Supplementary

Appendix 1

PseudoCode

Algorithm: training & prediction
Input: Images & Labels & Images for Testing
Output: Class & probability of test image

Build data loader //load data
Image normalization //Centralized processing by de-averaging
Create Category Labels //HER2+&HER2-
Build the network // define different networks such as resnet or transformer
Define loss //cross entropy loss is used here
Define the optimizer //The adam optimizer is used here
For epoch in range //iterate over each epoch
 Net.train // Adjust the network to training mode
 Define running loss //Average loss of statistical training process
 For step, data in enumerate(train_bar) // Iterate over the training dataset
  Divide data into labels and images
  Clear gradient information
  Forward Propagation
  Calculate prediction vs true loss
  Backpropagation
  Update each node parameter
 Net.val // Adjust the network to verify mode
 For step, data in enumerate(train_bar) // Iterate over the validation dataset
  Divide data into labels and images
  .Pass the image into the network, and forward the output to get the output
  The category corresponding to the output maximum value
  Accumulate the correct number of samples
  Calculate the correct rate
 If now > before
  update parameters
 Else keep the previous parameters
Save optimal model
Pass in the test image
Image Normalization
Expand the image dimension to [B,C,H,W]
Load category name
Initialize the network
Load Weights
Invoke val mode
Call softmax to scale the network output into probabilities
Get the index value at the maximum
Return Category Name & Predicted Probability

https://dx.doi.org/10.21037/qims-22-967

Python code for training

```
import os
import sys
import json

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from tqdm import tqdm

from model import resnet34


def main():
  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
  print("using {} device.".format(device))

  data_transform = {
    "train": transforms.Compose([transforms.RandomResizedCrop(224),
                 transforms.RandomHorizontalFlip(),
                 transforms.ToTensor(),
                 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])]),
    "val": transforms.Compose([transforms.Resize(256),
               transforms.CenterCrop(224),
               transforms.ToTensor(),
               transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])}

  data_root = os.path.abspath(os.path.join(os.getcwd(), "../..")) # get data root path
  image_path = os.path.join(data_root, "data_set", "flower_data") # flower data set path
  assert os.path.exists(image_path), "{} path does not exist.".format(image_path)
  train_dataset = datasets.ImageFolder(root=os.path.join(image_path, "train"),
                transform=data_transform["train"])
  train_num = len(train_dataset)

  list = train_dataset.class_to_idx
  cla_dict = dict((val, key) for key, val in flower_list.items())
  # write dict into json file
  json_str = json.dumps(cla_dict, indent=4)
  with open('class_indices.json', 'w') as json_file:
    json_file.write(json_str)

  batch_size = 16
  nw = min([os.cpu_count(), batch_size if batch_size > 1 else 0, 8]) # number of workers
  print('Using {} dataloader workers every process'.format(nw))
```


https://dx.doi.org/10.21037/qims-22-967

```
train_loader = torch.utils.data.DataLoader(train_dataset,
                 batch_size=batch_size, shuffle=True,
                 num_workers=nw)

validate_dataset = datasets.ImageFolder(root=os.path.join(image_path, "val"),
                transform=data_transform["val"])
val_num = len(validate_dataset)
validate_loader = torch.utils.data.DataLoader(validate_dataset,
                  batch_size=batch_size, shuffle=False,
                  num_workers=nw)

print("using {} images for training, {} images for validation.".format(train_num,
                           val_num))

net = resnet34()
# load pretrain weights
# download url: https://download.pytorch.org/models/resnet34-333f7ec4.pth
model_weight_path = "./resnet34-pre.pth"
assert os.path.exists(model_weight_path), "file {} does not exist.".format(model_weight_path)
net.load_state_dict(torch.load(model_weight_path, map_location='cpu'))
# for param in net.parameters():
#  param.requires_grad = False

# change fc layer structure
in_channel = net.fc.in_features
net.fc = nn.Linear(in_channel, 5)
net.to(device)

# define loss function
loss_function = nn.CrossEntropyLoss()

# construct an optimizer
params = [p for p in net.parameters() if p.requires_grad]
optimizer = optim.Adam(params, lr=0.0001)

epochs = 3
best_acc = 0.0
save_path = './resNet34.pth'
train_steps = len(train_loader)
for epoch in range(epochs):
  # train
  net.train()
  running_loss = 0.0
  train_bar = tqdm(train_loader, file=sys.stdout)
  for step, data in enumerate(train_bar):
    images, labels = data
    optimizer.zero_grad()
    logits = net(images.to(device))
```


https://dx.doi.org/10.21037/qims-22-967

```
            loss = loss_function(logits, labels.to(device))
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()

            train_bar.desc = "train epoch[{}/{}] loss:{:.3f}".format(epoch + 1,
                                                                     epochs,
                                                                     loss)

        # validate
        net.eval()
        acc = 0.0 # accumulate accurate number / epoch
        with torch.no_grad():
            val_bar = tqdm(validate_loader, file=sys.stdout)
            for val_data in val_bar:
                val_images, val_labels = val_data
                outputs = net(val_images.to(device))
                # loss = loss_function(outputs, test_labels)
                predict_y = torch.max(outputs, dim=1)[1]
                acc += torch.eq(predict_y, val_labels.to(device)).sum().item()

                val_bar.desc = "valid epoch[{}/{}]".format(epoch + 1,
                                                           epochs)

        val_accurate = acc / val_num
        print('[epoch %d] train_loss: %.3f val_accuracy: %.3f' %
              (epoch + 1, running_loss / train_steps, val_accurate))

        if val_accurate > best_acc:
            best_acc = val_accurate
            torch.save(net.state_dict(), save_path)

    print('Finished Training')


if __name__ == '__main__':
    main()
```


https://dx.doi.org/10.21037/qims-22-967