

Wavelet radiomics features from multiphase CT images for screening hepatocellular carcinoma: analysis and comparison

Van Ha Tang^{1,2}, Soan T. M. Duong^{1,2,*}, Chanh D. Tr. Nguyen^{1,3}, Thanh M. Huynh^{1,3}, Vo T. Duc⁴, Chien Phan⁴, Huyen Le⁴, Trung Bui⁵, and Steven Q. H. Truong^{1,3}

¹VinBrain JSC, Vietnam

²Le Quy Don Technical University, Vietnam

³VinUniversity, Vietnam

⁴Ho Chi Minh City Medicine and Pharmacy University, Vietnam

⁵Adobe Research, USA

*Corresponding author, email: soanduong@lqdtu.edu.vn

Supplementary information

In this supplementary document, we present three additional pieces of information: a list of used radiomics features, logistic sparsity algorithm, and Python code for training and validating the deep convolutional neural networks (DCNNs) used in the paper.

List of used radiomics features

Feature extraction has been performed in both original CT and wavelet-filtered images. Each image of the original CT domain contributes 100 attributes, including 18 first-order statistics features, 14 shape features, and 68 texture features. However, each wavelet-filtered image contributes only 86 features (100 features excluding 14 shape features). The list of all the radiomics features that have been used is listed in the below tables.

Supplementary Table 1: The 18 radiomics first-order statistics features.

Feature types (abbre.)	Feature names
FIRST-ORDER	10Percentile
FIRST-ORDER	90Percentile
FIRST-ORDER	Energy
FIRST-ORDER	Entropy
FIRST-ORDER	InterquartileRange
FIRST-ORDER	Kurtosis
FIRST-ORDER	Maximum
FIRST-ORDER	MeanAbsoluteDeviation
FIRST-ORDER	Mean
FIRST-ORDER	Median
FIRST-ORDER	Minimum
FIRST-ORDER	Range
FIRST-ORDER	RobustMeanAbsoluteDeviation
FIRST-ORDER	RootMeanSquared
FIRST-ORDER	Skewness
FIRST-ORDER	TotalEnergy
FIRST-ORDER	Uniformity
FIRST-ORDER	Variance

Supplementary Table 2: The 14 radiomics shape features.

Feature types (abbrev.)	Feature names
SHAPE	Elongation
SHAPE	Flatness
SHAPE	LeastAxisLength
SHAPE	MajorAxisLength
SHAPE	Maximum2DDiameterColumn
SHAPE	Maximum2DDiameterRow
SHAPE	Maximum2DDiameterSlice
SHAPE	Maximum3DDiameter
SHAPE	MeshVolume
SHAPE	MinorAxisLength
SHAPE	Sphericity
SHAPE	SurfaceArea
SHAPE	SurfaceVolumeRatio
SHAPE	VoxelVolume

Supplementary Table 3: The 22 radiomics gray level co-occurrence matrix (GLCM) features.

Feature types (abbrev.)	Feature names
GLCM	Autocorrelation
GLCM	JointAverage
GLCM	ClusterProminence
GLCM	ClusterShade
GLCM	ClusterTendency
GLCM	Contrast
GLCM	Correlation
GLCM	DifferenceAverage
GLCM	DifferenceEntropy
GLCM	DifferenceVariance
GLCM	JointEnergy
GLCM	JointEntropy
GLCM	Imc1
GLCM	Imc2
GLCM	Idm
GLCM	Idmn
GLCM	Id
GLCM	Idn
GLCM	InverseVariance
GLCM	MaximumProbability
GLCM	SumEntropy
GLCM	SumSquares

Supplementary Table 4: The 16 radiomics gray level run length matrix (GLRLM) features.

Feature types (abbrev.)	Feature names
GLRLM	GrayLevelNonUniformity
GLRLM	GrayLevelNonUniformityNormalized
GLRLM	GrayLevelVariance
GLRLM	HighGrayLevelRunEmphasis
GLRLM	LongRunEmphasis
GLRLM	LongRunHighGrayLevelEmphasis
GLRLM	LongRunLowGrayLevelEmphasis
GLRLM	LowGrayLevelRunEmphasis
GLRLM	RunEntropy
GLRLM	RunLengthNonUniformity
GLRLM	RunLengthNonUniformityNormalized
GLRLM	RunPercentage
GLRLM	RunVariance
GLRLM	ShortRunEmphasis
GLRLM	ShortRunHighGrayLevelEmphasis
GLRLM	ShortRunLowGrayLevelEmphasis

Supplementary Table 5: The 16 radiomics gray level size zone matrix (GLSZM) features.

Feature types (abbrev.)	Feature names
GLSZM	GrayLevelNonUniformity
GLSZM	GrayLevelNonUniformityNormalized
GLSZM	GrayLevelVariance
GLSZM	HighGrayLevelZoneEmphasis
GLSZM	LargeAreaEmphasis
GLSZM	LargeAreaHighGrayLevelEmphasis
GLSZM	LargeAreaLowGrayLevelEmphasis
GLSZM	LowGrayLevelZoneEmphasis
GLSZM	SizeZoneNonUniformity
GLSZM	SizeZoneNonUniformityNormalized
GLSZM	SmallAreaEmphasis
GLSZM	SmallAreaHighGrayLevelEmphasis
GLSZM	SmallAreaLowGrayLevelEmphasis
GLSZM	ZoneEntropy
GLSZM	ZonePercentage
GLSZM	ZoneVariance

Supplementary Table 6: The 14 radiomics gray level dependence matrix (GLDM) features.

Feature types (abbre.)	Feature names
GLDM	DependenceEntropy
GLDM	DependenceNonUniformity
GLDM	DependenceNonUniformityNormalized
GLDM	DependenceVariance
GLDM	GrayLevelNonUniformity
GLDM	GrayLevelVariance
GLDM	HighGrayLevelEmphasis
GLDM	LargeDependenceEmphasis
GLDM	LargeDependenceHighGrayLevelEmphasis
GLDM	LargeDependenceLowGrayLevelEmphasis
GLDM	LowGrayLevelEmphasis
GLDM	SmallDependenceEmphasis
GLDM	SmallDependenceHighGrayLevelEmphasis
GLDM	SmallDependenceLowGrayLevelEmphasis

Logistic sparsity algorithm

This section presents the computational steps for solving the logistic sparsity model in (1):

$$\min_{\theta} \left\{ f(\theta) = \sum_{i=1}^M -\log p(y_i | \mathbf{x}_i, \theta) + \lambda \|\theta\|_1 \right\}. \quad (1)$$

Before solving this ℓ_1 regularized optimization problem, let us consider the unregularized version, i.e., the objective function $f(\theta)$ with the first term only:

$$\min_{\theta} \sum_{i=1}^M -\log p(y_i | \mathbf{x}_i, \theta). \quad (2)$$

We now derive a technique based on Newton's method to solve Problem (2) and then extend the derivation to the sparsity regularized version.

Newton's method solves Problem (2) in an iterative manner. At each iteration, this method uses the current point for approximating the objective function with the second-order Taylor expansion and searching for a step direction. In particular, let θ_k be the estimate at the k th iteration. Newton's technique computes a step direction γ_k by the following evaluation at the current point θ_k ,

$$\gamma_k = \theta_k - \mathbf{H}^{-1}(\theta_k) \mathbf{g}(\theta_k). \quad (3)$$

In (3), $\mathbf{H}(\theta_k)$ and $\mathbf{g}(\theta_k)$ are, respectively, the Hessian and gradient of the objective function evaluated at the current estimate θ_k . Once the step direction is obtained, the next estimate is computed as

$$\theta_{k+1} = (1-t) \theta_k + t \gamma_k. \quad (4)$$

Here, the step size $t \in [0, 1]$ can be determined using a backtracking line search¹, that is designed to minimize the objective function given in (2).

As the aim is to solve the ℓ_1 -regularized optimization problem (1) efficiently, we now show that instead of computing the step direction γ_k directly using (3), this point can be found as a solution to a weighted LS problem, see^{2,3}. In doing so, let us define two auxiliary variables—a diagonal matrix $\mathbf{A}$ and a vector $\mathbf{z}$,

$$\begin{aligned} A_{i,i} &= \sigma(\theta_k^T \mathbf{x}_i) [1 - \sigma(\theta_k^T \mathbf{x}_i)], \\ z_i &= \mathbf{x}_i^T \theta_k + \frac{[1 - \sigma(y_i \theta_k^T \mathbf{x}_i)] y_i}{A_{i,i}}, \end{aligned} \quad (5)$$

for $i = 1, 2, \dots, M$. Now, the Hessian and gradient are computed by $\mathbf{H}^{-1}(\theta_k) = -\mathbf{X}^T \mathbf{A} \mathbf{X}$, and $\mathbf{g}(\theta_k) = \mathbf{X}^T \mathbf{A} (\mathbf{z} - \mathbf{X} \theta_k)$. Equation (3) can therefore be rewritten as

$$\gamma_k = (\mathbf{X}^T \mathbf{A} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{A} \mathbf{z}. \quad (6)$$

The result in (6) means that the step direction γ_k is the solution to the following weighted LS problem:

$$\gamma_k = \arg \min_{\gamma} \|(\mathbf{A}^{\frac{1}{2}} \mathbf{X}) \gamma - \mathbf{A}^{\frac{1}{2}} \mathbf{z}\|_2^2. \quad (7)$$

For the ℓ_1 regularized optimization problem, we can augment this regularizer to obtain:

$$\gamma_k = \arg \min_{\gamma} \|(\mathbf{A}^{\frac{1}{2}} \mathbf{X}) \gamma - \mathbf{A}^{\frac{1}{2}} \mathbf{z}\|_2^2 + \lambda \|\gamma\|_1. \quad (8)$$

Thus, our task now is to solve Problem (8) to obtain the step direction, and then do the line search to determine the step size and finally update the next estimate using (4). Problem (8) can be solved efficiently using the proximal splitting method and the soft-thresholding technique. Let $\mathbf{a}_k$ be an auxiliary variable computed at the k th iteration,

$$\mathbf{a}_k = \gamma_k - \alpha (\mathbf{A}^{\frac{1}{2}} \mathbf{X})^T [(\mathbf{A}^{\frac{1}{2}} \mathbf{X}) \gamma - \mathbf{A}^{\frac{1}{2}} \mathbf{z}], \quad (9)$$

where the step size α is selected as $\alpha = 1 / \|(\mathbf{A}^{\frac{1}{2}} \mathbf{X})\|_2^2$ for fast convergence— $\|(\mathbf{A}^{\frac{1}{2}} \mathbf{X})\|_2^2$ being the Lipschitz constant of the gradient of the first term in (8). Then, the solution to (8) is obtained by applying a soft-thresholding operator to the variable $\mathbf{a}_k$ with a threshold $\tau = \alpha \lambda$:

$$\gamma_{k+1} = \mathcal{T}(\mathbf{a}_k, \tau). \quad (10)$$

Here, $\mathcal{T}(\cdot)$ is the component-wise shrinkage function given by,

$$\mathcal{T}(x, \tau) = \text{sgn}(x) \max(|x| - \tau, 0) = \frac{x}{|x|} \max(|x| - \tau, 0). \quad (11)$$

The steps of the proposed algorithm are summarized in Algorithm 1. Given the training set $\{\mathbf{x}_i, y_i\}$, for $i = 1, 2, \dots, M$, the algorithm starts by initializing the parameter θ in Step 3, then estimates the step direction in Steps 6–10, follows by the update of the parameter in Step 12. As the algorithm minimizes the cost function $f(\theta)$, its convergence can be ascertained when the changes of this function are very small, i.e., smaller than a pre-defined $\text{tol} = 10^{-4}$.

Algorithm 1: LSR for radiomics feature selection and HCC and non-HCC classification.

- 1: **Input:** M training samples $\{(\mathbf{x}_i, y_i), i = 1, \dots, M\}$, the feature matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M]^T \in \mathbb{R}^{M \times N}$, hyperparameter λ , and a tolerance $\text{tol} = 10^{-4}$, and $\text{MaxIter} = 100$.
 - 2: **Output:** the parameter vector θ .
 - 3: Initialize the parameter $\theta_0 = 0$, and iteration index $k = 0$.
 - 4: **repeat**
 - 5: Compute $\mathbf{A}$ and $\mathbf{z}$ using (5):
 - 6: $A_{i,i} = \sigma(\theta_k^T \mathbf{x}_i) [1 - \sigma(\theta_k^T \mathbf{x}_i)]$,
 - 7: $z_i = \mathbf{x}_i^T \theta_k + \frac{[1 - \sigma(y_i \theta_k^T \mathbf{x}_i)] y_i}{A_{i,i}}$.
 - 8: Estimate the step direction γ_{k+1} using (9) and (10):
 - 9: $\mathbf{a}_k = \gamma_k - \alpha (\mathbf{A}^{\frac{1}{2}} \mathbf{X})^T [(\mathbf{A}^{\frac{1}{2}} \mathbf{X}) \gamma - \mathbf{A}^{\frac{1}{2}} \mathbf{z}]$,
 - 10: $\gamma_{k+1} = \mathcal{T}(\mathbf{a}_k, \tau)$.
 - 11: Find the step size t by a backtracking line-search and update the parameter θ using (4):
 - 12: $\theta_{k+1} = (1 - t) \theta_k + t \gamma_{k+1}$.
 - 13: Increment $k = k + 1$, evaluate the objective function $f(\theta)$ in (1), and check for the stopping criterion.
 - 14: **until** $(|f(\theta)^{k+1} - f(\theta)^k| / |f(\theta)^k| < \text{tol})$ or $(k > \text{MaxIter})$
-

Python code for training and inference of deep convolutional neural networks (DCNNs)

This section presents the main Python code used to train DCNNs using transfer learning. In particular, the DCNNs are used as fixed feature extractors. In other words, their weights are frozen except for the final fully connected layer. This last layer is replaced with our binary HCC and non-HCC classifier with random weights, and only this layer is trained. Here, we use the *PyTorch* deep learning library⁴ to implement the classification task. First, we prepare the dataset by defining a class named *CustomImageDataset*. This class is inherited from the parent class of `torch.utils.data.Dataset` responsible for accessing and processing single instances of the data. Another concept is `DataLoader`, which is used to get data instances from the dataset, collect them in batches, and return them for usage in the training loop. The implementation of the *CustomImageDataset* is given below.

```
1 import torch
2 import torch.nn as nn
3 from torch.utils.data import Dataset
4 from torchvision import datasets, models, transforms
5 from scipy import ndimage
6 import copy
7
8 # This class is designed for data augmentation (performing image rotation)
9 class Rotate_transform(object):
10     # This method is to perform rotation on a single-phase image
11     def rotate_data(self, volume):
12         volume = volume.squeeze()
13         volume = volume.permute(1, 2, 0)
14         angles = [-20, -10, -5, 5, 10, 20]
15         idx = torch.randint(0, len(angles), (1,)) # pick angles to rotate image randomly
16         angle = angles[idx]
17         max_vol = torch.max(volume)
18         min_vol = torch.min(volume)
19         volume = ndimage.rotate(volume, angle, reshape=False)
20         volume = torch.from_numpy(volume)
21         volume[volume < min_vol] = min_vol
22         volume[volume > max_vol] = max_vol
23         volume = volume.permute(2, 0, 1)
24         return volume
25
26     # This method is to apply the rotation for all the three phase images
27     # The call method enables instances to behave like functions
28     def __call__(self, sample):
29         img_venous, img_delay, img_arterial, label = sample
30         img_venous = self.rotate_data(img_venous)
31         img_delay = self.rotate_data(img_delay)
32         img_arterial = self.rotate_data(img_arterial)
33         sample = img_venous, img_delay, img_arterial, label
34         return sample
35
36 # A user-defined custom dataset class specifically designed for handling the multiphase CT image data.
37 # It is created to handle our own dataset class by subclassing torch.utils.data.Dataset.
38 class CustomImageDataset(Dataset):
39     # Define a constructor: initialize the attributes:
40     def __init__(self, imgs, labels, phase_lst, transform = None, target_transform = None):
41         self.labels = labels
42         self.imgs = imgs
43         self.phase_lst = phase_lst
44         self.transform = transform
45         self.target_transform = target_transform
46
47     # Customize the behavior of the len() function when applied to instances of the class
48     def __len__(self):
49         return len(self.labels)
50
51     # allow instances of the class to be indexed and accessed, enabling object instances to support
52     # indexing and iteration.
53     def __getitem__(self, idx):
54         img_dict = dict()
55         for phase_ in self.phase_lst:
56             img = self.imgs[phase_][idx]
57             label = self.labels[idx]
58             image = normalize(img)
```

```

58     label = torch.tensor(label)
59     image = torch.from_numpy(image)
60     image = image.permute(2, 0, 1)
61     img_dict[phase_] = image
62     sample = img_dict['venous'], img_dict['delay'], img_dict['arterial'], label
63     if self.transform is not None:
64         sample = self.transform(sample)
65     return sample

```

Supplementary Listing 1. Define a CustomImageDataset for training and inference of the deep CNN models.

We now write a general function to train and validate a model, including scheduling the learning rate, tracking the training history, and saving the best model.

```

1 def train_validate_model(model, dataloaders, criterion, optimizer, scheduler, num_epochs,
2   checkpoint_file):
3     val_acc_history = []
4     train_acc_history = []
5     val_loss_history = []
6     train_loss_history = []
7     training_history = {}
8
9     best_model_wts = copy.deepcopy(model.state_dict())
10    best_acc = 0.0
11    best_loss = sys.float_info.max
12
13    for epoch in range(num_epochs):
14        print('Epoch {} / {}'.format(epoch, num_epochs - 1))
15        print('-' * 10)
16        # Each epoch has a training and validation phase
17        for phase in ['train', 'val']:
18            if phase == 'train':
19                model.train() # Set model to training mode
20            else:
21                model.eval() # Set model to evaluate mode
22
23            running_loss = 0.0
24            running_corrects = 0
25
26            # Iterate over data
27            for inputs_img_v, inputs_img_d, inputs_img_a, labels in dataloaders[phase]:
28                inputs_img_v = inputs_img_v.to(device)
29                inputs_img_d = inputs_img_d.to(device)
30                inputs_img_a = inputs_img_a.to(device)
31                labels = labels.to(device)
32
33                # zero the parameter gradients
34                optimizer.zero_grad()
35
36                # forward
37                # track history if only in train
38                with torch.set_grad_enabled(phase == 'train'):
39                    # Get model outputs and calculate loss
40
41                    outputs = model(inputs_img_v, inputs_img_d, inputs_img_a)
42                    labels = labels.long()
43                    loss = criterion(outputs, labels)
44                    _, preds = torch.max(outputs, 1)
45
46                    # backward and optimize only if in training phase
47                    if phase == 'train':
48                        loss.backward()
49                        optimizer.step()
50
51                # statistics
52                running_loss += loss.item() * inputs_img_v.size(0)
53                running_corrects += torch.sum(preds == labels.data)
54
55            epoch_loss = running_loss / len(dataloaders[phase].dataset)
56            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

```

```

56         if epoch >= 1:
57             print('{} Loss: {:.4f} Acc: {:.4f} Best Loss: {:.4f}'
58                   .format(phase, epoch_loss, epoch_acc, best_loss))
59         else:
60             print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc))
61
62         # deep copy the best model on the validation set
63         if phase == 'val' and epoch_acc > best_acc:
64             best_acc = epoch_acc
65             best_model_wts = copy.deepcopy(model.state_dict())
66
67         if phase == 'val' and epoch_loss < best_loss:
68             best_loss = epoch_loss
69             best_model_loss = copy.deepcopy(model.state_dict())
70
71         # track the training history
72         if phase == 'val':
73             val_acc_history.append(epoch_acc)
74             val_loss_history.append(epoch_loss)
75         else:
76             train_acc_history.append(epoch_acc)
77             train_loss_history.append(epoch_loss)
78
79         scheduler.step()
80         print(f'learning rate: {scheduler.get_last_lr()[0]:.7f}')
81
82     print('Training complete:')
83     print('Best val Acc: {:.4f}'.format(best_acc))
84     print('Best val loss: {:.4f}'.format(best_loss))
85
86     # load best model weights
87     model.load_state_dict(best_model_wts)
88
89     # save best model weights with respect to loss:
90     checkpoint = {'epoch': num_epochs,
91                  'model_state_dict': best_model_loss,
92                  'optimizer_state_dict': optimizer.state_dict(),
93                  'loss': train_loss_history,
94                  'val_loss': val_loss_history}
95
96     torch.save(checkpoint, checkpoint_file)
97     training_history = {'val_acc': val_acc_history,
98                       'val_loss': val_loss_history,
99                       'train_acc': train_acc_history,
100                      'train_loss': train_loss_history}
101
102     return model, training_history

```

Supplementary Listing 2. The training and validating procedure for the deep CNN models.

We now define a deep CNN model for the multiphase CT liver lesion imaging analysis for the binary classification of HCC and non-HCC. The model loads a pre-trained DCNN model including VGGNet⁵, ResNet⁶, DenseNet⁷, and GoogleNet⁸ and reset the final fully connected layer. Here, we customize it for processing our multiphase CT images.

```

1  # Define the class used to replace the last fully connected layer in pretrained models
2  class Identity(nn.Module):
3      def __init__(self):
4          super().__init__()
5
6      def forward(self, x):
7          return x
8
9  # Define a pretrained model for single-phase CT HCC and non-HCC classification
10 class Singlephase_PreTrained_Model(nn.Module):
11     def __init__(self, model_name = None, num_classes = 2):
12         super().__init__()
13         if model_name == 'densenet':
14             self.pretrained = models.densenet121(pretrained=True)
15             for param in self.pretrained.parameters(): # freeze all the learned weights.
16                 param.requires_grad = False

```



```

17     self.num_fttrs = self.pretrained.classifier.in_features
18     # set the last layer to the identity so the net becomes a feature extractor.
19     self.pretrained.classifier = Identity()
20
21     if model_name == 'resnet':
22         self.pretrained = models.resnet50(pretrained=True)
23         for param in self.pretrained.parameters():
24             param.requires_grad = False
25         self.num_fttrs = self.pretrained.fc.in_features
26         self.pretrained.fc = Identity()
27
28     if model_name == 'googlenet':
29         self.pretrained = models.googlenet(pretrained = True)
30         for param in self.pretrained.parameters():
31             param.requires_grad = False
32         self.num_fttrs = self.pretrained.fc.in_features
33         self.pretrained.fc = Identity()
34
35     if model_name == 'vgg':
36         self.pretrained = models.vgg11_bn(pretrained = True)
37         for param in self.pretrained.parameters():
38             param.requires_grad = False
39         self.num_fttrs = self.pretrained.classifier[6].in_features
40         self.pretrained.classifier[6] = Identity()
41
42     self.num_classes = num_classes
43     self.relu = nn.ReLU()
44     self.fc1 = nn.Linear(self.num_fttrs, 512)
45     self.fc2 = nn.Linear(512, self.num_classes)
46     self.Dropout = nn.Dropout(0.5)
47
48
49     def forward(self, x):
50         x = self.pretrained(x)
51         x = x.view(-1, self.num_fttrs)
52         x = self.relu(self.fc1(x))
53         x = self.Dropout(x)
54         x = self.fc2(x)
55         return x
56
57 # Define a pretrained model for multiphase CT HCC and non-HCC classification
58 class Multiphase_PreTrained_Model(nn.Module):
59     def __init__(self, model_name = None, num_classes = 2):
60         super().__init__()
61         self.pretrained_v = Singlephase_PreTrained_Model(model_name, num_classes)
62         self.pretrained_d = Singlephase_PreTrained_Model(model_name, num_classes)
63         self.pretrained_a = Singlephase_PreTrained_Model(model_name, num_classes)
64
65     def forward(self, x_v, x_d, x_a):
66         x_v = self.pretrained_v(x_v)
67         x_d = self.pretrained_d(x_d)
68         x_a = self.pretrained_a(x_a)
69         x = (x_v + x_d + x_a) / 3
70         return x

```

Supplementary Listing 3. Define a deep CNN model for the multiphase CT liver lesion HCC and non-HCC classification.

We can now use the given classes to make the Datasets and DataLoaders, perform necessary settings, and select the optimizer and criterion (loss function) to train and validate the model. The code is given below.

```

1 # Settings for training the model:
2 num_classes = 2          # Number of classes in the dataset
3 batch_size = 32          # Batch size for training (can be changed depending on how much memory we have)
4 num_epochs = 500         # Number of epochs
5 learning_rate = 1e-3     # Learning rate
6 model_name = 'resnet'    # Name of the pretrained model
7
8 # Define datasets and dataloaders:
9 phase_lst = ['venous', 'delay', 'arterial']
10 training_data = CustomImageDataset(imgs_valid_train, y_train, phase_lst, transform = transforms.Compose
    ([Rotate_transform()]))

```

```

11 test_data = CustomImageDataset(imgs_valid_test, y_test, phase_lst)
12 training_loader = torch.utils.data.DataLoader(training_data, batch_size = batch_size, shuffle = True)
13 test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, shuffle = False)
14 dataloaders_dict = {'train': training_loader, 'val': test_loader}
15
16 # Load the model, select optimizer, loss function, train and validate the model:
17 model = Multiphase_PreTrained_Model(model_name, num_classes)
18 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
19 model = model.to(device)
20 params_to_update = model.parameters()
21 optimizer = torch.optim.Adam(params_to_update, lr = learning_rate)
22 scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 50, gamma = 0.1)
23 criterion = nn.CrossEntropyLoss()
24 checkpoint_file = model_name + "_checkpoint.pth"
25 model_best, training_history = train_validate_model(model,
26                                                    dataloaders_dict,
27                                                    criterion,
28                                                    optimizer,
29                                                    scheduler,
30                                                    num_epochs = num_epochs,
31                                                    checkpoint_file = checkpoint_file)

```

Supplementary Listing 4. Script for preparing the dataloaders and selecting the optimizer and loss function to train and validate the model.

References

1. Boyd, S. & Vandenberghe, L. *Convex optimization* (Cambridge University Press, 2004).
2. Minka, T. A comparison of numerical optimizers for logistic regression (2003).
3. Lee, S.-I., Lee, H., Abbeel, P. & Ng, A. Efficient L1 regularized logistic regression. vol. 21 (2006).
4. Paszke, A. *et al.* Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 8024–8035 (Curran Associates, Inc., 2019).
5. Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations* (2015).
6. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778, DOI: [10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90) (2016).
7. Huang, G., Liu, Z., Van Der Maaten, L. & Weinberger, K. Q. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2261–2269, DOI: [10.1109/CVPR.2017.243](https://doi.org/10.1109/CVPR.2017.243) (2017).
8. Szegedy, C. *et al.* Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1–9, DOI: [10.1109/CVPR.2015.7298594](https://doi.org/10.1109/CVPR.2015.7298594) (2015).