NYCU Pattern Recognition, Homework 4

312553024, 江尚軒

Part. 1, Kaggle (70% [50% comes from the competition]):

(10%) Implementation Details

Model architecture & hyperparameters (5%)

```python
class BagMean(nn.Module):
    tabnine: test | explain | document | ask
    def __init__(self, instance_model):
        super(BagMean, self).__init__()
        self.instance_model = instance_model

    tabnine: test | explain | document | ask
    def forward(self, x):
        # print("x.shape:", x.shape)
        batch_size, num_instances, channels, height, width = x.size()
        # Flatten the instances into the batch dimension
        x = x.view(-1, channels, height, width)
        # Input the model
        x = self.instance_model(x)
        # Reshape back to [batch_size, num_instances, num_classes]
        x = x.view(batch_size, num_instances, -1)
        # Aggregate the instance scores (mean)
        # print("x.shape:", x.shape)
        x = torch.mean(x, dim=1)
        # x = torch.max(x, dim=1).values
        # print("x.shape:", x.shape)

        return x
```

Take the mean of all instance images for multiple instance learning.

```python
class ResNet18(nn.Module):
    # Change num_classes to 1 for binary classification with BCEWithLogitsLoss
    tabnine: test | explain | document | ask
    def __init__(self, num_classes=1):
        super(ResNet18, self).__init__()
        self.resnet = models.resnet18(weights=True)
        self.resnet.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.resnet.fc = nn.Sequential(
            nn.Dropout(0.5),  # Add dropout layer
            nn.Linear(self.resnet.fc.in_features, num_classes)
        )

    tabnine: test | explain | document | ask
    def forward(self, x):
        return self.resnet(x)
```

ResNet18 with pre-trained weights

```python
class ResNet34(nn.Module):
    # Change num_classes to 1 for binary classification with BCEWithLogitsLoss
    tabnine: test | explain | document | ask
    def __init__(self, num_classes=1):
        super(ResNet34, self).__init__()
        self.resnet = models.resnet34(weights=True)
        self.resnet.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.resnet.fc = nn.Sequential(
            nn.Dropout(0.5),  # Add dropout layer
            nn.Linear(self.resnet.fc.in_features, num_classes)
        )

    tabnine: test | explain | document | ask
    def forward(self, x):
        return self.resnet(x)
```

ResNet34 with pre-trained weights

```python
class ResNet50(nn.Module):
    # Change num_classes to 1 for binary classification with BCEWithLogitsLoss
    tabnine: test | explain | document | ask
    def __init__(self, num_classes=1):
        super(ResNet50, self).__init__()
        self.resnet = models.resnet50(weights=True)
        self.resnet.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.resnet.fc = nn.Sequential(
            nn.Dropout(0.5),  # Add dropout layer
            nn.Linear(self.resnet.fc.in_features, num_classes)
        )

    tabnine: test | explain | document | ask
    def forward(self, x):
        return self.resnet(x)
```

ResNet50 with pre-trained weights

```python
# Add arguments
parser.add_argument('--name', type=str, default="test", help="Model name")
parser.add_argument('--model', type=str, default="test", help="Model type")
parser.add_argument('--dataset_path', type=str, default='./dataset', help="Path to dataset")
parser.add_argument('--epochs', type=int, default=10, help="Number of epochs")
parser.add_argument('--batch_size', type=int, default=1, help="Batch size")
parser.add_argument('--learning_rate', type=float, default=0.00001, help="Learning rate")
parser.add_argument('--weight_decay', type=float, default=1e-4, help="L2 regularization weight decay")
parser.add_argument('--bag', type=str, default="mean", help="Model name")
```

Default hyperparameters

Training strategy (5%)

```python
def load_data(dataset_path, classes):
    bags = []
    labels = []

    # For each class
    for class_ in classes:
        train_dataset_path = os.path.join(
            dataset_path, "train", f"class_{class_}")
        train_dataset_files = os.listdir(train_dataset_path)
        print(f">> Load the {train_dataset_path}...")

        # For each file
        for train_dataset_file in train_dataset_files:
            train_dataset_file_path = os.path.join(
                train_dataset_path, train_dataset_file)

            # Load the file
            with open(train_dataset_file_path, 'rb') as f:
                data = pickle.load(f)

            bags.append(data)
            labels.append(class_)

    return bags, labels
```

Load data

```python
# Split the dataset into training and validation sets
train_bags, valid_bags, train_labels, valid_labels = train_test_split(
    bags, labels, test_size=0.1
)
```

Split the dataset into training and validation sets

```python
# Create BagDataset
print("> Create BagDataset...")
train_dataset = BagDataset(train_bags, train_labels, transform=train_transforms)
valid_dataset = BagDataset(valid_bags, valid_labels, transform=val_transforms)

print("len(train_dataset):", len(train_dataset))
print("len(valid_dataset):", len(valid_dataset))

# Create DataLoader
print("> Create DataLoader...")
train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=args.batch_size, shuffle=False)

print("len(train_loader):", len(train_loader))
print("len(valid_loader):", len(valid_loader))
```

Create dataset and dataloader

```python
if args.model == "CNN":
    instance_model = CNN()
elif args.model == "ResNet18":
    instance_model = ResNet18()
elif args.model == "ResNet34":
    instance_model = ResNet34()
elif args.model == "ResNet50":
    instance_model = ResNet50()
else:
    raise ValueError(f"Invalid model type: {args.model}")

if args.bag == "mean":
    model = BagMean(instance_model)
elif args.bag == "max":
    model = BagMax(instance_model)
else:
    raise ValueError(f"Invalid bag type: {args.bag}")
```

Initialize the model

```python
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate, weight_decay=args.weight_decay)
```

Initialize the loss function and optimizer

```python
# Set up TensorBoard writer
writer = SummaryWriter(log_dir=os.path.join("log", args.name))
os.makedirs(os.path.join("model", args.name), exist_ok=True)
```

Write the log with Tensorboard

```python
for epoch in tqdm(range(args.epochs), desc="Epoch", position=0):
    # Train the model
    # print(">> Train the model...")
    train_loss, train_accuracy = train_model(model, train_loader, criterion, optimizer)

    # Valid the model
    # print(">> Valid the model...")
    valid_loss, valid_accuracy = valid_model(model, valid_loader, criterion)

    # Write log
    writer.add_scalar('train/loss', train_loss, epoch)
    writer.add_scalar('train/accuracy', train_accuracy, epoch)
    writer.add_scalar('valid/loss', valid_loss, epoch)
    writer.add_scalar('valid/accuracy', valid_accuracy, epoch)

    print(f'\nEpoch: {epoch}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, Valid Loss: {valid_loss:.4f}, Valid Accuracy: {valid_accuracy:.4f}')

    # Save model after each epoch
    model_save_path = os.path.join(os.path.join("model", args.name), f'epoch={epoch}.pth')
    print(f"Save model to {model_save_path}...")
    # Save the entire model
    torch.save(model, model_save_path)
```

Train, validate, and save the model per epoch

(10%) Experimental Results

Evaluation metrics and learning curve (5%)

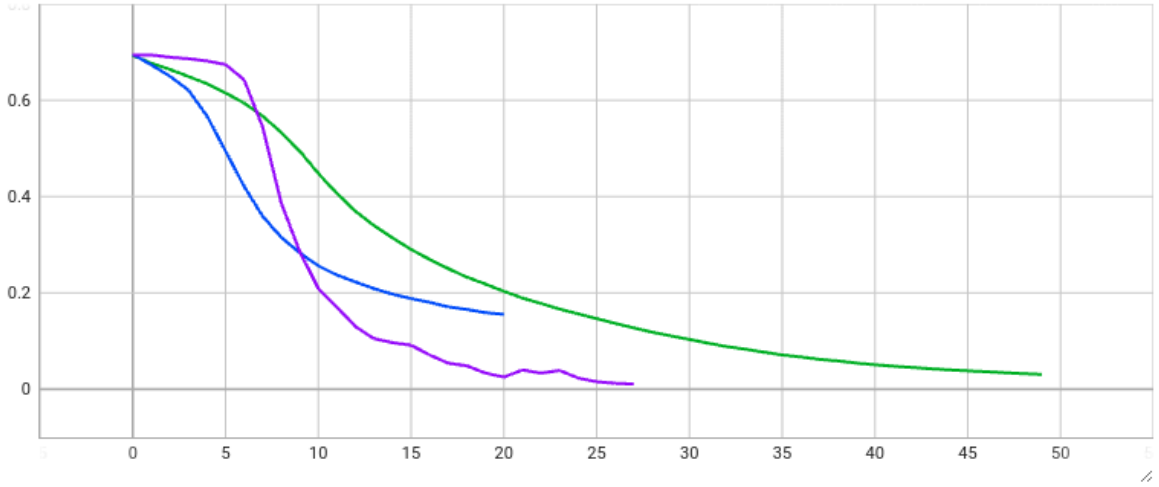
Evaluation metrics: Accuracy

```python
preds = torch.round(torch.sigmoid(outputs))
correct_predictions += torch.sum(preds == labels).item()
total_samples += labels.size(0)
```

```python
valid_accuracy = correct_predictions / total_samples
```

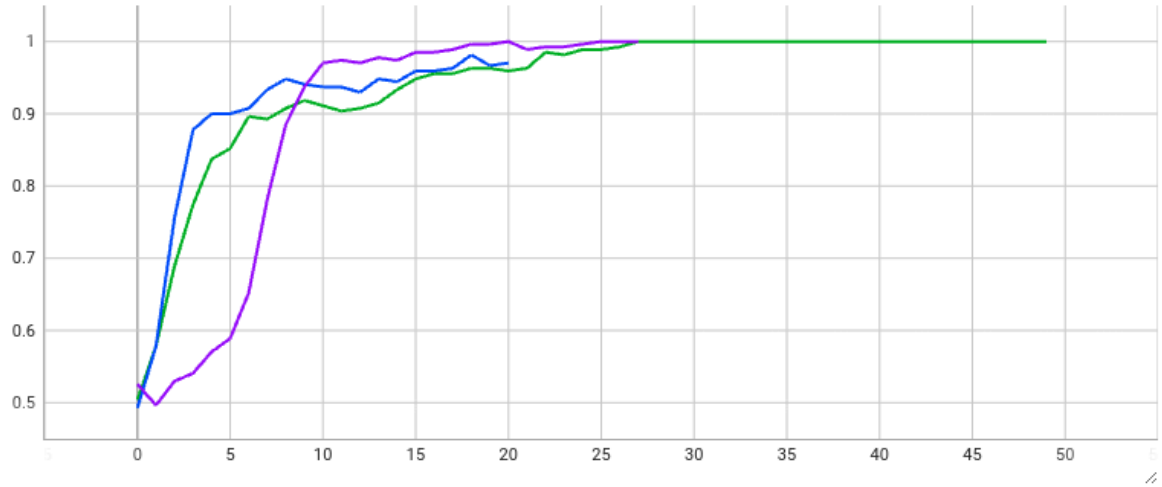
Learning curve:

train/loss

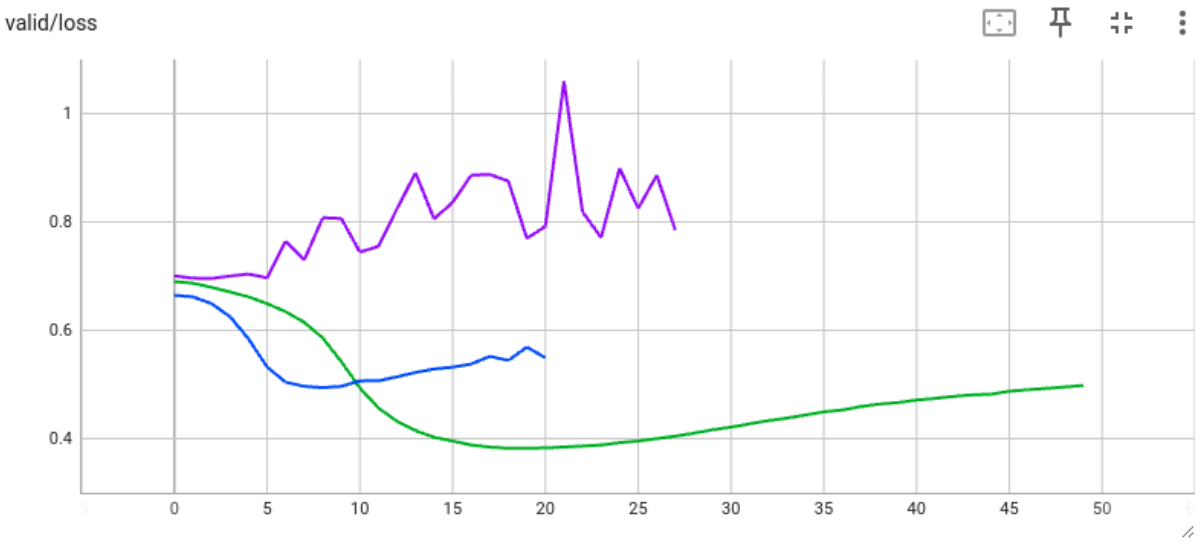


Run ↑	Value	Step	Relative
ResNet-8	0.0308	49	40.69 min
ResNet34-1	0.1555	20	27.1 min
ResNet50-3	0.0106	27	55.83 min

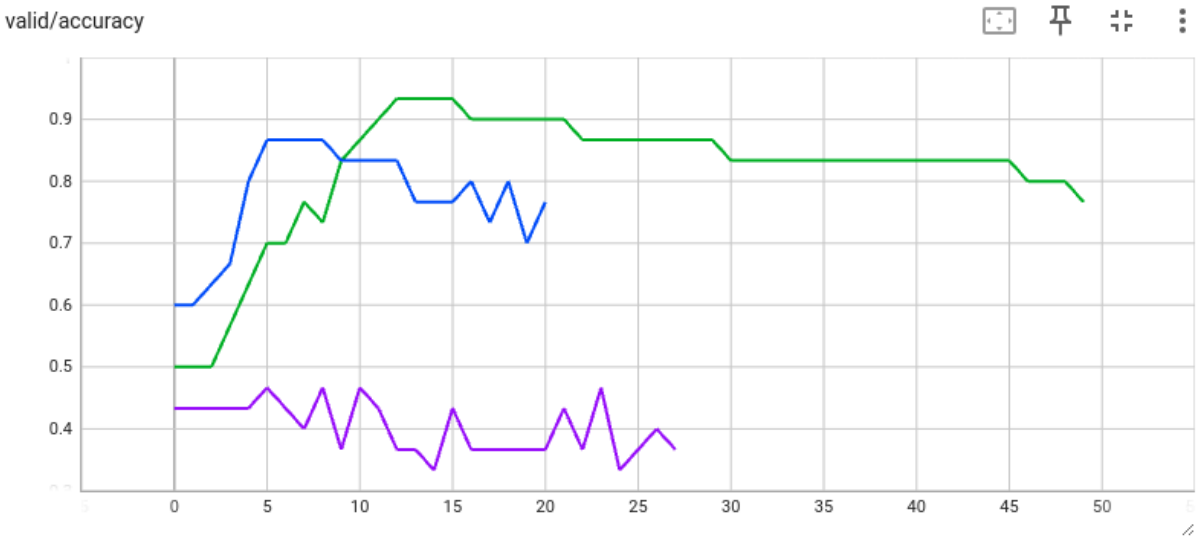
train/accuracy



Run ↑	Value	Step	Relative
ResNet-8	1	49	40.69 min
ResNet34-1	0.9704	20	27.1 min
ResNet50-3	1	27	55.83 min



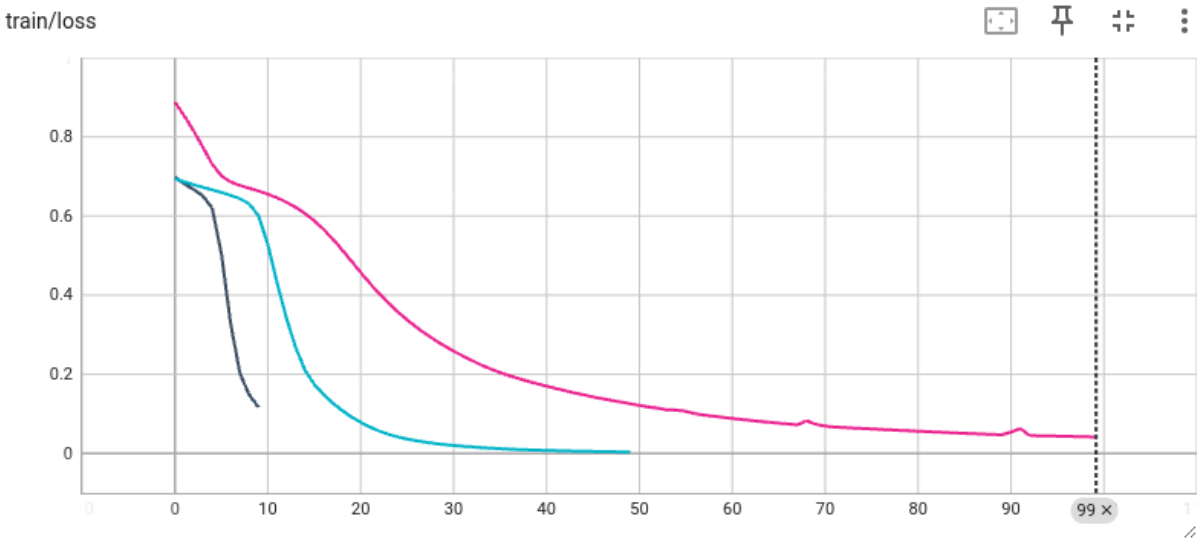
Run	Value	Step	Relative
ResNet-8	0.4982	49	40.69 min
ResNet34-1	0.5492	20	27.1 min
ResNet50-3	0.7849	27	55.83 min



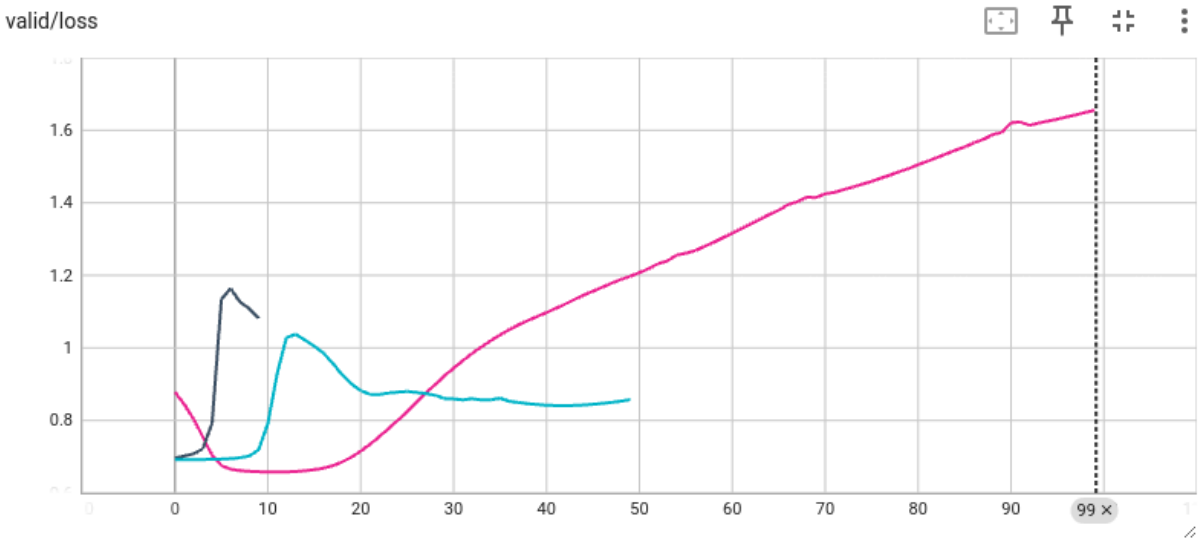
Run	Value	Step	Relative
ResNet-8	0.7667	49	40.69 min
ResNet34-1	0.7667	20	27.1 min
ResNet50-3	0.3667	27	55.83 min

Overall, ResNet18 performed the best during training, followed by ResNet34, while ResNet50 did not train successfully. I also observed that as the number of epochs increased, the train loss decreased, but the valid loss increased. This could be due to overfitting, so I added some techniques to prevent overfitting, such as L2 Regularization, Dropout, and Data Augmentation. However, the overfitting issue still persisted.

Ablation Study (5%)
Learning rate:



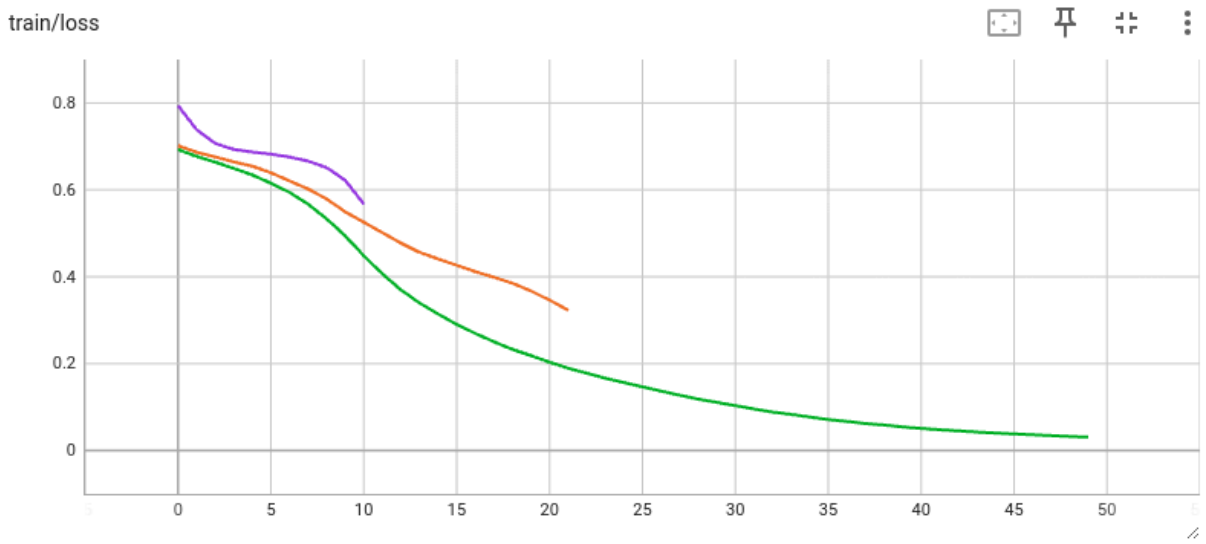
Run ↑	Value	Step	Relative
ResNet-1	0.1168	9	7.763 min
ResNet-2	0.0036	49	42.07 min
ResNet-3	0.0415	99	1.424 hr



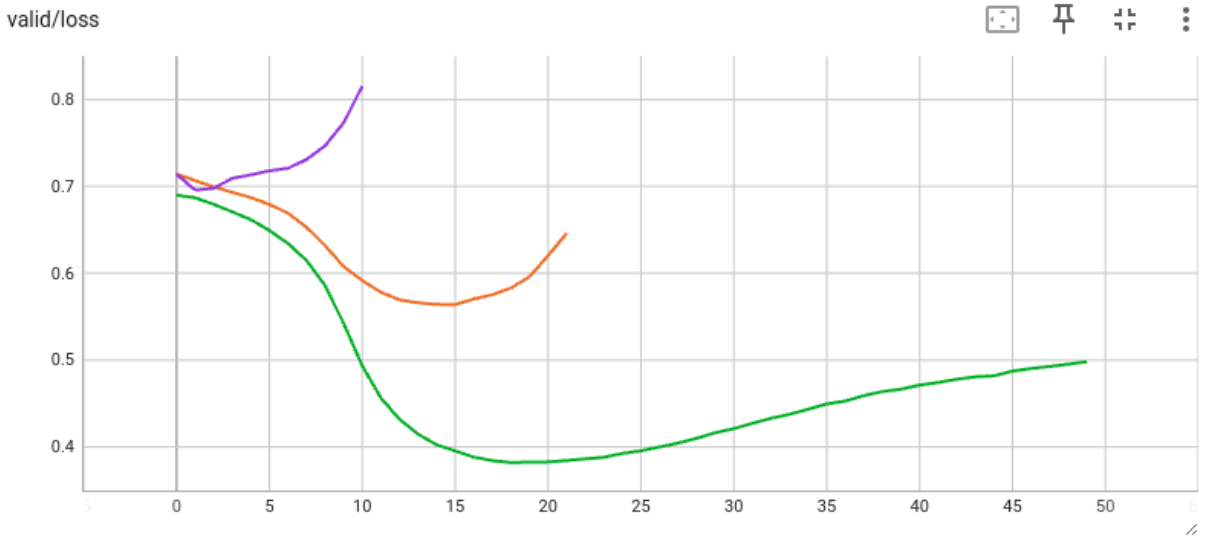
Run ↑	Value	Step	Relative
ResNet-1	1.0814	9	7.763 min
ResNet-2	0.8572	49	42.07 min
ResNet-3	1.655	99	1.424 hr

ResNet-1 is the ResNet18 with learning rate = 0.00001, ResNet-2 is the ResNet18 with learning rate = 0.000005, and ResNet-3 is the ResNet18 with learning rate = 0.000001. The learning rates for ResNet-1 and ResNet-2 were too high, resulting in no decrease in valid loss. Although the valid loss for ResNet-3 did decrease, it quickly overfitted as well.

Batch size:



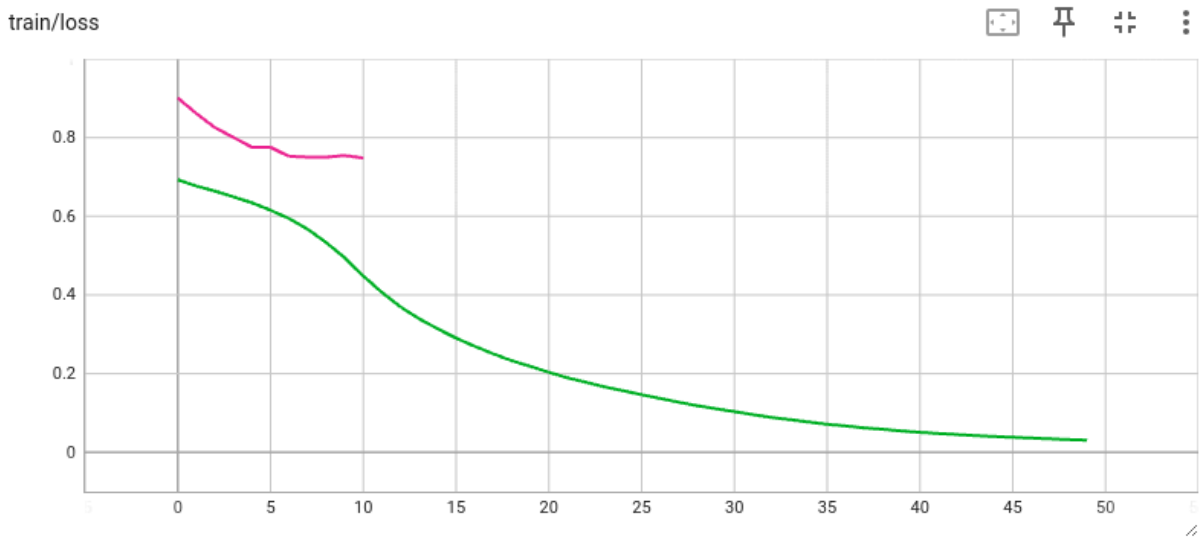
Run	Value	Step	Relative
ResNet-5	0.5674	10	8.868 min
ResNet-7	0.3229	21	18.06 min
ResNet-8	0.0308	49	40.69 min



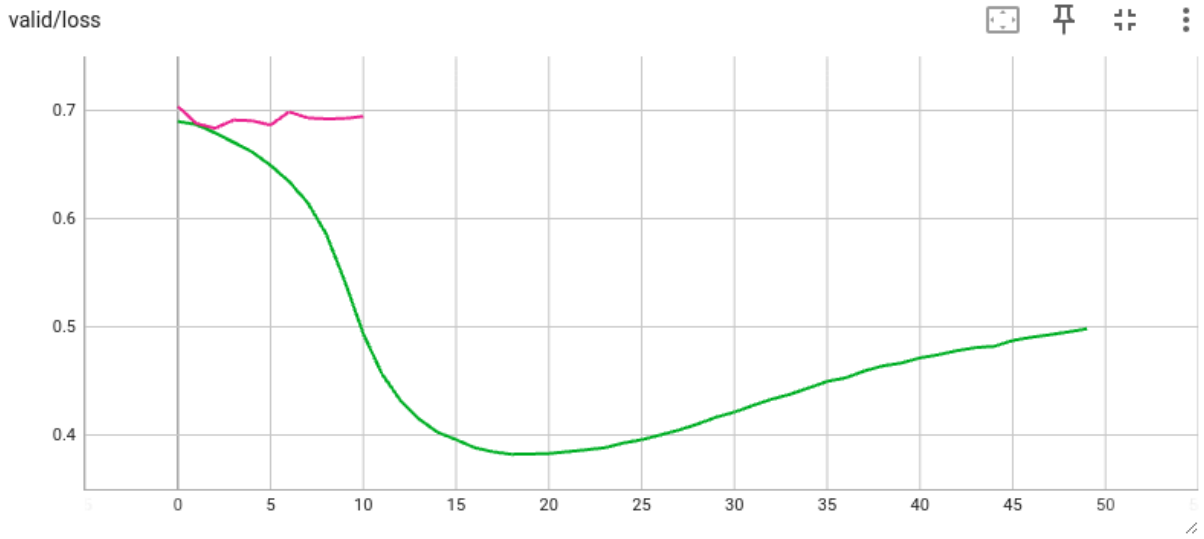
Run	Value	Step	Relative
ResNet-5	0.8153	10	8.868 min
ResNet-7	0.6459	21	18.06 min
ResNet-8	0.4982	49	40.69 min

ResNet-5 is the ResNet18 with batch size = 1, ResNet-7 is the ResNet18 with batch size = 2, and ResNet-8 is the ResNet18 with batch size = 4.
As can be seen from the figure, the larger the batch size, the smaller the valid loss. However, there is still the problem of overfitting.

Multiple Instance Learning:



Run ↑	Value	Step	Relative
ResNet-10	0.7479	10	8.571 min
ResNet-8	0.0308	49	40.69 min



Run ↑	Value	Step	Relative
ResNet-10	0.6945	10	8.571 min
ResNet-8	0.4982	49	40.69 min

ResNet-8 is the ResNet18 with taking the mean of all instance images for multiple instance learning. ResNet-10 is the ResNet18 with taking the max of all instance images for multiple instance learning.
I tried to take the maximum value of all instance images for multiple instance learning, but it failed.

Part. 2, Questions (30%):

1. (10%) Why Sigmoid or Tanh is not preferred to be used as the activation function in the hidden layer of the neural network? Please answer in detail.

   Sigmoid and Tanh are not preferred for the following reasons:

1. **Vanishing Gradient Problem**: The gradients of Sigmoid and Tanh are very small for inputs far from zero, causing the gradients to vanish during backpropagation. This slows down learning in deep networks.
2. **Non-zero Centered Output**: Sigmoid outputs range from 0 to 1, causing inefficient gradient updates. Tanh, while better at -1 to 1, still has vanishing gradients.
3. **Computational Inefficiency**: Both involve expensive exponential calculations, slowing training.

2. (10%) What is overfitting? Please provide at least three techniques with explanations to overcome this issue.

Overfitting occurs when a model learns the training data too well, capturing noise and details specific to the training set. This results in high accuracy on training data but poor performance on unseen test data. Essentially, the model fails to generalize to new data.

Techniques to Overcome Overfitting:

1. **Regularization**: Adds a penalty to the loss function to discourage complex models.
    a. **L2 Regularization (Weight Decay)**: Penalizes large weights by adding the sum of squared weights to the loss.
    b. **L1 Regularization**: Adds the sum of absolute values of weights to the loss, encouraging sparsity.
2. **Dropout**: Randomly drops neurons during training with a certain probability, forcing the network to learn redundant representations. Prevents reliance on specific neurons, improving generalization.
3. **Data Augmentation**: Increases training data diversity by applying random transformations like rotations, flips, and shifts. Helps the model become more robust to variations and reduces overfitting by providing more varied examples.

3. (15%) Given a valid kernel $k_1(x, x')$,prove that the following proposed functions are or are not valid kernels. If one is not a valid kernel, give an example of $k(x, x')$ that the corresponding K is not positive semidefinite and show its eigenvalues.

a. $k(x, x') = k_1(x, x') + \|x\|^2$

b. $k(x, x') = k_1(x, x') - 1$

c. $k(x, x') = k_1(x, x') + \exp(x^T x')$

d. $k(x, x') = \exp\big(k_1(x, x')\big) - 1$

(a) $k(x, x') = k_1(x, x') + \|x\|^2$

Proof:

The term $\|x\|^2$ is not dependent on x', which means it does not satisfy the properties of a Mercer kernel that depends on both x and x'. Therefore, this function does not necessarily produce a valid kernel.

Example:

Let $x = [1, 0]$ and $x' = [0, 1]$.

$\|x\|^2 = 1$ and $\|x'\|^2 = 1$.

If $k_1(x, x')$ is a valid kernel and $k_1(x, x') = 0$, then $k(x, x') = 1$ and $k(x', x) = 1$.

The resulting matrix for a set of points could be:

$$\begin{bmatrix} 1+1 & 0+1 \\ 0+1 & 1+1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

This matrix is positive semidefinite, but generally, adding $\|x\|^2$ could violate kernel properties, so we need to check specific cases to ensure validity.

(b) $k(x, x') = k_1(x, x') - 1$

Proof:

Subtracting a constant from a valid kernel affects the positive semidefiniteness.

Example:

Consider $k_1(x, x')$ is a valid kernel and produces a positive semidefinite matrix. Subtracting 1 might make some eigenvalues negative.

For a set x, x':

Let $k_1(x, x) = 1$, $k_1(x, x') = 0.5$, $k_1(x', x) = 0.5$, and $k_1(x', x') = 1$.

The original matrix is:

$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

Subtracting 1:

$$\begin{bmatrix} 1-1 & 0.5-1 \\ 0.5-1 & 1-1 \end{bmatrix} = \begin{bmatrix} 0 & -0.5 \\ -0.5 & 0 \end{bmatrix}$$

The eigenvalues of this matrix are ± 0.5, indicating it is not positive semidefinite.

(c) $k(x, x') = k_1(x, x') + \exp(x^T x')$

Proof:
If $k_1(x, x')$ is a valid kernel and $\exp(x^T x')$ is also a valid kernel, their sum is valid since the sum of positive semidefinite matrices is positive semidefinite.

Example:
For $x = [1, 0]$, $x' = [0, 1]$:
$x^T x' = 0$, so $\exp(0) = 1$.
If $k_1(x, x') = 0.5$, then $k(x, x') = 0.5 + 1 = 1.5$.
The resulting matrix would be the sum of the kernel matrices, ensuring positive semidefiniteness.

(d) $k(x, x') = \exp(k_1(x, x')) - 1$

Proof:
Exponentiating a valid kernel $k_1(x, x')$ does not guarantee the resulting function is a valid kernel.

Example:
Consider $k_1(x, x') = 0$ for simplicity.
$k(x, x') = \exp(0) - 1 = 1 - 1 = 0$.
For $k_1(x, x') = -1$:
$\exp(-1) - 1 \approx -0.6321$.
The kernel matrix:
$$\begin{bmatrix} \exp(k_1(x, x)) - 1 & \exp(k_1(x, x')) - 1 \\ \exp(k_1(x', x)) - 1 & \exp(k_1(x', x')) - 1 \end{bmatrix}$$
If $k_1(x, x) = 1$:
$$\begin{bmatrix} \exp(1) - 1 & \exp(0) - 1 \\ \exp(0) - 1 & \exp(1) - 1 \end{bmatrix} \approx \begin{bmatrix} 1.718 & 0 \\ 0 & 1.718 \end{bmatrix}$$
The eigenvalues of the resulting matrix might not be positive, invalidating it as a kernel.