

**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	K-Fold Evaluation	3
1.2	Establishing a Baseline	5
1.2.1	Shallow Convolutional Neural Network	5
1.2.2	End-to-End Training with InceptionV3	7
1.2.3	Baseline Metrics	9
1.3	InceptionV3 with Transfer Learning	9
1.3.1	Base Model Trained on RadImageNet Dataset	9
1.3.2	Base Model Trained on InceptionV3 Dataset	10
1.3.3	Comparison between RadImageNet and ImageNet	11
1.4	Hyperparameter Search	11
1.4.1	Hyperparameter Search Regime I	11
1.4.2	Hyperparameter Search Regime II	15
1.4.3	Final Hyperparameters	16
1.5	Final Model Performance	16
A	Additional Materials	18
A.1	Project Code and Github Repository	18
A.1.1	Initial Evaluation Models	18
A.1.2	Hyperparameter Search Code	18
A.1.3	Analysis Notebooks	18

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 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 initialisation¹ [4], 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.

```
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  $\frac{k-1}{k}$  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](#))

¹This is particularly true on small datasets with unbalanced classes like ours.

```

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):

        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](#))

```

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](#))

1.2 Establishing a Baseline

1.2.1 Shallow Convolutional Neural Network

```
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.lenet1998: 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.lenet1998,
            self.classifier
        ])

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

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

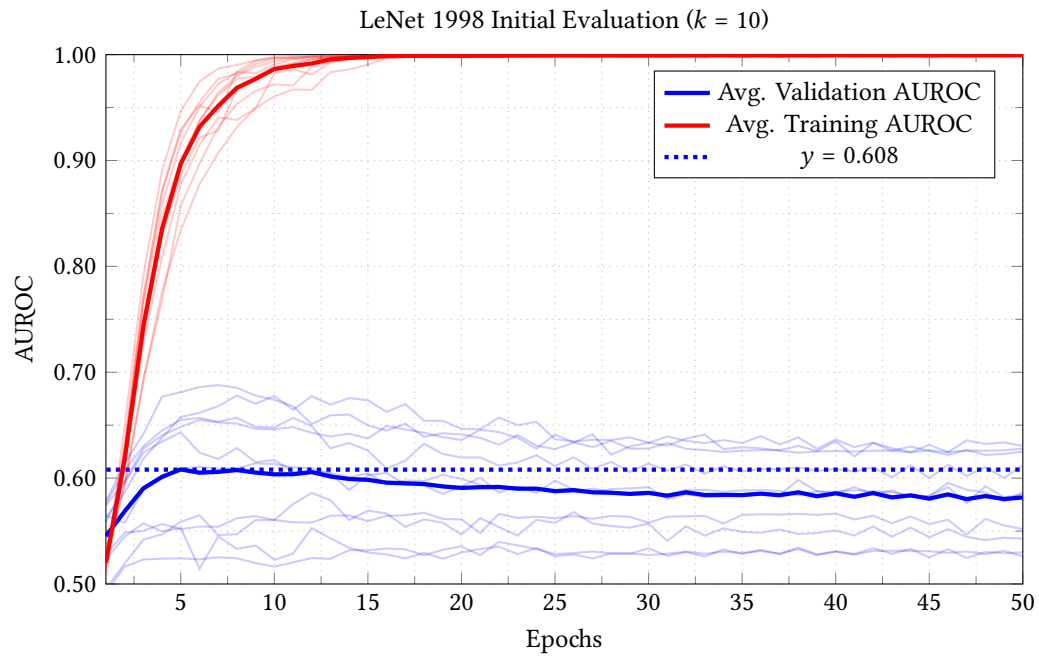


Figure 1.1: Baseline shallow CNN based on the LeNet 1998 architecture

1.2.2 End-to-End Training with InceptionV3

```
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](#))

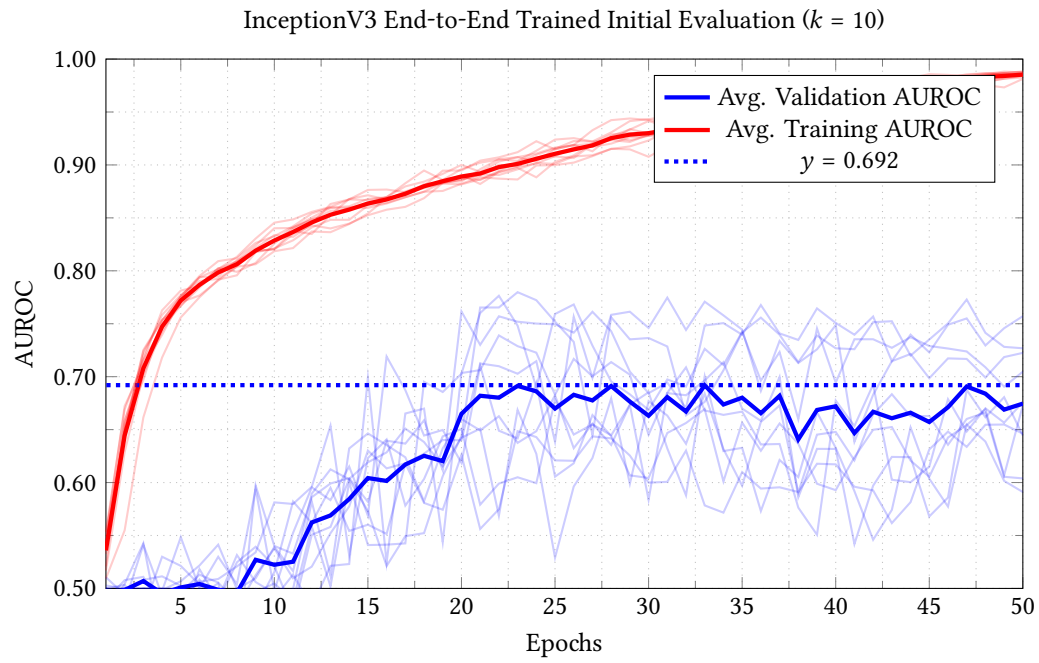


Figure 1.2: InceptionV3 Model Trained on Study Data.

1.2.3 Baseline Metrics

1.3 InceptionV3 with Transfer Learning

1.3.1 Base Model Trained on RadImageNet Dataset

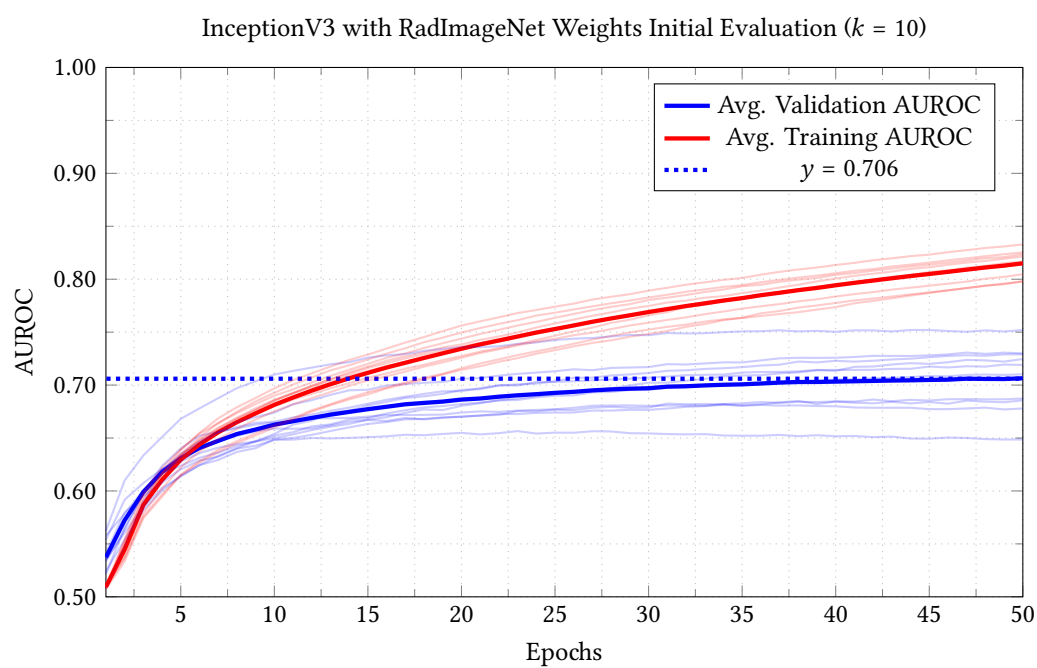


Figure 1.3: InceptionV3 with RadImageNet Weights

1.3.2 Base Model Trained on InceptionV3 Dataset

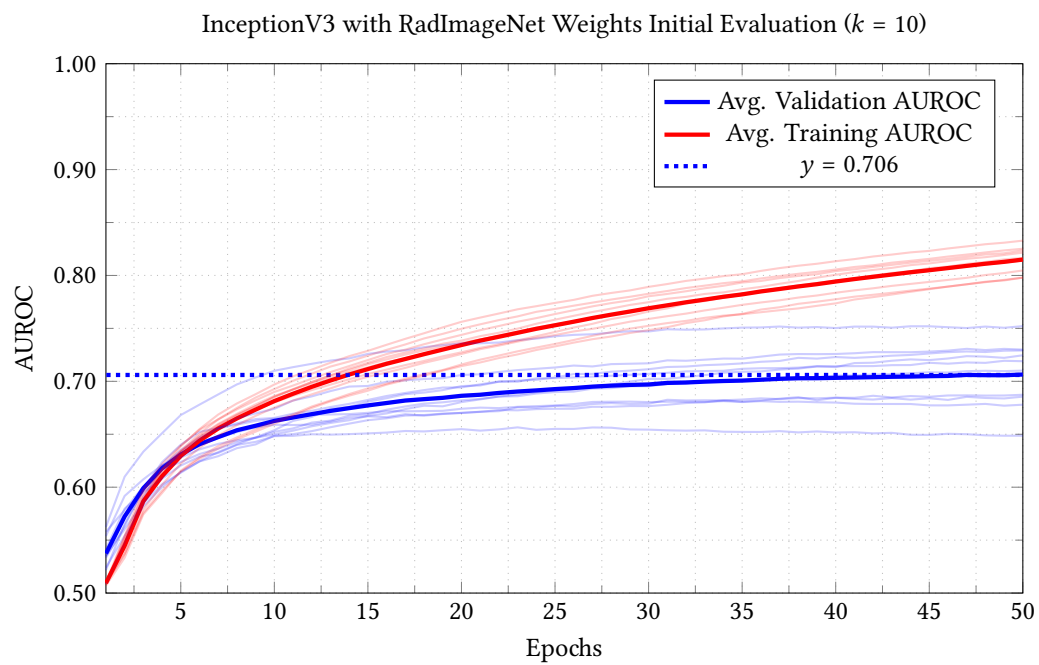


Figure 1.4: InceptionV3 with RadImageNet Weights

1.3.3 Comparison between RadImageNet and ImageNet

1.4 Hyperparameter Search

1.4.1 Hyperparameter Search Regime I

```
def hyperparameter_search(trials: int, kfold: 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=kfold,
            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](#))

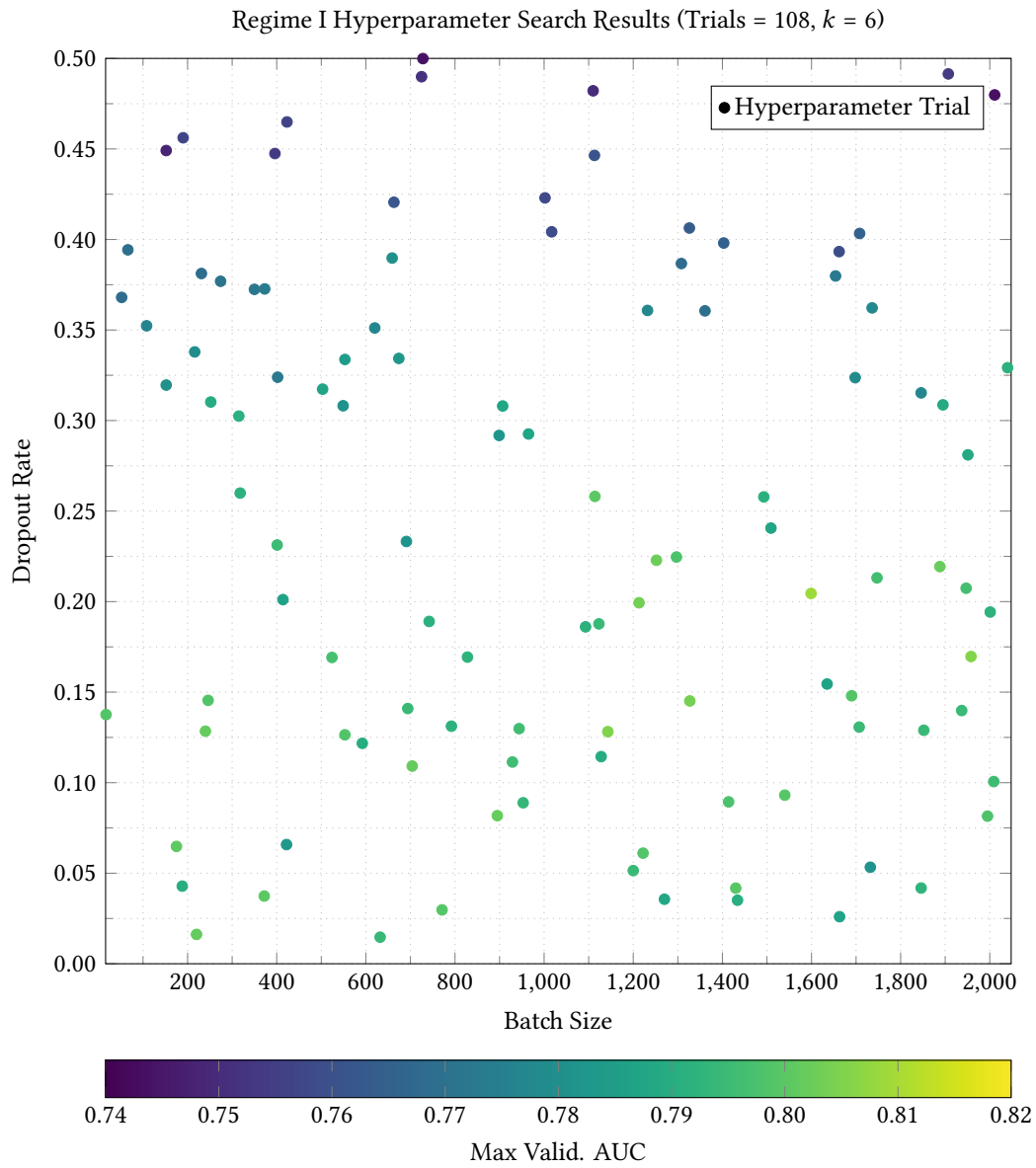


Figure 1.5: Results for the Hyperparameter Search Regime I

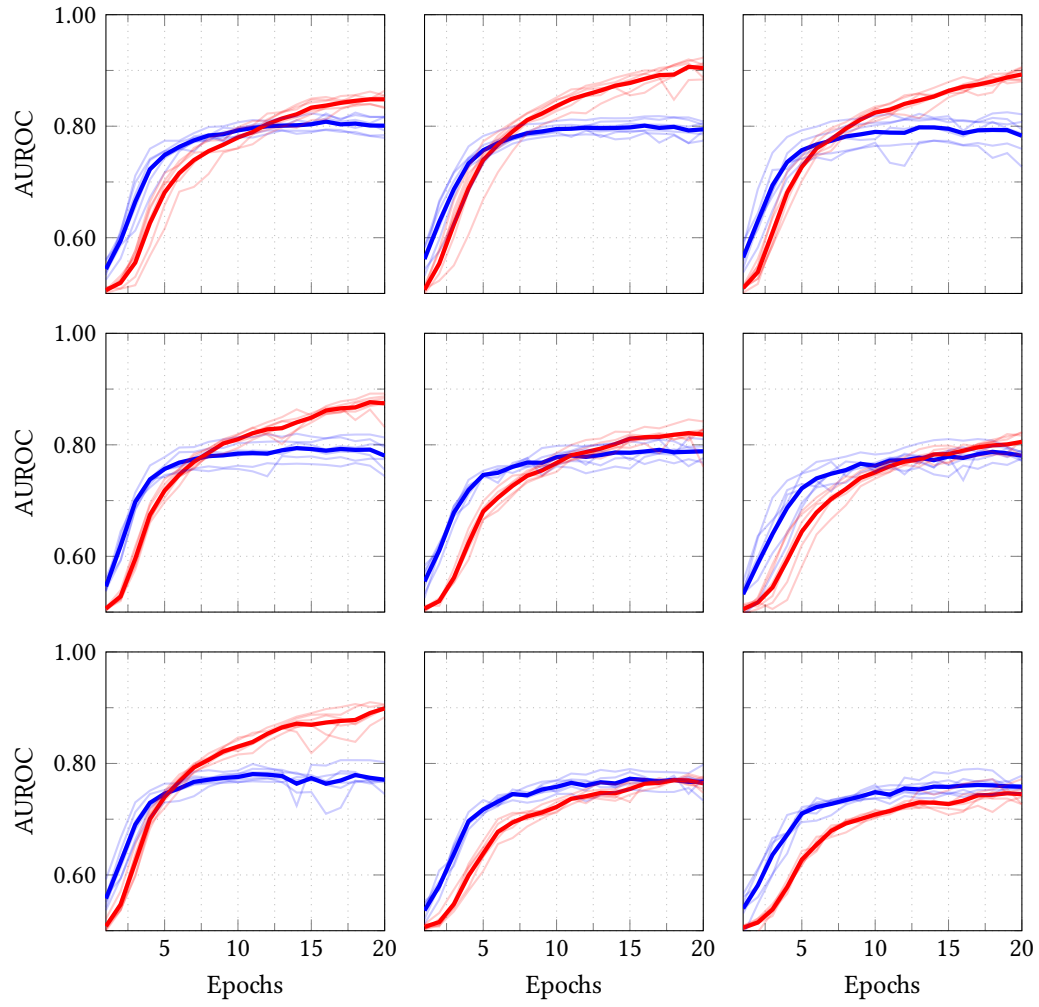


Figure 1.6: Examples of model performance from hyperparameter regime I search.

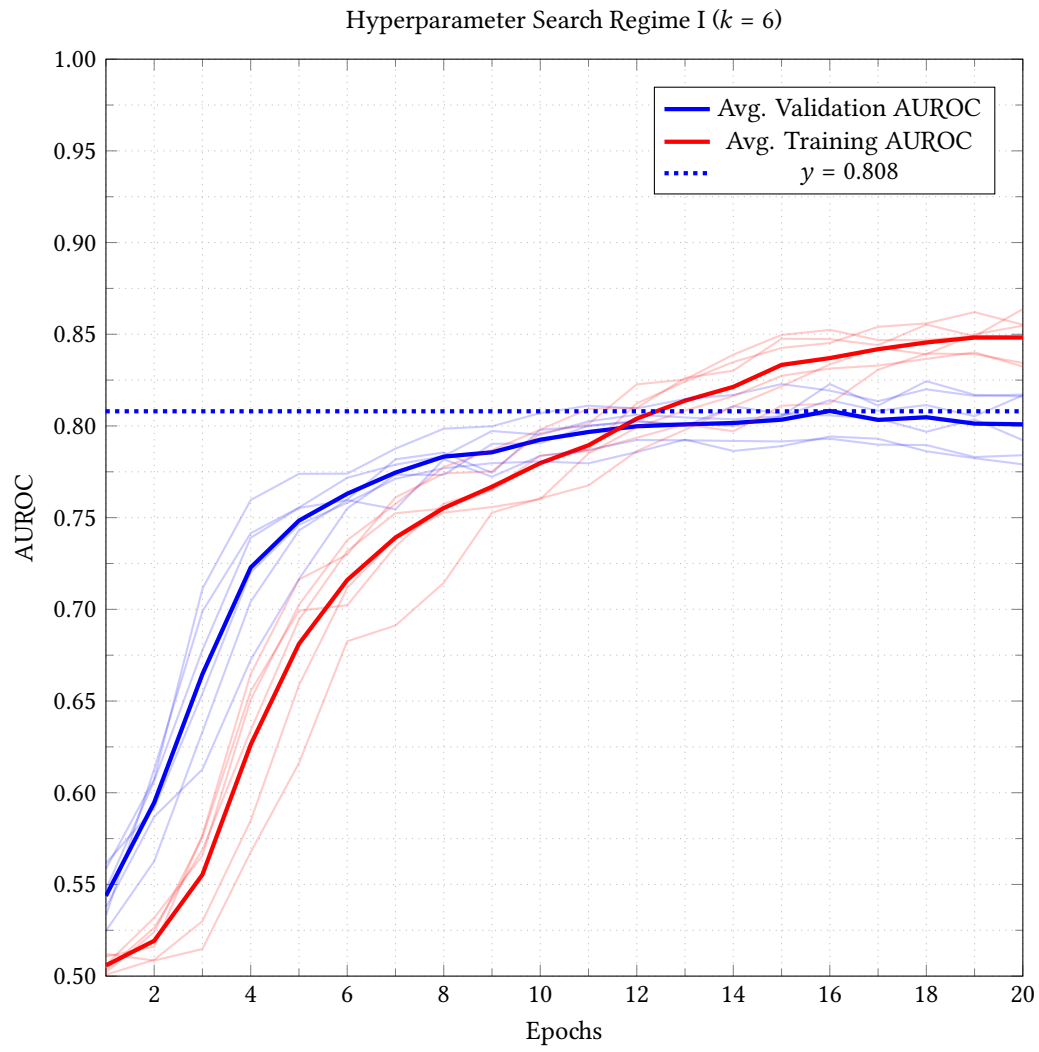


Figure 1.7: Best performing model in Regime I

1.4.2 Hyperparameter Search Regime II

```
def learning_rate_gridsearch(kfolds: int = 6) -> list[dict[str, Union[int, float,
↳ list[tf.keras.callbacks.History]]]]:
    # Grid i:  $1.0 \times 10^{-1} \leq \text{learning\_rate} \leq 1.0 \times 10^{-4}$ 
    learning_rates: list = [1 * np.float_power(10, -exp) for exp in range(1, 5)]
    # Grid j:  $1.0 \times 10^{-1} \leq \text{epsilon\_rate} \leq 1.0 \times 10^{-8}$ 
    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](#))



Figure 1.8: Results for the Hyperparameter Search Regime II

1.4.3 Final Hyperparameters

1.5 Final Model Performance

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](https://doi.org/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](https://doi.org/10.1148/ryai.210315). eprint: <https://doi.org/10.1148/ryai.210315>. [Online]. Available: <https://doi.org/10.1148/ryai.210315>.
- [4] 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](https://doi.org/10.1007/s10462-021-10033-z). [Online]. Available: <https://doi.org/10.1007/s10462-021-10033-z>.

Appendix A

Additional Materials

A.1 Project Code and Github Repository

All of the Python code used in this project (including experiment and analysis code) are available within the project Git repository, hosted on [Github](#). The code is located within the `python/` directory of the repository root:

<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>