Transfer-Learning-on-Dogs-vs-Cats

August 15, 2019

[1]:
```
#pragma cling add_library_path("/Users/krshrimali/Downloads/libtorch/lib/")
#pragma cling add_include_path("/Users/krshrimali/Downloads/libtorch/include/
 ↪")
#pragma cling add_include_path("/Users/krshrimali/Downloads/libtorch/include/
 ↪torch/csrc/api/include/")
#pragma cling add_library_path("/usr/local/Cellar/opencv/4.1.0_2/lib")
#pragma cling add_include_path("/usr/local/Cellar/opencv/4.1.0_2/include/
 ↪opencv4")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/libiomp5.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/libmklml.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/libc10.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/libtorch.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/
 ↪libcaffe2_detectron_ops.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/
 ↪libcaffe2_module_test_dynamic.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/
 ↪libcaffe2_observers.dylib")
#pragma cling load("/Users/krshrimali/Downloads/libtorch/lib/libshm.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_datasets.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_aruco.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_bgsegm.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_bioinspired.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_calib3d.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_ccalib.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_core.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_dnn_objdetect.4.1.0.dylib")
```

```
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_dnn.4.1.0.
 ↪dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_dpm.4.1.0.
 ↪dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_face.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_features2d.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_flann.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_freetype.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_fuzzy.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_gapi.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_hfs.4.1.0.
 ↪dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_highgui.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_img_hash.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_imgcodecs.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_imgproc.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_line_descriptor.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_ml.4.1.0.
 ↪dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_objdetect.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_optflow.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_phase_unwrapping.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_photo.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_plot.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_quality.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_reg.4.1.0.
 ↪dylib")
```

```
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_rgbd.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_saliency.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_sfm.4.1.0.
 ↪dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_shape.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_stereo.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_stitching.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_structured_light.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_superres.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_surface_matching.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_text.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_tracking.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_video.4.1.
 ↪0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_videoio.4.
 ↪1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_videostab.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_xfeatures2d.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_ximgproc.
 ↪4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/
 ↪libopencv_xobjdetect.4.1.0.dylib")
#pragma cling load("/usr/local/Cellar/opencv/4.1.0_2/lib/libopencv_xphoto.4.
 ↪1.0.dylib")
```

[2]:
```
#include <torch/torch.h>
#include <torch/script.h>
#include <iostream>
#include <dirent.h>
#include <opencv2/opencv.hpp>
```

0.1 Transfer Learning

Before we go ahead and discuss the **Why** question of Transfer Learning, let's have a look at **What is Transfer Learning?** Let's have a look at the Notes from CS231n on Transfer Learning:

> In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

There are 3 scenarios possible:

1. When the data you have is similar (but not enough) to data trained on pre-trained model: Take an example of a pre-trained model trained on ImageNet dataset (containing 1000 classes). And the data we have has Dogs and Cats classes. Fortunate enough, ImageNet has some of the classes of Dog and Cat breeds and thus the model must have learned important features from the data. Let's say we don't have enough data but since the data is similar to the breeds in the ImageNet data set, we can simply use the ConvNet (except the last FC layer) to extract features from our dataset and train only the last Linear (FC) layer. We do this by the following code snippet in `Python`:

```
from torchvision import models
# Download and load the pre-trained model
model = models.resnet18(pretrained=True)

# Set upgrading the gradients to False
for param in model.parameters():
    param.requires_grad = False

# Change the output features to the FC Layer and set it to upgrade gradients as True
resnet18.fc = torch.nn.Linear(512, 2)
for param in resnet18.fc.parameters():
    param.requires_grad = True
```

2. When you have enough data (and is similar to the data trained with pre-trained model): Then you might go for fine tuning the weights of all the layers in the network. This is largely due to the reason that we know we won't overfit because we have enough data.
3. Using pre-trained models might just be enough if you have the data which matches the classes in the original data set.

Transfer Learning came into existence (the answer of **Why Transfer Learning?**) because of some major reasons, which include:

1. Lack of resources or data set to train a CNN. At times, we either don't have enough data or we don't have enough resources to train a CNN from scratch.
2. Random Initialization of weights vs Initialization of weights from the pre-trained model. Sometimes, it's just better to initialize weights from the pre-trained model (as it must have learned the generic features from it's data set) instead of randomly initializing the weights.

0.2 Setting up the data with PyTorch C++ API

At every stage, we will compare the Python and C++ codes to do the same thing, to make the analogy easier and understandable. Starting with setting up the data we have. Note that we do have enough data and it is also similar to the original data set of ImageNet, but since I don't have enough resources to fine tune through the whole network, we perform Transfer Learning on the final FC layer only.

Starting with loading the dataset, as discussed in the blogs before, I will just post a flow chart of procedure.

Let's go ahead and define the required utility functions to define Custom Dataset class.

0.3 Utility Function - 1: `read_data` and `read_label`

Documentation of `read_data` function.

```
torch::Tensor read_data(std::string location)

Function to return image read at location given as type torch::Tensor
     Resizes image to (224, 224, 3)
     Parameters
     ==========
     1. location (std::string type) - required to load image from the location

     Returns
     ==========
     torch::Tensor type - image read as tensor
```

[3]:
```
torch::Tensor read_data(std::string location) {
    /*
     Function to return image read at location given as type torch::Tensor
     Resizes image to (224, 224, 3)
     Parameters
     ==========
     1. location (std::string type) - required to load image from the location

     Returns
     ==========
     torch::Tensor type - image read as tensor
    */
    cv::Mat img = cv::imread(location, 1);
    cv::resize(img, img, cv::Size(224, 224), cv::INTER_CUBIC);
    torch::Tensor img_tensor = torch::from_blob(img.data, {img.rows, img.cols,␣
 ↪3}, torch::kByte);
    img_tensor = img_tensor.permute({2, 0, 1});
    return img_tensor.clone();
}
```

Documentation of `read_label` function.

```
torch::Tensor read_label(int label)

Function to return label from int (0, 1 for binary and 0, 1, ..., n-1 for n-class classificatio
     Parameters
     ==========
     1. label (int type) - required to convert int to tensor

     Returns
     ==========
     torch::Tensor type - label read as tensor
```

[4]:
```
torch::Tensor read_label(int label) {
    /*
     Function to return label from int (0, 1 for binary and 0, 1, ..., n-1 for
 ↪n-class classification) as type torch::Tensor
     Parameters
     ==========
     1. label (int type) - required to convert int to tensor

     Returns
     ==========
     torch::Tensor type - label read as tensor
    */
    torch::Tensor label_tensor = torch::full({1}, label);
    return label_tensor.clone();
}
```

0.4 Utility Function 2: process_images and process_labels

Documentation of process_images function.

```
Function returns vector of tensors (images) read from the list of images in a folder
     Parameters
     ==========
     1. list_images (std::vector<std::string> type) - list of image paths in a folder to be rea

     Returns
     ==========
     std::vector<torch::Tensor> type - Images read as tensors
```

[5]:
```
std::vector<torch::Tensor> process_images(std::vector<std::string> list_images)
 ↪{
    /*
     Function returns vector of tensors (images) read from the list of images
 ↪in a folder
     Parameters
     ==========
```

```
    1. list_images (std::vector<std::string> type) - list of image paths in a␣
 ↪folder to be read

    Returns
    ===========
    std::vector<torch::Tensor> type - Images read as tensors
    */
   std::vector<torch::Tensor> states;
   for(std::vector<std::string>::iterator it = list_images.begin(); it !=␣
 ↪list_images.end(); ++it) {
       torch::Tensor img = read_data(*it);
       states.push_back(img);
   }
   return states;
}
```

Documentation of process_labels function.

```
Function returns vector of tensors (labels) read from the list of labels
    Parameters
    ===========
    1. list_labels (std::vector<int> list_labels) -

    Returns
    ===========
    std::vector<torch::Tensor> type - returns vector of tensors (labels)
```

[6]:
```
std::vector<torch::Tensor> process_labels(std::vector<int> list_labels) {
    /*
    Function returns vector of tensors (labels) read from the list of labels
    Parameters
    ===========
    1. list_labels (std::vector<int> type) - required to convert int to tensor␣
 ↪labels

    Returns
    ===========
    std::vector<torch::Tensor> type - returns vector of tensors (labels)
    */
   std::vector<torch::Tensor> labels;
   for(std::vector<int>::iterator it = list_labels.begin(); it != list_labels.
 ↪end(); ++it) {
       torch::Tensor label = read_label(*it);
       labels.push_back(label);
   }
   return labels;
}
```

0.5 Utility Function 3: `load_images` and `load_labels`:

Documentation of `load_images` function.

```
Function returns vector of strings (image paths) read from the folder name given
     Parameters
     ===========
     1. folder_name (std::string type) - name of folder containing images

     Returns
     ===========
     std::vector<std::string> type - returns vector of image paths
```

[7]:
```
std::vector<std::string> load_images(std::string folder_name) {
    /*
    Function returns vector of strings (image paths) read from the folder name given
     Parameters
     ===========
     1. folder_name (std::string type) - name of folder containing images

     Returns
     ===========
     std::vector<std::string> type - returns vector of image paths
    */
    std::vector<std::string> list_images;

    std::string base_name = folder_name;

    DIR* dir;
    struct dirent *ent;

    if((dir = opendir(base_name.c_str())) != NULL) {
        while((ent = readdir(dir)) != NULL) {
            std::string filename = ent->d_name;
            if(filename.length() > 4 && filename.substr(filename.length() - 3) == "jpg") {
                std::string newf = base_name + filename;
                list_images.push_back(newf);
            }
        }
    }

    return list_images;
}
```

Documentation of `load_labels` function.

```
Function returns vector of int (labels) to each image in the folder (folder_name)
```

```
    Parameters
    ==========
    1. folder_name (std::string type) - name of folder containing images
    2. label (int type) - label of the class (0 or 1 in case of binary, 0 1 ... n-1 in case o
    Returns
    ==========
    std::vector<std::string> type - returns vector of labels assigned to each image of each c
```

[8]:
```
std::vector<int> load_labels(std::string folder_name, int label) {
    /*
    Function returns vector of int (labels) to each image in the folder
 ↪(folder_name)
     Parameters
     ==========
     1. folder_name (std::string type) - name of folder containing images
     2. label (int type) - label of the class (0 or 1 in case of binary, 0 1 ...
 ↪ n-1 in case of n-class classification)
     Returns
     ==========
     std::vector<std::string> type - returns vector of labels assigned to each
 ↪image of each class
    */
    std::vector<int> list_labels;
    DIR* dir;

    std::string base_name = folder_name;

    struct dirent *ent;

    if((dir = opendir(base_name.c_str())) != NULL) {
        while((ent = readdir(dir)) != NULL) {
            std::string filename = ent->d_name;
            if(filename.length() > 4 && filename.substr(filename.length() - 3)
 ↪== "jpg") {
                list_labels.push_back(label);
            }
        }
    }

    return list_labels;
}
```

Since we are done with all the utility functions, we can go ahead and define the `CustomDataset` class.

[9]:
```
class CustomDataset : public torch::data::Dataset<CustomDataset> {
private:
    /* data */
    // Should be 2 tensors
```

```
    std::vector<torch::Tensor> states, labels;
    size_t ds_size;
public:
    CustomDataset(std::vector<std::string> list_images, std::vector<int>␣
 ↪list_labels) {
        states = process_images(list_images);
        labels = process_labels(list_labels);
        ds_size = states.size();
    };

    torch::data::Example<> get(size_t index) override {
        /* This should return {torch::Tensor, torch::Tensor} */
        torch::Tensor sample_img = states.at(index);
        torch::Tensor sample_label = labels.at(index);
        return {sample_img.clone(), sample_label.clone()};
    };

    torch::optional<size_t> size() const override {
        return ds_size;
    };
};
```

0.6 Trainining the FC Layer

Let's first have a look at ResNet18 Network Architecture

Reference: https://www.researchgate.net/figure/ResNet-18-Architecture_tbl1_322476121

The next step is to train the Fully Connected layer that we inserted at the end of the network (`linear_layer`). This one should be pretty straight forward, let's see how to do it.

Documentation of `train()` function.

```
This function trains the network on our data loader using optimizer.

    Also saves the model as model.pt after every epoch.
    Parameters
    ===========
    1. net (torch::jit::script::Module type) - Pre-trained model without last FC layer
    2. lin (torch::nn::Linear type) - last FC layer with revised out_features depending on the
    3. data_loader (DataLoader& type) - Training data loader
    4. optimizer (torch::optim::Optimizer& type) - Optimizer like Adam, SGD etc.
    5. size_t (dataset_size type) - Size of training dataset

    Returns
    ===========
    Nothing (void)
```

[10]:
```
template<typename Dataloader>
```

```
void train(torch::jit::script::Module net, torch::nn::Linear lin, Dataloader&␣
↪data_loader, torch::optim::Optimizer& optimizer, size_t dataset_size) {
    /*
    This function trains the network on our data loader using optimizer.

    Also saves the model as model.pt after every epoch.
    Parameters
    ==========
    1. net (torch::jit::script::Module type) - Pre-trained model without last␣
↪FC layer
    2. lin (torch::nn::Linear type) - last FC layer with revised out_features␣
↪depending on the number of classes
    3. data_loader (DataLoader& type) - Training data loader
    4. optimizer (torch::optim::Optimizer& type) - Optimizer like Adam, SGD␣
↪etc.
    5. size_t (dataset_size type) - Size of training dataset

    Returns
    ==========
    Nothing (void)
    */

   float batch_index = 0;

   for(int i=0; i<15; i++) {
       float mse = 0;
       float Acc = 0.0;

       for(auto& batch: *data_loader) {
           auto data = batch.data;
           auto target = batch.target.squeeze();

           // Should be of length: batch_size
           data = data.to(torch::kF32);
           target = target.to(torch::kInt64);

           std::vector<torch::jit::IValue> input;
           input.push_back(data);
           optimizer.zero_grad();

           auto output = net.forward(input).toTensor();
           // For transfer learning
           output = output.view({output.size(0), -1});
           output = lin(output);

           auto loss = torch::nll_loss(torch::log_softmax(output, 1), target);
```

```
            loss.backward();
            optimizer.step();

            auto acc = output.argmax(1).eq(target).sum();

            Acc += acc.template item<float>();
            mse += loss.template item<float>();

            batch_index += 1;
        }

        mse = mse/float(batch_index); // Take mean of loss
        std::cout << "Epoch: " << i  << ", " << "Accuracy: " << Acc/
 ↪dataset_size << ", " << "MSE: " << mse << std::endl;
        net.save("model.pt");
    }
}
```

0.7 Setting up the Dataset for training

Let's go ahead and load our dataset into `DataLoader` class.

The distribution of the dataset is: `Cat Images`: 200 and `Dog Images`: 200.

[11]:
```
// Set folder names for cat and dog images
std::string name_cats = "/Users/krshrimali/Documents/krshrimali-blogs/dataset/
 ↪train/cat_test/";
std::string name_dogs = "/Users/krshrimali/Documents/krshrimali-blogs/dataset/
 ↪train/dog_test/";

std::vector<std::string> images_cats = load_images(name_cats);
std::vector<int> labels_cats = load_labels(name_cats, 0);

std::vector<std::string> images_dogs = load_images(name_dogs);
std::vector<int> labels_dogs = load_labels(name_dogs, 1);
```

[12]:
```
std::vector<std::string> images_total;

for(auto const& value: images_cats) {
    images_total.push_back(value);
}

for(auto const& value: images_dogs) {
    images_total.push_back(value);
}
```

[13]:
```
std::vector<int> labels_total;

for(auto const& value: labels_cats) {
```

```
    labels_total.push_back(value);
}

for(auto const& value: labels_dogs) {
    labels_total.push_back(value);
}
```

[14]:
```
auto custom_dataset = CustomDataset(images_total, labels_total).map(torch::data:
 ↪:transforms::Stack<>());
auto data_loader = torch::data::make_data_loader<torch::data::samplers::
 ↪RandomSampler>(std::move(custom_dataset), 4);
```

0.8 Loading the pre-trained model

The steps to load the pre-trained model and perform Transfer Learning are listed below:

1. Download the pre-trained model of ResNet18.
2. Load pre-trained model.
3. Change output features of the final FC layer of the model loaded. (Number of classes would change from 1000 - ImageNet to 2 - Dogs vs Cats).
4. Define optimizer on parameters from the final FC layer to be trained.
5. Train the FC layer on Dogs vs Cats dataset
6. Save the model (#TODO)

Let's go step by step.

Step-1: Download the pre-trained model of ResNet18

Thanks to the developers, we do have C++ models available in torchvision (https://github.com/pytorch/vision/pull/728) but for this tutorial, transferring the pre-trained model from Python to C++ using `torch.jit` is a good idea, as most PyTorch models in the wild are written in Python right now, and people can use this tutorial to learn how to trace their Python model and transfer it to C++.)

First we download the pre-trained model and save it in the form of `torch.jit.trace` format to our local drive.

```
# Reference: #TODO- Add Link
from torchvision import models
# Download and load the pre-trained model
model = models.resnet18(pretrained=True)

# Set upgrading the gradients to False
for param in model.parameters():
    param.requires_grad = False

# Save the model except the final FC Layer
resnet18 = torch.nn.Sequential(*list(resnet18.children())[:-1])

example_input = torch.rand(1, 3, 224, 224)
script_module = torch.jit.trace(resnet18, example_input)
script_module.save('resnet18_without_last_layer.pt')
```

We will be using `resnet18_without_last_layer.pt` model file as our pre-trained model for transfer learning.

Step-2: Load the pre-trained model

Let's go ahead and load the pre-trained model using `torch::jit` module. Note that the reason we have converted `torch.nn.Module` to `torch.jit.ScriptModule` type, is because C++ API currently does not support loading Python `torch.nn.Module` models directly.

[15]:
```
torch::jit::script::Module module;
module = torch::jit::load("/Users/krshrimali/Documents/krshrimali-blogs/codes/
 ↪transfer-learning/transfer-learning/build/resnet18_without_lastlayer.pt");
```

0.9 Experimentation

Since we are almost done with defining required functions, let's go ahead and define the optimizer on our last FC layer and train the FC layer on our dataset.

[16]:
```
torch::nn::Linear lin(512, 2); // the last layer of resnet, which we want to␣
 ↪replace, has dimensions 512x1000
torch::optim::Adam opt(lin->parameters(), torch::optim::AdamOptions(1e-3 /
 ↪*learning rate*/));
```

[17]:
```
train(module, lin, data_loader, opt, custom_dataset.size().value());
```

```
Epoch: 0, Accuracy: 0.745, MSE: 0.491316
Epoch: 1, Accuracy: 0.8825, MSE: 0.151102
Epoch: 2, Accuracy: 0.845, MSE: 0.111356
Epoch: 3, Accuracy: 0.8025, MSE: 0.106991
Epoch: 4, Accuracy: 0.885, MSE: 0.0578496
Epoch: 5, Accuracy: 0.865, MSE: 0.0572935
Epoch: 6, Accuracy: 0.9, MSE: 0.0337119
Epoch: 7, Accuracy: 0.855, MSE: 0.0399212
Epoch: 8, Accuracy: 0.865, MSE: 0.0347004
Epoch: 9, Accuracy: 0.8425, MSE: 0.0341781
Epoch: 10, Accuracy: 0.8825, MSE: 0.0239106
Epoch: 11, Accuracy: 0.86, MSE: 0.026371
Epoch: 12, Accuracy: 0.86, MSE: 0.0269321
Epoch: 13, Accuracy: 0.88, MSE: 0.0224421
Epoch: 14, Accuracy: 0.8875, MSE: 0.0172629
```