Assessing Fracture Healing with Artificial Intelligence:

Using Transfer Learning to Predict the *Radiographic Union Score for Tibial Fractures*, in the Radiography of High-Energy Trauma

Shen Zhou Hong Goldsmiths, UoL

April 4th, 2023

Contents

1 Implementation and Analysis **2**

1.1 One-Hot Encoding for Labels . . . 3
1.2 K-Fold Evaluation . . . 4
1.3 Establishing Baseline Performance Targets . . . 6
1.3.1 Shallow Convolutional Neural Network . . . 6
1.3.2 End-to-End Training with InceptionV3 . . . 9
1.3.3 Baseline Metrics . . . 10
1.4 InceptionV3 with Transfer Learning . . . 11
1.4.1 Base Model Trained on RadImageNet Dataset . . . 12
1.4.2 Base Model Trained on InceptionV3 Dataset . . . 12
1.4.3 Comparison between RadImageNet and ImageNet . . . 13
1.5 Hyperparameter Search . . . 14
1.5.1 Hyperparameter Search Regime I . . . 14
1.5.2 Hyperparameter Search Regime II . . . 20
1.5.3 Final Hyperparameters . . . 22

A Additional Materials **24**

A.1 Project Code and Github Repository . . . 24
A.1.1 Initial Evaluation Models . . . 24
A.1.2 Hyperparameter Search Code . . . 24
A.1.3 Analysis Notebooks . . . 24

Chapter 1

Implementation and Analysis

In this chapter, we will present the implementation of the study methodology. Recall that the methodology has three components. We will begin with the establishment of an initial baseline, by creating and training a classical 'shallow' convolutional neural network based upon LeCun et al.'s 1998 LeNet model. [1] This classical CNN baseline will serve as the minimal performance standard that our model will aim to surpass. Next, utilising the InceptionV3 architecture which will serve as our transfer-learning base model, we will train an end-to-end (i.e. without transfer learning) model on our radiography dataset. This will serve as an additional baseline that will allow us to validate the transfer-learning *technique* against regular end-to-end training.

Following the establishment of these two baselines, we will proceed to begin an initial evaluation of two different transfer-learning base models. We will compare the performance of InceptionV3 trained with ImageNet weights [2], against InceptionV3 trained with RadImageNet [3] weights. This initial evaluation will help us explore whether a base model trained on the smaller, but domain-specific RadImageNet dataset will have any advantages over the larger, but general ImageNet dataset. We will select the better performing base model out of the two options, and proceed to optimize the model's hyperparameters.

Our model's hyperparameter search procedure consists of two steps, which we term hyperparameter search Regime I and hyperparameter search Regime II. As per our methodology, in Regime I we find the optimal batch size and dropout rate for our model. This is done using a stochastic search process where the hyperparameter space of the model is randomly sampled for t trials, where each trial consists of a k-fold cross-validation of the model with the selected hyperparameters. Once the optimal combination of batch size and dropout rate are found, we will set these hyperparameters

as constant and proceed to the second hyperparameter search regime. In Regime II we find the optimal learning rate and epsilon value ϵ for the Adam optimizer, by conducting a grid search over a selection of possible values.

1.1 One-Hot Encoding for Labels

Recall that our model must predict a RUST score from a pair of input radiographs. RUST scores are measures of orthopaedic union (i.e. healing), and ordinarily consist of two subscores for each view (the anterior-posterior and medial-lateral views), quantifying the development of bone calluses and bridging over the fracture line. The score components are as follows:

Radiographic Feature	Score
Fracture Line, No Callus	1
Fracture Line, Visible Callus	2
No Fracture Line, Bridging Callus	3
No Fracture Line, Remodeled.	4

Table 1.1: Radiographic Union Score for Tibial Fractures (RUST) Rubric

These score components[1] are then used to assess the features of a fracture from the anterior, posterior, medial, and lateral cortices:

	Subscore
Anterior Cortex	
Posterior Cortex	
Medial Cortex	
Lateral Cortex	

Table 1.2: RUST Scoring Instrument

Finally the resulting subscore components are summed in order to yield a RUST score for the fracture as a whole.

For this study, our model is designed to predict every component subscore. Hence, the label will consist of a 18-value `tf.Tensor` with the shape (18,), consisting of 16 one-hot encoded values for the RUST subscores (four for each cortice), as well as two additional one-hot values to represent the view (anterioposterior or medial-lateral).

[1] Note that the original Whelan et al. paper [4] does not include a value for remodelled fractures, however as the METRC dataset includes this category, we will be using the modified RUST variant specific to Johns Hopkins for this study.

1.2 K-Fold Evaluation

Before we begin, we must first implement our k-fold cross-validation routine. Since model performance is sensitive to the network's random weight initialisation[2] [5], our methodology requires k-fold cross-validation to be conducted on every experiment (i.e. model run). My implementation of the k-fold cross-validation process consists of two parts: a function which will divide the dataset into k folds, as well as a function that runs the k-fold cross-validation on the given model. The the `k_fold_dataset()` function is given as follows:[3]

```python
def k_fold_dataset(ds: tf.data.Dataset, k: int = 10) -> list[tuple[tf.data.Dataset,
↪  tf.data.Dataset]]:
    # First shard the given dataset into k individual folds.
    list_of_folds: list[tf.data.Dataset] = []
    for i in range(k):
        fold: tf.data.Dataset = ds.shard(num_shards=k, index=i)
        list_of_folds.append(fold)

    # Next, generate a list of train and validation dataset tuples
    list_of_ds_pairs: list[tuple[tf.data.Dataset, tf.data.Dataset]] = []
    for i, holdout_fold in enumerate(list_of_folds):
        ds_valid: tf.data.Dataset = holdout_fold

        # Select every fold except holdout_fold as the training folds
        training_folds: list[tf.data.Dataset] = list_of_folds[:i] +
        ↪  list_of_folds[i+1:]

        # ds_train size is (k-1)/k of the original dataset
        ds_train: tf.data.Dataset = training_folds[0]
        for fold in training_folds[1:]:
            ds_train = ds_train.concatenate(fold)

        ds_pair: tuple[tf.data.Dataset, tf.data.Dataset] = (ds_train, ds_valid)
        list_of_ds_pairs.append(ds_pair)

    return list_of_ds_pairs
```

Listing 1: Sharding dataset for K-Fold Cross Validation (Github)

One thing of note, is that our `k_fold_dataset()` function conducts all dataset-related operations using the Tensorflow's high-performance `tf.data.Dataset` API. This allows support for pre-fetch, caching, and other low-level optimisations. This function serves as a dependency which is called by `cross_validate()`, which runs the actual K-fold cross validation experiments on the given model:

[2] This is particularly true on small datasets with unbalanced classes like ours.

[3] The code listings provided in this document *are for illustration only*. The actual implementation is generally longer, and contains docstrings, debugging instrumentation, file I/O logic, as well as additional function arguments. Every listing will have a link to it's corresponding implementation in the git repository.

```python
def cross_validate(ModelClass: tf.keras.Model, ds: tf.data.Dataset, epochs: int = 50,
↪  batch_size: int = 128, k: int = 10) -> list[tf.keras.callbacks.History]:
    history_list: list[tf.keras.callbacks.History] = []
    train_valid_pairs: list[tf.data.Dataset] = k_fold_dataset(ds, k)

    for i, (ds_train, ds_valid) in enumerate(train_valid_pairs):
        # Reset tensorflow gradient tape
        tf.keras.backend.clear_session()
        model = ModelClass()
        model.compile(
            optimizer=tf.keras.optimizers.Adam(),
            loss=tf.keras.losses.BinaryCrossentropy(),
            metrics=metrics
        )
        history = model.fit(
            ds_train,
            validation_data=ds_valid,
            epochs=epochs,
            batch_size=batch_size,
        )
        history_list.append(history.history)

    return history_list
```

Listing 2: K-Fold Cross Validation (Github)

The output of every k-fold cross-validation experiment will be a 'history list' containing k `tf.keras.callbacks.History` objects. This `History` object will contain training and validation metrics which will be used to calculate the average metric over k folds:

```python
def calculate_mean_metrics(kfold_metrics: list[dict[str, float]]) -> dict[str,
↪  list[float]]:
    # Initialise aggregate metrics with appropriate keys
    aggregate_metrics: dict[str, list[float]] = {}
    for fold in kfold_metrics:
        for metric in fold.keys():
            if metric not in aggregate_metrics:
                aggregate_metrics[metric] = []

    # Calculate the average metric per epoch for every fold
    number_of_folds: int = len(kfold_metrics)
    for metric in aggregate_metrics.keys():
        number_of_epochs: int = len(kfold_metrics[0][metric])
        for epoch in range(number_of_epochs):
            # A list of every value for that given metric in this epoch across folds
            values_per_epoch: list[float] = [x[metric][epoch] for x in kfold_metrics]
            mean_per_epoch  : float = sum(values_per_epoch) / number_of_folds
            aggregate_metrics[metric].append(mean_per_epoch)

    return aggregate_metrics
```

Listing 3: Calculating Mean Metrics from K-Fold Data (Github)

The above code now completes the prerequisites necessary for data gathering.

1.3 Establishing Baseline Performance Targets

In this section, we will establish the baseline performance targets for our transfer-learning model by training and developing two models which will represent alternative approaches to the problem of multilabel classification on a small dataset. The baseline models will be: a 'shallow' CNN following LeCun et al.'s classical 1998 LeNet architecture [1], and an InceptionV3 model that is directly end-to-end trained on our radiography dataset. We explicitly choose the above two models as our baseline for comparison, because they each help validate a different aspect of this project: whether a deep neural network is appropriate for the task in the first place, and whether the *technique* of transfer learning is appropriate for our dataset. The second question of whether or not our technique is necessary is why we train a version of our model's architecture directly on the radiography data, in order to obtain a performance measure of using the same model architecture *without* transfer learning. At minimum, our transfer-learning model must achieve a better performance (as measured by it's AUROC score) over the two baseline models.

The performance of the baseline models will be measured as the highest observed *average* AUROC, found using k-fold cross-validation with $k = 10$. The value of $k = 10$ is chosen because the resulting per-fold training and validation splits are no larger than a conventional train, test, and validation split of 70%, 15%, 15%, where:

- Training and Validation Set (`ds_train + ds_valid`): 2490 (85%):
  - K-Fold Cross-Validation, $K = 10$:
    - Training Set: 2241 (~76%)
    - Validation Set: 249 (~8.5% per fold)
- Hold-out Test Set (`ds_test`): 441 (15%)

Larger k values yield a more thorough measurement of a model's performance at the cost of additional computational costs, while lower k values risk lowering the training-validation split ratio until the training set is too small for adequate training. For this initial evaluation, as we wish to yield a benchmark for baseline performance, we will be using a k value of 10. For the hyperparameter search regime, we will be using $k = 6$ in order to lower computational costs.

1.3.1 Shallow Convolutional Neural Network

For the first benchmark, we begin by implementing the shallow convolutional neural network described by LeCun et al in [1] in Tensorflow. Our implementation follows the original paper, with a slight modification in the final classifier, in order to output the 18-vector one-hot encoded label predictions. Note the presence of only two convolutional layers — this is typical for early CNNs of that period.

```python
class LeNet1998(tf.keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.input_layer: tf.Tensor = layers.InputLayer(input_shape=(299, 299, 3))
        self.data_augmentation: tf.keras.Sequential = tf.keras.Sequential([
            layers.RandomFlip(seed=RNG_SEED),
        ])

        self.lenet1999: tf.keras.Model = tf.keras.Sequential([
            layers.Conv2D(6, kernel_size=5, strides=1,  activation='tanh',
            ↪  padding='same'),
            layers.AveragePooling2D(),
            layers.Conv2D(16, kernel_size=5, strides=1, activation='tanh',
            ↪  padding='valid'),
            layers.AveragePooling2D(),
        ])

        self.classifier: tf.keras.Sequential = tf.keras.Sequential([
            layers.Flatten(),
            layers.Dense(1024, activation='relu'),
            layers.Dense(18, activation='sigmoid')
        ])

        self.model: tf.keras.Sequential = tf.keras.Sequential([
                self.input_layer,
                self.data_augmentation,
                self.lenet1999,
                self.classifier
        ])

    def call(self, inputs):
        return self.model(inputs)
```

Listing 4: The LeNet 1998 Shallow CNN Model (Github)

We implement our version of the LeNet architecture by subclassing `tf.keras.Model` class, which is then passed on to our `cross_validate()` function to be evaluated. This entire experiment is conducted within a Jupyter notebook which is made available as a self-contained, reproducible unit within the project repository (Github). Running the experiment yields our first AUROC to performance graph:

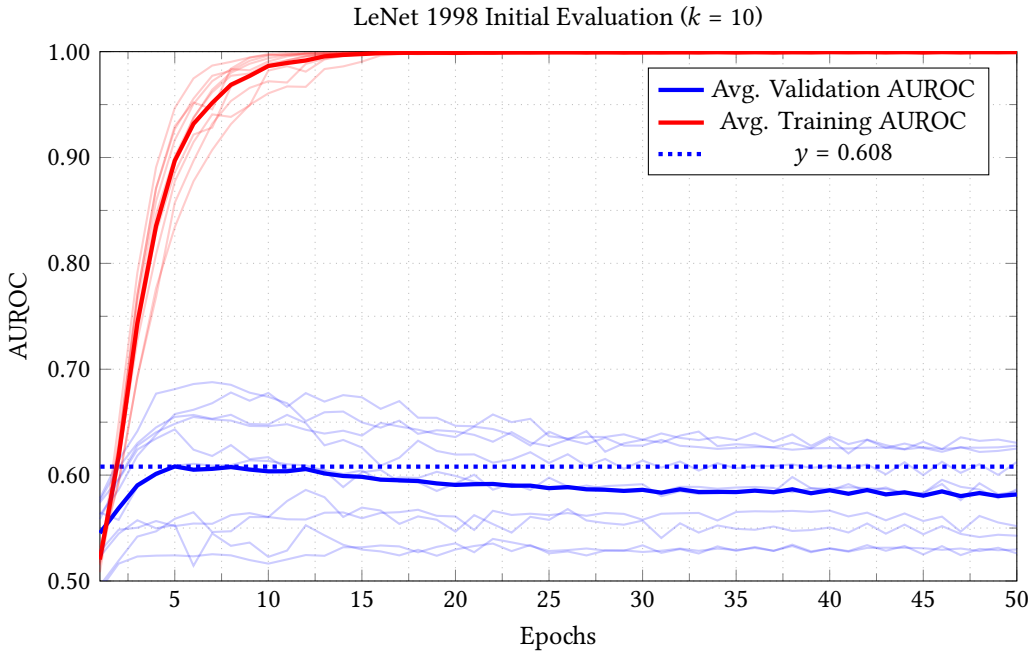

Figure 1.1: Baseline shallow CNN based on the LeNet 1998 architecture

The bold lines in the chart represent the *average* training (red) and validation (blue) AUROC, as measured after performing k-fold cross-validation on 10 folds (k = 10). The transparent lines indicate the observed training and validation AUROC per each individual fold: this per-fold performance has been charted in order for us to better observe the consistency of model performance per epoch. Variations in performance per fold is due to a combination of different random starting conditions (due to random weight initialisation at the start of a model's training), as well as variances in floating-point calculations.

What information does our data for the LeNet model tell us? First, we can observe severe overfitting: by epoch 10, performance on the training set asymptomatically approaches 1.0 (as quantified by AUROC). However, the validation performance remains minimal: generally averaging around 0.60, with certain instances of the model performing little better than chance (0.50). This information indicates that a classical 'shallow' CNN lacks the representational power to extract the features necessary to perform classification on our dataset. Indeed, it appears that the LeNet model fails to converge at all. This is to be expected: and our experiment yields a minimal baseline AUROC value of 0.608 that our subsequent models must beat. Likewise, by demonstrating that classical 'shallow' CNNs are unable to solve our problem, we make the

case for using a 'deep' neural network: in the form of the InceptionV3 architecture, which we will explore in the following section.

1.3.2 End-to-End Training with InceptionV3

Having validated the necessity of using a deep convolutional neural network to solve this *multiclass*, *multilabel* classification task, our next question would be: "is it necessary to use the technique of *transfer-learning* on our dataset, or would a regular end-to-end training process suffice?" Although the small size of our dataset indicates that transfer learning is appropriate, it is important for us to validate our assumptions through empirical data. Hence, we arrive at the establishment of the second baseline model: end-to-end training InceptionV3 on our dataset. Let us start by defining our base model:

```python
class TransferLearningModel(tf.keras.Model):
    def __init__(self, dropout_rate: float, **kwargs):
        super().__init__(**kwargs)

        self.input_layer: tf.Tensor = layers.InputLayer(input_shape=(299, 299, 3))
        self.data_augmentation: tf.keras.Sequential = tf.keras.Sequential([
            layers.RandomFlip(seed=RNG_SEED),
        ])

        self.inceptionv3: tf.keras.Model = tf.keras.applications.InceptionV3(
            include_top=False,
            weights='imagenet'
        )
        self.inceptionv3.trainable = False

        self.classifier: tf.keras.Sequential = tf.keras.Sequential([
            layers.GlobalMaxPooling2D(),
            layers.Dense(1024, activation='relu'),
            layers.Dropout(dropout_rate),
            layers.Dense( 512, activation='relu'),
            layers.Dropout(dropout_rate),
            layers.Dense( 256, activation='relu'),
            layers.Dropout(dropout_rate),
            layers.Dense(  18, activation='sigmoid')
        ])

        self.model: tf.keras.Sequential = tf.keras.Sequential([
            self.input_layer,
            self.data_augmentation,
            self.inceptionv3,
            self.classifier
        ])

    def call(self, inputs):
        return self.model(inputs)
```

Listing 5: Model Class for InceptionV3 (Github)

We define a `class TransferLearningModel` which will be instantiated by every k-fold validation trial. Note that for this particular experiment, as we are establishing an

end-to-end trained baseline, we will be setting `self.inceptionv3(weights=None)` and the attribute `self.inceptionv3.trainable = True`. Naturally in the actual implementation (Github) this is done through an argument in the class constructor, however the listing is simplified for the purpose of size and readability. So what happens now when we run the kfold experiment (Github)?

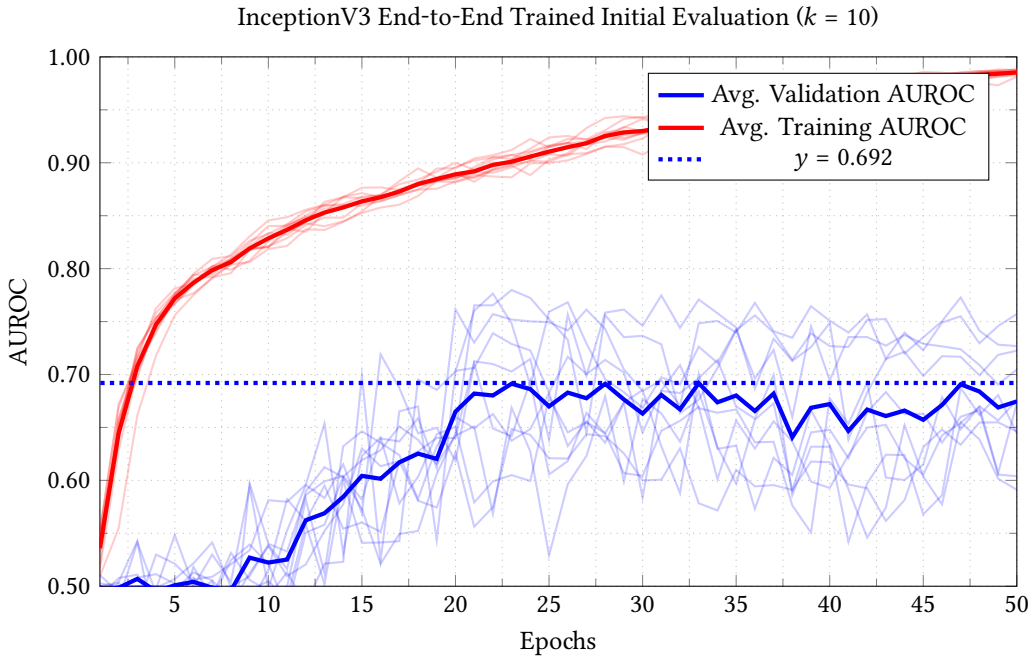

Figure 1.2: InceptionV3 Model Trained on Study Data.

We can observe that the end-to-end variant performs marginally: achieving an average AUROC of 0.692. However, upon a closer examination it is clear that the validation AUROC of each individual k-fold trial is highly erratic. The large spikes in validation AUROC indicates a failure to converge, as the dataset is too small for the number of tunable weights in the model. The InceptionV3 model has 189 layers, with a combined total of 23.9 million trainable weights: representing a parameter space several orders of magnitude larger than our dataset. The highly erratic validation AUROC is only a symptom of the model's inability to converge, and demonstrates clearly that regular end-to-end training is insufficient and inappropriate for our dataset.

1.3.3 Baseline Metrics

Having completed assessing our two baseline models, we are left with the following metrics that will help us in our own evaluation:

Baseline	Validation AUROC
Random (no better than chance)	0.50
Classical Shallow CNN (LeNet)	0.61
End-to-End Model (InceptionV3)	0.69

Table 1.3: Baseline Benchmarks

The first level of performance that our subsequent models are expected to achieve is a validation AUROC > 0.50. As a measurement of classifier performance, an AUROC of 0.50 indicates performance no better than chance (i.e. the same as choosing by random). If we are unable to meet the minimum baseline of > 0.50, then our entire approach may be unrealistic and infeasible. The second baseline that we must achieve is a performance of > 0.61. For that is the best performance measured from a classical 'shallow' CNN. As deep neural networks, with their dozens (if not hundreds) of layers incur a computational cost that is an order of magnitude above classical 'shallow' CNNs, if our model is not able to exceed the performance of a regular CNN, it will be better to develop a regular CNN instead. Finally, the last baseline that we established allows us to validate the suitability of the transfer-learning technique. If our model is unable to meet an AUROC of > 0.69, then we will be better served to train our model architecture directly on our dataset.

With this information in hand, it is time for us to embark on the second part of our study: developing a transfer-learning model to infer the RUST score of radiographs of long-bone fractures. In the transfer learning technique, a base model is trained on a larger dataset, before having it's weights frozen, and then used as a component of a classifier trained on the task-specific dataset. A key decision in this process is the choice of the larger dataset that the base model will be trained on. Thus, we are lead to the second part of our study: evaluating the performance of InceptionV3 trained on ImageNet, and RadImageNet.

1.4 InceptionV3 with Transfer Learning

Let us begin by evaluating the performance of the RadImageNet weights. Following the same procedure as we did earlier, we instantiate a InceptionV3 model with our classifier, and set the attribute `self.inceptionv3(weights='radimagenet.h5')` and `self.inceptionv3.trainable = False`. This class instance now has InceptionV3 with pre-trained ImageNet weights, which we have frozen. Now when we run the training process, only our classifier will be trained. We conduct the experiment in the following Jupyter notebook (Github).

1.4.1 Base Model Trained on RadImageNet Dataset

The model trained on RadImageNet weights achieves an average validation AUROC of 0.706. Note how unlike the version of InceptionV3 that was end-to-end trained, the per-fold validation AUROC is fairly consistent: we do not see any large spikes in validation performance. Likewise, although the model begins to exhibit overfitting after epoch 15, the degree of overfitting is relatively well controlled.

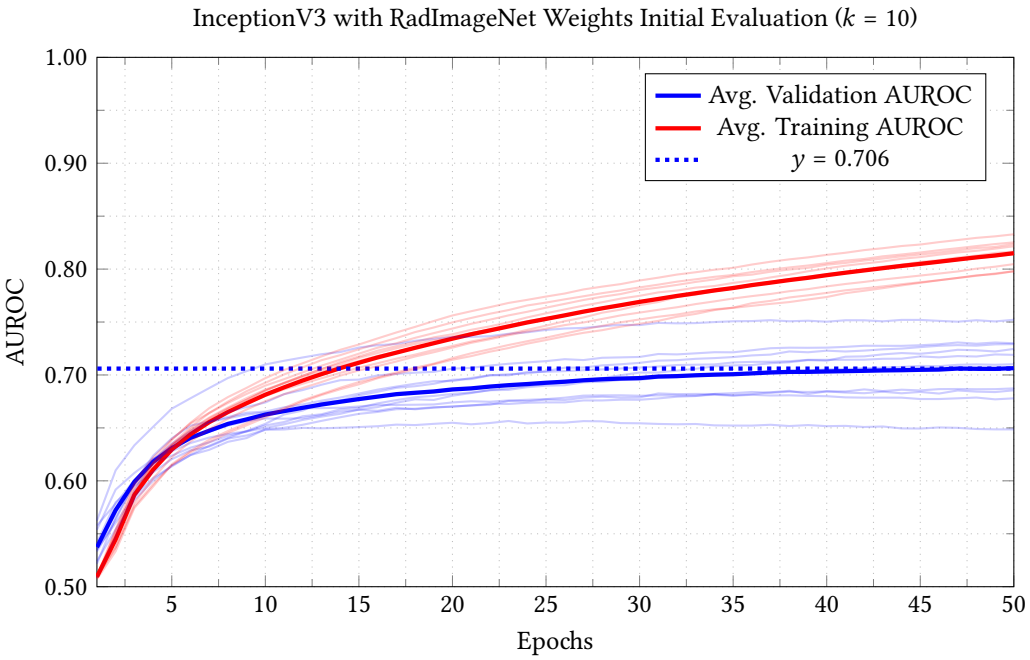

Figure 1.3: InceptionV3 with RadImageNet Weights

This preliminary information helps inform us that the technique of transfer learning is appropriate for our use case and dataset. All that follows now is for us to evaluate the ImageNet weights, and compare their performances together.

1.4.2 Base Model Trained on InceptionV3 Dataset

We now conduct the same experiment in a separate Jupyter notebook with ImageNet weights (Github). Recall that the difference between RadImagenet and ImageNet is that former contains approximately 4.1 million images [3], while the latter contains around 15.0 million [2]. While the RadImageNet dataset is exclusively sourced from medical imagery, including radiographs — the vastly larger ImageNet dataset has the potential to perform better, simply because the model was trained on a larger dataset. Does this assumption hold true?

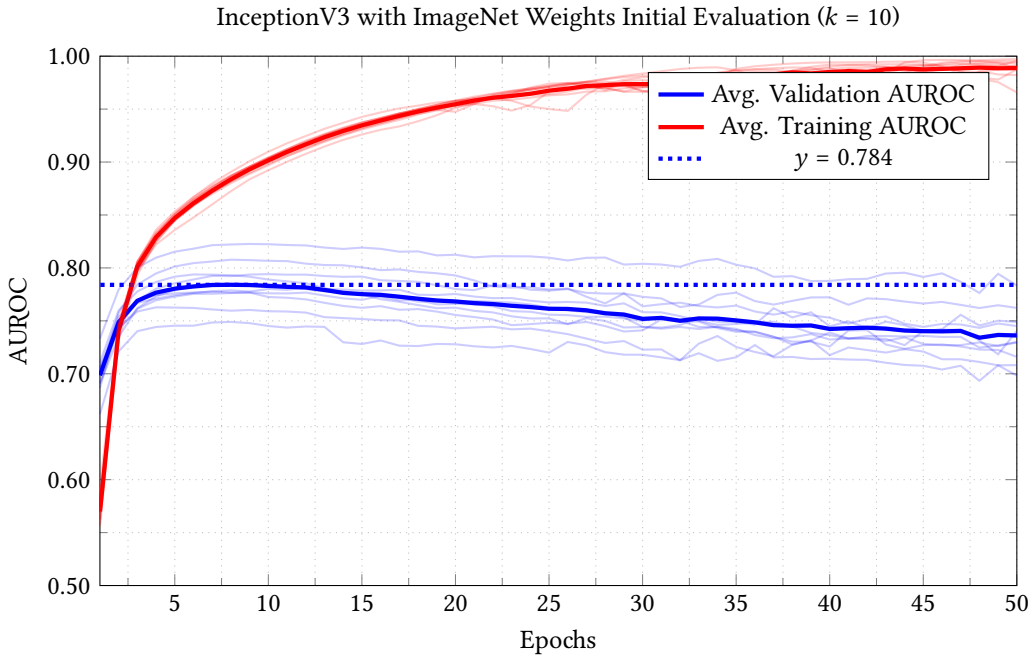

Figure 1.4: InceptionV3 with ImageNet Weights

Our data indicates that the InceptionV3 weights yield a higher validation AUROC of 0.784, in comparison to the RadImageNet model with an AUROC of 0.706. This means that although the RadImageNet dataset was more domain-specific to our needs (i.e. medical radiography classification), it appears the sheer size of the ImageNet datset yielded weights which performed better.

1.4.3 Comparison between RadImageNet and ImageNet

However, despite this difference in 'naive' (i.e. untuned, without hyperparameter optimisation), the performance characteristics of both models are quite different. Observe how the overfitting profile of the RadImageNet model is less severe than that of ImageNet, despite achieving a lower validation AUROC overall. Likewise, while the validation AUROC of the ImageNet model begins to decrease past epoch 15 due to overfitting, the validation AUROC of RadImageNet is still growing by epoch 50. Although our methodology calls for us to select the better performing model out of both of them, the information shown here offers room for further investigation — which can be the subject of a future study.

At this point, having assessed the 'naive' performance of both the RadImageNet weights and the regular ImageNet weights, we will select the better-performing

ImageNet weights as the basis for our transfer learning model. Going forward, we are now ready to tackle our problem directly: and begin the process of hyperparameter search.

1.5 Hyperparameter Search

Our methodology calls for a two-part hyperparameter search, consisting of Hyperparameter Search Regime I, and Hyperparameter Search Regime II. Regime I is concerned with finding the best combination of batch size and dropout, while Regime II is for the best learning rate and epsilon ϵ for the Adam optimizer. We conduct this hyperparameter search in two regimes as a compromise to the amount of compute resource that this project has. Although every hyperparameter influences every other hyperparameter, in practice it is rarely possible to conduct a hyperparameter search which varies over every hyperparameter under consideration at the same time.

Instead, by constraining ourselves to two hyperparameters per search regime, we get to avoid the 'curse of dimensionality'[4]. Regime I is concerned with hyperparameters that are specific to the model, while Regime II is concerned with hyperparameters which are specific to the optimizer.

1.5.1 Hyperparameter Search Regime I

Regime I uses a stochastic search process where the hyperparameter hypothesis space is randomly sampled t times. We use random search for Regime I because random search performs better than grid search over large hypothesis spaces, where the process of randomly sampling the hypothesis space will yield better hyperparameters as t increases, in a manner similar to Monte Carlo methods in statistics. [6] We begin by defining the range of batch size from 16 to 2048, and the range of possible dropout rates from 0.00 to 0.50. Next we construct a search function which conducts t trials, where during each trial a random portion of the hypothesis space is sampled, and then evaluated using k-fold validation with $k = 6$. We use a slightly lower k value as a concession to the amount of compute resources available: given 20 epochs of training per k-fold, and 6 folds per trial, this hyperparameter search regime conducts a total of 12,000 epochs of training over the course of 2 days (Github).

[4]The concept in which the size of a hypothesis space grows combinatorially with the number of free parameters, i.e. degrees of freedom, which are added.

```python
def hyperparameter_search(trials: int, kfolds: int = 6, epochs: int = 20) ->
↪  list[dict[str, Union[int, float, list[tf.keras.callbacks.History]]]]:
    search_results: list[dict[str, any]] = []

    for trial in range(trials):
        # Randomly pick hyperparameter options
        rng = np.random.default_rng()
        batch_size  : int   = rng.integers(16, 2048, endpoint=True)
        dropout_rate: float = rng.uniform(0.0, 0.5)

        # Conduct K-Fold cross-validation with given hyperparameters
        results: list[tf.keras.callbacks.History] = cross_validate(
            TransferLearningModel,
            ds_train_and_valid,
            k=kfolds
            epochs=epochs,
            batch_size=batch_size,
            model_kwargs={"dropout_rate": dropout_rate},
        )

        search_results.append({
            "batch_size"  : batch_size,
            "dropout_rate": dropout_rate,
            "history_list": k_fold_results
        })

    return search_results
```

Listing 6: Hyperparameter Search Regime I (Github)

The results of this hyperparameter search are then plotted, with the maximum observed validation AUROC represented as the colour of the data point in the following scatter plot. Note that the color-map is scaled according to the highest and lowest observed average validation AUROC.

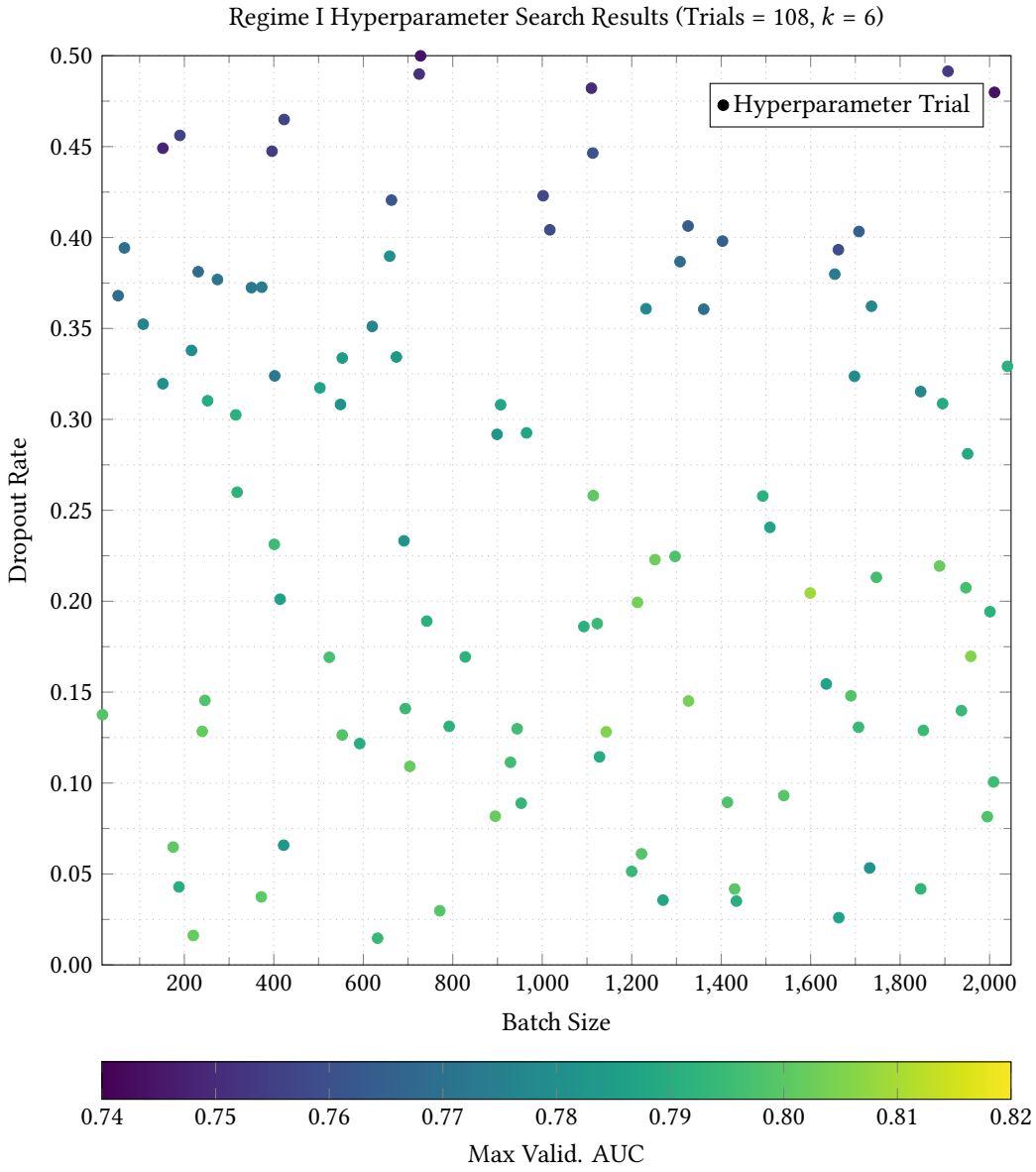

Figure 1.5: Results for the Hyperparameter Search Regime I

What were some of the learning rate and batch size combinations that we found? The following table lists out the top ten hyperparameter options.

Batch Size	Dropout Rate	Max Validation AUROC
1599	0.20450534699237183	0.8082009057203928
1958	0.16968093634911133	0.8052726884682974
1143	0.12816312484235065	0.8047231038411459
1327	0.14511673948299006	0.8034322460492452
1213	0.19936088970848825	0.8029904663562775
1252	0.22284567287235169	0.8021498719851176
1888	0.21933964138972928	0.8016549249490103
895	0.08177228898982053	0.8012685179710388
2009	0.10061578442603508	0.7949380973974863
401	0.23126040688192015	0.794610470533371

Table 1.4: Top Ten Hyperparameter Options for Regime I

As we can from a glance, a reasonable batch size seems to hover between 1500 to 2000. Likewise, a good dropout value seems to be around 0.20. This corresponds the colour gradient that we can see in the scatter plot. As an additional visualisation, we will also graph a random assortment of nine hyperparameter choices from our hypothesis space sample:

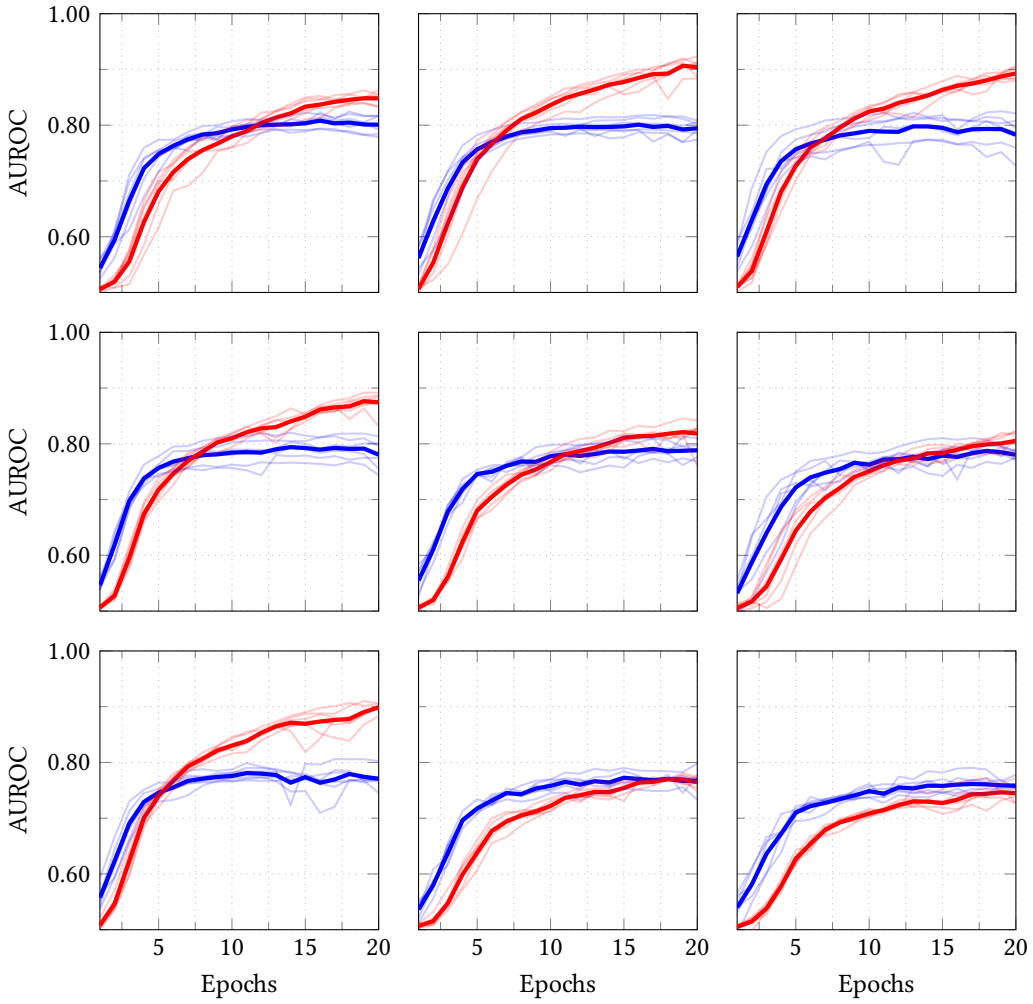

Figure 1.6: Random examples of models from hyperparameter search regime I.

The graphs above represent a selection of nine batch size and dropout combinations, sorted by best-performing to worse-performing. The first graph represents the top-performing hyperparameter combination, which we will graphs in greater detail below:

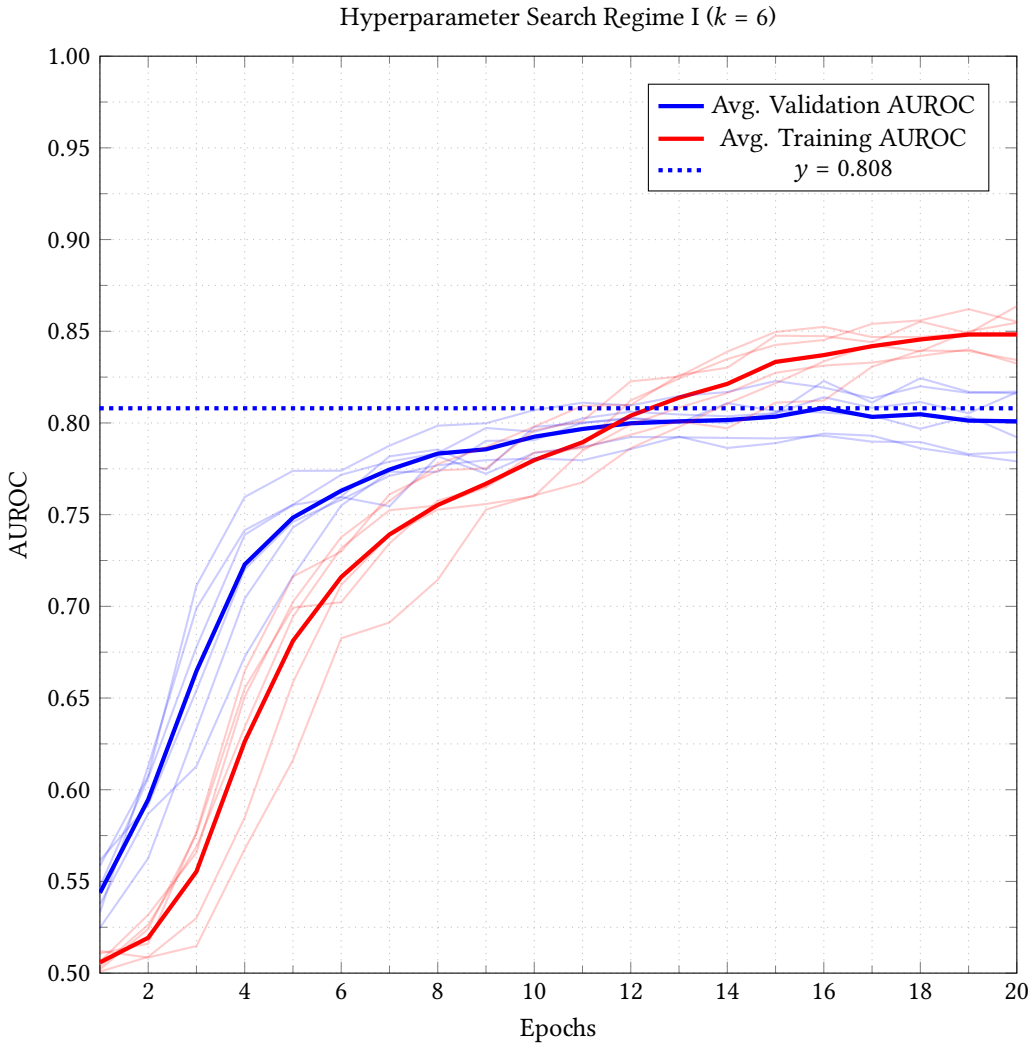

Figure 1.7: Best performing model in Regime I

Thus, we complete the first hyperparameter search regime, obtaining the hyperparameter options of 1599 for batch size and 0.205 for dropout. Going forward in all subsequent models, we will use those values, after rounding them up slightly to the nearest whole figure: `batch_size: int = 1600, dropout_rate: float = 0.20`. With this information, we may now move on to the second hyperparameter search regime, where we find the best hyperparameters for the Adam optimizer.

1.5.2 Hyperparameter Search Regime II

For Regime II, we aim to find the best learning rate and epsilon ϵ (an Adam-specific momentum factor). Possible values for epsilon and learning rate span not merely a continuous interval, but across several orders of magnitude. Due to the sheer size of the hypothesis space, and a lack of compute resources, I decided to perform a grid search for Regime II. We begin by implementing a grid search function, as seen below. We sample the grid at discrete intervals for both learning rate and epsilon:

```
def learning_rate_gridsearch(kfolds: int = 6) -> list[dict[str, Union[int, float,
↪  list[tf.keras.callbacks.History]]]]:
    # Grid i: 1.0 × 10^-1 ≤ learning_rate ≤ 1.0 × 10^-4
    learning_rates: list = [1 * np.float_power(10, -exp) for exp in range(1, 5)]
    # Grid j: 1.0 × 10^-1 ≤ epsilon_rate ≤ 1.0 × 10^-8
    epsilon_rates : list = [1 * np.float_power(10, -exp) for exp in range(1, 9)]

    search_results: list[dict[str, Union[int, float,
    ↪  list[tf.keras.callbacks.History]]]] = []
    for i, learning_rate in enumerate(learning_rates):
        for j, epsilon_rate in enumerate(epsilon_rates):
            # Conduct K-Fold Experiment
            k_fold_results: list[tf.keras.callbacks.History] = cross_validate(
                TransferLearningModel,
                ds_train_and_valid,
                k=kfolds,
                epochs=EPOCHS,
                batch_size=BATCH_SIZE,
                model_kwargs={"dropout_rate": DROPOUT_RATE}
                optimizer_kwargs={"learning_rate": learning_rate, "epsilon":
                ↪  epsilon_rate},
            )
            search_results.append({
                "learning_rate": learning_rate,
                "epsilon_rate" : epsilon_rate,
                "history_list" : k_fold_results
            })

    return search_results
```

Listing 7: Hyperparameter Search Regime II (Github)

Running the above search routine in a Jupyter notebook (Github) on our hosted compute provider, we obtain the following results:

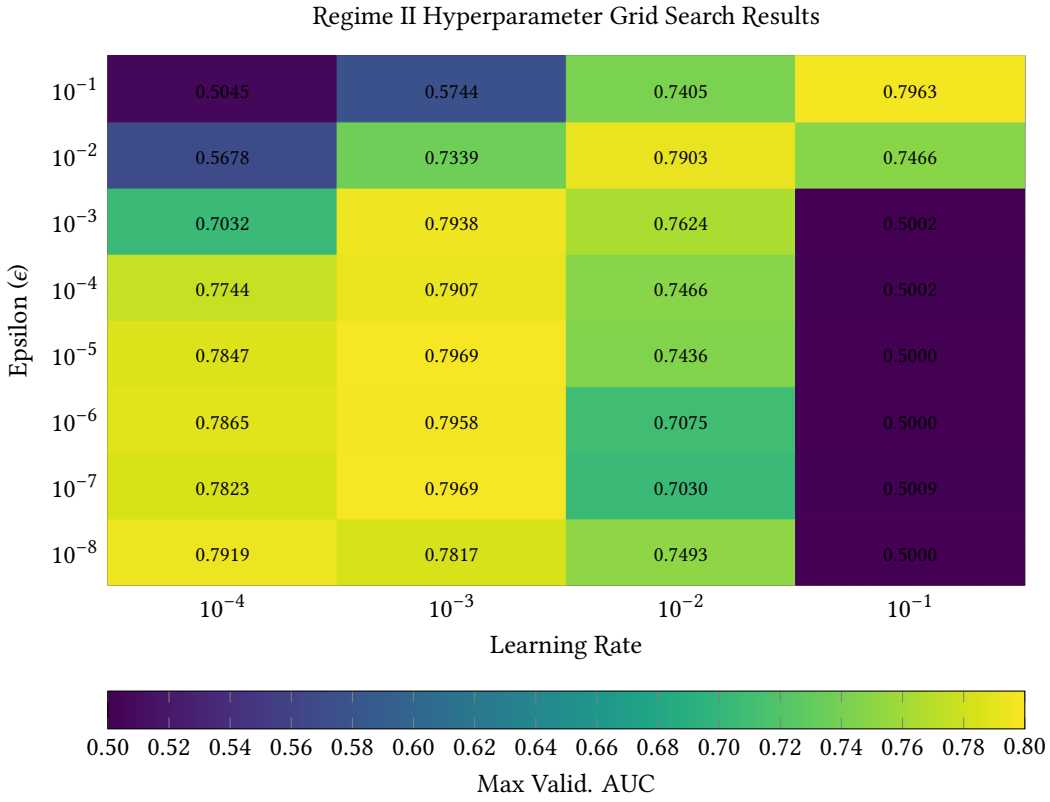

Figure 1.8: Results for the Hyperparameter Search Regime II

Learning Rate	Epsilon ϵ	Max Validation AUROC
1.0×10^{-3}	1.0×10^{-7}	0.7969321310520172
1.0×10^{-3}	1.0×10^{-5}	0.7969251374403635
1.0×10^{-1}	1.0×10^{-1}	0.7962859372297922
1.0×10^{-3}	1.0×10^{-6}	0.7957556545734406
1.0×10^{-3}	1.0×10^{-3}	0.7938481668631235
1.0×10^{-4}	1.0×10^{-8}	0.7918900847434998
1.0×10^{-3}	1.0×10^{-4}	0.7907490432262421
1.0×10^{-2}	1.0×10^{-2}	0.7903381884098053
1.0×10^{-4}	1.0×10^{-6}	0.7865246037642161
1.0×10^{-4}	1.0×10^{-5}	0.7846748034159342

Table 1.5: Top Ten Hyperparameter Options for Regime II

1.5.3 Final Hyperparameters

Thus, we have completed our hyperparameter search. After implementing and running Regime I and Regime II (Github), we were able to find the following hyperparameter options for our final model:

Batch Size	Dropout Rate	Learning Rate	Epsilon (ϵ)
1600	0.20	1.0×10^{-3}	1.0×10^{-7}

Table 1.6: Final Hyperparameters for InceptionV3 with ImageNet Weights

The implementation, analysis, and search of our model hyperparameters are now complete. Now we are ready for the final evaluation of the complete model on our hold-out test set. This will be done in the final part of our project, in the Part IV evaluation.

Bibliography

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. DOI: 10.1109/5.726791.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," pp. 248–255, 2009.

[3] X. Mei, Z. Liu, P. M. Robson, *et al.*, "Radimagenet: An open radiologic deep learning research dataset for effective transfer learning," *Radiology: Artificial Intelligence*, vol. 0, no. ja, e210315, 0. DOI: 10.1148/ryai.210315. eprint: https://doi.org/10.1148/ryai.210315. [Online]. Available: https://doi.org/10.1148/ryai.210315.

[4] D. B. Whelan, M. Bhandari, D. Stephen, *et al.*, "Development of the radiographic union score for tibial fractures for the assessment of tibial fracture healing after intramedullary fixation," *Journal of Trauma and Acute Care Surgery*, vol. 68, no. 3, 2010, ISSN: 2163-0755. [Online]. Available: https://journals.lww.com/jtrauma/Fulltext/2010/03000/Development_of_the_Radiographic_Union_Score_for.24.aspx.

[5] M. V. Narkhede, P. P. Bartakke, and M. S. Sutaone, "A review on weight initialization strategies for neural networks," *Artificial Intelligence Review*, vol. 55, no. 1, pp. 291–322, Jan. 1, 2022, ISSN: 1573-7462. DOI: 10.1007/s10462-021-10033-z. [Online]. Available: https://doi.org/10.1007/s10462-021-10033-z.

[6] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, Dec. 2, 2012.

Appendix A

Additional Materials

A.1 Project Code and Github Repository

All of the implementation details, model architecture, and data are made available in this project's Git repository (Github). Every code listing contains a link to the specific implementation, and different experiments also contain links to their corresponding Jupyter notebooks where the code was originally run. As a part of this project's commitment to reproducibility, all Jupyter notebooks are documented, and readers are encouraged to follow along and run the experiments for themselves. For further information, please see the repository `README.md`.

https://github.com/ShenZhouHong/radiography-ai-project/

A.1.1 Initial Evaluation Models

Jupyter notebooks used to run the initial evaluations of LeNet 1998, InceptionV3 with end-to-end training, and initial transfer learning models:

https://github.com/ShenZhouHong/radiography-ai-project/tree/master/python/initial-evaluation

A.1.2 Hyperparameter Search Code

Jupyter notebooks used to perform the hyperparameter search regime.

https://github.com/ShenZhouHong/radiography-ai-project/tree/master/python/hyperparam-search

A.1.3 Analysis Notebooks

Jupyter notebooks used to analyse the raw data, process for insights and visualisations, and output CSV files:

https://github.com/ShenZhouHong/radiography-ai-project/tree/master/python/analysis