

# Machine Learning II: Deep Learning and Applications

## Homework 1

Due date: Feb 16

### Instructions

- Make a copy of this notebook in your own Colab and complete the questions there.
- You can add more cells if necessary. You may also add descriptions to your code, though it is not mandatory.
- Make sure the notebook can run through by *Runtime -> Run all*. **Keep all cell outputs** for grading.
- Submit the link of your notebook [here](#). Please **enable editing or comments** so that you can receive feedback from TAs.

Install PyTorch and Skorch.

```
pip install -q torch skorch torchvision torchttext
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import skorch
import sklearn
import numpy as np
import matplotlib.pyplot as plt
```

## 1. Tensor Operations (20 points)

Tensor operations are important in deep learning models. In this part, you are required to implement some common tensor operations in PyTorch.

Tensor operations are important in deep learning models. In this part, we will review some commonly-used tensor operations in PyTorch.

### 1) Tensor squeezing, unsqueezing and viewing

Tensor squeezing, unsqueezing and viewing are important methods to change the dimension of a Tensor, and the corresponding functions are [torch.squeeze](#), [torch.unsqueeze](#) and [torch.Tensor.view](#). Please read the documents of the functions, and finish the following practice.

```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])

# Add two new dimensions to x by using the function torch.unsqueeze, so that the size of x becomes
(3, 1, 2, 1).

# Remove the two dimensions justed added by using the function torch.squeeze, and change the size of
```

```
x back to (3, 2).
```

```
# x is now a two-dimensional tensor, or in other words a matrix. Now use the function
torch.Tensor.view and change x to a one-dimensional vector with size being (6).
```

## 2) Tensor concatenation and stack

Tensor concatenation and stack are operations to combine small tensors into big tensors. The corresponding functions are [torch.cat](#) and [torch.stack](#). Please read the documents of the functions, and finish the following practice.

```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])

# y is a tensor with size being (3, 2)
y = torch.Tensor([[-1, -2], [-3, -4], [-5, -6]])

# Our goal is to generate a tensor z with size as (2, 3, 2), and z[0,:,:] = x, z[1,:,:] = y.

# Use torch.stack to generate such a z

# Use torch.cat and torch.unsqueeze to generate such a z
```

## 3) Tensor expansion

Tensor expansion is to expand a tensor into a larger tensor along singleton dimensions. The corresponding functions are [torch.Tensor.expand](#) and [torch.Tensor.expand\\_as](#). Please read the documents of the functions, and finish the following practice.

```
# x is a tensor with size being (3)
x = torch.Tensor([1, 2, 3])

# Our goal is to generate a tensor z with size (2, 3), so that z[0,:,:] = x, z[1,:,:] = x.

# [TO DO]
# Change the size of x into (1, 3) by using torch.unsqueeze.

# [TO DO]
# Then expand the new tensor to the target tensor by using torch.Tensor.expand.
```

## 4) Tensor reduction in a given dimension

In deep learning, we often need to compute the mean/sum/max/min value in a given dimension of a tensor. Please read the document of [torch.mean](#), [torch.sum](#), [torch.max](#), [torch.min](#), [torch.topk](#), and finish the following practice.

```
# x is a random tensor with size being (10, 50)
x = torch.randn(10, 50)

# Compute the mean value for each row of x.
# You need to generate a tensor x_mean of size (10), and x_mean[k, :] is the mean value of the k-th
row of x.
```

```

# Compute the sum value for each row of x.
# You need to generate a tensor x_sum of size (10).

# Compute the max value for each row of x.
# You need to generate a tensor x_max of size (10).

# Compute the min value for each row of x.
# You need to generate a tensor x_min of size (10).

# Compute the top-5 values for each row of x.
# You need to generate a tensor x_mean of size (10, 5), and x_top[k, :] is the top-5 values of each
row in x.

```

## Convolutional Neural Networks (40 points)

Implement a convolutional neural network for image classification on CIFAR-10 dataset.

CIFAR-10 is an image dataset of 10 categories. Each image has a size of 32x32 pixels. The following code will download the dataset, and split it into `train` and `test`. For this question, we use the default validation split generated by Skorch.

```

train = torchvision.datasets.CIFAR10("./data", train=True, download=True)
test = torchvision.datasets.CIFAR10("./data", train=False, download=True)

```

The following code visualizes some samples in the dataset. You may use it to debug your model if necessary.

```

def plot(data, labels=None, num_sample=5):
    n = min(len(data), num_sample)
    for i in range(n):
        plt.subplot(1, n, i+1)
        plt.imshow(data[i], cmap="gray")
        plt.xticks([])
        plt.yticks([])
        if labels is not None:
            plt.title(labels[i])

train.labels = [train.classes[target] for target in train.targets]
plot(train.data, train.labels)

```

### 1) Basic CNN implementation

Consider a basic CNN model

- It has 3 convolutional layers, followed by a linear layer.
- Each convolutional layer has a kernel size of 3, a padding of 1.
- ReLU activation is applied on every hidden layer.

Please implement this model in the following section. You will need to tune the hyperparameters and fill the results in the table.

#### a) Implement convolutional layers

Implement the initialization function and the forward function of the CNN.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # implement parameter definitions here

    def forward(self, images):
        # implement the forward function here
        return None
```

## b) Tune hyperparameters

Train the CNN model on CIFAR-10 dataset. Tune the number of channels, optimizer, learning rate and the number of epochs for best validation accuracy.

```
# implement hyperparameters here
model = skorch.NeuralNetClassifier(CNN, criterion=torch.nn.CrossEntropyLoss,
                                  device="cuda")
# implement input normalization & type cast here
model.fit(train.data, train.targets)
```

Write down **validation accuracy** of your model under different hyperparameter settings. Note the validation set is automatically split by Skorch during `model.fit()`.

**Hint:** You may need more epochs for SGD than Adam.

#channel for each layer \ optimizer	SGD	Adam
(128, 128, 128)		
(256, 256, 256)		
(512, 512, 512)		

## 2) Full CNN implementation

Based on the CNN in the previous question, implement a full CNN model with max pooling layer.

- Add a max pooling layer after each convolutional layer.
- Each max pooling layer has a kernel size of 2 and a stride of 2.

Please implement this model in the following section. You will need to tune the hyperparameters and fill the results in the table. You are also required to complete the questions.

### a) Implement max pooling layers

Copy the CNN implementation in previous question. Implement max pooling layers.

```
class CNN_MaxPool(nn.Module):
    def __init__(self):
        super(CNN_MaxPool, self).__init__()
        # implement parameter definitions here
```

```
def forward(self, images):
    # implement the forward function here
    return None
```

## b) Tune hyperparameters

Based on best optimizer you found in the previous problem, tune the number of channels and learning rate for best validation accuracy.

```
# implement hyperparameters here
model = skorch.NeuralNetClassifier(CNN_MaxPool,
                                  criterion=torch.nn.CrossEntropyLoss,
                                  device="cuda")
# implement input normalization & type cast here
model.fit(train.data, train.targets)
```

Write down the **validation accuracy** of your model under different hyperparameter settings.

#channel for each layer	validation accuracy
(128, 128, 128)	
(128, 256, 512)	
(256, 256, 256)	
(256, 512, 1024)	
(512, 512, 512)	
(512, 1024, 2048)	

For the best model you have, test it on the test set.

It is fine if you found some hyperparameter combination better than those listed in the tables.

```
# implement the same input normalization & type cast here
test.predictions = model.predict(test.data)
sklearn.metrics.accuracy_score(test.targets, test.predictions)
```

How much **test accuracy** do you get?

**Your Answer:**

What can you conclude for the design of CNN structure?

**Your Answer:**

## Recurrent Neural Networks (40 points)

Next, let's use PyTorch to implement a recurrent neural network for sentiment analysis, i.e., classifying sentences into given sentiment labels, including positive, negative and neutral.

We use a benchmark dataset (i.e., SST) for this task. First, let's download the SST dataset, and do some preprocessing to build vocabulary and split the dataset into training/validation/test sets. Also, let's define the training and evaluation function. Please do not modify the functions.

```

import copy
import torch
from torch import nn
from torch import optim
import torchtext
from torchtext import data
from torchtext import datasets

TEXT = data.Field(sequential=True, batch_first=True, lower=True)
LABEL = data.LabelField()

# load data splits
train_data, val_data, test_data = datasets.SST.splits(TEXT, LABEL)

# build dictionary
TEXT.build_vocab(train_data)
LABEL.build_vocab(train_data)

vocab_size = len(TEXT.vocab)
label_size = len(LABEL.vocab)
padding_idx = TEXT.vocab.stoi['<pad>']
embedding_dim = 128
hidden_dim = 128

# build iterators
train_iter, val_iter, test_iter = data.BucketIterator.splits(
    (train_data, val_data, test_data),
    batch_size=32)

# train a model
# DO NOT MODIFY
def train(model, iterator, optimizer, criterion):
    total_loss, total_correct, total_prediction = 0.0, 0.0, 0.0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        logits = model(batch.text.cuda())
        predictions = torch.max(logits, dim=-1)[1]
        loss = criterion(logits, batch.label.cuda())
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        total_correct += torch.eq(predictions, batch.label.cuda()).sum().item()
        total_prediction += batch.label.size(0)
    return total_loss / len(iterator), total_correct / total_prediction

# evaluate a model
# DO NOT MODIFY
def evaluate(model, iterator, criterion):
    total_loss, total_correct, total_prediction = 0.0, 0.0, 0.0

```

```

model.eval()
with torch.no_grad():
    for batch in iterator:
        logits = model(batch.text.cuda())
        predictions = torch.max(logits, dim=-1)[1]
        loss = criterion(logits, batch.label.cuda())

        total_loss += loss.item()
        total_correct += torch.eq(predictions, batch.label.cuda()).sum().item()
        total_prediction += batch.label.size(0)
    return total_loss / len(iterator), total_correct / total_prediction

```

Next, we are ready to build our RNN model for sentiment analysis. In the following codes, we have provided several hyperparameters we needed for building the model, including vocabulary size (`vocab_size`), the word embedding dimension (`embedding_dim`), the hidden layer dimension (`hidden_dim`), the number of layers (`num_layers`) and the number of sentence labels (`label_size`). Please fill in the missing codes, and implement an LSTM model.

```

class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, label_size, padding_idx):
        super(RNNClassifier, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.label_size = label_size
        self.num_layers = 1

        # add the layers required for sentiment analysis.

    def zero_state(self, batch_size):
        # implement the function, which returns an initial hidden state.
        return None

    def forward(self, text):
        # implement the forward function of the model.
        return None

```

Finally, we are ready to train the model and compute the accuracy.

```

model = RNNClassifier(vocab_size, embedding_dim, hidden_dim, label_size, padding_idx)
# tune the optimizer type and hyperparameters here
optimizer = optim.SGD(model.parameters(), lr=1e-4)
criterion = nn.CrossEntropyLoss()
model.cuda()
criterion.cuda()

# train and test the model
# DO NOT MODIFY
best_valid_acc = 0.0
best_state_dict = copy.deepcopy(model.state_dict())
for epoch in range(20):
    train_loss, train_acc = train(model, train_iter, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, val_iter, criterion)

```

```

print('Epoch {} | Train loss {:.3f} | Valid loss {:.3f} | Valid acc {:.3f}'.format(epoch,
train_loss, valid_loss, valid_acc))

if valid_acc > best_valid_acc:
    best_valid_acc = valid_acc
    best_state_dict = copy.deepcopy(model.state_dict())

```

Once we find the best hyperparameters for the validation set, we can now evaluate our model on the test set.

```

model.load_state_dict(best_state_dict)
test_loss, test_acc = evaluate(model, test_iter, criterion)
print('Test loss {:.3f} | Test acc {:.3f}'.format(test_loss, test_acc))

```

## 1) Implement the RNN model

The current codes of the RNN model are not complete, so let's first complete the codes to implement a standard RNN model by filling in the [block](#).

- **Subtask 1-1: Creating all the Required Layers in Your Model**

Remember that when building a deep learning model, we first need to complete the **init** function by creating all the required layers. In our case, since we are using RNNs for sentence classification, we need an embedding layer to transform words into word embeddings, a RNN layer to transform word embeddings into sentence encodings, an activation function, and a linear layer as well as a softmax function for sentence classification.

Based on that, please create all the necessary layers of your RNN model in the **init** function.

- **Subtask 1-2: Implementing the Function for Initializing Hidden States**

Remember that when applying a RNN unit to transform word embeddings into sentence encodings, the RNN unit starts from an initial hidden vector with all zero values, and sequentially read each word to update the hidden vector. Finally, the hidden vector obtained after reading the last word is treated as the sentence encoding.

In this step, please implement the **zero\_state** function, which returns a batch of initial hidden vectors given a batch size. Hint: your function should return a tensor with all the values being zero, and you may refer to the [official document](#) for the correct shape of the tensor.

- **Subtask 1-3: Implementing the Forward Function**

Finally, we are ready to build the forward function, which takes a batch of sentences as inputs and returns the a batch of logits. To be more specific, the input is given by the tensor called  $\text{text}$ , and the size of the tensor is  $(B, L)$ , with  $B$  being the batch size,  $L$  being the maximum length of sentences in this batch and  $\text{text}[i, j]$  being the interger id of the  $j$ -th word in the  $i$ -th sentence. Given this tensor as input, your forward function should return a logit tensor of size  $(B, C)$ , with  $B$  being the batch size and  $C$  being the number of possible classes.

Please implement the forward function based on the above instructions.

## 2) Compare Different Optimizers

In the previous task, we have implemented a RNN model for sentiment analysis, or more generally sentence classification.



To better understand several concepts in deep learning, let's do some ablation studies by using the model we have just implemented.

The first task is to try different optimizers for your model, where for each optimizer, you may also try different options of learning rate.

- **Subtask 2-1: Completing the Table**

We have provided the following table for different combinations of optimizers and learning rate, please write down the **validation accuracy** of your model with different optimizers and learning rates.

	0.1	0.01	0.001	0.0001
SGD				
Adam				
RMSprop				

- **Subtask 2-2: Explaining your Observations**

Based on your results, briefly explain your observations, e.g., which optimizer works the best, what is the optimal learning rate for each optimizer?

*Your Answer:*

### 3) Compare the Results under Different Epoches

In this task, we hope to compare the results of our model under different training epoches, and answer a question.

- **Subtask 3-1: Completing the Table**

We have provided the following table, please write down the **training accuracy** and **validation accuracy** of your model under different epoches.

	10	20	30	40	50
Training Accuracy					
Validation Accuracy					

- **Subtask 3-2: Answering the Question**

Is it always better to train a model for more epoches? How can we decide when should we stop training?

*Your Answer:*

### 4) Compare Different Model Capacities/Configurations

In practice, we may also vary the capacity of our model to find the optimal choice. In this part, please try different configurations of your model, which have different model capacities. Based on your observation, please also answer a question.

- **Subtask 4-1: Completing the Table**

Please write down the **validation accuracy** of your model under different model capacities (i.e., specified by the word embedding dimension and the hidden layer dimension).

Embedding dim / Hidden dim	64	128	256
----------------------------	----	-----	-----

Embedding dim / Hidden dim	64	128	256
64			
128			
256			

- **Subtask 4-2: Answering the Question**

Is it always better to increase model capacities in this case? Is it always better to increase model capacities in general? How to decide a proper model capacity in practice?

*Your Answer:*