Building and training a model

Practical Application: Addressing a Classification and Segmentation Challenge

1
Deep Learning workflow

1.1 Example of PyTorch workflow

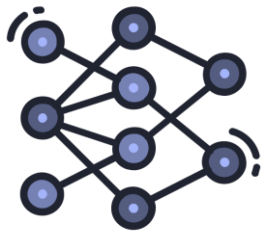
What we are going to cover:

In this bootcamp, we will explore a standard PyTorch workflow, which can be adapted as necessary while following the main outline of steps. Steps 1-3 will be improved through experimentation until we get the desired results in step 4.

1. Getting and preparing data	2. Building or picking a pretrained model	3. Training	4. Evaluation	5. Deployment
▪ Collect the (labeled) data ▪ Split them into a training /testing set ▪ Apply transforms	2 options: ▪ Train a model from scratch ▪ Transfer learning using a pretrained model	▪ Pick a loss function and optimizer ▪ Build a training loop ▪ Fit the model to the training set	Calculate performance metrics on the testing set such as accuracy, recall and precision to evaluate the model	Embeb the model wherever you need (desktop apps, enterprise systems…) and use it to make predictions

1.1 Example of PyTorch workflow

In this module we will explore the 2º and 3rd phase (Building and training a model from scratch)



Building or picking a pretrained model

Training

1. <u>Model Architecture Design:</u>

Select or design an appropriate model architecture that suits your problem domain.

<u>2. Prepare the loss function, optimizer and training loop</u>:

- **<u>Loss Function Selection</u>**: Determine the appropriate loss function that measures the discrepancy between the predicted outputs and the actual targets.
- **<u>Optimization Algorithm:</u>** Select an optimization algorithm that updates the model's parameters iteratively to minimize the loss function.
- **<u>Training Loop Creation:</u>** Set up a loop that trains the model using batches of data.

<u>3. Hyperparameter Tuning</u>:

Adjust the hyperparameters of the model and training loop, such as learning rate, batch size, and number of training epochs, to optimize the model's performance.

2
Build the neural network

2.1 Building a baseline model

Neural networks comprise of layers/modules that perform operations on data. The **torch.nn** namespace provides all the building blocks you need to build your own neural network. Let's build a baseline model by subclassing **nn.Module**. For that, we have to implement this two functions:

- The **__init__** method is used to initialize the different layers and components of the neural network. Additionally, model-specific settings, such as the number of hidden units and filter sizes, can be specified.

- The **forward** method defines how the input data flows through the layers to produce the final output. It receives the input data as a parameter and returns the output of the neural network.

```
class ClassificationModel(nn.Module):
    def __init__(self, output_shape:int):
        super().__init__()
        self.layer_stack = nn.Sequential(
            nn.Flatten(),
            nn.Linear(3*224*224, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, output_shape),
        )

    def forward(self, x):
        return self.layer_stack(x)
```

nn.Sequential is an ordered container of modules. The data is passed through all the modules in the same order as defined.

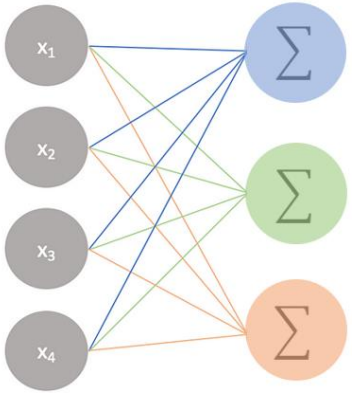
In this scenario, the neural network is designed to process RGB images with a resolution of 224x224. However, we have the option to customize this resolution by passing it as a parameter during initialization.

When we create an instance of our ClassificationModel, we will specify the number of outputs based on the number of classes we want to classify.

2.1 Building a baseline model

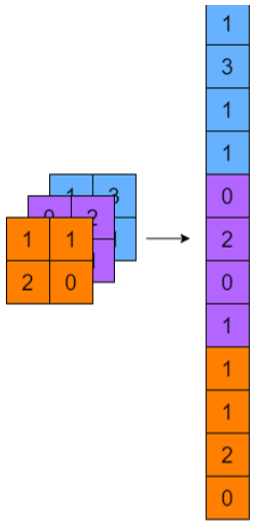
```python
class ClassificationModel(nn.Module):
    def __init__(self, output_shape:int):
        super().__init__()
        self.layer_stack = nn.Sequential(
            nn.Flatten(),
            nn.Linear(3*224*224, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, output_shape),
        )

    def forward(self, x):
        return self.layer_stack(x)
```

nn.Linear is a module that applies a linear transformation on the input using its stored weights and biases.

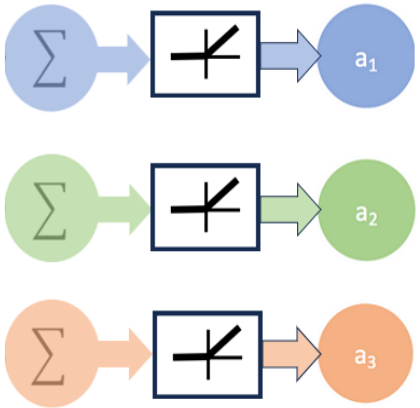


$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix}$$

nn.Flatten takes a multidimensional tensor as input and converts it into a one-dimensional tensor. In this case, it converts each 3D 224x224x3 image into a contiguous array of 150.528 pixel values.



The non-linear activations are applied after linear transformations to help neural networks learn a wide variety of phenomena. In this model, we use **nn.ReLU** between our linear layers, but there are other activations to introduce non-linearity in your model.



$$\begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow{activation} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

2.2 Building a Convolutional Neural Network (CNN)

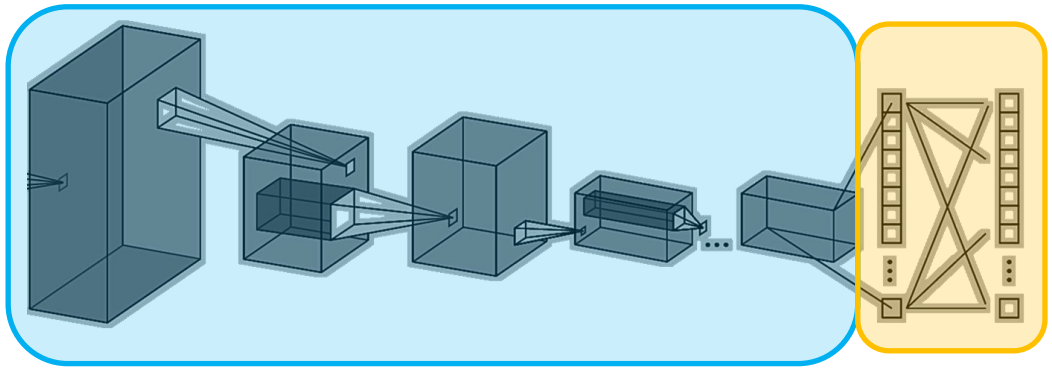
```python
class CNNClassifier(nn.Module):
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
        super().__init__()
        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels = input_shape,
                      out_channels = hidden_units,
                      kernel_size = 3,
                      stride = 1,
                      padding = 1),
            nn.ReLU(),
            nn.Conv2d(in_channels = hidden_units,
                      out_channels = hidden_units,
                      kernel_size = 3,
                      stride = 1,
                      padding = 1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2,
                         stride = 2)
        )
        self.block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=hidden_units*7*7,
                      out_features=output_shape)
        )

    def forward(self, x: torch.Tensor):
        x = self.block_1(x)
        x = self.block_2(x)
        x = self.classifier(x)
        return x
```

Now, we are going to create a Convolutional Neural Network (CNN or ConvNet), which are known for their capabilities to find patterns in visual data. Keep in mind that we covered the CNN architecture in the first module of the bootcamp.

The model showed is known as TinyVGG from the CNN Explainer website. It follows the typical structure of a CNN:

[Convolutional layer -> Activation layer -> Pooling layer] -> **Fully Connected layer**

These components can be flexibly scaled up and repeated multiple times.

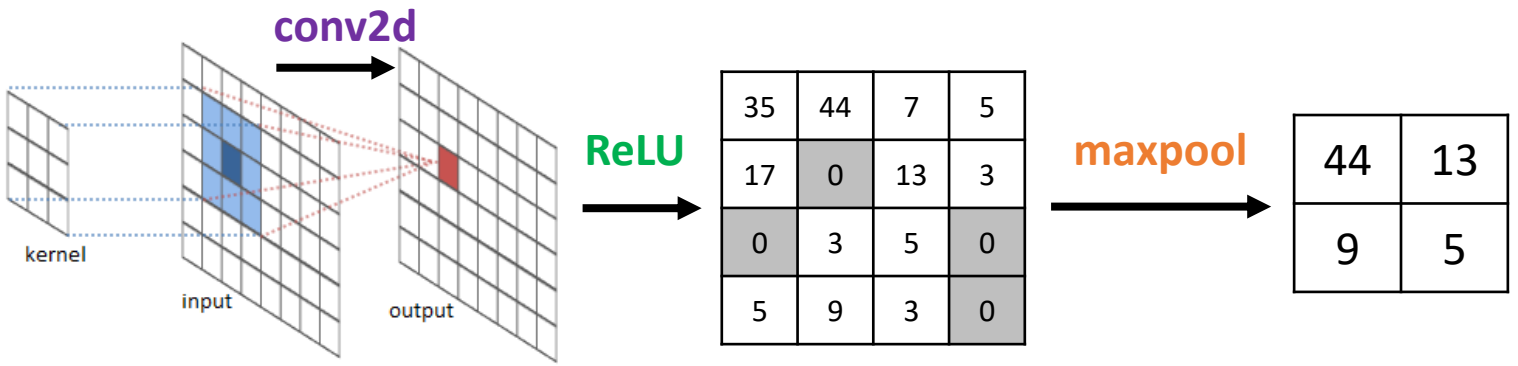


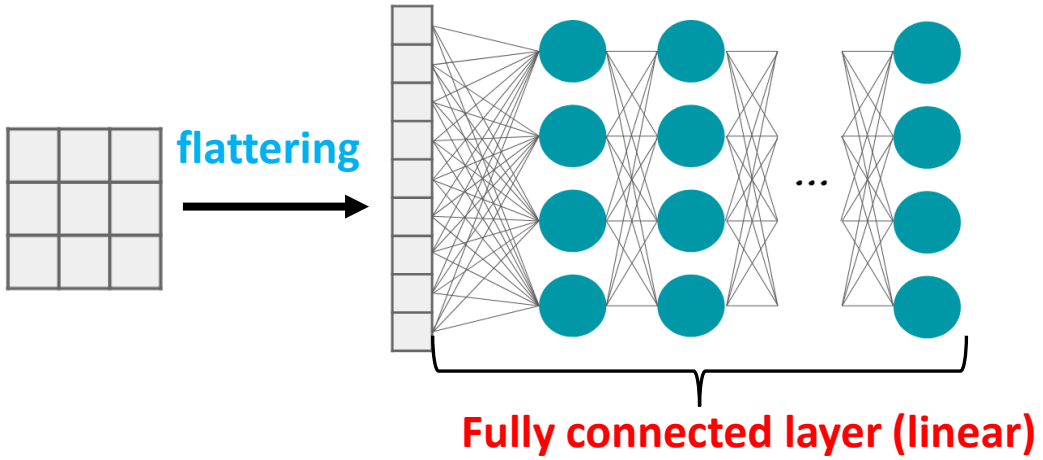
2.2 Building a Convolutional Neural Network (CNN)

```
self.block_1 = nn.Sequential(
    nn.Conv2d(in_channels = input_shape,
              out_channels = hidden_units,
              kernel_size = 3,
              stride = 1,
              padding = 1),
    nn.ReLU(),
    nn.Conv2d(in_channels = hidden_units,
              out_channels = hidden_units,
              kernel_size = 3,
              stride = 1,
              padding = 1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size = 2,
                 stride = 2)
)
```

```
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(in_features = hidden_units*7*7,
              out_features = output_shape)
)
```

To learn more about the mentioned functions, you can visit the Pyorch documentation: nn.conv2, nn.ReLU(), nn.MaxPool2d(), nn.Flatten(), nn.Linear().





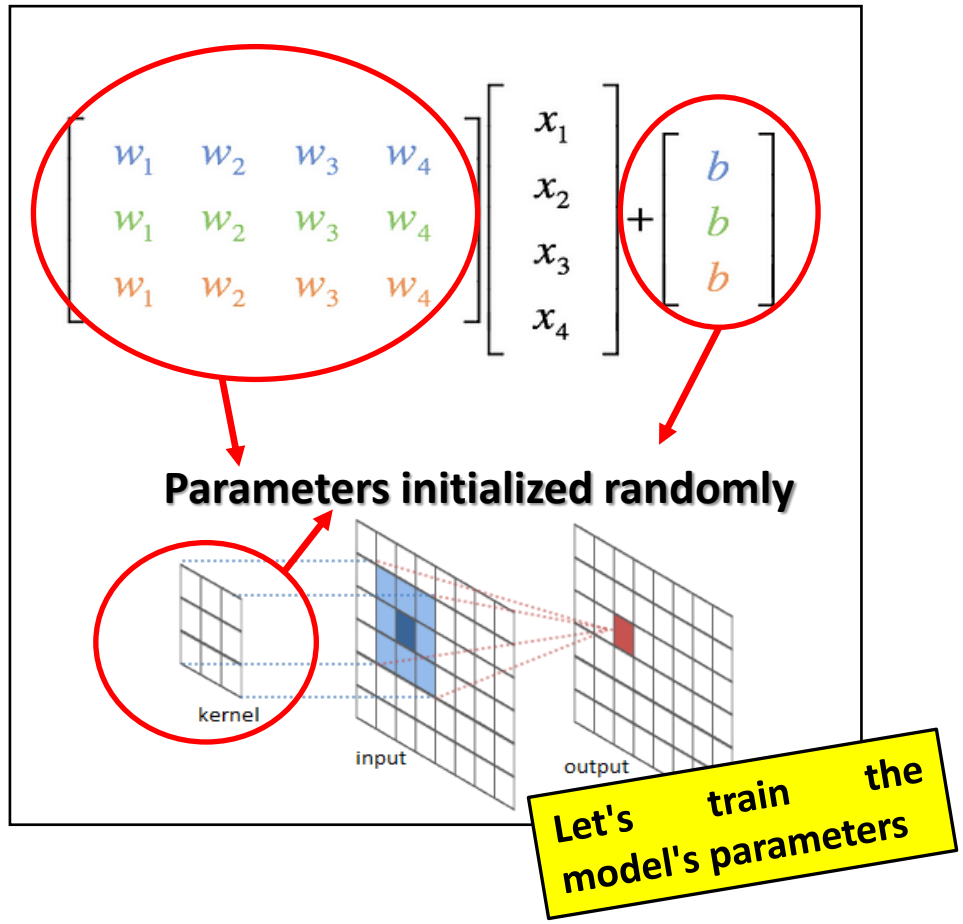
2.3 Model parameters

To begin using our neural network, we create an instance of our ClassificationModel and then transfer it to the device:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
class_names = ['airplanes','cars','ship'] # 3 classes

model = ClassificationModel(len(class_names))
model.to(device)
```

In the notebooks associated with these slides, we will create and train a model on our datasets for a multiclass classification problem.

At this stage, we have defined the model's architecture, including the layers and their connections. However, the model's parameters have been initialized randomly. This means that the weights, biases, or kernel values start with arbitrary initial values. The next step is to train the models using the data we prepared in section 2.



3
Training loop

3.1 Backpropagation algorithm

The backpropagation algorithm enables a neural network to learn from labeled data and adjust its internal parameters to make accurate predictions. The process of training a neural network through backpropagation involves the following steps:

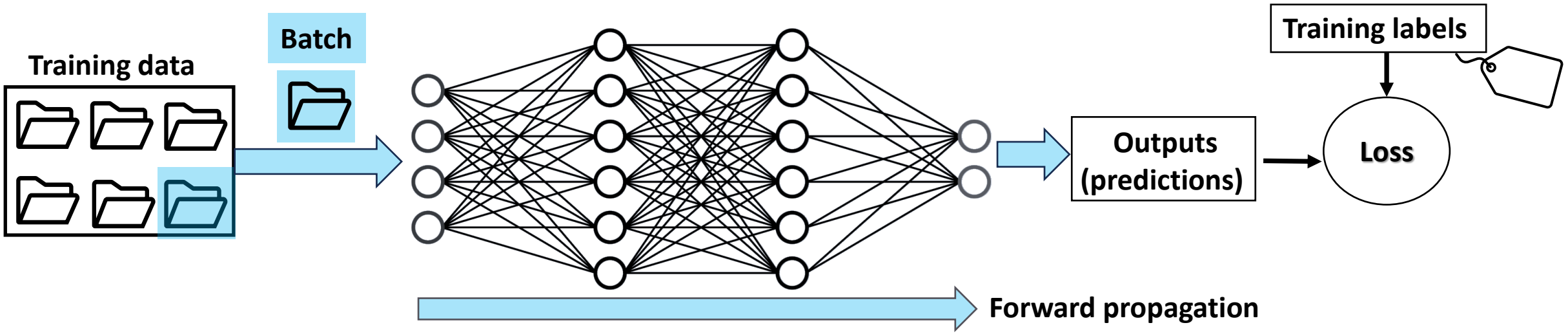
Step 1. Initialization: The network parameters are initialized with random values. This step is performed only once in the training phase.

Step 2. Forward propagation: A mini-batch of our training dataset is propagated from the first layer to the last, obtaining model predictions at its output.

```
outputs = model(inputs)
```

Step 3. Loss Computation: Calculate the loss between the predicted outputs of the network and the actual targets using a suitable loss function.

```
loss = criterion(outputs, labels)
```



3.1 Backpropagation algorithm

Step 4. Backpropagation:

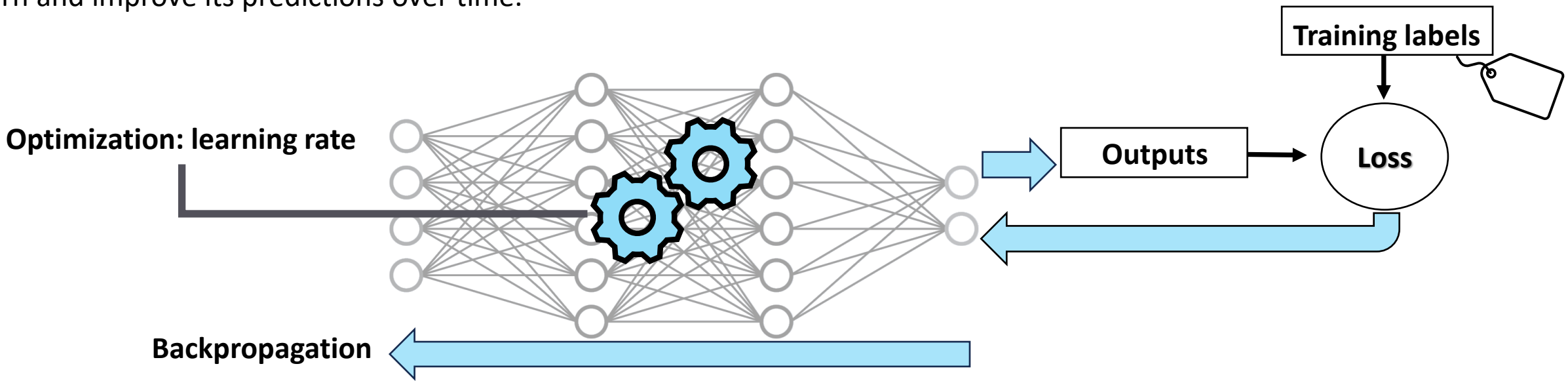
The error is propagated from the final layer back to the initial layer of the neural network. During backpropagation, the partial derivatives of the loss function with respect to each network parameter are calculated, forming the gradient vector. This gradient vector represents the direction and magnitude of the adjustments needed to minimize the loss function.

```
loss.backward()
```

Step 5. Optimization:

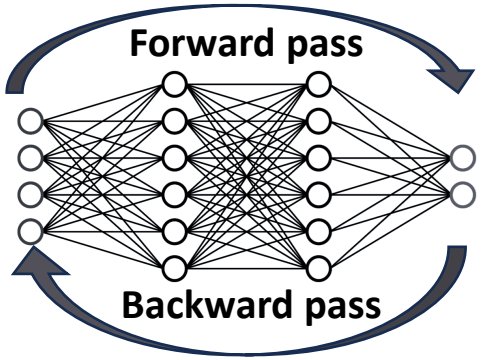
The network parameters are adjusted using the gradient descent method with a specific learning rate. By iteratively updating the parameters in the direction opposite to the gradient, the network gradually minimizes the loss function value. This process allows the network to learn and improve its predictions over time.

```
optimizer.step()
```

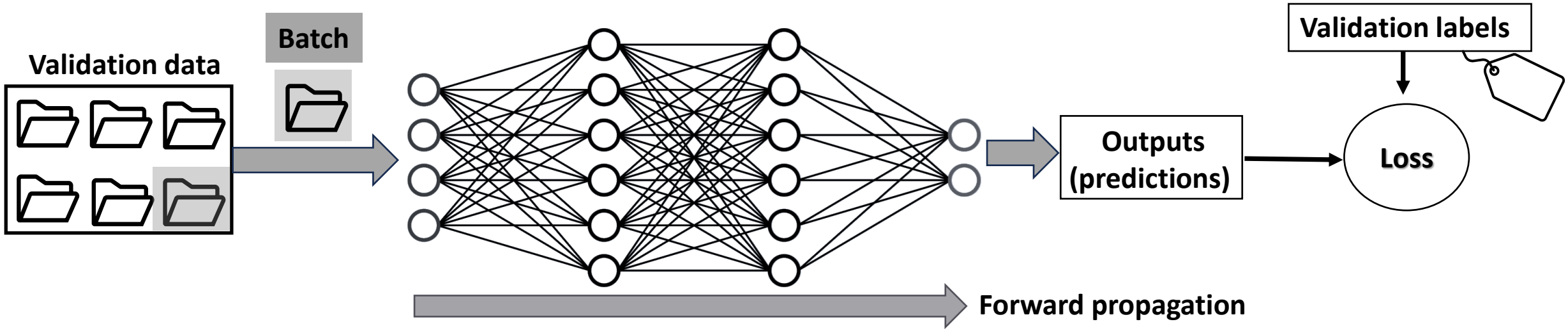


3.1 Backpropagation algorithm

We then start again from step 2, until the entire set of input data passes through the network, completing 1 epoch or iteration. The training process will end when the number of epochs previously established by the programmer have been performed, making the same input data pass through the network several times.



If a validation dataset is present, the network's performance is evaluated by assessing its predictions on it after each training iteration. This allows us to estimate the model's performance during training, providing us with information that can help us to adjust the hyperparameters of the network. Be aware: **Backpropagation is not used on the validation dataset**; it is solely employed during the training phase of a neural network to calculate gradients and update parameters. The validation dataset is used separately to evaluate the performance and generalization of the trained network.



3.2 Hyperparameters

The hyperparameters are values provided by the programmers that allow to control the optimization process. Unlike the parameters of the model that are learned during training, hyperparameters are predetermined and not updated based on the data. Some examples of hyperparameters include:

- **Number of epochs:** It specifies the number of times the entire dataset is passed forward and backward through the neural network during the training process.

```
num_epochs = 50
for epoch in range(num_epochs):
    # Training loop…
```

- **Learning Rate:** The learning rate defines the speed at which the neural network will learn. It determines the step size for parameter updates during optimization, influencing how much the network's parameters are adjusted based on the calculated gradients.

```
optimizer = optim.SGD(
params = model.parameters(),
lr = 0.001)
```

- **Batch Size:** We previously discussed the concept of batch size during the data preparation module. It determines the number of samples to be propagated through the network before its parameters are updated.

```
Train_dataloader =
torch.utils.data.DataLoader(
    train_dataset,
    batch_size = 32,
    shuffle = True
)
```

3.3 PyTorch modules

Below are the fundamental PyTorch modules commonly utilized for constructing neural networks and developing training algorithms:

torch.nn	torch.nn.Module	torch.optim
The module includes a wide range of pre-defined layers, loss functions (criterions), and activation functions that are essential for constructing complex neural architectures.	Base class for all neural network modules. When constructing a neural network in PyTorch, it is important to create models that inherit from nn.Module and include the implementation of a forward() method.	It provides a collection of optimization algorithms that facilitate the training of neural network models, including popular ones like Stochastic Gradient Descent (SGD), Adam, and RMSprop, among others.

Once the architecture of our model is defined, we need to choose an appropriate cost function (or criterion) that suits the problem and an optimizer. Here's an example of the code where we define the SGD optimizer and use the cross-entropy loss function, typically employed for multi-class problems:

```
criterion = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(params = model.parameters(), lr = 0.001)
```

Next, we move on to constructing our training loop, and we'll explore two different implementations. In the first example, we'll exclusively use the training data, while in the second one, we'll see how to define the loop when a validation subset is present.

3.4 Build a Pytorch training loop: Using only the training set

```
model.train()
for epoch in range(epochs):
    for batch, (inputs, labels) in enumerate(train_dataloader):
        # Send data to GPU if available
        inputs, labels = inputs.to(device), labels.to(device)
        # 1. Forward pass
        outputs = model(inputs)
        # 2. Calculate loss (per batch)
        loss = criterion(outputs, labels)
        # 3. Loss backward
        loss.backward()
        # 4. Optimizer step
        optimizer.step()
        # 5. Optimizer zero grad
        optimizer.zero_grad()
```

Set model in training mode (which is the default state of the model)

Pass the data through the model for a number of epochs

Iterate through the dataloader to process the data in batches

For each batch of inputs and labels, we move them to the specified device

Pass the data through the model, using its forward() method

Compute the loss value, which measures the discrepancy between the model's predictions and the actual values.

Compute the gradients of the model's parameters with 'requires_grad=True' setting

Update the model's parameters based on the computed gradients from backpropagation.

Clear the gradients of the model's parameters to avoid accumulation from previous iterations

3.4 Build a Pytorch training loop: Training and validation sets

```python
for epoch in range(num_epochs):
    for phase in ['train', 'val']:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            torch.save(model.state_dict(), best_model_params_path)
```

In this version, **every epoch** includes both a **training and validation phase**. The model's mode is adjusted accordingly, either set to training or validation mode, depending on the current phase.

In each phase, we **loop through the data**. For every batch of inputs and labels, we transfer them to the designated device for processing.

Forward pass: Pass the data through the model and compute the loss value. Enable **gradient tracking** (requires_grad = True) **only during the training phase**.

Perform **backpropagation** and **optimization** only when the model is in the training phase. This ensures that gradients are computed and model parameters are updated solely during training.

Compute statistics that can be used to track the learning process, such as printing them during training or using them for plotting to visualize the model's performance over time.

Save the model's state dictionary if the accuracy on the **validation dataset** is better than the previous best accuracy.

4
Dealing with underfitting and overfitting

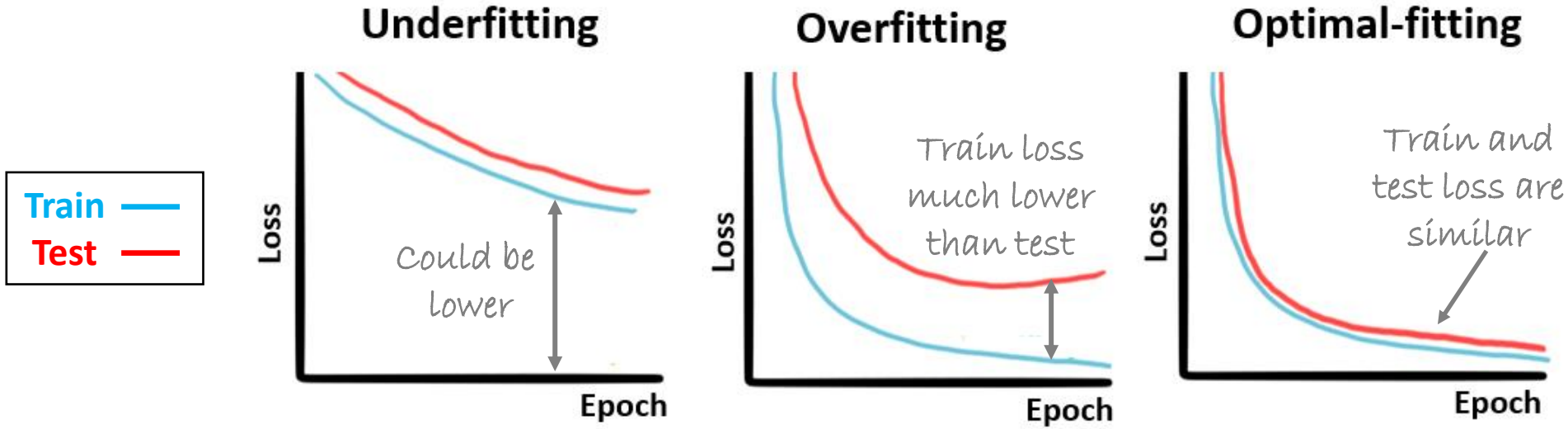
4.1 Underfitting and overfitting

There can be two situations where the neural network fails to generalize knowledge correctly:

- **Underfitting** occurs when the model is not sufficiently complex to accurately detect the relationships and features of the data, either due to a lack of data or erroneous or insufficient learning.

- **Overfitting** happens when the model fits too closely to the training data, learning its peculiarities, and making it unable to perform well with new input data.

To monitor the learning process, we will plot the changes in the cost function values during training. This visualization can help us identify signs of underfitting or overfitting:



4.2 Methods to reduce overfitting

DATA

Get more data: By obtaining more diverse and representative data, the model can better generalize and capture underlying patterns. You need to have enough data that represents the diversity and complexity of the problem you are trying to solve.

Data augmentation: Increase the diversity of your training dataset without collecting more data.

MODEL ARCHITECTURE

Simplify your model: Consider reducing the complexity of your model by decreasing the number of layers, neurons, or using simpler architectures. This can help prevent the model from memorizing noise or specific patterns in the training data.

Use dropout: Introduce dropout layers during training to randomly deactivate neurons, preventing over-reliance on specific features and promoting robustness.

Transfer Learning: Leverage the knowledge and patterns learned from solving one task and apply them to improve performance on a different but related task.

TRAINING ALGORITHM

Use learning rate decay: Decrease the learning rate as a model trains to facilitate better model convergence.

Use early stopping: Stop the training process once the model's performance on a validation set starts to degrade.

Apply regularization techniques: Use methods like L1 or L2 regularization to penalize large weights and prevent overfitting.

4.3 Methods to reduce underfitting

DATA

Use appropriate data: To prevent underfitting, you need to have relevant data that captures the features and patterns that are important for your model to learn. Therefore, you should always collect, clean, and preprocess your data carefully and check for any missing, noisy, or imbalanced data.

MODEL ARCHITECTURE

Add more layers/neurons to your model: Increase the model's capacity by adding additional layers, neurons, or employing a more complex architecture to better capture the underlying patterns in the data.

Transfer Learning: Reduce underfitting by leveraging knowledge from a pre-trained model on a source task. This approach provides a head start to the model's learning process on a target task with limited data, enabling it to generalize better

TRAINING ALGORITHM

Adjust learning rate: Modify the learning rate as needed during training. Decreasing the learning rate can help the model converge more gradually and accurately fit the training data, while increasing it can speed up convergence and improve performance on certain tasks.

Train for longer: Increase the number of training epochs to give the model more opportunities to learn from the data and fit the training set better.

5
Loading and saving models

5.1 Loading and Saving Models in PyTorch

In PyTorch, saving and loading a model is achieved using the **`torch.save()`** and **`torch.load()`** functions.

Saving/Loading model's state_dict (recommended):

When saving a model for inference, it is only necessary to save the trained model's learned parameters. In this case, when loading the model, we need to instantiate the model first and then load its state dictionary.

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

Create a model object with the architecture specified by TheModelClass.

Update the model's parameters with the trained weights stored in the file.

Saving/Loading the entire model:

You can also save and load the entire model, including the architecture and the state_dict:

Save:

```
torch.save(model, PATH)
```

Load:

```
model = torch.load(PATH)
model.eval()
```

To learn more about saving and loading PyTorch models, you can visit the Pyorch documentation

5.1 Loading and Saving Models in PyTorch

Saving & Loading Model Across Devices

If you intend to load the model on a different device (CPU or GPU) than the one used for saving it, you can easily achieve this by utilizing the map_location argument in the torch.load() function. For instance:

Save on GPU :

```
torch.save(model.state_dict(), PATH)
```

Load on CPU :

```
device = torch.device('cpu')
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location = device))
model.to(device)
```

Considerations:

- A common PyTorch convention is to save models using either a **.pt** or **.pth** file extension.

- Make sure to call **input = input.to(device)** on any input tensors that you feed to the model so that both the data and the model remain on the same device (CPU or GPU).

6
Classification

6.1 Classification tasks

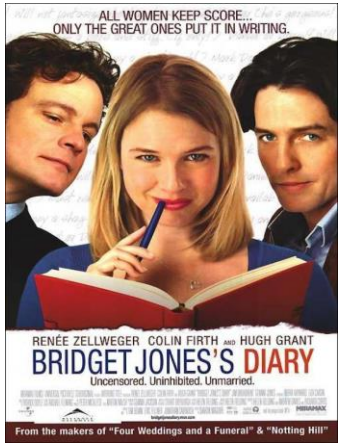
Binary Classification:
There are only two possible classes or labels to predict. The model's objective is to classify input data into one of the two categories.

Example:
Detecting if an X-ray image shows signs of a specific medical condition, such as pneumonia or not.

Labels:
Pneumonia
Not pneumonia

Multilabel Classification:
Each input sample can have multiple labels simultaneously. The model predicts a set of binary labels for each input rather than assigning a single class.

Example:
Categorizing movies into various genres. A single movie can fall under multiple genres simultaneously.



Labels:
Action
Comedy
Drama
Science Fiction
Romance
Horror
Fantasy

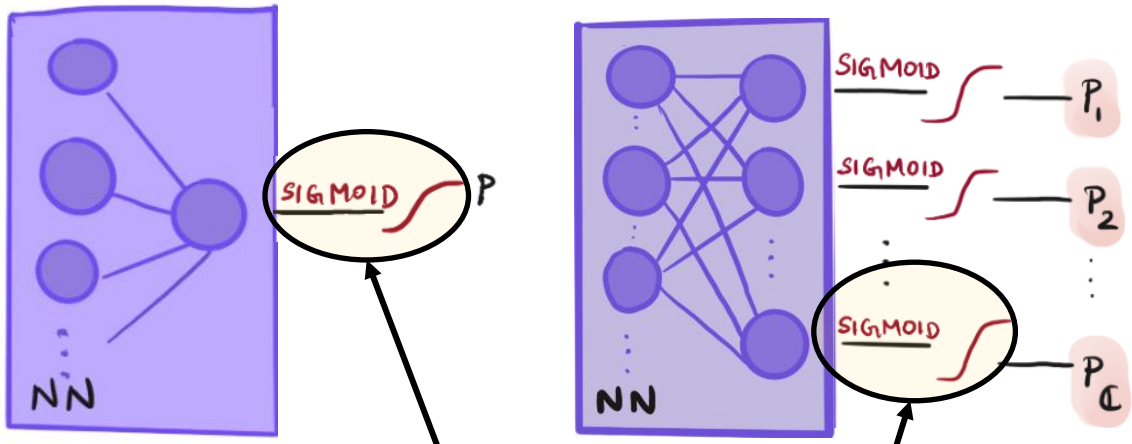
Multiclass Classification:
The model is trained to categorize input data into multiple exclusive classes, predicting the correct label among the options.

Example:
Classifying images of flowers into different species, such as roses, sunflowers, and daisies.

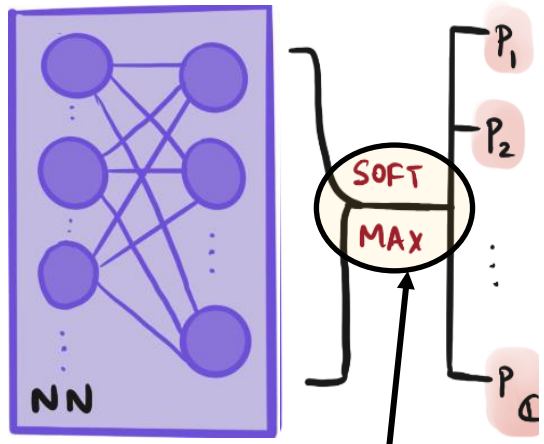
Labels:
Roses
Sunflowers
Daisies
Tulips
Lilies
Orchids

6.1 Classification tasks

	Binary	Multilabel	Multiclass
Outpt layer shape (out_features)	1	1 per class	1 per class
Output activation function	Sigmoid `torch.sigmoid()`	Sigmoid `torch.sigmoid()`	Softmax `torch.nn.softmax()`
Example of loss function	Binary Cross entropy `torch.nn.BCELoss()`	Binary Cross entropy `Torch.nn.BCELoss()`	Cross entropy `torch.nn.CrossEntropyLoss()`



Sigmoid function outputs values in the range [0,1], making it suitable for obtaining individual class probabilities.



The output of the softmax function can be interpreted as a vector of probabilities, indicating the likelihood of belonging to each class, with the probabilities for each class adding up to 1.

6.2 Inference example: using softmax function

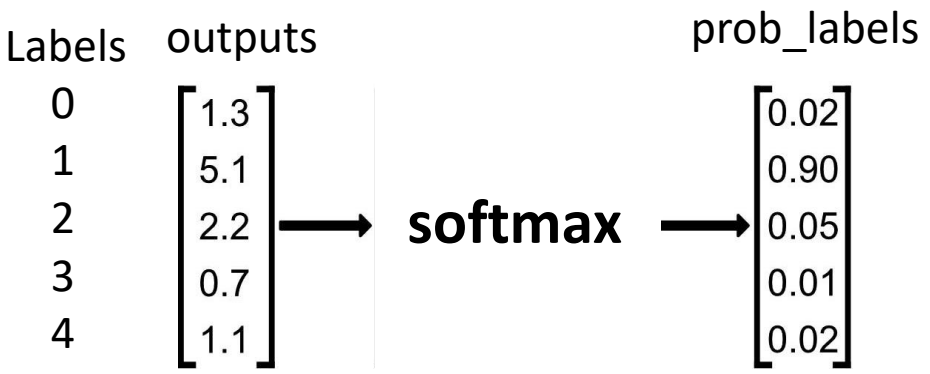
Remember our CNNClassiier model (defined in the 2.2 section in this document). The last linear layer of the neural network returns raw values in [- ∞, ∞]:

```
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(in_features = hidden_units*7*7,
              out_features = output_shape)
)
```

The final layer in our model outputs a specific number of values indicated by output_shape, each falling within the range[- ∞, ∞]

This values returned by the model are passed to the nn.Softmax module to be scaled to values [0, 1] representing the model's predicted probabilities for each class. *dim* parameter indicates the dimension along which the values must sum to 1.

```
outputs = model(input)
prob_labels = torch.softmax(outputs, dim = 1)
predicted_label = prob_labels.argmax(dim = 1)
```



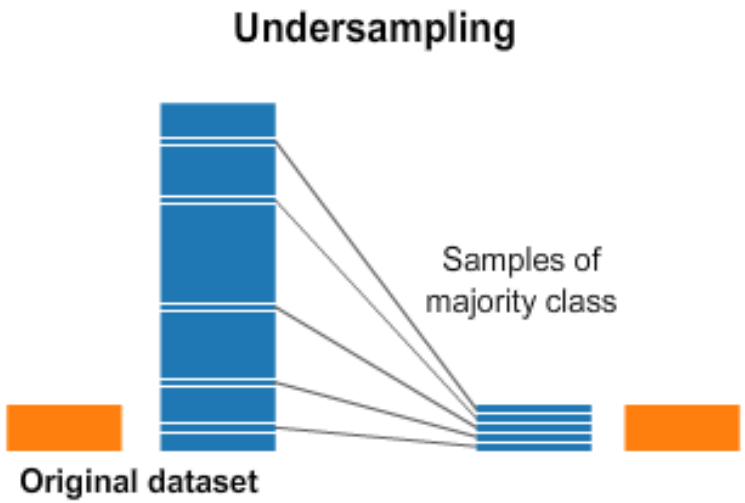
Calculate the predicted label by selecting the class with the highest probability from the prob_labels tensor.
In this case: predicted_label = 1

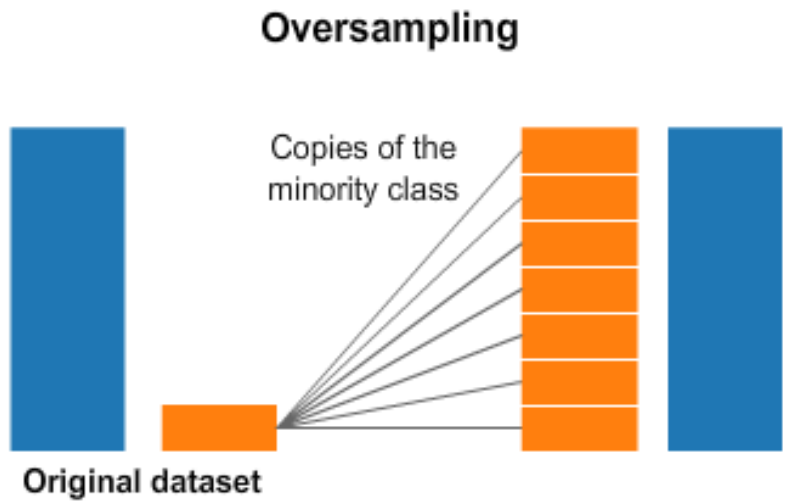
6.3 Imbalanced data handling

Class imbalance occurs when there is a significant disparity in the number of instances between different classes in a classification problem. It means that some classes may have a much larger or smaller representation in the dataset compared to others, potentially affecting the model's ability to learn from the data and make accurate predictions for underrepresented classes.

If dealing with imbalanced classes, use techniques like **class weighting, oversampling** or **undersampling** to balance the class distribution and improve the model's performance on minority classes.

It involves assigning higher weights to the minority class during model training to make it more influential in the learning process.



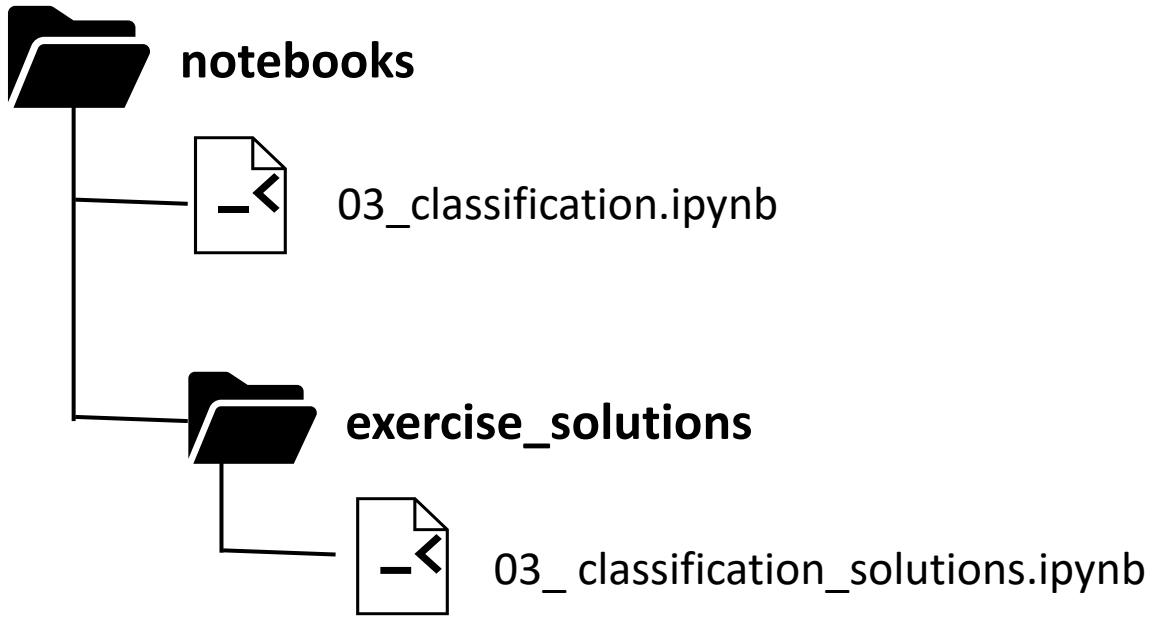


6.4 Code examples

Let's code!

Addressing a classification Challenge:

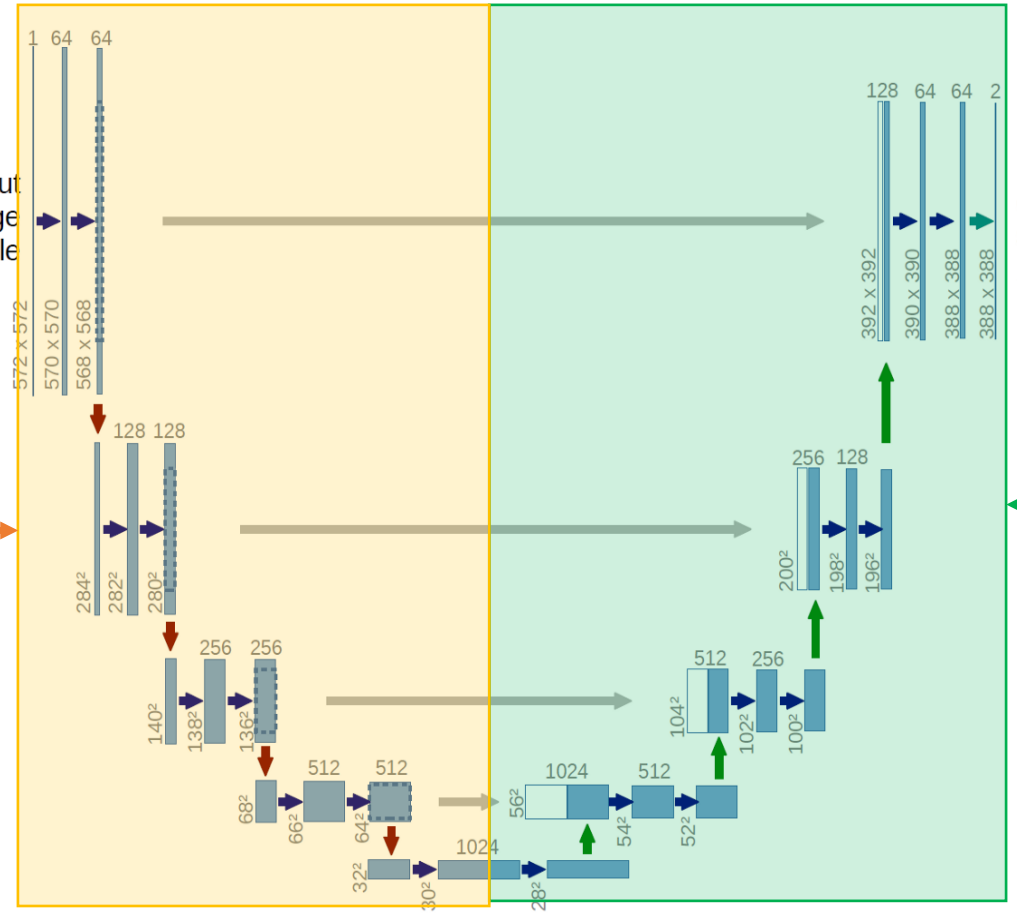
To address the multiclass image classification challenge, we will construct our own model, as we have previously seen, and train it using the datasets prepared in the prior module (section 02: data preprocessing). The main goal is to accurately classify each image into its appropriate category.

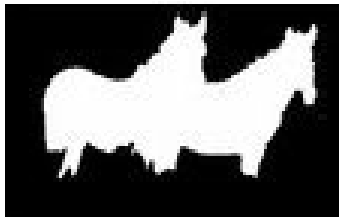


7
Segmentation

7.1 Semantic segmentaion task: U-Net network

So far, we have been working with classification problems. Now, we will address a semantic segmentation problem where the objective is to classify each pixel in an image into one of multiple predefined categories or classes. To achieve this, we will construct a U-Net architecture, which can be divided into an encoder-decoder path or contracting-expansive path equivalently.



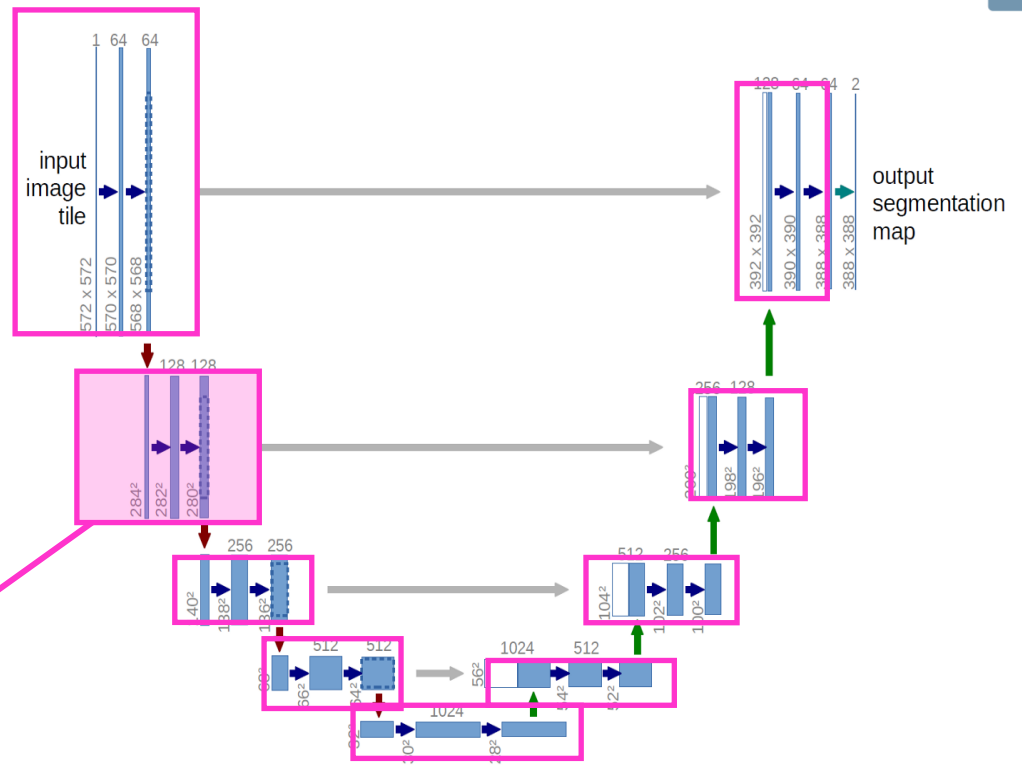


Encoder (left side): Uses a typical CNN structure. This part extracts relevant features from image.

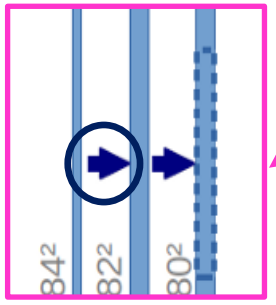
Decoder (right side): This part builds segmentation mask from features.

7.2 U-Net network: Define the basic convolutional block

```python
class conv_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size=3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, inputs):
        x = self.conv1(inputs)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        return x
```



Representation



Description → conv 3x3, ReLU **(x2)**

Code

```
x = self.conv1(inputs)
x = self.relu(x)
```

(x2)

7.2 U-Net network: Define the encoder and decoder

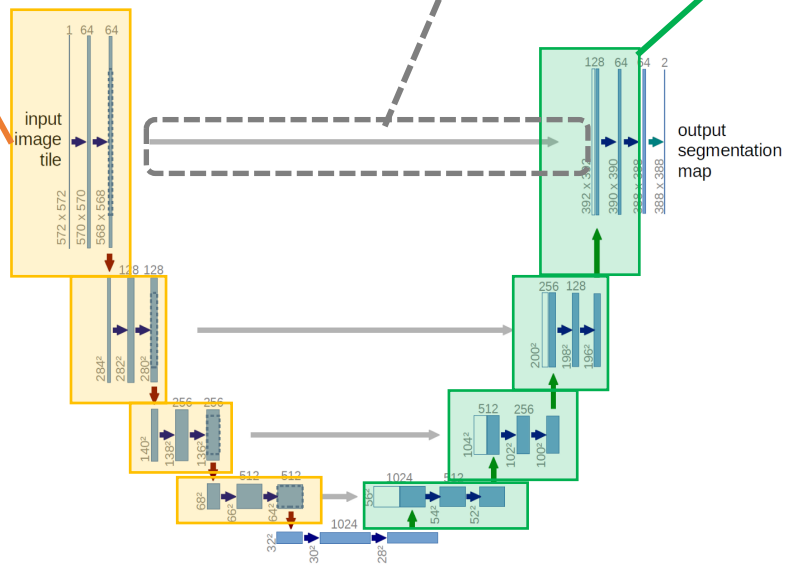
```python
class encoder_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv = conv_block(in_c, out_c)
        self.pool = nn.MaxPool2d((2, 2))
    def forward(self, inputs):
        x = self.conv(inputs)
        p = self.pool(x)
        return x, p
```

```python
class decoder_block(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_c, out_c, kernel_size=2,
                                     stride=2, padding=0)
        self.conv = conv_block(out_c + out_c, out_c)
    def forward(self, inputs, skip):
        x = self.up(inputs)
        x = torch.cat([x, skip], axis=1)
        x = self.conv(x)
        return x
```

Convolutional block

+

max pool 2x2



up-conv 2x2

+

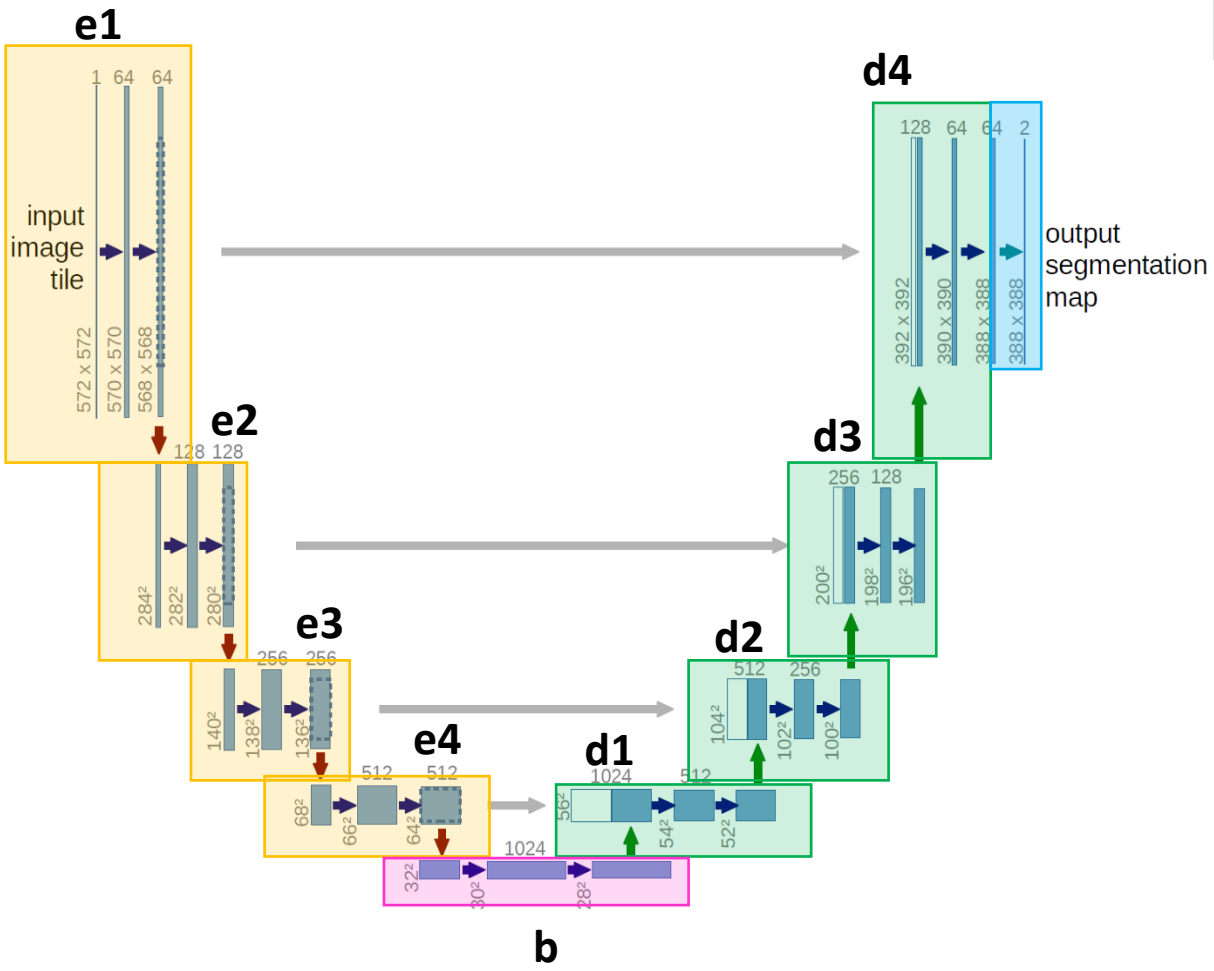
copy and crop

+

Convolutional block

7.2 U-Net network: Build the model

After defining the building blocks of our architecture, we can proceed to construct the class responsible for implementing the U-Net network.




```python
class UNet(nn.Module):
    def __init__(self, output_channels, input_channels = 3):
        super().__init__()
        # Encoder
        self.e1 = encoder_block(input_channels, 64)
        self.e2 = encoder_block(64, 128)
        self.e3 = encoder_block(128, 256)
        self.e4 = encoder_block(256, 512)

        # Bottleneck
        self.b = conv_block(512, 1024)

        # Decoder
        self.d1 = decoder_block(1024, 512)
        self.d2 = decoder_block(512, 256)
        self.d3 = decoder_block(256, 128)
        self.d4 = decoder_block(128, 64)

        # Output
        self.outputs = nn.Conv2d(64, output_channels, kernel_size = 1, padding = 0)

    def forward(self, inputs):
        # Encoder
        s1, p1 = self.e1(inputs)
        s2, p2 = self.e2(p1)
        s3, p3 = self.e3(p2)
        s4, p4 = self.e4(p3)

        # Bottleneck
        b = self.b(p4)

        # Decoder
        d1 = self.d1(b, s4)
        d2 = self.d2(d1, s3)
        d3 = self.d3(d2, s2)
        d4 = self.d4(d3, s1)
        return self.outputs(d4)
```

7.3 Code examples

Let's code!

Addressing a Segmentation Challenge:

To practice image segmentation using convolutional networks, we will implement the U-Net network described in the previous slides. Subsequently, we will train it using a training loop similar to the one we utilized for classification tasks. For this specific project, we will generate a synthetic dataset that will be employed to train and validate our model in the segmentation of various geometric figures. The model will assign specific colors to the pixels of each type of figure; for example, triangles will be colored green, circles will be colored yellow, and so on.