know where to look for

- http://neuralnetworksanddeeplearning.com/ Chapter 1
- d2l.ai Chapter 3 and Chapter 4
- https://numpy.org/doc/

Last lecture:

Takeaway for the last lecture:

- logistic regression: model which computes inner product $+b$ with one output, then combines it with the logistic sigmoid
- a possible loss: binary cross entropy - the neg-log of the output probability for the ground truth class y for a given sample (x, y)
- neg-log (close to 0) = close to ∞, $-\ln(1.0) = 0$
- binary cross entropy - derived from maximum likelihood principle to observe the ground truth labels
- logistic regression: 1-layer NN with sigmoid activation function

The key you need to understand sigmoid activation and cross entropy loss, is how the exponential and the (neg) logarithm look like.

What you will see today

- Pytorch basic structures: Tensors and their properties
- Basic operations with tensors
- Broadcasting - Important!!!
- PyTorch basic boilerplate code: logistic regression code in PyTorch for FashionMNIST
- DataSet Class

Outline

Readings

- Installation https://pytorch.org/get-started/locally/
- quick intro: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- cheat sheet: https://pytorch.org/tutorials/beginner/ptcheat.html

```
import torch
import torchvision #for models, dataloading utils, data transforms
```

Package	Description
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

Just a collection of numbers, which is indexable along several axes

```
a[2]
a[i,j]
a[c,h,w]
```

Tensor? (in quizzes)

Same as in physics lectures

- 1-tensor: object/array with 1-dimensional way to index it, vector
  $a[i] \leftrightarrow a_i$
- 2-tensor: object/array with 2-dimensional way to index it, matrix
  $a[i, k] \leftrightarrow a_{i,k}$
- 3-tensor: object/array with 3-dimensional way to index it
  $a[i, k, l] \leftrightarrow a_{i,k,l}$
- n-tensor: object/array with n-dimensional way to index its numbers
- Tensor in pytorch:

  RWB: a representation of an array-like structure A_i, $A_{i,j,k}$, $A_{i,j,k}$ or $A_{i_1,\ldots,i_n}$ with benefits (for storing computed gradients).

see the pdf from week1 exercises

```
a= torch.rand((3)) #1-tensor, vector
#a[i]
b= torch.rand((2,4))  #2-tensor, matrix
#b[i,k]
c= torch.rand((7,5,3)) #3-tensor
#c[i,k,l]
```

A tensor

- an ordered set of numbers which is indexable
  - 1-tensor – vector
  - 2-tensor – matrix
  - k-tensor – generalization of vectors and matrices to more indexable dimensions

see the pdf from week1 exercises
A tensor

- an ordered set of numbers which is indexable
- it has a shape vector (indexable dimensions, counts per dim)
- it has a dtype
- it has a device placement

```
c= torch.rand((7,5,3)) #3-tensor
print(c.shape) #[7,5,3]
print(c.dtype)
print(c.device)
```

see the pdf from week1 exercises

- change tensor dtype and device placement

```
c= torch.rand((7,5,3)) #3-tensor
e=c.to(torch.float64) # DOUBLE IS bad and slow on most GPUs
cuda0 = torch.device('cuda:0')
f=c.to(cuda0)
g=c.to(f)#changes dtype and device

res= tensor1.to(othertensor2)
```

- with fixed values:

```
x= torch.zeros((5,1))
y= torch.ones((5))
z= torch.empty((3,2,3))
a = torch.empty((64,32,3,3)).fill_(32.) # initializes to some value
b= a.new_full((3,2),42.) # with same type and device as tensor a
c = torch.full((2, 3), 3.141592)
d = torch.randn((2, 3))
```

- from a saved tensor:
  https://pytorch.org/docs/stable/generated/torch.save.html
  https://pytorch.org/docs/stable/generated/torch.load.html

see the pdf from week1 exercises

- Tensor vs numpy array

```
#to numpy
c= torch.rand((7,5,3)) #3-tensor
np_c = c.data.cpu().numpy()
print(type(np_c))

#to PyTorch
e= np.ones((3,5))
f= torch.tensor(e) #copies
print(f.shape)
f2= torch.from_numpy() # shares mem, SIDE EFFECTS
```

- from numpy:

```
a=np.random.normal(5,size=(2,3)).astype('float32')
x=torch.tensor(a) # this copies data
x2=torch.as_tensor(a) # this does NOT COPY data,
#and does nothing if its already a tensor with right type, etc.
x3=torch.from_numpy(a) # this does NOT COPY data
#when this can be inappropriate? not resizable tensor as limitation
```

- to numpy:

```
nparr = a.data.cpu().numpy() # a.cpu().numpy() works only if it has no grad field!!
```

```
x=torch.ones((10))
y=x.view((2,5))
z=x.view((-1,5)) #-1 joker
```

- Be careful: Which elements ends up where in this case?

```
x=torch.ones((4,2,3))
y=x.view((-1,12))
```

```
print(a.device)
b= a.to('cuda:0')
```

Important knowledge: on multi-GPU systems without job managers restrict yourself to one device, dont grab all GPUs!

```
CUDA_VISIBLE_DEVICES=1 python3 blablascript.py
```

This uses GPU 1 from the output of nvidia_smi

Outline

Exercise will be on broadcasting. Its important for coding.

```
a= torch.full((2,3),3.)
b= torch.full((5,1,3),3.)
c= a+b
```

What will `c.shape` be ?

https://pytorch.org/docs/stable/notes/broadcasting.html

same holds for many other element-wise binary operators like $-, *, /, ** n$

$$
\begin{aligned}
a &= torch.ones((4)) \\
b &= torch.ones((1,4)) \\
&\quad torch.add(a,b) && \rightarrow (1,4) \\
a &= torch.ones((4)) \\
b &= torch.ones((4,1)) \\
&\quad torch.add(a,b) && \rightarrow (4,4)!!! \\
a &= torch.ones((3)) \\
b &= torch.ones((4,1)) \\
&\quad torch.add(a,b) && \rightarrow (4,3) \\
a &= torch.ones((3)) \\
b &= torch.ones((1,4)) \\
&\quad torch.add(a,b) && \rightarrow ERR
\end{aligned}
$$

- smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

Why do we need broadcasting?

- it is **MUCHHH** faster than nested for-loops
- avoid for loops in favor of broadcasting or built-in torch operations whenever possible
- for loops are good for test cases to ensure that code works correct, less suitable for serious model training or inference

for-loops = inexperienced coder (or very sleepy prof needs exercises to get done ;))

1– the smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

2– then check whether they are compatible – they are incompatible if in one dimension both tensors have sizes > 1 which are not equal. if they are incompatible, you will get an error.

3– whenever a dimension with size 1 meets a dimension with a size $k > 1$, then the smaller vector is replicated/copied $k - 1$ times in this dimension until he reaches in this dimension size k and your actual operation is applied

Example:

start	after insert	after copying
(4,1)	(4,1)	(4,4)
(4)	(1,4)	(4,4)

More examples:

start	after insert	after copying
(1,3)	(1,3)	(1,3)
(3)	(1,3)	(1,3)

start	after insert	after copying
(2,3)	(1,2,3)	(5,2,3)
(5,1,3)	(5,1,3)	(5,2,3)

start	after insert	after copying
(1,7)	(1,1,1,7)	(5,2,3,7)
(5,2,3,7)	(5,2,3,7)	(5,2,3,7)

start	after insert	after copying
(4,1)	(1,4,1)	ERR
(2,3,7)	(2,3,7)	ERR

if broadcasting is too mind-boggling (why you dont do an exercise to have your heart beat faster), then apply `.unsqueeze(dim)` on your tensor, until both tensors have the same number of dimension axes. The only thing what is done then, is copying along $dim = 1$ axes.

Outline

`torch.mm(a,b)` a, b both 1-tensors: dot product, not broadcasting.

$$a.size() = (d), \;\; b.size() = (d)$$
$$torch.dot(a, b) = a \cdot b = \sum_i a_i b_i = \sum_i a[i]b[i]$$
$$\rightarrow torch.dot(a, b).size() = () \text{ a scalar!}$$

`torch.mm(A,B)` A, B both 2-tensors: matrix multiplication, not broadcasting.

$$A.size() = (i, k), \;\; B.size() = (k, l)$$
$$torch.mm(A, B)[i, l] = \sum_{k'} A_{i,k'} B_{k',l} = \sum_k A[i, k']B[k', l]$$
$$\rightarrow torch.mm(A, B).size() = (i, l)$$

`torch.bmm(A,B)` **batched** matrix multiplication, not broadcasting. A, B must be 3-tensors. multiplication along last dim of A and second dim of B.

$$A.size() = (b, i, k), \;\; B.size() = (b, k, l)$$
$$torch.bmm(A, B)[b, i, l] = \sum_k A_{b,i,k} B_{b,k,l} = \sum_k A[b, i, k] B[b, k, l]$$
$$\rightarrow torch.bmm(A, B).size() = (b, i, l)$$

`torch.bmm(A,B)` performs for every index k a matrix multiplication between $A[k, :, :]$ and $B[k, :, :]$
– its a for loop over k of `torch.mm(A[k,:,:], B[k,:,:])`

Think: $torch.bmm(A, B)$ given a known shape of A puts what restrictions on B??

example: want to compute matrix vector product by `mm(...)`: $(vA)_l = \sum_k^K v_k A_{k,l}$,
however: $v.shape = (K)$.

v is 1-tensor, cannot use `torch.mm(`v, A`)`.
Solution: add a dim in v first:

$$v = v.unsqueeze(0) \quad (K) \to (1, K)$$

This is now a 2-tensor, and can use `torch.mm(...)` on it.

`torch.squeeze(A,dim=2)` - remove singleton dim $(a, b, 1, c) \to (a, b, c)$
`torch.unsqueeze(A,dim=1)` - insert singleton dim $(a, b, c) \to (a, 1, b, c)$
`torch.unsqueeze(A,dim=0)` - insert singleton dim $(a, b, c) \to (1, a, b, c)$

example: want to compute matrix vector product by `mm(...)`: $(vA)_l = \sum_k^K v_k A_{k,l}$,
however: $v.shape = (K)$.

v is 1-tensor, cannot use `torch.mm(`v, A`)`. add a dim in v first:

$torch.mm(v.unsqueeze(0), A)$ $(1, K) \cdot (K, L) \to (1, L)$

$torch.mm(v.unsqueeze(0), A).squeeze(0)$ $(1, K) \cdot (K, L) \to (1, L) \to (L)$

`torch.squeeze(A,dim=2)` - remove singleton dim $(a, b, 1, c) \to (a, b, c)$
`torch.unsqueeze(A,dim=1)` - insert singleton dim $(a, b, c) \to (a, 1, b, c)$
`torch.unsqueeze(A,dim=0)` - insert singleton dim $(a, b, c) \to (1, a, b, c)$

Two ways:

```
a = np.empty((3,5,7))
print(a.shape)
a=a[:,np.newaxis,:,:] # cumbersome if too many axes
print(a.shape)
#OR
a = np.empty((3,5,7))
a=  np.expand_dims(a, axis=1 )
print(a.shape)
a=  np.squeeze(a, axis=1 ) # the inverse operation
print(a.shape)
```

...

`torch.transpose(A,dim1,dim2)` swaps two dimensions

`torch.Tensor.permute(*dims)` permutes a set of dimensions rather than just swapping two

Basic code example: `fmnist_pytorch_logreg_class.py`

The computational flow is as follows:

1. define dataset class and dataloader class
2. define prediction model
3. define loss
4. define an optimizer
5. initialize model parameters, usually when model gets instantiated
6. Loop over epochs. every epoch has a train and a validation phase

What does one need to run any model in pytorch?

Loop over epochs. every epoch has a train and a validation phase

1. train phase: loop over minibatches of the training data
    1. set model to train mode
    2. fetch input and ground truth tensors, move them to a device (usually cpu or cuda for GPU)
    3. compute model prediction
    4. compute loss
    5. set accumulated gradients of model parameters to zero
    6. run `loss.backward()` to compute gradients of the loss function with respect to parameters
    7. run optimizer to apply gradients to update model parameters
2. validation phase: loop over minibatches of the validation data
    1. set model to evaluation mode
    2. fetch input and ground truth tensors, move them to a device
    3. compute model prediction
    4. compute loss in minibatch
    5. compute loss averaged/accumulated over all minibatches
    6. if averaged loss is lower than the best loss so far, save the state dictionary of the model containing the model parameters
3. return best model parameters

We will go these things through in `fmnist_pytorch_logreg_class.py`

```
torch.manual_seed(3)
```

- not a course in magic !
  https://en.wikipedia.org/wiki/The_Alchemist_Discovering_Phosphorus#/media/File:Joseph_Wright_of_Derby_The_Alchemist.jpg
- make your experiment reproducible science
  - seed all used random generators (numpy, cuda, ...). Be aware which routines might be not deterministic.

```
#parameters
batchsize=32
maxnumepochs=3

device=torch.device("cpu") #or  device=torch.device("cuda:0")
```

- samples on GPUs are processed in parallel in minibatches
- epochs: number of times we iterate through the data set for training

```
datatransforms = transforms.Compose(
[
  transforms.ToTensor(),
  transforms.Normalize((0.1307,), (0.3081,))
])
```

- `transforms.ToTensor()` converts the output of the DataSet class from `PIL.Image` to a PyTorch Tensor
- `transforms.Normalize((0.1307,), (0.3081,))` - subtracts every RGB subpixel a mean per channel $m[c]$ and divides by a standard deviation $s[c]$ per channel

$$img[b, c, h, w] = \frac{img[b, c, h, w] - m[c]}{s[c]}$$

reason: this makes training outcomes more stable if data is on a limited scale around 0, and gradients are more easily centered around 0

  - More in a separate lecture on Data Augmentations

```
ds_trainval = datasets.FashionMNIST('./data', train=True, download=True,
transform=datatransforms),
ds_test =  datasets.FashionMNIST('./data', train=False, download=True,
transform=datatransforms)
```

- `datasets.FashionMNIST( ... )` defines an instance of a map-style DataSet class https://pytorch.org/docs/stable/data.html#dataset-types
- iterable-style Datasets: provides a data sample using a python iterator, suitable for streaming or otherwise dynamic data sources (e.g. Kafka topics, influxdata telegraf, database queries)
- map-style Datasets, provides a datasample using by asking for the i-th data point. Suitable for static data with a fixed dataset size, e.g. benchmark datasets, images on disk.
- why the code has a 'trainval' : ?

- assumes that our data is static and we have a definition of the 5-th, 690-th, 1002-nd data sample.

A class derived from `torch.utils.DataSet` which implements two functions:

```
class ds(torch.utils.DataSet):

  def __init__(self,...):
    super(self).__init__()
    # store dataset filepath on your disk, some parameters
    # store transforms as a class member, so that they can be used in  __getitem__
    # store filenames often practical
    pass

  def __len__(self):
    # returns the number of samples in the dataset
    pass

  def __getitem__(self,i):
    # returns the i-th sample and its label, and additional data
    pass
```

- assumes that our data is static and we have a definition of the 5-th, 690-th, 1002-nd data sample.

A class derived from `torch.utils.DataSet` which implements two functions:

```
class ds(torch.utils.DataSet)"

  def __init__(self,...):
    super(self).__init__()
    pass

  def __len__(self):
    # returns the number of samples in the dataset
    pass

  def __getitem__(self,i):
    # returns the i-th sample and its label, and additional data
    pass
```

- sometimes `__len__` is not the whole dataset size, e.g. when one has very imbalanced class counts. Then might take all from the smallest class, and a subset of the larger classes
- `__getitem__` does all the processing (loading an image, transforming it into RGBA, resizing images), including calling necessary PyTorch transforms.
  - additional data: case of medical test data: might want to know the filename or patient id for error inspection!
  - consider the case of WSIs in histopathology: possibly subsampling a cell-rich region for training
  - return type can be a dictionary

```
dl_train = torch.utils.data.DataLoader(ds['trainval'], batch_size=batchsize, shuffle=False, sampler= ... )

 dl_val= torch.utils.data.DataLoader(ds['trainval'], batch_size=batchsize, shuffle=False, sampler= ... )

dl_test=  torch.utils.data.DataLoader(ds['test'], batch_size=batchsize, shuffle=False)
```

- role: aggregates samples into minibatches for batched processing.
  - usually `shuffle=True` necessary for the TrainingData set, here it is set to False due to the usage of a SubsetRandomSampler
- pytorchs dataset and dataloader are convenience interfaces
- one could use ones own dataloader instead of pytorchs dataset and dataloader. Care must be taken in case of multithreading (not to load the same data twice)

```
# model
model = onelinear(indims,numcl).to(device)

#loss for training of the model!
loss = torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction

lrates=[0.01, 0.001]

best_hyperparameter= None
weights_chosen = None
bestmeasure = None

for lr in lrates: # try a few learning rates

  print('\n\n\n####################NEW RUN###################')
  print('############################################')
  print('############################################')
```

- loss: the loss function used to train the model by measuring the deviation between prediction and ground truth, and then computing the gradient of the deviation

```
#optimizer here, because of lr
optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9) # which parameters to optimize during tra

# train on train and eval on val data
best_epoch, best_perfmeasure, bestweights = train_modelcv(dataloader_cvtrain = dataloaders['train'],
dataloader_cvtest = dataloaders['val'] ,  model = model ,  criterion = loss , optimizer = optimizer,
scheduler = None, num_epochs = maxnumepochs , device = device)
```

- `optimizer`: applies the computed gradients to change the trainable parameters of the model. Details on optimizers in a later lecture.
- `train_modelcv` : trains one epoch and measures the loss/performance using validation data
- IMPORTANT: we do NOT evaluate on test data before the final mapping has been chosen. For that reason we use validation data at the end of an epoch:
  `dataloader_cvtest = dataloaders['val']`

```
for lr in lrates:

#train and eval code here omitted

  if best_hyperparameter is None:

    best_hyperparameter = lr
    weights_chosen = bestweights
    bestmeasure = best_perfmeasure

  elif best_perfmeasure > bestmeasure:

    best_hyperparameter = lr
    weights_chosen = bestweights
    bestmeasure = best_perfmeasure

# end of for loop over hyperparameters here!
model.load_state_dict(weights_chosen)

accuracy,_ = evaluate(model = model , dataloader  = dataloaders['test'], criterion = None, device = device)

print('accuracy val',bestmeasure.item() , 'accuracy test',accuracy.item()  )
```

- selects the weights for the best hyperparameter, loads them after trying all hyperparameters,
- one final evaluation

```
best_measure = 0
best_epoch =-1

for epoch in range(num_epochs):
  print('Epoch {}/{}'.format(epoch, num_epochs - 1))
  print('-' * 10)

  losses = train_epoch(model,  dataloader_cvtrain,  criterion,  device , optimizer )
  #scheduler.step()
  measure,_ = evaluate(model, dataloader_cvtest, criterion = None, device = device)

  print(' perfmeasure', measure.item() )

  # store current parameters because they are the best or not?
  if measure > best_measure: # > or < depends on higher is better or lower is better?
    bestweights= model.state_dict()
    best_measure = measure
    best_epoch = epoch
    print('current best', measure.item(), ' at epoch ', best_epoch)

return best_epoch, best_measure, bestweights
```

- iterates over all epochs. Within one epoch
  - train on train data in `train_epoch`
  - evaluate on validation data in `evaluate`
- for slow big models: save current best parameters to disk after each epoch

```
def train_epoch(model,  trainloader,  criterion, device, optimizer ):

    model.train() # IMPORTANT!!!

    losses = []
    for batch_idx, data in enumerate(trainloader):

        inputs=data[0].to(device)
        labels=data[1].to(device)


        outputs = model(inputs)
        loss = criterion(outputs, labels)

        optimizer.zero_grad() #reset accumulated gradients
        loss.backward() #compute new gradients
        optimizer.step() # apply new gradients to change model parameters

        losses.append(loss.item())
        if batch_idx%100==0:
          print('mn',np.mean(losses))

    return losses
```

- iterates over all minibatches of the dataloader

Important in training

Put the model into training mode.

Important in Validation/Testing

Put the model into Eval mode.

Error source

- some neural network layers (later lecture) like `BatchNorm` and `Dropout` behave in a randomized or batch-dependent fashion in training mode.
- using training mode for evaluation results in unusable/invalid predictions

The actual evaluation for one epoch

```
model.eval() # IMPORTANT!!!

with torch.no_grad(): # do not record computations for computing the gradient

  datasize = 0
  accuracy = 0
  avgloss = 0
  for ctr, data in enumerate(dataloader):

      #print ('epoch at',len(dataloader.dataset), ctr)

      inputs = data[0].to(device)
      outputs = model(inputs)

      labels = data[1]

      # computing some loss
      cpuout= outputs.to('cpu')
      if criterion is not None:
        curloss = criterion(cpuout, labels)
        avgloss = ( avgloss*datasize + curloss ) / ( datasize + inputs.shape[0])

      # for computing the accuracy
      labels = labels.float()
      _, preds = torch.max(cpuout, 1) # get predicted class
      accuracy =  (  accuracy*datasize + torch.sum(preds == labels) ) / ( datasize + inputs.shape[0])

      datasize += inputs.shape[0] #update datasize used in accuracy comp

if criterion is None:
  avgloss = None

return accuracy, avgloss
```

- iterates over all minibatches of the dataloader

```
model.eval() # IMPORTANT!!!

with torch.no_grad(): # do not record computations for computing the gradient

  datasize = 0
  accuracy = 0
  avgloss = 0
  for ctr, data in enumerate(dataloader):

      #print ('epoch at',len(dataloader.dataset), ctr)

      inputs = data[0].to(device)
      outputs = model(inputs)

      labels = data[1]

      # computing some loss
      cpuout= outputs.to('cpu')
      if criterion is not None:
        curloss = criterion(cpuout, labels)
        avgloss = ( avgloss*datasize + curloss ) / ( datasize + inputs.shape[0])

      # for computing the accuracy
      labels = labels.float()
      _, preds = torch.max(cpuout, 1) # get predicted class
      accuracy =  (  accuracy*datasize + torch.sum(preds == labels) ) / ( datasize + inputs.shape[0])

      datasize += inputs.shape[0] #update datasize used in accuracy comp

if criterion is None:
  avgloss = None

return accuracy, avgloss
```

- IMPORTANT `model.eval()`
- **`with`** `torch.no_grad()` to not record ops/data over eval data for gradient computation

- `outputs = model(inputs)` computes the model prediction (training time: possibly randomized/altered by `BatchNorm` and `Dropout` layers)
- This calls the `def forward(self,x):` of a model:

```
class onelinear(torch.nn.Module):
  def __init__(self,dims, numout):

    super().__init__() #initialize base class

    self.bias=torch.nn.Parameter(data=torch.zeros(numout), requires_grad=True)
    # random init shape must be (dims,numout), requires_grad to True
    self.w=torch.nn.Parameter(data=torch.randn( (dims,numout) ), requires_grad=True)

  def forward(self,x):
    # compute the prediction over batched input x

    v=x.view((-1,28*28)) # flatten the image to (batchsize,dims), -1 allows to guess the number of elements
    y=self.bias+ torch.mm(v,self.w)
    return y
```

- tensors which are trainable parameters (= could be adapted in training), must be wrapped in `torch.nn.Parameter(...)`
- this is done for most builtin layers like `torch.nn.Linear()`, `torch.nn.Conv2d()`
- `model.parameters()` returns them all via an python iterator

Important

- Put the model into training mode when training.
- Put the model into Eval mode when computing predictions on validation or test data.
- shuffle your training data (via dataloader)
- use `with torch.no_grad():` when computing predictions on validation or test data (counterexample: you want to create adversarial samples or compute explanations, then you need the gradients on test data).

```python
def train_epoch(model,  trainloader,  criterion, device, optimizer ):

    model.train() # IMPORTANT!!!

    losses = []
    for batch_idx, data in enumerate(trainloader):

        inputs=data[0].to(device)
        labels=data[1].to(device)


        outputs = model(inputs)
        loss = criterion(outputs, labels)

        optimizer.zero_grad() #reset accumulated gradients
        loss.backward() #compute new gradients
        optimizer.step() # apply new gradients to change model parameters

        losses.append(loss.item())
        if batch_idx%100==0:
          print('mn',np.mean(losses))

    return losses
```