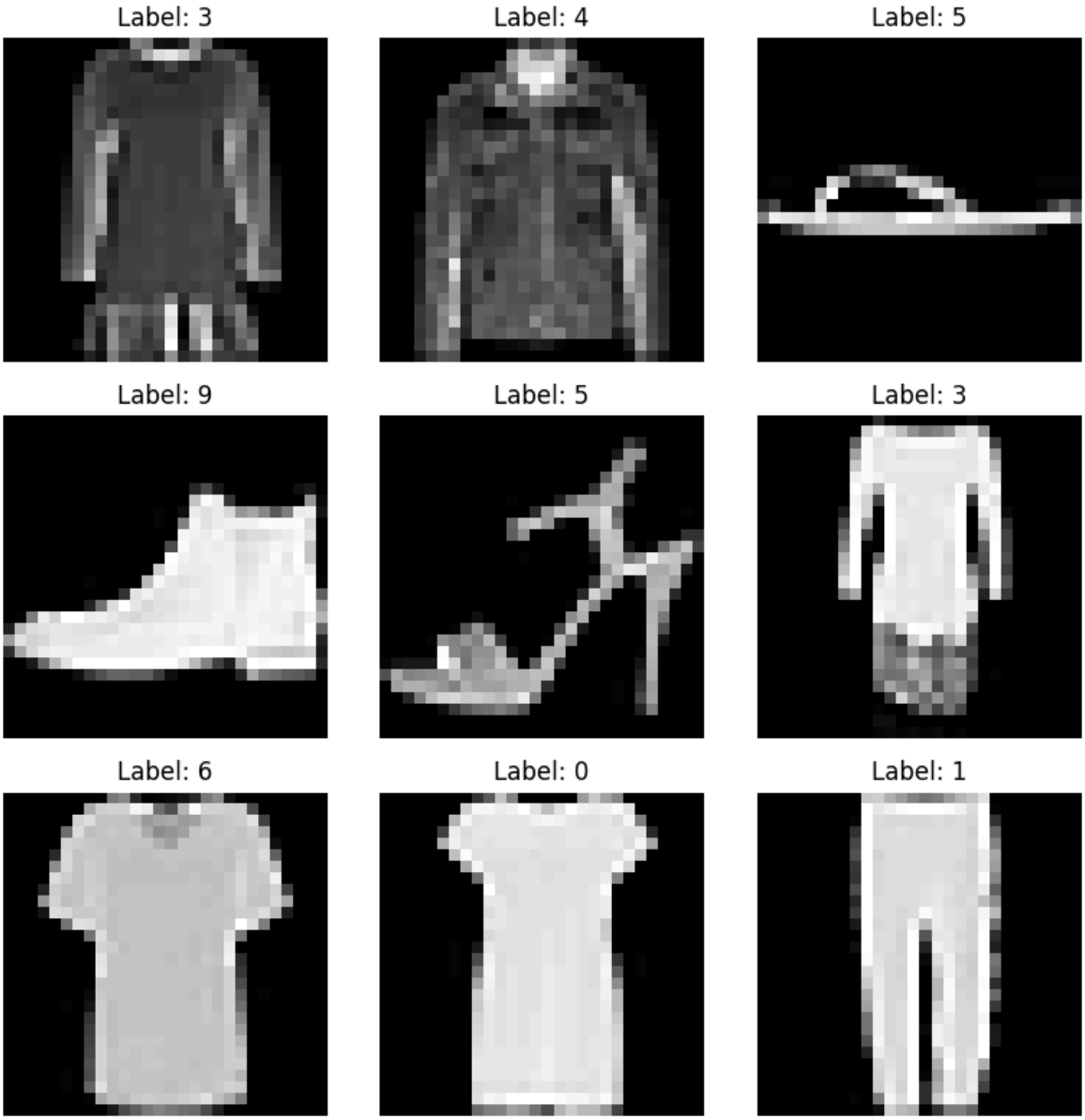
In [38]:
```
class MyDataModule(L.LightningDataModule): #LDM
    def __init__(self, batch_size=64):
        super().__init__()
        self.batch_size = batch_size
        self.transform = tv.transforms.Compose([
            tv.transforms.ToTensor()
           ,tv.transforms.Normalize((0.5,), (0.5,)) #normalize grayscale
        ]) #pipeline for image processing
```

```
        self.num_classes = 10
        self.input_shape = 28*28 #784 pixels
    def prepare_data(self):
        # This method can be used for one-time data preparation
        # For example, downloading datasets or initializing data sources

        tv.datasets.FashionMNIST(root="./data", train=True, download=True
        tv.datasets.FashionMNIST(root="./data", train=False, download=Tru

    def setup(self, stage=None):
        # Data setup for training, validation, and testing
        if stage == 'fit' or stage is None:

            #Load training set (RAM)
            train_dataset = tv.datasets.FashionMNIST(root="./data", train

            # Training validation split
            train_set_size = int(len(train_dataset) * 0.8) #80% train,
            valid_set_size = len(train_dataset) - train_set_size # 20%val
            seed = torch.Generator().manual_seed(42) # random seed
            train_set, valid_set = random_split(train_dataset, [train_set

            self.train_dataset = train_set
            self.val_dataset = valid_set

        if stage == 'test' or stage is None:
            test_dataset = tv.datasets.FashionMNIST(root="./data", train=
            self.test_dataset = test_dataset

        if stage == 'predict' or stage is None:
            #select the first 5 elements of a dataloader just for this ex
            test_dataset = tv.datasets.FashionMNIST(root="./data", train=
            self.predict_dataset = Subset(test_dataset, range(0, 5)) #jus

    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=self.batch_size,

    def val_dataloader(self):
        return DataLoader(self.val_dataset, batch_size=self.batch_size, s

    def test_dataloader(self):
        return DataLoader(self.test_dataset, batch_size=self.batch_size,

    def predict_dataloader(self):
        return DataLoader(self.predict_dataset, batch_size=self.batch_siz

dm = MyDataModule(batch_size=32)
# To access the x_dataloader we need to call prepare_data and setup.
dm.prepare_data()
#dm.setup(stage="fit")
dm.setup()
```

Visualizing the FMNIST dataset

In [39]:
```
import matplotlib.pyplot as plt
import random
```

```python
# Create a 3x3 subplot grid
fig, axes = plt.subplots(3, 3, figsize=(8, 8))

# Randomly select 9 images from the training dataset
random_indices = random.sample(range(len(dm.train_dataset)), 9)

# Loop through each subplot and plot an image with its label
for i, ax in enumerate(axes.flat):
    # Get image and label from the dataset
    img, labelidx = dm.train_dataset[random_indices[i]]

    # Move tensor to CPU if needed (for MPS/CUDA compatibility)
    if img.device.type != "cpu":
        img = img.cpu()

    img = img.view(28, 28, 1)

    # Plot the image
    ax.imshow(img.squeeze(), cmap='gray')

    # Print the label
    ax.set_title(f'Label: {labelidx}')

    # Remove axes
    ax.axis('off')

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the plot
plt.show()
```



Let's dive deeper into the modified code to visualize a 3x3 grid of random images from the FashionMNIST dataset:

1. **Importing Libraries**: We start by importing the necessary libraries, including `matplotlib.pyplot` for visualization and `random` for generating random indices.

2. **Creating Subplots**:

   - We use `plt.subplots(3, 3, figsize=(8, 8))` to create a 3x3 grid of subplots with a specific size (8x8 inches).
   - This function returns a figure object (`fig`) and an array of axes objects (`axes`), which represent the individual subplots.

3. **Randomly Selecting Images**:

   - We generate 9 random indices using `random.sample(range(len(dm.train_dataset)), 9)` .
   - These indices are used to select 9 random images from the training dataset.

- `len(dm.train_dataset)` returns the total number of images in the training dataset, and `random.sample()` selects 9 unique indices from this range.

4. **Plotting Images and Labels**:
   - We iterate over each subplot using `enumerate(axes.flat)`.
   - For each subplot, we retrieve the corresponding image and label from the dataset using the randomly selected indices.
   - The image is reshaped to match the dimensions expected by `imshow()` using `img.view(28, 28, 1)`.
   - We plot the image using `ax.imshow()` and specify `cmap='gray'` to display it in grayscale.
   - The label is printed as the title of the subplot using `ax.set_title(f'Label: {labelidx}')`.
5. **Removing Axes**:
   - We remove the axes from each subplot using `ax.axis('off')` to clean up the visualization.
6. **Adjusting Layout**:
   - We call `plt.tight_layout()` to adjust the layout of the subplots and prevent overlap between them.
7. **Displaying the Plot**:
   - Finally, we use `plt.show()` to display the plot containing the 3x3 grid of images.

Understanding a PyTorch Lightning Model

This code defines a PyTorch Lightning module named `Mymodel`. Let's break down what each part does:

1. **Initialization**:
   - The `__init__` method initializes the model with parameters such as `input_shape`, `num_classes`, and `hidden_units`.
   - It defines the layers of the model using a list `layers`. The layers include a flattening layer, a series of fully connected (linear) layers with batch normalization and ReLU activation, and a final linear layer for classification.
   - It sets up the loss function (`nn.CrossEntropyLoss()`) and accuracy metric (`Accuracy`).
2. **Forward Pass**:
   - The `forward` method defines the forward pass of the model, which applies the defined layers to the input data.
3. **Optimizer Configuration**:

- The `configure_optimizers` method specifies the optimizer used for training, in this case, Adam optimizer with a learning rate of 0.001.

4. **Training Step**:
   - The `training_step` method defines the operations performed during a single training step.
   - It computes the logits (raw predictions) using the forward pass, calculates the loss and accuracy, and logs them for monitoring.
5. **Validation and Test Steps**:
   - Similar to the training step, the `validation_step` and `test_step` methods define the operations during validation and testing.
   - They compute logits, calculate loss and accuracy, and log these metrics.
6. **Prediction Step**:
   - The `predict_step` method is optional and defined here for inference purposes. It performs the forward pass without computing loss or accuracy and returns the raw predictions.
7. **Instantiation**:
   - The model is instantiated with the specified `input_shape`, `num_classes`, and `hidden_units`.

Overall, this code provides a structured and modular way to define a neural network model for classification tasks using PyTorch Lightning, along with functionalities for training, validation, testing, and inference.

In [40]:
```
class Mymodel(L.LightningModule):
    def __init__(self, input_shape, num_classes, hidden_units, lambda_L2=
        super().__init__()

        #self.classifier = nn.Sequential(
        #    nn.Flatten(),
        #    nn.Linear(input_shape, 16),
        #    nn.ReLU(),
        #    nn.Linear(16, num_classes)
        #)
        self.hidden_units = hidden_units  # Store the list of hidden unit
        layers = []
        layers.append(nn.Flatten()) #input layer
        # Add hidden layers with ReLU activation
        for units in hidden_units: #hidden layers
            layers.append(nn.Linear(input_shape, units)) #logits
            layers.append(nn.BatchNorm1d(units))
            layers.append(nn.LeakyReLU(negative_slope=0.01))
            input_shape = units
        layers.append(nn.Linear(input_shape, num_classes))#output layer,
        self.classifier = nn.Sequential(*layers) # here the output are lo

        self.lambda_L2 = lambda_L2  # L2 regularization strength
        self.loss = nn.CrossEntropyLoss() #Multiclass
        self.accuracy = Accuracy(task="multiclass",  num_classes=num_clas
    #for inference
    def forward(self, x):
```

```
        return self.classifier(x) #logits

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.001, weight_decay

    def training_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs) #this calls forward. Do not use .forward
        loss = self.loss(logits, labels) # here is the softmax function!

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)

        self.log('train_loss', loss, on_step=True, on_epoch=True, logger=
        self.log('train_acc', acc, on_step=True, on_epoch=True, logger=Tr
        return loss

    def validation_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs) #this calls forward
        loss = self.loss(logits, labels)

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)
        self.log('val_loss', loss, on_step=True, on_epoch=True, logger=Tr
        self.log('val_acc', acc, on_step=True, on_epoch=True, logger=True
        return loss

    def test_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs) #this calls forward
        loss = self.loss(logits, labels)

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)
        self.log('test_loss', loss, on_step=True, on_epoch=True, logger=T
        self.log('test_acc', acc, on_step=True, on_epoch=True, logger=Tru
        return loss
    def predict_step(self, batch): #not necessary if dataloader with cust
        # this calls forward for the inputs only
        inputs, labels = batch
        return self(inputs) #return logits, this calls forward

hidden_units = [32, 64, 16]  # Example hidden units list MLP, 3 hidden la
lambda_L2 = 0.0 #regularizer
model = Mymodel(input_shape=dm.input_shape, #28*28
                num_classes=dm.num_classes, #10
                hidden_units=hidden_units,
                lambda_L2=lambda_L2)#regularization L2
```

Visualizing the model

A model visualization provides valuable insights into the structure and complexity of the neural network model, helping you understand its architecture and assess its capabilities.

The first way to visualize a model is by simply printing the `model` object. In PyTorch Lightning, the `model` is typically an instance of a LightningModule, which encapsulates the neural network model along with training and evaluation logic. By printing the `model`, you'll see its structure and possibly any attributes it contains.

In [41]:
```
print(model) #hidden_units = [32 64 16]
```

```
Mymodel(
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=32, bias=True)
    (2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (3): LeakyReLU(negative_slope=0.01)
    (4): Linear(in_features=32, out_features=64, bias=True)
    (5): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Linear(in_features=64, out_features=16, bias=True)
    (8): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (9): LeakyReLU(negative_slope=0.01)
    (10): Linear(in_features=16, out_features=10, bias=True)
  )
  (loss): CrossEntropyLoss()
  (accuracy): MulticlassAccuracy()
)
```

Another way to visualize the model is using the `torchsummary` library

In [42]:
```
%pip install torchinfo
```

```
Requirement already satisfied: torchinfo in /Library/Frameworks/Python.fra
mework/Versions/3.10/lib/python3.10/site-packages (1.8.0)

[notice] A new release of pip is available: 24.3.1 -> 25.3
[notice] To update, run: pip3 install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [43]:
```
from torchinfo import summary
# Use torchinfo to print the summary (compatible with MPS)
summary(model.to(device), input_size=(1, 1, 28, 28))  # input_size=(batch
```

Out[43]:
```
==========================================================================
==================
Layer (type:depth-idx)                   Output Shape              Param
#
==========================================================================
==================
Mymodel                                  [1, 10]                   --
├─Sequential: 1-1                        [1, 10]                   --
│    └─Flatten: 2-1                      [1, 784]                  --
│    └─Linear: 2-2                       [1, 32]                   25,12
0
│    └─BatchNorm1d: 2-3                  [1, 32]                   64
│    └─LeakyReLU: 2-4                    [1, 32]                   --
│    └─Linear: 2-5                       [1, 64]                   2,112
│    └─BatchNorm1d: 2-6                  [1, 64]                   128
│    └─LeakyReLU: 2-7                    [1, 64]                   --
│    └─Linear: 2-8                       [1, 16]                   1,040
│    └─BatchNorm1d: 2-9                  [1, 16]                   32
│    └─LeakyReLU: 2-10                   [1, 16]                   --
│    └─Linear: 2-11                      [1, 10]                   170
==========================================================================
==================
Total params: 28,666
Trainable params: 28,666
Non-trainable params: 0
Total mult-adds (M): 0.03
==========================================================================
==================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.11
Estimated Total Size (MB): 0.12
==========================================================================
==================
```

Let's break down the code:

```
from torchsummary import summary
```
This line imports the `summary` function from the `torchsummary` library. This function is used to print a summary of the PyTorch model's architecture, including information about the layers, output shapes, and the total number of parameters.

```
summary(model, input_size=(1, 28, 28))
```
Here, the `summary` function is called with two arguments:

- `model`: This is the PyTorch model. It represents the underlying neural network architecture.
- `input_size=(1, 28, 28)`: This specifies the input size expected by the model. The input size is defined as a tuple `(channels, height, width)`, where `1` indicates the number of channels (grayscale image), and `28` is the height and width of the input image from FMNIST dataset.

The `summary` function will then generate and print a summary of the model's architecture, which includes details such as:

- Layer names

- Output shapes of each layer
- Number of parameters (trainable and non-trainable)
- Total number of parameters in the model

Understanding Callbacks in PyTorch Lightning for Enhanced Training Control

A callback in programming is a function or a piece of code that is passed as an argument to another function or method. It allows you to customize or extend the behavior of the function or method without modifying its actual code. In the context of machine learning frameworks like PyTorch Lightning, a callback is a piece of code that is executed at specific points during the training process, such as after each epoch or after a certain metric threshold is reached.

Concretely, we use the following callbacks

In [44]:
```
# Initialize Callbacks
early_stop_callback = L.pytorch.callbacks.EarlyStopping(monitor="val_loss
                                                         mode="min",
                                                         patience=3)
checkpoint_callback = L.pytorch.callbacks.ModelCheckpoint(save_top_k=5,
                                                          save_last=True,
                                                          monitor="val_lo
                                                          mode='min')
```

Now, let's break down the code snippet:

1. **EarlyStopping Callback**:
    - `early_stop_callback` is initialized with an instance of the `EarlyStopping` callback.
    - This callback monitors the validation loss (`monitor="val_loss"`) and tries to minimize it (`mode="min"`).
    - It waits for `patience` number of epochs (in this case, 3 epochs) before considering stopping the training if the monitored quantity (validation loss) doesn't improve.
2. **ModelCheckpoint Callback**:
    - `checkpoint_callback` is initialized with an instance of the `ModelCheckpoint` callback.
    - This callback saves the model's checkpoints during training to disk.
    - It saves the top 5 models based on validation loss (`save_top_k=5`).
    - Additionally, it saves the last model (`save_last=True`) after training completes.
    - Similar to `EarlyStopping` , it monitors the validation loss (`monitor="val_loss"`) and tries to minimize it (`mode="min"`).

In summary, the provided code initializes two callbacks: `EarlyStopping` and `ModelCheckpoint`. These callbacks provide additional functionality to the training process, such as stopping training early if the model's performance stops improving (`EarlyStopping`) and saving model checkpoints for later use (`ModelCheckpoint`). These callbacks enhance the flexibility and functionality of the training loop, making it easier to manage and monitor the training process effectively.

Setting up PyTorch Lightning Trainer

This code snippet demonstrates how to instantiate a PyTorch Lightning Trainer object with GPU acceleration if a CUDA-enabled GPU device is available, and without GPU acceleration if not. This flexibility allows the code to seamlessly adapt to different hardware configurations, optimizing performance based on available resources.

In [45]:
```
trainer = None
print('*'*80)
print("You are using", device)
print('*'*80)
trainer = L.Trainer(max_epochs=50,
                    callbacks=[early_stop_callback, checkpoint_callback],
                    accelerator="gpu" if device == "cuda" else "cpu",
                    devices=1)#only works on script mode
```

```
GPU available: True (mps), used: False
TPU available: False, using: 0 TPU cores
********************************************************************************
******
You are using mps
********************************************************************************
******
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-pac
kages/lightning/pytorch/trainer/setup.py:175: GPU available but not used.
You can set it by doing `Trainer(accelerator='gpu')`.
```

Explanation:

1. **Checking Device Availability**:
   - The code checks if the `device` variable is set to `"cuda"`, indicating the presence of a CUDA-enabled GPU device.
   - If `device == "cuda"`, it signifies that the code will utilize GPU acceleration for training.
2. **Instantiating Trainer Object**:
   - If a GPU device is available (`device == "cuda"`), the Trainer object is instantiated with GPU acceleration enabled.
     - `max_epochs=50` specifies that the training will run for a maximum of 50 epochs.

- `callbacks=[early_stop_callback, checkpoint_callback]` specifies a list of callbacks to be used during training, including early stopping and model checkpointing.
- `accelerator="gpu"` indicates that GPU acceleration will be utilized.
- `devices=1` specifies the number of GPU devices to use. Here, it's set to 1, indicating that only one GPU will be used for training.

- If a GPU device is not available (`device != "cuda"`), the Trainer object is instantiated without GPU acceleration.
  - The parameters remain the same as above, but without specifying the `accelerator` or `devices` parameters. By default, if GPU acceleration is not specified, PyTorch Lightning will use CPU for training.

Let's train the model

This code snippet illustrates the usage of the `fit()` method to train a PyTorch Lightning model (`model`) using data provided by a LightningDataModule (`dm`). It specifies the path for saving the model checkpoints during training (`ckpt_path='last'`) to ensure that the trained model's progress is saved for future use or evaluation.

In [46]:
```
trainer.fit(model, dm,
            #checkpoint_callback.best_model_path  or 'best'.
            #use last if it is the first time you train.
            ckpt_path='last'
            )
```

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-pac
kages/lightning/pytorch/trainer/connectors/checkpoint_connector.py:190: .f
it(ckpt_path="last") is set, but there is no last checkpoint available. No
checkpoint will be loaded. HINT: Set `ModelCheckpoint(..., save_last=True)
`.
```

	Name	Type	Params	Mode	FLOPs
0	classifier	Sequential	28.7 K	train	0
1	loss	CrossEntropyLoss	0	train	0
2	accuracy	MulticlassAccuracy	0	train	0

Trainable params: 28.7 K
Non-trainable params: 0
Total params: 28.7 K
Total estimated model params size (MB): 0
Modules in train mode: 14
Modules in eval mode: 0
Total FLOPs: 0

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
install "ipywidgets" for Jupyter support
  warnings.warn('install "ipywidgets" for Jupyter support')
```

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'val_dataloader' does not have many wo
increasing the value of the `num_workers` argument` to `num_workers=
```

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'train_dataloader' does not have many w
increasing the value of the `num_workers` argument` to `num_workers=
```

Explanation:

- **trainer.fit() Method**:
  - `trainer.fit()` is a method provided by the PyTorch Lightning Trainer object, which is responsible for training the model.
  - It takes two main arguments:
    - The first argument (`model`) is the LightningModule instance that represents the neural network model to be trained.
    - The second argument (`dm`) is the LightningDataModule instance that provides the data loaders for training, validation, and testing.
  - In this case, `model` is the model object to be trained, and `dm` is the data module object that contains the data loaders necessary for training.
- **Checkpoint Path (`ckpt_path`)**:
  - The `ckpt_path` parameter specifies the path to load or save the model checkpoints during training.
  - In this code, `ckpt_path='last'` indicates that the Trainer will save the last model checkpoint after each epoch. This checkpoint will be used to resume training if interrupted or to evaluate the model after training.
  - Alternatively, the commented out line (`#checkpoint_callback.best_model_path or 'best'`) suggests an option to specify the path of the best model checkpoint based on a provided checkpoint callback (`checkpoint_callback`). It assumes that the `best_model_path` attribute of the `checkpoint_callback` contains the path to the best model checkpoint saved during training. If the attribute is not available or if the callback is not used, it defaults to `'best'`.

Evaluating the model on the validation set

This code snippet demonstrates how to use the `validate()` method of the PyTorch Lightning Trainer object to evaluate a trained model (`model`) on a validation dataset. By specifying the data loader(s) for validation and the path to the model checkpoint, the Trainer ensures that the evaluation is performed using the best model obtained during training.

In [47]:
```
trainer.validate(model, dataloaders=dm.val_dataloader(), ckpt_path='best'
```

```
Restoring states from the checkpoint path at /Users/efrainmanosalvas/Libra
ry/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/U
SFQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/ch
eckpoints/epoch=12-step=19500.ckpt
Loaded model weights from the checkpoint at /Users/efrainmanosalvas/Librar
y/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/US
FQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/che
ckpoints/epoch=12-step=19500.ckpt
```

Validate metric	DataLoader 0
val_acc_epoch	0.8883333206176758
val_loss_epoch	0.3090919852256775

Out[47]:
```
[{'val_loss_epoch': 0.3090919852256775, 'val_acc_epoch': 0.8883333206176
758}]
```

Explanation:

- **trainer.validate() Method**:
  - `trainer.validate()` is a method provided by the PyTorch Lightning Trainer object, which is used for model validation.
  - It takes several arguments:
    - The first argument (`model`) is the LightningModule instance representing the neural network model to be evaluated.
    - The `dataloaders` argument specifies the data loader(s) to be used for validation. In this case, it's provided as `dm.val_dataloader()`, which returns the validation data loader from the given LightningDataModule (`dm`).
    - The `ckpt_path` parameter specifies the path to load the model checkpoint before validation. In this code snippet, `ckpt_path='best'` indicates that the Trainer will load the best model checkpoint saved during training for validation. This ensures that the best-performing model is used for evaluation.

Visualize the training process

The `%tensorboard --logdir lightning_logs` command launches TensorBoard and instructs it to load the log files from the `lightning_logs` directory. TensorBoard then displays visualizations of the training progress, including metrics, loss curves, histograms of model parameters, and more, allowing you to analyze and monitor the performance of your PyTorch Lightning models interactively.

Explanation:

- The `%load_ext tensorboard` command is a Jupyter Notebook magic command used to load the TensorBoard extension within the Jupyter Notebook environment.
- **`%tensorboard` Magic Command**:
  - The `%tensorboard` command is a Jupyter Notebook magic command that allows you to launch TensorBoard directly within the Jupyter Notebook environment.
  - This command is typically used when you want to visualize logs and metrics generated during the training of machine learning models.
- **`--logdir` Argument**:
  - The `--logdir` argument specifies the directory where the log files are stored. In this case, it's set to `lightning_logs`.
  - PyTorch Lightning automatically logs various metrics, such as loss, accuracy, etc., during training, validation, and testing stages. These logs are saved in the `lightning_logs` directory by default.

In [ ]:
```
# Install tensorboard extension (use notebook magic so it installs into t
%pip install tensorboard
```

In [48]:
```
# Load the TensorBoard notebook extension
#%load_ext tensorboard
# No funciona con entornos de ejecución locales ()
%reload_ext tensorboard
```

In [49]:
```
%tensorboard --logdir "/content/lightning_logs"
```

```
Reusing TensorBoard on port 6006 (pid 57262), started 11:30:14 ago. (Use
'!kill 57262' to kill it.)
```

TensorBoard INACTIVE

No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Dec 10, 2025, 12:02:41 PM

Log directory: /content/lightning_logs

Test the model

This code snippet demonstrates how to use the `test()` method of the PyTorch Lightning Trainer object to evaluate a trained model (`model`) on a test dataset. By specifying the data loader(s) for testing and the path to the model checkpoint, the

Trainer ensures that the evaluation is performed using the best model obtained during training.

In [50]:
```
# test the model
#dm.setup(stage='test')
trainer.test(model, dataloaders=dm.test_dataloader(), ckpt_path='best')
```

```
Restoring states from the checkpoint path at /Users/efrainmanosalvas/Libra
ry/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/U
SFQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/ch
eckpoints/epoch=12-step=19500.ckpt
Loaded model weights from the checkpoint at /Users/efrainmanosalvas/Librar
y/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/US
FQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/che
ckpoints/epoch=12-step=19500.ckpt
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-pac
kages/lightning/pytorch/trainer/connectors/data_connector.py:434: The 'tes
t_dataloader' does not have many workers which may be a bottleneck. Consid
er increasing the value of the `num_workers` argument` to `num_workers=13`
in the `DataLoader` to improve performance.
```

Test metric	DataLoader 0
test_acc_epoch	0.8827000260353088
test_loss_epoch	0.3339481055736542

Out[50]:
```
[{'test_loss_epoch': 0.3339481055736542, 'test_acc_epoch': 0.88270002603
53088}]
```

Explanation:

- **`trainer.test()` Method**:
  - `trainer.test()` is a method provided by the PyTorch Lightning Trainer object, used for model evaluation on a test dataset.
  - It takes several arguments:
    - The first argument (`model`) is the LightningModule instance representing the neural network model to be evaluated.
    - The `dataloaders` argument specifies the data loader(s) to be used for testing. In this case, it's provided as `dm.test_dataloader()`, which returns the test data loader from the given LightningDataModule (`dm`).
    - The `ckpt_path` parameter specifies the path to load the model checkpoint before testing. In this code snippet, `ckpt_path='best'` indicates that the Trainer will load the best model checkpoint saved during training for testing. This ensures that the best-performing model is used for evaluation.

Making predictions: Inference stage

Inference, in the context of deep learning, refers to the process of using a trained neural network model to make predictions or draw conclusions from new, unseen data. During inference, input data is fed into the trained model, which processes the data through its layers to produce output predictions. These predictions could include class labels, regression values, or any other desired outcome depending on the task at hand.

In PyTorch Lightning, there are two distinct stages in which a trained model is evaluated: the **predict (inference)** stage and the **test** stage. Here's a clarification of the difference between these two stages:

1. **Predict (Inference) Stage**:
    - **Purpose**: The predict stage is primarily focused on making predictions or inferences on new, unseen data.
    - **Usage**: It is used when you want to apply the trained model to new data to obtain predictions or outputs.
    - **Input**: Typically, input data is provided to the model for inference, and the model's output predictions are obtained.
    - **Scenario**: This stage is commonly used in real-world applications where the trained model is deployed to make predictions on user input, images, text, or any other data.
    - **Flexibility**: PyTorch Lightning provides flexibility in conducting inference, allowing you to override the `predict_step` method in your LightningModule to customize the inference process.
2. **Test Stage**:
    - **Purpose**: The test stage is focused on evaluating the performance of the trained model on a separate test dataset.
    - **Usage**: It is used to assess the generalization ability and overall performance of the model on unseen data.
    - **Input**: Input data from the test dataset is provided to the model, and predictions are compared against ground-truth labels to compute evaluation metrics such as accuracy, precision, recall, etc.
    - **Scenario**: This stage is commonly used during model development and experimentation to validate the model's performance before deployment.
    - **Evaluation**: PyTorch Lightning provides convenience methods like `trainer.test()` to automate the evaluation process during the test stage. These methods handle the forward pass, compute metrics, and provide evaluation results.

In PyTorch Lightning, there are multiple approaches to perform inference, each with its own advantages and considerations. These approaches range from leveraging PyTorch Lightning's built-in capabilities, such as overriding the `predict_step` method, to more manual methods like directly calling the model's `forward` method. Each approach offers flexibility and trade-offs in terms of simplicity, efficiency, and customization.

Here, we present 3 options to perform predictions. **Option 1** is the preferred method as it leverages PyTorch Lightning's built-in capabilities and is more streamlined. **Option 1** and **Option 2** are commented.

Option 1:

This option demonstrates the preferred method for making inferences in PyTorch Lightning by overriding the `predict_step` method. Here's how it works:

- **Predict Step Override**:
  - The `predict_step` method is overridden within the LightningModule (see above the neural network model).
  - In this method, you call the `forward` method directly. This is the same method used during training and validation but tailored specifically for inference.
  - You don't need to call `eval()` or use `torch.no_grad()` explicitly, as PyTorch Lightning handles this internally.
- **Inference Process**:
  - The LightningDataModule is set up for inference using `dm.setup(stage='predict')`.
  - The Trainer's `predict` method is called, passing the model and the dataloader for inference.
  - Predicted logits are obtained, and softmax is applied to obtain probabilities.
  - Predicted labels are inferred from probabilities.
  - Ground-truth labels are extracted for comparison.
  - Finally, the probabilities are visualized.

Option 2 (commented in the code below):

This option demonstrates an alternative approach where you define a custom dataloader that only returns inputs for inference:

- **Custom Dataloader**:
  - You define a custom collate function (`custom_collate`) to extract only the inputs from the batch.
  - The dataloader is set up with this custom collate function.
- **Inference Process**:
  - The Trainer's `predict` method is called, passing only the dataloader (as it now only returns inputs).
  - Predicted logits are obtained.
- **No Predict Step**:
  - Options 2 does not require a `predict_step` method like option 1. Instead, it relies on custom dataloaders.

Option 3:

This option demonstrates directly calling the `forward` method for inference through `model(inputs)`:

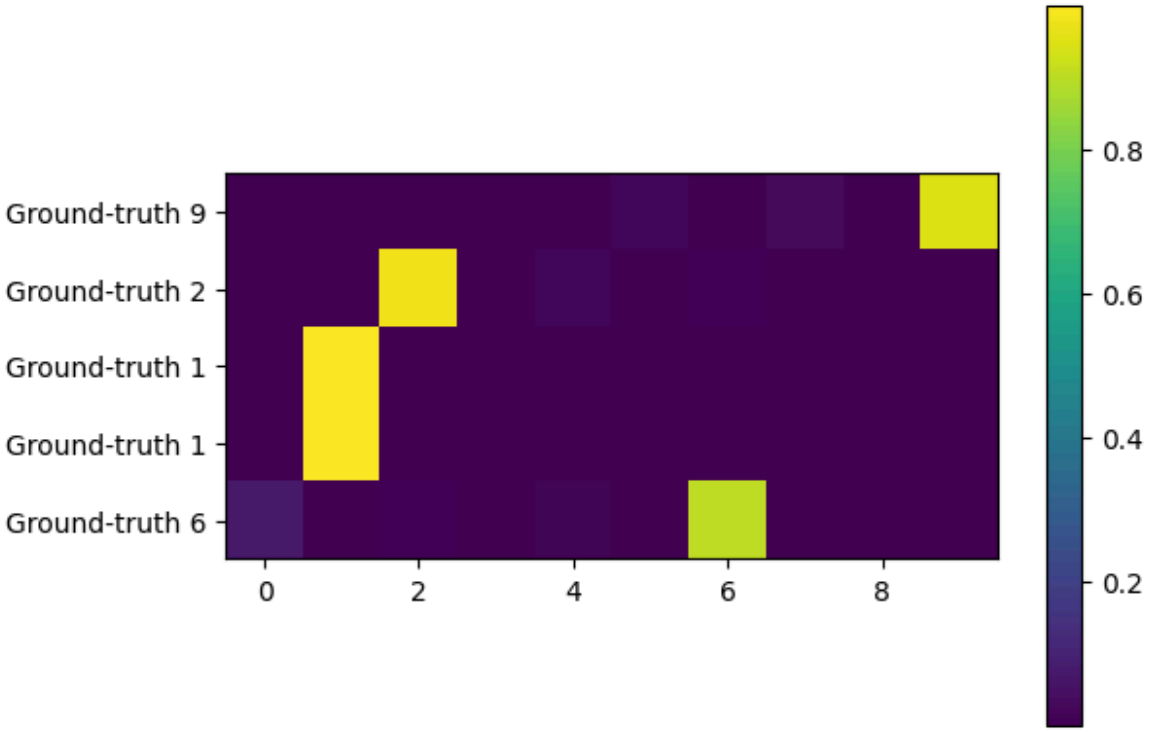
- **Using `forward`**:
  - You directly call the `forward` method of the model.
  - You must manually handle model evaluation (`eval()`) and gradient tracking suppression (`torch.no_grad()`).
- **Inference Process**:
  - The model is set to evaluation mode using `model.eval()`.
  - Inference is performed directly by calling the `forward` method on inputs.
  - Gradient tracking is disabled within the `torch.no_grad()` context.
  - Predicted logits are obtained.
- **No Predict Step**:
  - Options 3 does not require a `predict_step` method like option 1. Instead, it relies on direct `forward` calls for inference through `model(inputs)`. Note that by calling model(inputs), you're implicitly invoking the forward method of the model.

In [51]:
```
#OPTION 1 (PREFERRED). overrides predict_step that calls forward

#this calls the overridden predict_step which in turn calls forward.
# no need to call eval() or no_grad()

#dm.setup(stage='predict')
pred_logits = trainer.predict(model, dataloaders=dm.predict_dataloader(),
print("pred_logits[0].shape is", pred_logits[0].shape)

# Assuming 'pred_logits' is a list of tensors, e.g., [tensor1, tensor2, t
# Get the predicted probs for each example in the list using softmax
predicted_probs_list = [torch.softmax(logits, dim=1) for logits in pred_l
print("predicted_probs_list is", predicted_probs_list)
# 'predicted_labels_list' now contains the inferred class labels for each
predicted_labels_list = [torch.argmax(probs, dim=1) for probs in predicte
print("predicted_labels_list is", predicted_labels_list)
# Assuming your DataLoader returns (inputs, labels) for each batch
# Use list comprehension to extract labels for all examples in the DataLo
labels_list = [labels for _, labels in [batch for batch in dm.predict_dat
labels_list = labels_list[0].tolist() # because the previous step return
# 'labels_list' now contains the labels for all examples in the DataLoade

print("labels_list is", labels_list)

#visualize the probabilities
# Stack the list of tensors into a 2D tensor
stacked_tensor = torch.stack(predicted_probs_list, dim=0).view(5,10)
# Reshape the 2D tensor into a 2D numpy array
numpy_array = stacked_tensor.numpy()
# Visualize the 2D numpy array as an image
plt.imshow(numpy_array, cmap='viridis')  # You can choose a different col
```

```
# Convert the tensors in labels_list to strings
string_labels = [str(label) for label in labels_list]
prefixed_strings = ["Ground-truth " + string for string in string_labels]
# Set the y tick labels to the string values
plt.yticks(range(len(string_labels)), prefixed_strings)
plt.colorbar()
plt.show()


#OPTION 2 no predict_step needed. Need to define a dataloader that return
#def custom_collate(batch):
#    # Extract only the inputs and return them as a list
#    list_inputs = [item[0] for item in batch]
#    stacked_inputs = torch.stack(list_inputs, dim=0)
#    return stacked_inputs

#imgx, labelx = test_dataset[0]
#predict_dataset = data.TensorDataset(imgx)
#predict_dataloader = data.DataLoader(predict_dataset, batch_size=1, coll
#print(predict_dataloader.dataset[0])
#pred_logits = trainer.predict(dataloaders=predict_dataloader)
#print((pred_logits[0]))


# OPTION 3, calling directly forward.
#When using forward, you are responsible to call eval() and use the no_gr
#model.eval()
#with torch.no_grad():
#    inputs, lab = test_dataloader.dataset[0]
#    pred = model(inputs) #this calls forward
#    print(pred)
```

```
Restoring states from the checkpoint path at /Users/efrainmanosalvas/Libra
ry/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/U
SFQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/ch
eckpoints/epoch=12-step=19500.ckpt
Loaded model weights from the checkpoint at /Users/efrainmanosalvas/Librar
y/CloudStorage/GoogleDrive-angel.manosalvas@gmail.com/My Drive/PERSONAL/US
FQ/USFQ/APRENDIZAJE SUPERVISADO/SEMANA_3/TEMP/lightning_logs/version_1/che
ckpoints/epoch=12-step=19500.ckpt
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-pac
kages/lightning/pytorch/trainer/connectors/data_connector.py:434: The 'pre
dict_dataloader' does not have many workers which may be a bottleneck. Con
sider increasing the value of the `num_workers` argument` to `num_workers=
13` in the `DataLoader` to improve performance.
```

```
pred_logits[0].shape is torch.Size([5, 10])
predicted_probs_list is [tensor([[1.0080e-05, 5.4111e-06, 1.7194e-05, 6.66
19e-06, 1.0788e-06, 1.7351e-02,
         6.7804e-06, 2.9760e-02, 8.1410e-06, 9.5283e-01],
        [3.4650e-05, 3.5763e-06, 9.7802e-01, 1.5044e-05, 1.6084e-02, 6.354
2e-08,
         5.8398e-03, 1.8970e-07, 3.5588e-08, 2.9185e-07],
        [1.3817e-06, 9.9997e-01, 8.3112e-08, 2.3345e-05, 8.1814e-07, 2.939
1e-08,
         5.5988e-06, 5.2450e-08, 1.7515e-07, 1.9557e-08],
        [7.5857e-07, 9.9979e-01, 2.5571e-07, 1.9072e-04, 2.5467e-06, 6.520
5e-07,
         1.6635e-05, 8.1458e-08, 2.5479e-08, 1.8987e-08],
        [7.0102e-02, 7.5535e-05, 1.0615e-02, 1.4290e-03, 1.1727e-02, 1.032
4e-04,
         9.0547e-01, 6.1244e-06, 4.6573e-04, 7.1418e-06]])]
predicted_labels_list is [tensor([9, 2, 1, 1, 6])]
labels_list is [9, 2, 1, 1, 6]
```

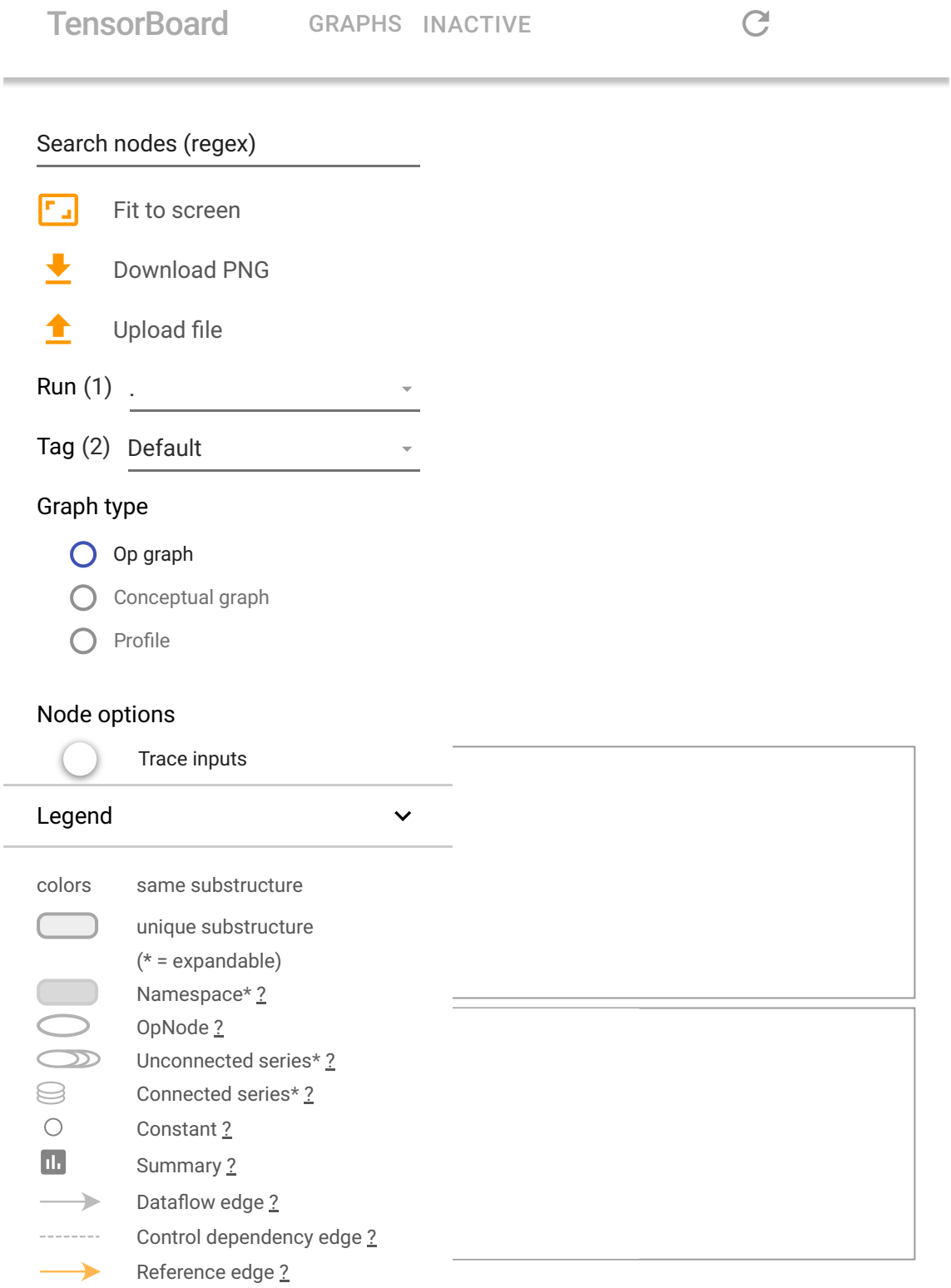


Visualizing the model with Tensorboard

In [52]:
```
from torch.utils.tensorboard import SummaryWriter
X_probe = torch.randn(5,28,28) #just a (batch,width, height) probe input
writer = SummaryWriter("torchlogs/")
writer.add_graph(model, X_probe)
writer.close()
```

In [53]:
```
%tensorboard --logdir torchlogs # requieres to previously execute in juyp
```

```
Reusing TensorBoard on port 6007 (pid 57439), started 11:30:01 ago. (Use
'!kill 57439' to kill it.)
```

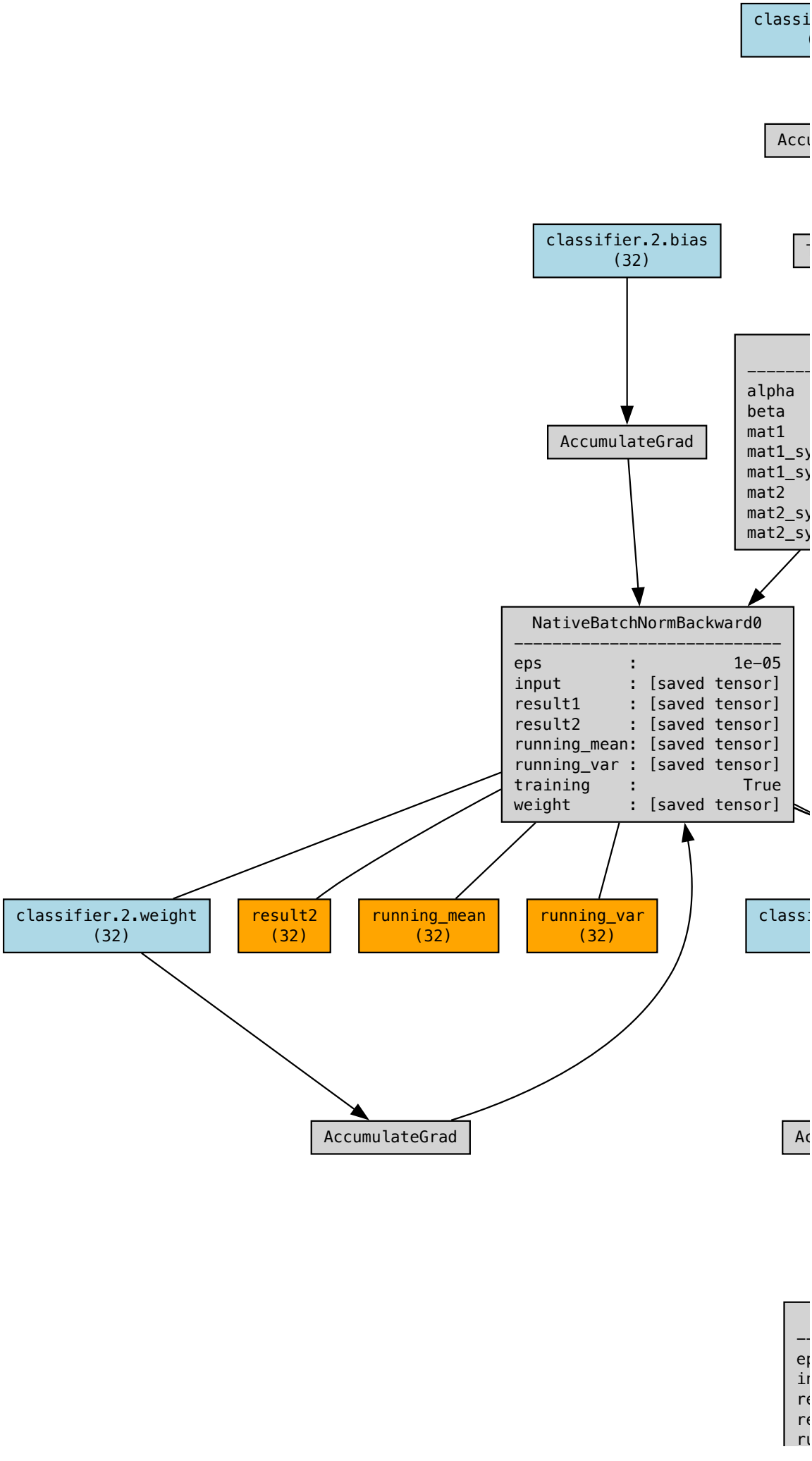


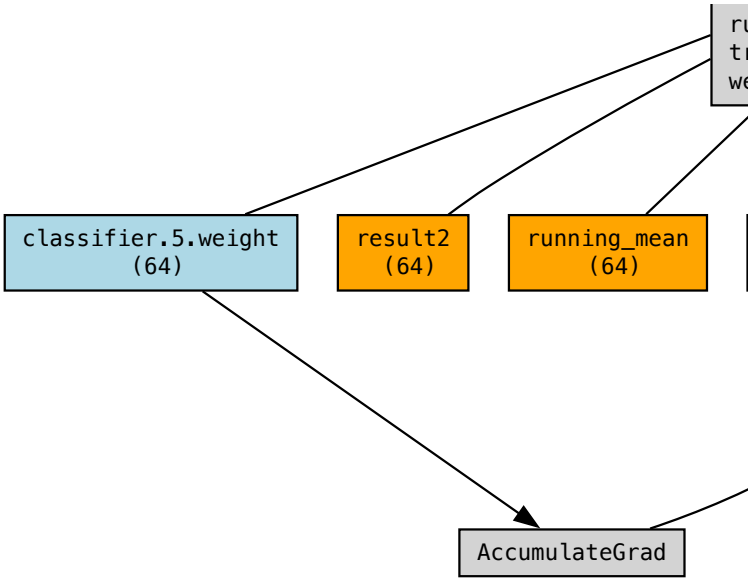
In [54]:
```
#ADVANCED VISUALIZATION WITH TORCHVIZ
%pip install torchviz
```

```
Requirement already satisfied: torchviz in /Library/Frameworks/Python.fram
ework/Versions/3.10/lib/python3.10/site-packages (0.0.3)
Requirement already satisfied: torch in /Library/Frameworks/Python.framewo
rk/Versions/3.10/lib/python3.10/site-packages (from torchviz) (2.9.1)
Requirement already satisfied: graphviz in /Library/Frameworks/Python.fram
ework/Versions/3.10/lib/python3.10/site-packages (from torchviz) (0.21)
Requirement already satisfied: filelock in /Library/Frameworks/Python.fram
ework/Versions/3.10/lib/python3.10/site-packages (from torch->torchviz)
(3.19.1)
Requirement already satisfied: typing-extensions>=4.10.0 in /Library/Frame
works/Python.framework/Versions/3.10/lib/python3.10/site-packages (from to
rch->torchviz) (4.15.0)
Requirement already satisfied: sympy>=1.13.3 in /Library/Frameworks/Pytho
n.framework/Versions/3.10/lib/python3.10/site-packages (from torch->torchv
iz) (1.14.0)
Requirement already satisfied: networkx>=2.5.1 in /Library/Frameworks/Pyth
on.framework/Versions/3.10/lib/python3.10/site-packages (from torch->torch
viz) (3.4.2)
Requirement already satisfied: jinja2 in /Library/Frameworks/Python.framew
ork/Versions/3.10/lib/python3.10/site-packages (from torch->torchviz) (3.
1.6)
Requirement already satisfied: fsspec>=0.8.5 in /Library/Frameworks/Pytho
n.framework/Versions/3.10/lib/python3.10/site-packages (from torch->torchv
iz) (2025.9.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /Library/Frameworks/P
ython.framework/Versions/3.10/lib/python3.10/site-packages (from sympy>=1.
13.3->torch->torchviz) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /Library/Frameworks/Pyth
on.framework/Versions/3.10/lib/python3.10/site-packages (from jinja2->torc
h->torchviz) (3.0.3)

[notice] A new release of pip is available: 24.3.1 -> 25.3
[notice] To update, run: pip3 install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [55]:
```
#ADVANCED VISUALIZATION WITH TORCHVIZ
from torchviz import make_dot
X_probe = torch.randn(5,28,28) #just a (batch,width, height) probe input
y = model(X_probe)
make_dot(y.mean(), params=dict(model.named_parameters()), show_attrs=True
```

Out[55]:


classifier.2.bias
(32)
AccumulateGrad
NativeBatchNormBackward0
eps : 1e-05
input : [saved tensor]
result1 : [saved tensor]
result2 : [saved tensor]
running_mean: [saved tensor]
running_var : [saved tensor]
training : True
weight : [saved tensor]
classifier.2.weight
(32)
result2
(32)
running_mean
(32)
running_var
(32)
AccumulateGrad


classifier.5.weight
(64)
result2
(64)
running_mean
(64)
AccumulateGrad

✅ Activity: K-Fold Cross-Validation

Modify the code to incorporate K-Fold Cross-Validation (CV) to optimize the number of neurons in each hidden layer. Keep the number of hidden layers fixed at 3, the L2 regularization parameter fixed at 1e-4, and vary the number of neurons in each hidden layer using the following values: 16, 64, and 256. For this activity, utilize `sklearn.model_selection.StratifiedKFold` with a random seed of 42. Test your code with `k=5` and complete the table below with the accuracy for each run. Additionally, calculate the p-value using the model with 16 neurons as the pivot. Analyze which of the three models is the best.

	16 neurons per Hidden Layer	64 neurons per Hidden Layer	256 neurons per Hidden Layer
Fold 1			
Fold 2			
Fold 3			
Fold 4			
Fold 5			
Mean (std) -->			
p-value -->	-----		

In [58]:
```
from sklearn.model_selection import StratifiedKFold

from scipy import stats

import numpy as np

import pandas as pd


# Parameters

k_folds = 5

neuron_configs = [16, 64, 256]  # Number of neurons per hidden layer

num_hidden_layers = 3

lambda_L2 = 1e-4

random_seed = 42


# Set random seeds for reproducibility

torch.manual_seed(random_seed)

np.random.seed(random_seed)

L.seed_everything(random_seed)
```

```
# Load full training dataset for K-Fold
full_train_dataset = tv.datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=tv.transforms.ToTensor()
)

# Extract data and labels for StratifiedKFold
X_full = full_train_dataset.data.numpy()
y_full = full_train_dataset.targets.numpy()

# Flatten images for easier handling
X_full_flat = X_full.reshape(X_full.shape[0], -1)

# Initialize StratifiedKFold
skf = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=random

# Dictionary to store results
results = {16: [], 64: [], 256: []}

print("=" * 80)
print("K-FOLD CROSS-VALIDATION FOR FMNIST")
print("=" * 80)
print(f"Configuration:")
print(f"  - Number of folds: {k_folds}")
print(f"  - Number of hidden layers: {num_hidden_layers}")
print(f"  - L2 regularization: {lambda_L2}")
print(f"  - Neuron configurations: {neuron_configs}")
print(f"  - Random seed: {random_seed}")
```

```
print("=" * 80)


# Iterate over each neuron configuration
for neurons_per_layer in neuron_configs:
    print(f"\n{'='*80}")
    print(f"TESTING MODEL WITH {neurons_per_layer} NEURONS PER LAYER")
    print(f"{'='*80}")

    fold_accuracies = []

    # Iterate over each fold
    for fold, (train_idx, val_idx) in enumerate(skf.split(X_full_flat, y_
        print(f"\n--- Fold {fold + 1}/{k_folds} ---")

        # Create train and validation subsets
        train_subset = torch.utils.data.Subset(full_train_dataset, train_
        val_subset = torch.utils.data.Subset(full_train_dataset, val_idx)

        # Create data loaders
        train_loader = torch.utils.data.DataLoader(
            train_subset,
            batch_size=64,
            shuffle=True,
            num_workers=0
        )

        val_loader = torch.utils.data.DataLoader(
            val_subset,
            batch_size=64,
            shuffle=False,
```

```
        num_workers=0
    )


    # Create model with specified number of neurons in each layer
    hidden_units = [neurons_per_layer] * num_hidden_layers
    model = Mymodel(
        input_shape=28*28,
        num_classes=10,
        hidden_units=hidden_units,
        lambda_L2=lambda_L2
    )


    # Create trainer
    trainer = L.Trainer(
        max_epochs=20,
        logger=False,
        enable_checkpointing=False,
        enable_progress_bar=True,
        enable_model_summary=False,
        accelerator='auto'
    )


    # Train the model
    trainer.fit(model, train_loader, val_loader)


    # Validate and get accuracy
    val_results = trainer.validate(model, val_loader, verbose=False)

    # Extract validation accuracy (handle different key names)
    if 'val_acc_epoch' in val_results[0]:
        val_accuracy = val_results[0]['val_acc_epoch']
    elif 'val_acc' in val_results[0]:
        val_accuracy = val_results[0]['val_acc']
```

```
        else:
            print(f"Available keys: {list(val_results[0].keys())}")
            raise KeyError("Could not find validation accuracy in results



        fold_accuracies.append(val_accuracy)

        print(f"Fold {fold + 1} Validation Accuracy: {val_accuracy:.4f}")



    # Store results for this configuration

    results[neurons_per_layer] = fold_accuracies

    mean_acc = np.mean(fold_accuracies)

    std_acc = np.std(fold_accuracies)



    print(f"\n{neurons_per_layer} neurons - Mean Accuracy: {mean_acc:.4f}



# ============================================================================

# ============================================================================
# RESULTS TABLE
# ============================================================================
print("\n" + "="*80)
print("RESULTS SUMMARY TABLE - K-FOLD CROSS-VALIDATION")
print("="*80)

# Create detailed results table
print("\n" + "-"*80)
print(f"{'Model':<20} | {'Fold 1':>10} | {'Fold 2':>10} | {'Fold 3':>10}
print("-"*80)

for neurons in neuron_configs:
    fold_results = [f"{acc:.4f}" for acc in results[neurons]]
    print(f"{neurons} neurons {'':<11} | {fold_results[0]:>10} | {fold_re

print("-"*80)

# Statistics table
print("\n" + "-"*80)
print(f"{'Model':<20} | {'Mean':>12} | {'Std Dev':>12} | {'Min':>12} | {'
print("-"*80)

for neurons in neuron_configs:
    fold_accs = results[neurons]
    mean_val = np.mean(fold_accs)
    std_val = np.std(fold_accs)
    min_val = np.min(fold_accs)
    max_val = np.max(fold_accs)
    print(f"{neurons} neurons {'':<11} | {mean_val:>12.4f} | {std_val:>12

print("-"*80)
```

```
# DataFrame for easy copy-paste
print("\n" + "="*80)
print("PANDAS DATAFRAME FORMAT")
print("="*80 + "\n")
df_results = pd.DataFrame(results)
df_results.index = [f'Fold {i+1}' for i in range(k_folds)]
df_results.loc['Mean'] = df_results.mean()
df_results.loc['Std'] = df_results.std()
print(df_results.to_string())

# STATISTICAL ANALYSIS (P-VALUES)

# ============================================================================

print("\n" + "="*80)

print("STATISTICAL ANALYSIS - P-VALUES (using 16 neurons as pivot)")

print("="*80)



# Using 16 neurons as the pivot/baseline

baseline = results[16]



print(f"\nPaired t-tests (comparing each model against {16}-neuron model)

print("-" * 60)



for neurons in neuron_configs:

    if neurons != 16:

        # Paired t-test

        t_stat, p_value = stats.ttest_rel(baseline, results[neurons])



        print(f"\n{16} neurons vs {neurons} neurons:")

        print(f"  t-statistic: {t_stat:.4f}")

        print(f"  p-value: {p_value:.4f}")



        if p_value < 0.05:

            mean_16 = np.mean(baseline)

            mean_comp = np.mean(results[neurons])

            if mean_comp > mean_16:
```

```
                print(f"  ✓ {neurons}-neuron model is SIGNIFICANTLY BETTE
            else:
                print(f"  ✗ {neurons}-neuron model is SIGNIFICANTLY WORSE
        else:
            print(f"  ≈ No significant difference (p >= 0.05)")


# ================================================================================
# BEST MODEL ANALYSIS
# ================================================================================
print("\n" + "="*80)
print("BEST MODEL ANALYSIS")
print("="*80)

mean_accuracies = {neurons: np.mean(results[neurons]) for neurons in neur
std_accuracies = {neurons: np.std(results[neurons]) for neurons in neuron

best_neurons = max(mean_accuracies, key=mean_accuracies.get)
best_mean = mean_accuracies[best_neurons]
best_std = std_accuracies[best_neurons]

print(f"\nRanking by Mean Accuracy:")
print("-" * 60)
for rank, (neurons, mean_acc) in enumerate(sorted(mean_accuracies.items()
    std_acc = std_accuracies[neurons]
    marker = "🏆" if rank == 1 else f"{rank}."
    print(f"{marker} {neurons} neurons: {mean_acc:.4f} ± {std_acc:.4f}")

print(f"\n{'='*80}")
print(f"CONCLUSION:")
print(f"{'='*80}")
```

```
print(f"The best model uses {best_neurons} neurons per hidden layer")

print(f"  - Mean Accuracy: {best_mean:.4f}")

print(f"  - Standard Deviation: {best_std:.4f}")

print(f"  - Configuration: {num_hidden_layers} hidden layers, L2={lambda_

print(f"{'='*80}")
```

```
Seed set to 42
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
install "ipywidgets" for Jupyter support
  warnings.warn('install "ipywidgets" for Jupyter support')
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
s/data_connector.py:434: The 'val_dataloader' does not have many wo
increasing the value of the `num_workers` argument` to `num_workers=
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
2: You called `self.log('val_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
2: You called `self.log('val_acc', ..., logger=True)` but have no lo
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
s/data_connector.py:434: The 'train_dataloader' does not have many w
increasing the value of the `num_workers` argument` to `num_workers=
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
2: You called `self.log('train_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s:
2: You called `self.log('train_acc', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
```

```
================================================================================
======
K-FOLD CROSS-VALIDATION FOR FMNIST
================================================================================
======
Configuration:
  - Number of folds: 5
  - Number of hidden layers: 3
  - L2 regularization: 0.0001
  - Neuron configurations: [16, 64, 256]
  - Random seed: 42
================================================================================
======

================================================================================
======
TESTING MODEL WITH 16 NEURONS PER LAYER
================================================================================
======

--- Fold 1/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.
```

```
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 1 Validation Accuracy: 0.8794

--- Fold 2/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 2 Validation Accuracy: 0.8697

--- Fold 3/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 3 Validation Accuracy: 0.8568

--- Fold 4/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 4 Validation Accuracy: 0.8738

--- Fold 5/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
install "ipywidgets" for Jupyter support
  warnings.warn('install "ipywidgets" for Jupyter support')
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'val_dataloader' does not have many wo
increasing the value of the `num_workers` argument` to `num_workers=
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('val_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('val_acc', ..., logger=True)` but have no lo
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'train_dataloader' does not have many w
increasing the value of the `num_workers` argument` to `num_workers=
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('train_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('train_acc', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
```

```
Fold 5 Validation Accuracy: 0.8745

16 neurons - Mean Accuracy: 0.8709 ± 0.0077

================================================================================
======
TESTING MODEL WITH 64 NEURONS PER LAYER
================================================================================
======

--- Fold 1/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 1 Validation Accuracy: 0.8932

--- Fold 2/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 2 Validation Accuracy: 0.8904

--- Fold 3/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 3 Validation Accuracy: 0.8953

--- Fold 4/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
Fold 4 Validation Accuracy: 0.8853

--- Fold 5/5 ---
`Trainer.fit` stopped: `max_epochs=20` reached.

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
install "ipywidgets" for Jupyter support
  warnings.warn('install "ipywidgets" for Jupyter support')
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'val_dataloader' does not have many wo
increasing the value of the `num_workers` argument` to `num_workers=
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('val_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('val_acc', ..., logger=True)` but have no l
`Trainer(logger=ALogger(...))`
```

```
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
s/data_connector.py:434: The 'train_dataloader' does not have many w
increasing the value of the `num_workers` argument` to `num_workers=

/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('train_loss', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`

/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/s
2: You called `self.log('train_acc', ..., logger=True)` but have no
`Trainer(logger=ALogger(...))`

 Fold 5 Validation Accuracy: 0.8823

 64 neurons - Mean Accuracy: 0.8893 ± 0.0049

 ================================================================================
 ======
 TESTING MODEL WITH 256 NEURONS PER LAYER
 ================================================================================
 ======

 --- Fold 1/5 ---
 `Trainer.fit` stopped: `max_epochs=20` reached.

 GPU available: True (mps), used: True
 TPU available: False, using: 0 TPU cores
 Fold 1 Validation Accuracy: 0.8882

 --- Fold 2/5 ---
 `Trainer.fit` stopped: `max_epochs=20` reached.

 GPU available: True (mps), used: True
 TPU available: False, using: 0 TPU cores
 Fold 2 Validation Accuracy: 0.8836

 --- Fold 3/5 ---
 `Trainer.fit` stopped: `max_epochs=20` reached.

 GPU available: True (mps), used: True
 TPU available: False, using: 0 TPU cores
 Fold 3 Validation Accuracy: 0.8811

 --- Fold 4/5 ---
 `Trainer.fit` stopped: `max_epochs=20` reached.

 GPU available: True (mps), used: True
 TPU available: False, using: 0 TPU cores
 Fold 4 Validation Accuracy: 0.8831

 --- Fold 5/5 ---
 `Trainer.fit` stopped: `max_epochs=20` reached.
```

```
Fold 5 Validation Accuracy: 0.8905

256 neurons - Mean Accuracy: 0.8853 ± 0.0035

================================================================================
======
RESULTS SUMMARY TABLE
================================================================================
======
             16        64        256
Fold 1  0.879417  0.893250  0.888167
Fold 2  0.869667  0.890417  0.883583
Fold 3  0.856833  0.895333  0.881083
Fold 4  0.873833  0.885333  0.883083
Fold 5  0.874500  0.882250  0.890500
Mean    0.870850  0.889317  0.885283
Std     0.007661  0.004872  0.003490

================================================================================
======
STATISTICAL ANALYSIS - P-VALUES (using 16 neurons as pivot)
================================================================================
======

Paired t-tests (comparing each model against 16-neuron model):
------------------------------------------------------------

16 neurons vs 64 neurons:
  t-statistic: -3.3960
  p-value: 0.0274
  ✓ 64-neuron model is SIGNIFICANTLY BETTER (p < 0.05)

16 neurons vs 256 neurons:
  t-statistic: -5.1310
  p-value: 0.0068
  ✓ 256-neuron model is SIGNIFICANTLY BETTER (p < 0.05)

================================================================================
======
BEST MODEL ANALYSIS
================================================================================
======

Ranking by Mean Accuracy:
------------------------------------------------------------
🏆 64 neurons: 0.8893 ± 0.0049
2. 256 neurons: 0.8853 ± 0.0035
3. 16 neurons: 0.8709 ± 0.0077

================================================================================
======
CONCLUSION:
================================================================================
======
The best model uses 64 neurons per hidden layer
  - Mean Accuracy: 0.8893
  - Standard Deviation: 0.0049
  - Configuration: 3 hidden layers, L2=0.0001
================================================================================
======
```

✅ Activity: Implementing L1 Regularization

Modify the model to accept a new hyperparameter `Lambda_L1` to implement L1 regularization. Note that `torch.optim.Adam` does not implement L1 regularization. Tip: Modify `training_step` to include the L1 norm in the loss function calculation. You don't need to do k-fold cross-validation. Just run your implementation once to check if it works correctly.

References

[1] https://lightning.ai/

[2] https://www.appsilon.com/post/visualize-pytorch-neural-networks

In [60]:
```
# Modified Mymodel class with L1 Regularization support
class MymodelWithL1(L.LightningModule):
    def __init__(self, input_shape, num_classes, hidden_units, lambda_L2=
        super().__init__()

        self.hidden_units = hidden_units  # Store the list of hidden unit
        layers = []
        layers.append(nn.Flatten())  # input layer

        # Add hidden layers with ReLU activation
        for units in hidden_units:  # hidden layers
            layers.append(nn.Linear(input_shape, units))  # logits
            layers.append(nn.BatchNorm1d(units))
            layers.append(nn.LeakyReLU(negative_slope=0.01))
            input_shape = units

        layers.append(nn.Linear(input_shape, num_classes))  # output laye
        self.classifier = nn.Sequential(*layers)  # here the output are l

        self.lambda_L2 = lambda_L2  # L2 regularization strength
        self.lambda_L1 = lambda_L1  # L1 regularization strength
        self.loss = nn.CrossEntropyLoss()  # Multiclass
        self.accuracy = Accuracy(task="multiclass", num_classes=num_class

    # for inference
    def forward(self, x):
        return self.classifier(x)  # logits

    def configure_optimizers(self):
        # Adam optimizer with L2 regularization (weight_decay)
        return torch.optim.Adam(self.parameters(), lr=0.001, weight_decay

    def training_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs)  # this calls forward. Do not use .forward

        # Calculate base loss
        loss = self.loss(logits, labels)  # here is the softmax function!
```

```
        # Add L1 regularization to the loss
        if self.lambda_L1 > 0:
            l1_norm = sum(p.abs().sum() for p in self.parameters())
            loss = loss + self.lambda_L1 * l1_norm

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)

        self.log('train_loss', loss, on_step=True, on_epoch=True, logger=
        self.log('train_acc', acc, on_step=True, on_epoch=True, logger=Tr
        return loss

    def validation_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs)  # this calls forward
        loss = self.loss(logits, labels)

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)

        self.log('val_loss', loss, on_step=True, on_epoch=True, logger=Tr
        self.log('val_acc', acc, on_step=True, on_epoch=True, logger=True
        return loss

    def test_step(self, batch, batch_idx):
        inputs, labels = batch
        logits = self(inputs)
        loss = self.loss(logits, labels)

        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, labels)

        self.log('test_loss', loss, on_step=True, on_epoch=True, logger=T
        self.log('test_acc', acc, on_step=True, on_epoch=True, logger=Tru
        return loss


# ================================================================================
# TEST THE L1 REGULARIZATION IMPLEMENTATION
# ================================================================================
print("="*80)
print("TESTING L1 REGULARIZATION IMPLEMENTATION")
print("="*80)

# Create a simple test with the data module
dm = MyDataModule(batch_size=64)
dm.setup(stage='fit')

# Test with L1 regularization
lambda_L1_test = 1e-5
lambda_L2_test = 1e-4

print(f"\nConfiguration:")
print(f"  - Hidden layers: 3")
print(f"  - Neurons per layer: [128, 64, 32]")
print(f"  - L1 regularization (lambda_L1): {lambda_L1_test}")
print(f"  - L2 regularization (lambda_L2): {lambda_L2_test}")
print(f"  - Epochs: 10")

# Create model with L1 regularization
```

```
model_with_l1 = MymodelWithL1(
    input_shape=28*28,
    num_classes=10,
    hidden_units=[128, 64, 32],
    lambda_L2=lambda_L2_test,
    lambda_L1=lambda_L1_test
)

# Create trainer
trainer = L.Trainer(
    max_epochs=10,
    logger=True,
    enable_checkpointing=False,
    enable_progress_bar=True,
    accelerator='auto'
)

print("\n" + "="*80)
print("Training model with L1 regularization...")
print("="*80)

# Train the model
trainer.fit(model_with_l1, dm)

# Test the model
print("\n" + "="*80)
print("Testing model...")
print("="*80)
test_results = trainer.test(model_with_l1, dm)

print("\n" + "="*80)
print("RESULTS")
print("="*80)
print(f"Test Accuracy: {test_results[0]['test_acc_epoch']:.4f}")
print(f"Test Loss: {test_results[0]['test_loss_epoch']:.4f}")

print("\n" + "="*80)
print("L1 REGULARIZATION SUCCESSFULLY IMPLEMENTED!")
print("="*80)
print("\nKey modifications:")
print("  1. Added 'lambda_L1' parameter to __init__")
print("  2. Modified training_step to calculate L1 norm of all parameters
print("  3. Added L1 penalty to the loss: loss = loss + lambda_L1 * l1_no
print("  4. L1 norm calculated as: sum(|parameter|) for all parameters")
print("="*80)
```

```
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
```

```
================================================================================
======
TESTING L1 REGULARIZATION IMPLEMENTATION
================================================================================
======

Configuration:
  - Hidden layers: 3
  - Neurons per layer: [128, 64, 32]
  - L1 regularization (lambda_L1): 1e-05
  - L2 regularization (lambda_L2): 0.0001
  - Epochs: 10

================================================================================
======
Training model with L1 regularization...
================================================================================
======
```

	Name	Type	Params	Mode	FLOPs
0	classifier	Sequential	111 K	train	0
1	loss	CrossEntropyLoss	0	train	0
2	accuracy	MulticlassAccuracy	0	train	0

Trainable params: 111 K
Non-trainable params: 0
Total params: 111 K
Total estimated model params size (MB): 0
Modules in train mode: 14
Modules in eval mode: 0
Total FLOPs: 0

```
`Trainer.fit` stopped: `max_epochs=10` reached.
```

```
================================================================================
======
Testing model...
================================================================================
======
```

Test metric	DataLoader 0
test_acc_epoch	0.8798999786376953
test_loss_epoch	0.34048524498939514

```
================================================================================
======
RESULTS
================================================================================
======
Test Accuracy: 0.8799
Test Loss: 0.3405

================================================================================
======
L1 REGULARIZATION SUCCESSFULLY IMPLEMENTED!
================================================================================
======

Key modifications:
  1. Added 'lambda_L1' parameter to __init__
  2. Modified training_step to calculate L1 norm of all parameters
  3. Added L1 penalty to the loss: loss = loss + lambda_L1 * l1_norm
  4. L1 norm calculated as: sum(|parameter|) for all parameters
================================================================================
======
```