Software Engineering Department
Braude College

Capstone Project Phase B – 61998

Lumbar Spine Degenerative Classification using an Optimized CNN (24-2-R-1)

Supervisor:

Miri Weiss-Cohen

Submitters:

Almog Kadosh 315699439

Daniel Zada 318223278

GitHub repo: https://github.com/danielZada97/Capstone-Project-24-2-R-1

Table Of Content

1.Introduction

Low back pain ranks as the primary cause of disability globally, affecting 619 million individuals in 2020, as reported by the World Health Organization. This common affliction often arises from degenerative spinal disorders like spondylosis, which deteriorates intervertebral discs and leads to spinal stenosis and nerve compression. LBP affects individuals of all ages, peaking at 50-55 years and being more prevalent in women. Chronic LBP significantly impacts older adults aged 80-85, causing work loss, reduced quality of life, and economic burden. This study evaluates standardized definitions of degenerative changes to reduce variability in lumbar spine Magnetic resonance imaging (MRI) interpretations among subspecialty-trained doctors and explores using an AI-trained model to classify lower back problems using MRI images and LBP classifications. [1]

The human back is composed of several key components. The spine, which consists of thirty-three vertebrae (Fig. 1), is divided into four regions: 7 cervical vertebrae (C1-C7), 12 thoracic vertebrae (T1-T12), 5 lumbar vertebrae (L1-L5), and fused sacral and coccygeal vertebrae. Intervertebral discs act as cushions between the vertebrae, providing shock absorption and flexibility. Various muscles, including the erector spinae, trapezius, and latissimus dorsi, support and move the spine. Ligaments connect bones and stabilize joints, while facet joints between vertebrae allow limited movement and contribute to spinal stability. Finally, the spinal cord runs through the vertebrae, with nerve roots branching out to innervate different body parts [1].

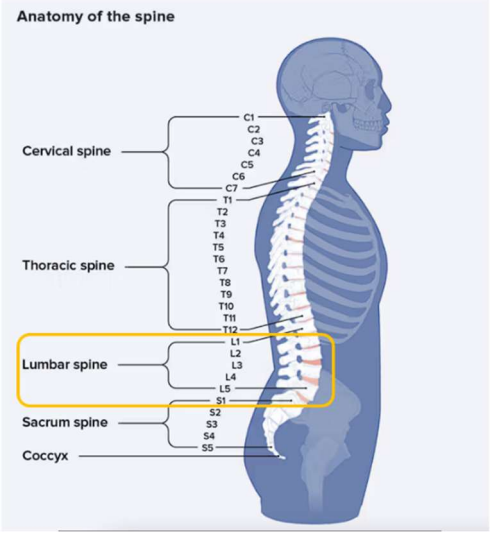


Fig.1 human spinal skeleton showing cervical, thoracic , lumbar and saccro-coccygeal regions

Back pain can be caused by a variety of factors, including both lifestyle choices and underlying medical conditions. Risk factors such as weak core muscles, obesity, physically demanding jobs, and chronic stress can all contribute to back pain. Age, genetics, and even improper lifting techniques can also play a role. Back pain can also stem from structural problems in the spine, discs, muscles, or ligaments, such as strains, sprains, and degenerative disc disease. In some cases, inflammatory conditions like ankylosing spondylitis or other medical conditions like osteoporosis, fibromyalgia, or even kidney stones can be the culprit. While the specific cause of back pain may not

always be clear, understanding these various risk factors and potential underlying conditions is a crucial first step in addressing and preventing back pain.[3]

Magnetic resonance imaging (MRI) is a key tool for diagnosing lumbar spine problems. It provides clear images of the spine's structures, such as vertebrae, discs, and nerves, helping detect conditions like spinal stenosis, nerve compression, and degeneration. By using MRI, we can accurately identify these issues and develop effective treatment plans.

The main problem in diagnosing lumbar spine issues is the lack of a clear system for interpreting MRI scans. Different doctors, like neuroradiologists and musculoskeletal radiologists, often interpret the same scans in different ways because they focus on different details. This makes diagnoses less reliable and can delay proper treatment. To solve this, a standard system is needed to ensure consistent and accurate evaluations.

Our goal is to classify lumbar spine degenerations using CNN to improve the efficiency and accuracy of evaluating MRI images. By automating the diagnostic procedure, we aim to provide assisting medical frameworks to ease physicians' workloads. As a result, we aim to significantly reduce the diagnostic time for patients, resulting in more efficient and effective treatment plans for conditions such as spinal degeneration.

2.Background

2.1 Magnetic Resonance Imaging:

Magnetic Resonance Imaging (MRI) is a medical tool that helps doctors see inside the body without surgery. It uses strong magnets and radio waves to create detailed images of organs, tissues, and other structures. When the MRI is done, sensors pick up signals from the body and turn them into pictures. To make the images clearer, doctors sometimes use a contrast dye like Gadolinium. Patients need to stay still during the scan to ensure the images are accurate.

MRI is very useful for looking at the spine because it shows soft tissues like discs, ligaments, and nerves clearly. It helps doctors find problems such as spinal stenosis, nerve compression, or damage to the discs. MRI also shows inflammation or fluid in the spine, which is important for planning treatments. This makes it a key tool for diagnosing back problems and deciding on the right care.[8]

However, MRI has some limits. It doesn't show bones as well as CT scans because bones don't give off the signals that MRI can detect. Instead, bones appear as black areas in the images. If doctors need to check for bone fractures or injuries, they usually use CT scans, which are better at showing the details of bones. MRI and CT scans often work together to give a full picture of the spine's condition.

Doctors often use MRI to grade and classify spine problems, such as disc degeneration, to decide how severe the issue is. They use systems like the Pfirrmann grading scale for this. However, reading and analyzing MRI scans takes a lot of time, and doctors sometimes face heavy workloads. This can cause delays in diagnosis and treatment, which is frustrating for both doctors and patients.

Another problem with MRI is the chance of human error. Studies show that many mistakes in radiology happen because important details are missed in the images. This shows how hard it can be to correctly diagnose problems using MRI alone. To fix this, technology like artificial intelligence can help doctors by quickly analyzing MRI scans. This can save time, reduce mistakes, and improve the way spinal problems are diagnosed and treated.

Fig 2 MRI of the lower-back

2.2 Convolutional Neural Networks-CNN:

Convolutional Neural Networks (CNNs) are a sophisticated subset of deep learning frameworks specifically engineered to analyze visual data efficiently. CNNs are adept at automatically learning spatial hierarchies of features through multiple layers, making them particularly effective for tasks involving image and video processing. The architecture of a CNN usually involves several convolutional layers which help in extracting various features from the images. These layers are often mixed with pooling layers that reduce the spatial size of the representation, thus reducing the parameter counts and computation in the network. This setup is typically followed by one or more fully connected layers that perform the high-level reasoning in the neural network (Fig .3).

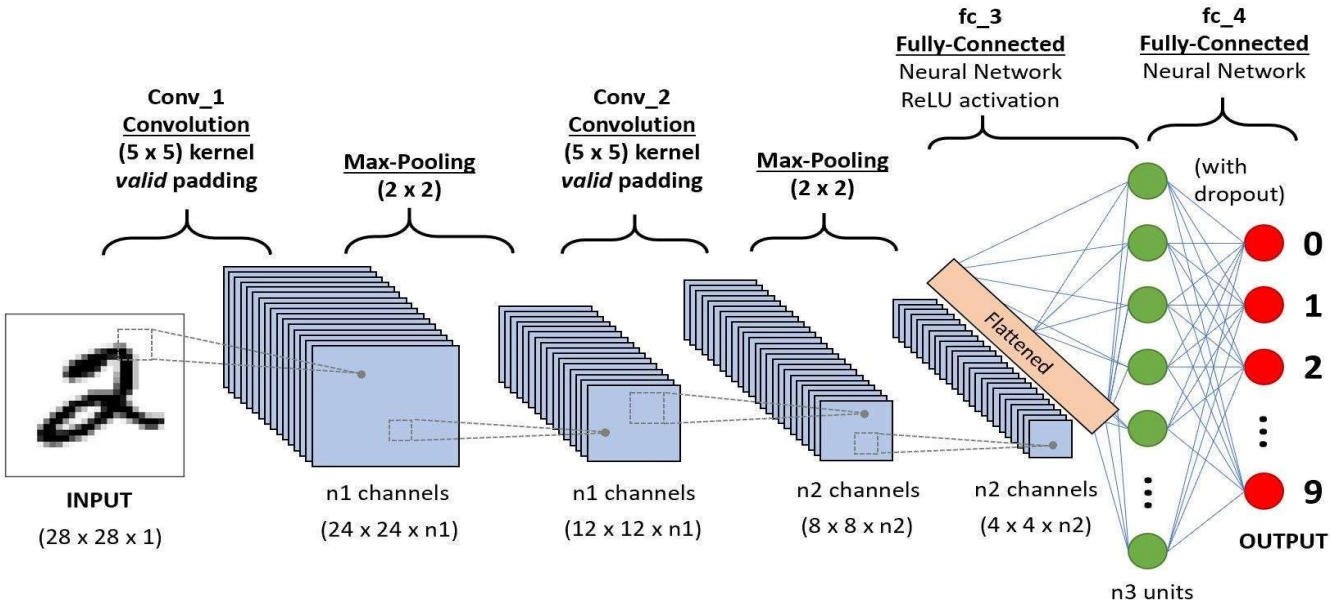


Fig 3 A traditional CNN sequence to classify handwritten digits

Convolutional layers: responsible for extracting spatial hierarchies of features from input images. They use filters or kernels (small matrices) that slide across the image. Each filter detects a specific feature like edges, curves, corners, or textures. Each filter result in a feature map which is a new image that highlights where did the filter detect certain features in the original image. CNNs usually use many filters in a single convolutional layer, each detecting a different feature we want to detect, which leads to multiple feature maps giving the network a better understanding of the input image. By stacking multiple convolutional layers, the network can detect more complex features, starting with simple patterns in the early layers and building up to more complex shapes and objects in deeper layers.(Fig.4)

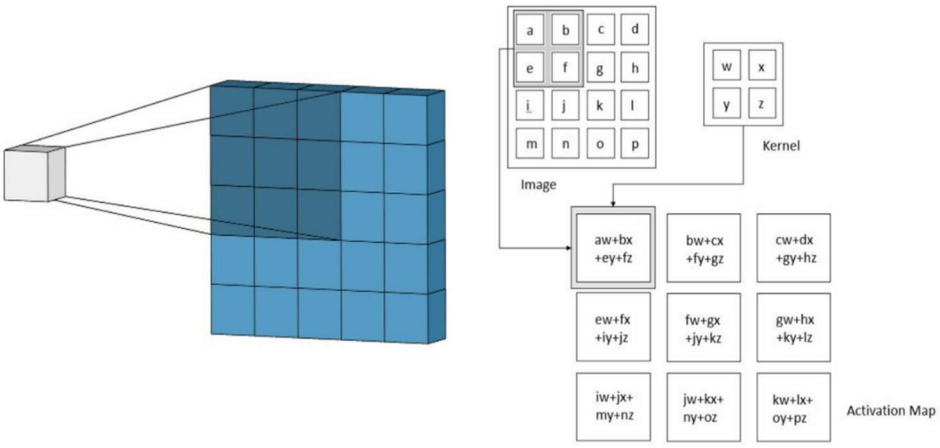


Fig 4: Appling filter over image process

Pooling layers: max pooling works by taking a small region from the feature map (usually 2X2) and selecting the maximum value from this region. This is done for every region as it slides over the entire feature map. Max pooling plays a crucial role in reducing the spatial dimensions of the feature maps while retaining essential information. (Fig. 5).

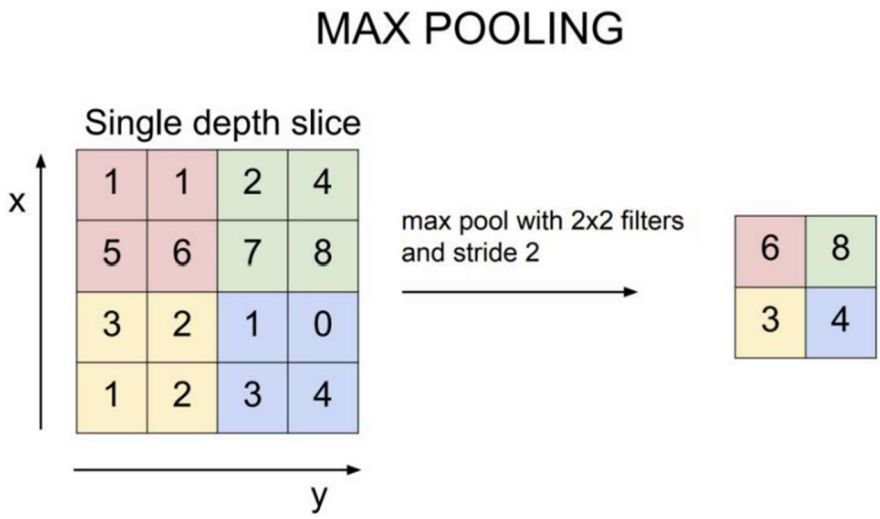


Fig 5: Example of max pooling operation of 2x2 filters and stride 2

Fully connected layers: the final layers of a CNN. They take the high-level features learned from the previous layers and map them to specific classes or outputs. By leveraging these extracted features, fully connected layers can make predictions or perform classification tasks by assigning weights to each feature they received and determine which class is best suited as a result.

Additionally, Neural networks leverage several types of activation functions to introduce non-linearities and enable learning complex patterns. Non-linearity is crucial for modeling real-world relationships between objects that do not have straightforward, linear connections and uses an activation function that introduces nonlinearity.
The most widely used activation function in CNNs, is ReLU as defined in Eq. 1.

$$f(x) = \max(0, x)$$

The ReLU function thresholds inputs at zero, outputting 0 for negative values and passing positive values unchanged, acting linearly with a gradient of 1 for inputs above zero. This characteristic is essential in avoiding the vanishing gradient problem during backpropagation.

2.3 Transfer learning

Transfer learning (TL) with convolutional neural networks (CNNs) improves performance on new tasks by using knowledge from tasks already learned. It is particularly useful in medical image analysis, as it solves the problem of limited data and reduces the time and resources needed for training. For this project, we use a pretrained CNN model that was trained on ImageNet for image classification, leveraging its existing weights to classify lumbar spine degenerations.
In inductive transfer learning, tasks in the source and target domains are different but share similarities. For example, a neural network trained on general images like ImageNet can be adapted to diagnose medical conditions. By reusing the features it has already learned, the model becomes more effective in the new, specialized task. (Fig. 6)
Multitask learning is another method where one model is trained to handle multiple tasks, such as recognizing objects and classifying scenes. This approach helps the model identify shared patterns across tasks, improving its efficiency and ability to generalize.
In this project, the pretrained model is fine-tuned using a dataset of MRI scans to adapt it for the specific task of classifying lumbar spine conditions. Fine-tuning involves adjusting parameters like the learning rate and training epochs, allowing the model to retain useful features from its original training while optimizing performance for the new task. This process ensures the model delivers accurate and efficient results for medical image classification.

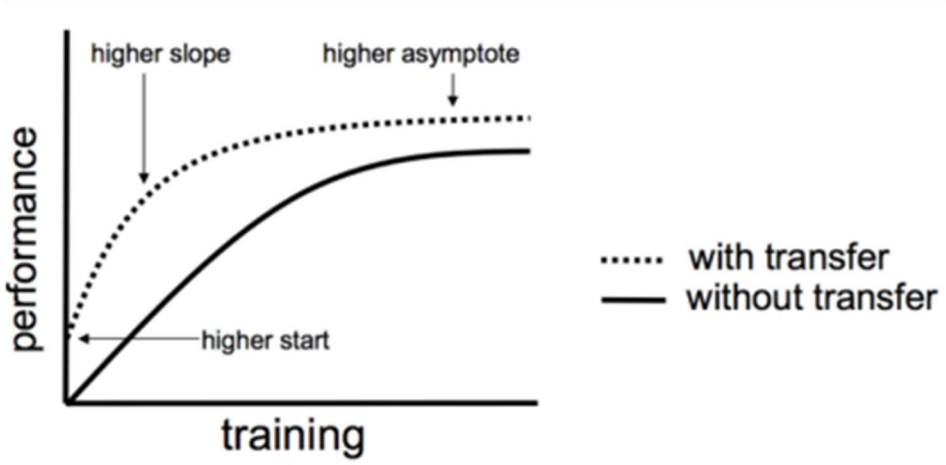


Fig 6: traditional machine learning vs. transfer learning

2.4 Loss function

in machine learning, loss function measures how well a model's predictions match the actual data. It quantifies the difference between the predicted values and the actual values - the true probability value of each sample will be 1 for the class that it belongs to and 0 for the rest of the classes. Loss function provides a metric that the model aims to minimize during training. The loss function guides the optimization process by indicating how far off the predictions are, the goal is to adjust the model parameters to reduce this loss.

Cross Entropy loss function

Cross-entropy, also known as logarithmic loss or log loss, is a loss function used in machine learning to measure the performance of a classification model.
cross-entropy measures the difference between the discovered probability distribution of a classification model and the predicted values.
The cross-entropy loss function is used to find the optimal solution by adjusting the weights of a machine learning model during training. The objective is to minimize the error between the actual and predicted outcomes. A lower cross-entropy value indicates better performance.

For multi-class classification problems, where an instance could belong to one of many classes, the cross-entropy loss is generalized to Eq. 1:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} * \log(\hat{y}_{i,c})$$

Eq.1- multi-class cross entropy equation

where:

- C is the number of classes.
- $y_{i,c}$ is a binary indicator (0 or 1) if class label c is the correct classification for instance i.
- $\hat{y}_{i,c}$ is the predicted probability of instance i being of class c.
- N is the number of instances in the data set

Each inner sum computes the loss for each class label per sample by taking the logarithm of the predicted probability for the true class and summing these values across all classes. The result is then averaged over all samples .

3. Dataset

For our project, we are using the RSNA 2024 Lumbar Spine Degenerative Classification dataset, provided by the Radiological Society of North America (RSNA) in collaboration with the American Society of Neuroradiology (ASNR) through a Kaggle competition. This extensive dataset is designed to improve artificial intelligence (AI) in detecting and classifying degenerative spine conditions using lumbar spine MRI images.

The dataset includes 147,320 files totaling 35.34 GB, comprising both DICOM (DCM) MRI images and CSV files with classification results and annotations. It covers five specific degenerative conditions: Left Neural Foraminal Narrowing, Right Neural Foraminal Narrowing, Left Subarticular Stenosis, Right Subarticular Stenosis, and Spinal Canal Stenosis. Each condition is evaluated at various intervertebral levels. The severity of these conditions is categorized into three levels: Normal/Mild, Moderate, and Severe, allowing for detailed analysis and classification. The data is split into training and test sets, with train.csv providing labels and annotations for the training images.

To prepare the data for our project, we developed a Python script to convert the DICOM files into PNG format. This conversion makes it easier to display and work with the MRI images in our project, as shown in Figure 7. The RSNA challenge planning team collected this dataset from nine sites across five continents, and over fifty experts carefully labeled the images to ensure high-quality and consistent annotations. This collaborative effort provides a solid foundation for developing accurate and reliable AI models for classifying lumbar spine degeneration.

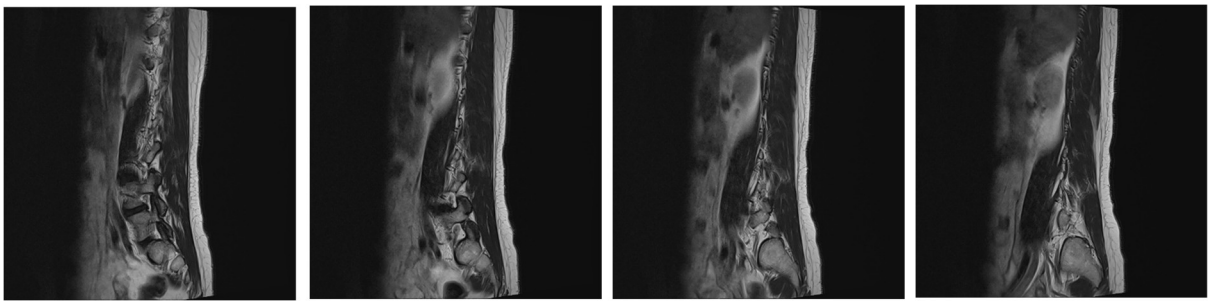


Fig 7 MRI of the lower-back from Kaggle dataset.[3]

The dataset includes three types of MRI sequences: Sagittal T1, Sagittal T2/STIR, and Axial T2. Sagittal T1 images offer detailed views of the spinal anatomy by highlighting bone marrow and soft tissues, making it easier to see structural details. Sagittal T2/STIR sequences enhance the visibility of fluids and edema, which helps in identifying inflammation and lesions. Axial T2 images provide cross-sectional views of the spine, assisting in the assessment of the spinal canal and foraminal spaces. These different sequences provide comprehensive information that supports accurate diagnosis and classification of spine conditions.

Using this dataset, our project focuses on training and testing AI models to accurately classify the severity of lumbar spine degenerative conditions. By analyzing the Sagittal T1, Sagittal T2/STIR, and Axial T2 MRI sequences, our models aim to replicate expert diagnostic capabilities, ultimately contributing to better patient outcomes through improved diagnostic tool

4. General description

In this project, we created an AI-based system to make it easier to analyze MRI images for patients with lumbar spine degenerative diseases. These diseases are often difficult to diagnose because analyzing MRI scans takes time and can vary between doctors. Our system was designed to help solve these problems by providing fast and accurate classifications.

The system uses the DenseNet deep learning model and transfer learning, which helps improve accuracy and reduces the amount of training needed. It can classify MRI scans into five specific conditions: Left Neural Foraminal Narrowing, Right Neural Foraminal Narrowing, Left Subarticular Stenosis, Right Subarticular Stenosis, and Spinal Canal Stenosis. These conditions are analyzed at different levels of the lumbar spine area (Fig.1) , from L1/L2 to L5/S1, giving detailed and specific results.

To make the system easy to use, we built a user-friendly interface. The backend was developed using Python, which handles tasks like processing images and making predictions with the AI model. The frontend was created with React, allowing users to upload MRI images and see the classification results clearly. This combination ensures that the system works efficiently and is simple for doctors to use.

The main goal of the system is to help doctors diagnose these spine conditions faster and more accurately. By automating the classification process, it reduces the chances of mistakes and allows doctors to focus on deciding the best treatment for their patients. This can also help patients by getting them the right treatment sooner and improving their outcomes.

The system is meant for medical professionals, such as radiologists, neurologists, and orthopedic surgeons, who work with MRI images. It provides them with a reliable tool to standardize and speed up their work while reducing errors.

In conclusion, this project combines AI and a well-designed interface to improve how lumbar spine conditions are diagnosed. By simplifying the process and providing consistent results, it helps both doctors and patients achieve better outcomes.

5.Describing the solution

For this project, we developed an AI-based system that uses the DenseNet deep learning architecture combined with transfer learning to classify MRI images of the lumbar spine into specific degenerative conditions.

5.1 How the system works:

1. **Input and dataset (MyDataset.py)**:
   The system takes MRI images of the lumbar spine from the Kaggle dataset [2] as input. These images include different views like axial T2, sagittal T1, and sagittal T2 sequences (Fig. 8), which provide comprehensive anatomical details. MRI scans show the spine in different ways to help identify problems.
   These views work together to give a full picture of the spine.
   In order to preprocess and handle the data efficiently we made a custom dataset class in python called MyDataset.py that access the images in the dataset locally and retrieve the images according to the MRI scan type as mentioned previously.

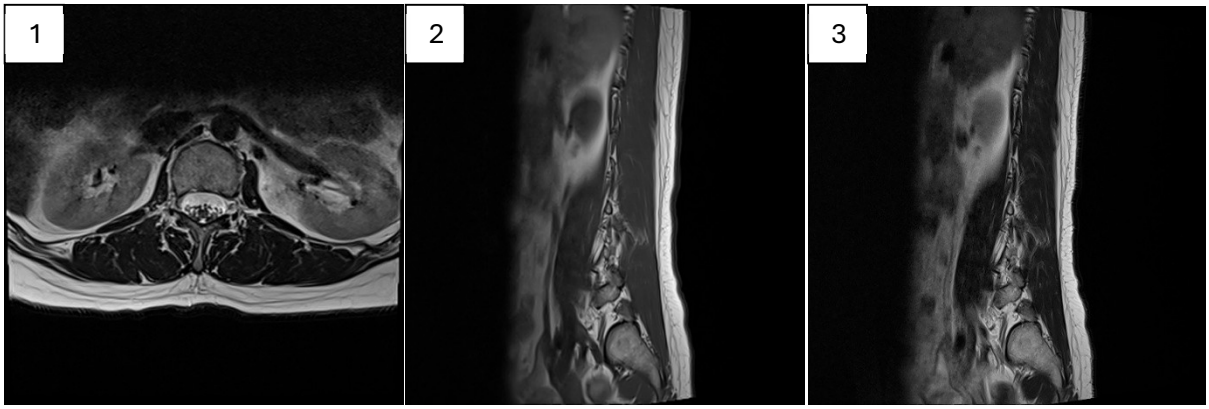


Fig.8 Axial T2 (indicated by number 1), Sagittal T1 (indicated by number 2) and Sagittal T2 (indicated by number 3) MRI images of the lumbar spine.

Preprocessing: The preprocessing pipeline prepares MRI images by resizing them to 512x512 pixels, normalizing pixel values for consistent scaling, and combining multiple image sequences into a single 30-channel tensor. During training, augmentations such as brightness adjustments, blurring, distortion, and cropping are applied (using the Albumentations library in python) to improve the model's ability to generalize and handle variability in the data. These steps ensure the input is clean, consistent, and diverse, enabling the DenseNet model to perform effectively.
In addition, for the image classifying process we also used the same transformations for input images.

2. **Model Architecture:**

DenseNet-121: In our project, we used DenseNet-121, a type of convolutional neural network (CNN) known for its smart and efficient design. Unlike other CNNs, DenseNet-121 connects every layer to all the layers before it. This means each layer can reuse

features learned by previous layers, which makes the model more efficient and reduces unnecessary calculations (Fig. 9). It also helps solve the problem of vanishing gradients, allowing the model to train effectively even with many layers. Overall, this design makes DenseNet-121 powerful and lightweight compared to other deep learning models.

To fit our project needs, we customized DenseNet-121. The original model processes 3-channel images (like RGB photos), but we modified it to handle 30 input channels to include MRI data from sagittal T1, sagittal T2, and axial T2 views. We also adjusted the model to predict 75 classes, representing five spinal conditions at five spinal levels (L1/L2 to L5/S1) and three levels of severity: Normal/Mild, Moderate, and Severe. This setup allowed DenseNet-121 to efficiently classify MRI images and provide detailed diagnostic results.

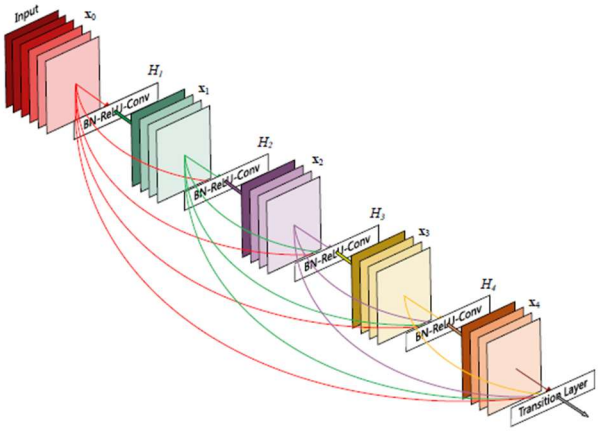


Fig.9 5-layer dense block with a growth rate of k = 4. Each layer takes all preceding feature-maps as input [3] ** change the image

Transfer Learning: In our project, we used transfer learning by starting with a DenseNet-121 model pre-trained on ImageNet, a large dataset of general images. This approach allowed us to take advantage of the model's prior knowledge, such as recognizing edges, textures, and patterns, which are common across many types of images. By doing this, the model didn't need to learn basic features from scratch, saving training time and making it more effective for our smaller medical dataset.

We fine-tuned the pre-trained model to adapt it specifically for classifying lumbar spine MRI images. This involved adjusting the model's weights while keeping its previously learned features. Only the final layers of the network were trained fully from scratch, ensuring the model focused on the unique patterns found in medical images. Transfer learning not only improved accuracy but also helped the model perform and train faster.

3. Training process(Training.py)

Loss function: we used weighted cross-entropy loss to handle the imbalance in class distribution within the dataset. Some classes, like "Severe," appeared less frequently compared to others, such as "Normal/Mild"(Fig. 10). Without adjustments, the model might focus more on the common classes and struggle to correctly predict the rarer ones.

To solve this, we assigned higher weights to less frequent classes and lower weights to more common ones. This encouraged the model to pay more attention to the underrepresented classes during training. The cross-entropy loss itself measures the difference between the model's predicted probabilities and the true labels, helping the model learn to make accurate classifications. By adding weights, we ensured fair learning across all classes, improving the model's overall performance, especially for rarer but critical conditions.

D	C	B	A	
spinal_canal_stenosis_l3_l4	spinal_canal_stenosis_l2_l3	spinal_canal_stenosis_l1_l2	study_id	1
Normal/Mild	Normal/Mild	Normal/Mild	4003253	2
Moderate	Normal/Mild	Normal/Mild	4646740	3
Normal/Mild	Normal/Mild	Normal/Mild	7143189	4
Normal/Mild	Normal/Mild	Normal/Mild	8785691	5
Normal/Mild	Normal/Mild	Normal/Mild	10728036	6
Normal/Mild	Normal/Mild	Normal/Mild	11340341	7
Normal/Mild	Normal/Mild	Normal/Mild	11943292	8
Normal/Mild	Normal/Mild	Normal/Mild	13317052	9
Normal/Mild	Normal/Mild	Normal/Mild	22191399	10
Normal/Mild	Normal/Mild	Normal/Mild	26342422	11
Moderate	Moderate	Normal/Mild	29931867	12
Normal/Mild	Normal/Mild	Normal/Mild	33736057	13
Normal/Mild	Normal/Mild	Normal/Mild	38281420	14

Fig.10 sample from train.csv from Kaggle dataset [

Optimizer and Scheduler: we used the AdamW optimizer because it decouples weight decay from gradient updates, providing stability and effectively handling sparse gradients. Additionally, we employed a cosine learning rate scheduler with a warm-up phase to gradually ramp up the learning rate, then smoothly decay it, leading to more consistent and reliable convergence.

Cross-Validation: The dataset was divided into three folds for training and validation, a strategy often referred to as "k-fold cross-validation." This approach ensures that each fold serves as both a training set and a validation set at different stages, helping the model learn from varied data splits and reducing the risk of overfitting.

Augmentation: Albumentations library was used to systematically apply data augmentation techniques such as brightness adjustments, random blurring, and rotations to the training images. These transformations help the model become more robust by exposing it to varied lighting conditions, slight movements, and different viewing angles, effectively increasing the diversity of the training data.

Validation and Evaluation: At the end of each training epoch, the model was evaluated on the validation set to assess its performance. Metrics such as validation accuracy and loss were carefully tracked to measure how well the model generalized to new data. Monitoring these metrics was critical to identifying potential issues, such as overfitting, where the model performs well on training data but poorly on validation data. If the validation metrics indicated a drop in performance, it signaled the need for adjustments to training strategies or hyperparameters, such as the learning rate or batch size. This step ensured the model not only learned from the training data but also retained its ability to perform consistently on new data.

Model Checkpointing: To ensure the best version of the model was preserved, model checkpointing was implemented. After each validation cycle, the model's weights were saved if the validation accuracy improved compared to previous epochs. By saving only the best-performing weights, we guaranteed that the final model represented its most optimal state during training. This approach prevented the loss of progress and allowed us to select the best-trained model for deployment without having to retrain from scratch. Checkpointing also made the process more efficient by ensuring that only meaningful improvements were captured and stored. After each fold a CSV file was saved in order to save to results of the training, and allowing us to showcase the training and validation results in a graph.

4. **Classification process (Classification.py):**

The classification process in the code was designed to analyze MRI images and produce detailed predictions about lumbar spine conditions and their severity. It involved a sequence of steps to prepare the input, use the trained DenseNet-121 model for predictions, and interpret the results for clear and actionable output.

Input Preprocessing: Before classification, MRI images were preprocessed to match the model's input requirements. The code normalized pixel values and resized the images to 512x512 pixels. Multiple MRI sequences, such as sagittal T1, sagittal T2, and axial T2, were stacked to create a 30-channel input tensor. This ensured the model had access to comprehensive anatomical data for accurate classification. Preprocessing also included converting images into tensors compatible with PyTorch for efficient processing by the model.

Inference with the Trained Model: The prepared input data was fed into the DenseNet-121 model, which was trained specifically for this task. The model analyzed the input and produced raw scores, called logits, for each of the 75 possible outputs. These scores represented how likely each condition and severity was for the five spinal levels (L1/L2 to L5/S1).

Postprocessing and Predictions: The logits were converted into probabilities using a softmax function, which made it easier to determine the most likely condition and severity. These probabilities were then matched to their corresponding labels, such as "Normal/Mild," "Moderate," or "Severe," to make the results easy to understand.

Filtering and Output Generation: To keep the output focused, predictions labeled as "Normal/Mild" were filtered out, and the results highlighted only "Moderate" or "Severe" cases. This made it easier for doctors to quickly identify the more critical conditions. The results were presented in a clear report, showing detailed information about each condition and spinal level.

6. Tools and Libraries

The first major decision we faced was choosing a programming language for machine learning. After researching several options like Python, JavaScript, Java, C++, and Julia, we decided on Python [5] due to its simplicity, extensive ecosystem of machine learning libraries, and strong community support. Once we chose Python, we explored the two most popular machine learning frameworks, PyTorch and TensorFlow. After comparing their advantages and disadvantages, we selected PyTorch [10] for its simplicity, easier learning curve, better performance in benchmark tests, efficient GPU usage, and widespread adoption in modern research and projects. Our project was built using Python 3.11, which provided a fast and reliable environment for tasks like data processing, model training, and visualization. Python's ease of use, readability, and compatibility with libraries like PyTorch, Pandas, and NumPy made it the ideal choice for our MRI classification system.

The libraries that were used in this project:

1. **PyTorch**
   PyTorch was the main tool we used for our project, providing everything we needed to create, train, and test our DenseNet-121 model. It allowed us to customize the model to handle 30 input channels and 75 output classes, making it perfect for MRI classification. PyTorch's Dataset and DataLoader helped us manage large datasets by batching and preparing data as needed. Its automatic system for calculating gradients made the training process easier. The AdamW optimizer ensured stable updates to the model's weights, and with PyTorch's GPU support, we used CUDA to speed up training significantly. Features like mixed precision training and gradient accumulation helped reduce memory usage and make training more efficient. PyTorch's flexible design also made it easy to debug and try different settings, which helped us improve the model for accurate MRI classification.

2. **Pandas**
   Pandas is a data manipulation and analysis library widely used in data science and machine learning projects. Pandas helped us organize and manage the dataset. It was used to load the CSV files containing information about MRI images and their labels. Tasks like cleaning missing values, converting labels into numerical categories, and preparing data for training and testing were done with Pandas.

3. **NumPy**

   NumPy is a fundamental library for numerical computing in Python, known for its support of multi-dimensional arrays and high-performance mathematical operations. NumPy was used for working with arrays. It allowed us to combine MRI slices into a single tensor (30 channels) and perform mathematical operations efficiently.

4. **Glob**

   Glob is a standard Python library for pattern matching in file paths, often used to locate files in directories. Glob was used to find and load image files stored in folders. It allowed us to easily locate specific files, such as sagittal T1, sagittal T2, and axial T2 slices, without manually specifying file paths.

5. **Pillow (PIL)**

   Pillow is a popular Python library for image processing tasks like opening, resizing, and converting image formats. Pillow was used to handle images, including loading, resizing, and converting them to grayscale. It ensured that the images were in the correct format before feeding them into the model.

6. **TDQM**

   tqdm is a lightweight library that provides progress bars for loops, making it easier to track the progress of long-running tasks. This library provided progress bars for tasks like loading data, training and validating the model.

7. **Scikit-learn (sklearn)**

   Scikit-learn is a popular library for machine learning tasks, offering tools for data preprocessing, model evaluation, and metrics calculation. In our project, it was used to split the dataset into training and testing sets using cross-validation and evaluate the model using metrics like accuracy.

8. **Timm**

   Timm is a library that provides a wide range of pre-trained models for computer vision tasks, making it easy to use state-of-the-art architectures. It allowed us to use a pre-trained DenseNet-121 model, which we fine-tuned for our specific task, saving time and improving model accuracy.
   Timm allowed us to define the models hyperparameters easily and efficiently, For example the dropout rate we said we were going to use in phase a of the project.

9. **Transformers**

   Transformers is a library developed by Hugging Face for state-of-the-art natural language processing and computer vision models. Though not heavily implemented in this project, it was explored as an option for using advanced architectures to enhance model performance in future iterations.

10. **Matplotlib:**
    Matplotlib is a widely used plotting library for creating static, interactive, and animated visualizations. In our project, it was used to generate graphs showing the model's performance, such as training accuracy and loss curves, making it easier to understand and present results.

11. **Albumentations**:
    Albumentations is a fast and flexible library for image augmentations, widely used in computer vision tasks. It allowed us to apply transformations to MRI images, such as resizing, normalization, brightness adjustments, and adding noise. These augmentations were essential for improving the model's robustness and generalization by simulating variations in the data. Albumentations made preprocessing efficient and ensured that the model could handle diverse input conditions during training.

In addition, we used Google Colab Pro for our project, leveraging its A100 runtime, which includes high-performance GPUs, extra RAM, and fast storage. This setup was proficient for handling our large MRI dataset and significantly sped up the training of our DenseNet-121 model. Colab Pro's longer runtimes and easy integration with Google Drive allowed us to train complex models without interruptions.

The next step after developing the classification model and logic was to create a user-friendly interface for easy classification. For this, we chose **React** [11] as the frontend framework. React is flexible, efficient, and well-suited for building interactive interfaces. Its component-based design allowed us to create reusable elements, and its ability to quickly update and render components ensured a smooth user experience. These features made React an excellent choice for developing the user interface for our project.

For the backend, we used **Flask**, a lightweight and flexible web framework. Flask is simple to use and integrates well with Python, making it ideal for connecting our machine learning model to the user interface. In our project, Flask handled tasks like processing user requests, passing MRI images to the model, and sending the classification results back to the React frontend. Its built-in tools made setting up endpoints and managing data easy, ensuring smooth integration between the frontend and backend.

7. Challenges

During this project, we faced several challenges related to data processing, resource limitations, and fine-tuning our model. One major challenge in the beginning of the process was converting the DICOM images into PNG images while organizing the data in a structured way. Handling large datasets with varying formats required careful preprocessing, including resizing and normalization, to ensure consistency. Managing these operations while maintaining the integrity of the data proved to be time-consuming and computationally demanding.
in order to tackle this challenge we made a data preprocessing script (data_preprocessing.py) dedicated to converting and organizing the images according to the study id and image type (sagittal T1, sagittal T2, and axial T2).

Another significant challenge was the limitation of hardware resources. Our initial setups lacked sufficient RAM and GPU power to efficiently process the data or train the model. This pushed us to seek alternative solutions, including cloud-based resources, which introduced additional considerations for managing large file transfers and maintaining project environments. eventually to overcome this we used the google Colab pro services available online, which let us as described earlier to use a cloud environment with better resources that allowed us to make the training process faster and in more efficient way.

Dealing with the variability in the dataset was also difficult. MRI images differed in resolution and orientation, requiring consistent preprocessing to make the data usable. Additionally, class imbalance posed a challenge, as some categories of conditions were underrepresented. Balancing the dataset through augmentations like flipping or rotating images added complexity to the preprocessing pipeline and required significant processing power.in order to introduce non-linearity and help the model to train better and recognize features in a more elaborate way was to use the Albumentations library in python, which helped us introduce augmented images for the training phase.

Through these difficulties, we gained a deeper understanding of the complexities involved in preprocessing, structuring data, and training deep learning models in a resource-constrained environment. These experiences helped us develop a more robust and scalable approach to completing the project.

8. Project results

In this phase of the project, we focused on improving our system by testing and fine-tuning the model to find the best hyperparameters for optimal performance. We adjusted various parameters in the code and ran the program multiple times to measure its performance. For each run, we recorded key metrics such as accuracy, loss, F1 score, precision, and recall. These results were saved in a CSV file for easy reference and analysis, and we later used them to create graphs that helped us compare the outcomes.

To test the model, we experimented with different settings:

- Epochs and Folds: We tested training the model for 50 and 100 epochs, using 5 folds for cross-validation in each case.
- Batch Size: We tried two batch sizes: 32 and 64.
- Learning Rate: We evaluated multiple learning rates, including 0.001, 0.00001, and 0.0000001.
- Dropout Rate: We experimented with dropout rates of 0.2 and 0.5 to try to prevent overfitting.

To handle the imbalance in the dataset, we calculated class-specific weights based on the relative frequency of each class. The weights are set as weights =0.43, 2.04, 5.27, where the most frequent class ("mild/Normal") is given the smallest weight (0.43), and the least frequent classes ("moderate" and "severe") are given higher weights (2.04 and 5.27, respectively). This ensures that the model pays more attention to the underrepresented classes by penalizing errors on them more heavily during training. We used these weights in the cross-entropy loss function. This approach helps balance the model's learning and reduces the risk of it being biased toward the majority class.

After testing various combinations of these settings, the best results were achieved with a learning rate of 0.00001, 50 epochs, and a batch size of 32. This configuration produced a validation accuracy of 83%, which we selected as our final model. The consistent approach to testing and recording results allowed us to confidently identify the most effective hyperparameters and improve the model's performance.

Below we will present the results, for both training and validating, and showing the graphs for the best fold of each learning rate combination:

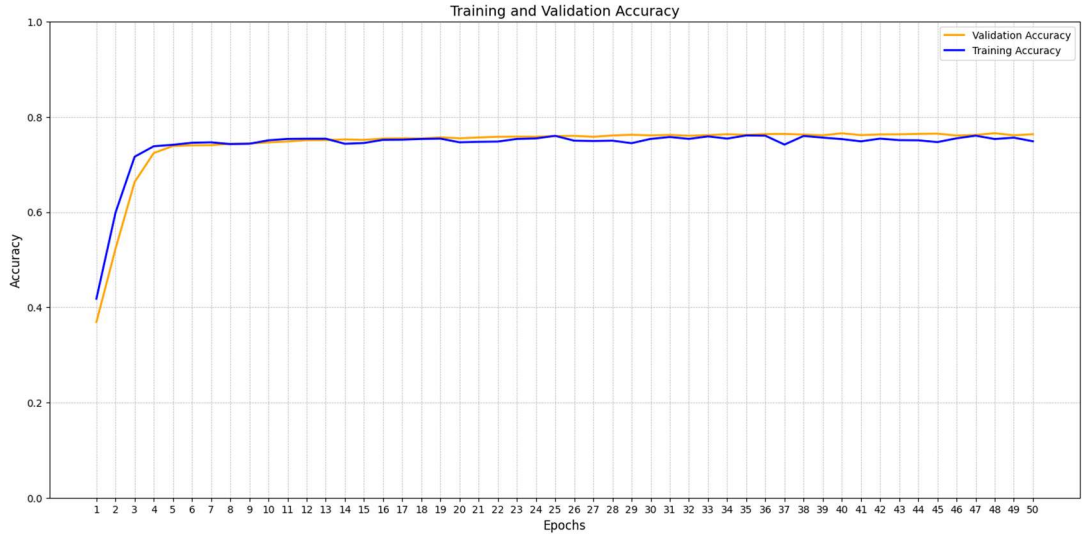
9. Results

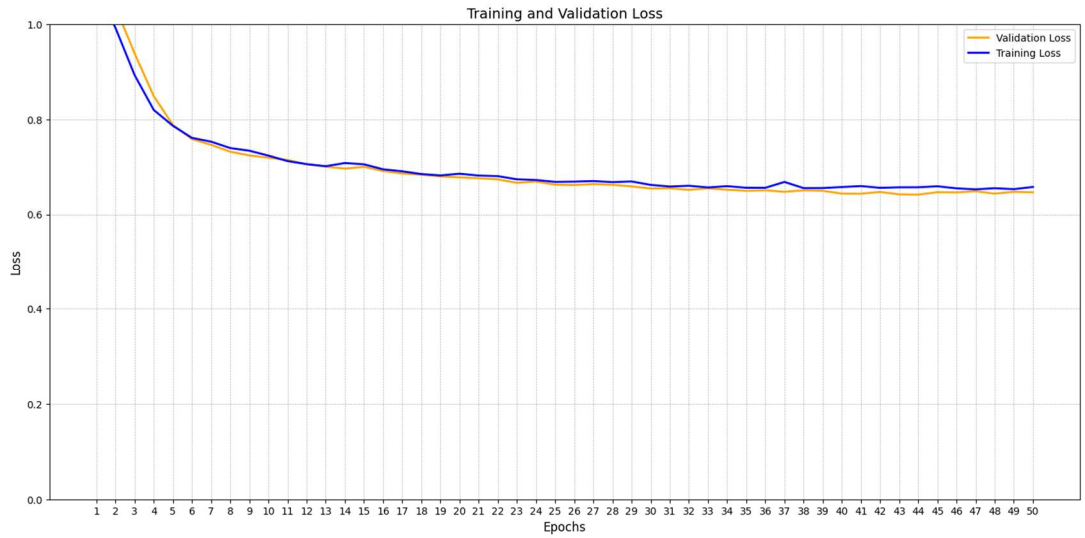
9.1 Result graphs

Below are the results of the training process:

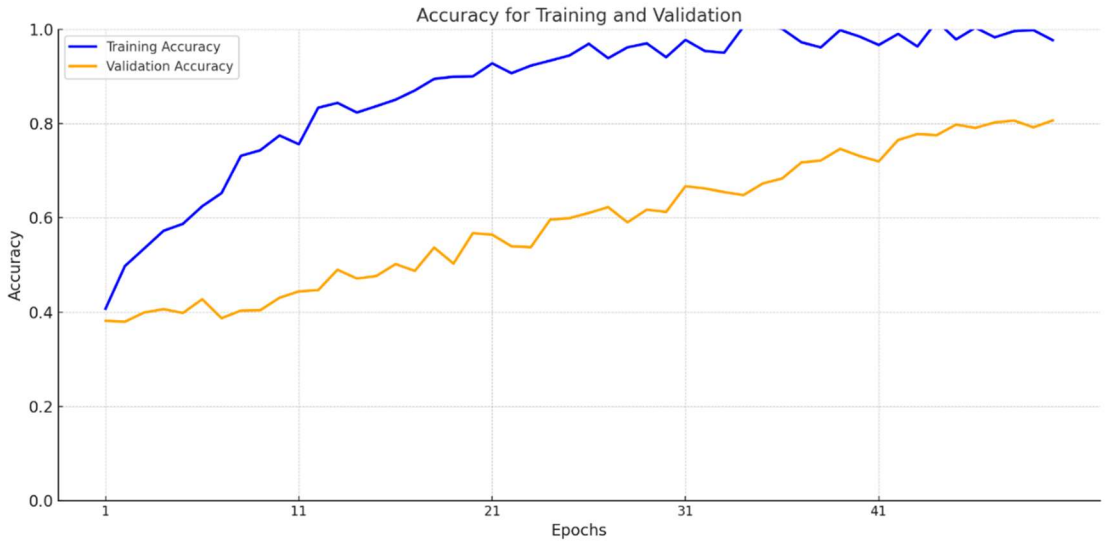
We tested different hyperparameters, mainly epochs (50 and 100) and learning rate(0.001,0.00001,0.0000001. Batch size changed, using 32 and 64, the result was very close to each other. And lastly, we tested different dropout rates (0.2 and 0.5)

Batch size: 32, Learning rate: 0.0001, Drop rate: 0.2, Epochs:50

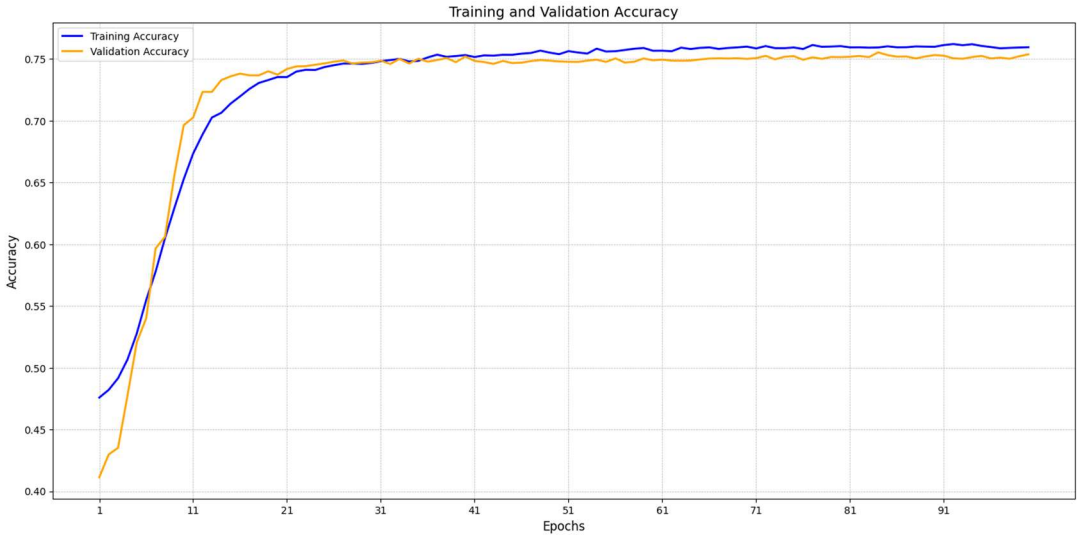


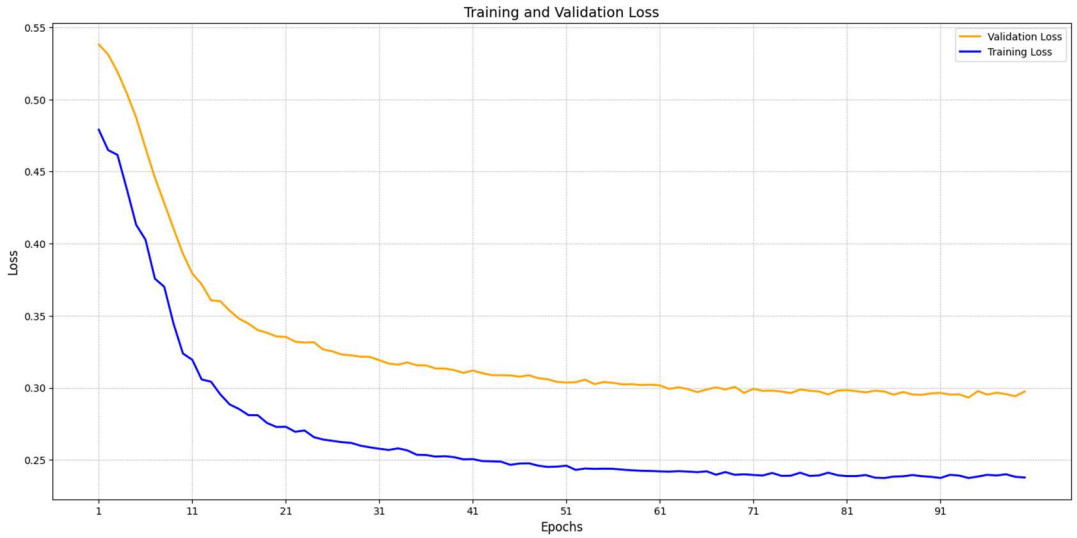


Batch size: 32, Learning rate: 0.00001, Drop rate: 0.2, Epochs:50

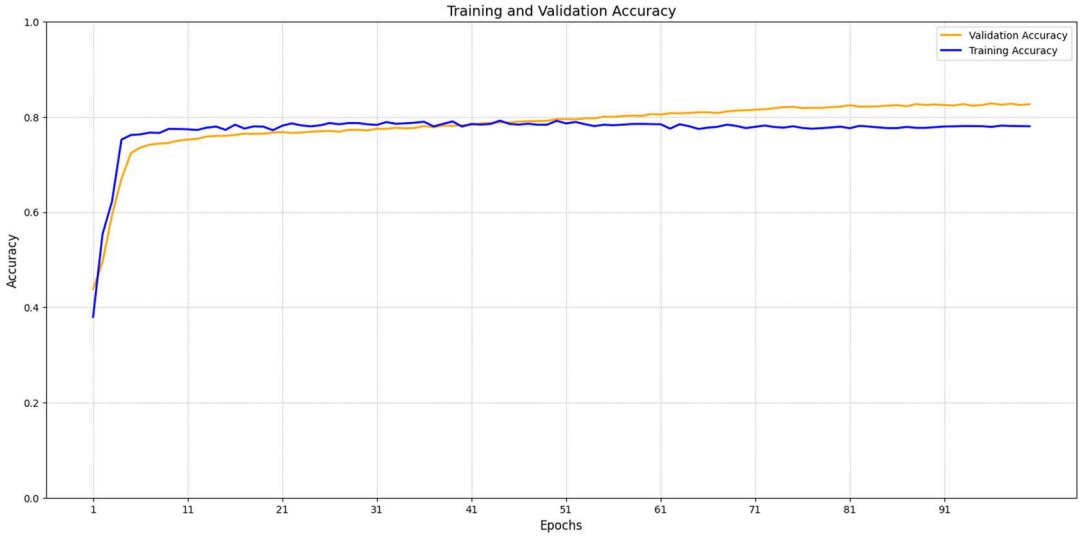


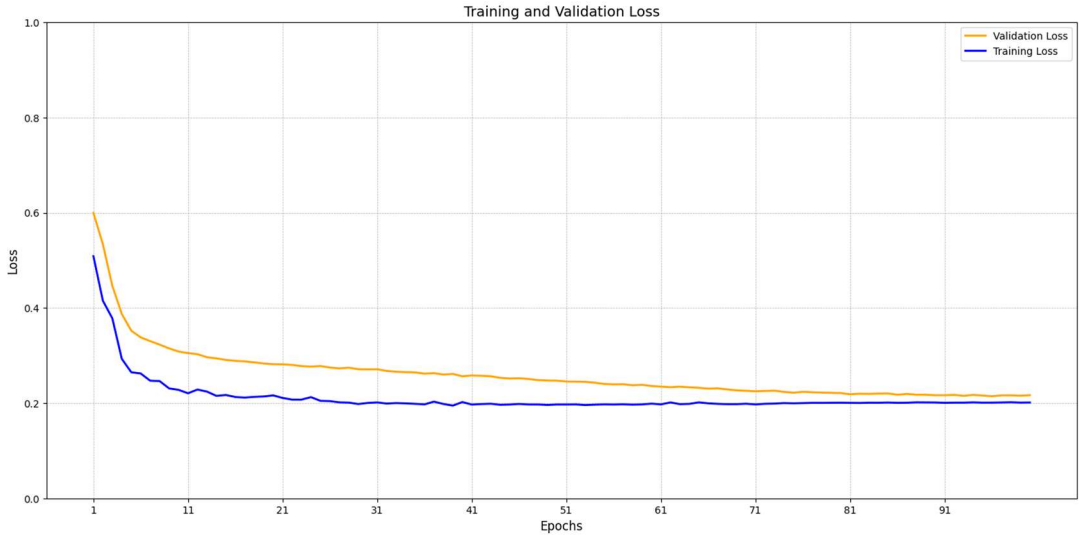
Batch size: 32, Learning rate: 0.0001, Drop rate: 0.2, Epochs:100



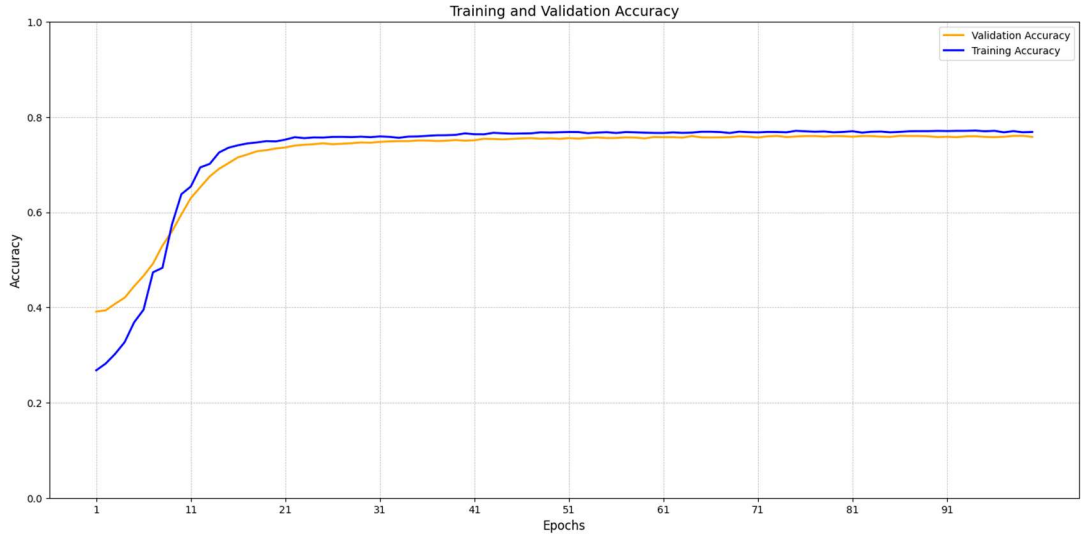


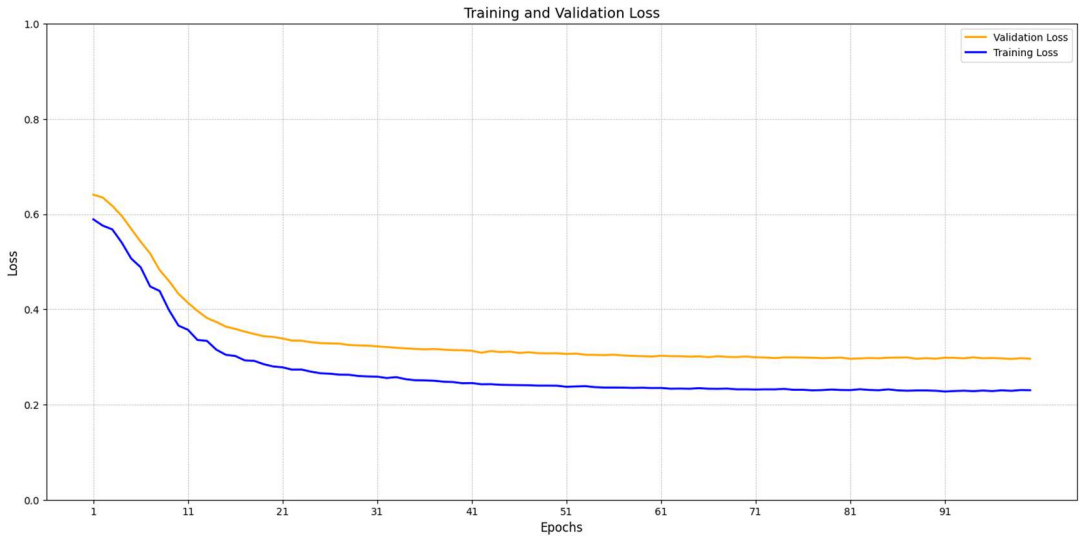
Batch size: 32, Learning rate: 0.00001, Drop rate: 0.2, Epochs:100



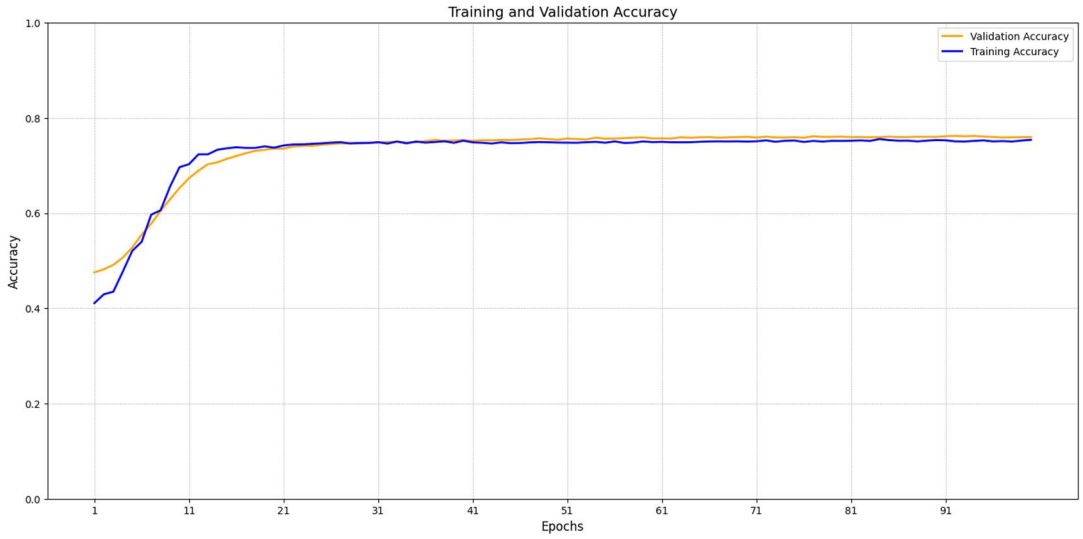


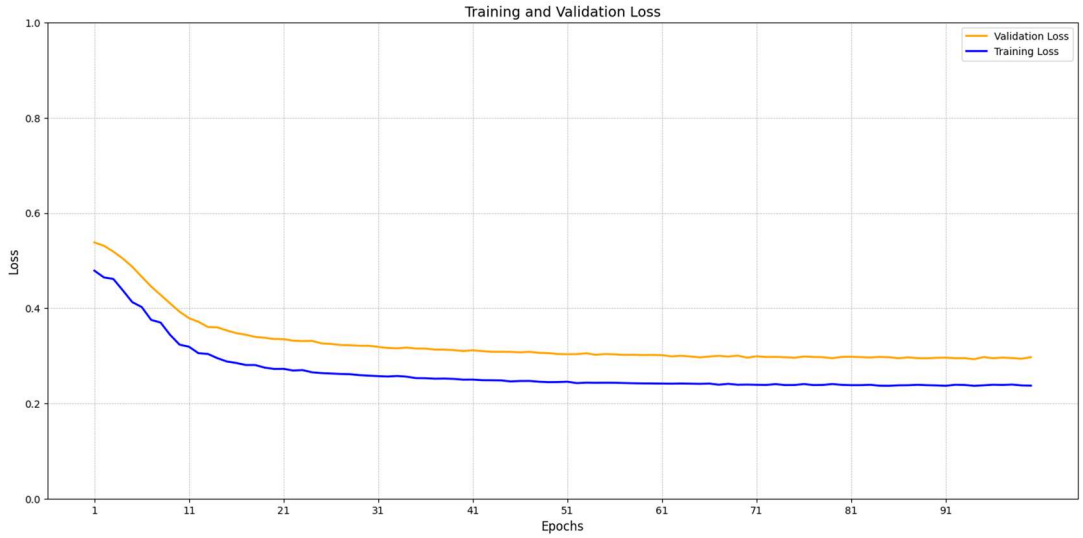
Batch size: 32, Learning rate: 0.00001, Drop rate: 0.5, Epochs:100





Batch size: 64, Learning rate: 0.00001, Drop rate: 0.5, Epochs:100





9.2 Result table and project matrics

Classification is a supervised machine learning method where the model tries to predict the correct label or class of a given input data. In classification, the model is fully trained using the training data, where images are labeled correctly, and then it is evaluated on test data before being used to perform prediction on new unseen data.

Based on accuracy, precision, recall and F1 score, we evaluated the performance of the classification system .

First, the following measures were evaluated:

- **true positive (tp)** = The model will predict lumbar spine degeneration, and the lumbar spine MRI will be classified as lumbar spine degeneration.

- **true negative(tn)** = The model will predict no lumbar spine degeneration, and the lumbar spine MRI will be classified as no lumbar spine degeneration.

- **false positive(fp)** = The model will predict lumbar spine degenerations, but the lumbar spine MRI will be classified as no lumbar spine degenerations.

- **false negative(fn)** = The model will predict no lumbar spine degeneration, but the lumbar spine MRI will be classified as lumbar spine degeneration.

Secondly, we will count each category instances and measure the following parameters:

- **Accuracy** – The percentage of "true" predictions out of all the predictions = $\frac{(tp + tn)}{(tp + tn + fp + fp)}$

- **Precision** – Out of all the positive predictions, how many of them were correct = $\frac{(tp)}{(tp + fp)}$

- **Recall /sensitivity** -Determines how good was the model at predicting real "yes" events, which means that we count the true positive instances out of all the instances where the MRI will be classified as a **lumbar spine degeneration** = $\frac{(tp)}{(tp + fn)}$

-**Recall/specificity** - Determines how good was the model at predicting real "no" events, which means that we count the true negative instances out of all the instances where the MRI will be classified as a **no lumbar spine degeneration** =$\frac{(tn)}{(tn + fp)}$

- **F1-score** - Used for imbalanced datasets (where the number of instances of each class/label is approximately the same).F1 score is the harmonic mean of precision and recall = $\frac{(2 * Recall * Precision)}{(Recall + Precision)}$

Parameters	Accuracy	Loss	Recall	Precision	F1
batch size:32 learning-rate:1e-3 dropout:0.2 epochs:50	0.77	0.64	0.74	0.79	0.76
batch size:32 learning-rate:1e-5 dropout:0.2 epochs:50	0.83	0.25	0.79	0.85	0.82
batch size:32 learning-rate:1e-7 dropout:0.2 epochs:50	0.41	0.62	0.38	0.43	0.40
batch size:32 learning-rate:1e-3 dropout:0.2 epochs:100	0.76	0.29	0.73	0.78	0.75
batch size:32 learning-rate:1e-5 dropout:0.2 epochs:100	0.81	0.26	0.78	0.83	0.80
batch size:32 learning-rate:1e-7 dropout:0.2 epochs:100	0.4	0.66	0.37	0.42	0.39
batch size:32 learning-rate:1e-3 dropout:0.5 epochs:50	0.75	0.43	0.72	0.77	0.74
batch size:32 learning-rate:1e-5 dropout:0.5 epochs:50	0.79	0.24	0.76	0.81	0.78
batch size:32 learning-rate:1e-7 dropout:0.5 epochs:50	0.39	0.65	0.36	0.41	0.38
batch size:32 learning-rate:1e-3 dropout:0.5 epochs:100	0.76	0.37	0.73	0.78	0.75
batch size:32 learning-rate:1e-5 dropout:0.5 epochs:100	0.78	0.25	0.75	0.8	0.77
batch size:32 learning-rate:1e-7 dropout:0.5 epochs:100	0.42	0.63	0.39	0.44	0.41
batch size:64 learning-rate:1e-3 dropout:0.2 epochs:50	0.74	0.4	0.71	0.76	0.73
batch size:64 learning-rate:1e-5 dropout:0.2 epochs:50	0.76	0.32	0.73	0.78	0.75
batch size:64 learning-rate:1e-7 dropout:0.2 epochs:50	0.33	0.69	0.3	0.35	0.32
batch size:64 learning-rate:1e-3 dropout:0.2 epochs:100	0.74	0.38	0.71	0.76	0.73
batch size:64 learning-rate:1e-5 dropout:0.2 epochs:100	0.78	0.3	0.75	0.8	0.77
batch size:64 learning-rate:1e-7 dropout:0.2 epochs:100	0.43	0.56	0.4	0.45	0.42
batch size:64 learning-rate:1e-3 dropout:0.5 epochs:50	0.76	0.29	0.73	0.78	0.75
batch size:64 learning-rate:1e-5 dropout:0.5 epochs:50	0.76	0.29	0.73	0.78	0.75
batch size:64 learning-rate:1e-7 dropout:0.5 epochs:50	0.42	0.61	0.39	0.44	0.41
batch size:64 learning-rate:1e-3 dropout:0.5 epochs:100	0.8	0.21	0.77	0.82	0.79
batch size:64 learning-rate:1e-5 dropout:0.5 epochs:100	0.77	0.35	0.74	0.79	0.76
batch size:64 learning-rate:1e-7 dropout:0.5 epochs:100	0.39	0.69	0.34	0.4	0.31

The project achieved an 83% validation accuracy in classifying lumbar spine degenerative conditions using an optimized DenseNet-121 model. Techniques like transfer learning, weighted loss, and data augmentation improved accuracy and addressed class imbalance. The system provides fast, reliable classifications, supporting efficient and accurate medical diagnoses.

in addition, to test our project, we had a testing MRI image inputted to our model, for example we took the image in fig 11 and classified it:

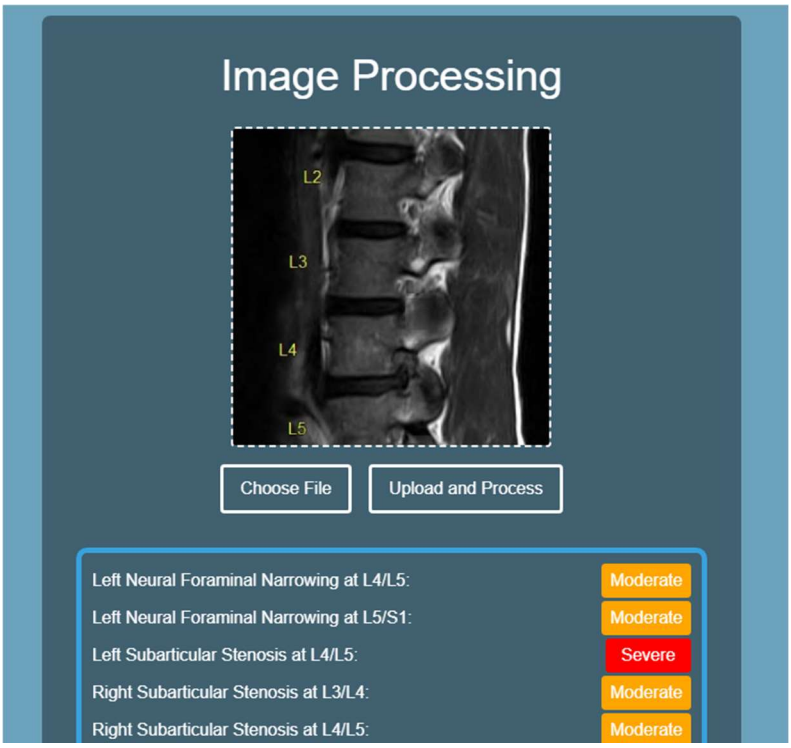


Fig.11 screenshot of the system at work, after classifying the MRI image

And here is the source image:

Figure 2. Figure 2 shows a T2 sagittal MRI of the lumbar spine with grade 0 or absence of stenosis at L2-3, grade 1 stenosis or mild foraminal stenosis at L3-4 with some fat obliteration in both the vertical and transverse directions, grade 2 stenosis or moderate foraminal stenosis at L5-S1 with perineural fat obliteration in all four directions without morphologic change in the nerve root, and grade 3 stenosis or severe foraminal stenosis at L4-5 showing complete perineural fat obliteration with nerve root collapse or deformation.

Fig.12 screenshot of the original image and description [13]

We can see that the system correctly classified the stenosis with the correct severity, and predicting foraminal narrowing in addition to the stenosis, which was not mentioned in the image.

10. User manual

1. Code setup:

In the GitHub repository, you will find a requirements.txt file that lists all the necessary dependencies for this project. Before getting started, make sure you are using Python 3.10 or an older version, and if you plan to use GPU acceleration for faster training, make sure that CUDA drivers are installed. (The script automatically checks for GPU availability by confirming that *torch.cuda.is_available()* is True.) Once your environment is ready, simply open a terminal and run *pip install -r requirements.txt,* This command will install all the required libraries in one go.

Training.py:

in this code we are employing the training and validation of the project, we are building a pipeline to input data and building a model using a DenseNet-121 architecture.
To properly run the code, you will need to adjust the paths for the resources used in the script.
rd- the CSV file (train.csv) path on your local environment.
DATA_PATH- the path to the images folder as set in the image preprocessing stage as described earlier.
OUTPUT_DIR- a path for the results of the code (trained model and result csv files)
In addition, we use several hardware adjustments for the code:
device- set the device to use GPU via Cuda if available, CPU otherwise.
N_WORKERS- number of processors available in the system to run the code, it is recommended to use *os.cpu_count()-2* to use the resources efficiently.
Next is the hyperparameters settings:
Set *LR* for the learning rate you want to use, *BATCH_SIZE* for the batch size, *EPOCHS* for epochs and *DROPOUT* for the dropout rate.
Lastly, we want to save the results in our results folder, we will define 2 variables for it:
fname- for the trained model name
csv_path- for the csv results, saving the training and validation accuracy and loss for each epoch in each fold separately so we can make the results graphs later.

2. Code overview:

Training.py:

Setup & Data

- CSV Input: A train.csv file maps each study_id to 25 spine labels. Missing labels are replaced with -100 so they're ignored by the loss.
- MRI Slices: Three sequences (Sagittal T1, Sagittal T2/STIR, Axial T2) form a 30-channel volume in the custom Mydataset class.

Transforms & Model

- Albumentations: Applies brightness/contrast, blur, distortion, dropout, and normalization on the training set. The validation set is only resized and normalized.
- DenseNet: Built with timm with 30 input channels and 75 output classes (25 spine levels × 3 classes each).

Cross-Validation

- KFold: Splits the data into several folds (default=3).
- Each fold trains on a subset and validates on the remaining fold's data

Training & Validation

- Mixed Precision: Uses GradScaler for AMP, along with gradient accumulation if GPU memory is limited.
- Loss: Averages sub-losses across each spine label (3-class classification).
- Cosine Scheduler: Gradually adjusts the learning rate.
- Model Checkpoints: Whenever validation accuracy improves, saves a .pt file

Results & Logging

- Metrics: Stores per-epoch training and validation loss/accuracy in CSV files.
- Final Summary: Prints a fold-by-fold overview of performance for quick comparison.

Mydataset:

This code creates a costume dataset as we said earlier in the project.

Initialization:

- Takes a DataFrame (df), a root folder (data_path), and an optional transform (transform).
- Remembers phase (e.g., 'train' or 'valid')—though it's not heavily used here.

Data Loading:

- For each row, fetches the study_id and associated labels.
- Allocates a zero array x of shape [512, 512, 30] (the 30 channels).
- Reads 10 slices from Sagittal T1, 10 from Sagittal T2_STIR, and 10 from Axial T2 (spaced evenly by step in the axial folder).

Transforms & Output:

- Applies Albumentations transform if provided.
- Transposes x to [channels, height, width]
- Returns (x, label), where label is an array of sub-labels.

Classification.py:

This is the file that has the classification process logics, it loads the best model we have from the training process, uses it as a classification model and output the results as a dictionary.

Model Definition:

- A DenseNet is created via timm, configured for 30 input channels and 75 output classes.

Loading:

- *load_model(weights_path)* instantiates a DenseNet, loads weights from a .pt file, moves it to DEVICE, and sets eval mode.

Preprocessing:

- *preprocess_image(image)* converts an input PIL.Image to a NumPy array, repeats channels to create a 30-channel volume, then applies Albumentations (Resize, Normalize) and converts it to a PyTorch tensor.

Inference:

- *classify_image(image)* loads the model, preprocesses the given image, performs a forward pass to get logits, and uses softmax for probabilities.
- Maps each of the 25 sub-labels (5 conditions × 5 levels) to a predicted severity (Normal/Mild, Moderate, Severe), returning only those not labeled Normal/Mild.

The React front-end and Flask backend communicate through HTTP. When a user uploads an image in the React app, the axios library sends a POST request with the image to the Flask server. Flask then processes the image using the classification code and sends back the results in a JSON response. React displays these results to the user, so they can see the classification immediately. This setup keeps the interface, and the processing logic separate but connected via simple network requests.

Running the application:

1. **Install Packages**

- In the React project folder, run *npm install* to install dependencies (like axios).

2. **Start the App**

- In your GUI folder, open a terminal and run *npm start*; open http://localhost:3000 in your browser.

3. **Upload an Image**

- Click “Choose File” and select your MRI image.
- A preview appears on-screen.

4. **Get Classification**

- Click “Upload and Process” to send the image to the Flask server.
- The server returns a classification, which is displayed below the preview.
- If it is done successfully, the returned object will be printed in the web console (press F12 to open).

5. **Check Flask**

- Make sure Flask is running at http://127.0.0.1:5000/process-image.
- If there’s an error, check the server logs or browser console.
- If the image upload succeeds, the information will be shown in the terminal of the flask application.

11. Maintenance guide

Short Maintenance Guide

1. **Data and Paths**

- Update the CSV path (*rd*), image directory (*DATA_PATH*), and output folder (*OUTPUT_DIR*) if you move or rename files.
- In MyDataset, confirm the folder structure matches your MRI slices (Sagittal T1, Sagittal T2/STIR, Axial T2).

2. **Hyperparameters**

- Adjust learning rate (*LR*), batch size (*BATCH_SIZE*), epochs (*EPOCHS*), and dropout (*DROPOUT*) near the top of the training script to tune performance or handle larger/smaller datasets.

3. **Model Updates**

- If you switch to another architecture (e.g., ResNet instead of DenseNet), replace *MODEL_NAME* and confirm *IN_CHANS* and *N_CLASSES* match your new design.

4. **Label/Condition Changes**

- If you add or remove conditions (e.g., new spine pathologies), update the 25 sub-labels logic (or how you split 75 outputs) in both training and inference scripts.

- Make corresponding changes in the *CONDITIONS* and *LEVELS* lists, as well as in the post-processing steps.

5. **React & Flask**

- In the React app, check the endpoint URL (http://127.0.0.1:5000/process-image). If you change ports or deploy the Flask server elsewhere, update this URL.

- If you need new classification outputs, adjust the display logic in App.js accordingly.

6. **Troubleshooting**

- **Out of memory**: Lower the batch size or reduce image size.
- **Slow performance**: Increase *N_WORKERS* in data loaders or reduce heavy augmentations.
- **CORS errors**: Enable *CORS* in Flask if you change the default ports or domains.

12. Project achievements

Our project successfully met its objectives by creating an accurate and efficient system for classifying lumbar spine conditions from MRI scans. We built a working network using DenseNet, a special type of neural network that connects layers in a unique way to improve learning. This network was trained to classify MRI scans into five different lumbar spine conditions: Left Neural Foraminal Narrowing, Right Neural Foraminal Narrowing, Left Subarticular Stenosis, Right Subarticular Stenosis, and Spinal Canal Stenosis. It also graded the severity of these conditions (normal/mild, moderate, severe) across the disc levels L1/L2 to L5/S1 (which we were not sure if we can do in phase A of the project).

One of our big goals was to make sure the system was accurate, and we achieved better than 80% accuracy, proving the model is reliable and can consistently make correct predictions. We also wanted to speed up the process for doctors, and by using DenseNet and transfer learning, which allows the system to start learning from pre-trained features instead of starting from scratch, the model trained faster and performed better even with less data. The high-quality dataset we used from Kaggle, with clear labels, helped the system learn how to recognize and classify conditions effectively.

In addition, we built a simple to use GUI for doctors to use and easily classify the MRI images.

Overall, the project met its metrics by building a dependable, efficient system that can make fast, accurate assessments of lumbar spine degenerations, helping doctors spend less time analyzing scans and providing patients with quicker results.

13. Future work

We plan to expand our system to handle more types of spine conditions and possibly include scans beyond the lumbar region. By adding more labeled data and refining how our model detects even smaller changes, the system could become more sensitive and accurate for various spinal issues. We also hope to explore additional imaging techniques and advanced segmentation methods to give doctors even more detailed information about each patient's spine.

On the application side, we want to build a larger web platform where users can create accounts, upload multiple scans, and keep a history of past results. This would help doctors track how conditions change over time and make it easier for patients to receive fast, accurate results. By linking this tool to electronic health record systems, doctors can save time by automatically storing each classification result and sharing it across different medical departments.

14. References

1. World Health Organization. (2023, June 19). *Low back pain*. https://www.who.int/news-room/fact-sheets/detail/low-back-pain

2. DO Neurological Surgery. *Spine & disc anatomy*. https://www.doneurosurgery.com/spine--disc-anatomy.html

3. Kaggle.com, *RSNA 2024 lumbar spine degenerative classification* [Data set]. https://www.kaggle.com/competitions/rsna-2024-lumbar-spine-degenerative-classification/data

4. Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2018). Densely connected convolutional networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, 4700-4708. https://doi.org/10.1109/CVPR.2017.243

5. Kim, H. E., Cosa-Linan, A., Santhanam, N., Jannesari, M., Maros, M. E., & Ganslandt, T. (2022). Transfer learning for medical image classification: A literature review. BMC Medical Imaging, 22(69). https://doi.org/10.1186/s12880-022-00793-7

6. Python Software Foundation.*Python documentation (version 3.11)* https://docs.python.org/3.11/

7. Liawrungrueang, W., Kim, P., Kotheeranurak, V., Jitpakdee, K., & Sarasombath, P. (2023). Automatic detection, classification, and grading of lumbar intervertebral disc degeneration using an artificial neural network model. https://www.mdpi.com/2075-4418/13/4/663

8. National Institute of Biomedical Imaging and Bioengineering. (n.d.). Magnetic Resonance Imaging (MRI). National Institutes of Health. https://www.nibib.nih.gov/science-education/science-topics/magnetic-resonance-imaging-mri

9. Kaggle. (2024). *RSNA 2024 lumbar spine degenerative classification*. Kaggle. Retrieved from https://www.kaggle.com/competitions/rsna-2024-lumbar-spine-degenerative-classification

10. PyTorch. *PyTorch documentation (version 1.9.1)*. https://pytorch.org/docs/1.9.1/

11. Meta Platforms, Inc. *React documentation*. https://react.dev/

12. Carrino, J. A., Morrison, W. B., Zou, K. H., Steffen, R. T., Manoni, G. R., & Schweitzer, M. E. (2001). Lumbar disc herniation: Accuracy of MR imaging findings. American Journal of Roentgenology, 177(5), 1205-1212. https://doi.org/10.2214/AJR.13.11493

13. Cavazos, D. R., Higginbotham, D. O., Nham, F., Court, T., McCarty, S., Sethi, A., & Vaidya, R. (2023). *Neuroforaminal stenosis in the lumbosacral spine: A scoping review of pathophysiology, clinical manifestations, diagnostic imaging, and treatment*. *Spartan Medical Research Journal*, 8(1). https://doi.org/10.51894/001c.87848

14. GitHub repo: https://github.com/danielZada97/Capstone-Project-24-2-R-1