In [1]:
```
%matplotlib inline
```

Modeling Hydraulic Throughput in Porous Media Using Convolutional Neural Networks

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Assignment 3 - Hydraulic Cross Section

June 2024

Ata Bekişoğlu - *1697048*

Kaggle Group Name: *abekis*

Table of Contents

Introduction

Modeling the laminar flow of a viscous fluid through a porous medium, such as a sponge, involves predicting hydraulic throughput from binary images representing the medium's cross-section. Each pixel in these images is either open (allowing fluid flow) or closed (impermeable). The goal is to quantify hydraulic throughput, a parameter independent of fluid properties, channel length, and pressure drop.

To address this, the analysis begins with ensuring dimensional homogeneity and scaling consistency. Symmetries in the problem are then explored and incorporated into convolutional neural networks (CNNs), chosen for their ability to capture spatial hierarchies in image data. Data augmentation and disambiguation techniques are used to improve model robustness and generalization. Three CNN architectures are trained: a simple model, and two models integrating symmetry properties. Their performance is evaluated across regular, augmented, and disambiguated datasets, leading to insights on the validity of these approaches.

Problem Description

The problem at hand is modeling the laminar flow of a viscous fluid through a porous medium (e.g., a sponge) within a long channel. The porous medium is represented by binary images where each pixel is either "open" (allowing fluid flow) or "closed" (impermeable to fluid flow). My main goal is to quantify the **hydraulic throughput** S, a geometric parameter that remains independent of the fluid properties channel length and pressure drop using the relation:

$$S = \frac{\mu L Q}{\Delta P} \tag{1}$$

Where Q is the volumetric flow rate, μ is the fluid viscosity, L is the channel length, and ΔP is the pressure drop. My task is to predict S from the binary cross-sectional images from the porous medium.

Making Sense of the Data

In this section, I aim to explore the dimensional properties of the data, generate visualizations to aid with the general understanding of the problem at hand and explore the statistical properties that might help build better neural networks.

The dataset consists of two `.npy` (NumPy) files with $N = 1660$ samples of 40×40 pixel area images and hydraulic throughput S targets.

In [2]:
```
import numpy as np

X = np.load("data/pub_input.npy")
y = np.load("data/pub_out.npy")
print(X.shape, y.shape)
```

```
(1660, 40, 40) (1660,)
```

Dimensionality, Homogeneity and Scaling

Dimensionality

Dimensional homogeneity ensures that the quantities are consistent in terms of their SI units. Here, I analyze the dimensionality of the problem, apply scaling, and ensure homogeneity to accurately predict hydraulic throughput. The variables in the problem have the following dimensions in SI units:

- Volumetric flow rate Q, is measured in cubic meters per second $[m^3 s^{-1}]$.
- Pressure drop ΔP, is measured in Pascals (Pa), also equivalent to $[kg \cdot m^{-1} \cdot s^{-2}]$.
- Channel length L, is measured in meters $[m]$.
- Fluid viscosity μ is measured in Pascal-seconds ($Pa \cdot s$) which is equivalent to $[kg \cdot m^{-1} \cdot s^{-1}]$.
- **Pixel area A is measured in square meters $[m^2]$**.
- **The hydraulic throughput S is calculated using (1) which gives**:

$$S = \frac{[kg \cdot m^{-1} \cdot s^{-1}] \cdot [m] \cdot [m^3 \cdot s^{-1}]}{[kg \cdot m^{-1} \cdot s^{-2}]} = [m^4] \tag{2}$$

Scaling

To make the fitting problem 1-degree homogeneous the target variable S is scaled by the square root to reduce its SI unit from m^4 to m^2. This ensures that the units of the target variable align with the input image units.

$$S_i' = \sqrt{S_i} \tag{3}$$

To apply this logic in Python:

In [3]:
```
y = np.sqrt(y)
```

Visualizing the Problem

To better understand the dataset, several visualizations are performed, including visualizing some of the binary cross-section images along with their corresponding hydraulic throughput values, distribution of S, and a density plot.

The basic properties of the hydraulic throughput:

In [4]:
```
print(f'Min: {np.min(y):0.2f}, \
        Max: {np.max(y):0.2f},\
        Mean: {np.mean(y):0.2f}, \
        Standard Deviation: {np.std(y):0.2f}, \
        Correlation Between Consecutive S: {np.corrcoef(y[:-1], y[1:])[0, 1]:0.2f}')
```

```
Min: 2.65,         Max: 554.82,        Mean: 110.99,         Standard Deviation: 144.51,         Correlation Between Consecutive S: 0.40
```

The hydraulic throughput values are spread out, with many values clustered around lower values indicated by the mean. The high standard deviation suggests substantial variability in the hydraulic throughput across different images. There is also a small amount of positive correlation between consecutive throughputs of cross-sections $(S_i - S_{i-1})$ meaning there is a small influence between consecutive cross-sections.

And, for the areas (total area per image):

In [5]:
```
total_area = X.sum(axis=(1, 2))
print(f'Min: {np.min(total_area):0.2f}, \
        Max: {np.max(total_area):0.2f},\
```

```
        Mean: {np.mean(total_area):0.2f}, \
        Standard Deviation: {np.std(total_area):0.2f}')
```

```
Min: 7.00,         Max: 1435.00,         Mean: 384.02,         Standard Deviation: 409.31
```

The total open area per image varies widely, indicating diverse cross-section porosity levels. The mean area of 384.02 pixels suggests that, on average, about half of the pixels in the images are open. The high standard deviation reflects significant heterogeneity in the distribution of open areas across the images.

Binary Cross-Section Images:

The first 5 examples from `X` are plotted and annotated with their corresponding hydraulic throughput S'. Cross-sections are denoted in CS.

In [6]:
```
import matplotlib.pyplot as plt

def plot_image(ax, image, index, add=None):
    ax.imshow(image, cmap='gray')
    ax.set_title(f"$ CS_{{{index + 1}}} $ {add or ''}\n $ S' = {y[index]:.2f} \quad [m^2] $ ")  # oh god
    ax.set_xlabel('$ x \ [m] $')
    ax.set_ylabel('$ x \ [m] $')

fig, axes = plt.subplots(1, 5, figsize=(15, 4))
for i in range(5):
    plot_image(axes[i], X[i], i)

plt.suptitle('Figure 1: Binary Cross-Sections with Corresponding Hydraulic Throughput Values', fontsize=15, style='italic')
plt.tight_layout()
plt.show()
```

Figure 1: Binary Cross-Sections with Corresponding Hydraulic Throughput Values

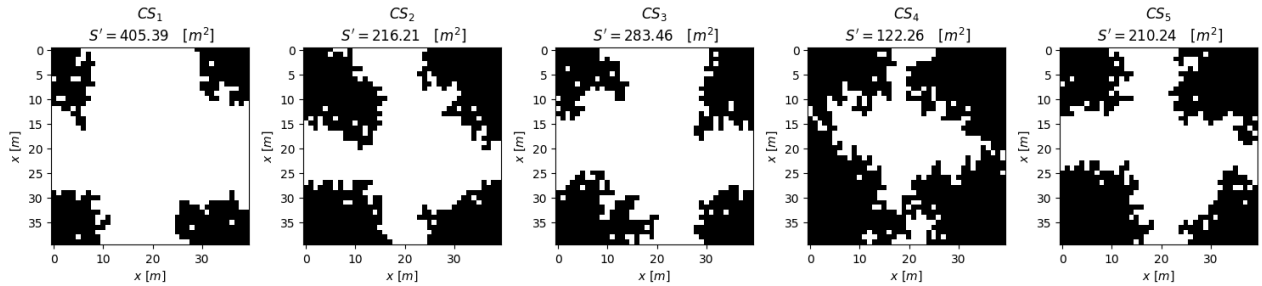


Figure 1 shows five examples of open (white) and closed (black) pixels affecting the hydraulic throughput S. Cross-sections with more continuous pathways for fluid flow tend to have higher S values, indicating greater hydraulic throughput.

Distribution of Hydraulic Throughput Values & Density Plot of Hydraulic Throughput vs. Number of Open Pixels:

A histogram is plotted to visualize the distribution of hydraulic throughput values (Figure 2). Additionally, the number of open pixels and how they correspond to S values is plotted (Figure 3) in a density plot.

In [7]:
```
import seaborn as sns

fig, axes = plt.subplots(1, 2, figsize=(12, 3))

# histogram of hydraulic throughput values
axes[0].hist(y, bins=24)
axes[0].set_xlabel('Hydraulic Throughput ($S$) $[m^2]$')
axes[0].set_ylabel('Count')
axes[0].set_title('Figure 2: Distribution of Hydraulic Throughput Values', style='italic')

# hydraulic throughput vs. number of open pixels
dp = sns.kdeplot(x=total_area, y=y, cmap='coolwarm', fill=True, ax=axes[1])
axes[1].set_xlabel('Number of Open Pixels')
axes[1].set_ylabel('($S$) $[m^2]$')
axes[1].set_title('Figure 3: Density Plot of Hydraulic Throughput vs. Open Pixels', style='italic')
axes[1].set_xlim(0)
axes[1].set_ylim(0)
plt.colorbar(dp.collections[-1], ax=axes[1], label='Density', shrink=0.9).ax.yaxis.offsetText.set_visible(False)

plt.tight_layout()
plt.show()
```

Figure 2: Distribution of Hydraulic Throughput Values

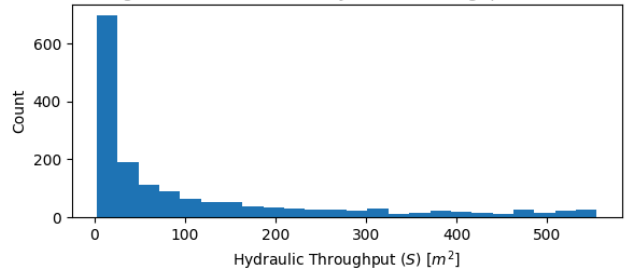


Figure 3: Density Plot of Hydraulic Throughput vs. Open Pixels

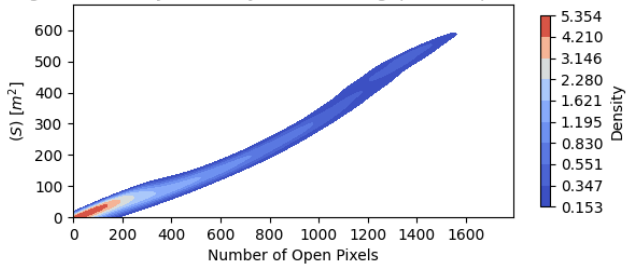


Figure 2 shows that most hydraulic throughput S values are concentrated at the lower end. This indicates that higher S values are less frequent suggesting that many cross-sections have limited pathways for fluid flow, leading to lower hydraulic throughput.

Figure 3 shows a positive correlation between the number of open pixels and hydraulic throughput. The highest density of points is around lower open pixel counts and lower S values, indicating that most cross-sections have fewer open pixels and consequently lower hydraulic throughput. As the number of open pixels increases, S also increases implying that cross-sections with more open pixels provide more continuous pathways for fluid flow, resulting in higher hydraulic

throughput. The concentration of lower S values in *Figure 2* aligns with the scatter plot, indicating that many images have fewer open pixels, hence lower hydraulic throughput.

Exploring Symmetries in the Problem

Understanding the symmetries of the problem is key to designing an effective neural network model. Here, each symmetry is analyzed to determine which ones can be used while preserving the integrity of the white (open) pixel clusters. These symmetries are then discussed on whether they can be used for data augmentation or not.

Mathematically, symmetries can be described by groups. A relationship between a function (such as a neural network layer) and a symmetry group by considering its *equivariance* properties. A map $f\colon X \to Y$ is said to be equivariant with respect to the actions $\rho\colon G \times X \to X$ and $\rho'\colon G \times Y \to Y$ of a group G on X and Y if[6]:

$$f(\rho_g(x)) = \rho'_g(f(x)) \tag{4}$$

For this property to hold a set of problem specific criteria must be laid out:

1. After any transformation the number of white pixels must remain constant in the image.
2. The correlation analysis in Visualizing the Problem indicates a small positive correlation between consecutive cross-sections. Any transformation should maintain this property.
3. White pixel clusters in the images represent large regions in which fluid can flow through the porous medium. These clusters are important for determining the hydraulic throughput as they form the pathways through which liquid can flow (*Figure 4*). Any transformation applied to the image must preserve the structure of the clusters.

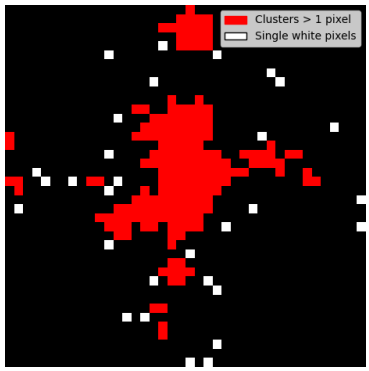


Figure 4: Open pixel clusters for hydraulic throughput

To serve as a safeguard for each transformation the `count_pixels()` method is implemented:

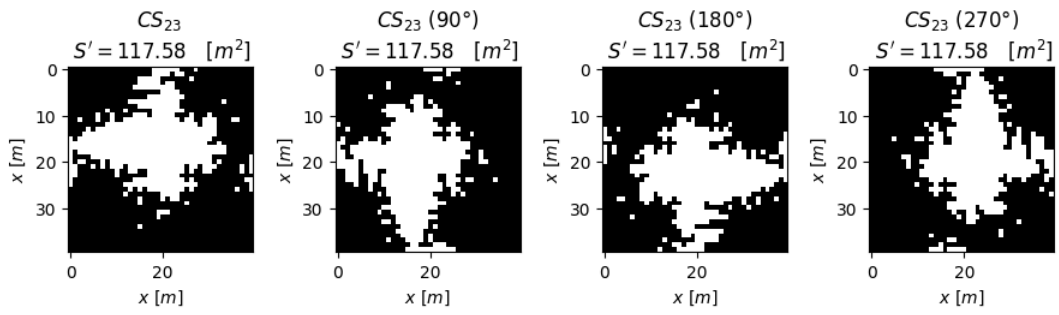
In [8]:
```
def count_pixels(original_img, transformed_img):
    c_original, c_transformed =  np.sum(original_img), np.sum(transformed_img)
    assert c_original == c_transformed, f'White pixel count not preserved! {c_original} != {c_transformed}'
```

Rotational Symmetry (R)

Rotational symmetry involves rotating the image by $90\degree$, $180\degree$, and $270\degree$. This transformation preserves both the white pixel count and the clusters. These transformations can be seen in *Figure 5*. This symmetry is adequate for use in data data augmentation.

In [9]:
```
fig, axes = plt.subplots(1, 4, figsize=(11, 2))
plot_image(axes[0], X[22], 22)
plot_image(axes[1], np.rot90(X[22]), 22, add='($90\degree$)')
plot_image(axes[2], np.rot90(X[22], 2), 22, add='($180\degree$)')
plot_image(axes[3], np.rot90(X[22], 3), 22, add='($270\degree$)')

plt.suptitle('Figure 5: Rotated Cross-Sections over $ 90\degree $ Intervals', fontsize=12, style='italic', y=1.3)
plt.show()
```

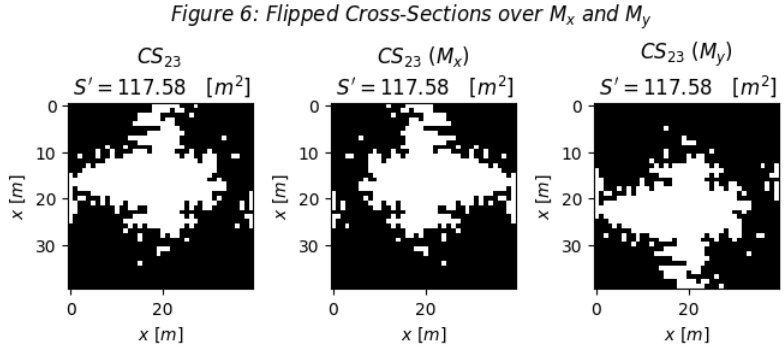
Figure 5: Rotated Cross-Sections over 90° Intervals



Flip Symmetry (M)

Flip symmetry involves flipping the image either horizontally on the x-axis (M_x) or vertically on the y-axis (M_y) seen in *Figure 6*. This transformation is significant as it maintains the relative positions and connectivity of the white pixel clusters, making sure that the fluid pathways are preserved. Similar to the rotational symmetry this method can also be used for data augmentation.

In [10]:
```
fig, axes = plt.subplots(1, 3, figsize=(8, 2))
plot_image(axes[0], X[22], 22)
plot_image(axes[1], np.fliplr(X[22]), 22, add='($M_x$)')
plot_image(axes[2], np.flipud(X[22]), 22, add='($M_y$)')

plt.suptitle('Figure 6: Flipped Cross-Sections over $M_x$ and $M_y$', fontsize=12, style='italic', y=1.3)
plt.show()
```



Translational Symmetries (T)

Translational symmetry involves shifting the image in various directions and two dimensions, T_x and T_y. This can be done using two approaches: wrapping or padding.

- **Wrapping**: This method revolves around shifting pixels from one end of the image to the opposite side. Wrapping is not a valid method since it might disrupt the connectivity of open pixel clusters.
- **Padding**: Adding padding (extra black pixels) to the image to allow for shifts without losing white pixels. This implementation is also invalid since it changes the image size making it incompatible with the fixed input size expected by the neural network.

Translational symmetries cannot be used for data augmentation because the two approaches violate the invariance criteria. However, Convolutional Neural Networks (CNNs) inherently possess properties that handle translational symmetries. This is explored further in Equivariance Properties of CNNs.

General Permutations

General permutations involve arbitrary rearrangement of pixels within the image. This transformation disrupts the structure and connectivity of white pixel clusters, leading to significant changes in the image (*Figure 7*). Hence this method is also deemed invalid.

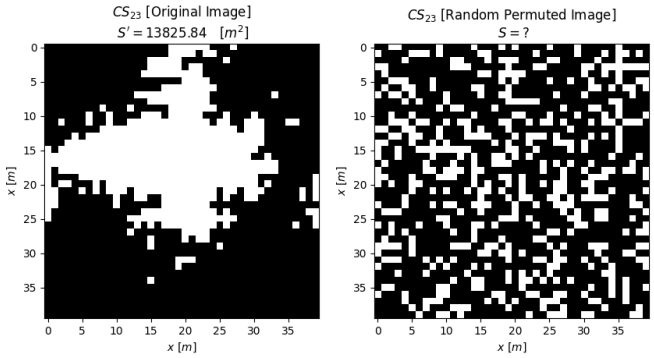


Figure 7: Original Cross-Section vs. Random Permutation
Original image (left) with calculated hydraulic throughput vs. randomly permuted image (right), showing the invalidity of general permutations due to loss of structural integrity.

Combined Symmetries ($R \times M$)

The combined symmetries of rotations and flip symmetries can be described by the direct product of rotation group R and the flip reflection group M. These combined symmetries belong to the dihedral group D_4 which is the symmetry group of the square[3], resulting in 8 unique transformations (*Table 1*).

	Id	R_{90}	R_{180}	R_{270}	M_x	M_y
Id	Id^*	R_{90}^*	R_{180}^*	R_{270}^*	M_x^*	M_y^*
$\mathbf{M_x}$	M_x	$M_x \cdot R_{90}^*$	R_{180}	$M_x \cdot R_{270}^*$	Id	R_{180}
$\mathbf{M_y}$	M_y	$M_y \cdot R_{90}$	R_{180}	$M_y \cdot R_{270}$	R_{180}	Id

Table 1: All possible transformations and their combinations for Identity (Id), Rotation (R), and Flip (M).
**: Unique transformations, 8 in total.*

Symmetries and Convolutional Neural Networks

Deep Learning techniques, particularly convolutional neural networks (CNNs), have achieved significant success in processing image-based data, where the data presents a well-defined structure in the regular grid of pixels[4]. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution (*Figure 8*). *Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*[5]

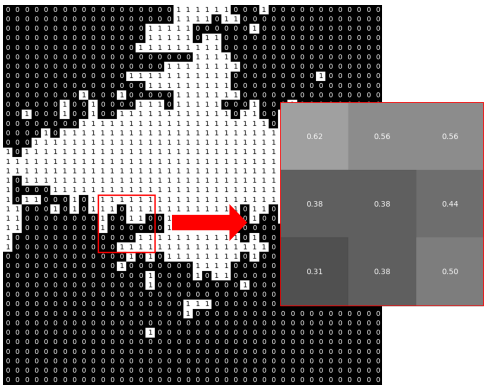


Figure 8: Convolutional Operation Applied on a Cross-Section
In the image a 4×4 convolutional filter is applied to a 6×6 subset of the cross-section, resulting in a 3×3 output grid.

Convolution in CNNs incorporates a sliding filter (kernel, seen in the left red box in *Figure 8*), across the input image and computing the dot products between the filter and the overlapping regions of the input. For an input I and a filter K, the convolution operation S_c at position (i, j) is given by[5] (5) where m and n are the image dimensions:

$$S_c(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n) \tag{5}$$

This generates a feature map (right red box in *Figure 8*) that highlights the important features such as edges and patterns. By stacking convolutional layers, CNNs learn hierarchical feature representations: lower levels capture simple features, while higher layers detect complex patterns. CNNs can process multiple channels, for example, RGB images have three channels for red, green, and blue. In the context of my images, only a single channel is used since the images are binary.

Equivariance Properties of CNNs

Convolutional neural networks (CNNs) exhibit a key property known as equivariance, particularly translation equivariance. This means that if the input image is translated, the output feature maps are correspondingly translated. *This property allows CNNs to effectively recognize patterns regardless of their position in the input image, making them highly suitable for tasks in computer vision where object detection and recognition are paramount*[4].

The translation equivariance of CNNs comes from the convolution operation, where filters are applied uniformly across the entire spatial dimension of the input. As a result, features detected by these filters maintain their relative spatial relationships even when the input image is shifted. CNN layers are not invariant, but rather preserve discrete translation symmetries at each layer; if the input image is shifted by a certain number of pixels, the output of the CNN layer (ignoring boundary effects) is shifted by a corresponding amount[6].

On the other hand, rotational and reflectional symmetries are not inherent to CNNs, meaning that there is a need for 'pushing' the model to learn these representations. This can be achieved by (1) data augmentation, (2) disambiguation, or (3) specialized network designs[7].

Data Augmentation

As previously stated, CNNs are inherently translation equivariant due to the convolutional operation, other symmetries such as the rotations and the flips are not accounted for out of the box. By artificially expanding the training dataset with transformed versions of the original data, the model is exposed to a greater variety of data, enabling the model to learn more invariant and extensive features.

The data augmentation method `augment()` takes in an image and generates all possible transformations from *Table 1*. This function also utilizes the `count_pixels()` method defined above to make sure the criteria is met. In Python:

In [11]:
```python
def augment(image):
    transformations = [
        lambda x: x,  # original image
        lambda x: np.rot90(x, 1), lambda x: np.rot90(x, 2), lambda x: np.rot90(x, 3),  # r90, r180, r270
        np.fliplr, np.flipud,  # flip x, y
        lambda x: np.flipud(np.rot90(x, 1)), lambda x: np.flipud(np.rot90(x, 3))  # # flip X+r90, flip X+r270
    ]
    X_aug = [transform(image) for transform in transformations]

    return [count_pixels(image, img) or img for img in X_aug]  # call count_pixels() to make sure the criteria is still met!
```

Then, the `augment()` method is expanded to be applied on the entire dataset. It is observed from the print statement that the dataset is now 8 times its original size.

In [12]:
```python
X_aug = np.array([aug_img for img in X for aug_img in augment(img)])
y_aug = np.array([target for target in y for _ in range(len(augment(X[0])))])

print(f'Original dataset shape: {X.shape}, Augmented dataset shape: {X_aug.shape}')
```

```
Original dataset shape: (1660, 40, 40), Augmented dataset shape: (13280, 40, 40)
```

Disambiguation

In contrast to data augmentation, data disambiguation is the process of transforming data into its canonical form, this ensures consistency by reducing variability in the data. In the context of the problem, disambiguation helps in standardizing the representation of cross-section images so that the neural network can learn more effectively without being confused by different symmetrical versions of the same image.

Given an image X, a set of all possible transformations $\mathcal{T}$ is applied (6). The goal is to find the canonical from $X_{canonical}$ among all of these transformations.

$$\mathcal{T}(X) = \{t(X) \mid t \in \text{Transformations}\} \tag{6}$$

The canonical form $X_{canonical}$ is then defined as the transformation that *minimizes* the lexicographical order of the byte representation of the image[8] [9]:

$$X_{canonical} = \arg\min_{t \in \mathcal{T}} \text{bytes}(\text{t}(\text{X})) \tag{7}$$

This is implemented within the `disambiguate()` function:

In [13]:
```python
def disambiguate(image):
    transformations = augment(image)  # generate T, Equation 6
    return min(transformations, key=lambda t: t.flatten().tobytes())  # return X_canonical, Equation 7

# and, applying to the entire dataset:
X_disambiguated = np.array([disambiguate(img) for img in X])
y_disambiguated = y  # to have consistency among the variables, pass the values of y to y_disam...
```

Discussion on Data Augmentation and Disambiguation

Data augmentation increases the variety and robustness of the dataset by introducing rotations and flips, which enhances the model's generalization capabilities. However, it also significantly enlarges the dataset size, leading to longer training times and higher computational demands. Additionally, it may introduce redundancy, as the same structure appears in multiple orientations, and could potentially lead to overfitting if not managed properly.

Data disambiguation ensures consistent data representation by standardizing each unique structure into a canonical form, which reduces dataset size and improves model efficiency by focusing on essential features. Despite these benefits, disambiguation is more complex and computationally intensive to implement, requiring additional steps to determine the canonical form. This process may also reduce the model's ability to recognize symmetries directly from the data.

Method	Pros	Cons
Data Augmentation	- Increases data variety and robustness	- Enlarges dataset size
	- Improves generalization	- Leads to longer training times
	- Easy to implement	- Potential for overfitting
Data Disambiguation	- Ensures consistent data representation	- More complex and computationally intensive
	- Reduces dataset size	- May reduce model's ability to recognize symmetries
	- Improves model efficiency	- Requires careful selection of canonical form

Table 2: Pros and cons of data augmentation and disambiguation.

The last method *'Hardwiring' Symmetries* is discussed further in Symmetric-Model (1) and Symmetric-Model (2)

Building the Neural Network

In this section, I discuss the framework for building and training multiple Convolutional Neural Network architectures. I start with simple models with no symmetry considerations and progressively build up the models to incorporate symmetries inherent in the problem. Further, methods to both improve model accuracy and symmetry considerations such as pooling and regularization are discussed. Finally, remarks on overfitting and model complexity are explored.

Reproducibility

To ensure the reproducibility of results, I set seeds for NumPy and PyTorch and fix the Python hash seed. This ensures that the experiments can be replicated with consistent results.

In [14]:
```python
# Core
import os
import random
# Torch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
# Util
import pandas as pd
from sklearn.model_selection import train_test_split

# Reproducibility bit
np.random.seed(5); torch.manual_seed(5); random.seed(5)
os.environ['PYTHONHASHSEED'] = '5'  # just to make sure everything is passed onto seed=5

torch.__version__
```

Out[14]:
```
'2.2.2+cu121'
```

CNNs can be computationally intensive and time-consuming due to the amount of operations needed per layer. I utilize GPU acceleration with CUDA to make sure these computations run parallel and ultimately reduce training time. With CUDA enabled extra steps need to be taken to ensure reproducibility.

In [15]:
```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Running PyTorch on: {device}")

if torch.cuda.is_available():
    device_id = torch.cuda.current_device()
    print(f'Allocated Memory {torch.cuda.memory_allocated(device_id)}')
    print(f'Reserved Memory {torch.cuda.memory_reserved(device_id)}')

    # Extra reproducibility for CUDA/CUDNN
```

```
    torch.cuda.manual_seed(5)
    torch.backends.cudnn.deterministic = True  # make sure all operations are deterministic
    torch.backends.cudnn.benchmark = False
```

```
Running PyTorch on: cuda
Allocated Memory 0
Reserved Memory 0
```

Defining Necessary Functions

A set of necessary functions need to be defined before moving on to the training and validation of CNN models. These take the role of preparing the data, defining the loss function, and methods to prevent overfitting.

Prepare the Data

Before training the models, the data is prepared into PyTorch readable format using tensors. The binary images need an additional dimension to represent the channel mentioned in Symmetries and Convolutional Neural Networks, this is achieved by using `.unsqueeze(1)` over the tensors. Similarly, the target tensors are also unsqueezed to add an extra dimension, ensuring they are compatible with the model's output format.

With the method `to_torch()` the data is first split into `80:20` ratio training and testing datasets, which are then passed onto the tensors using the `SpongeData` class.

In [16]:
```
class SpongeData(Dataset):
    def __init__(self, X, y):
        self.features = torch.tensor(X, dtype=torch.float32).unsqueeze(1).to(device)  # pass the data/tensors onto CPU or CUDA and unsqueeze
        self.targets = torch.tensor(y, dtype=torch.float32).unsqueeze(1).to(device)

    def __len__(self):
        return self.features.size()[0]

    def __getitem__(self, idx):
        return self.features[idx], self.targets[idx]

def to_torch(X, y ,split=0.2):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=split)

    train = SpongeData(X_train, y_train)
    test = SpongeData(X_test, y_test)

    return train, test

def to_numpy(*tensors):
    """this method is used to convert tensors back into numpy arrays"""
    return [tensor.cpu().numpy() if isinstance(tensor, torch.Tensor) else tensor for tensor in tensors]
```

In [17]:
```
train_dataset, test_dataset = to_torch(X, y)
train_dataset_aug, test_dataset_aug = to_torch(X_aug, y_aug)
train_dataset_disam, test_dataset_disam = to_torch(X_disambiguated, y_disambiguated)

data_dict = {
    'regular': {'train': train_dataset, 'test': test_dataset},
    'augmented': {'train': train_dataset_aug, 'test': test_dataset_aug},
    'disambiguated': {'train': train_dataset_disam, 'test': test_dataset_disam}
}

# Print shapes for verification
for dataset_type, datasets in data_dict.items():
    print(f'Shape of features ({dataset_type} dataset): {datasets["train"].features.shape}')
    print(f'Shape of targets ({dataset_type} dataset): {datasets["train"].targets.shape}\n')
```

```
Shape of features (regular dataset): torch.Size([1328, 1, 40, 40])
Shape of targets (regular dataset): torch.Size([1328, 1])

Shape of features (augmented dataset): torch.Size([10624, 1, 40, 40])
Shape of targets (augmented dataset): torch.Size([10624, 1])

Shape of features (disambiguated dataset): torch.Size([1328, 1, 40, 40])
Shape of targets (disambiguated dataset): torch.Size([1328, 1])
```

Loss Function

The Root Mean Square Percentage Error (RMSPE) (8) is used as the loss function for training the neural network. The RMSPE is used for this problem as it normalizes the prediction errors relative to the true values, where $T^{(i)}$ is the true value and $S^{(i)}$ is the predicted value at position i:

$$\mathrm{RMSPE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{S^{(i)} - T^{(i)}}{T^{(i)}}\right)^2} \tag{8}$$

Which is implemented in the `RMSPE` class:

In [18]:
```
class RMSPE(nn.Module):
    def __init__(self):
        super(RMSPE, self).__init__()

    def forward(self, y_pred, y_true):
        return torch.sqrt(torch.mean(((y_true - y_pred) / y_true) ** 2))
```

Early Stopping

To combat overfitting in the models, early stopping is implemented using the `EarlyStopping` class. Overfitting occurs when a model learns excessively from the training data, capturing relevant patterns and inconsequential noise[10] ultimately leading to poor generalization of the unseen data. Early stopping monitors the

model's performance on a validation set and halts training when the performance stops improving.

In [19]:
```python
class EarlyStopping:
    def __init__(self, tolerance=30, delta=0.001):
        self.tolerance = tolerance  # halts the training if the val_loss hasn't improved after the tolerance
        self.delta = delta  # minimum change in the loss to qualify as an improvement
        self.counter = 0
        self.early_stop = False
        self.best_score = None

    def __call__(self, val_loss):
        score = -val_loss
        if self.best_score is None:
            self.best_score = score
        elif (self.tolerance > 0) and (score < self.best_score + self.delta):  # when tolerance is negative early stopping is turned off
            self.counter += 1
            if self.counter >= self.tolerance:
                self.early_stop = True
        else:
            self.best_score = score
            self.counter = 0
```

Plotting & Model Evaluation

Visualizing a model's performance can be significant to understanding underlying patterns and issues that might lead to overfitting. The `plot_losses_and_predictions()` function plots the training and testing losses over epochs on the log scale. Additionally, a scatter plot of predicted vs. actual throughput values is visualized.

In [20]:
```python
def plot_losses_and_predictions(res, y_test, fig_no, add=None):
    train, test, y_test, y_pred = to_numpy(res['train_hist'], res['val_hist'], y_test, res['y_pred'])

    fig, axs = plt.subplots(1, 2, figsize=(14, 5))
    # plot_losses from the lab notebooks
    axs[0].semilogy(train, label='Train', marker='.')
    axs[0].semilogy(test, label='Test', marker='.')
    axs[0].set_xlabel('Epochs')
    axs[0].set_ylabel('Loss (RMSPE)')
    axs[0].legend()
    axs[0].set_title('Training and Testing Losses')

    # actual vs predicted scatter plot
    axs[1].scatter(y_test, y_pred, label='Predictions')
    axs[1].plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--', lw=2, label='ideal fit')
    axs[1].set_xlabel('Actual $S$ $[m^2]$')
    axs[1].set_ylabel('Predicted $\hat{S}$ $[m^2]$')
    axs[1].legend()
    axs[1].set_title('Actual vs Predicted Throughput')

    add = add if add else ''
    plt.suptitle(f'Figure {fig_no}: Model Performance and Predictions for {res["model_name"]} {add}', fontsize=15, style='italic')

    total_params = sum(param.numel() for param in res['model'].parameters())

    text  = f'Total number of parameters: {total_params:,} \t\t Training time (seconds): {res["time_took"]:.2f}'.expandtabs()
    fig.text(0.5, -0.05, text, ha='center', fontsize=12, style='italic')

    plt.tight_layout()
    plt.show()
```

Training Logic

The provided training function incorporates several techniques to combat overfitting and improve the various models' performance:

Learning Rate Scheduling:

The learning rate scheduler `ReduceLROnPlateau` is employed to dynamically adjust the learning rate based on the validation loss. If the validation loss does not improve for a certain number of epochs (patience), the learning rate is reduced by a factor, which ultimately helps in fine-tuning the model during training[11].

Weight Decay:

Weight decay is a kind of L_2 regularization that is added to the loss function by the `ADAM` optimizer. The change on the loss function can be represented by RMSPE_{L_2} where θ represents the model parameters and λ is the regularization term:

$$\text{RMSPE}_{L_2}(\theta) = \text{RMSPE}(\theta) + \lambda \sum_i \theta_i^2 \tag{9}$$

GradScaler and Autocast:

To reduce the training time and memory usage the `GradScaler` and `autocast` are used. Together, they enable mixed-precision training by using both 16-bit and 32-bit floating point types.

Returning the Best Model:

This function keeps track of the best model weights based on the validation loss. After training, it returns the model with the lowest validation loss score, ensuring the best-performing model is used.

These are then combined with the main training logic and splitting the data into batches under the `train()` function:

In [21]:
```python
import time
from torch.cuda.amp import GradScaler, autocast
from tqdm.notebook import tqdm  # for the beautiful progress bar

def train(train_dataset, val_dataset, model, loss_fn=RMSPE(), num_epochs=200, batch_size=64, learning_rate=1e-3, early_stop=50, verbosity=-1, use
```

```
    # Prepare the data using data loaders - will split into batch sizes
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=0)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=0)

    # ---- Initialize variables of interest and history ----- #
    train_hist, val_hist = [], []
    best_loss, best_wights = np.inf, None

    # ----- Initialize the methods ------- #
    early_stopper = EarlyStopping(tolerance=early_stop, delta=0.001)  # initialize the early stopping logic
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-5)  # ADAM optimizer + weight decay
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.01, patience=10, min_lr=1e-6)  # actively updating the

    # ----- Get the model ready ----- #
    model.to(device)  # pass it onto CUDA or CPU
    scaler = GradScaler()  # CUDA automatic gradient scaling

    # ----- Training ----- #
    tqdm_pbar = tqdm(range(num_epochs), desc='Initializing') if use_pbar else range(num_epochs)  # progress bar
    start_time = time.time()

    for epoch in range(num_epochs):
        model.train()
        train_loss, val_loss = 0.0, 0.0

        for i, (X_batch, y_batch) in enumerate(train_loader, 0):
            optimizer.zero_grad()  # main forward pass
            with autocast():
                y_pred = model(X_batch)
                loss = loss_fn(y_pred, y_batch)

            scaler.scale(loss).backward()  # backward pass
            scaler.step(optimizer); scaler.update()

            train_loss += loss.item()

        # ----- Validation ----- #
        model.eval()
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                y_pred = model(X_batch)
                loss = loss_fn(y_pred, y_batch)
                val_loss += loss.item()
        model.train()

        # Get the losses & store
        train_loss /= len(train_loader)
        val_loss /= len(val_loader)
        train_hist.append(train_loss)
        val_hist.append(val_loss)

        scheduler.step(val_loss)  # update learning rate

        # ------- EarlyStopping -------#
        if val_loss < best_loss:  # getting the best model alongside early stopping
            best_loss = val_loss
            best_weights = model.state_dict()

        early_stopper(val_loss)
        if early_stopper.early_stop:
            print(f'Early stopping at epoch {epoch}, validation loss: {val_loss:.8f}')
            break


        # -------- Printing --------- #
        if (verbosity > 0) and (epoch % verbosity == 0):
            print(f"Epoch [{epoch}/{num_epochs}] Train Loss: {train_loss:.8f}, Validation Loss: {val_loss:.8f}")

        if use_pbar:
            tqdm_pbar.set_description(
            f"Epoch [{epoch + 1}/{num_epochs}] Train Loss: {train_loss:.8f},"
            f" Val Loss: {val_loss:.8f}, Early Stop: {early_stopper.counter}")
            tqdm_pbar.update(1)


    # ------- Returning best model --------
    if best_weights is not None:
        model.load_state_dict(best_weights)
        if use_pbar and (verbosity < 0):
            print(f'Returning the best model with validation loss: {best_loss:.4f}')

    # return (best) model parameters alongside train & val history
    return {
        'model_name': model.__class__.__name__, 'model': model,
        'train_hist': train_hist, 'val_hist': val_hist,
        'time_took': time.time() - start_time
    }

def predict(model, validation_tensor):
    with torch.no_grad():
        y_pred = model(validation_tensor.to(device))
        return np.array(to_numpy(y_pred))
```

In [22]:
```
results = dict()  # dictionary to store model information
```

Simple Model with No Symmetry Considerations

The `SimpleHydraulicCNN` model is a simple CNN that does not account for the rotational and mirror symmetries, due to the use of convolutional layers, the translational symmetry is inherently accounted for. This model consists of three convolutional layers followed by *max-pooling* layers, and two fully connected (dense) layers with ReLU in between (*Figure 9*). The convolutional layers extract spatial features from the images, while the fully connected layers perform the final regression.

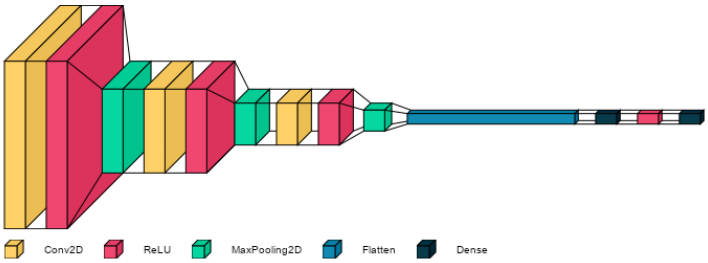


Figure 9: Model Architecture for the Simple Model

Model Architecture

1. The convolutional layers with increasing filter counts `16, 32, 64` capture hierarchical features from the 40×40 input images. This architecture allows the model to detect edges in a given image at the initial layers and patterns at deeper layers.
2. The usage of **pooling** layers is crucial in this model since they reduce the spatial dimensions of a feature map generated by a convolutional layer[12]. This operation also ensures that the model CNN is *translation invariant*. Max-pooling operation selects the maximum value from each window (kernel size) of the feature map to decrease complexity and be a safeguard against overfitting. For an input feature map X, the max-pooling operation over a $k \times k$ region where Y is the pooled output, and i and j are the spatial coordinates:

$$Y(i,j) = \max_{0 \le m,n < k} X(i \cdot k + m, j \cdot k + n) \tag{10}$$

This model results in a total of $228,112$ parameters which is quite a lot, but will be used as the max parameter baseline.

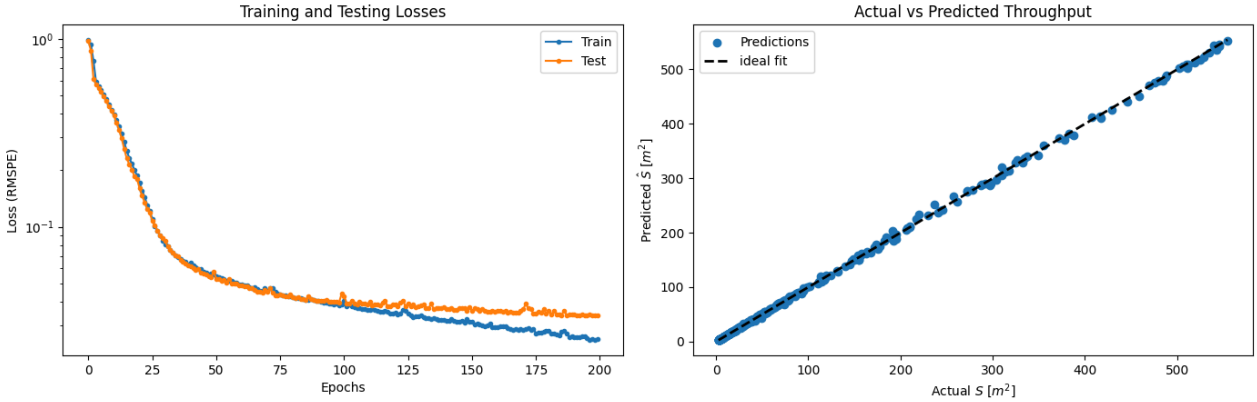
The `SimpleHydraulicCNN` class is implemented to reflect this architecture:

In [23]:
```python
class SimpleHydraulicCNN(nn.Module):
    def __init__(self):
        super(SimpleHydraulicCNN, self).__init__()
        # Convolutional layers 16 → 32 → 64 respectively
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1, bias=False)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1, bias=False)

        # Max pooling - max pooling is added between each convolutional layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Dense layers
        self.fc1 = nn.Linear(64 * 5 * 5, 128, bias=False)
        self.fc2 = nn.Linear(128, 1, bias=False)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # 20x20
        x = self.pool(F.relu(self.conv2(x)))  # 10x10
        x = self.pool(F.relu(self.conv3(x)))  # 5x5
        x = x.view(-1, 64 * 5 * 5)  # 1600 → 128
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

In [24]:
```python
model_name = 'SimpleNet'
results[model_name] = train(train_dataset, test_dataset,
                            model=SimpleHydraulicCNN(),
                            loss_fn=RMSPE(),
                            num_epochs=200,
                            batch_size=64,
                            learning_rate=1e-4,
                            early_stop=25,
                            verbosity=-1)

results[model_name]['y_pred'] = predict(results[model_name]['model'], test_dataset.features)
plot_losses_and_predictions(results[model_name], test_dataset.targets, fig_no=10)
```

```
Initializing:   0%|          | 0/200 [00:00<?, ?it/s]
Early stopping at epoch 199, validation loss: 0.03361848
Returning the best model with validation loss: 0.0336
```

Figure 10: Model Performance and Predictions for SimpleHydraulicCNN



Total number of parameters: 228,112 *Training time (seconds): 37.41*

While maintaining a simple structure, the `SimpleHydraulicCNN` performs relatively well with final $RMSPE$ in the 0.03 range. Both the training and testing validation losses converge around this value and due to the early stopping and the learning rate scheduling they show no sign of overfitting. The scatter plot shows a tight fit around the `ideal fit` line. With a training time of around half a minute and $228,112$ parameters this model serves as a baseline for further improvements.

Symmetric Model (1)

This symmetry-aware model architecture is the `SymmetryNet` architecture. This model is designed to utilize the symmetries in the data by generating all 8 possible symmetries from Combined Symmetries ($R \times M$). Each symmetry is processed through a modular 'carrier network', which in this implementation is the `SimpleHydraulicCNN`. The outputs from these symmetry transformations are then concatenated and passed through a final linear layer to produce the predicted hydraulic throughput $\hat{S}$ (*Figure 11*).

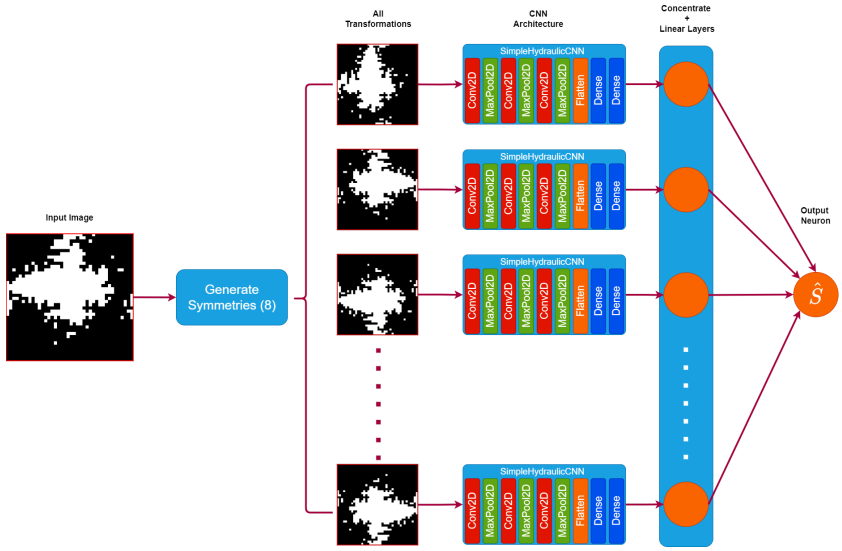


Figure 11: Model Architecture for the Symmetry Respecting Model (1)

Model Architecture

1. The `SimpleHydraulicCNN` is used as the **carrier network** for processing each symmetry. This modular design allows for flexibility in selecting different carrier networks if needed. The model generates all 8 symmetries of the input image, including the identity transformation, rotations (90°, 180°, 270°), horizontal and vertical flips, and combinations of flips with rotations. This architecture ensures that the model parameters of the original model (plus 8) is kept while symmetries are incorporated into the model.

2. For the concentration of the symmetries, this model **avoids group averaging**. This decision ensures that each symmetry transformation is treated independently, allowing the model to learn robust features from each transformation separately. Group averaging can cause the model to converge to a less optimal solution by smoothing out the gradients, which can prevent the network from fully capturing the unique features presented by each symmetry. By processing each symmetry independently, the model maintains the distinct information from each transformed view.

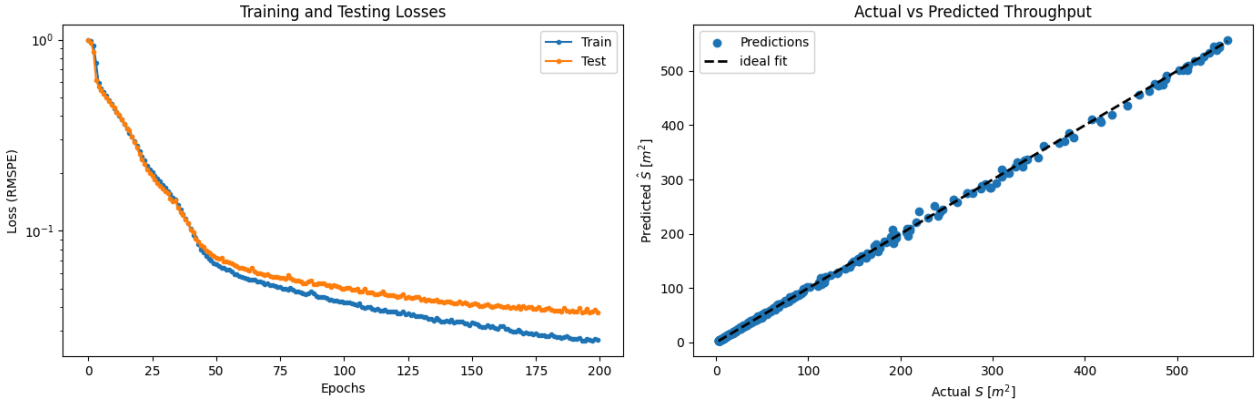
In [25]:
```python
class SymmetryNet(nn.Module):
    def __init__(self, carrier_network):
        super(SymmetryNet, self).__init__()
        self.carrier_network = carrier_network  # so I can select which model to be the carrier network
        self.fc = nn.Linear(8, 1, bias=False)

    def forward(self, x):
        # generate all 8 of the symmetries
        transformations = [
            lambda x: x,                                    # Id
            lambda x: torch.rot90(x, 1, [2, 3]),            # R90
            lambda x: torch.rot90(x, 2, [2, 3]),            # R180
            lambda x: torch.rot90(x, 3, [2, 3]),            # R270
            lambda x: torch.flip(x, [2]),                   # FlipX
```

```
            lambda x: torch.flip(x, [3]),                          # FlipY
            lambda x: torch.rot90(torch.flip(x, [2]), 1, [2, 3]),  # FlipX + R90
            lambda x: torch.rot90(torch.flip(x, [2]), 3, [2, 3])   # FlipY + R270
        ]

        outputs = [self.carrier_network(transform(x)) for transform in transformations] # pass each transformation through the network
        combined_out = torch.cat(outputs, dim=1) # conenctrate into [batch_size, 8]

        out = self.fc(combined_out)  # last linear layer of 8x1
        return out
```

In [26]:
```
model_name = 'SymmetryNet'
results[model_name] = train(train_dataset, test_dataset,
                            model=SymmetryNet(carrier_network=SimpleHydraulicCNN()),
                            loss_fn=RMSPE(),
                            num_epochs=200,
                            batch_size=64,
                            learning_rate=1e-4,
                            early_stop=25,
                            verbosity=-1)

results[model_name]['y_pred'] = predict(results[model_name]['model'], test_dataset.features)
plot_losses_and_predictions(results[model_name], test_dataset.targets, fig_no=12, add='(Simple)')
```

```
Initializing:   0%|          | 0/200 [00:00<?, ?it/s]
Returning the best model with validation loss: 0.0374
```

Figure 12: Model Performance and Predictions for SymmetryNet (Simple)



Total number of parameters: 228,120 *Training time (seconds): 131.14*

The training and testing losses for `SymmetryNet` show effective learning and generalization. The training loss stabilizes around 0.03, while the validation loss converges slightly higher at approximately 0.04, indicating minimal overfitting.

The scatter plot of actual versus predicted throughput values demonstrates high prediction accuracy, with points closely clustering around the ideal fit line. This model introduces 8 extra parameters to any carrier network (`SimpleHydraulicCNN` in my case) which allows for improvement through the carrier network while still incorporating symmetries.

Symmetric Model (2)

The final symmetry-aware model architecture is the `SymmetryNet` with a `ResNet` -based carrier network. This implementation is designed to reduce the number of parameters and potentially reduce losses by utilizing residual connections. Like the previous model, the `SymmetryNet` generates all 8 possible symmetries which are then passed through the `ResNet` architecture.

Model Architecture

The `ResNet` architecture is used as the carrier network since it incorporates residual blocks (*Figure 13*) and grouped convolutions to potentially increase efficiency and performance.

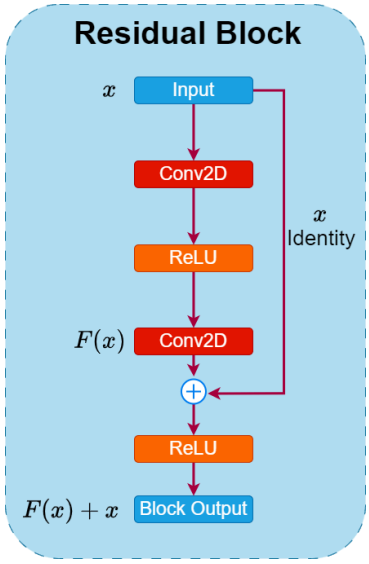


Figure 13: Residual Blocks used in the ResNet architecture

1. Residual blocks help reduce the vanishing gradient problem[13], which ultimately allows deeper networks. Each residual block consists of two convolutional layers and a shortcut connection that bypasses these layers (*Figure 13*). The output of a residual block can be represented with (11), this shortcut adds the input x directly to the output of the convolutional layers, which helps maintain the gradient flow through the network.

$$\text{Output} = \text{ReLU}(\text{Conv2}(\text{ReLU}(\text{Conv1}(x))) + \text{Shortcut}(x)) \tag{11}$$

2. Grouped convolutions reduce the number of parameters and computational complexity by dividing the input channels into smaller groups, each processed independently over the GPU. For a convolutional layer with $G = 8$ groups, the operation can be expressed as:

$$Y_{i,j} = \sum_{g=1}^{8} (W_g \cdot x_g) \tag{12}$$

Where W_g and x_g represent the weights and input channels of the g-th group. This technique ultimately reduces the number of total parameters and increases efficiency.

The `ResNet` architecture is implemented alongside the `ResidualBlock` class and then passed through the `SymmetryNet` architecture:

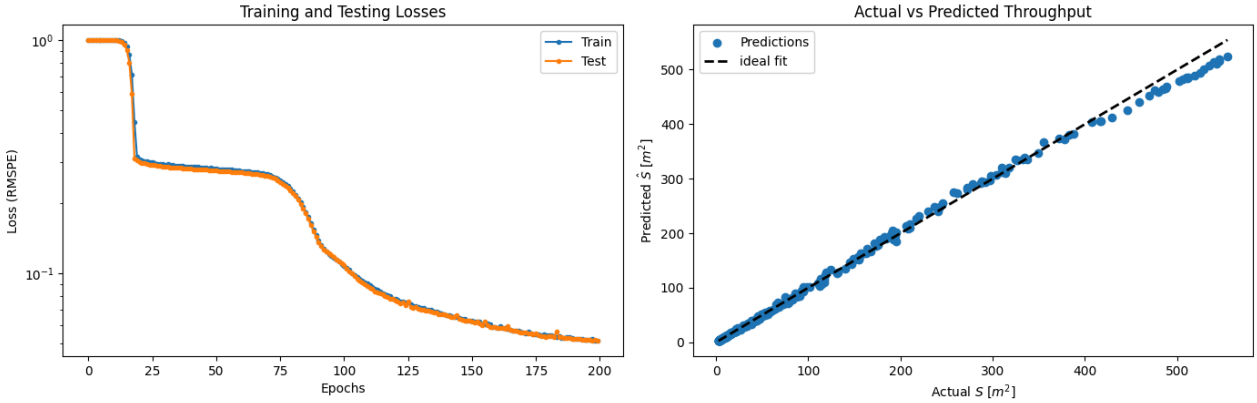
In [27]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, groups=2):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False, groups=groups) # grouped convs
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False, groups=groups)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False)) # create the shortcut l

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = self.conv2(out)
        out += self.shortcut(x)  # f(x) + x
        out = F.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, stride=1, padding=1, bias=False)
        self.res1 = ResidualBlock(8, 16, stride=2, groups=8)
        self.res2 = ResidualBlock(16, 32, stride=2, groups=8)
        self.res3 = ResidualBlock(32, 64, stride=2, groups=8)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(64, 16, bias=False)
        self.fc2 = nn.Linear(16, 1, bias=False)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)
        x = self.global_pool(x)
        x = x.view(-1, 64)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

In [28]:
```
model_name = 'SymResNet'
results[model_name] = train(train_dataset, test_dataset,
                            model=SymmetryNet(
                                carrier_network=ResNet()),  # set the carrier network as ResNet
                            loss_fn=RMSPE(),
                            num_epochs=200,
                            batch_size=64,
                            learning_rate=1e-4,
                            early_stop=25,
                            verbosity=-1)
```

```
results[model_name]['y_pred'] = predict(results[model_name]['model'], test_dataset.features)
plot_losses_and_predictions(results[model_name], test_dataset.targets, fig_no=14, add='(ResNet)')
```

```
Initializing:   0%|          | 0/200 [00:00<?, ?it/s]
Returning the best model with validation loss: 0.0515
```

Figure 14: Model Performance and Predictions for SymmetryNet (ResNet)



Total number of parameters: 12,880 *Training time (seconds): 332.08*

The final model `SymResNet` performs the worst of the three with the given hyperparameters, Over the losses this model still shows a promising trend toward lower losses and, thus is kept in the experiment. Over 200 epochs and the given parameters, the fit over the ideal line also shows underfitting, meaning that given the right amount of data and correct hyperparameters, this model can improve on these results. This model decreases the parameter count by a factor of 17, but still needs a substantial training time due to the symmetry generation.

Results

Model Performances

In this section, performance results of three different neural network models is presented: `SimpleHydraulicCNN`, `SymmetryNet`, and `SymResNet`. The evaluation is conducted across three datasets: regular data, augmented data, and disambiguated data. For each dataset, the models are trained with different batch sizes (16, 64, and 128), and the final loss, number of epochs, and training time are recorded. These help compare the efficiency and accuracy of the models under various conditions and provide insights into the impact of incorporating symmetries, data augmentation, and data disambiguation on model performance. The number of epochs is presented with `ES()` given a model has early stopped at that epoch.

	SimpleHydraulicCNN			SymmetryNet			SymResNet		
Batch Size	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**
16	0.027	ES(71)	17.697	0.014	ES(62)	56.92	0.017	ES(62)	165.208
64	0.028	ES(53)	3.825	0.015	ES(102)	24.612	0.015	ES(51)	41.434
128	0.028	ES(53)	2.191	0.014	ES(54)	10.295	0.014	ES(52)	29.615

*Table 3: Model results for batch sizes 16, 64, 128 for the **regular dataset***

	SimpleHydraulicCNN			SymmetryNet			SymResNet		
Batch Size	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**
16	0.014	ES(117)	234.057	0.003	ES(75)	713.348	0.006	ES(70)	1247.731
64	0.013	ES(65)	57.843	0.002	ES(68)	153.787	0.006	ES(50)	253.277
128	0.013	ES(82)	48.847	0.002	ES(59)	86.572	0.005	ES(50)	159.853

*Table 4: Model results for batch sizes 16, 64, 128 for the **augmented dataset***

	SimpleHydraulicCNN			SymmetryNet			SymResNet		
Batch Size	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**	**Final Loss**	**Epochs**	**Time (s)**
16	0.027	ES(52)	13.556	0.015	ES(81)	79.656	0.005	ES(54)	120.669
64	0.028	ES(65)	4.392	0.014	ES(66)	16.795	0.006	ES(51)	33.821
128	0.028	ES(53)	2.199	0.014	ES(79)	13.82	0.006	ES(53)	22.194

*Table 5: Model results for batch sizes 16, 64, 128 for the **disambiguated dataset***

- For the regular dataset, *Table 3* shows that `SymmetryNet` consistently achieves the lowest final loss across all batch sizes, indicating the highest accuracy. However, it requires more epochs and training time, particularly with a batch size of 16. `SimpleHydraulicCNN` achieves the fastest training times but with slightly higher final losses. `SymResNet` provides moderate accuracy and training times, performing slightly better than `SimpleHydraulicCNN` at batch size 128. Overall, `SymmetryNet` is the best-performing model over the regular dataset with shorter training times than `SymResNet` and better validation accuracy than `SimpleHydarulicCNN`.

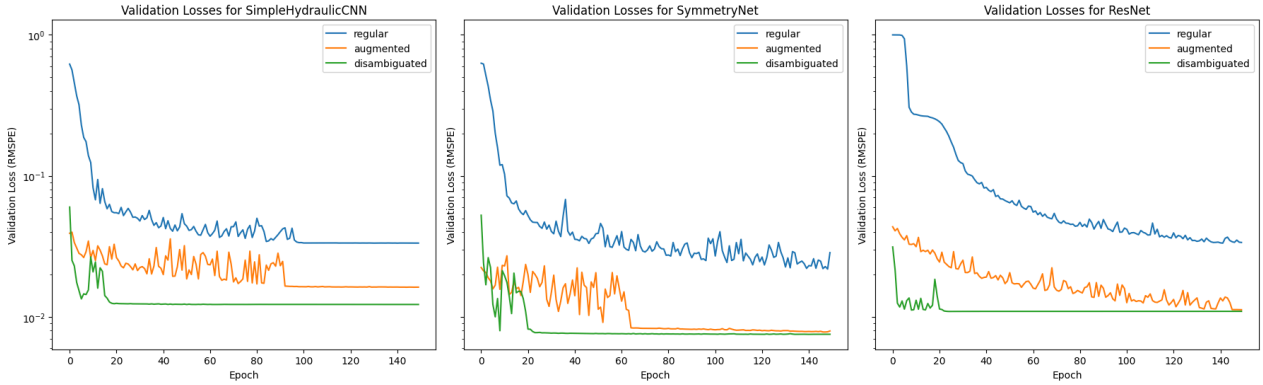
- In *Table 4*, for the augmented dataset, `SymmetryNet` consistently achieves the lowest final loss, indicating the highest accuracy but with significantly longer training times. `SimpleHydraulicCNN` shows improved accuracy over the regular dataset but still has higher final losses than the other models. `SymResNet` has better accuracy than `SimpleHydraulicCNN` but longer training times.
- In *Table 5*, for the disambiguated dataset, `SymResNet` achieves the lowest final losses, indicating the highest accuracy. `SymmetryNet` also performs well but with slightly higher losses and longer training times. `SimpleHydraulicCNN` is the fastest in training but has the highest final losses.

Effect of Datasets

The following plots display the validation losses for the `SimpleHydraulicCNN`, `SymmetryNet`, and `SymResNet` models across three datasets: regular, augmented, and disambiguated. These visualizations provide insights into how each model performs with different data preprocessing techniques.

In [40]:
```
df_losses = pd.read_csv('data/val_losses.csv')

models, datasets = df_losses['Model'].unique(), df_losses['Dataset'].unique()

fig, axes = plt.subplots(1, 3, figsize=(18, 6), sharey=True)
for ax, model in zip(axes, models):
    ax.set_title(f'Validation Losses for {model}')
    for dataset in datasets:
        data = df_losses[(df_losses['Model'] == model) & (df_losses['Dataset'] == dataset)]
        ax.semilogy(data['Epoch'], data['Validation Loss'], label=dataset)
    ax.legend()
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Validation Loss (RMSPE)')

plt.suptitle(f'Figure 15: Validation Losses for Models Over Datasets', fontsize=18, style='italic')

plt.tight_layout()
plt.show()
```

Figure 15: Validation Losses for Models Over Datasets



For all models, the disambiguated dataset consistently results in the lowest and most stable validation losses, indicating better performance and generalization. The augmented dataset also shows improved performance over the regular dataset, with lower validation losses and better stability. `SymResNet` and `SymmetryNet` improve with data augmentation and disambiguation, achieving significantly better results than `SimpleHydraulicCNN`.

Computational Efficiency

This section compares the average training times of `SimpleHydraulicCNN`, `SymmetryNet`, and `SymResNet` models across different batch sizes to assess computational efficiency.

In [41]:
```
df_model_results = pd.read_csv('data/model_results.csv')

time_and_batch = (df_model_results
                        .groupby(['model_name', 'batch_size'])['training_time']
                        .mean()
                        .unstack()
                        .reindex(['SimpleHydraulicCNN', 'SymmetryNet', 'SymResNet']))

model_and_param = (df_model_results
                        .groupby('model_name')['num_params']
                        .first()
                        .reindex(['SimpleHydraulicCNN', 'SymmetryNet', 'SymResNet']))

fig, ax = plt.subplots(figsize=(6, 4))
time_and_batch.plot(kind='bar', ax=ax, position=0)

ax.set_xlabel('Model Name')
ax.set_ylabel('Average Training Time (seconds)')
ax.set_title('Figure 16: Average Training Time per Batch Size for Different Models', fontsize=11, style='italic')
ax.legend(title='Batch Size')
ax.set_xticklabels([f'{model}\n({params:,}K)' for model, params in model_and_param.items()], rotation=0, ha='left')
plt.tight_layout()
plt.show()
```

Figure 16: Average Training Time per Batch Size for Different Models

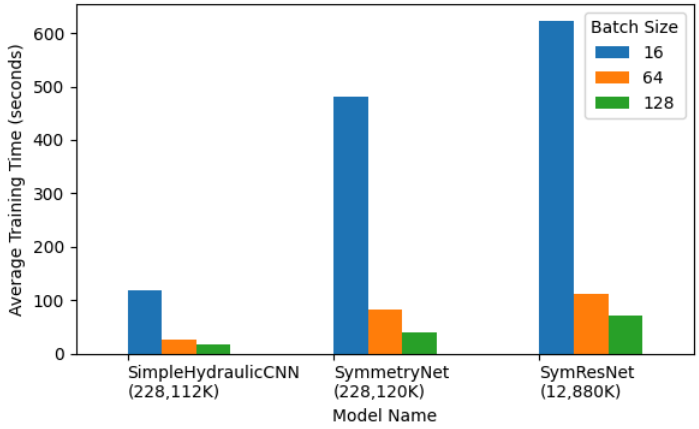


Figure 16 shows that `SimpleHydraulicCNN` has the shortest training times for all batch sizes, indicating the highest efficiency. `SymResNet` has the longest training times, especially for smaller batch sizes, due to its larger parameter count. `SymmetryNet` falls between the two, with moderate training times. Increasing the batch size reduces training time for all models, highlighting the efficiency benefits of larger batch sizes. Overall, `SimpleHydraulicCNN` is the most efficient, while `SymResNet` trades off training time for higher accuracy.

Discussion

The results indicate that incorporating symmetries into CNN models significantly enhances performance. SymmetryNet consistently achieved the lowest final losses across all datasets and batch sizes, demonstrating the highest accuracy. However, it required more epochs and longer training times, especially with smaller batch sizes. SimpleHydraulicCNN, while achieving the fastest training times, had slightly higher final losses. SymResNet provided a balanced performance, with moderate accuracy and training times, performing slightly better than SimpleHydraulicCNN at larger batch sizes.

The disambiguated dataset consistently resulted in the lowest and most stable validation losses across all models, indicating better performance and generalization. Both SymResNet and SymmetryNet improved with data augmentation and disambiguation, obtaining better results than SimpleHydraulicCNN.

Overall, SymmetryNet is the best-performing model with the lowest final losses. However, for applications where training time is crucial, SimpleHydraulicCNN can be utilized. SymResNet offers a good balance between accuracy and training time, while maintaining a low number of parameters. Increasing batch size for all models enhances efficiency, with SimpleHydraulicCNN being the most efficient overall, while SymResNet trades off training time for higher accuracy.

Conclusion

This report explored the modeling of laminar flow through a porous medium using convolutional neural networks (CNNs). It began with a detailed analysis of dimensionality and scaling to ensure accurate predictions. The symmetries inherent in the problem were examined, identifying rotational, flip, and translational symmetries that could be used for model improvement. The report then discussed the integration of these symmetries into CNN architectures, emphasizing their equivariance properties. Various data augmentation and disambiguation techniques were applied to enhance model robustness and generalization. Three neural network architectures were trained: SimpleHydraulicCNN, SymmetryNet, and SymResNet, with the latter two incorporating symmetry considerations. Design choices such as no group averaging of the symmetries and residual connections were introduced.

The performance of these models was analyzed across regular, augmented, and disambiguated datasets. SymmetryNet consistently achieved the lowest final losses, demonstrating the highest accuracy. SimpleHydraulicCNN was the most efficient in terms of training time, while SymResNet provided a good balance between accuracy and training efficiency. The disambiguated dataset resulted in the most stable and lowest validation losses, highlighting the effectiveness of this technique.

References

[1] "Invariant." Math is Fun, Rod Pierce, 2023, www.mathsisfun.com/definitions/invariant.html. Accessed 22 June 2024.

[2] "Invariant." *Encyclopedia of Mathematics*, European Mathematical Society, 6 May 2022, www.encyclopediaofmath.org/wiki/Invariant. Accessed 22 June 2024.

[3] "Definition: Dihedral Group D4." *ProofWiki*, ProofWiki, 19 Dec. 2023, proofwiki.org/wiki/Definition:Dihedral_Group_D4. Accessed 22 June 2024.

[4] Teh, Thomas, et al. "Generalised Structural CNNs (SCNNs) for Time Series Data with Arbitrary Graph Topology." *arXiv*, 15 Mar. 2018, arxiv.org/pdf/1803.05419. Accessed 22 June 2024.

[5] Goodfellow, Ian, et al. "Convolutional Networks." *Deep Learning*, MIT Press, 2016, www.deeplearningbook.org/contents/convnets.html. Accessed 22 June 2024.

[6] Bogatskiy, Alexander, et al. "Symmetry Group Equivariant Architectures for Physics: A Snowmass 2022 White Paper." *arXiv*, 11 Mar. 2022, arxiv.org/pdf/2203.06153. Accessed 22 June 2024.

[7] Baeldung. "Translation Invariance and Equivariance in Computer Vision." *Baeldung*, reviewed by Michal Aibin, 18 Mar. 2024, www.baeldung.com/cs/translation-invariance-equivariance. Accessed 22 June 2024.

[8] Rocke, David M. "Canonical Correlation Analysis." *University of California, Davis*, 2024, dmrocke.ucdavis.edu/Class/EAL205C/Canonical.pdf. Accessed 22 June 2024.

[9] "Lexicographic Order." *Wikipedia*, Wikimedia Foundation, 22 June 2024, en.wikipedia.org/wiki/Lexicographic_order. Accessed 22 June 2024.

[10] Moss, Alanna. "What Is Overfitting in Machine Learning?" *TechTarget*, TechTarget, May 2024, www.techtarget.com/whatis/definition/overfitting-in-machine-learning. Accessed 22 June 2024.

[11] Martin, Théo. "A (Very Short) Visual Introduction to Learning Rate Schedulers (With Code)." *Medium*, 9 July 2023, medium.com/@theom/a-very-short-visual-introduction-to-learning-rate-schedulers-with-code-189eddffdb00. Accessed 22 June 2024.

[12] Zhao, Lei, and Zhonglin Zhang. "An Improved Pooling Method for Convolutional Neural Networks." *Scientific Reports*, vol. 14, no. 1589, 18 Jan. 2024, www.nature.com/articles/s41598-024-51258-6. Accessed 22 June 2024.

[13] Singh, Amit. "Breaking the Gradient Barrier: How ResNet Rescued Deep Learning from the Vanishing Gradient Problem." *Medium*, 19 Feb. 2023, medium.com/@singhamit_/breaking-the-gradient-barrier-how-resnet-rescued-deep-learning-from-the-vanishing-gradient-problem-ffc7f33eae1a. Accessed 22 June 2024.

[14] "ChatGPT." *OpenAI, GPT-4o*, OpenAI, 2024, www.openai.com. Accessed 22 June 2024.

AI Usage

For assistance in the following sections *ChatGPT-4o* was used:

[1] Assistance in the generation of the CNN architecture image in Simple Model with No Symmetry Considerations.

[2] Assistance in exploring model architecture techniques such as depthwise-separable, residuals, and adaptive pooling.

[3] Assistance in generating an HTML table from $\LaTeX$ in Model Performances

---

Supplementary 1: Submitting Predictions to Kaggle

In [42]:
```
def to_kaggle(model):
    X_kaggle = np.load('data/pri_in.npy')
    X_kaggle_tensor = torch.tensor(X_kaggle, dtype=torch.float32).unsqueeze(1).to(device)
    y_pred_kaggle = predict(model, X_kaggle_tensor).flatten()

    df_submission = pd.DataFrame({'Id': np.arange(len(y_pred_kaggle)), 'Solution': y_pred_kaggle})
    print(f'Saving submission [{df_submission.shape}]')
    df_submission.to_csv('submission_abekis.csv', index=False)

to_kaggle(results['SymResNet']['model'])
```

```
Saving submission [(500, 2)]
```

And the total number of parameters:

In [43]:
```
sum(param.numel() for param in results['SymResNet']['model'].parameters())
```

Out[43]:
```
12880
```

Supplementary 2: Model Results Files

The results in the Results section uses two `.csv` files that are the product of multiple models run over different hyperparameters and datasets. Here the two datasets are provided.

In [44]:
```
df_model_results
```

Out[44]:

	timestamp	model_id	model_name	num_epochs	batch_size	data_type	learning_rate	final_loss	early_stop	num_params
0	2024-06-21 22:11:00.728621	SimpleHydraulicCNN_16_100	SimpleHydraulicCNN	100	16	regular	0.001	0.028463	NaN	228112
1	2024-06-21 22:11:20.510635	SimpleHydraulicCNN_16_250	SimpleHydraulicCNN	250	16	regular	0.001	0.027188	85.0	228112
2	2024-06-21 22:11:38.208347	SimpleHydraulicCNN_16_500	SimpleHydraulicCNN	500	16	regular	0.001	0.026644	71.0	228112
3	2024-06-21 22:11:42.219992	SimpleHydraulicCNN_64_100	SimpleHydraulicCNN	100	64	regular	0.001	0.028015	50.0	228112
4	2024-06-21 22:11:46.047958	SimpleHydraulicCNN_64_250	SimpleHydraulicCNN	250	64	regular	0.001	0.028002	53.0	228112
...	...	...	...	...	...	...	...	...	...	...
76	2024-06-22 05:09:56.994048	SymResNet_64_250_Dis	SymResNet	250	64	disambiguated	0.001	0.006049	50.0	12880
77	2024-06-22 05:10:29.620354	SymResNet_64_500_Dis	SymResNet	500	64	disambiguated	0.001	0.006061	52.0	12880
78	2024-06-22 05:10:51.816903	SymResNet_128_100_Dis	SymResNet	100	128	disambiguated	0.001	0.006021	53.0	12880
79	2024-06-22 05:11:13.878592	SymResNet_128_250_Dis	SymResNet	250	128	disambiguated	0.001	0.006059	51.0	12880
80	2024-06-22 05:11:37.504540	SymResNet_128_500_Dis	SymResNet	500	128	disambiguated	0.001	0.006034	56.0	12880

81 rows × 11 columns

In [45]:
```
df_losses
```

Out[45]:

	Model	Dataset	Epoch	Validation Loss
0	SimpleHydraulicCNN	regular	0	0.620265
1	SimpleHydraulicCNN	regular	1	0.563416
2	SimpleHydraulicCNN	regular	2	0.457421
3	SimpleHydraulicCNN	regular	3	0.369629
4	SimpleHydraulicCNN	regular	4	0.319354
...	...	...	...	...
1345	ResNet	disambiguated	145	0.010928
1346	ResNet	disambiguated	146	0.010927
1347	ResNet	disambiguated	147	0.010929
1348	ResNet	disambiguated	148	0.010928
1349	ResNet	disambiguated	149	0.010930

1350 rows × 4 columns

In [46]:
```
#Goodbye!
```