UNIVERSITÀ
DEGLI STUDI
DI MILANO

MACHINE LEARNING PROJECT

Prof. Nicolò Cesa-Bianchi

OBJECT DETECTION USING CONVOLUTIONAL NEURAL NETWORKS FOR Muffin vs Chihuahua Dataset

Shojaat Joodi Bigdilo – 14088A

July 2023

Contents

Neural Networks and Deep Learning

Neural Networks

Neural Networks or Artificial Neural Networks is a machine learning model based on the biological neural networks present in the brains of humans.

A neural network learns without being explicitly using previous examples or data and tries to construct a model that is able to correctly predict future unseen data.

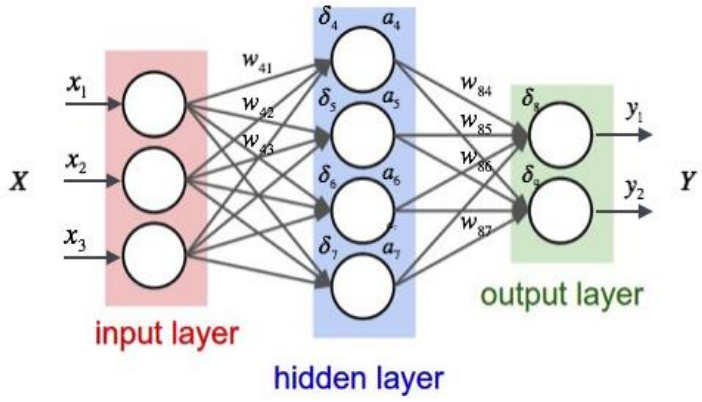


Figure 1: Artificial Neural Network

An artificial neural network is formed by connecting nodes called **artificial neurons** modeled after the real neurons present in the biological brain and each connection between nodes in subsequent layers model the synapses in the biological brain. The connection between the neurons is called edges. Each edge has a weight associated with it and each neuron performs a weighted sum of the inputs from nodes in the previous layers and then passes the sum through an **activation function** aka threshold or bias.

Weights on the edges are updated an in iterative manner. This weight update rule is based on the use of **Gradient Descent.** The algorithm used for updating the weights iteratively is called **Backpropagation.**

Output is calculated at the output layer again by the weighted sum of all the outputs of the nodes in the previous layer and then the **error** is calculated by comparing the calculated output and the real output of the training example that was fed in to generate the activations.

Several **Error/Loss Functions** are available for the calculation of error at the output layer. one of the most common and the one that we've used in our network (discussed later) for binary classification is the **Binary Cross-Entropy Function.**

- ### Backpropagation

  In backpropagation the error computed at the output is backpropagated throughout the network using gradient descent. Each weight gets an update respective to the derivative

of the error function w.r.t the weight. It utilizes the chain-rule to iteratively compute derivatives at each layer and then the weights are updated accordingly.

$$E = \frac{1}{2}(t - y)^2$$

Equation 1: Mean Squared Error Function

So, if we use the mean squared error function to calculate the error/loss on the output node then the change in each weight can be calculated using the following equation.

$$\Delta w_{ij} = -n\frac{\partial E}{\partial Wij} = n\delta_j O_i$$

Equation 2: Weight Update Rule

- Loss Function (Cross-Entropy)

$$H(y) = -\sum_i y_i log p_i$$

Equation 3: Cross-Entropy Loss

Cross-entropy loss aka log loss is a function that calculates the probability value for each node in the output layer in a neural network. The cross-entropy loss increases as the predicted probabilities diverges from the actual values for the probabilities. A log loss of 0 would mean a 100% accurate model.

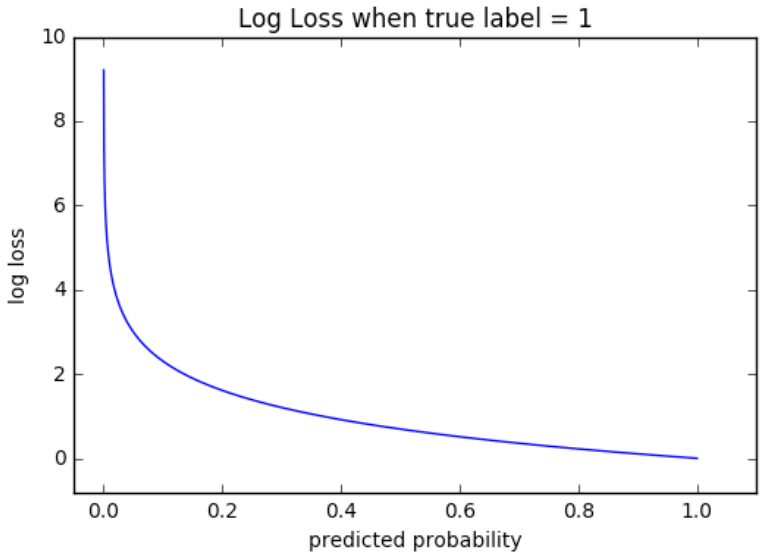


Figure 2: Cross-Entropy Loss Function

- Zero-one Loss

  The simplest loss function is the zero-one loss. It literally counts how many mistakes a hypothesis function h(x) makes on the training set. For every single example it suffers a loss of 1 if it is mispredicted, and 0 otherwise. The normalized zero-one loss returns the fraction of misclassified training samples, also often referred to as the training error.

$$\mathcal{L}_{0/1}(h) = \frac{1}{n}\sum_{i=1}^{n} \delta_{h(\mathbf{x}_i)\neq y_i}, \text{ where } \delta_{h(\mathbf{x}_i)\neq y_i} = \begin{cases} 1, & \text{if } h(\mathbf{x}_i) \neq y_i \\ 0, & \text{o.w.} \end{cases}$$

Deep Learning

A neural consists of three layers, an input layer, a hidden layer and an output layer. In deep learning or deep neural networks, multiple hidden layers are used to generate even more complex networks able to tackle even more challenging problems. A deep neural network is able to find even non-linear separation between different classes.

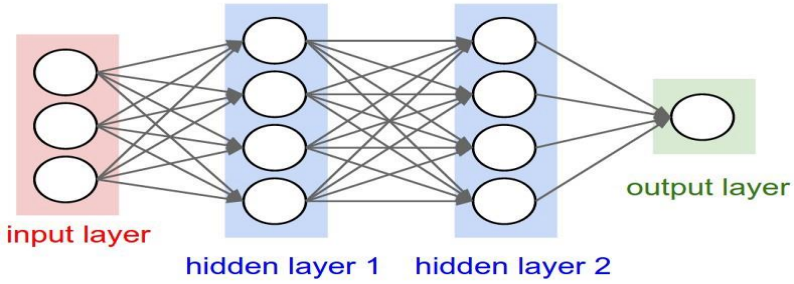


Figure 3: Deep Neural Network

Convolutional Neural Networks

A Convolutional Neural Network (CNN) is an implementation of Deep Neural Networks which are mainly used when working with imagery data.

Convolutional neural networks use multiple hidden layers of different purposes to either extract important features from the image (**convolutional layers**) or manipulate the image for aggregating pixel data (**pooling layers**). Convolutional neural networks allow for the extraction of positional and rotational invariant features from an image and there are usually much better than a general deep network as the object in consideration could be placed anywhere in unseen future images and therefore these positional and rotational invariant features help locate the object with ease.

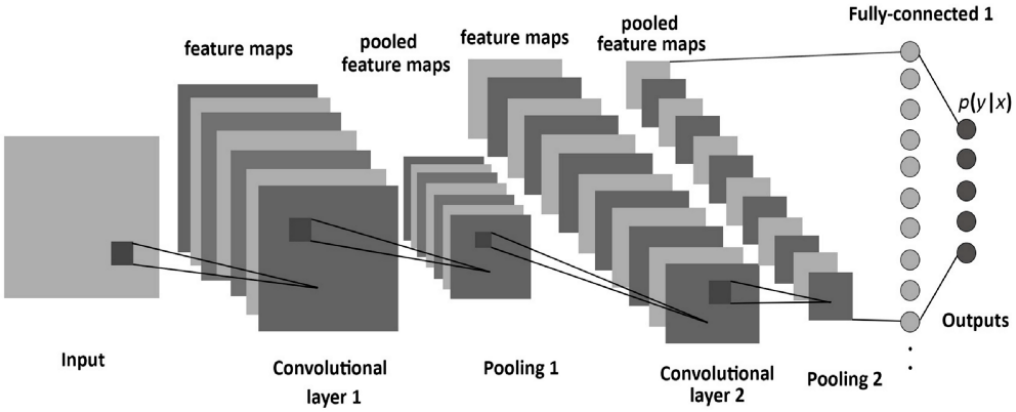


Figure 4: Convolutional Neural Network Architecture

- Convolutional Layer

  Convolutional layers apply a convolution operation to the input and passes the result to the next layer. A convolutional filter or feature filter is passed over the whole input image and activations are recorded. Each filter/convolutional layer generates a new feature map. **Convolution** is a process of combining the input data with a filter or a kernel, which is a small matrix of weights, to produce a feature map or an output matrix that represents the presence of certain features in the input. Filter or a kernel can be used to detect certain features in the input, such as edges, corners, colors, etc.

- Feature Maps

  Convolutional layers produce multiple feature maps (also called channels or feature planes) as outputs. Each feature map represents the response of a specific filter to different features in the input.

- Pooling Layer

  Pooling layers are used to aggregate pixel data to help positional and rotational invariant feature extraction. This reduces the size of the feature maps and extract the most important features.

- Fully Connected Layer

  Fully connected layers connect every neuron in one layer to another layer. In fully connected layers, the neuron applies a linear transformation to the input vector through a weight's matrix. A non-linear transformation is then applied to the product through a non-linear activation function.

- Dropout

  Dropout reduces the over-fitting by using a Dropout layer after every FC layer. Dropout layer has a probability, (p), associated with it and is applied at every neuron of the

response map separately. It randomly switches off the activation with the probability p. In other words, it randomly sets a fraction of the input units to zero during training, which helps prevent overfitting and improves the model's generalization ability.

- Batch Normalization

  Batch normalization is a technique that improves the performance and stability of deep neural networks by normalizing the inputs of each layer to have a mean of zero and a standard deviation of one. It is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.

Neural Networks Architectures

Neural network architectures refer to the specific structures and arrangements of layers within a neural network. Different architectures are designed to address various types of tasks and data. Convolutional Neural Networks (CNNs) have been widely used and evolved over the years with various architectures designed to address different challenges in computer vision tasks. Here are some popular CNN architectures:

AlexNet

AlexNet is a deep convolutional neural network architecture that was introduced by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012. The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer.

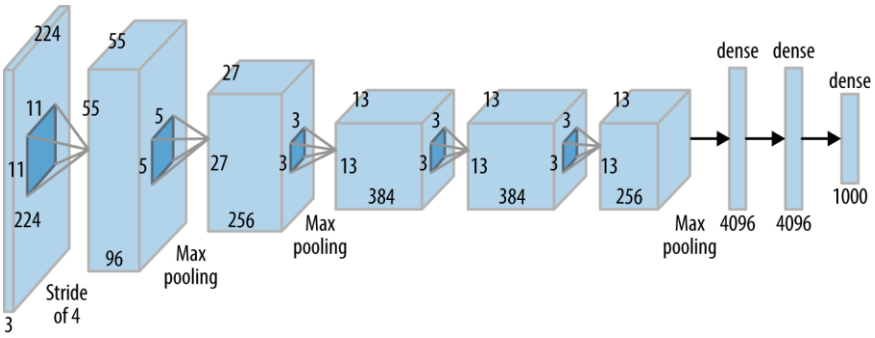


Figure 5: AlexNet Architecture

VGG16

VGGNet, or the Visual Geometry Group Network, is a convolutional neural network architecture proposed was introduced by Karen Simonyan and Andrew Zisserman in 2014 and gained popularity due to its

simplicity and strong performance in image classification tasks. VGGNet comes in different variants based on its depth, namely VGG16 and VGG19.

In VGG16 the numbers indicate the number of layers (including convolutional and fully connected layers) in the network. Only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Replace large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3*3 kernel-sized filters one after another.

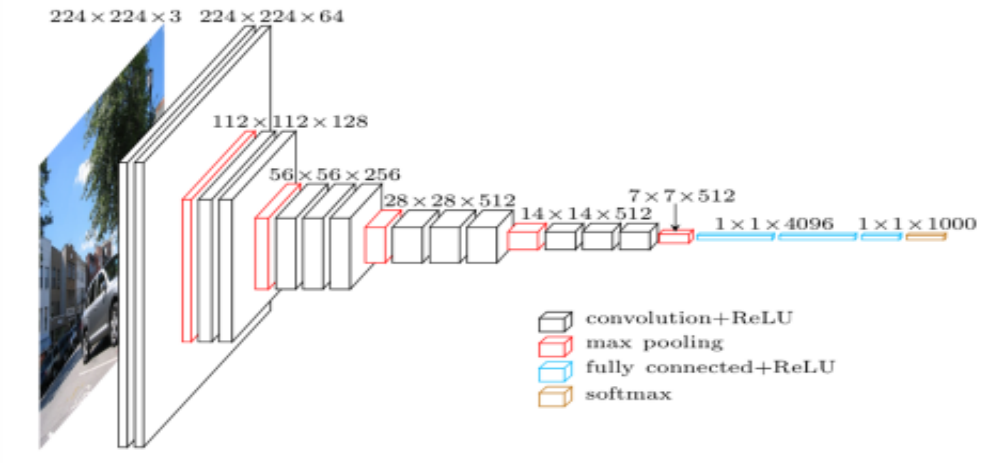


Figure 6: VGG16 Architecture

ResNet

ResNet, also known as Residual Neural Network, is a deep convolutional neural network architecture that was developed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2015. It was designed to overcome the challenge of vanishing gradients in extremely deep networks. By incorporating residual connections, ResNet allows the effective training of networks with hundreds or even thousands of layers. This breakthrough enables the exploration of deeper and more powerful neural networks in various applications.

The motivation behind ResNet arises from the need to tackle complex problems by adding more layers to Deep Neural Networks, which leads to enhanced accuracy and performance. The rationale behind increasing the number of layers is that each layer learns increasingly intricate features. For instance, in image recognition, early layers may detect edges, intermediate layers may identify textures, and subsequent layers can recognize objects and more. But it has been found that there is a maximum threshold for depth with the traditional Convolutional neural network model.

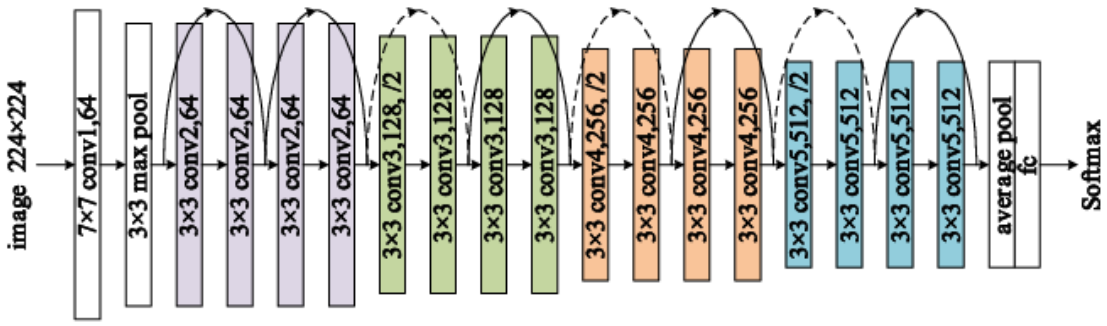


Figure 7: ResNet18 Architecture

CNN Model Implementation

In our project we use Deep Convolutional Neural Networks to extract important features from Muffin and Chihuahua images and make a proper model in order to classify new images in test dataset. To initially train our network, our dataset which consists of 6000 images of Muffin and Chihuahua, we used 4733 images for train and validation. We train the network using 2559 Chihuahua and 2174 Muffin images with the idea that if our network is able to learn some features that can detect new image from test data.

Data Preparing

Our training dataset contains separate image folders for Muffin and Chihuahua (2559 Chihuahua and 2174 Muffin images). To prepare the dataset for training, we take all the data i.e. the Muffin and Chihuahua images with their labels. The data is split accordingly into train, test and validation sets.

In the first step, we read data from folders and add them in a list. Then we assign 1 to Muffin and 0 to Chihuahua as target value and create a NumPy array out of the lists.

The images are scaled down to a consistent size (120x120 pixels) to ensure uniformity. Then images are preprocessed by converting them to RGB pixel values.

We load all the images in train and test dataset and convert them into a form that our neural network understands. Specifically, PyTorch works with Tensor objects. (A tensor is just a multidimensional matrix, i.e. an N-d array.) To easily convert our image data into tensors, we use "pickle" function to save dataset to pkl file. Finally, we shuffled the train data since the labels of the Muffin and Chihuahua images were in order. shuffling allows the model to learn from a diverse set of samples, prevents overfitting to specific patterns, and aids in efficient convergence during training.

Figure 8. Data Preprocessing phase code

```
def DataPreparing(path):
    filenames = glob.glob(path)
    y_all = []
    for file in filenames:
        _,_,_,_,_,_,_,imgY,_= file.split("/")
        if imgY == 'muffin':
            imgY = 0
        else:
            imgY = 1

        y_all.append(imgY)
        #endfor
    X_all = []
    imagesX = [cv2.imread(img) for img in filenames]
    for img in imagesX:
        img = cv2.resize(img,(120, 120))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        X_all.append(img)
        #endfor
    return X_all,y_all
```

```
ReadTrainimagesFromDisk = False
if ReadTrainimagesFromDisk:
    Data_Path_train = root_path + 'Dataset/train/*/*.jpg'
    print(Data_Path_train)
    X_lst_train, y_lst_train = DataPreparing(Data_Path_train)


    print('Saving train dataset to {}'.format(root_path + 'Dataset/train'))
    train_data = {}
    train_data["X_lst_train"] = X_lst_train
    train_data["y_lst_train"] = y_lst_train

    # save dataset to pkl File
    with open(root_path + 'Dataset/train1.pkl', "wb") as outfile:
        pickle.dump(train_data, outfile)
else:
    # Retrieve dataset from Saved pkl File
    with open(root_path + 'Dataset/train.pkl', "rb") as file:
        train_data = pickle.load(file)
    X_lst_train = train_data["X_lst_train"]
    y_lst_train = train_data["y_lst_train"]

X_np_train = np.asarray(X_lst_train)
y_np_train = np.asarray(y_lst_train)
```

```
[ ]  # Defining a function for shuffling
     def fun(arr1, arr2):
         assert len(arr1) == len(arr2)
         p = np.random.permutation(len(arr1))
         return arr1[p], arr2[p]

     # Calling this function
     X_np_train, y_np_train = fun(X_np_train, y_np_train)
     print(y_np_train[1:100])
```

Training Loop

In Training Loop, we focus on training and evaluating neural network models for binary classification. Here are the main points:

Implement a **training loop**: a train loop created to goes through the training dataset, passing the input data through the network, calculating the loss, and updating the model's parameters to minimize the loss. This typically involves forward and backward propagation.

Sigmoid function is used as an activation function. Since we have our output layer as logits, means there are in [-inf, +inf] range, we need to apply sigmoid and then round to convert them to 0 or 1.

Choosing a **loss function**: Since this task is a binary classification, therefore the **Binary** cross-entropy loss function is appropriate. It measures the difference between the predicted probability and the actual label for each sample. And also, as part of the task we calculate the **Zero-one** loss too for each model.

Selecting an **optimizer**: Once we calculate the error, we also need to define how the model should react to that feedback. The optimizer determines how the network learns from feedback. Therefore, an optimizer algorithm, such as **Adam**, **SGD**, are picked to update the model's parameters during training. The optimizer adjusts the model's weights per batch based on computed gradients and the learning rate.

learning rate: The learning rate determines the step size for parameter updates. It controls the speed at which the model learns. Higher rates may lead to quick convergence but risk overshooting the optimal solution, while lower rates may result in slower convergence. We implement our Models with three different learning rate 0.005, 0.05 and 0.1

Batch: During training we divide the dataset into **mini-batches** of size 64 to perform forward and backward passes on each batch, and update the parameters accordingly.

Different epochs: To train and validate the neural network models, we used 30 epochs. Each epoch represents a complete pass through the training dataset. Within the epoch loop, iterate over the training dataset in mini-batches. Pass each mini-batch through the network, calculate the loss, and update the model's parameters as before. After each epoch, the model's performance is evaluated on the validation dataset. Iterate over the validation dataset and compute the validation loss and accuracy. Finally, we store the loss and accuracy values for each epoch in separate lists or arrays. By training for multiple epochs, the models have the opportunity to learn from the data and refine their parameters gradually. Additionally, tracking the loss and accuracy metrics for each epoch provides insights into the model's learning progress and performance.

Figure 9. Training Loop phase code

```
def train_loop(model, optimizer, writer, x_train, y_train, x_val, y_val, batch_size, num_epochs, fold_num):
    total_step = x_train.shape[0]
    val_batch = 1
    epoch_loss = []
    epoch_accuracy = []
    for epoch in range(num_epochs):
        # training loop
        model.train()
        correct = 0
        total = 0
        train_loss = 0
        train_zero_one_loss = 0
        for i in range(0, len(x_train), batch_size):
            # slice dataset for batch training, and then convert it to tensor and move it to GPU (cuda)
            batch_x = torch.from_numpy(x_train[slice(i, i + batch_size)]).to(torch.float32).to(device).permute(0,3,1,2)
            batch_y = torch.from_numpy(y_train[slice(i, i + batch_size)]).to(torch.float32).unsqueeze(1).to(device)
            logits = model(batch_x)
            # we have our output layer as logits, means there are in [-inf,+inf] range, we need to apply sigmoid
            y_pred = torch.sigmoid(logits).round().int() # = (torch.sigmoid(logits) >= 0.5).int()
            # loss values
            loss = criterion(logits, batch_y) # we pass logits here as our Loss dunction is BCEWithLogitsLoss
            loss_zero_one = zero_one_loss_fn(y_pred, batch_y)
            train_loss += loss
            train_zero_one_loss += loss_zero_one
            # Zero your gradients for every batch!
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            #print(y_pred)
            total += batch_y.size(0)
            correct += (y_pred == batch_y).sum().item()
            if i % (batch_size*20) == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{total_step}], Zero one Loss:
                {loss_zero_one.item():.4f}, Entropy Loss: {loss.item():.4f}')
        acc_train = 100 * correct / total
        train_loss /= total
        train_zero_one_loss /= total
        print(f'Epoch [{epoch+1}], Accuracy on {x_train.shape[0]} training images: {acc_train} % , Zero one Loss:
        {loss_zero_one.item():.4f}, Entropy Loss: {train_loss.item():.4f}')

        # Validation
        model.eval()
        correct = 0
        total = 0
```

5-fold Cross Validation

And also, **cross-validation** is introduced to estimate the model's performance and address overfitting. The dataset is divided into five subsets or folds. You iterate through each fold, using four folds for training and one-fold for validation. The risk estimate (typically zero-one loss) is computed for each fold using the validation set. This process is repeated for each network architecture and hyperparameter combination, allowing you to compare performance and select the architecture and hyperparameters with the lowest risk estimate.

Figure 10. Cross-Validation phase code

```
# running with 5-fold cross validation
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42, )

writer = SummaryWriter(log_dir="./runs/ML")

fold = 1
for train_index, val_index in skf.split(X_np_train, y_np_train):
    print(f'==== Training the Fold {fold} ====')
    X_train, X_Val = X_np_train[train_index], X_np_train[val_index]
    y_train, y_Val = y_np_train[train_index], y_np_train[val_index]

    # we will perform model train, eval and ... after this for each fold.
    model = train_loop(model, optimizer, writer, X_train, y_train, X_Val, y_Val, batch_size, num_epochs, fold)

    fold += 1
```

Testing Loop

Once each architecture trained, we assess the models' performance on a separate testing set. We pass the test data through the trained models, calculate predicted labels, and compare them with the ground truth labels.

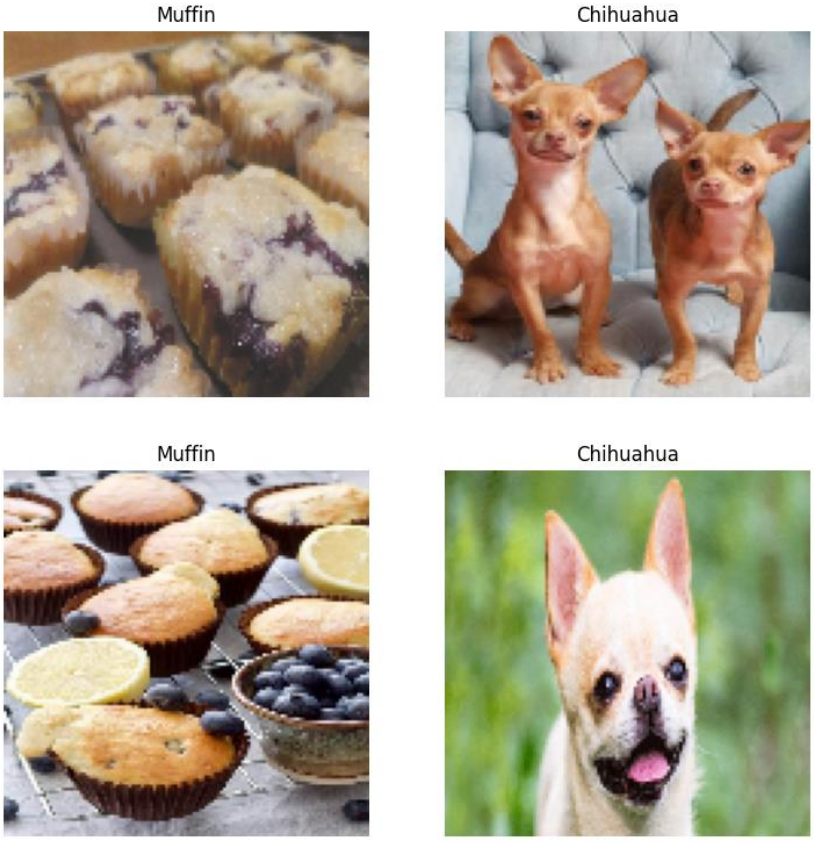
Figure 11. Testing Loop phase code

```
# we can now perform test on real test data
def test_loop(model, x_test, y_test, criterion, batch_size=1):
  total_step = x_test.shape[0]
  # Validation
  model.eval()
  with torch.no_grad():
    correct = 0
    total = 0
    zero_one_loss = 0
    loss = 0
    y_preds = []
    for i in range(0, len(x_test), batch_size):
        # slice dataset for batch training, and then convert it to tensor and move it to GPU (cuda)
        batch_x = torch.from_numpy(x_test[slice(i, i + batch_size)]).to(torch.float32).to(device).permute(0,3,1,2)
        batch_y = torch.from_numpy(y_test[slice(i, i + batch_size)]).to(torch.float32).unsqueeze(1).to(device)

        logits = model(batch_x)

        # we have our output layer as logits, means there are in [-inf,+inf] range, we need to apply sigmoid and then round to convert them to 0/1
        y_pred = torch.sigmoid(logits).round().int() # = (torch.sigmoid(logits) >= 0.5).int()
        y_preds.append(y_pred.item())
        # loss values
        loss += criterion(logits, batch_y) # we pass logits here as our Loss dunction is BCEWithLogitsLoss
        zero_one_loss += zero_one_loss_fn(y_pred, batch_y)

        total += batch_y.size(0)
        correct += (y_pred == batch_y).sum().item()

    acc_val = 100 * correct / total
    loss /= total
    zero_one_loss /= total
    print(f'Accuracy on {x_test.shape[0]} test images: {acc_val} % , Zero one Loss: {zero_one_loss.item():.4f}, Entropy Loss: {loss.item():.4f}')

    return y_preds
```

Plotting some of our train images

We plotted some of train data to check does we labeled right all images.

Figure 12. Random photo checking



CNN Architectures Implementation

Multiple network architectures are trained with, varying hyperparameters like the number of layers, activation functions, and learning rates. This allows for comparison and analysis of their performance.

AlexNet Code Implementation

Table 1 and Figure 13 gives the architecture of the used Convolutional Neural Network. The network contains 5 convolutional layer and 3 fully connected layers. After each convolutional layer, batch normalization and MaxPool are applied. ReLU as an activation function is used in each layer. Finally, before fully connected layer, the input data is reshaped.

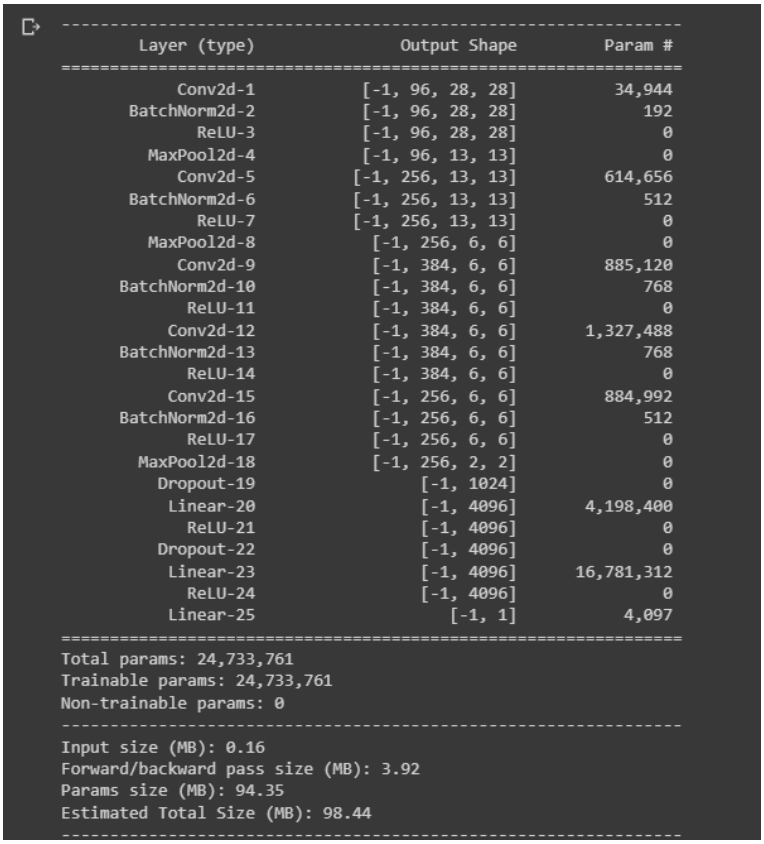
ReLU (Rectified Linear Unit) is a non-linear activation function that introduces non-linearity to the network, allowing it to learn and represent complex relationships in the data. ReLU helps alleviate the vanishing gradient problem, which can occur during backpropagation. By allowing the flow of positive

gradients, ReLU helps gradients propagate more effectively, enabling better learning and training of deeper networks.

Figure 13: AlexNet Code Implementation

```
class AlexNet(nn.Module):
    def __init__(self, num_classes=1):
        super(AlexNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=0),
            nn.BatchNorm2d(96),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 3, stride = 2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 3, stride = 2))
        self.layer3 = nn.Sequential(
            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU())
        self.layer4 = nn.Sequential(
            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU())
        self.layer5 = nn.Sequential(
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 3, stride = 2))
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(1024, 4096),
            nn.ReLU())
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU())
        self.fc2= nn.Sequential(
            nn.Linear(4096, num_classes))

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

Table 1: AlexNet Architecture



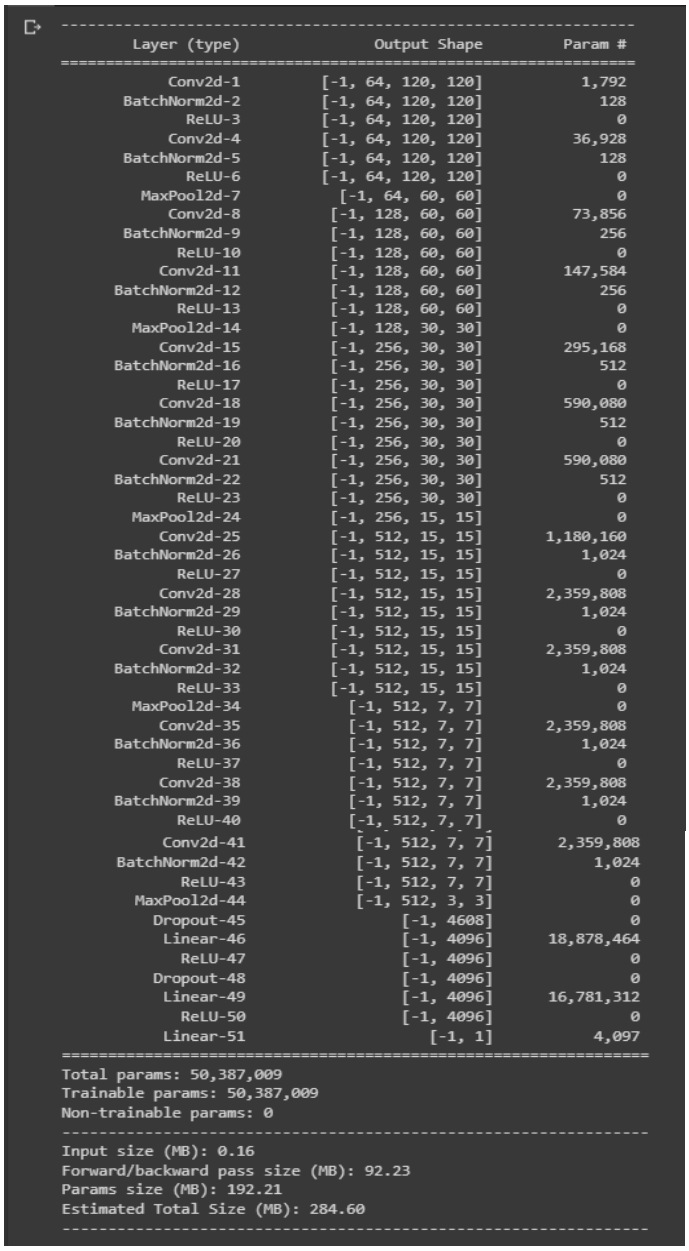
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 96, 28, 28]          34,944
       BatchNorm2d-2          [-1, 96, 28, 28]             192
              ReLU-3          [-1, 96, 28, 28]               0
         MaxPool2d-4          [-1, 96, 13, 13]               0
            Conv2d-5         [-1, 256, 13, 13]         614,656
       BatchNorm2d-6         [-1, 256, 13, 13]             512
              ReLU-7         [-1, 256, 13, 13]               0
         MaxPool2d-8           [-1, 256, 6, 6]               0
            Conv2d-9           [-1, 384, 6, 6]         885,120
      BatchNorm2d-10           [-1, 384, 6, 6]             768
             ReLU-11           [-1, 384, 6, 6]               0
           Conv2d-12           [-1, 384, 6, 6]       1,327,488
      BatchNorm2d-13           [-1, 384, 6, 6]             768
             ReLU-14           [-1, 384, 6, 6]               0
           Conv2d-15           [-1, 256, 6, 6]         884,992
      BatchNorm2d-16           [-1, 256, 6, 6]             512
             ReLU-17           [-1, 256, 6, 6]               0
        MaxPool2d-18           [-1, 256, 2, 2]               0
          Dropout-19                [-1, 1024]               0
           Linear-20                [-1, 4096]       4,198,400
             ReLU-21                [-1, 4096]               0
          Dropout-22                [-1, 4096]               0
           Linear-23                [-1, 4096]      16,781,312
             ReLU-24                [-1, 4096]               0
           Linear-25                   [-1, 1]           4,097
================================================================
Total params: 24,733,761
Trainable params: 24,733,761
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.16
Forward/backward pass size (MB): 3.92
Params size (MB): 94.35
Estimated Total Size (MB): 98.44
----------------------------------------------------------------
```

VGG16 Code Implementation

Figure 13: VGG16 Code Implementation (some part of code)

```
class VGG16(nn.Module):
    def __init__(self, num_classes=1):
        super(VGG16, self).__init__()
        self.convs = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2),

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),

            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2),

            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),

            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),

            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2),

            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),

            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))

        self.fcs = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4608, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes))

    def forward(self, x):
        out = self.convs(x)
        out = out.reshape(out.size(0), -1)
        out = self.fcs(out)
        return out
```

Figure 13: VGG16 Architecture



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 120, 120]           1,792
       BatchNorm2d-2         [-1, 64, 120, 120]             128
              ReLU-3         [-1, 64, 120, 120]               0
            Conv2d-4         [-1, 64, 120, 120]          36,928
       BatchNorm2d-5         [-1, 64, 120, 120]             128
              ReLU-6         [-1, 64, 120, 120]               0
         MaxPool2d-7           [-1, 64, 60, 60]               0
            Conv2d-8          [-1, 128, 60, 60]          73,856
       BatchNorm2d-9          [-1, 128, 60, 60]             256
             ReLU-10          [-1, 128, 60, 60]               0
           Conv2d-11          [-1, 128, 60, 60]         147,584
      BatchNorm2d-12          [-1, 128, 60, 60]             256
             ReLU-13          [-1, 128, 60, 60]               0
        MaxPool2d-14          [-1, 128, 30, 30]               0
           Conv2d-15          [-1, 256, 30, 30]         295,168
      BatchNorm2d-16          [-1, 256, 30, 30]             512
             ReLU-17          [-1, 256, 30, 30]               0
           Conv2d-18          [-1, 256, 30, 30]         590,080
      BatchNorm2d-19          [-1, 256, 30, 30]             512
             ReLU-20          [-1, 256, 30, 30]               0
           Conv2d-21          [-1, 256, 30, 30]         590,080
      BatchNorm2d-22          [-1, 256, 30, 30]             512
             ReLU-23          [-1, 256, 30, 30]               0
        MaxPool2d-24          [-1, 256, 15, 15]               0
           Conv2d-25          [-1, 512, 15, 15]       1,180,160
      BatchNorm2d-26          [-1, 512, 15, 15]           1,024
             ReLU-27          [-1, 512, 15, 15]               0
           Conv2d-28          [-1, 512, 15, 15]       2,359,808
      BatchNorm2d-29          [-1, 512, 15, 15]           1,024
             ReLU-30          [-1, 512, 15, 15]               0
           Conv2d-31          [-1, 512, 15, 15]       2,359,808
      BatchNorm2d-32          [-1, 512, 15, 15]           1,024
             ReLU-33          [-1, 512, 15, 15]               0
        MaxPool2d-34            [-1, 512, 7, 7]               0
           Conv2d-35            [-1, 512, 7, 7]       2,359,808
      BatchNorm2d-36            [-1, 512, 7, 7]           1,024
             ReLU-37            [-1, 512, 7, 7]               0
           Conv2d-38            [-1, 512, 7, 7]       2,359,808
      BatchNorm2d-39            [-1, 512, 7, 7]           1,024
             ReLU-40            [-1, 512, 7, 7]               0
           Conv2d-41            [-1, 512, 7, 7]       2,359,808
      BatchNorm2d-42            [-1, 512, 7, 7]           1,024
             ReLU-43            [-1, 512, 7, 7]               0
        MaxPool2d-44            [-1, 512, 3, 3]               0
          Dropout-45                 [-1, 4608]               0
           Linear-46                 [-1, 4096]      18,878,464
             ReLU-47                 [-1, 4096]               0
          Dropout-48                 [-1, 4096]               0
           Linear-49                 [-1, 4096]      16,781,312
             ReLU-50                 [-1, 4096]               0
           Linear-51                    [-1, 1]           4,097
================================================================
Total params: 50,387,009
Trainable params: 50,387,009
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.16
Forward/backward pass size (MB): 92.23
Params size (MB): 192.21
Estimated Total Size (MB): 284.60
----------------------------------------------------------------
```

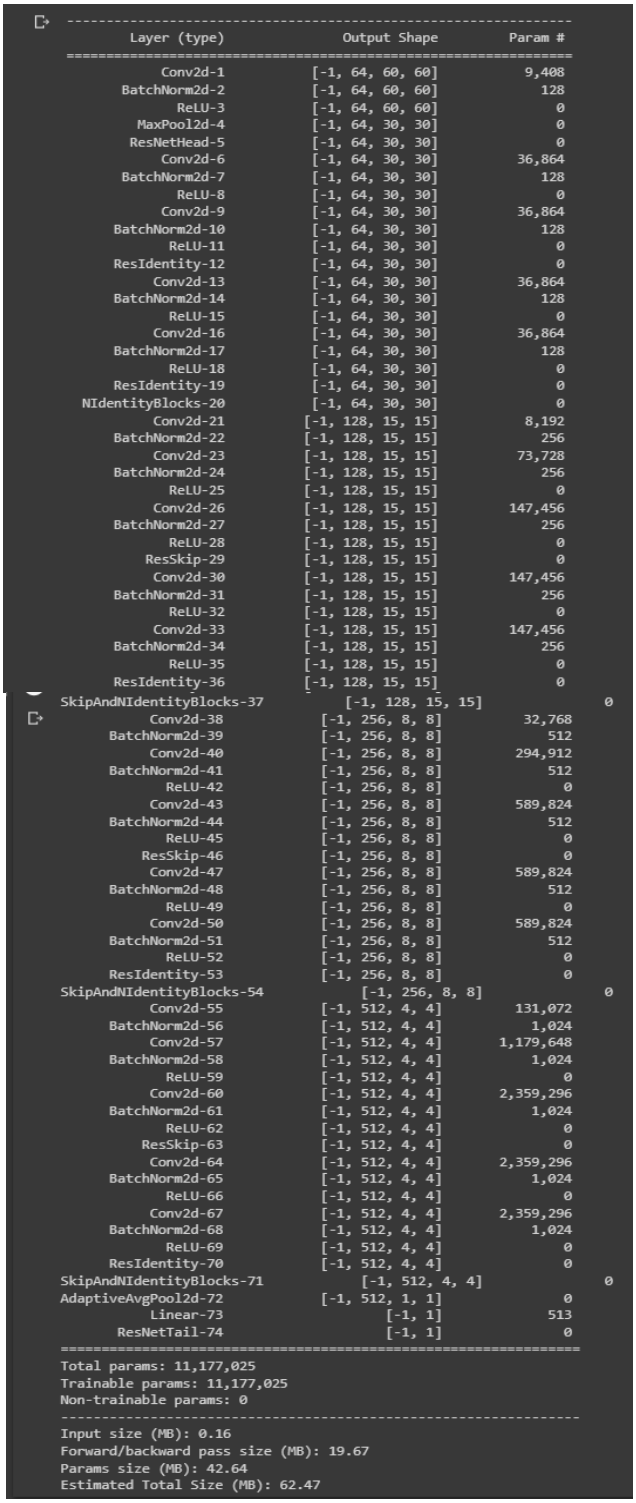
ResNet Code Implementation

Figure 13: ResNet Code Implementation (some part of code)

```
class ResIdentity(nn.Module):
    # identity block with empty skip connection
    def __init__(self, num_inputs):
        super(ResIdentity, self).__init__()
        self.convs = nn.Sequential(
            nn.Conv2d(num_inputs, num_inputs, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(num_inputs),
            nn.ReLU(inplace=True),
            nn.Conv2d(num_inputs, num_inputs, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(num_inputs),
            nn.ReLU(inplace=True))

    def forward(self, x):
        identity = x
        x = self.convs(x)
        x = x + identity
        return x


class ResSkip(nn.Module):
    # skip connection
    def __init__(self, num_inputs, num_outputs): # input channels, number of outputs
        super(ResSkip, self).__init__()
        self.skips = nn.Sequential(
            nn.Conv2d(num_inputs, num_outputs, kernel_size=1, stride=2, padding=0, bias=False),
            nn.BatchNorm2d(num_outputs))

        self.convs = nn.Sequential(
            nn.Conv2d(num_inputs, num_outputs, kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(num_outputs),
            nn.ReLU(inplace=True),
            nn.Conv2d(num_outputs, num_outputs, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(num_outputs))

        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
```

```
class ResNetTail(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(ResNetTail, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.lin = nn.Linear(num_inputs, num_outputs)

    def forward(self, x):
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.lin(x)
        return x


class ResNet(nn.Module):
    def __init__(self, num_inputs, num_outputs, repeat):
        super(ResNet, self).__init__()
        self.res_net = nn.Sequential(
            ResNetHead(num_inputs, 64),
            NIdentityBlocks(64, repeat[0]),
            SkipAndNIdentityBlocks(64, 128, repeat[1]),
            SkipAndNIdentityBlocks(128, 256, repeat[2]),
            SkipAndNIdentityBlocks(256, 512, repeat[3]),
            ResNetTail(512, num_outputs))

    def forward(self, x):
        x = self.res_net(x)
        return x
```

Figure 13: ResNet Architecture



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 60, 60]           9,408
       BatchNorm2d-2           [-1, 64, 60, 60]             128
              ReLU-3           [-1, 64, 60, 60]               0
         MaxPool2d-4           [-1, 64, 30, 30]               0
        ResNetHead-5           [-1, 64, 30, 30]               0
            Conv2d-6           [-1, 64, 30, 30]          36,864
       BatchNorm2d-7           [-1, 64, 30, 30]             128
              ReLU-8           [-1, 64, 30, 30]               0
            Conv2d-9           [-1, 64, 30, 30]          36,864
      BatchNorm2d-10           [-1, 64, 30, 30]             128
             ReLU-11           [-1, 64, 30, 30]               0
      ResIdentity-12           [-1, 64, 30, 30]               0
           Conv2d-13           [-1, 64, 30, 30]          36,864
      BatchNorm2d-14           [-1, 64, 30, 30]             128
             ReLU-15           [-1, 64, 30, 30]               0
           Conv2d-16           [-1, 64, 30, 30]          36,864
      BatchNorm2d-17           [-1, 64, 30, 30]             128
             ReLU-18           [-1, 64, 30, 30]               0
      ResIdentity-19           [-1, 64, 30, 30]               0
  NIdentityBlocks-20           [-1, 64, 30, 30]               0
           Conv2d-21          [-1, 128, 15, 15]           8,192
      BatchNorm2d-22          [-1, 128, 15, 15]             256
           Conv2d-23          [-1, 128, 15, 15]          73,728
      BatchNorm2d-24          [-1, 128, 15, 15]             256
             ReLU-25          [-1, 128, 15, 15]               0
           Conv2d-26          [-1, 128, 15, 15]         147,456
      BatchNorm2d-27          [-1, 128, 15, 15]             256
             ReLU-28          [-1, 128, 15, 15]               0
         ResSkip-29          [-1, 128, 15, 15]               0
           Conv2d-30          [-1, 128, 15, 15]         147,456
      BatchNorm2d-31          [-1, 128, 15, 15]             256
             ReLU-32          [-1, 128, 15, 15]               0
           Conv2d-33          [-1, 128, 15, 15]         147,456
      BatchNorm2d-34          [-1, 128, 15, 15]             256
             ReLU-35          [-1, 128, 15, 15]               0
      ResIdentity-36          [-1, 128, 15, 15]               0
SkipAndNIdentityBlocks-37          [-1, 128, 15, 15]               0
           Conv2d-38            [-1, 256, 8, 8]          32,768
      BatchNorm2d-39            [-1, 256, 8, 8]             512
           Conv2d-40            [-1, 256, 8, 8]         294,912
      BatchNorm2d-41            [-1, 256, 8, 8]             512
             ReLU-42            [-1, 256, 8, 8]               0
           Conv2d-43            [-1, 256, 8, 8]         589,824
      BatchNorm2d-44            [-1, 256, 8, 8]             512
             ReLU-45            [-1, 256, 8, 8]               0
         ResSkip-46            [-1, 256, 8, 8]               0
           Conv2d-47            [-1, 256, 8, 8]         589,824
      BatchNorm2d-48            [-1, 256, 8, 8]             512
             ReLU-49            [-1, 256, 8, 8]               0
           Conv2d-50            [-1, 256, 8, 8]         589,824
      BatchNorm2d-51            [-1, 256, 8, 8]             512
             ReLU-52            [-1, 256, 8, 8]               0
      ResIdentity-53            [-1, 256, 8, 8]               0
SkipAndNIdentityBlocks-54            [-1, 256, 8, 8]               0
           Conv2d-55            [-1, 512, 4, 4]         131,072
      BatchNorm2d-56            [-1, 512, 4, 4]           1,024
           Conv2d-57            [-1, 512, 4, 4]       1,179,648
      BatchNorm2d-58            [-1, 512, 4, 4]           1,024
             ReLU-59            [-1, 512, 4, 4]               0
           Conv2d-60            [-1, 512, 4, 4]       2,359,296
      BatchNorm2d-61            [-1, 512, 4, 4]           1,024
             ReLU-62            [-1, 512, 4, 4]               0
         ResSkip-63            [-1, 512, 4, 4]               0
           Conv2d-64            [-1, 512, 4, 4]       2,359,296
      BatchNorm2d-65            [-1, 512, 4, 4]           1,024
             ReLU-66            [-1, 512, 4, 4]               0
           Conv2d-67            [-1, 512, 4, 4]       2,359,296
      BatchNorm2d-68            [-1, 512, 4, 4]           1,024
             ReLU-69            [-1, 512, 4, 4]               0
      ResIdentity-70            [-1, 512, 4, 4]               0
SkipAndNIdentityBlocks-71            [-1, 512, 4, 4]               0
AdaptiveAvgPool2d-72            [-1, 512, 1, 1]               0
           Linear-73                  [-1, 1]             513
       ResNetTail-74                  [-1, 1]               0
================================================================
Total params: 11,177,025
Trainable params: 11,177,025
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.16
Forward/backward pass size (MB): 19.67
Params size (MB): 42.64
Estimated Total Size (MB): 62.47
```

Experimental Results

This step focuses on analyzing and comparing the performance of different network architectures and hyperparameters and drawing meaningful conclusions from the results obtained through cross-validation.

After conducting cross-validation and obtaining the risk estimates for each network architecture and hyperparameter combination, we perform a thorough analysis. The performance metrics (such as accuracy, loss) across the different architectures and hyperparameters are compared. We Identify patterns, trends, and significant differences in the models' performance.

We used **AlexNet, VGG16 and ResNet** architectures for classifying Muffin vs Chihuahua, with 30 epochs and 5 cross-validations. We used TensorBoard for plotting our results over epochs. Since our loss is Binary cross entropy and Zero-one loss, therefore we plot them for training and validation data together, then we plot accuracy for both train and validation data.

AlexNet Implementation Result

5-Fold Cross-Validation and Computing risk estimates

Since the results of a k-fold cross-validation is often summarized with the **Mean** of the model scores, therefore we calculated mean of all five folds. And also, it is also good practice to include a measure of the variance of the skill scores, such as the standard deviation or standard error.

The following 6 tables represent the **training** results of **ResNet** architectures using the SGD and Adam optimizer with different learning rates (0.005, 0.05, and 0.1) through 5-fold cross-validation. The tables show the performance metrics for both the training and validation sets in each fold. Finally, the accuracy and the **risk estimates (Zero-one Loss)** of 5-fold cross validation computed for all scenarios by using mean of all folds.

*Table 3. **AlexNet**, **Adam** Optimizer with Learning **rate 0.005** with 5-fold cross validation*

	Train			**Validation**		
Fold	Accuracy	Zero-one Loss	Entropy Loss	Accuracy	Zero-one Loss	Entropy Loss
1	85.634	0.002	0.005	73.318	0.267	1.941
2	98.279	0	0.001	88.981	0.11	0.849
3	98.122	0	0.001	90.044	0.1	0.43
4	99.067	0	0	94.249	0.058	0.283
5	99.890	0	0	98.941	0.016	0.062
Mean	96.198	0.0004	0.0014	89.106	0.110	0.713
St.deviation	5.9475	0.00089	0.002	9.660	0.095	0.744

*Table 4. AlexNet, **Adam** Optimizer with **Learning rate 0.05** with 5-fold cross validation*

	Train			**Validation**		
Fold	Accuracy	Zero-one Loss	Entropy Loss	Accuracy	Zero-one Loss	Entropy Loss
1	53.719	0.007	0.012	53.227	0.468	0.691
2	54.917	0.007	0.012	55.128	0.449	0.688
3	54.240	0.007	0.011	57.491	0.425	0.663
4	53.802	0.007	0.011	54.123	0.459	0.690
5	54.079	0.007	0.011	54.017	0.460	0.691
Mean	54.1514	0.007	0.0114	54.7972	0.452	0.684
St.deviation	0.476	0.004	0.00054	1.6503	0.016	0.012

*Table 5. AlexNet, **Adam** Optimizer with Learning rate 0.1 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero-one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	53.46	0.007	0.012	53.116	0.469	0.863
2	52.609	0.008	0.011	54.065	0.459	0.690
3	54.005	0.007	0.011	54.065	0.459	0.690
4	53.147	0.007	0.012	54.123	0.459	0.691
5	54.031	0.007	0.011	54.017	0.460	0.691
Mean	53.450	0.0072	0.0114	53.877	0.461	0.725
St.deviation	0.6009	0.00044	0.00054	0.4271	0.0043	0.0771

*Table 6. AlexNet, **SGD** Optimizer with Learning rate 0.005 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	99.347	0	0	82.00	0.18	0.857
2	100	0	0	98.721	0.013	0.033
3	99.993	0	0	98.655	0.013	0.045
4	100	0	0	99.778	0.002	0.008
5	100	0	0	99.425	0.006	0.019
Mean	99.868	0	0	95.715	0.042	0.192
St.deviation	0.2912	0	0	7.6820	0.0768	0.3717

*Table 7. AlexNet, **SGD** Optimizer with Learning rate 0.05 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	91.262	0.001	0.003	64.764	0.352	0.766
2	92.226	0.001	0.003	73.759	0.262	0.673
3	93.096	0.001	0.003	69.674	0.303	0.770
4	93.218	0.001	0.003	74.363	0.256	0.677
5	93.056	0.001	0.003	84.625	0.154	0.366
Mean	92.571	0.001	0.003	73.437	0.265	0.650
St.deviation	0.8309	0	0	7.3401	0.0731	0.1656

*Table 8. AlexNet, **SGD** Optimizer with Learning **rate 0.1** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	85.764	0.002	0.005	72.529	0.275	0.571
2	86.022	0.002	0.005	58.628	0.414	0.715
3	83.925	0.003	0.006	65.115	0.349	0.606
4	85.158	0.002	0.006	78.04	0.220	0.563
5	84.633	0.002	0.006	76.393	0.236	0.478
Mean	85.100	0.0022	0.0056	70.141	0.298	0.586
St.deviation	0.8504	0.00044	0.00054	8.1393	0.0814	0.0858

Risk Estimates of 5-fold cross-validation:

Finally, by combining the 6th row of above tables (Table 3- Table 8) we reach the following table that shows the risk estimates of 5-fold cross validation computed for all scenarios by using **Mean** of all folds.

Table 9. Risk Estimates of 5-fold cross-validation

			Train -Mean of folds			**Validation**- Mean of folds		
Model	**Optimizer**	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
AlexNet	**Adam**	0.005	96.198	0.0004	0.0014	89.106	0.110	0.713
		0.05	54.151	0.007	0.0114	54.7972	0.452	0.684
		0.1	53.450	0.0072	0.0114	53.877	0.461	0.725
	SGD	0.005	99.868	0	0	95.715	0.042	0.192
		0.05	92.571	0.001	0.003	73.437	0.265	0.650
		0.1	85.100	0.0022	0.0056	70.141	0.298	0.586

Description

a. For the **SGD** optimizer, the learning rate of 0.005 and 0.05 consistently result in higher accuracy and lower loss metrics on the training datasets, but the learning rate 0.05 for validation datasets does not perform well.

b. The learning rate of 0.1 tends to result in lower accuracy and higher loss metrics on both datasets for both optimizers.

c. The SGD optimizer generally performs well on the training dataset, achieving high accuracy and low loss metrics across all learning rates. But on the training dataset for the learning rate of 0.05 and 0.1 does not perform well.

d. For the **Adam** optimizer, the learning rate of 0.005 result in high accuracy and low loss metrics on both the training and validation datasets. However, the performance on the learning rate of 0.05 and 0.1 is not appropriate, which results in lower accuracy and higher loss metrics.

Model Recommendations: Based on the risk estimates, the SGD optimizer with a learning rate of 0.005 seems to be the most suitable choice for the AlexNet architecture. These configurations consistently achieve higher accuracy and lower loss metrics on both the training and validation datasets.

Accuracy:

The following charts display the Accuracy of AlexNet model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

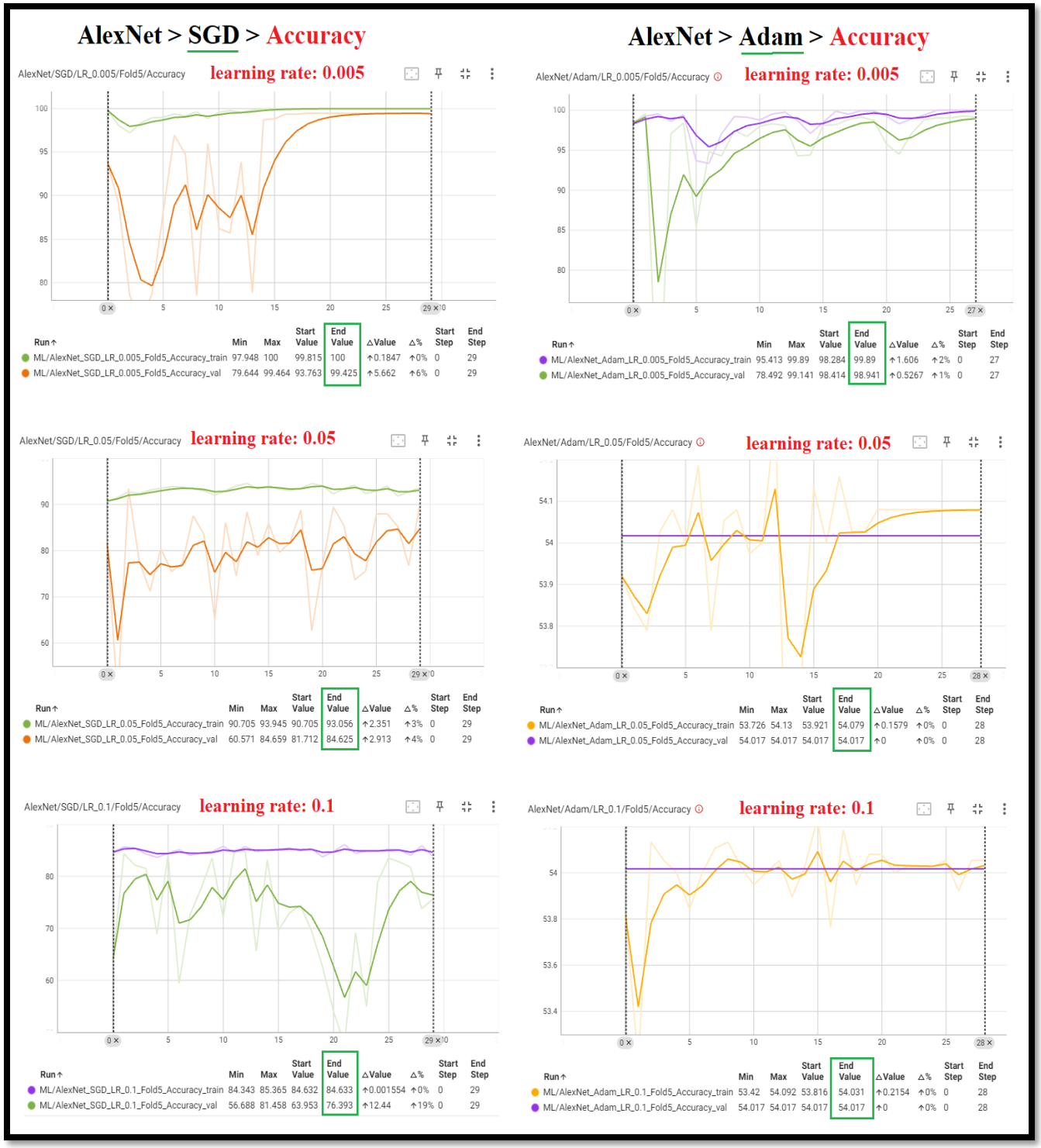


Figure 14: Accuracy of AlexNet model for different Epochs in train and validation data

Zero-One loss:

The following charts display the Zero-One loss of AlexNet model for the **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

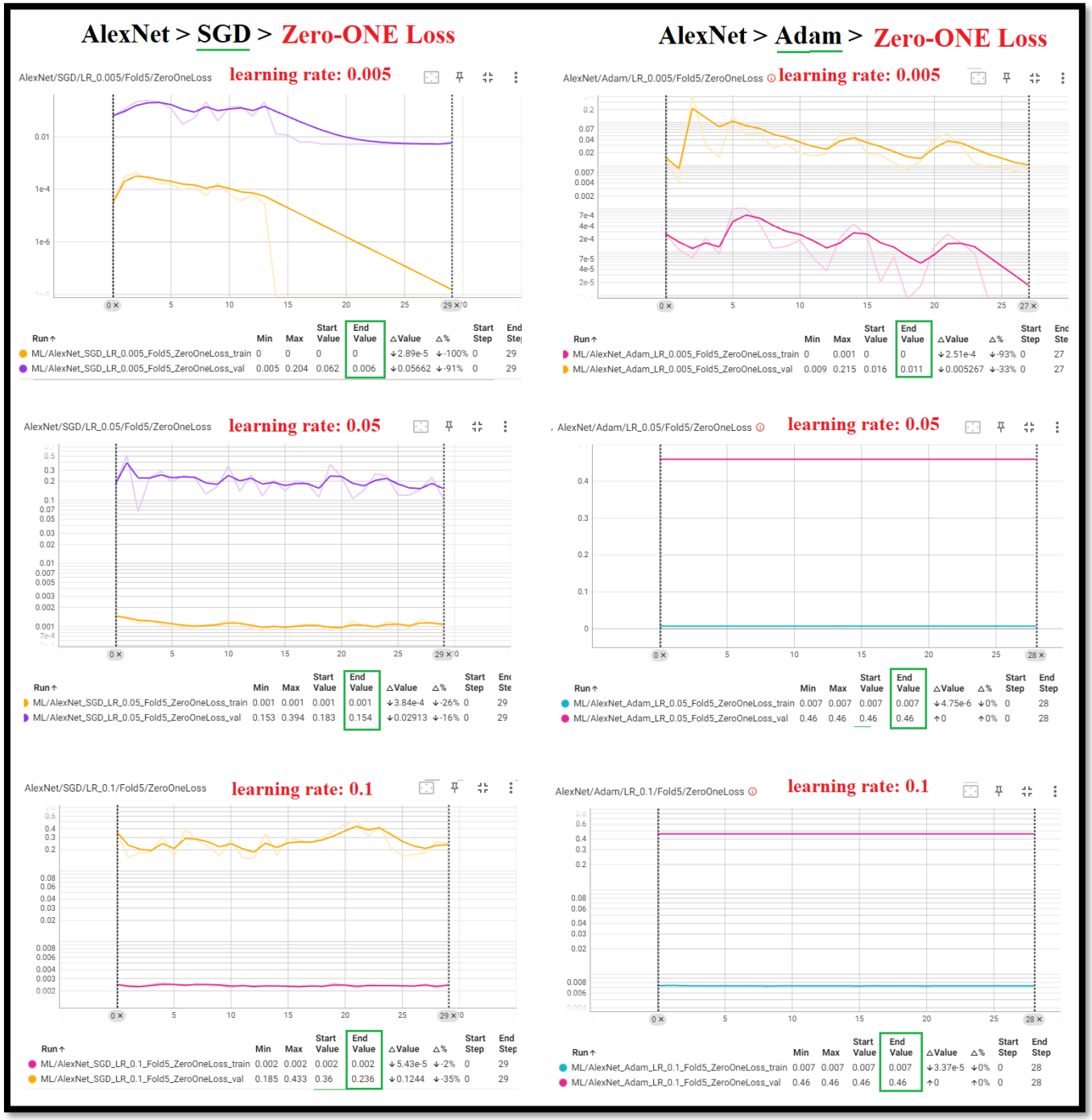


Figure 15: Zero-One loss of AlexNet model for different Epochs in train and validation data

Binary cross entropy loss (BCE):

The following charts display the Binary cross entropy loss (BCE) of AlexNet model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

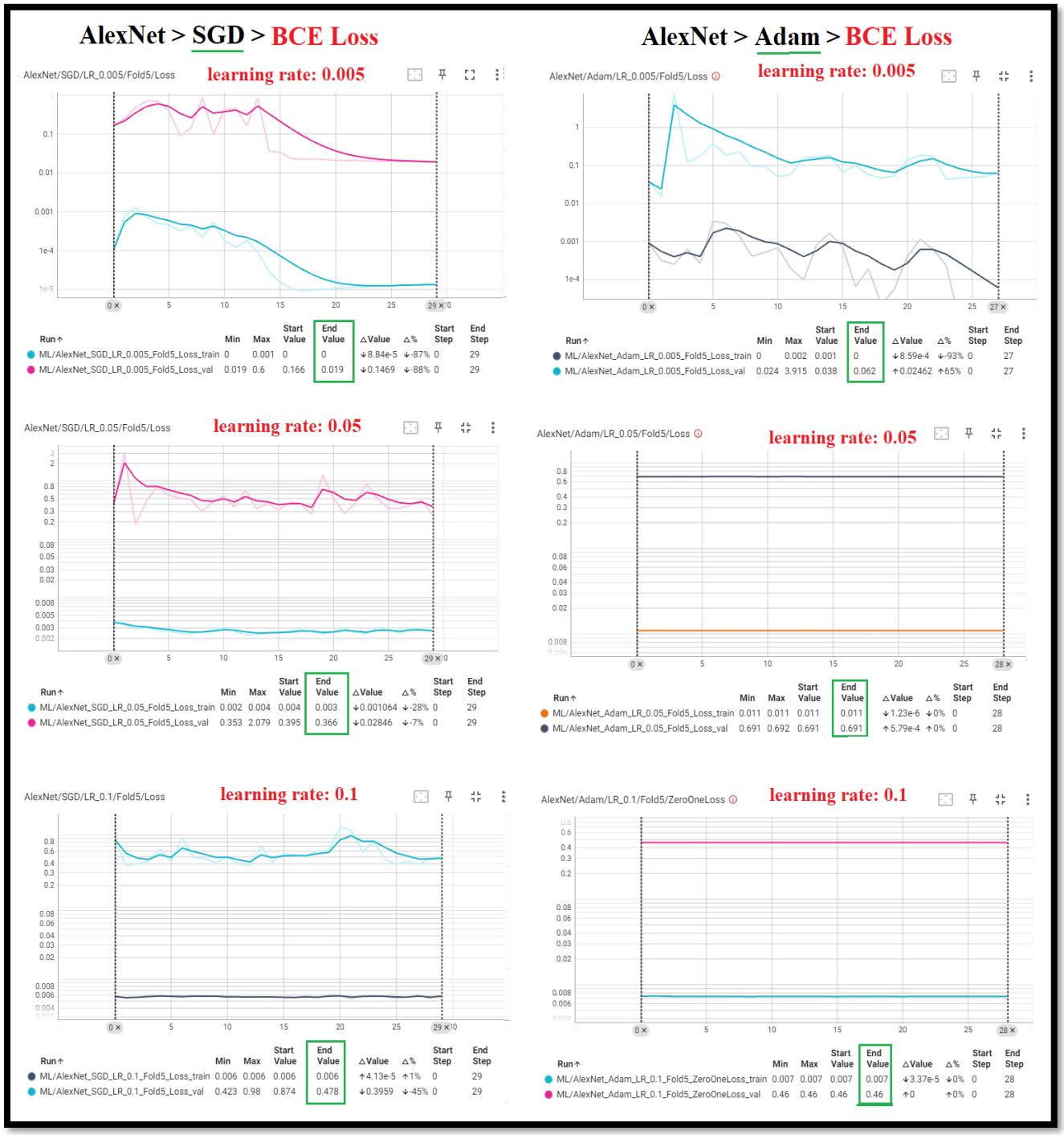


Figure 16: Binary cross entropy loss of AlexNet model for different Epochs in train and validation data

Quick review of all above figures in one table:

The following table displays quick review of three above figures for Accuracy, Binary cross-entropy (BCE) loss and Zero-one Loss for **Last Fold** (5th fold) of train and validation datasets.

*Table 9. **AlexNet** Architecture, both Optimizer and three Learning for **Last fold** of cross-validation*

		Train- Last fold (5th fold)			**Validation** - Last fold (5th fold)		
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	99.89	0	0	98.941	0.016	0.062
	0.05	54.079	0.007	0.011	54.017	0.460	0.691
	0.1	54.031	0.007	0.011	54.017	0.460	0.691
SGD	0.005	100	0	0	99.425	0.006	0.019
	0.05	93.056	0.001	0.003	84.625	0.154	0.366
	0.1	84.633	0.002	0.006	76.393	0.236	0.478

Description:

In **Adam** Optimizer with Learning Rate 0.005 the model achieves high accuracy on the training set (99.89%) for the last fold with zero-one loss of 0, indicating no misclassifications. The entropy loss is 0, indicating that the model's predictions are close to the true label probabilities. On the validation set, the model achieves an accuracy of 98.941% with a low zero-one loss of 0.016 and a relatively low entropy loss of 0.062.

While for both learning rates of 0.05 and 0.1, the model achieves much lower accuracy on the training set around 54%. This indicates that the model is misclassifying a significant portion of the training data. On the validation set, the accuracy remains similar to the training set (around 54%) with higher zero-one loss of 0.460 and entropy loss of 0.691. This suggests that the model is not generalizing well to unseen data and is performing poorly on the validation set.

In **SGD** Optimizer with Learning Rate 0.005 the model achieves perfect accuracy on the training set (100%) for the last fold, with zero-one loss of 0, indicating no misclassifications. On the validation set, the model achieves an accuracy of 99.425% with a very low zero-one loss of 0.006 and entropy loss of 0.019. While for both learning rates of 0.05 and 0.1, like Adam Optimizer, the model's accuracy on the validation set drops to around 84% and 76%, respectively, with zero-one losses of 0.154 and 0.236. This suggests that the model is not generalizing well to unseen data and is performing poorly on the validation set.

Overall, the SGD optimizer with a learning rate of 0.005 shows the best performance, achieving high accuracy and low losses on both the training and validation sets. For both optimizers, a learning rate of 0.05 and 0.1 leads to significantly poorer performance, with lower accuracy and higher losses on both sets.

Test Results of AlexNet model

Confusion matrix:

The following charts display the result of the **confusion matrix** of the **test** dataset for **AlexNet** model in different learning rate and Optimizers.

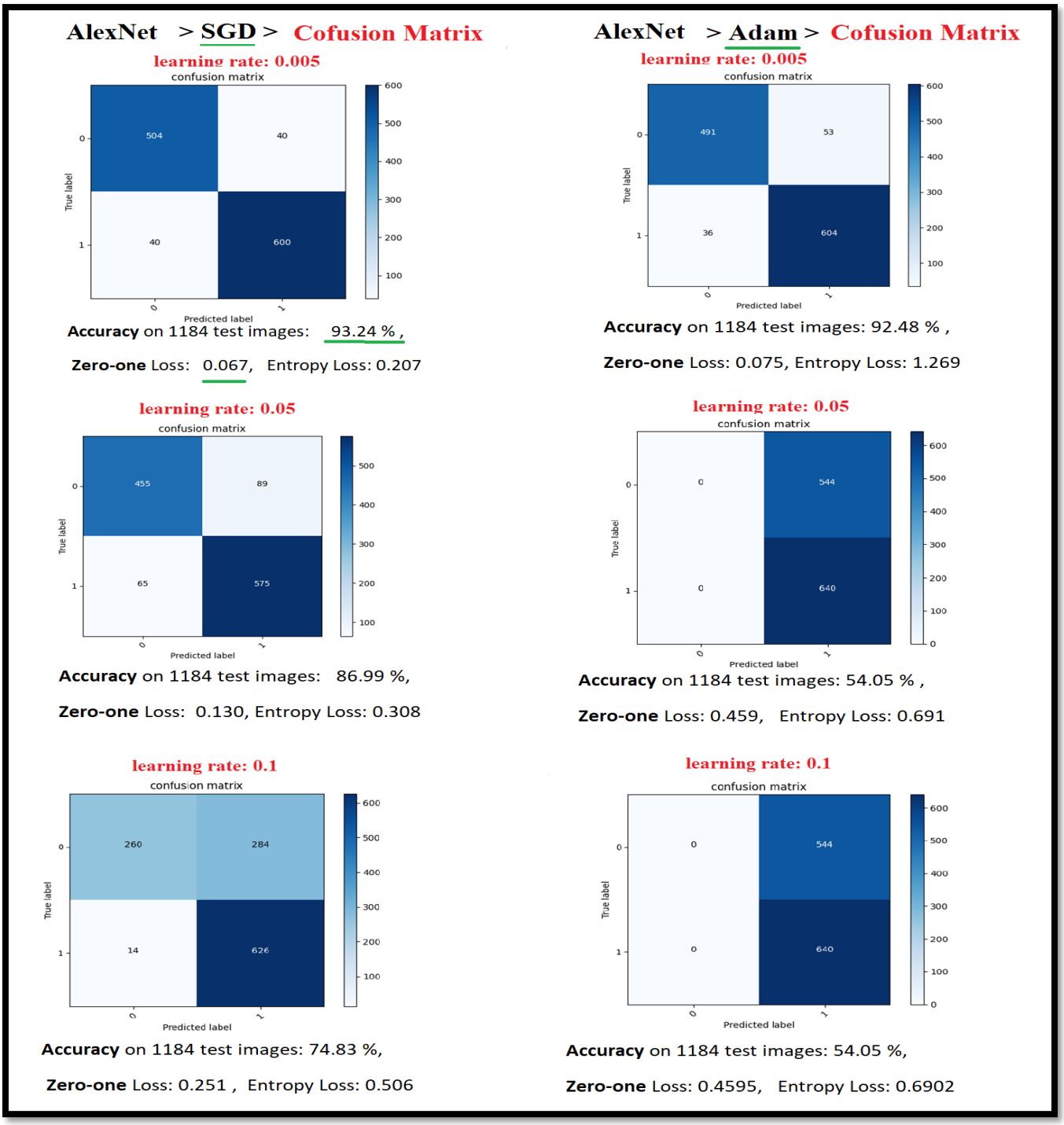


*Figure 17. Confusion matrix of AlexNet model for **Test** data for any Optimizer and Learning rate*

*Table 10. Test result of AlexNet Architecture, both Optimizer and three Learning, **Test** set (1184 images)*

		Test set			Muffin = 0		Chihuahua =1	
Optimizer	**Learning Rate**	Accuracy	Zero-one Loss	Entropy Loss	Num of Correct	Num of Incorrect	Num of Correct	Num of Incorrect
Adam	0.005	92.48	0.075	0.269	491	53	604	36
	0.05	54.05	0.459	0.691	0	544	640	0
	0.1	54.05	0.459	0.690	0	544	640	0
SGD	0.005	93.24	0.067	0.207	504	40	600	40
	0.05	86.99	0.130	0.308	455	89	575	65
	0.1	74.83	0.251	0.506	260	284	626	14

Description:

The AlexNet Architecture with **Adam** optimizer and learning rate of 0.005 shows good performance, achieving the high accuracy (92.48%) and the low zero-one loss (0.075). For Adam optimizer with learning rates of 0.05 and 0.1, the model performs extremely poorly, not classifying any samples correctly, which indicates a severe problem in the model's performance for these learning rates.

The AlexNet Architecture with **SGD** optimizer and learning rate of 0.005 performs the best, achieving a highest accuracy (93.24%) and lowest zero-one loss (0.067). A low zero-one loss value (in this case, 0.067) indicates that only a small percentage of samples are misclassified, further confirming the model's effectiveness. But as the learning rate increases for SGD optimizer (0.05 and 0.1), the model's performance degrades, leading to lower accuracy and higher zero-one loss. The model struggles to correctly classify a significant number of samples for these learning rates. For example, The SGD optimizer with a learning rate of 0.1 misclassify half of Muffin images, while for Chihuahua it performs well.

Summary of **AlexNet** Architecture:

The following table shows the result of model for all Train, Validation and Test set based on the optimizer type, learning rate and etc. And **risk estimates** of 5-fold cross validation computed for all scenarios.

*Table 11. **AlexNet** Architecture, both Optimizer and three Learning, for all Train, Validation and Test set*

		Train-mean of folds			**Validation**- mean of folds			**Test**		
Optimizer	Learning Rate	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	96.198	0.0004	0.0014	89.106	0.110	0.713	92.48	0.075	0.269
	0.05	54.151	0.007	0.0114	54.7972	0.452	0.684	54.05	0.459	0.691
	0.1	53.450	0.0072	0.0114	53.877	0.461	0.725	54.05	0.459	0.690
SGD	0.005	99.868	0	0	95.715	0.042	0.192	93.24	0.067	0.207
	0.05	92.571	0.001	0.003	73.437	0.265	0.650	86.99	0.130	0.308
	0.1	85.100	0.0022	0.0056	70.141	0.298	0.586	74.83	0.251	0.506

The result shows that not only in train and validation but also in test set AlexNet Architecture with **SGD** optimizer and learning rate of 0.005 performs better than other scenarios.

Overall, The AlexNet model with SGD optimizer and a learning rate of 0.005 performs the best. Means that it has learned to generalize well from the training data, making accurate predictions on unseen samples from the test set.

VGG16 Implementation Result

5-Fold Cross-Validation and Computing risk estimates

The following tables represent the results of training VGG16 architectures using the SGD and Adam optimizer with different learning rates (0.005, 0.05, and 0.1) through 5-fold cross-validation. The tables show the performance metrics for both the training and validation sets in each fold. Finally, the accuracy and the **risk estimates (Zero-one Loss)** of 5-fold cross validation computed for all scenarios by using mean of all folds.

*Table 12. VGG16 Architecture, **Adam** Optimizer with Learning **rate 0.005** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	71.706	0.004	0.011	74.613	0.254	0.509
2	91.747	0.001	0.004	85.301	0.191	0.350
3	93.594	0.001	0.003	90.029	0.1	0.257
4	97.501	0	0.001	93.101	0.069	0.233
5	98.441	0.001	0.001	94.361	0.354	0.048
Mean	90.597	0.0014	0.004	87.481	0.193	0.279
St.deviation	10.912	0.00151	0.00412	7.9960	0.1158	0.1687

*Table 13. VGG16 Architecture, **Adam** Optimizer with Learning **rate 0.05** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	54.047	0.007	0.012	54.050	0.460	0.690
2	53.998	0.007	0.011	54.064	0.459	0.698
3	54.054	0.007	0.011	54.063	0.459	0.692
4	54.028	0.007	0.011	54.123	0.459	0.690
5	53.949	0.007	0.011	54.017	0.460	0.690
Mean	54.015	0.007	0.011	54.063	0.459	0.692
St.deviation	0.04289	9.69E-19	0.000447	0.03835	0.000547	0.00346

*Table 14. VGG16 Architecture, **Adam** Optimizer with Learning rate 0.1 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	53.238	0.007	0.011	53.88	0.461	1.739
2	54.057	0.007	0.011	53.537	0.465	1.451
3	54.057	0.007	0.011	53.960	0.460	0.777
4	53.977	0.007	0.011	54.123	0.459	0.689
5	54.067	0.007	0.011	54.017	0.460	0.689
Mean	53.879	0.007	0.011	53.903	0.461	1.069
St.deviation	0.36027	9.69E-19	0	0.22311	0.002345	0.4921

*Table 15. VGG16 Architecture, **SGD** Optimizer with Learning rate 0.**005** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	99.641	0	0	95.085	0.049	0.193
2	99.999	0	0	98.403	0.016	0.034
3	100	0	0	99.109	0.009	0.02
4	100	0	0	99.566	0.004	0.014
5	100	0	0	99.122	0.009	0.029
Mean	99.928	0	0	98.257	0.0174	0.058
St.deviation	0.1604	0	0	1.8214	0.01817	0.0758

*Table 16. VGG16 Architecture, **SGD** Optimizer with Learning rate 0.05 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	91.832	0.001	0.003	74.908	0.251	0.531
2	93.750	0.001	0.003	81.865	0.181	0.524
3	93.642	0.001	0.003	84.554	0.154	0.369
4	93.937	0.001	0.002	67.192	0.328	0.965
5	93.665	0.001	0.002	83.165	0.168	0.530
Mean	93.365	0.001	0.0026	78.336	0.2164	0.5838
St.deviation	0.8649	0	0.00054	7.2521	0.0727	0.2240

*Table 17. VGG16 Architecture, **SGD** Optimizer with Learning rate 0.1 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	85.085	0.002	0.006	71.120	0.289	0.757
2	85.569	0.002	0.006	62.154	0.378	3.78
3	84.701	0.002	0.006	62.327	0.377	1.109
4	83.025	0.003	0.006	75.343	0.463	0.532
5	84.353	0.002	0.006	68.407	0.316	1.417
Mean	84.546	0.0022	0.006	67.870	0.364	1.519
St.deviation	0.963	0.000	0.000	5.703	0.067	1.308

Risk Estimates of 5-fold cross-validation:

Finally, by combining the 6th row of above tables (Table 12- Table 17) we reach the following table that shows the risk estimates of 5-fold cross validation computed for all scenarios by using **Mean** of all folds.

Table 9. Risk Estimates of 5-fold cross-validation

			Train-Mean of folds			**Validation**- Mean of folds		
Model	**Optimizer**	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
VGG16	**Adam**	0.005	90.597	0.0014	0.004	87.481	0.193	0.279
		0.05	54.015	0.007	0.011	54.063	0.459	0.692
		0.1	53.879	0.007	0.011	53.903	0.461	1.069
	SGD	0.005	99.928	0	0	98.257	0.0174	0.058
		0.05	93.365	0.001	0.0026	78.336	0.2164	0.583
		0.1	84.546	0.0022	0.006	67.870	0.364	1.519

Description:

a. The VGG16 Architecture with **Adam** optimizer and learning rate of 0.005 shows good performance, achieving a relatively high accuracy (90.60%) and low loss metrics on both the training and validation datasets. The learning rates of 0.05 and 0.1 result in significantly lower accuracy and higher loss metrics on both datasets. These learning rates do not seem to be optimal for this model and dataset.

b. The VGG16 Architecture with **SGD** optimizer and learning rate of 0.005 shows excellent performance, achieving almost perfect accuracy (99.93%) on the training dataset and high accuracy (98.26%) on the validation dataset. The zero-one loss is extremely low for both datasets, indicating very few misclassifications. The learning rate of 0.05 also performs well, with high accuracy and relatively low loss metrics on the train dataset, but significantly lower accuracy and higher loss metrics on the validation dataset. The learning rate of 0.1, however, results in a noticeable drop in performance, with lower accuracy and higher loss metrics on both datasets.

c. The learning rate of 0.1 should be avoided from both optimizers as it results in decreased performance.

Model Recommendations:

For the VGG16 architecture with the **SGD** optimizer, the learning rate of 0.005 is viable options, as it demonstrates excellent performance on both training and validation datasets.

For the VGG16 architecture with the **Adam** optimizer, the learning rate of 0.005 appears to be the suitable choice, providing a good trade-off between training and validation performance.

Overall, the table offers valuable insights into the model's performance under different optimizer and learning rate settings. The risk estimates allow for a comparative analysis of different scenarios, helping to identify the best-performing configuration for the VGG16 model on this specific task.

Accuracy:

The following charts display the Accuracy of VGG16 model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

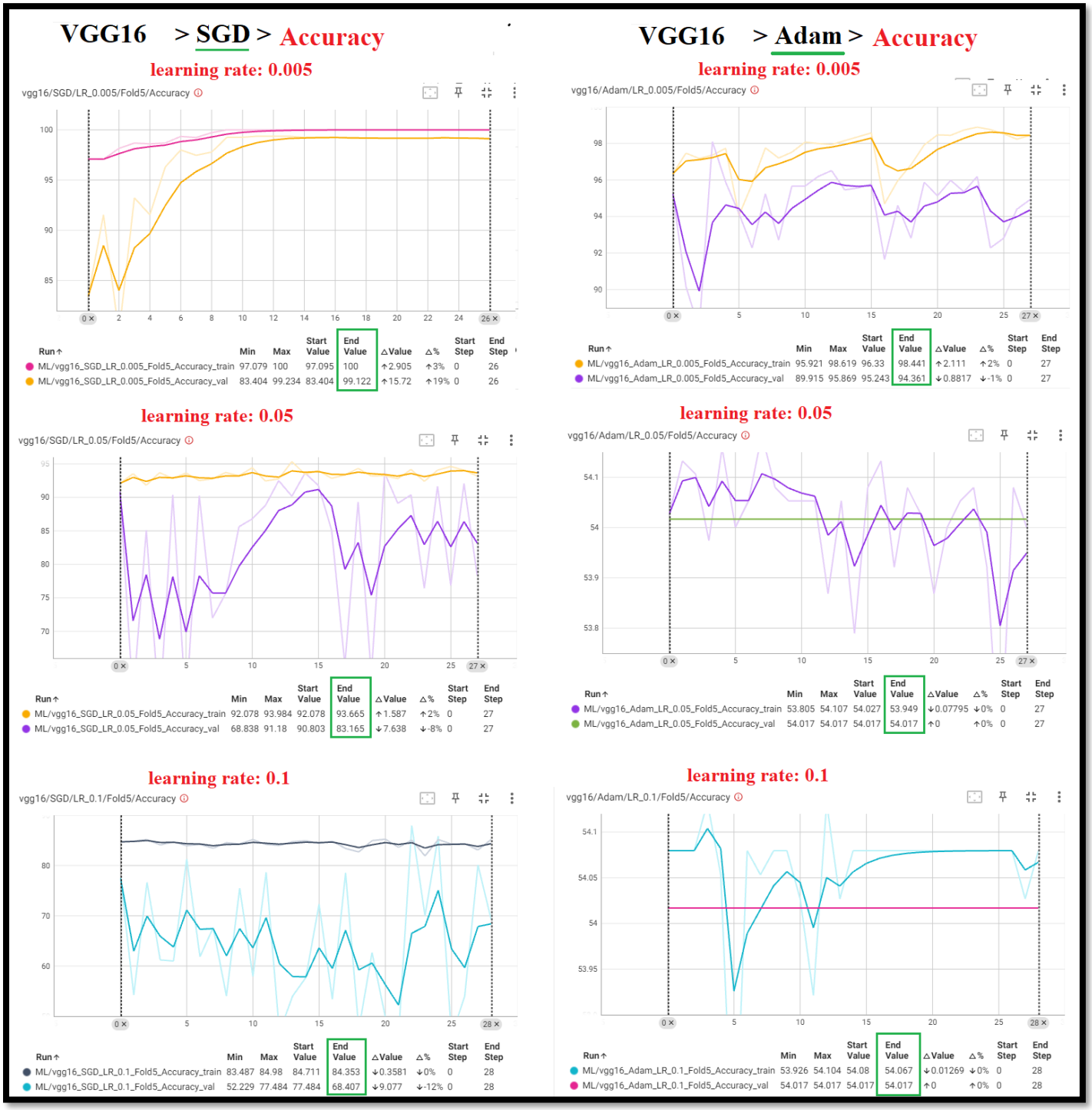


Figure 18: Accuracy of VGG16 model for different Epochs in train and validation data

Zero-One loss:

The following charts display the Zero-One loss of VGG16 model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

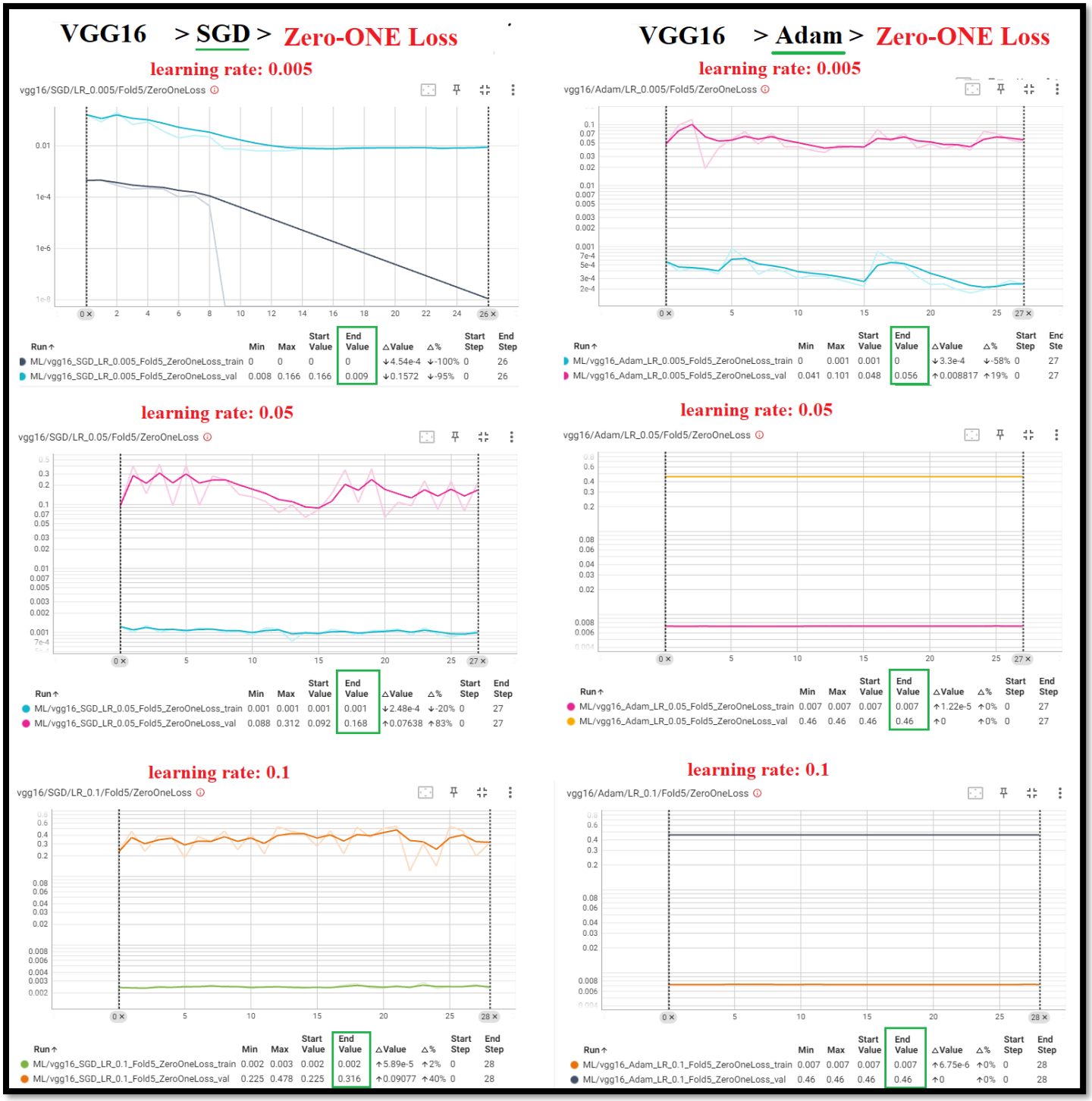


Figure 19: Zero-One loss of VGG16 model for different Epochs in train and validation data

Binary cross entropy loss (BCE):

The following charts display the Binary cross entropy loss (BCE) of AlexNet model for Last fold (5th fold) in different epochs for different leaning rate and Optimizer.

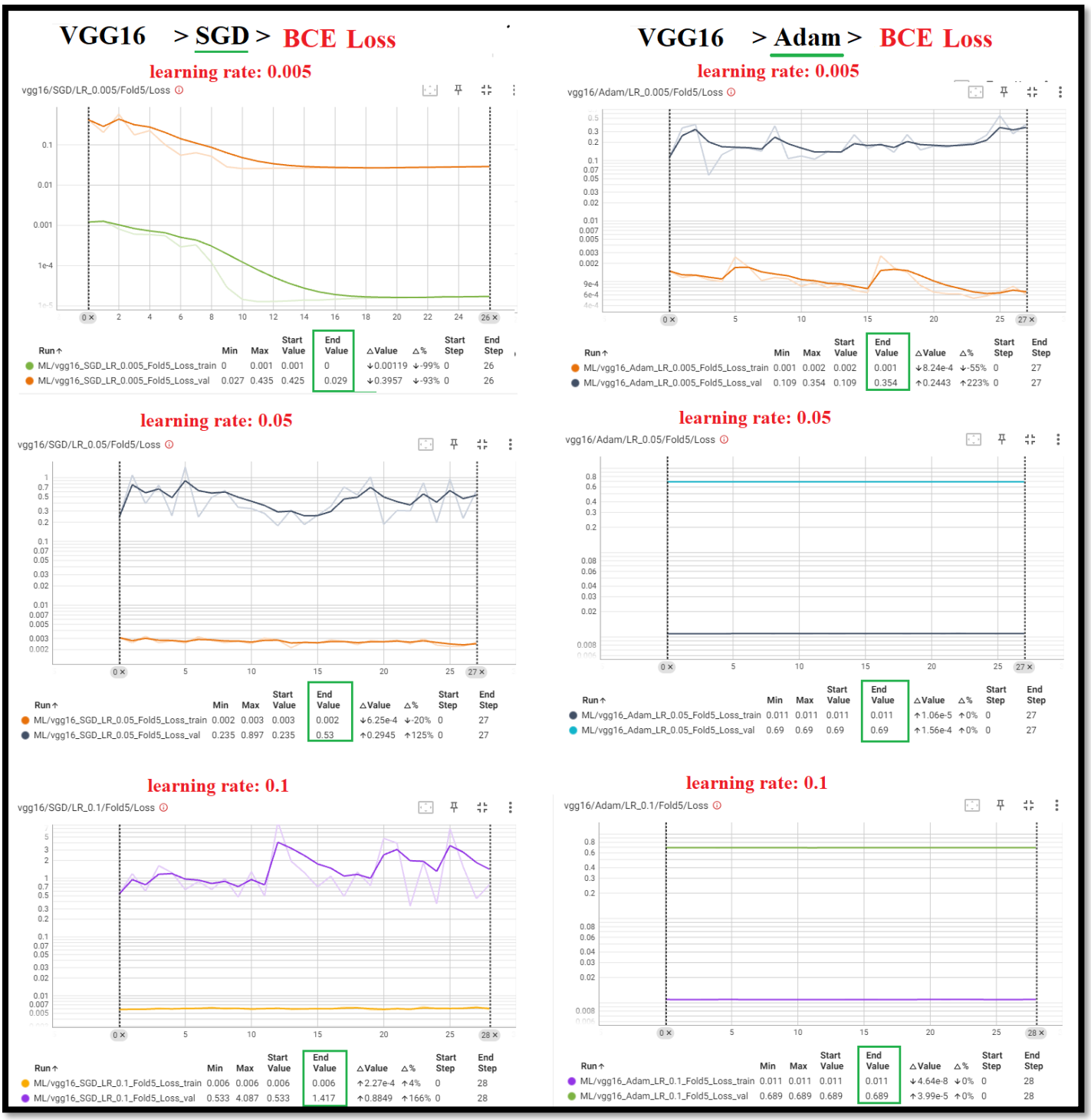


Figure 20: Binary cross entropy loss of VGG16 model, for each fold, for Adam and lr: 0.1

Quick review of all above figures in one table:

The following table displays quick review of all above figures for Accuracy, Binary cross-entropy (BCE) loss and Zero-one Loss for **Last Fold (5th fold**) of train and validation datasets.

*Table 18. **VGG16** Architecture, both Optimizer and three Learning for **Last fold** of cross-validation*

		Train- Last fold			**Validation** - Last fold		
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	98.441	0.001	0.001	94.361	0.354	0.048
	0.05	53.949	0.007	0.011	54.017	0.460	0.690
	0.1	54.067	0.007	0.011	54.017	0.460	0.689
SGD	0.005	100	0	0	99.122	0.009	0.029
	0.05	93.665	0.001	0.002	83.165	0.168	0.530
	0.1	84.353	0.002	0.006	68.407	0.316	1.417

Test dataset Result of VGG16 model:

Confusion matrix:

The following charts display the result of confusion matrix of **test** dataset for **VGG16** model in different leaning rate and Optimizers.

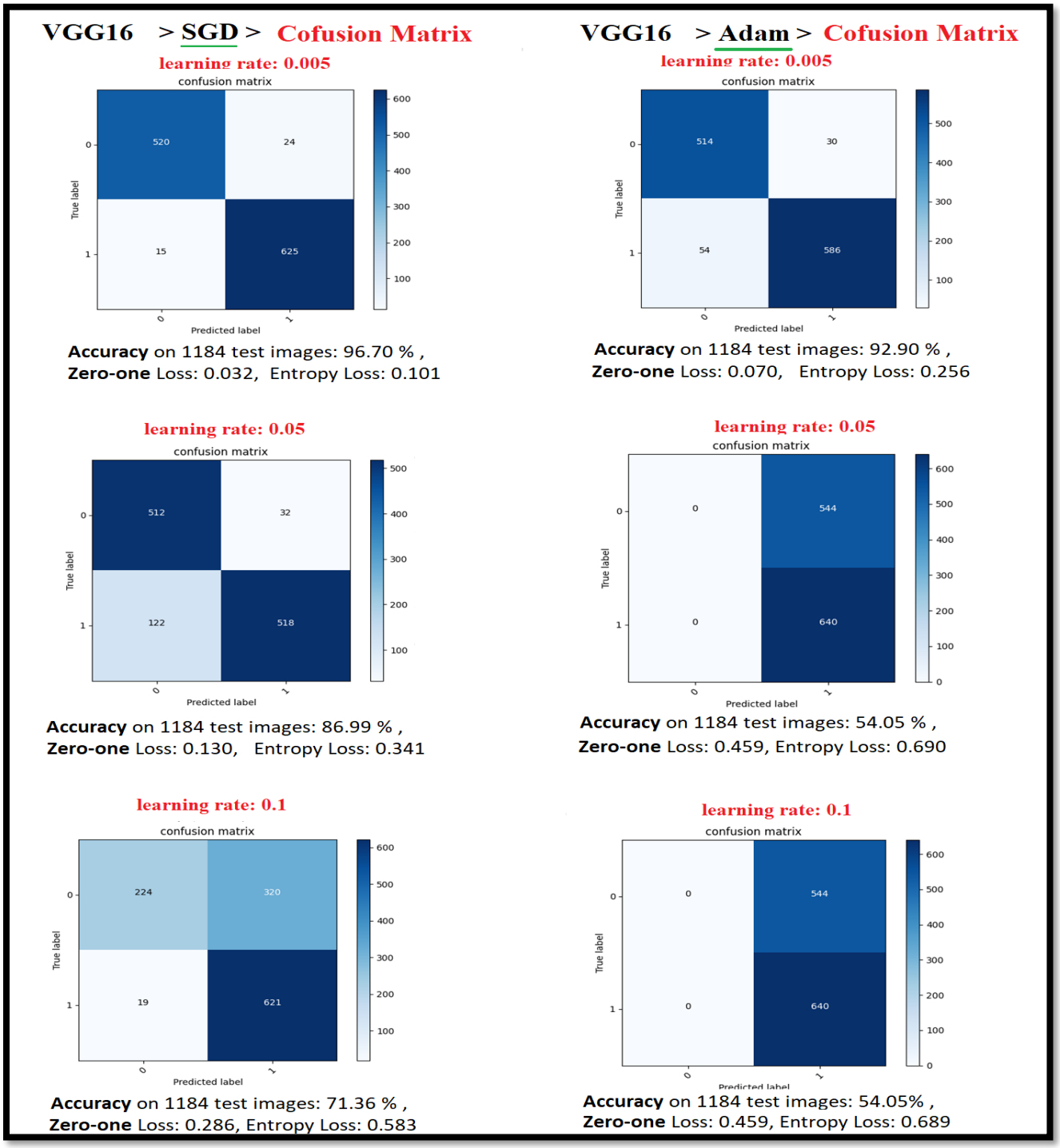


Figure 20. Confusion matrix of VGG16 model for Test data for any Optimizer and Learning rate

Table 19. VGG16 Architecture, both Optimizer and three Learning, Test set (1184 images)

		Test set			Muffin = 0		Chihuahua =1	
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Num of Correct	Num of Incorrect	Num of Correct	Num of Incorrect
Adam	0.005	92.90	0.070	0.256	514	30	586	54
	0.05	54.05	0.459	0.690	0	544	640	0
	0.1	54.05	0.459	0.690	0	544	640	0
SGD	0.005	96.70	0.032	0.101	520	24	625	15
	0.05	86.99	0.130	0.341	512	32	518	122
	0.1	71.36	0.286	0.583	224	320	621	19

Description:

The table shows that using **VGG16** model with **Adam** and a learning rate of 0.005, the model achieves an accuracy of 92.90%, meaning that it correctly classifies around 92.90% of the test images, and the zero-one loss is 0.032, suggesting that only a small percentage of samples are misclassified. A lower zero-one loss indicates better model performance. While VGG16 with Adam optimizer in learning rate of 0.05 and 0.1 cannot predict well the Muffin. This represents that large value of leaning rate does not operate well for this optimizer in this case.

Num of Correct and Num of Incorrect columns represent the number of correctly classified and misclassified samples for each class (Muffin and Chihuahua). For example, when using Adam with a learning rate of 0.005, the model correctly classifies 514 images of Muffins and 586 images of Chihuahuas, while misclassifying 30 Muffins and 54 Chihuahuas.

The SGD optimizer with a learning rate of 0.005 performs the best, achieving a highest accuracy (96.70%) and lowest zero-one loss (0.032). A low zero-one loss value (in this case, 0.032) indicates that only a small percentage of samples are misclassified, further confirming the model's effectiveness. But as the learning rate increases for SGD optimizer (0.05 and 0.1), the model's performance degrades, leading to lower accuracy and higher zero-one loss. The model struggles to correctly classify a significant number of samples for these learning rates. For example, The SGD optimizer with a learning rate of 0.1 misclassify almost 40 percent of Muffin images, while for Chihuahua it performs well.

Summary of **VGG16** Architecture:

The following table shows the result of model for all Train, Validation and Test set based on the optimizer type, learning rate and etc. And **risk estimates of 5-fold cross validation** computed for all scenarios.

*Table 20. **VGG16** Architecture, both Optimizer and three Learning, for all Train, Validation and Test set*

		Train-Mean of folds			**Validation**- Mean of folds			**Test**		
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	90.597	0.0014	0.004	87.481	0.193	0.279	92.90	0.070	0.256
	0.05	54.015	0.007	0.011	54.063	0.459	0.692	54.05	0.459	0.690
	0.1	53.879	0.007	0.011	53.903	0.461	1.069	54.05	0.459	0.690
SGD	0.005	99.928	0	0	98.257	0.0174	0.058	96.70	0.032	0.101
	0.05	93.365	0.001	0.0026	78.336	0.2164	0.5838	86.99	0.130	0.341
	0.1	84.546	0.0022	0.006	67.870	0.364	1.519	71.36	0.286	0.583

Description:

The result shows that not only in train and validation but also in test set VGG16 Architecture with **SGD** optimizer and learning rate of 0.005 performs better than other scenarios.

Overall, The VGG16 model with SGD optimizer and a learning rate of 0.005 performs the best. Means that it has learned to generalize well from the training data, making accurate predictions on unseen samples from the test set.

ResNet Implementation Result

5-Fold Cross-Validation and Computing risk estimates

The following tables represent the results of training ResNet architectures using the SGD and Adam optimizer with different learning rates (0.005, 0.05, and 0.1) through 5-fold cross-validation. The tables show the performance metrics for both the training and validation sets in each fold. Finally, the accuracy and the **risk estimates (Zero-one Loss)** of 5-fold cross validation computed for all scenarios by using mean of all folds.

*Table 31. ResNet Architecture, **Adam** Optimizer with Learning **rate 0.005** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	98.564	0	0.001	79.624	0.204	1.151
2	99.66	0.001	0	88.896	0.033	0.732
3	100	0	0	99.471	0.005	0.018
4	100	0	0	99.577	0.155	0.021
5	100	0	0	99.047	0.01	0.043
Mean	99.645	0.0002	0.0002	93.323	0.0814	0.393
St.deviation	0.622	0.000	0.000	8.901	0.092	0.522

*Table 42. ResNet Architecture, **Adam** Optimizer with Learning **rate 0.05** with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	98.646	0	0.001	79.136	0.209	1.418
2	100	0	0	98.415	0.016	0.059
3	99.995	0	0	97.646	0.024	0.073
4	100	0	0	99.048	0.01	0.025
5	100	0	0	99.471	0.005	0.030
Mean	99.728	0	0.0002	94.743	0.0528	0.321
St.deviation	0.605	0.000	0.000	8.752	0.088	0.614

*Table 53. ResNet Architecture, **Adam** Optimizer with Learning rate 0.1 with 5-fold cross validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	96.476	0.001	0.001	75.904	0.241	1.835
2	99.997	0	0	95.949	0.041	0.18
3	99.992	0	0	97.602	0.024	0.074
4	100	0	0	99.645	0.004	0.013
5	100	0	0	99.471	0.005	0.029
Mean	99.293	0.0002	0.0002	93.714	0.063	0.4262
St.deviation	1.575	0.000	0.000	10.070	0.101	0.790

*Table 24. ResNet Architecture, **SGD** Optimizer with Learning **rate 0.005** with 5-fold cross-validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	100	0	0	92.71	0.073	0.234
2	100	0	0	97.041	0.03	0.088
3	100	0	0	96.748	0.033	0.092
4	100	0	0	96.999	0.191	0.098
5	100	0	0	96.128	0.039	0.143
Mean	100	0	0	95.925	0.0732	0.131
St.deviation	0	0	0	1.834	0.068	0.062

*Table 25. ResNet Architecture, **SGD** Optimizer with Learning rate 0.05 with 5-fold cross-validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	93.295	0.001	0.003	81.554	0.184	0.457
2	95.909	0.001	0.002	72.841	0.272	0.766
3	95.860	0.001	0.002	74.726	0.253	0.731
4	96.317	0.001	0.002	87.945	0.121	0.316
5	95.917	0.001	0.002	80.983	0.190	0.499
Mean	95.460	0.001	0.0022	79.610	0.204	0.553
St.deviation	1.224	0.000	0.000	6.017	0.060	0.191

*Table 26. ResNet Architecture, **SGD** Optimizer with Learning rate 0.1 with 5-fold cross-validation*

	Train			Validation		
Fold	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
1	87.071	0.002	0.005	73.924	0.261	0.687
2	87.679	0.002	0.005	64.864	0.351	0.962
3	87.813	0.002	0.005	63.156	0.368	1.059
4	86.810	0.002	0.005	65.709	0.343	0.775
5	87.072	0.002	0.005	66.096	0.339	0.8
Mean	87.289	0.002	0.005	66.750	0.332	0.856
St.deviation	0.433	0.000	0.000	4.167	0.041	0.151

Description:

Information of Validation in first fold for all tables are not optimal, which shows that our validation dataset in this fold is not appropriate.

The ResNet Architecture with Adam optimizer for all three learning rates in all folds, except first fold, both train and validation operate well.

Both optimizers achieve high accuracy on the training set, often close to 100%, suggesting the models can memorize the training data well. However, for validation set results are different.

For Adam, a learning rate of 0.05 generally shows better validation performance compared to the other learning rates, with good generalization and consistent results.

For SGD, a learning rate of 0.005 also demonstrates relatively good validation performance, with consistent results across the folds and fewer misclassifications.

Risk Estimates of 5-fold cross-validation:

Finally, by combining the 6th row of above tables (Table 21- Table 26) we reach the following table that shows the risk estimates of 5-fold cross validation computed for all scenarios by using **Mean** of all folds.

Table 27. Risk Estimates of 5-fold cross-validation

			Train-Mean of folds			**Validation**- Mean of folds		
Model	**Optimizer**	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
VGG16	**Adam**	0.005	99.645	0.0002	0.0002	93.323	0.0814	0.393
		0.05	99.728	0	0.0002	94.743	0.0528	0.321
		0.1	99.293	0.0002	0.0002	93.714	0.063	0.4262
	SGD	0.005	100	0	0	95.925	0.0732	0.131
		0.05	95.460	0.001	0.0022	79.610	0.204	0.553
		0.1	87.289	0.002	0.005	66.750	0.332	0.856

Description:

a. The ResNet Architecture with **Adam** optimizer and learning rate of 0.05 shows outstanding performance, achieving almost perfect accuracy (99.728%) on the training dataset and high accuracy (94.743%) on the validation dataset. The zero-one loss is extremely low, indicating very few misclassifications. The learning rate of 0.005 and 0.1 also perform well, with nearly perfect accuracy and low loss metrics on both datasets.

b. The ResNet Architecture with **SGD** optimizer and learning rate of 0.005 demonstrates best performance, achieving perfect accuracy (100%) on the training dataset and high accuracy (95.925%) on the validation dataset. The zero-one loss is extremely low, indicating excellent performance with almost no misclassifications in train set and the zero-one loss for validation set is 0.0732. The SGD Optimizer with the learning rates of 0.1 and 0.05, while still achieving a respectable accuracy on the training dataset, show a noticeable drop in performance on the validation dataset, with high Zero-One loss metrics compared to the learning rates of 0.005.

Model Recommendations:

For the ResNet architecture with the **Adam** optimizer, the learning rates of 0.005, 0.05, and 0.1 are all excellent choices, providing near-perfect performance on both training and validation datasets. For the ResNet architecture with the **SGD** optimizer, the learning rates of 0.005 is the best choice.

Accuracy:

The following charts display the Accuracy of **ResNet** model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

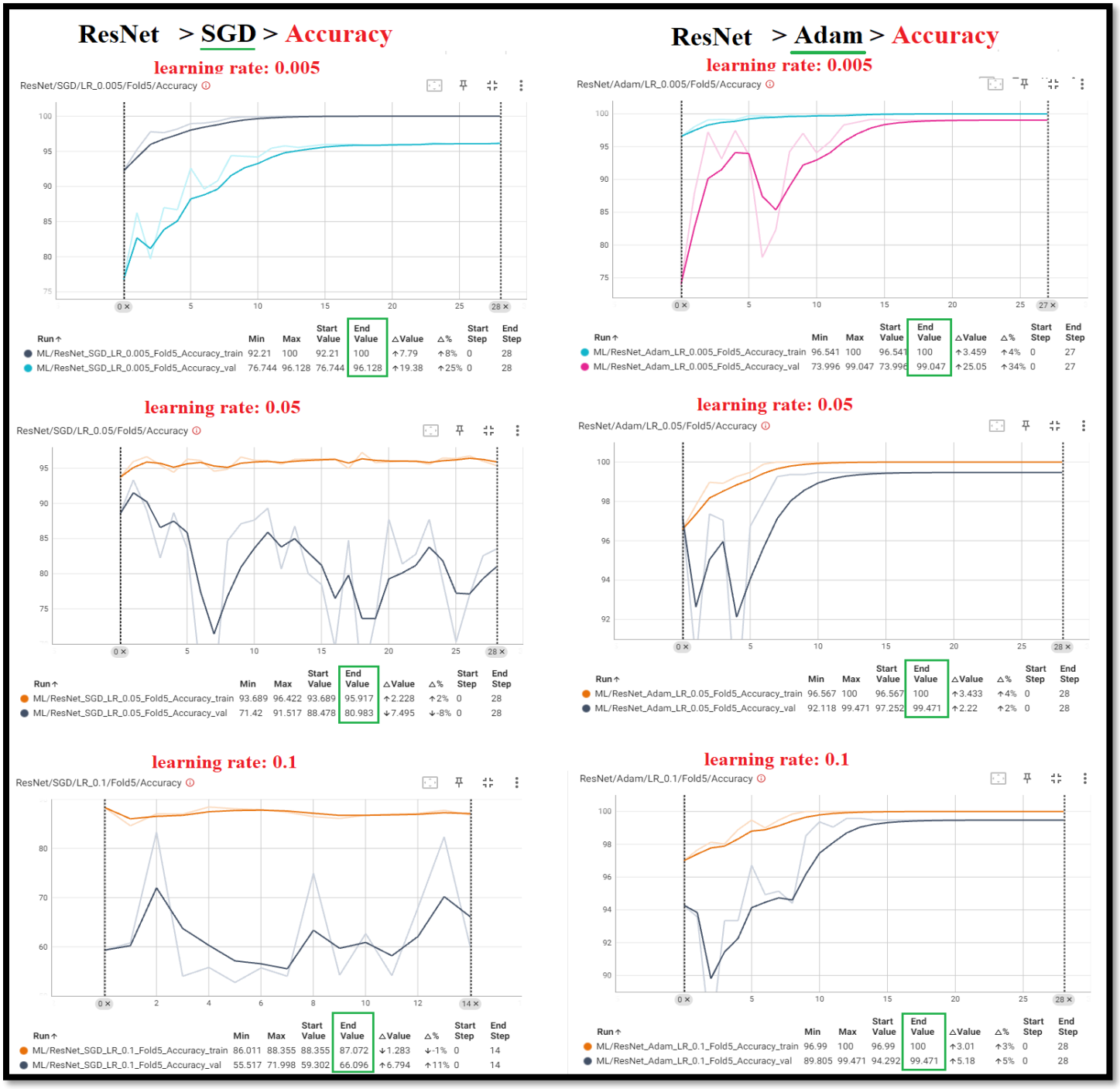


Figure 21: Accuracy of ResNet model for different Epochs in train and validation data, in all scenarios, 5th fold

Zero-One loss:

The following charts display the Zero-One loss of **ResNet** model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

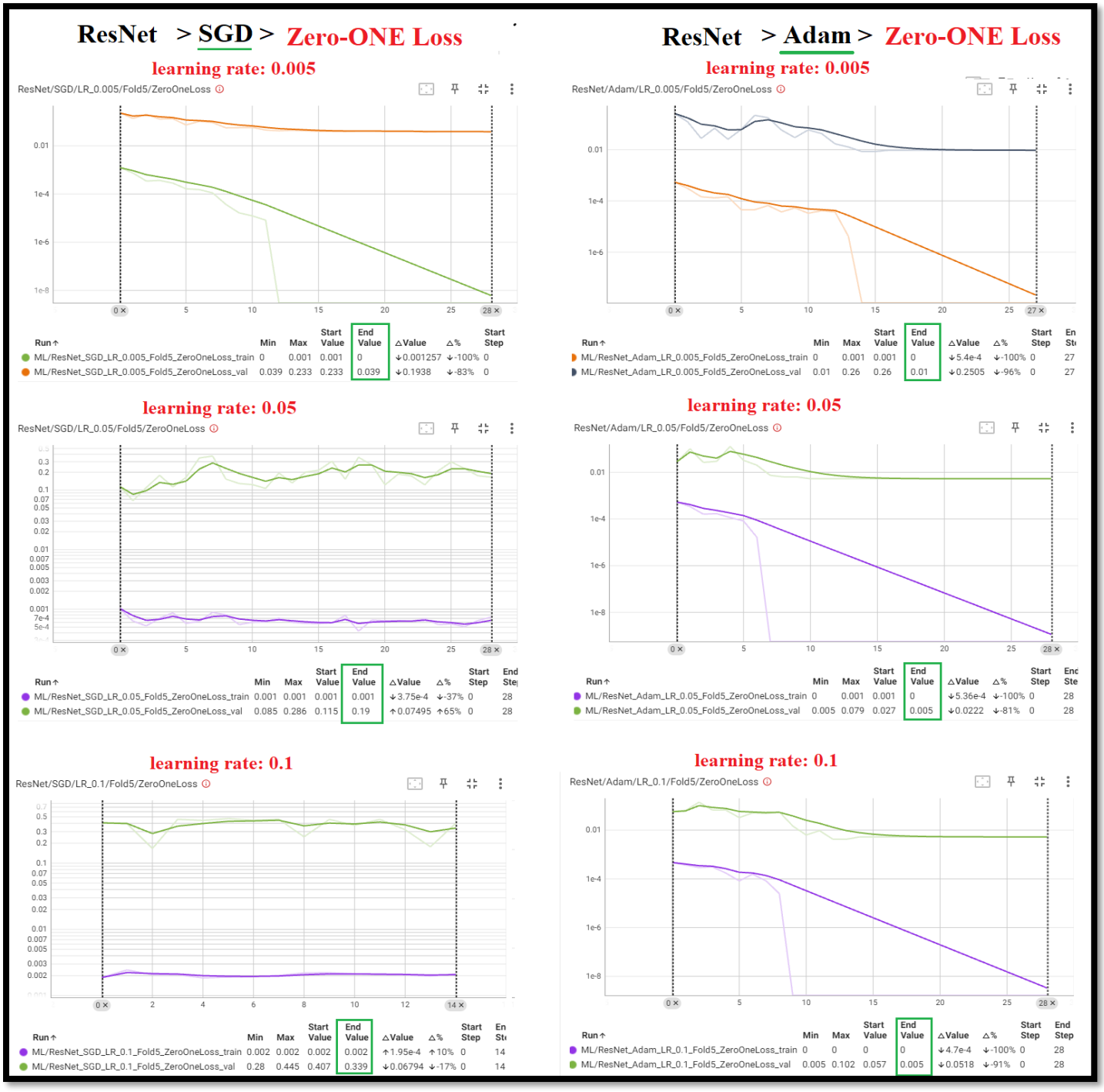


Figure 22: Zero-One loss of ResNet model for different Epochs in train and validation set, in all scenarios, 5th fold

Binary cross entropy loss (BCE):

The following charts display the Binary cross entropy loss (BCE) of **ResNet** model for **Last fold (5th fold)** in different epochs for different leaning rate and Optimizer.

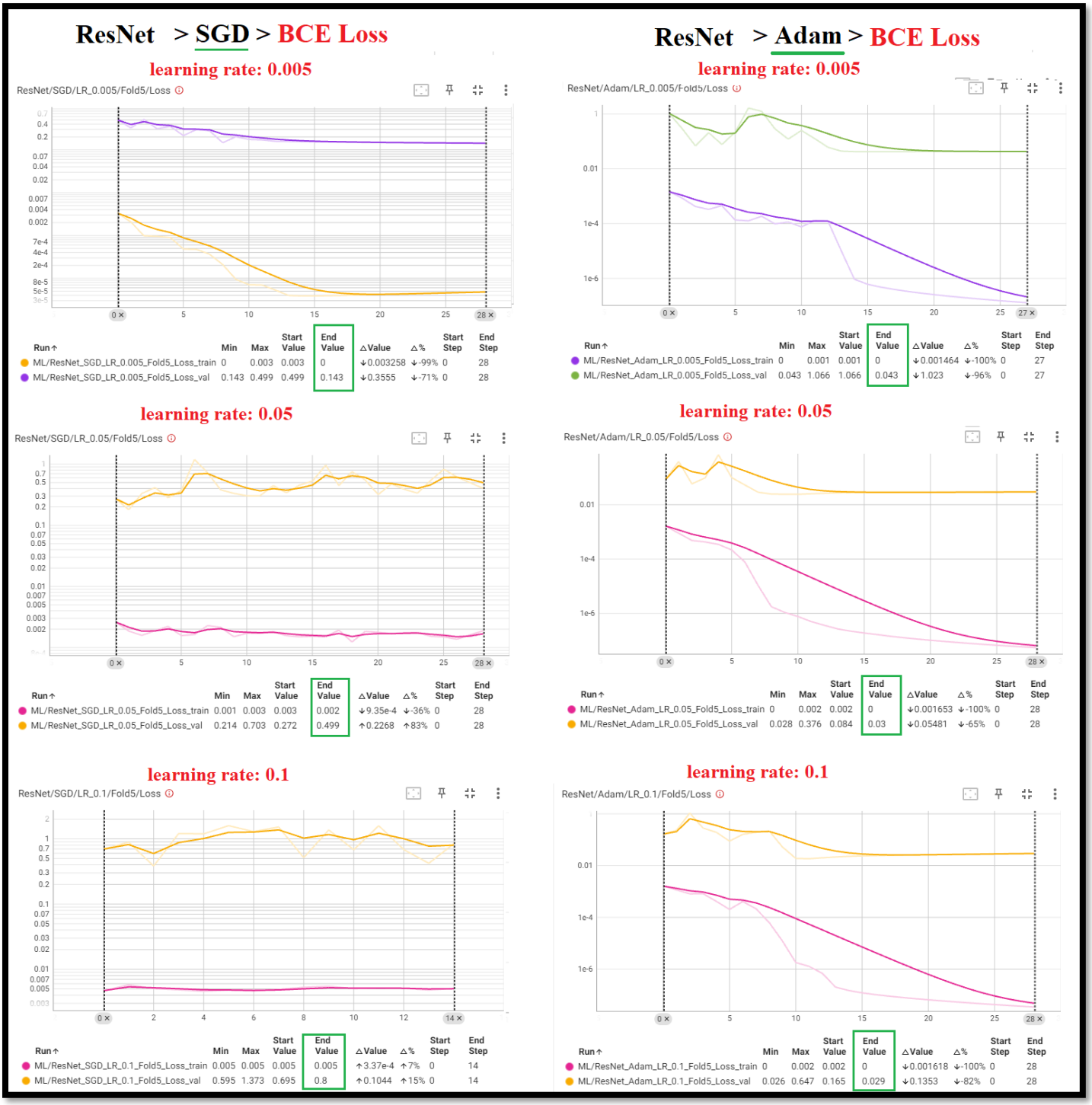


Figure 23: Binary cross entropy loss of ResNet model for all Epochs in all scenarios, 5th fold

Quick review of all above figures in one table:

The following table displays quick review of all above figures for Accuracy, Binary cross-entropy (BCE) loss and Zero-one Loss for **Last Fold** (5th fold) of train and validation datasets.

*Table 67. **ResNet** Architecture for both Optimizer and three Learning for **Last fold** of cross-validation*

		Train			***Validation***		
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	100	0	0	99.047	0.01	0.043
	0.05	100	0	0	99.471	0.005	0.030
	0.1	100	0	0	99.471	0.005	0.029
SGD	0.005	100	0	0	96.128	0.039	0.143
	0.05	95.917	0.001	0.002	80.983	0.190	0.499
	0.1	87.072	0.002	0.005	66.096	0.339	0.8

Description:

ResNet Architecture with **Adam** optimizer in last fold, achieves perfect accuracy (100%) on the training data for all learning rates, indicating that the models have learned the training data well and can classify it without errors. The models also perform well on the validation data, with accuracy ranging from 99.047% to 99.471%. This indicates that the models have generalizability and can perform well on unseen data. The zero-one loss and entropy loss metrics on the validation data are also relatively low, indicating that the models' predictions are reliable on the validation data.

ResNet Architecture with **SGD** optimizer in last fold, achieves perfect accuracy (100%) on the training data for learning rate 0.005, indicating that the models have learned the training data well and can classify it without errors. The model also performs well on the validation data, with accuracy 96.128%. The SGD optimizer generally performs well on the training dataset with learning rate 0.05, achieving good accuracy and low Zero-one loss. But on the validation dataset for the learning rate of 0.05 and 0.1 does not perform well.

Test Dataset Result of ResNet Model:

Confusion matrix:

The following charts display the result of confusion matrix of test dataset for ResNet model in different leaning rate and Optimizers.

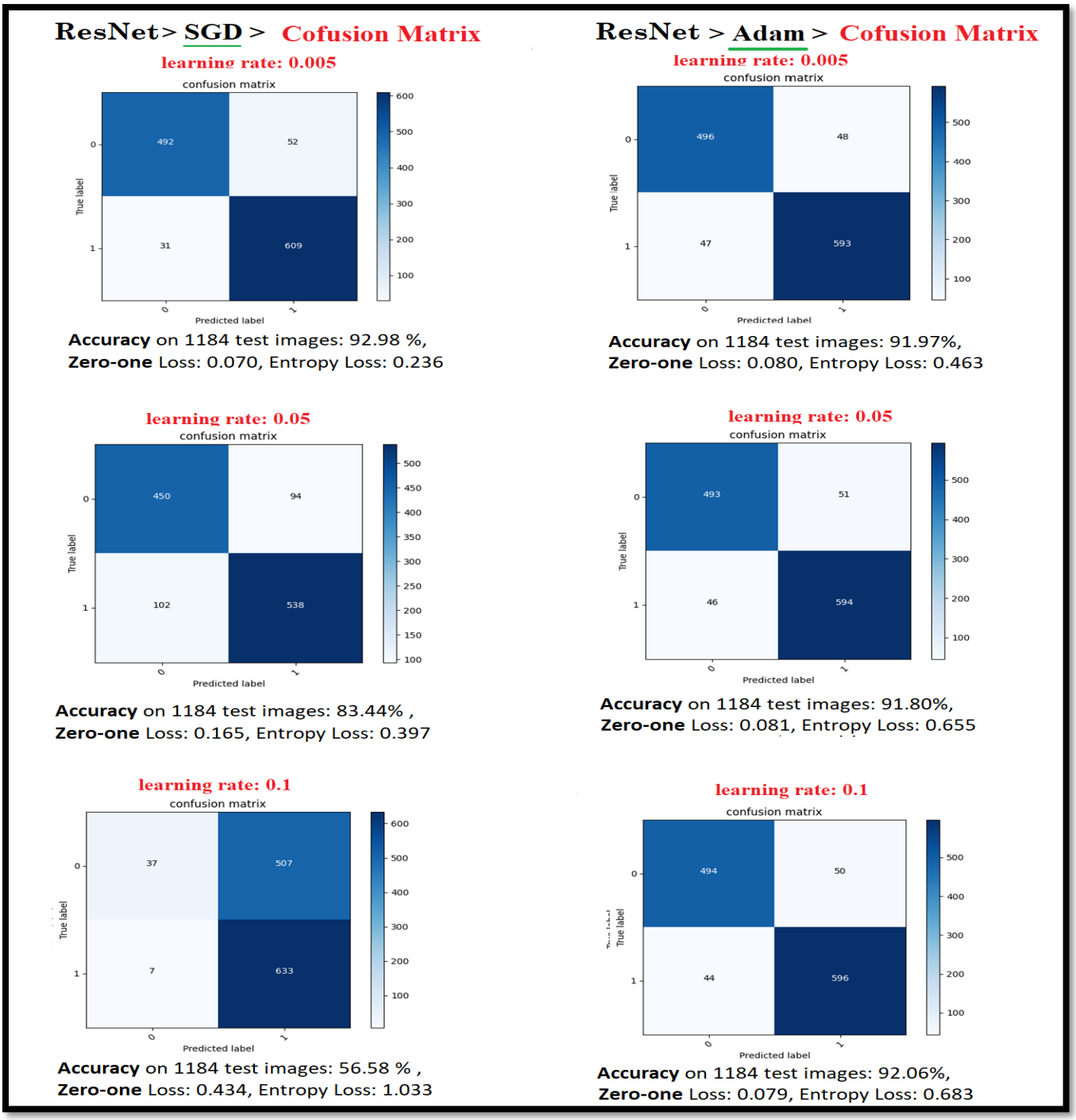


Figure 24. Confusion matrix of ResNet model for Test data for any Optimizer and Learning rate

Table 78. Test result of ResNet Architecture, both Optimizer and three Learning, Test set (1184 images)

		Test set			Muffin = 0		Chihuahua =1	
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Num of Correct	Num of Incorrect	Num of Correct	Num of Incorrect
Adam	0.005	91.97	0.080	0.463	496	48	593	47
	0.05	91.80	0.081	0.655	493	51	594	46
	0.1	92.06	0.079	0.683	494	50	596	44
SGD	0.005	92.98	0.070	0.236	492	52	609	31
	0.05	83.44	0.165	0.397	450	94	538	102
	0.1	56.58	0.434	1.033	37	507	633	7

Description:

The table shows that using **ResNet** model with **SGD** optimizer and learning rate of 0.005 performs the best. The model achieves an accuracy of 92.98%, meaning that it correctly classifies around 92.98% of the test images, and the zero-one loss is 0.07, suggesting that only a small percentage of samples are misclassified. While SGD optimizer with learning rate of 0.1 cannot predict well the Muffin. This represents that large value of leaning rate does not operate well for this optimizer in this case.

ResNet model with **Adam** optimizer for all learning rate of 0.005, 0.05 and 0.1 shows similar accuracy and zero-one loss. This represents that ResNet model with Adam optimizer are not so sensitive to learning rate.

Num of Correct and Num of Incorrect: These columns represent the number of correctly classified and misclassified samples for each class (Muffin and Chihuahua). For instance, when using Adam with a learning rate of 0.005, the model correctly classifies 496 images of Muffins and 593 images of Chihuahuas, while misclassifying 48 Muffins and 47 Chihuahuas.

Summary of **ResNet** Architecture:

The following table shows the result of model for all Train, Validation and Test set based on the optimizer type, learning rate and etc. And **risk estimates** of 5-fold cross validation computed for all scenarios.

*Table 89. **ResNet** Architecture, both Optimizer and three Learning, for all Train, Validation and Test set*

		Train-mean of folds			**Validation**- mean of folds			**Test**		
Optimizer	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
Adam	0.005	99.645	0.0002	0.0002	93.323	0.0814	0.393	91.97	0.080	0.463
	0.05	99.728	0	0.0002	94.743	0.0528	0.321	91.80	0.081	0.655
	0.1	99.293	0.0002	0.0002	93.714	0.063	0.4262	92.06	0.079	0.683
SGD	0.005	100	0	0	95.925	0.0732	0.131	92.98	0.070	0.236
	0.05	95.460	0.001	0.0022	79.610	0.204	0.553	83.44	0.165	0.397
	0.1	87.289	0.002	0.005	66.750	0.332	0.856	56.58	0.434	1.033

Description:

The result shows that not only in train and validation but also in test set ResNet Architecture with SGD optimizer and learning rate of 0.005 performs better than other scenarios.

Overall, The ResNet model with SGD optimizer and a learning rate of 0.005 performs the best. Means that it has learned to generalize well from the training data, making accurate predictions on unseen samples from the test set.

Summary and Comparison

The following table provides a comprehensive overview of the mean performance of all CNN architectures over multiple folds with different optimizers (Adam and SGD) and three learning rates (0.005, 0.05, and 0.1) in Train, Validation Dataset. Additionally, the results are provided for the Test dataset, which assesses the models' performance on unseen data.

Table 30: Value of Loss and Accuracy for all CNN Architectures in Train, validation and Test Dataset

			Train-mean of folds			**Validation**- mean of folds			**Test**		
Model	**Optimizer**	**Learning Rate**	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss	Accuracy	Zero one Loss	Entropy Loss
AlexNet	**Adam**	0.005	96.198	0.0004	0.0014	89.106	0.110	0.713	92.48	0.075	1.269
		0.05	54.151	0.007	0.0114	54.7972	0.452	0.684	54.05	0.459	0.691
		0.1	53.450	0.0072	0.0114	53.877	0.461	0.725	54.05	0.459	0.690
	SGD	0.005	99.868	0	0	95.715	0.042	0.192	93.24	0.067	0.207
		0.05	92.571	0.001	0.003	73.437	0.265	0.650	86.99	0.130	0.308
		0.1	85.100	0.0022	0.0056	70.141	0.298	0.586	74.83	0.251	0.506
VGG16	**Adam**	0.005	90.597	0.0014	0.004	87.481	0.193	0.279	92.90	0.070	0.256
		0.05	54.015	0.007	0.011	54.063	0.459	0.692	54.05	0.459	0.690
		0.1	53.879	0.007	0.011	53.903	0.461	1.069	54.05	0.459	0.690
	SGD	0.005	99.928	0	0	98.257	0.0174	0.058	96.70	0.032	0.101
		0.05	93.365	0.001	0.0026	78.336	0.2164	0.5838	86.99	0.130	0.341
		0.1	84.546	0.0022	0.006	67.870	0.364	1.519	71.36	0.286	0.583
ResNet	**Adam**	0.005	99.645	0.0002	0.0002	93.323	0.0814	0.393	91.97	0.080	0.463
		0.05	99.728	0	0.0002	94.743	0.0528	0.321	91.80	0.081	0.655
		0.1	99.293	0.0002	0.0002	93.714	0.063	0.4262	92.06	0.079	0.683
	SGD	0.005	100	0	0	95.925	0.0732	0.131	92.98	0.070	0.236
		0.05	95.460	0.001	0.0022	79.610	0.204	0.553	83.44	0.165	0.397
		0.1	87.289	0.002	0.005	66.750	0.332	0.856	56.58	0.434	1.033

The above table displays that learning rate of **0.005** for all model (AlexNet, VGG16, ResNet) in both optimizers (Adam and SGD) performs well in comparison with other scenarios.

And also, that **SGD** optimizer for all model (AlexNet, VGG16, ResNet) in each of learning rate performs better than Adam optimizer.

The VGG16 with SGD optimizer and learning rate of 0.005 has the best performance in comparison with other scenarios, achieving a highest accuracy (96.70%) and lowest zero-one loss (0.032) in test dataset. This scenario also has the high accuracy in train and validation (99.92% and 98.25% respectively) set and so low Zero-One loss (0 and 0.058 respectively).

For some combinations, the zero-one loss and entropy loss on the Test dataset are relatively higher, indicating that the models may struggle with certain classifications. For example, VGG16 with Adam optimizer in learning rate of 0.05 and 0.1 cannot predict well the Muffin.

In comparison with other models just ResNet model with Adam optimizer shows similar accuracy and zero-one loss for all learning rate of 0.005, 0.05 and 0.1. This represents that ResNet model with Adam optimizer are not so sensitive to learning rate.

Some models, especially with learning rates of 0.1, show low accuracy and higher loss metrics, indicating that this learning rate may not be the best choice for these models on this specific task.

Model Recommendations: Based on the Test dataset results, it appears that certain combinations of model, optimizer, and learning rate perform better than others. For example, using the SGD optimizer with a learning rate of 0.005 for VGG16 and ResNet consistently results in higher accuracy and lower loss metrics. However, further experimentation and tuning may be required to optimize the model's performance for the specific task at hand.

Overall, the table provides comprehensive insights into how these three architectures performs on the train, validation and test datasets under various settings of optimizers and learning rates. It enables a detailed analysis of the model's classification performance for individual classes and helps identify the best-performing combination for the given task of distinguishing between Muffins and Chihuahuas.

References

1. https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5
2. https://duchesnay.github.io/pystatsml/deep_learning/dl_cnn_cifar10_pytorch.html
3. https://www.mygreatlearning.com/blog/resnet/
4. http://www.cs.cornell.edu/courses/cs4780/2018sp/lectures/lecturenote01_MLsetup.html#:~:text=Zero%2Done%20loss%3A,is%20mispredicted%2C%20and%200%20otherwise.
5. https://builtin.com/machine-learning/fully-connected-layer
6. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
7. https://machinelearningmastery.com/k-fold-cross-validation/
8. Code for plotting confusion matrix. https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
9. Pytorch Documentation https://pytorch.org/docs/stable/index.html
10. Scikit-learn documentation https://scikit-learn.org
11. Numpy Documentation http://www.numpy.org/
12. Matplotlib Documentation https://matplotlib.org/
13. **Datasets** https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification
14. Ou, X., Yan, P., Zhang, Y., Tu, B., Zhang, G., Wu, J., & Li, W. (2019). Moving Object Detection Method via ResNet-18 With Encoder–Decoder Structure in Complex Scenes. IEEE Access, 7, 108152-108160.