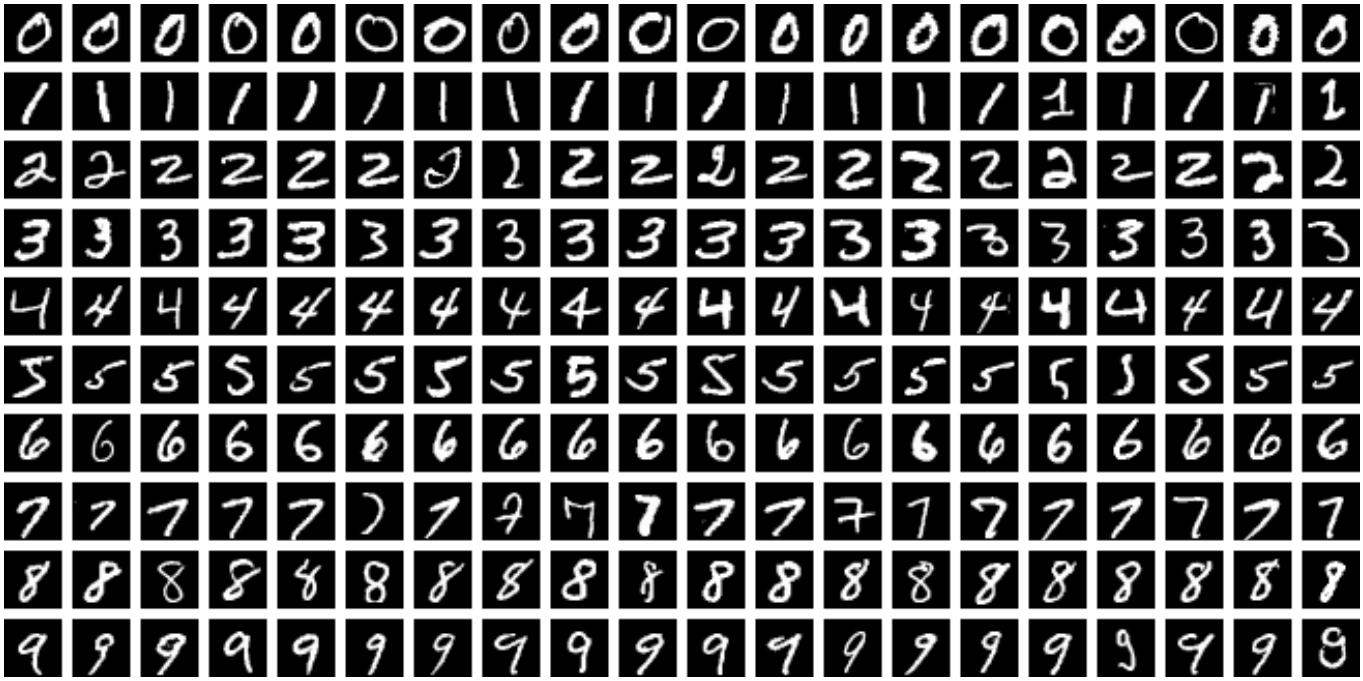
4th week practical lesson: Building Your First Fully Connected Neural Network (FCNN)

Dataset: MNIST Handwritten Digits

The **MNIST** (Modified National Institute of Standards and Technology) dataset is the 'Hello World' of deep learning. It consists of images of handwritten digits (0-9).



Dataset Characteristics

- **Content**: 70,000 greyscale images of handwritten digits.
- **Source**: Training data comes from American Census Bureau employees, while testing data comes from American high school students. This split ensures a slightly challenging and **balanced** test set.
- **Format**: Each image is 28×28 pixels, with a single channel (greyscale).

Results of Different Architectures

- K-NN - 0.96% error
- Random forest - 2.8% error
- 2-layer FCNN - 1.6% error
- single CNN - 0.25% error

- Complex CNN ensemble - 0.09% error

Data Splitting: Training, Validation, and Test

It's crucial to divide our data into three distinct sets to get an **unbiased** measure of our model's performance:

1. **Training Set**: Used to **update the model's weights** via backpropagation.
2. **Validation Set (Val)**: Used to monitor the model's performance **during training**. It helps us tune **hyperparameters** (like the number of epochs) and detect **overfitting** early. *The model never learns directly from this set.*
3. **Test Set**: Used only once, *after* training is completely finished, to provide the final, **unbiased evaluation** of the model's generalization ability.

A usual split is 70% training, 10% validation, 20% test set

Setup and Reproducibility (Random Seed)

We import the necessary libraries and set a **random seed**. Setting a seed is essential for **reproducibility**; it ensures that the random initialization of model weights and the shuffling of data are the same every time you run the notebook. This makes experiments comparable.

The PyTorch Tensor: NumPy on Steroids

- numpy.ndarray is similar to torch.Tensor
  - both are n-dimensional arrays storing data
  - the way to index, slice, and manipulate them is nearly identical
  - torch.tensor, torch.zeros, torch.matmul
- PyTorch Power: PyTorch Tensors can be seamlessly moved to a GPU (like an NVIDIA CUDA core) for massive parallel computation, which is essential for training large neural networks.
- PyTorch Power 2: Automatic Differentiation
  - This is the most critical concept for deep learning and where PyTorch truly diverges from NumPy.
  - The Problem: Training a neural network requires calculating gradients (derivatives) of a loss function with respect to the network's parameters (weights and biases) during backpropagation. Writing these derivative calculations manually is complex and error-prone.
  - PyTorch Solution (Autograd): PyTorch's autograd engine automatically tracks all operations performed on a Tensor. If you enable tracking, PyTorch builds a dynamic computational graph in the background. Once you calculate a loss, you call .backward(), and PyTorch uses this graph to automatically compute all the necessary gradients.

```
torch_tensor = torch.from_numpy(numpy_array)
numpy_array = torch_tensor.numpy()
```

```
import torch                                    # The core PyTorch libr
import torch.nn as nn                           # Contains classes for
import torch.nn.functional as F                 # Provides functions us
import torch.optim as optim                     # Contains optimization

import torchvision.transforms as transforms     # For image manipulatio
import torchvision.datasets as datasets         # Provides access to st
import torch.utils.data as torchdata            # Module for data loadi

import matplotlib.pyplot as plt                 # For plotting graphs a
import numpy as np                              # For numerical operati

import random

SEED = 42                                       # The chosen seed value

random.seed(SEED)                               # Sets the seed for Pyt
np.random.seed(SEED)                            # Sets the seed for Num
torch.manual_seed(SEED)                         # Sets the seed for PyT
torch.cuda.manual_seed_all(SEED)              # Use if running on multi
```

```
# Create a tensor
tensor = torch.randn(3, 3)

# Move it to the GPU (if available)
if torch.cuda.is_available():
    device = torch.device("cuda")
    gpu_tensor = tensor.to(device)
    print(f"Tensor is on device: {gpu_tensor.device}")
else:
    print("CUDA not available. Tensor remains on CPU.")
```

```
# 1. Create a Tensor and tell PyTorch to track operations (the magic sw
x = torch.tensor(2.0, requires_grad=True)

# 2. Perform operations (build the graph: y = 2x^2 + 5)
y = 2 * x**2 + 5

# 3. Compute the loss (the derivative of y w.r.t x is dy/dx = 4x)
z = y.sum()

# 4. Run backpropagation (PyTorch computes the gradient)
z.backward()

# 5. Access the gradient (dy/dx at x=2.0 is 4 * 2.0 = 8.0)
print(f"Calculated Gradient (dy/dx): {x.grad.item()}")
```

Hyperparameters

These are configuration variables set *before* training begins. They are not learned by the model.

Why Powers of 2 for Batch Size?

Modern GPUs are optimized for operations on data structures whose sizes are powers of 2 (16, 32, 64, 128, etc.). This is because:

- **Memory alignment**: GPU memory is organized in blocks that are powers of 2
- **Parallel processing**: GPU cores process data in chunks (warps) of 32 threads
- **Efficiency**: Operations on power-of-2 sizes can use bit-shift operations instead of division

Using batch sizes like 64 or 128 typically results in 10-30% faster training compared to odd sizes like 63 or 100.

```
input_size = 784        # **Input Size**: 28 * 28 pixels, the flattened
output_classes = 10     # **Output Classes**: Digits 0-9 (10 classes).
batch_size_train = 64   # **Batch Size**: The number of samples process

learning_rate = 0.001   # **Learning Rate**: The step size for the opti

n_epochs = 500            # **Epochs**: The number of full passes over
log_interval = 100      # How often (in batches) to print a training st

train_val_split_ratio = 0.9 # Split 90% of the original 60k training da
```

Data Loading and Preprocessing

Normalization

We normalize the data to have a **zero mean** and **unit standard deviation**. This ensures all pixel values are on a similar scale, stabilizing the training process and preventing large values from dominating the gradient updates.

```
# Download the training data temporarily to calculate the mean and stan
train_data = datasets.MNIST(root = '.data', train = True, download = Tr

# Calculate mean and standard deviation (scaling pixel values from [0,
mean = train_data.data.float().mean() / 255.0
std = train_data.data.float().std() / 255.0

print(f"Calculated Mean: {mean.item():.4f}")
print(f"Calculated Std: {std.item():.4f}")
```

```
# 1. Define the transformation pipeline
img_transforms = transforms.Compose([
                                transforms.ToTensor(), # Converts the
                                transforms.Normalize(mean = mean.item(
                                ])

# 2. Load the full 60,000 training data with transformations applied
train_data_full = datasets.MNIST(root = ".data", train = True, download

# 3. Load the 10,000 test data with transformations applied
test_data = datasets.MNIST(root = ".data", train = False, download = Tr

# 4. Split the full training data into the proper Training and Validati
n_train_examples = int(len(train_data_full) * train_val_split_ratio) #
n_valid_examples = len(train_data_full) - n_train_examples           #

train_data, valid_data = torchdata.random_split(train_data_full, [n_tra
```

Data Loaders

A **DataLoader** is a PyTorch utility that handles the loading of data in batches. It efficiently manages:

- **Batching**: Grouping samples into `batch_size_train` (64).
- **Shuffling**: We only **shuffle the training data** (`shuffle=True`) to prevent the model from learning the order of samples, which could introduce unwanted biases.
- **Loading**: Preparing the next batch while the GPU is processing the current one (using multiple worker threads).

```
# Training DataLoader: Shuffles and batches the training data.
train_loader = torchdata.DataLoader(train_data, batch_size=batch_size_t

# Validation/Test Loaders: Batches data, but no shuffling is needed for
val_loader = torchdata.DataLoader(valid_data, batch_size=batch_size_tra
test_loader = torchdata.DataLoader(test_data, batch_size=batch_size_tra

# Check the shape of a single batch: (Batch Size, Channels, Height, Wid
examples = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(examples)
print(f"Shape of the first batch: {example_data.shape}")
```

Network Architecture: Fully Connected Neural Network (FCNN)

The FCNN (or Multi-Layer Perceptron) is built using `nn.Module`. In an FCNN, every neuron in one layer is connected to every neuron in the next layer. This works well for our simple, flattened image data.

```
class FCNN(nn.Module):
    # __init__: Constructor, defines the layers (modules) of the networ
    def __init__(self, input_size, output_classes):
        super(FCNN, self).__init__() # Calls the constructor of the par

        self.fc1 = nn.Linear(input_size, 1024)  # **nn.Linear** (Fully
        self.fc2 = nn.Linear(1024, 2048)         # Hidden Layer 2: Maps
        self.fc3 = nn.Linear(2048, output_classes) # **Output Layer**:
        self.relu = nn.ReLU()        # **nn.ReLU** (Rectified Linear Un
        # # nn.Sequential stacks layers sequentially.
        # self.network = nn.Sequential(
        #     nn.Linear(input_size, 1024),  # **nn.Linear** (Fully Conn
        #     nn.ReLU(inplace=True),        # **nn.ReLU** (Rectified Li
        #     nn.Linear(1024, 2048),        # Hidden Layer 2.
        #     nn.ReLU(inplace=True),        # ReLU activation.
        #     nn.Linear(2048, output_classes) # **Output Layer**: Maps
        # )

    # forward: Defines the flow of data through the network
    def forward(self, x):
        # Input x is (Batch, 1, 28, 28). FCNNs require a flat vector.
        x = x.view(-1, input_size) # **Flattening**: Reshapes the image
        x = self.fc1(x)            # Pass the flattened data through th
        x = self.relu(x)           # Apply ReLU activation.
        x = self.fc2(x)            # Pass through the second fully conn
        x = self.relu(x)           # Apply ReLU activation.
        x = self.fc3(x)            # Pass through the output layer to g

        # x = self.network(x)       # Pass the flattened data through
        return x                   # Returns the raw logit scores.
```

Device, Loss Function, and Optimizer

Device Selection

We check if a **GPU** (CUDA device) is available on the machine (e.g., in a Colab runtime). If so, we use the GPU for faster matrix operations; otherwise, we default to the CPU.

Loss Function

We use **`nn.CrossEntropyLoss`**. This is the standard loss function for multi-class classification, as it efficiently combines the Softmax activation (to get probabilities) and the Negative Log Likelihood loss (to measure the error).

Optimizer

We use **`optim.SGD` (Stochastic Gradient Descent)**. The optimizer is responsible for updating the model's parameters (`net.parameters()`) based on the calculated gradients and the `learning_rate`.

```
# Device Selection: Check for CUDA/GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Model Initialization
net = FCNN(input_size, output_classes).to(device=device) # Instantiate

# Loss Function
criterion = nn.CrossEntropyLoss() # Standard for multi-class classifica

# Optimizer
optimizer = optim.SGD(net.parameters(), lr=learning_rate) # SGD uses th
```

Let's run the model

Data loading

We load one image, visualize it, to see what we are dealing with

Model inference

We run our model instance using the input data, and check the result

It will not be appropriate

```
img, target = train_data.__getitem__(0)
print(img.shape)
print(target)
```

```
import matplotlib.pyplot as plt
plt.imshow(img.squeeze(), cmap='gray')
plt.show()

print(f"The image label is: {target}")
```

```
# Lets run our model
img = img.to(device=device)
output = net(img)
print(output)

most_likely_output = output.argmax()
print(f"The most likely output is: {most_likely_output}")
```

Training Function

`train()` Function (The Learning Process)

For each batch in the training set:

1. **`optimizer.zero_grad()`**: Clears accumulated gradients from the *previous* batch to prevent mixing.
2. **Forward Pass**: Calculates the output predictions.
3. **Calculate Loss**: Measures the error.
4. **`loss.backward()`**: Runs **backpropagation** to calculate the gradients (slopes) of the loss with respect to every weight and bias.
5. **`optimizer.step()`**: Updates the weights using the gradients and the learning rate ($W_{new} = W_{old} - \text{LR} \cdot \nabla L$$W_{new} = W_{old} - \text{LR} \cdot \nabla L$).

```
def train(model, device, train_loader, optimizer, criterion, epoch, log
    model.train() # Set model to training mode.
    train_losses = []

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device) # Move tensor

        optimizer.zero_grad()                          # 1. Zero the gra
        output = model(data)                           # 2. Forward pass
        loss = criterion(output, target)               # 3. Calculate Lo
        loss.backward()                                # 4. Backward pas
        optimizer.step()                               # 5. Update Weigh

        if batch_idx % log_interval == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(
            train_losses.append(loss.item())

    return train_losses
```

Validation Function

`validate()` Function (Evaluation)

This function measures performance on the validation/test set:

- **`model.eval()`**: Sets the model to evaluation mode (e.g., disables dropout if it were used).
- **`torch.no_grad()`**: **Crucial!** Tells PyTorch not to calculate or store gradients, saving memory and speeding up the process, since we are not updating weights.

```
def validate(model, device, val_loader, criterion, dataset_name="Valida
    model.eval()                                    # Set model to eva
    test_loss = 0
    correct = 0

    with torch.no_grad():                           # Disable gradient
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(val_loader)
    accuracy = 100. * correct / len(val_loader.dataset)

    print(f'\n{dataset_name} set: Average loss: {test_loss:.4f}, Accura

    return test_loss, accuracy
```

Main Training Loop and Learning Curves

We iterate through the epochs, calling the `train` and `validate` functions sequentially.

The **Learning Curve** plot is the most important diagnostic tool. It compares the Training Loss (how well the model fits the data it sees) and the Validation Loss (how well the model generalizes to unseen data).

Interpreting Learning Curves

Curve Behavior	Diagnostic	Solution
Training Loss $\gg$$\gg$ Validation Loss	**Underfitting** (High Bias)	The model is too simple, or the
Training Loss $\ll$$\ll$ Validation Loss	**Overfitting** (High Variance)	The model has memorized the
Training Loss $\approx$$\approx$ Validation Loss	**Good Fit**	The model is learning the gene

```
# Lists to store metrics for plotting the learning curve
train_losses = []
val_losses = []
val_accuracies = []

# Main Training Loop
for epoch in range(1, n_epochs + 1):
    print(f"\n{'='*20} EPOCH {epoch} {'='*20}")

    # 1. Train for one epoch
    epoch_train_losses = train(net, device, train_loader, optimizer, cr
    train_losses.extend(epoch_train_losses)

    # 2. Validate after one epoch
    val_loss, val_accuracy = validate(net, device, val_loader, criterio
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

print(f"\n\n--- Training Complete ---")
```

```
len(train_loader)
```

```
# Plotting the Training and Validation Loss Curve
plt.figure(figsize=(10, 5))
# Training loss is plotted per batch/iteration
plt.plot(train_losses, label='Training Loss (per 100 batches)')
# Validation loss is plotted once per epoch
val_loss_x_axis = [i * len(train_loader)/log_interval for i in range(1, 
plt.plot(val_loss_x_axis, val_losses, 'ro-', label='Validation Loss (per

plt.title('Learning Curve: Training vs. Validation Loss')
plt.xlabel(f'Training Iteration (Batch, {log_interval} steps)')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

Final Evaluation (Test Set)

This step performs the final, single, and unbiased evaluation on the completely unseen Test Set.

```
print("--- Final Test Set Evaluation ---")
# The validate function is reused for the test set evaluation
test_loss, test_accuracy = validate(net, device, test_loader, criterion,
```

```
rand_ind = random.randint(0, len(test_loader.dataset))
print(f"Random Index: {rand_ind}")
img, target = test_data.__getitem__(rand_ind)
print(img.shape)
print(target)
```

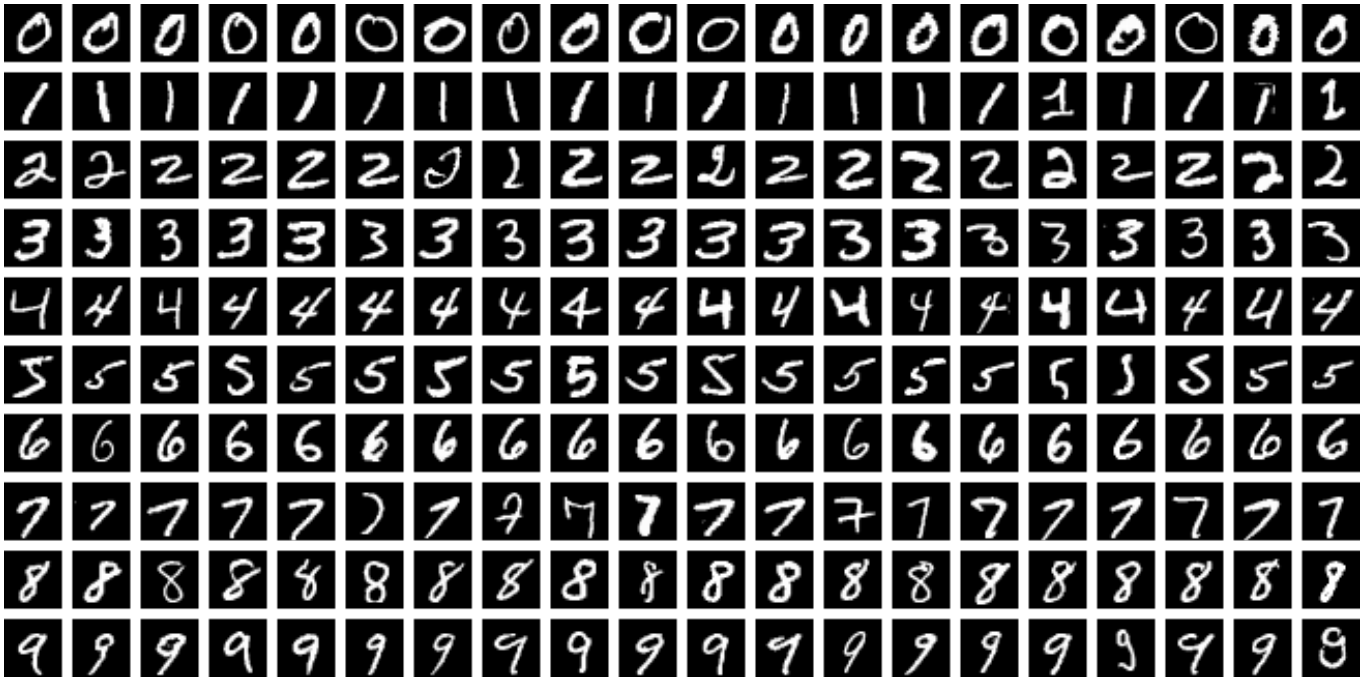
```
import matplotlib.pyplot as plt
plt.imshow(img.squeeze(), cmap='gray')
plt.show()

print(f"The image label is: {target}")
```

```
# Lets run our model
img = img.to(device=device)
output = net(img)
print(output)

most_likely_output = output.argmax()
print(f"The most likely output is: {most_likely_output}")
```

5th week practical lesson: Building Your First Convolutional Neural Network (CNN)

Dataset: MNIST Handwritten Digits

The **MNIST** (Modified National Institute of Standards and Technology) dataset is the 'Hello World' of deep learning. It consists of images of handwritten digits (0-9).



Dataset Characteristics

- **Content**: 70,000 greyscale images of handwritten digits.
- **Source**: Training data comes from American Census Bureau employees, while testing data comes from American high school students. This split ensures a slightly challenging and **balanced** test set.
- **Format**: Each image is 28×28 pixels, with a single channel (greyscale).

Results of Different Architectures

- K-NN - 0.96% error
- Random forest - 2.8% error
- 2-layer FCNN - 1.6% error
- single CNN - 0.25% error

- Complex CNN ensemble - 0.09% error

Data Splitting: Training, Validation, and Test

It's crucial to divide our data into three distinct sets to get an **unbiased** measure of our model's performance:

1. **Training Set**: Used to **update the model's weights** via backpropagation.
2. **Validation Set (Val)**: Used to monitor the model's performance **during training**. It helps us tune **hyperparameters** (like the number of epochs) and detect **overfitting** early. *The model never learns directly from this set.*
3. **Test Set**: Used only once, *after* training is completely finished, to provide the final, **unbiased evaluation** of the model's generalization ability.

A usual split is 70% training, 10% validation, 20% test set

```
import torch                                    # The core PyTorch libr
import torch.nn as nn                           # Contains classes for
import torch.nn.functional as F                 # Provides functions us
import torch.optim as optim                     # Contains optimization

import torchvision.transforms as transforms     # For image manipulatio
import torchvision.datasets as datasets         # Provides access to st
import torch.utils.data as torchdata            # Module for data loadi

import matplotlib.pyplot as plt                 # For plotting graphs a
import numpy as np                              # For numerical operati

import random

SEED = 42                                       # The chosen seed value

random.seed(SEED)                               # Sets the seed for Pyt
np.random.seed(SEED)                            # Sets the seed for Num
torch.manual_seed(SEED)                         # Sets the seed for PyT
torch.cuda.manual_seed_all(SEED)               # Use if running on multi

import copy
```

```
# Create a tensor
tensor = torch.randn(3, 3)

# Move it to the GPU (if available)
if torch.cuda.is_available():
    device = torch.device("cuda")
    gpu_tensor = tensor.to(device)
    print(f"Tensor is on device: {gpu_tensor.device}")
else:
    print("CUDA not available. Tensor remains on CPU.")
```

```
Tensor is on device: cuda:0
```

Hyperparameters

These are configuration variables set *before* training begins. They are not learned by the model.

Why Powers of 2 for Batch Size?

Modern GPUs are optimized for operations on data structures whose sizes are powers of 2 (16, 32, 64, 128, etc.). This is because:

- **Memory alignment**: GPU memory is organized in blocks that are powers of 2
- **Parallel processing**: GPU cores process data in chunks (warps) of 32 threads
- **Efficiency**: Operations on power-of-2 sizes can use bit-shift operations instead of division

Using batch sizes like 64 or 128 typically results in 10-30% faster training compared to odd sizes like 63 or 100.

```
input_size = 784        # **Input Size**: 28 * 28 pixels, the flattened
output_classes = 10     # **Output Classes**: Digits 0-9 (10 classes).
batch_size_train = 64   # **Batch Size**: The number of samples process

learning_rate = 0.001   # **Learning Rate**: The step size for the opti

n_epochs = 5            # **Epochs**: The number of full passes over th
log_interval = 100      # How often (in batches) to print a training st

train_val_split_ratio = 0.9 # Split 90% of the original 60k training da
```

Data Loading and Preprocessing

Normalization

We normalize the data to have a **zero mean** and **unit standard deviation**. This ensures all pixel values are on a similar scale, stabilizing the training process and preventing large values from dominating the gradient updates.

```
# Download the training data temporarily to calculate the mean and stan
train_data = datasets.MNIST(root = '.data', train = True, download = Tr

# Calculate mean and standard deviation (scaling pixel values from [0,
mean = train_data.data.float().mean() / 255.0
std = train_data.data.float().std() / 255.0

print(f"Calculated Mean: {mean.item():.4f}")
print(f"Calculated Std: {std.item():.4f}")
```

```
100%|██████████| 9.91M/9.91M [00:01<00:00, 5.62MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 132kB/s]
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.23MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 13.8MB/s]
Calculated Mean: 0.1307
Calculated Std: 0.3081
```

Augmentation

Augmentation means some arbitrary transformation on an image to create a more diverse, general dataset

- `RandomRotation` - randomly rotates the image between `(-x, +x)` degrees, where we have set `x = 5`. Note, the `fill=(0,)` is due to a bug in some versions of torchvision.
- `RandomCrop` - this first adds `padding` around our image, 2 pixels here, to artificially make it bigger, before taking a random `28x28` square crop of the image.

These operate on PIL (python imaging library) images

```
# 1. Define the transformation pipeline with augmentation

train_transforms = transforms.Compose([
    transforms.RandomRotation(5, fill = 0),
    transforms.RandomCrop(28, padding=2),
    transforms.ToTensor(),
    transforms.Normalize(mean = mean.item(), std = std.item())
])

test_transforms = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean = mean.item(), std = std.item())
    ]
)

# 2. Load the full 60,000 training data with transformations applied
train_data_full = datasets.MNIST(root = ".data", train = True, download

# 3. Load the 10,000 test data with transformations applied
test_data = datasets.MNIST(root = ".data", train = False, download = Tr

# 4. Split the full training data into the proper Training and Validati
n_train_examples = int(len(train_data_full) * train_val_split_ratio) #
n_valid_examples = len(train_data_full) - n_train_examples           #

print(f"Number of training examples: {n_train_examples}")
print(f"Number of validation examples: {n_valid_examples}")
print(f"Number of test examples: {len(test_data)}")

train_data, valid_data = torchdata.random_split(train_data_full, [n_tra
```

```
Number of training examples: 54000
Number of validation examples: 6000
Number of test examples: 10000
```

We can now simply replace the validation set's transform by overwriting it with our test transforms from above.

As the validation set is a `Subset` of the training set, if we change the transforms of one, then by default Torchvision will change the transforms of the other. To stop this from happening, we make a `deepcopy` of the validation data.

```
valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

Data Loaders

A **DataLoader** is a PyTorch utility that handles the loading of data in batches. It efficiently manages:

- **Batching**: Grouping samples into `batch_size_train` (64).
- **Shuffling**: We only **shuffle the training data** (`shuffle=True`) to prevent the model from learning the order of samples, which could introduce unwanted biases.
- **Loading**: Preparing the next batch while the GPU is processing the current one (using multiple worker threads).

```
# Training DataLoader: Shuffles and batches the training data.
train_loader = torchdata.DataLoader(train_data, batch_size=batch_size_t

# Validation/Test Loaders: Batches data, but no shuffling is needed for
val_loader = torchdata.DataLoader(valid_data, batch_size=batch_size_tra
test_loader = torchdata.DataLoader(test_data, batch_size=batch_size_tra

# Check the shape of a single batch: (Batch Size, Channels, Height, Wid
examples = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(examples)

print(f"Shape of the first batch: {example_data.shape}")
```

```
Shape of the first batch: torch.Size([64, 1, 28, 28])
```

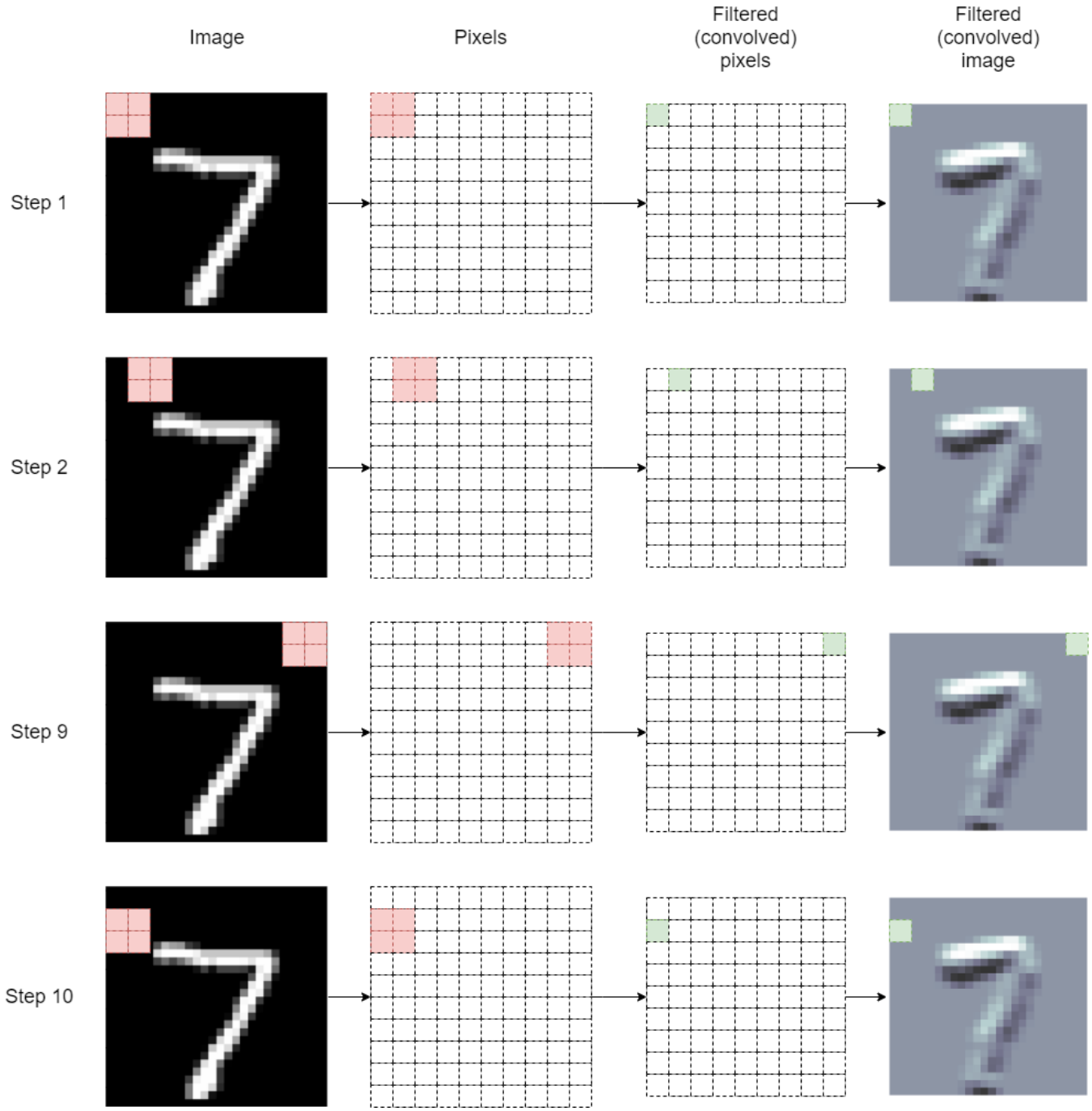
Defining the Model

The **LeNet architectures**, and almost every modern neural network computer vision architecture, makes use of **convolutional neural network (CNN) layers**.

So, what is a CNN layer? Each convolutional layer has a number of *filters*, also commonly referred to as *kernels*. A filter is a (usually) square matrix that slides across the pixels in an image from left-to-right, top-to-bottom. At each "step", the filter performs a convolution operation on the image. The output of the convolutional layer is the result of these convolutions after the filter's final "step".

Note: in machine learning frameworks there aren't actually "steps", the result for every filter location is calculated at the same time, in parallel. This is a lot faster than actually stepping through the image, but thinking about it in terms of steps makes everything easier to visualize.

Let's have a look at a single 2x2 filter passing over an image. We'll pretend the image is 10x10 pixels in this example.



The filter (red) slides over the pixels of the image, stepping one pixel at a time. The size of the steps is called the *stride*, and we use a stride of one in this implementation, which means the filter moves one pixel at a time horizontally and moves one pixel down

once it reaches the end of a row. The result of the convolution operation (green) is a pixel in the filtered image. **All of these convolutions produce a new, filtered image.**

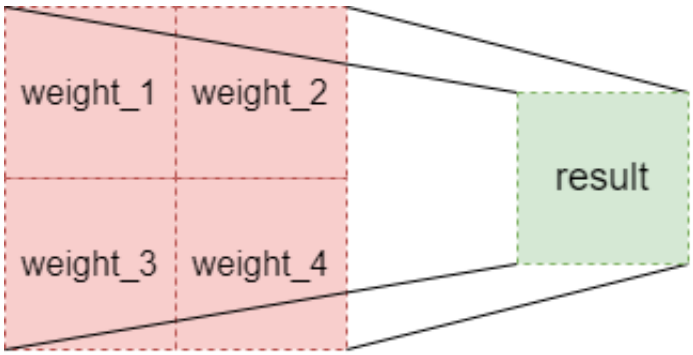
Notice how the image coming out of the CNN layer is **smaller** than the image coming into the CNN. This is because the 2x2 filter has only nine steps horizontally and vertically. If we wanted to keep the output image the same size as the input image, we could add padding - usually black pixels - around our image.

With padding and step size (stride), the size of the output image is:

$$\text{height}_{\text{out}} = \frac{\text{height}_{\text{in}} + 2\cdot \text{padding}- \text{filter}_{\text{height}}}{\text{stride}} + 1$$$$\text{height}_{\text{out}} = \frac{\text{height}_{\text{in}} + 2\cdot \text{padding}- \text{filter}_{\text{height}}}{\text{stride}} + 1$$

$$\text{width}_{\text{out}} = \frac{\text{width}_{\text{in}}+ 2\cdot \text{padding} - \text{filter}_{\text{width}}}{\text{stride}} + 1$$$$\text{width}_{\text{out}} = \frac{\text{width}_{\text{in}}+ 2\cdot \text{padding} - \text{filter}_{\text{width}}}{\text{stride}} + 1$$

How do we calculate the values of the output pixels using the filter? It's simply multiplying and adding! Each of the input image pixels covered by a filter is multiplied by the filter's weight over that pixel. All of these products are then summed together to get the value of the pixel in the output image.



result =
pixel_1 * weight_1 +
pixel_2 * weight_2 +
pixel_3 * weight_3 +
pixel_4 * weight_4

The same weights are used by the filter over the whole image. **The weights do not change depending on the filter's location within the image.** One nice thing about this is that the filters (and the convolutional layers themselves) are *translation invariant*, that means it doesn't matter where a feature (curve, edge, line) appears in an image, the convolutional layer will find all occurrences of it.

The weights for the filters, much like the weights of the linear layers in multilayer perceptrons, are learned via gradient descent and backpropagation.

Why are convolutional neural networks structured in this way? Filters applied across an

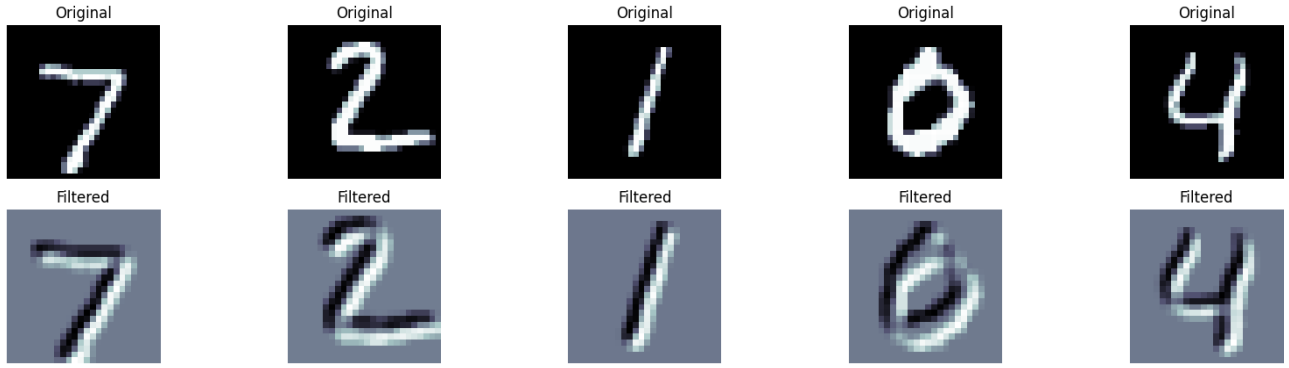
image in this way **can be used to detect patterns** such as horizontal and vertical lines within an image. These patterns can be thought of as features of the image, which our CNN extracts. These extracted features can then be combined in further layers of the neural network with other extracted features and together create higher level features, e.g. a certain position and orientation of two lines to make a cross, which can indicate the centre of a handwritten 4.

CNNs are also inspired by classic computer vision techniques, like Sobel filters. Let's try manually choosing weights of a 3x3 filter to make Sobel filters and apply them to some MNIST digits to see what type of things our CNN layers can learn.

The `plot_filter` function takes in a batch of images and a two-dimensional filter and plots the output of that filter applied to all of the images.

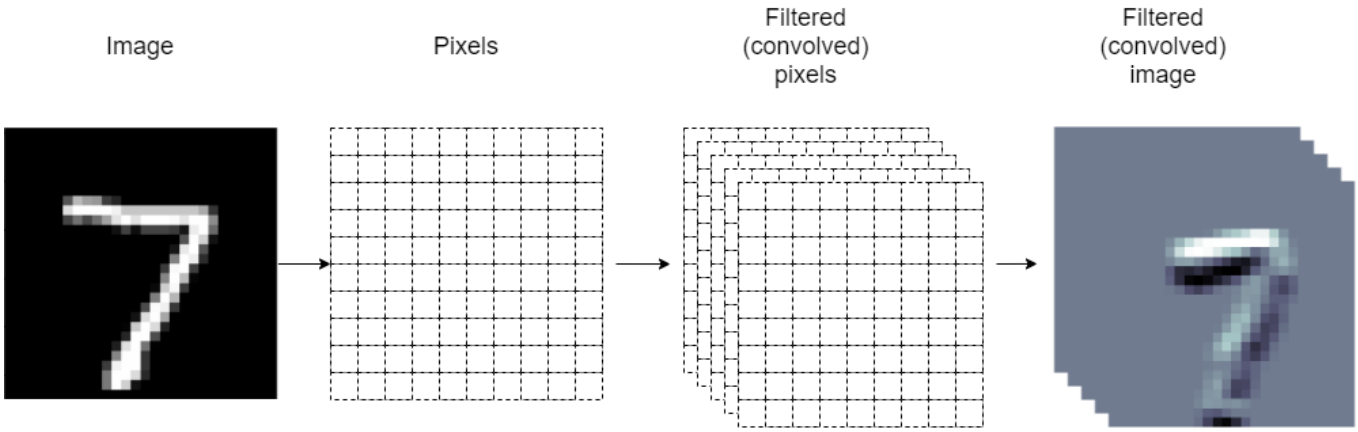
```
def plot_filter(images, filter):

    images = images = torch.cat([i.unsqueeze(0) for i in images],
                                dim=0).cpu()
    filter = torch.FloatTensor(filter).unsqueeze(0).unsqueeze(0).cpu()

    n_images = images.shape[0]

    filtered_images = F.conv2d(images, filter)

    fig = plt.figure(figsize=(20, 5))

    for i in range(n_images):

        ax = fig.add_subplot(2, n_images, i+1)
        ax.imshow(images[i].squeeze(0), cmap='bone')
        ax.set_title('Original')
        ax.axis('off')

        image = filtered_images[i].squeeze(0)

        ax = fig.add_subplot(2, n_images, n_images+i+1)
        ax.imshow(image, cmap='bone')
        ax.set_title('Filtered')
        ax.axis('off')

# We visualize 5 images
N_IMAGES = 5

# Get the first N_IMAGES images into a list
first_n_data_pairs = [test_data[i] for i in range(N_IMAGES)]
```

```
images = [image for image, label in first_n_data_pairs]

horizontal_filter = [[-1, -2, -1],
                     [0, 0, 0],
                     [1, 2, 1]]


vertical_filter = [[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]]


diagonal_filter = [[2, 1, 0],
                   [1, 0, -1],
                   [0, -1, -2]]

plot_filter(images, diagonal_filter)
```

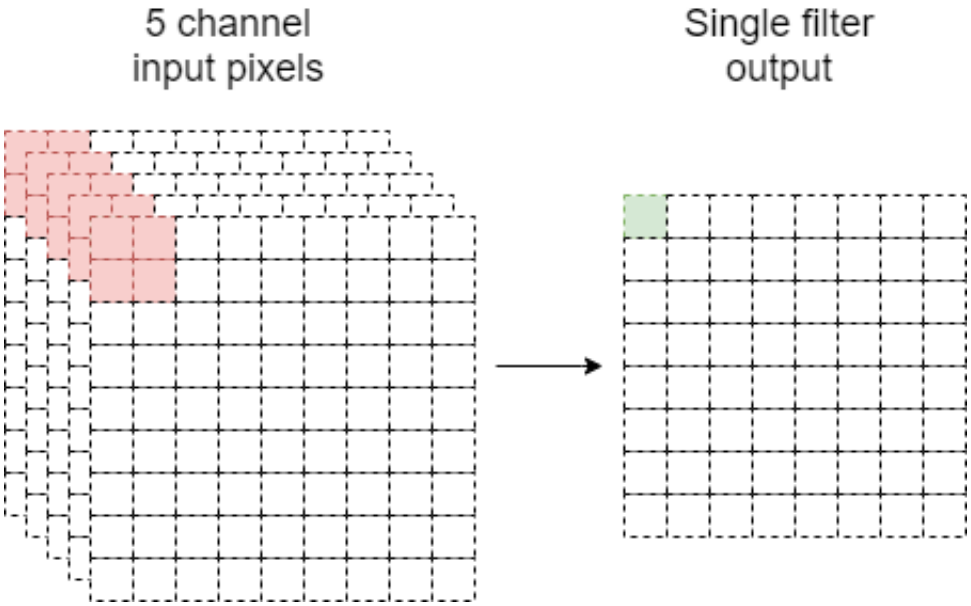


The great thing about convolutional layers is that each layer doesn't just have a single filter. It has as many filters as you want. Each filter has their own set of weights, so (in theory) is learning to extract different features.

The image below shows what happens when we use a convolutional layer with five filters. The original image with a single color channel (as it's black and white) has five filters applied to it to get five filtered images. These images are then stacked together to get what we can think of as a single image with five channels.



What about when you now want to pass this five channel filtered image to another convolutional layer? Now, that convolutional layer won't just have a height and a width, but it will also have a depth equal to the number of channels in the input image.



As you can see, the filter has a height, width and depth of 2x2x5. All the 20 pixel values covered by this filter are multiplied by the filter's weight and then summed. The result of this will have as many channels as there are filters, and a subsequent convolutional layer will have to have filters with a depth equal to that number of channels.

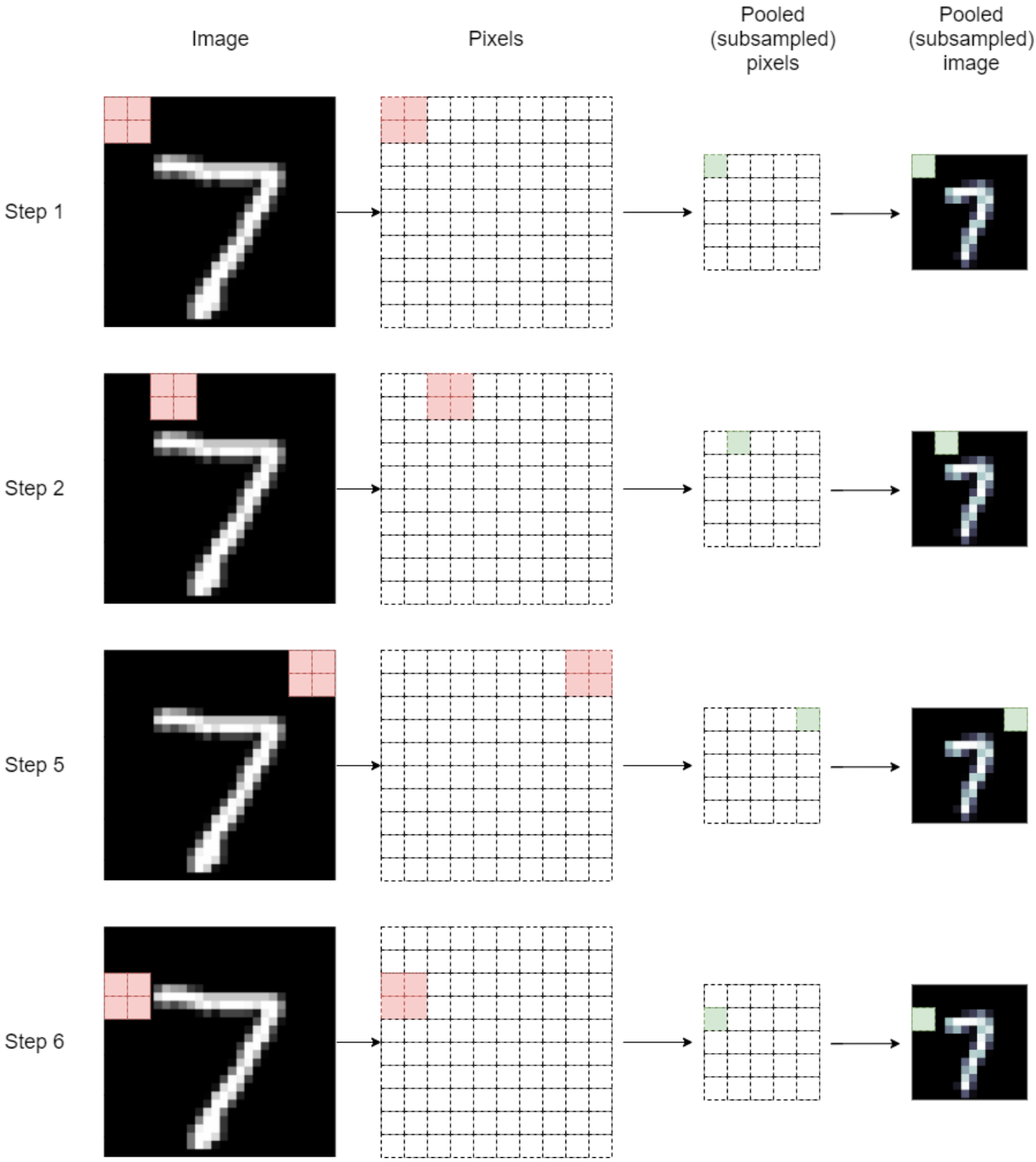
Hopefully that's enough on convolutional layers, but if not, then there are plenty of other resources about them online.

Next, we'll talk about the subsampling layers. These are layers used to reduce the size/resolution of their input and are commonly applied to the output of convolutional layers. The most common two methods of subsampling are *max pooling* and *mean pooling* (also known as *average pooling*), and thus subsampling is often know as *pooling*.

Why do we want to reduce the resolution of the image? It speeds up our model, as convolution operations are expensive. If we subsample and half the size of our image before it passes into the next convolutional layer, that's a significant speed-up.

Subsampling layers aren't too different to convolutional layers. They have a filter with a size and a stride. However, pooling layers do not have any parameters - weights and biases. They simply perform an operation on the image. Max pooling returns the maximum of the values covered by the filter, and we can think of it as extracting the single most important feature under the filter. Mean/average pooling returns the mean/average of the values covered by the filter and we can think of it as equally weighting all features under the filter.

Let's look at a 2x2 pooling operation, with a stride of 2, over an image:



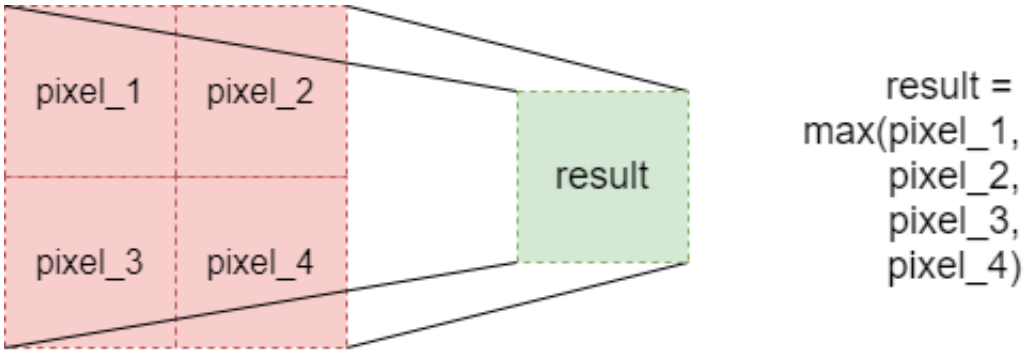
Commonly, and by default in PyTorch, the stride for the height and the width is the height and the width of the filter and each pixel is only seen by the pooling layer once, thus:

$$\text{height}_{\text{out}} = \frac{\text{height}_\text{in}}{\text{filter}_{\text{height}}}$$$$\text{height}_{\text{out}} = \frac{\text{height}_\text{in}}{\text{filter}_{\text{height}}}$$

$$\text{width}_{\text{out}} = \frac{\text{width}_\text{in}}

{\text{filter}_{\text{width}}}$$$$\text{width}_{\text{out}} = \frac{\text{width}_\text{in}}{\text{filter}_{\text{width}}}$$

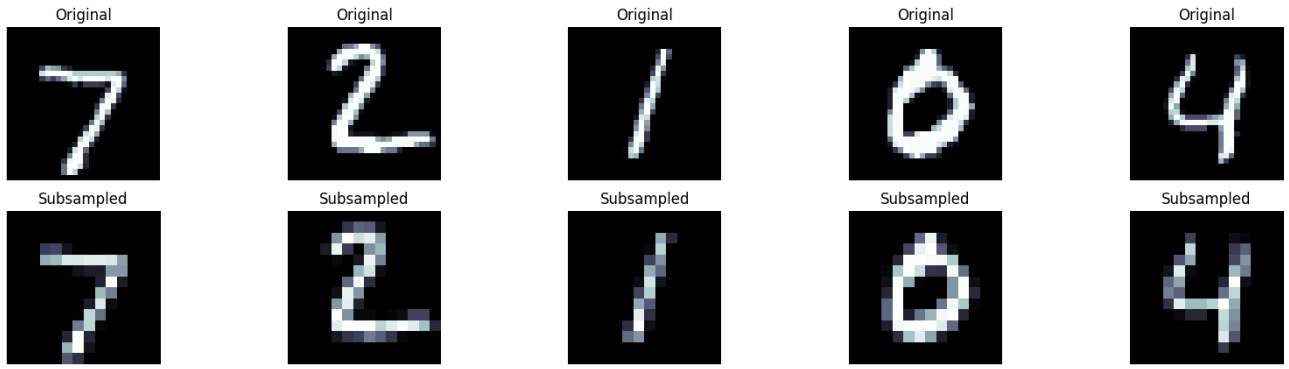
For max pooling, the value of the output for each filter location is:



Let's create a function that allows us to see the outputs of a pooling layer on a batch of images.

```
def plot_subsample(images, pool_type, pool_size):
    """
    Plot the output of a pooling layer on a batch of images.
    Pool type options: 'max', 'mean', 'avg'
    """
    images = torch.cat([i.unsqueeze(0) for i in images], dim=0).cpu()

    # Select the appropriate functions to perform pooling
    if pool_type.lower() == 'max':
        pool = F.max_pool2d
    elif pool_type.lower() in ['mean', 'avg']:
        pool = F.avg_pool2d
    else:
        raise ValueError(f'pool_type must be either max or mean, got: {

    n_images = images.shape[0]

    # Use the kernel size from the function argument
    pooled_images = pool(images, kernel_size= pool_size)

    fig = plt.figure(figsize=(20, 5))

    for i in range(n_images):

        ax = fig.add_subplot(2, n_images, i+1)
        ax.imshow(images[i].squeeze(0), cmap='bone')
        ax.set_title('Original')
        ax.axis('off')

        image = pooled_images[i].squeeze(0)

        ax = fig.add_subplot(2, n_images, n_images+i+1)
        ax.imshow(image, cmap='bone')
        ax.set_title('Subsampled')
        ax.axis('off')
```

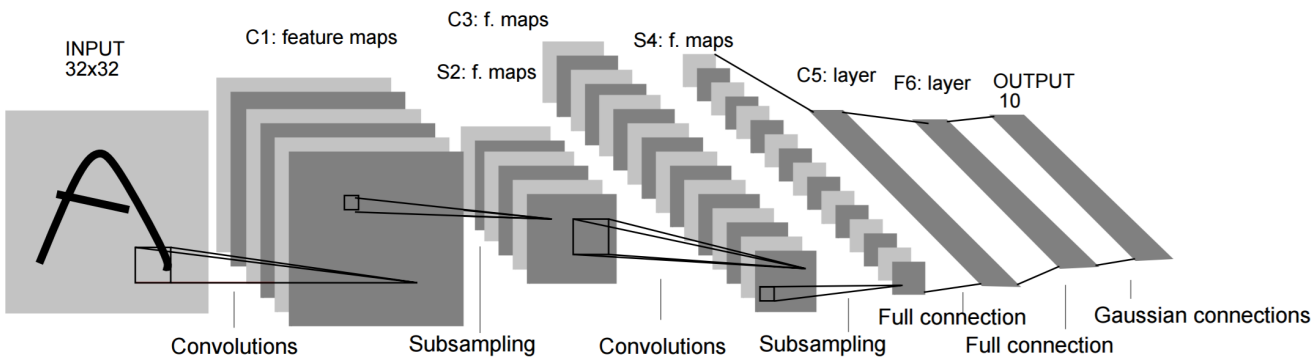
```
plot_subsample(images, 'avg', 2)
```



Similarly to convolutional layers, if the input image has more than one channel, the filter has a depth equal to the number of channels. Thus, if we did a max pool with a 5x5x6 filter (height and width of 5 and depth of 6), the output at each pixel would be the maximum value of all 150 pixels covered by the filter.

Defining the Model

Now we've gone through all the concepts, we can implement our model.



Our actual implementation is going to slightly differ from the actual LeNet architecture, as it was built to handle 32x32 images, whereas the MNIST dataset consists of 28x28 images. We're also not going to use the Gaussian connections at the end, instead we'll just use a standard linear layer.

The first layer in our model is a convolutional layer with 6 filters (PyTorch calls them `out_channels`) and a kernel size of 5. This turns our `[1, 28, 28]` image into `[6, 24, 24]`. We then downsample our image with a max pooling layer that has a filter size of 2 to get a `[6, 12, 12]` image. This is then passed through an activation function, ReLU in this case, which is applied elementwise and does not change the of the image.

Afterwards, we pass the image to the second convolutional layer with 16 filters that are 5x5x6, a height and width of 5 and a depth of 6 as our previous convolutional layer had 6 filters. This gives us an image size of `[16, 8, 8]` which we then max pool to half the height and width to `[16, 4, 4]` and then pass through another ReLU function.

We then flatten our `[16, 4, 4]` image to `[256]` and pass this through three linear layers. Each of the linear layers are followed by another ReLU, except for the last.

We return the results from the final linear layer as well as from the flattened result of the second convolutional layer, which we can plot in lower dimensions later.

Note that you should always apply your activation function **after** the pooling layer. You will get the exact same results if you apply the activation function before, however this means you will be applying your activation function to a larger number of inputs, increasing the computation required. Using the activation function after the image has been reduced in size means it will be applied to fewer inputs and thus use less computation.

```
class LeNet(nn.Module):
    def __init__(self, output_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 1,
                               out_channels=6,
                               kernel_size = 5)
        self.conv2 = nn.Conv2d(in_channels=6,
                               out_channels=16,
                               kernel_size=5)

        self.fc_1 = nn.Linear(in_features=4*4*16, out_features=120)
        self.fc_2 = nn.Linear(120, 84)
```

```
        self.fc_3 = nn.Linear(84, output_dim)


    def forward(self, x):

        # if no batch dimension, unsqueeze
        if x.dim() == 3:
            x = x.unsqueeze(0)
            # [1, 1, 28, 28]

        # x = [batch size, 1, 28, 28]
        x = self.conv1(x)

        # x = [batch size, 6, 24, 24]
        x = F.max_pool2d(x, kernel_size=2)

        # x = [batch size, 6, 12, 12]
        x = F.relu(x)
        x = self.conv2(x)

        # x = [batch size, 16, 8, 8]
        x = F.max_pool2d(x, kernel_size = 2)

        # x = [batch size, 16, 4, 4]
        x = F.relu(x)

        x = x.view(x.shape[0], -1)

        # x = [batch size, 16*4*4 = 256]

        h = x

        x = self.fc_1(x)
        x = F.relu(x)
        # x = [batch size, 120]

        x = self.fc_2(x)
        x = F.relu(x)
        # x = batch size, 84]

        x = self.fc_3(x)
        # x = [batch size, output dim]

        return x, h
```

Device, Loss Function, and Optimizer

Device Selection

We check if a **GPU** (CUDA device) is available on the machine (e.g., in a Colab runtime). If so, we use the GPU for faster matrix operations; otherwise, we default to the CPU.

Loss Function

We use **`nn.CrossEntropyLoss`**. This is the standard loss function for multi-class classification, as it efficiently combines the Softmax activation (to get probabilities) and the Negative Log Likelihood loss (to measure the error).

Optimizer

We use **`optim.SGD` (Stochastic Gradient Descent)**. The optimizer is responsible for updating the model's parameters (`net.parameters()`) based on the calculated gradients and the `learning_rate`.

```
# Device Selection: Check for CUDA/GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Model Initialization
net = LeNet(output_classes).to(device)   # Instantiate the FCNN and mov

# Loss Function
criterion = nn.CrossEntropyLoss() # Standard for multi-class classifica

# Optimizer
optimizer = optim.Adam(net.parameters(), lr=learning_rate)  # Use the A
```

```
Using device: cuda
```

Let's run the model

Data loading

We load one image, visualize it, to see what we are dealing with

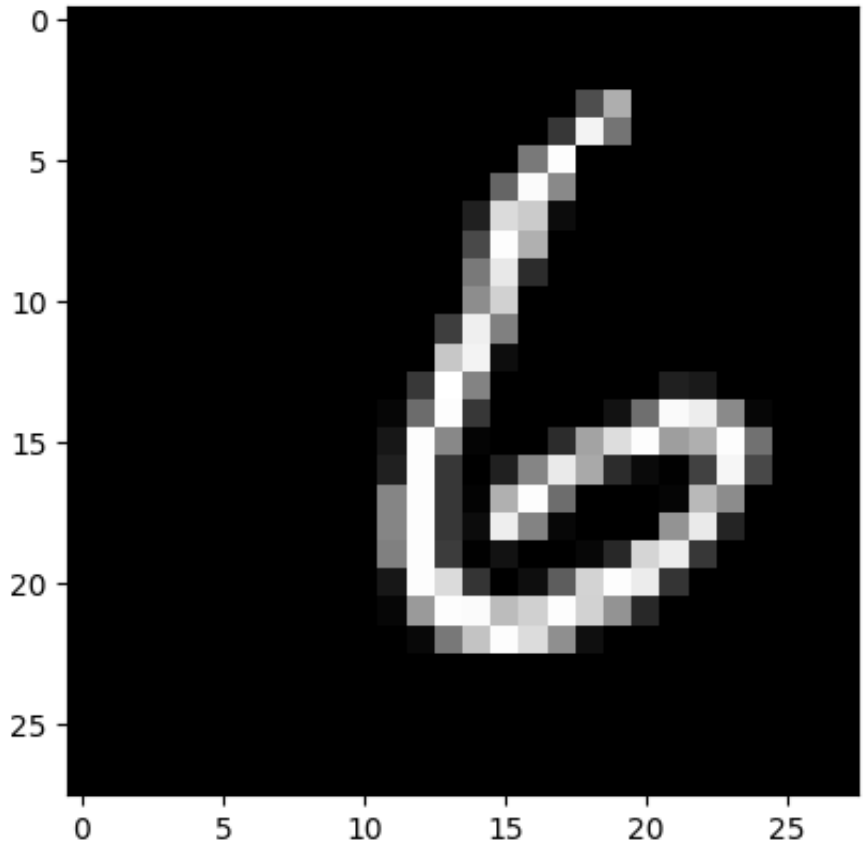
Model inference

We run our model instance using the input data, and check the result

It will not be appropriate

```
img, target = train_data.__getitem__(random.randint(0, len(train_data))
print(img.shape)
print(target)

import matplotlib.pyplot as plt
plt.imshow(img.squeeze(), cmap='gray')
plt.show()

print(f"The image label is: {target}")
```

```
torch.Size([1, 28, 28])
6
```




```
The image label is: 6
```

```
# Lets run our model
img = img.to(device=device)
output, hidden = net(img)
print(output)

most_likely_output = output.argmax()
print(f"The most likely output is: {most_likely_output}")
```

```
tensor([[-0.0705, -0.0056, -0.0217,  0.0276, -0.0082, -0.0713, -0.0849,
          0.1291,  0.0110]], device='cuda:0', grad_fn=<AddmmBackward0>)
The most likely output is: 8
```

Training Function

`train()` Function (The Learning Process)

For each batch in the training set:

1. **`optimizer.zero_grad()`**: Clears accumulated gradients from the *previous* batch to prevent mixing.
2. **Forward Pass**: Calculates the output predictions.
3. **Calculate Loss**: Measures the error.
4. **`loss.backward()`**: Runs **backpropagation** to calculate the gradients (slopes) of the loss with respect to every weight and bias.
5. **`optimizer.step()`**: Updates the weights using the gradients and the learning rate ($W_{new} = W_{old} - \text{LR} \cdot \nabla L$$W_{new} = W_{old} - \text{LR} \cdot \nabla L$).

```
def train(model, device, train_loader, optimizer, criterion, epoch, log
    model.train() # Set model to training mode.
    train_losses = []

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device) # Move tensor

        optimizer.zero_grad()                          # 1. Zero the gra
        output, hidden = model(data)                       # 2. Forwa
        loss = criterion(output, target)                       # 3. C
        loss.backward()                          # 4. Backward pass (Bac
        optimizer.step()                          # 5. Update Weights.

        if batch_idx % log_interval == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(
            train_losses.append(loss.item())

    return train_losses
```

Validation Function

`validate()` Function (Evaluation)

This function measures performance on the validation/test set:

- **`model.eval()`**: Sets the model to evaluation mode (e.g., disables dropout if it were used).
- **`torch.no_grad()`**: **Crucial!** Tells PyTorch not to calculate or store gradients, saving memory and speeding up the process, since we are not updating weights.

```
def validate(model, device, val_loader, criterion, dataset_name="Valida
    model.eval()                                    # Set model to eva
    test_loss = 0
    correct = 0

    with torch.no_grad():                           # Disable gradient
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output, hidden = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(val_loader)
    accuracy = 100. * correct / len(val_loader.dataset)

    print(f'\n{dataset_name} set: Average loss: {test_loss:.4f}, Accura

    return test_loss, accuracy
```

Main Training Loop and Learning Curves

We iterate through the epochs, calling the `train` and `validate` functions sequentially.

The **Learning Curve** plot is the most important diagnostic tool. It compares the Training Loss (how well the model fits the data it sees) and the Validation Loss (how well the model generalizes to unseen data).

Interpreting Learning Curves

Curve Behavior	Diagnostic	Solution
Training Loss $\gg$$\gg$ Validation Loss	**Underfitting** (High Bias)	The model is too simple, or the
Training Loss $\ll$$\ll$ Validation Loss	**Overfitting** (High Variance)	The model has memorized the
Training Loss $\approx$$\approx$ Validation Loss	**Good Fit**	The model is learning the gene

```
# Lists to store metrics for plotting the learning curve
train_losses = []
val_losses = []
val_accuracies = []

# Main Training Loop
for epoch in range(1, n_epochs + 1):
```

```
    print(f"\n{'='*20} EPOCH {epoch} {'='*20}")

    # 1. Train for one epoch
    epoch_train_losses = train(net, device, train_loader, optimizer, cr
                               epoch, log_interval)
    train_losses.extend(epoch_train_losses)

    # 2. Validate after one epoch
    val_loss, val_accuracy = validate(net, device, val_loader, criterio
                                      dataset_name="Validation")
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

print(f"\n\n--- Training Complete ---")
```

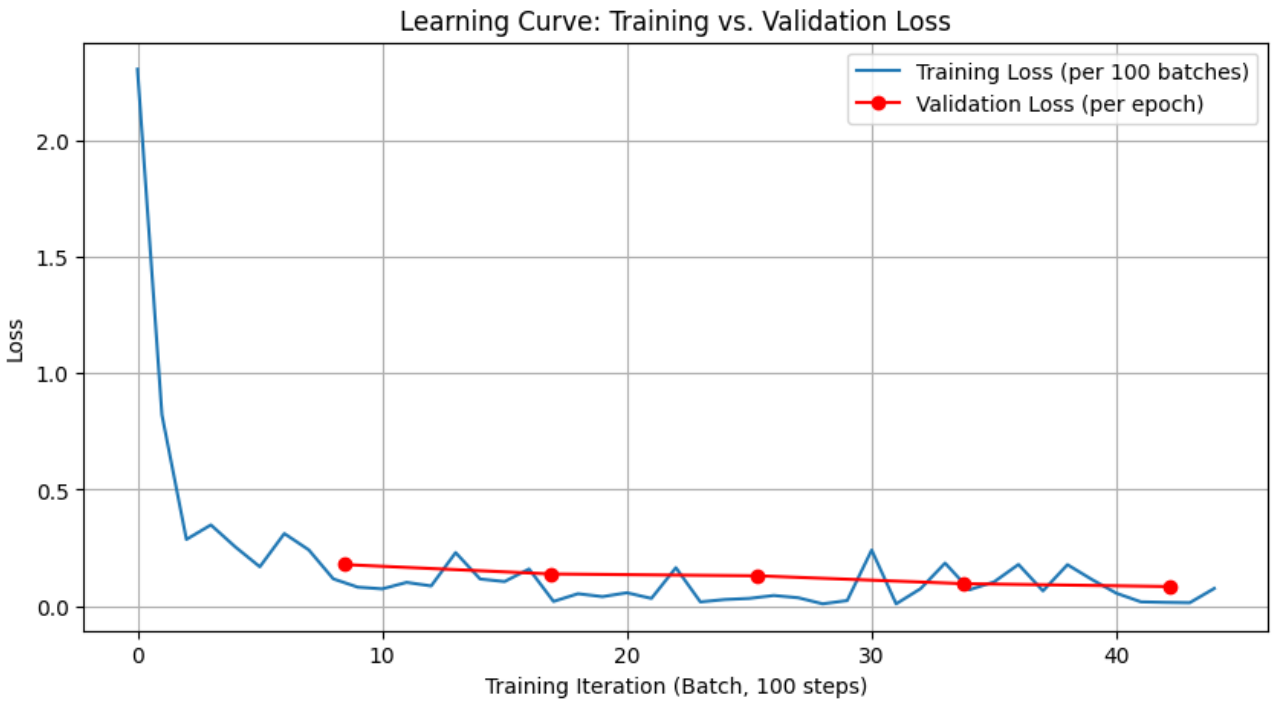
```
==================== EPOCH 1 ====================
Train Epoch: 1 [0/54000 (0%)]   Loss: 2.305903
Train Epoch: 1 [6400/54000 (12%)]       Loss: 0.821012
Train Epoch: 1 [12800/54000 (24%)]      Loss: 0.286306
Train Epoch: 1 [19200/54000 (36%)]      Loss: 0.348586
Train Epoch: 1 [25600/54000 (47%)]      Loss: 0.253906
Train Epoch: 1 [32000/54000 (59%)]      Loss: 0.168790
Train Epoch: 1 [38400/54000 (71%)]      Loss: 0.311526
Train Epoch: 1 [44800/54000 (83%)]      Loss: 0.241352
Train Epoch: 1 [51200/54000 (95%)]      Loss: 0.116963

Validation set: Average loss: 0.1783, Accuracy: 5677/6000 (94.62%)


==================== EPOCH 2 ====================
Train Epoch: 2 [0/54000 (0%)]   Loss: 0.081365
Train Epoch: 2 [6400/54000 (12%)]       Loss: 0.073871
Train Epoch: 2 [12800/54000 (24%)]      Loss: 0.101839
Train Epoch: 2 [19200/54000 (36%)]      Loss: 0.086470
Train Epoch: 2 [25600/54000 (47%)]      Loss: 0.229738
Train Epoch: 2 [32000/54000 (59%)]      Loss: 0.116234
Train Epoch: 2 [38400/54000 (71%)]      Loss: 0.105358
Train Epoch: 2 [44800/54000 (83%)]      Loss: 0.159289
Train Epoch: 2 [51200/54000 (95%)]      Loss: 0.020014

Validation set: Average loss: 0.1380, Accuracy: 5737/6000 (95.62%)


==================== EPOCH 3 ====================
Train Epoch: 3 [0/54000 (0%)]   Loss: 0.053043
Train Epoch: 3 [6400/54000 (12%)]       Loss: 0.040595
Train Epoch: 3 [12800/54000 (24%)]      Loss: 0.057211
Train Epoch: 3 [19200/54000 (36%)]      Loss: 0.032904
Train Epoch: 3 [25600/54000 (47%)]      Loss: 0.164947
Train Epoch: 3 [32000/54000 (59%)]      Loss: 0.017971
Train Epoch: 3 [38400/54000 (71%)]      Loss: 0.028201
```

```
Train Epoch: 3 [44800/54000 (83%)]      Loss: 0.032901
Train Epoch: 3 [51200/54000 (95%)]      Loss: 0.045484

Validation set: Average loss: 0.1301, Accuracy: 5771/6000 (96.18%)


==================== EPOCH 4 ====================
Train Epoch: 4 [0/54000 (0%)]   Loss: 0.035727
Train Epoch: 4 [6400/54000 (12%)]       Loss: 0.009700
Train Epoch: 4 [12800/54000 (24%)]      Loss: 0.023735
Train Epoch: 4 [19200/54000 (36%)]      Loss: 0.240533
Train Epoch: 4 [25600/54000 (47%)]      Loss: 0.009484
Train Epoch: 4 [32000/54000 (59%)]      Loss: 0.074825
Train Epoch: 4 [38400/54000 (71%)]      Loss: 0.184690
Train Epoch: 4 [44800/54000 (83%)]      Loss: 0.070042
Train Epoch: 4 [51200/54000 (95%)]      Loss: 0.103809

Validation set: Average loss: 0.0963, Accuracy: 5809/6000 (96.82%)


==================== EPOCH 5 ====================
Train Epoch: 5 [0/54000 (0%)]   Loss: 0.178777
```

```
len(train_loader)
```

```
844
```

```
# Plotting the Training and Validation Loss Curve
plt.figure(figsize=(10, 5))
# Training loss is plotted per batch/iteration
plt.plot(train_losses, label="Training Loss (per 100 batches)")
# Validation loss is plotted once per epoch
val_loss_x_axis = [i * len(train_loader)/log_interval for i in range(1,
plt.plot(val_loss_x_axis, val_losses, 'ro-', label='Validation Loss (pe

plt.title('Learning Curve: Training vs. Validation Loss')
plt.xlabel(f'Training Iteration (Batch, {log_interval} steps)')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



Final Evaluation (Test Set)

This step performs the final, single, and unbiased evaluation on the completely unseen Test Set.

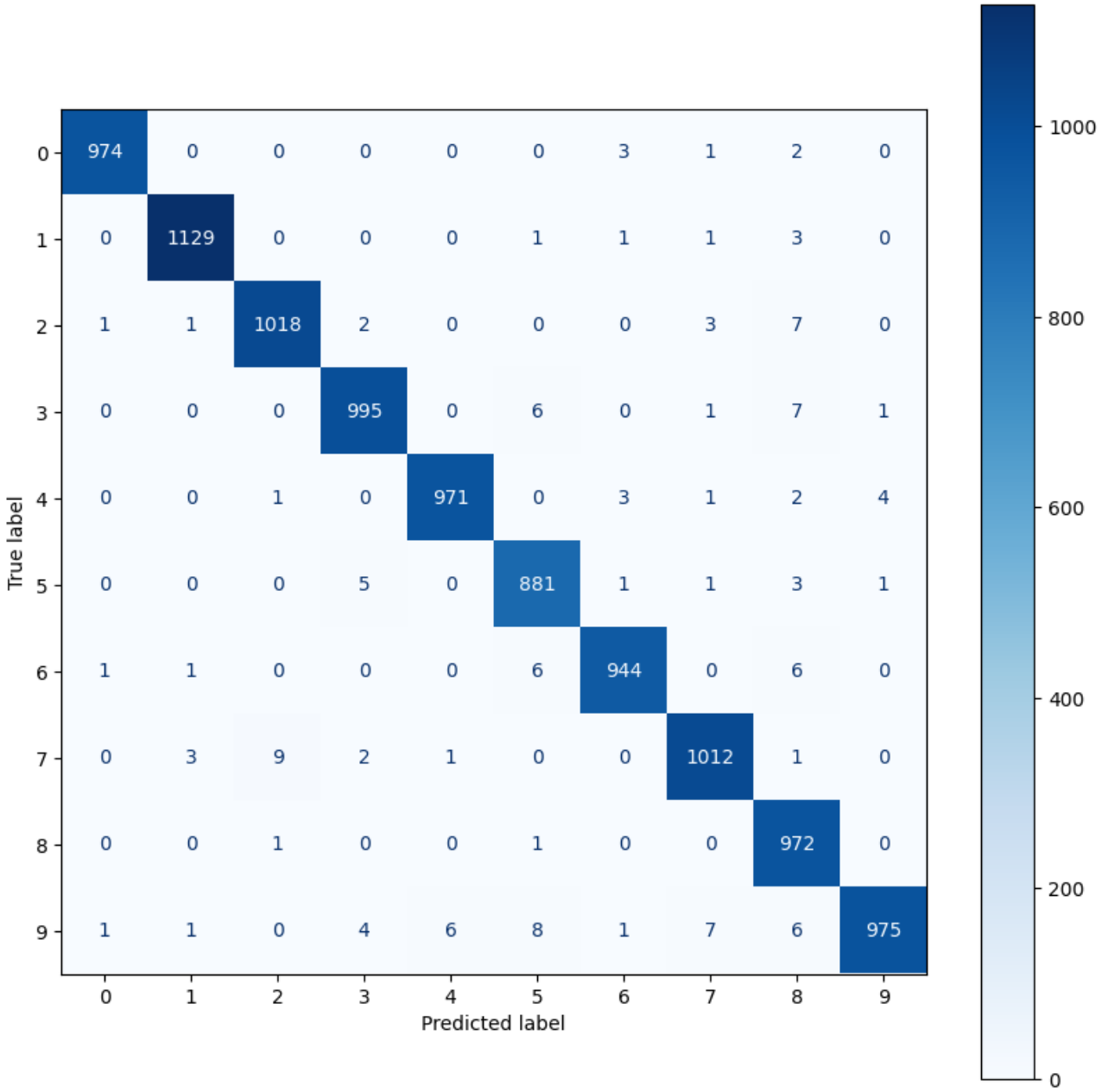
```
def get_predictions(model, iterator, device):

    model.eval()

    images = []
    labels = []
    probs = []

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)

            y_pred, _ = model(x)

            y_prob = F.softmax(y_pred, dim=-1)

            images.append(x.cpu())
            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim=0)
    labels = torch.cat(labels, dim=0)
    probs = torch.cat(probs, dim=0)

    return images, labels, probs
```

```
images, labels, probs = get_predictions(net, test_loader, device)
```

```
pred_labels = torch.argmax(probs, 1)
```

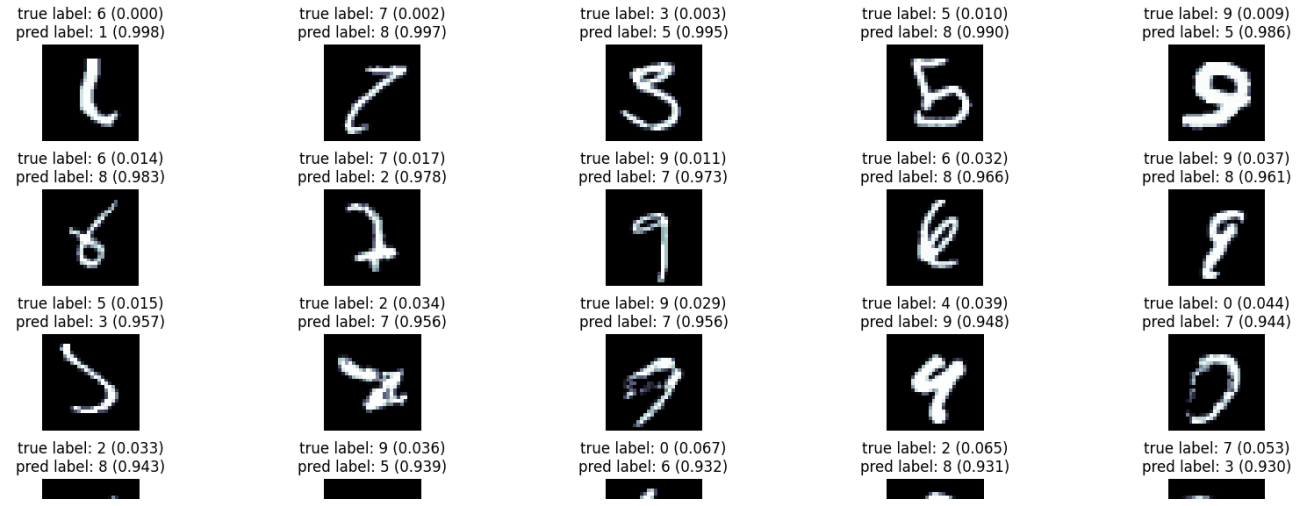
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay

def plot_confusion_matrix(labels, pred_labels):

    fig = plt.figure(figsize=(10, 10))
```

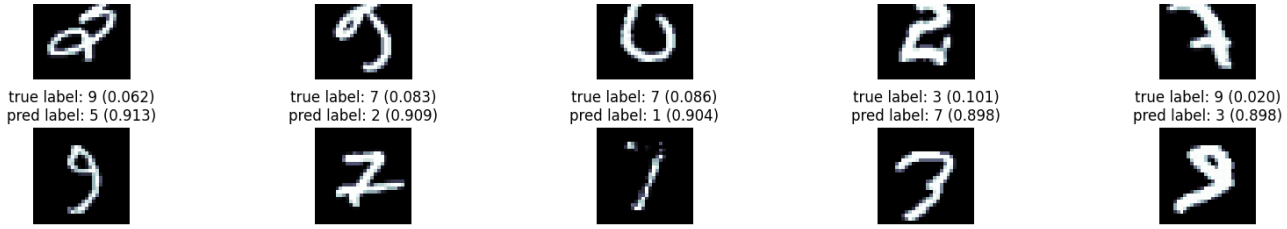
```
    ax = fig.add_subplot(1, 1, 1)
    cm = confusion_matrix(labels, pred_labels)
    cm = ConfusionMatrixDisplay(cm, display_labels=range(10))
    cm.plot(values_format='d', cmap='Blues', ax=ax)

plot_confusion_matrix(labels, pred_labels)
```



```
corrects = torch.eq(labels, pred_labels)
incorrect_examples = []

for image, label, prob, correct in zip(images, labels, probs, corrects)
    if not correct:
        incorrect_examples.append((image, label, prob))

incorrect_examples.sort(reverse=True,
                        key=lambda x: torch.max(x[2], dim=0).values)
```

```
def plot_most_incorrect(incorrect, n_images):

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize=(20, 10))
    for i in range(rows*cols):
        ax = fig.add_subplot(rows, cols, i+1)
        image, true_label, probs = incorrect[i]
        true_prob = probs[true_label]
        incorrect_prob, incorrect_label = torch.max(probs, dim=0)
        ax.imshow(image.view(28, 28).cpu().numpy(), cmap='bone')
        ax.set_title(f'true label: {true_label} ({true_prob:.3f})\n'
                     f'pred label: {incorrect_label} ({incorrect_prob:.
        ax.axis('off')
    fig.subplots_adjust(hspace=0.5)

N_IMAGES = 25

plot_most_incorrect(incorrect_examples, N_IMAGES)
```






```
print("--- Final Test Set Evaluation ---")
# The validate function is reused for the test set evaluation
test_loss, test_accuracy = validate(net, device, test_loader, criterion
```

```
--- Final Test Set Evaluation ---

Test set: Average loss: 0.0369, Accuracy: 9871/10000 (98.71%)
```

Data loading

In this notebook we'll show how to use torchvision to handle datasets that are not part of `torchvision.datasets`. Specificially we'll be using the 2011 version of the CUB200 dataset. This is a dataset with 200 different species of birds. Each species has around 60 images, which are around 500x500 pixels each. Our goal is to correctly determine which species an image belongs to - a 200-dimensional image classification problem.

As this is a relatively small dataset - ~12,000 images compared to MNIST's 70,000 images - we'll be using a pre-trained model and then performing transfer learning using discriminative fine-tuning.

Note: on the CUB200 dataset website there is a warning about some of the images in the dataset also appearing in ImageNet, which our pre-trained model was trained on. If any of those images are in our test set then this would be a form of "information leakage" as we are evaluating our model on images it has been trained on. However, the GitHub gist linked at the end of this article states that only 43 of the images appear in ImageNet. Even if they all ended up in the test set this would only be ~1% of all images in there so would have a negligible impact on performance.

Data Processing

As always, we'll start by importing all the necessary modules.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.optim.lr_scheduler as lr_scheduler
from torch.optim.lr_scheduler import _LRScheduler
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np

import copy
from collections import namedtuple
import os
import random
import shutil
import time
```

We set the random seeds for reproducibility.

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

We'll be using our own dataset instead of using one provided by `torchvision.datasets`.

The url of the CUB200-2011 dataset can be found on its website.

We can download the dataset using wget. `datasets.utils` contains some functionality for downloading and extract data which means we don't have to write it ourselves. We use the `extract_archive` function, which extracts a file to a given root folder. We should now have a `data/CUB_200_2011` folder which contains our entire dataset. We could also use linux commands for extracting the data

```
# Download directly from Caltech
!wget -O CUB_200_2011.tgz "https://data.caltech.edu/records/65de6-vp158

# Extract the tar file
ROOT = 'data'

datasets.utils.extract_archive('CUB_200_2011.tgz', ROOT)
```

```
--2025-10-15 13:20:46--  https://data.caltech.edu/records/65de6-vp158/fi
Resolving data.caltech.edu (data.caltech.edu)... 35.155.11.48
Connecting to data.caltech.edu (data.caltech.edu)|35.155.11.48|:443... c
HTTP request sent, awaiting response... 302 FOUND
Location: https://s3.us-west-2.amazonaws.com/caltechdata/96/97/8384-3670
--2025-10-15 13:20:47--  https://s3.us-west-2.amazonaws.com/caltechdata/
Resolving s3.us-west-2.amazonaws.com (s3.us-west-2.amazonaws.com)... 52.
Connecting to s3.us-west-2.amazonaws.com (s3.us-west-2.amazonaws.com)|52
HTTP request sent, awaiting response... 200 OK
Length: 1150585339 (1.1G) [application/octet-stream]
Saving to: ‘CUB_200_2011.tgz’

CUB_200_2011.tgz    100%[===================>]   1.07G  5.54MB/s    in 4

2025-10-15 13:25:28 (3.91 MB/s) - ‘CUB_200_2011.tgz’ saved [1150585339/1

'data'
```

To handle using custom datasets, torchvision provides a `datasets.ImageFolder` class.

`ImageFolder` expects data to be stored in the following way:

```
root/class_x/xxy.png
root/class_x/xxz.jpg
```

```
root/class_y/123.jpeg
root/class_y/nsdf3.png
root/class_y/asd932_.jpg
```

That is, each folder in the root directory is the name of a class, and within each of those folders are the images that correspond to that class. The images in the downloaded dataset are currently in the form of:

```
CUB_200_2011/images/class_a/image_1.jpg
CUB_200_2011/images/class_a/image_2.jpg

CUB_200_2011/images/class_b/image_1.jpg
CUB_200_2011/images/class_b/image_2.jpg

CUB_200_2011/images/class_c/image_1.jpg
CUB_200_2011/images/class_c/image_2.jpg
```

This means we could call `datasets.ImageFolder(root = '.data/CUB_200_2011/images)` and it would load all of the data. However, we want to split our data into train and test splits. This could be done with `data.random_split`, which we have used in the past to create our validation sets - but we will show how to manually create a `train` and `test` folder and store the relevant images in those folders. This way means that we only need to create a train/test split once and re-use it each time we re-run the notebook

We first set a `TRAIN_RATIO` which will decide what percentage of the images per class are used to make up the training set, with the remainder making up the test set. We create a `train` and `test` folder within the `CUB_200_2011` folder - after first deleting them if they already exist. Then, we get a list of all classes and loop through each class. For each class we get the image names, use the first `TRAIN_RATIO` of them for the training set and the remainder for the test set. We then copy - with `shutil.copyfile` - each of the images into their respective `train` or `test` folder. It is usually better to copy, rather than move, the images to create your custom splits just in case we accidentally mess up somewhere.

After running the below cell we have our training set as:

```
CUB_200_2011/images/train/class_a/image_1.jpg
```

```
CUB_200_2011/images/train/class_a/image_2.jpg

CUB_200_2011/images/train/class_b/image_1.jpg
CUB_200_2011/images/train/class_b/image_2.jpg

CUB_200_2011/images/train/class_b/image_1.jpg
CUB_200_2011/images/train/class_b/image_2.jpg
```

and our test set as:

```
CUB_200_2011/images/test/class_a/image_48.jpg
CUB_200_2011/images/test/class_a/image_49.jpg

CUB_200_2011/images/test/class_b/image_48.jpg
CUB_200_2011/images/test/class_b/image_49.jpg

CUB_200_2011/images/test/class_c/image_48.jpg
CUB_200_2011/images/test/class_c/image_49.jpg
```

This train/test split only needs to be created once and does not need to be created again on subsequent runs.

Note: `ImageFolder` will only load files that have image related extensions, i.e. jpg/jpeg/png, so if there was, for example, a `.txt` file in one of the class folders then it would not be loaded with the images. If we wanted more flexibility when deciding which files to load or not - such as not loading .png images or loading images with an esoteric format - then we could either use the `is_valid_file` argument of the `ImageFolder` class or use `DatasetFolder` and provide a list of valid extensions to the `extensions` argument.

```
TRAIN_RATIO = 0.8

ROOT = 'data'

# We define the paths to the different splits of data
data_dir = os.path.join(ROOT, 'CUB_200_2011')
images_dir = os.path.join(data_dir, 'images')
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')

# Delete train and test directories if they already exist
```

```
if os.path.exists(train_dir):
    shutil.rmtree(train_dir)
if os.path.exists(test_dir):
    shutil.rmtree(test_dir)

# Create train and test directories
os.makedirs(train_dir)
os.makedirs(test_dir)

classes = os.listdir(images_dir)

# Iterating through class folders
for c in classes:

    class_dir = os.path.join(images_dir, c)

    images = os.listdir(class_dir)

    n_train = int(len(images) * TRAIN_RATIO)

    train_images = images[:n_train]
    test_images = images[n_train:]

    os.makedirs(os.path.join(train_dir, c), exist_ok = True)
    os.makedirs(os.path.join(test_dir, c), exist_ok = True)

    # copy train files to train dir, test files to test dir
    for image in train_images:
        image_src = os.path.join(class_dir, image)
        image_dst = os.path.join(train_dir, c, image)
        shutil.copyfile(image_src, image_dst)

    for image in test_images:
        image_src = os.path.join(class_dir, image)
        image_dst = os.path.join(test_dir, c, image)
        shutil.copyfile(image_src, image_dst)
```

Now we've got our train/test splits we can go ahead and calculate the mean and standard deviation (std) of our dataset to normalize it. We're actually going to use a pre-trained model in this notebook so will be using the mean and std desired by the pre-trained data, so we don't actually have to calculate this - however it is left as an example.

Calculating the mean and std is slightly different than when using a dataset provided by torchvision as those datasets have all of the images stored as numpy arrays in the data's `data` attribute, whilst datasets loaded by `ImageFolder` and `DataFolder` do not.

First, we load the `train_data` from the `train` folder. Remember: the mean and std must only be calculated from the training data. This will load PIL images by default so we pass the `ToTensor` transform which converts all the PIL images to tensors and scales them from 0-255 to 0-1.

We then loop through each image and calculate the mean and std across the height and width dimensions with `dim = (1,2)`, summing all the means and stds and then finding the average by dividing them by the number of examples, `len(train_data)`.

Again, this only needs to be calculated once per dataset and the means and stds calculated here can be re-used without calculating them for other runs. The exception to this is if we used a different train/test split, then we would need to calculate these again.

```
train_data = datasets.ImageFolder(root = train_dir,
                                  transform = transforms.ToTensor())

means = torch.zeros(3)
stds = torch.zeros(3)

for img, label in train_data:
    means += torch.mean(img, dim = (1,2))
    stds += torch.std(img, dim = (1,2))

means /= len(train_data)
stds /= len(train_data)

print(f'Calculated means: {means}')
print(f'Calculated stds: {stds}')
```

```
Calculated means: tensor([0.4860, 0.4995, 0.4308])
Calculated stds: tensor([0.1822, 0.1812, 0.1929])
```

Now to actually load our data. As we are going to be using a pre-trained model we will need to ensure that our images are the same size and have the same normalization as those used to train the model - which we find on the torchvision models page.

There's a few things we need to consider in regards to data processing when using pre-trained models.

As mentioned in the torchvision models page:

> All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

Even though the VGG models can handle images as small as 32x32 and convolutional layers are translation invariant, our images still need to be resized as the pre-trained `classifier` layer is expecting certain features to appear in certain places within the flattened 512x7x7 output of the `features` layer after the adaptive average pooling. Using a different image size than was used to pre-train the model causes features sent to the `classifier` to be in different places than expected, and thus leads to poor performance when using the pre-trained model.

We need to use the same means and stds to make the colors of the images fed to the model with the pre-trained parameters be the same as they were to train the pre-trained model. Let's say the original dataset had lots of dark green images and the mean for the green channel was be relatively low, say 0.2, and the dataset we are going to use had lots of light green images with the mean for the green channel being around 0.8. The pre-trained model was trained with dark green pixels normalized to zero (subtracting the mean). If we incorrectly used the means and stds from our dataset we want to apply our model on, then light green pixels will be normalized to zero, thus our pre-trained model will think a given light green image is actually a dark green image. This will confuse our model and lead to poor performance.

We handle the resizing with the `Resize` transform, passing in the desired size. As the images are larger we can also get away with slightly higher rotations and crops within the `RandomRotation` and `RandomCrop` transforms. We pass the pre-trained mean and stds the same way we would pass our calculated means and transforms.

```
pretrained_size = 224
pretrained_means = [0.485, 0.456, 0.406]
pretrained_stds= [0.229, 0.224, 0.225]

train_transforms = transforms.Compose([
                           transforms.Resize(pretrained_size),
                           transforms.RandomRotation(5),
                           transforms.RandomHorizontalFlip(0.5),
                           transforms.RandomCrop(pretrained_size, paddi
                           transforms.ToTensor(),
                           transforms.Normalize(mean = pretrained_means
                                                std = pretrained_stds)
                       ])

test_transforms = transforms.Compose([
                           transforms.Resize(pretrained_size),
                           transforms.CenterCrop(pretrained_size),
                           transforms.ToTensor(),
                           transforms.Normalize(mean = pretrained_means
                                                std = pretrained_stds)
                       ])
```

We load our data with our transforms...

```
train_data = datasets.ImageFolder(root = train_dir,
                                  transform = train_transforms)

test_data = datasets.ImageFolder(root = test_dir,
                                 transform = test_transforms)
```

...create the validation split...

```
VALID_RATIO = 0.9

n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples

train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples, n_valid_e
```

...and then overwrite the validation transforms, making sure to do a `deepcopy` to stop this also changing the training data transforms.

```
valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

To make sure nothing has messed up we'll print the number of examples in each of the data splits - ensuring they add up to the number of examples indicated on the CUB200-2011 dataset website (11,788).

```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 8472
Number of validation examples: 942
Number of testing examples: 2374
```

Next, we'll create the iterators with the largest batch size that fits on our GPU.

```
BATCH_SIZE = 16

train_iterator = data.DataLoader(train_data,
                                 shuffle = True,
                                 batch_size = BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data,
                                 batch_size = BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size = BATCH_SIZE)
```

To ensure the images have been processed correctly we can plot a few of them - ensuring we re-normalize the images so their colors look right.

```
def normalize_image(image):
    image_min = image.min()
    image_max = image.max()
    image.clamp_(min = image_min, max = image_max)
    image.add_(-image_min).div_(image_max - image_min + 1e-5)
    return image
```

```
def plot_images(images, labels, classes, normalize = True):

    n_images = len(images)

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize = (15, 15))

    for i in range(rows*cols):

        ax = fig.add_subplot(rows, cols, i+1)

        image = images[i]

        if normalize:
            image = normalize_image(image)

        ax.imshow(image.permute(1, 2, 0).cpu().numpy())
        label = classes[labels[i]]
        ax.set_title(label)
        ax.axis('off')
```

We can see the images look fine, however the names of the classes provided by the folders containing the images are a little long and sometimes overlap with neighbouring images.

```
N_IMAGES = 25

images, labels = zip(*[(image, label) for image, label in
                       [train_data[i] for i in range(N_IMAGES)]])

classes = test_data.classes

plot_images(images, labels, classes)
```


140.Summer_Tanager
159.Black_and_white_Warbler
162.Canada_Warbler
169.Magnolia_Warbler
111.Loggerhead_Shrike
176.Prairie_Warbler
195.Carolina_Wren
144.Common_Tern
001.Black_footed_Albatross
189.Red_bellied_Woodpecker
046.Gadwall
199.Winter_Wren
077.Tropical_Kingbird
188.Pileated_Woodpecker
118.House_Sparrow

One way to solve the issue with the names of the classes would have been to manually change the names of the folders before we copied them over into the `train` and `test` folders.

Another approach is to directly change the names of each class provided by the dataset's `.classes`. We'll make a `format_label` function which will strip off the number at the start of each class and convert them into title case.

```
def format_label(label):
    label = label.split('.')[-1]
    label = label.replace('_', ' ')
    label = label.title()
    label = label.replace(' ', '')
    return label
```

Let's change the class names and re-plot the images with their new class names.

No more overlapping!

```
test_data.classes = [format_label(c) for c in test_data.classes]

classes = test_data.classes

plot_images(images, labels, classes)
```


PrairieWarbler
CarolinaWren
CommonTern
BlackFootedAlbatross
RedBelliedWoodpecker
Gadwall
WinterWren
TropicalKingbird
PileatedWoodpecker
HouseSparrow

VGG model

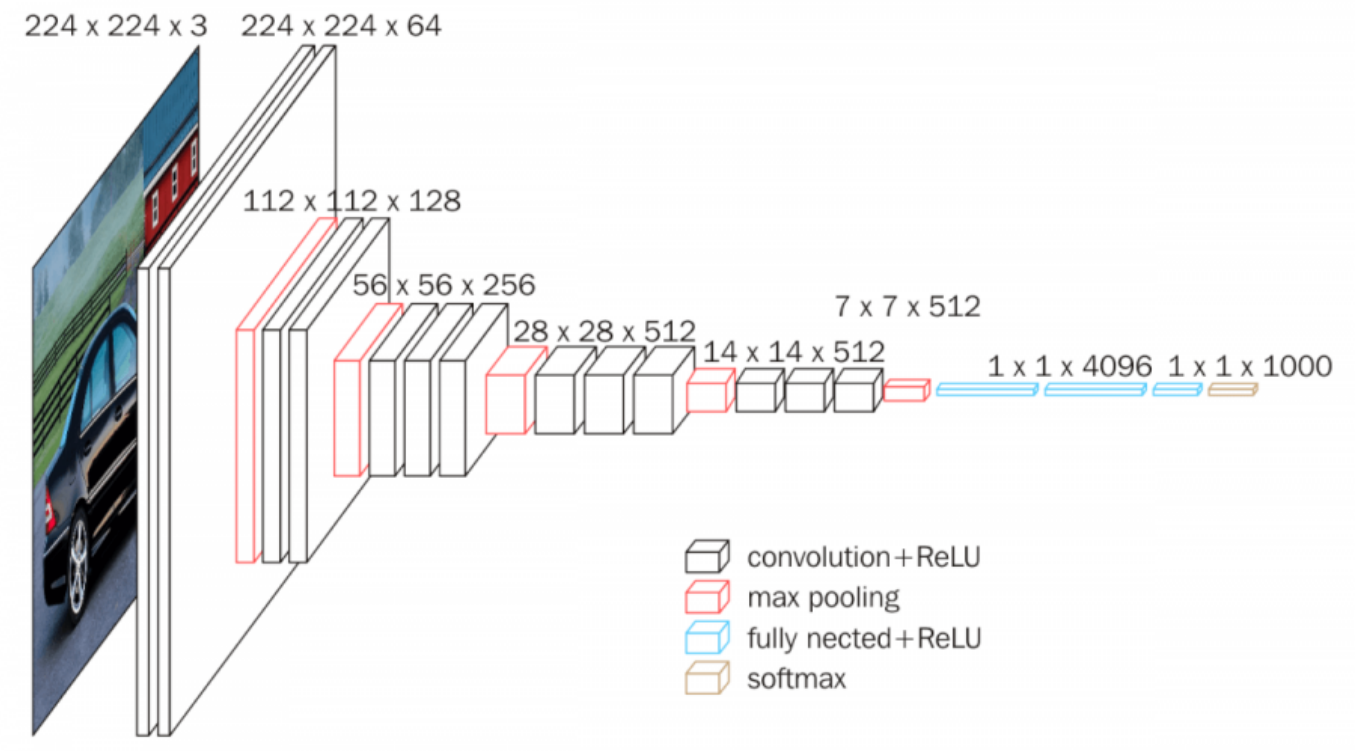
In this notebook we will be implementing one of the VGG model variants. VGG is a neural network model that uses convolutional neural network (CNN) layers and was designed for the ImageNet challenge, which it won in 2014.

VGG is not a single model, but a family of models that are all similar but have different *configurations*. Each configuration specifies the number of layers and the size of each layer. The configurations are listed in table 1 of the VGG paper and denoted by a letter, although recently they are just referred to as the number of layers with weights in the model, e.g. configuration "A" has 11 layers with weights so is known as VGG11.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Below is the architecture of configuration "D", also known as VGG16, for a 224x224

color image.



The other commonly used VGG variants are VGG11, VGG13 and VGG19, which correspond to configurations "A", "B", and "E". Configurations "A-LRN" and "C" - which is the same as "D" but with smaller filter sizes in some convolutional layers - are rarely used.

In this notebook, we will use the 2011 version of the CUB200 dataset. However we will be making use of *pre-trained* models.

Usually we will initialize our weights randomly - following some weight initialization scheme - and then train our model. Using a pre-trained model means some - potentially all - of our model's weights are not initialized randomly, but instead taken from a copy of our model that has already been trained on some task. The task the model has been pre-trained on does not necessarily have to match the "downstream task" - the task we want to use the pre-trained model on. For example, a model that has been trained to classify images can then be used to detect objects within an image.

The theory is that these pre-trained models have already learned high level features within images that will be useful for our task. This means we don't have to learn them from scratch when using the pre-trained model for our task, causing our model to converge earlier. We can also think of the pre-trained model as being a very good set of weights to initialize our model from, and using pre-trained models usually leads to a performance improvement compared to initializing our weights randomly.

The act of using a pre-trained model is generally known as *transfer learning*, as we are learning to transfer knowledge from one task to another. It is also referred to as *fine-tuning*, as we fine-tune our parameters trained on one task to the new, downstream task. The terms *transfer learning* and *fine-tuning* are used interchangably in machine learning.

We are also going to look into a technique called discriminative fine-tuning, initially introduced to improve transfer learning for text classification but has been used for computer vision tasks too.

Defining the Model

Usually the next thing we'd do is load our data, however this is quite different when using a pre-trained model, so we'll first introduce the VGG architecture and then show how to load a pre-trained VGG model.

Below we define the VGG11 model. The only new feature introduced here is the `AdaptiveAvgPool2d`. As well as the standard `AvgPool` and `MaxPool` layers, PyTorch has "adaptive" versions of those layers. In adaptive pooling layers we specify the desired output size of the pooling layer instead of the size of the filter used by the pooling layer. Here, we want an output size of 7x7. We know that all VGG configurations end with a convolutional layer that has 512 filters, thus if our `features` layer for each configuration always has a size of 7x7 we do not have to change the `classifier` for each VGG configuration. The advantage of using adaptive layers is that it allows us to apply our model to images of different sizes - down to a minimum size, which is 32x32 in VGG models.

Note: even though VGG net can handle images as small as 32x32 it is designed to give optimal performance for larger images. We handle this later on in this practice.

How do the adaptive pooling layers calculate the size of their filters? For each dimension, i.e. height and width, we calculate:

```
filter_size = (input_size + desired_size - 1) // desired_size
```

`//` means we round down to the nearest integer. So, if we wanted to filter a 32x32 image to 7x7 we would have a 6x6 filter. When the filter is applied to the image it will need to overlap at some points, i.e. some pixels will be covered by the filter twice.

Let's define the VGG11 architecture.

We can see that we always use the same size filter (2x2) and stride (2) in all of our max pool layers. As mentioned before, the default stride for pooling layers is equal to the kernel size.

For the convolutional layers we always use the same filter size (3x3) and padding (1). As a reminder, padding adds `padding` pixels with values of zero around each side of the image in each channel before the filter is applied. Each convolutional layer is followed by a ReLU non-linearity. We then set the `in_channels` to be equal to the number of

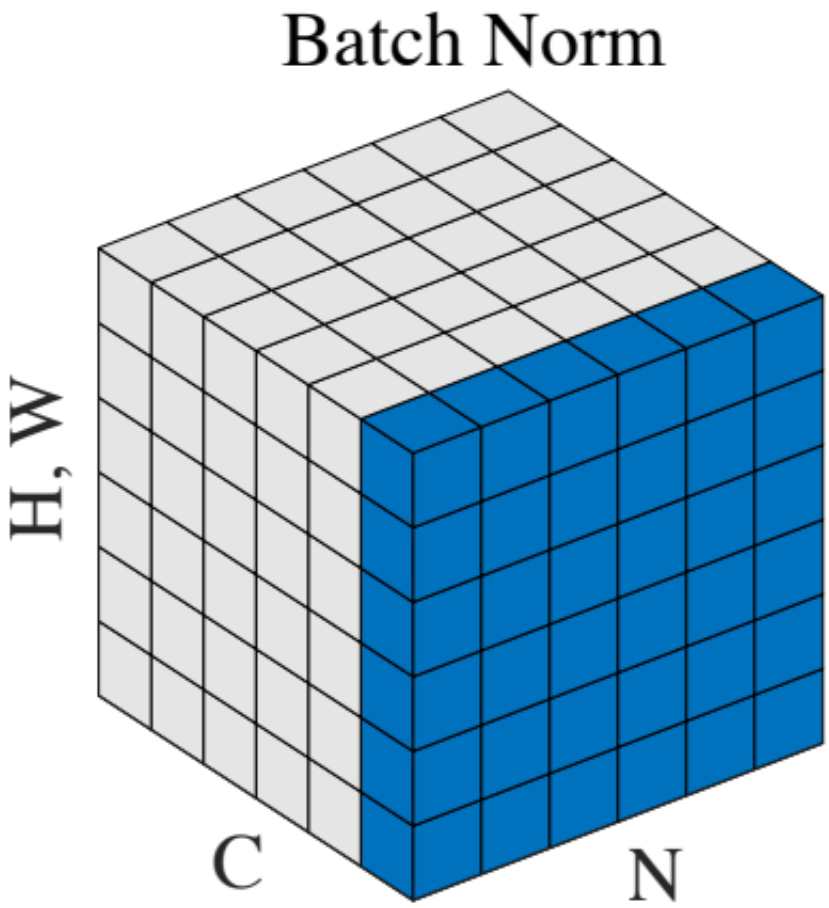
filters in the convolutional layer so the next convolutional layer has the correct `in_channels`.

Another new layer introduced here is batch normalization (BN) defined with `BatchNorm2d` and only used if `batch_norm = True`. Normalization is when we try to ensure that some data has a mean of zero and a standard deviation (std) of one as this improves learning - both stability and convergence speed - in machine learning models. Previously we have normalized our input data using PyTorch transforms. However, as our model begins to train and the parameters change then the mean and std output by each layer will also change. A change in mean and std of the output of one layer will cause a change in mean and std for all following layers. Our model has no way to correct these changes in means and stds once training has begun.

BN is a layer with learnable parameters - two per filter - denoted γ and β. The layer normalizes, scales and then shifts across the channel dimension of the input. The output of a BN layer is calculated by:

$$ \begin{align} \mu_\mathcal{B} &= \frac{1}{m}\sum^m_{i=1} x_i\\ \sigma^2_\mathcal{B} &= \frac{1}{m}\sum^m_{i=1}(x_i-\mu_\mathcal{B})^2\\ \hat{x}_i &= \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}\\ y_i &= \gamma \hat{x}_i + \beta = \text{BN}_{\gamma,\beta}(x_i) \end{align} $$

The batch $\mathcal{B}$ has m examples, $x_i,\dots,x_m$. We first calculate the mean and variance across each channel dimension of the batch, $\mu_\mathcal{B}$ and $\sigma^2_\mathcal{B}$. Then normalize the batch by subtracting the channel means and dividing by the channel stds (the square root of the variance plus a small epsilon term to avoid division by zero) across each channel. We then scale and shift each channel of this normalized batch of inputs, $\hat{x}_i$ using γ and β.



Why do we scale and shift? Why not leave the outputs with a mean of zero and a std of one? Perhaps there is a better mean and std for our task instead of zero and one. If this is the case then our model can learn this whilst training as γ and β are learnable parameters. However, to bias our model to start off with the idea that a mean of zero and a std of one is a good idea by default γ and β are initialized to one and zero.

There are a few things to consider with batch normalization during inference time (validation or testing). The first is that we don't want to calculate the mean and variance of our data to normalize it during inference. This is because an image in the validation or test set might be drastically different to an image in the training set, and we do not want to remove that information via normalization. The second is what to do with a batch size of one, common when deploying models. Calculating the mean and variance across a single example (a batch size of one) doesn't make sense.

Luckily, there is a solution to both these problems. Instead of using the actual mean and variance of a batch, we use an exponentially weighted moving average which we update every batch. Then, when using inference (with any batch size, including one) we use the saved weighted average of the means and variances.

This video has a good explanation of batch normalization and other types of normalization layers. For another explanation on why batch normalization helps, check out this paper.

One last thing to mention on batch normalization is that, in theory, it should be used **after** the activation function. Why would you normalize the output of a layer only to just ruin the normalization effect with an activation function? However, in the original VGG architecture they use batch normalization before the activation function, so we do too.

```
class VGG(nn.Module):
    """
    VGG11 architecture for educational purposes.
    Note: In practice, we'll use the pretrained version from torchvisio
    This is shown to understand the architecture.
    """
    def __init__(self, output_dim):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace = True),

            nn.MaxPool2d(kernel_size = 2),

            nn.Conv2d(64, 128, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace = True),

            nn.MaxPool2d(kernel_size = 2),

            nn.Conv2d(128, 256, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace = True),

            nn.Conv2d(256, 256, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace = True),

            nn.MaxPool2d(kernel_size = 2),

            nn.Conv2d(256, 512, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace = True),

            nn.Conv2d(512, 512, kernel_size = 3, padding = 1),
```

```
            nn.BatchNorm2d(512),
            nn.ReLU(inplace = True),

            nn.MaxPool2d(kernel_size = 2),

            nn.Conv2d(512, 512, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace = True),

            nn.Conv2d(512, 512, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace = True),

            nn.MaxPool2d(kernel_size = 2)
        )

        self.avgpool = nn.AdaptiveAvgPool2d(7)

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace = True),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace = True),
            nn.Dropout(0.5),
            nn.Linear(4096, output_dim),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        h = x.view(x.shape[0], -1)
        x = self.classifier(h)
        return x
```

```
OUTPUT_DIM = 200

model = VGG(OUTPUT_DIM)

print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
  )
  (avgpool): AdaptiveAvgPool2d(output_size=7)
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=200, bias=True)
  )
)
```

Pre-trained Models

In this notebook we aren't actually going to use a VGG model with parameters that have been randomly initialized VGG model. We are going to be using a VGG model with *pre-trained* parameters. Using a pre-trained model involves initializing our model with parameters that have already been trained for a certain task - usually not the exact same task we are trying to do ourselves.

Torchvision has ways to easily download a pre-trained model. We simply import the `torchvision.models` package, specify which model we want to use and then pass the argument to load pretrained weights. We can see a list of all available pre-trained models provided by torchvision here.

Let's import a pre-trained VGG11 with batch normalization. The first time this code is run the pre-trained parameters will be downloaded and are around 140MB for VGG11 with batch normalization.

```
import torchvision.models as models

pretrained_model = models.vgg11_bn(weights=models.VGG11_BN_Weights.DEFA

print(pretrained_model)
```

```
Downloading: "https://download.pytorch.org/models/vgg11_bn-6002323d.pth"
100%|██████████| 507M/507M [00:05<00:00, 90.6MB/s]
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_r
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
```

```
      (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
      (20): ReLU(inplace=True)
      (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
      (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
      (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
      (24): ReLU(inplace=True)
      (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1
      (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_r
      (27): ReLU(inplace=True)
      (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
```

As we can see, the pre-trained model loaded is exactly the same as the one we have defined with one exception - the output of the final linear layer.

All of torchvision's pre-trained models are trained as image classification models on the ImageNet dataset. A dataset of 224x224 color images with 1000 classes, therefore the final layer will have a 1000 dimensional output.

We can get the last layer specifically by indexing into the `classifier` layer of the pre-trained model.

```
pretrained_model.classifier[-1]
```

```
Linear(in_features=4096, out_features=1000, bias=True)
```

As our dataset, CUB200, only has 200 classes then we want the last layer to have a 200 dimensional output.

We'll define a new final linear layer which has to have an input size equal to that of the layer we are replacing - as it's input will be the 4096 dimensional output from the previous linear layer in the classifier. The output of this linear layer will be 200 dimensions - as our dataset has 200 classes.

```
IN_FEATURES = pretrained_model.classifier[-1].in_features

final_fc = nn.Linear(IN_FEATURES, OUTPUT_DIM)
```

We can directly overwrite the previous linear layer with our new linear layer.

Note that our `final_fc` will be initialized randomly. It is the only part of our model with its parameters not pre-trained.

```
pretrained_model.classifier[-1] = final_fc
```

We can then print out the `classifier` of our model to ensure the final linear layer now has an output dimension of 10.

```
print(pretrained_model.classifier)
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=200, bias=True)
)
```

```
model = pretrained_model
```

```
# Let's count the number of trainable model parameters
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad

print(f'The model has {count_parameters(model):,} trainable parameters'
```

```
The model has 129,591,240 trainable parameters
```

Instead of training all of the parameters we have loaded from a pre-trained model, we could instead only learn some of them and leave some "frozen" at their pre-trained values. As our model will then have less trainable parameters it will usually train faster and we can usually fit it on smaller GPUs.

We aren't going to freeze any parameters in this notebook, but if we wanted to freeze the `features` layer then we could do that with:

```
for parameter in model.features.parameters():
    parameter.requires_grad = False
```

We could also freeze the `classifier` layer, however we always want to train the last layer as this is what we have initialized randomly and needs to be trained. Freezing all but the last layer in the `classifier` can be done with:

```
for parameter in model.classifier[:-1].parameters():
    parameter.requires_grad = False
```

Training the Model

Generally when using a pre-trained model the learning rate used will be considerably lower.

First, we'll set up the optimizer with the initial learning rate that is much lower than we expect to use. Then we define the `device` to place our model on our GPU, if we have one. Next we define the `criterion` (loss function) and place the model on our device.

```
LR = 1e-3

optimizer = optim.Adam(model.parameters(), lr = LR)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

criterion = nn.CrossEntropyLoss()

model = model.to(device)
```

We can now create an optimizer with our intial learning rate and using discriminative fine-tuning.

The concept behind descriminative fine-tuning is that we use different learning rates for different layers in our models. The hypothesis is that early layers in a neural network learn to extract more general features, whilst later layers learn to extract more task specific features. If this is true, then the general features extracted by the early layers should be useful for any task, and we should change the pre-trained weights of them by a very small amount - if at all.

Note: discriminative fine-tuning should only be used when performing transfer learning from a pre-trained model. It is typically not necessary to use it when training a model from randomly initialized weights.

PyTorch allows us to set different learning rate values per parameter in our model. This is done by passing a list of dictionaries to the optimizer. Each dictionary should state the parameters (`'params'`) and also any other arguments that will override those given directly to the optimizer.

Here, instead of using a different learning rate for every single layer, we have split the parameters into two "groups": `features`, which contains all of the convolutional layers; and `classifier`, which contains all of the linear layers. `classifier` will be using the `LR` given directly to the optimizer and `features` will be using `LR / 10`, as specified in the first dictionary. Thus, our convolutional layers have a learning rate 10x less than the linear layers.

```
params = [
          {'params': model.features.parameters(), 'lr': LR / 10},
          {'params': model.classifier.parameters()}
         ]

optimizer = optim.Adam(params, lr = LR)
```

Now all of the set-up is done, the rest of the notebook is pretty standard from here on out.

We create a function to calculate accuracy...

```
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim = True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

...create a function that implements a training loop...

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in iterator:

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

...create a function that performs an evaluation loop...

```
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)
            y = y.to(device)

            y_pred = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

...and a helper function to tell us how long an epoch takes.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Finally, we train our model.

As our images have been resized to be significantly larger and our model has significantly more parameters training takes considerably longer. However, when performing transfer learning we usually train for significantly less epochs and are still able to achieve much higher accuracy than before.

```
EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(EPOCHS):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, train_iterator, optimizer, cri
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion,

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut4-model.pt')

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:
```

```
Epoch: 01 | Epoch Time: 3m 1s
        Train Loss: 1.108 | Train Acc: 68.50%
         Val. Loss: 1.640 |  Val. Acc: 56.70%
Epoch: 02 | Epoch Time: 3m 2s
        Train Loss: 1.027 | Train Acc: 70.14%
         Val. Loss: 1.526 |  Val. Acc: 61.52%
Epoch: 03 | Epoch Time: 2m 58s
        Train Loss: 0.999 | Train Acc: 71.47%
         Val. Loss: 1.503 |  Val. Acc: 63.82%
Epoch: 04 | Epoch Time: 2m 24s
        Train Loss: 0.890 | Train Acc: 74.53%
         Val. Loss: 1.650 |  Val. Acc: 59.61%
Epoch: 05 | Epoch Time: 2m 24s
        Train Loss: 0.858 | Train Acc: 74.96%
         Val. Loss: 1.606 |  Val. Acc: 62.11%
```

```
model.load_state_dict(torch.load('tut4-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion, device)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 1.477 | Test Acc: 61.98%
```

This is an optional part which only serve as a reference for your future work

We'll examine the model now. First, we'll get the predictions for each of the examples in the test set...

```
def get_predictions(model, iterator):

    model.eval()

    images = []
    labels = []
    probs = []

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)

            y_pred = model(x)

            y_prob = F.softmax(y_pred, dim = -1)
            top_pred = y_prob.argmax(1, keepdim = True)

            images.append(x.cpu())
            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim = 0)
    labels = torch.cat(labels, dim = 0)
    probs = torch.cat(probs, dim = 0)

    return images, labels, probs
```

```
images, labels, probs = get_predictions(model, test_iterator)
```

...then get the predicted labels for each image...

```
pred_labels = torch.argmax(probs, 1)
```

We'll then find out which predictions were incorrect and then sort these incorrect predictions by how confident our model was.

```
corrects = torch.eq(labels, pred_labels)
```

```
incorrect_examples = []

for image, label, prob, correct in zip(images, labels, probs, corrects)
    if not correct:
        incorrect_examples.append((image, label, prob))

incorrect_examples.sort(reverse = True, key = lambda x: torch.max(x[2],
```

We can then plot these most confident incorrect predictions.

```
def normalize_image(image):
    image_min = image.min()
    image_max = image.max()
    image.clamp_(min = image_min, max = image_max)
    image.add_(-image_min).div_(image_max - image_min + 1e-5)
    return image
```

```
def plot_most_incorrect(incorrect, classes, n_images, normalize = True)

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize = (25, 20))

    for i in range(rows*cols):

        ax = fig.add_subplot(rows, cols, i+1)

        image, true_label, probs = incorrect[i]
        image = image.permute(1, 2, 0)
        true_prob = probs[true_label]
        incorrect_prob, incorrect_label = torch.max(probs, dim = 0)
        true_class = classes[true_label]
        incorrect_class = classes[incorrect_label]

        if normalize:
            image = normalize_image(image)

        ax.imshow(image.cpu().numpy())
        ax.set_title(f'true label: {true_class} ({true_prob:.3f})\n' \
                     f'pred label: {incorrect_class} ({incorrect_prob:.
        ax.axis('off')

    fig.subplots_adjust(hspace = 0.4)
```

```
N_IMAGES = 24

plot_most_incorrect(incorrect_examples, classes, N_IMAGES)
```