CIS 6930 Topics in Computing for Data Science
Week 1b: Deep Learning Basics (2)

9/9/2021

Yoshihiko (Yoshi) Suhara

2pm-3:20pm & 3:30pm-4:50pm (5-6pm office hour)

Deep Learning Basics (2)

- Part 1: Deep Learning as Building Blocks
  - Model architectures
  - Optimizers
  - Activation functions
- Part 2: PyTorch Basics
- Part 3: (Hands-on session) The first PyTorch code

Part I: Deep Learning as Building Blocks

Basic Deep-Learning Building Blocks

- **(A) Model Architecture + (B) Training Strategy + (c) Data**
- i.e., What to Optimize and How to Optimize

Model Architecture

Multi-layer Neural Networks

AutoEncoders

Convolutional Neural Networks

Recurrent Neural Networks

Transformers

+

Training Strategy

Batch sampling
- **Batch size**

Optimizers
- **SGD**
- **Adam**
- **...**

Learning scheduler
Regularization
Early Stopping
...

+

Data

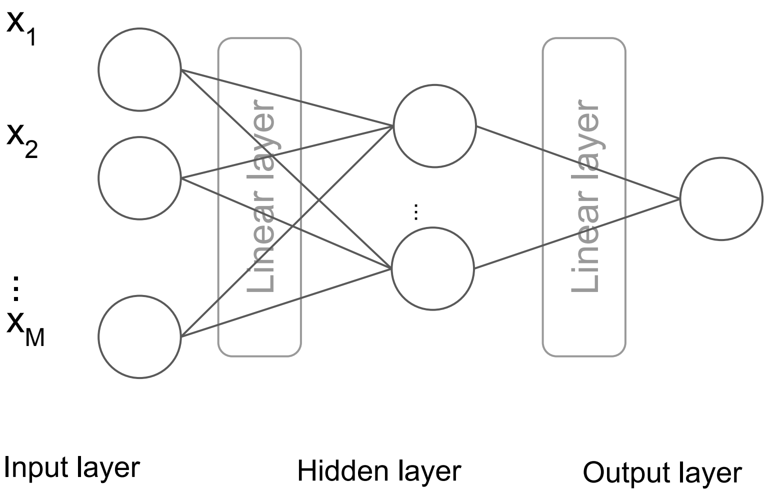
Deep-Learning Building Blocks: Starter Kit

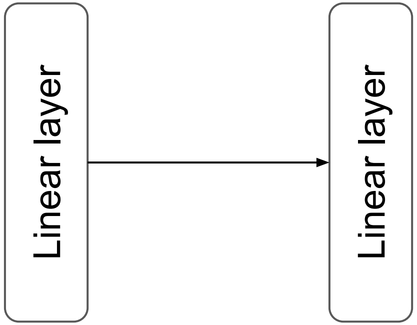
- Layers
  - Linear Layer
- Activation functions
  - Logistic sigmoid, tanh
  - ReLU, Leaky ReLU
- Optimizers
  - SGD w/wo Momentum)
  - Adam

Layers

Fully-connected Layer (aka Linear Layer)

- input dimension size & output dimension size (i.e., the shape of **W**)

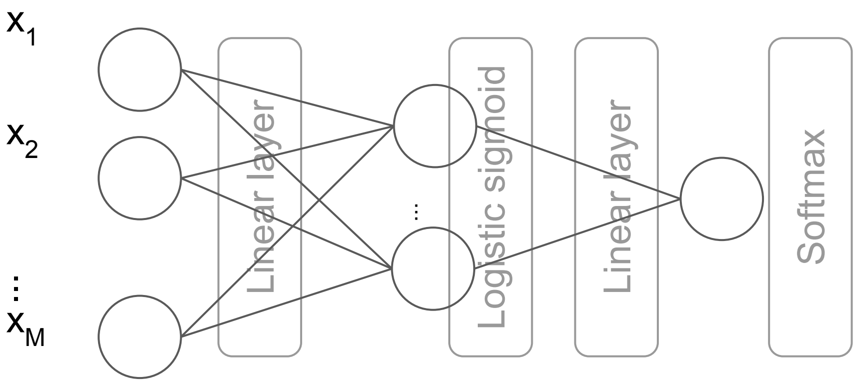




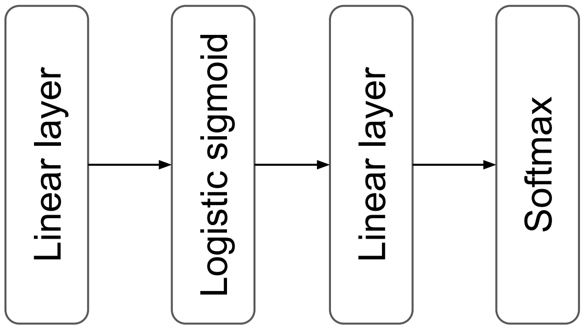
The same model architecture

Fully-connected Layer (aka Linear Layer)

- input dimension size & output dimension size (i.e., the shape of **W**)



Input layer Hidden layer Output layer

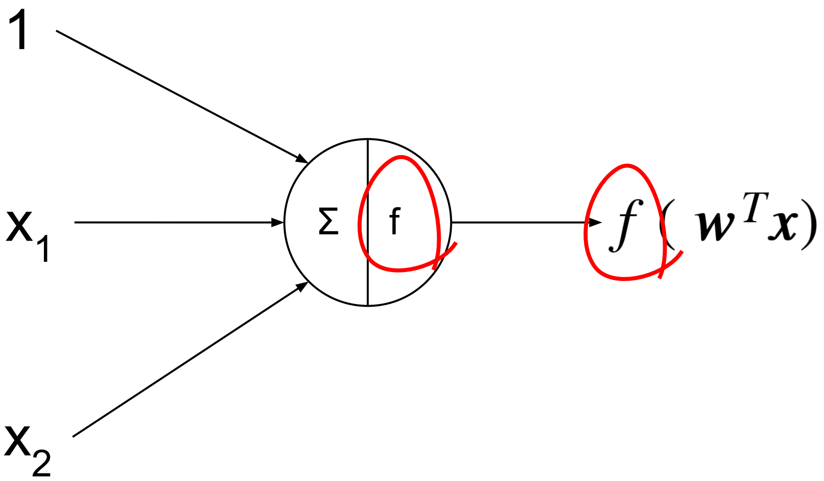


The same model architecture
(with activation functions)

Activation functions

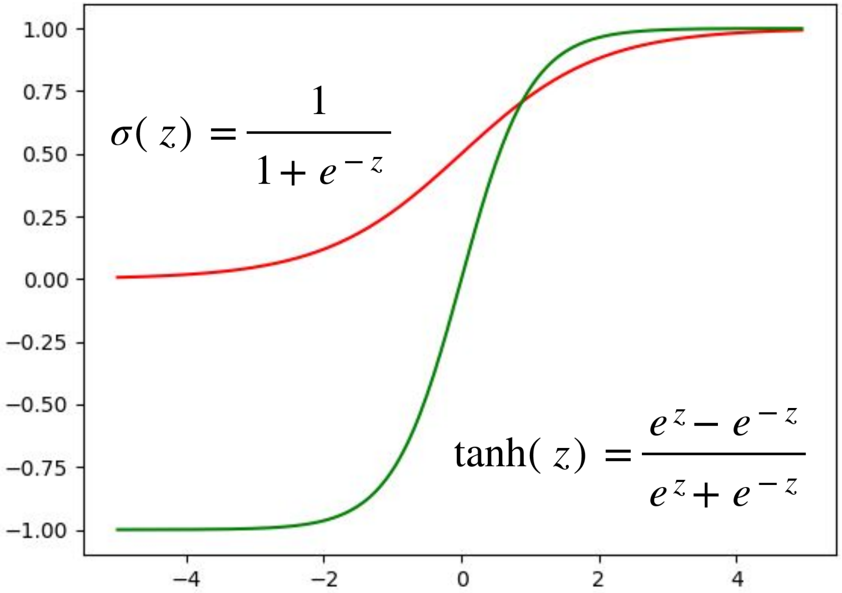
Why do we care about activation functions?

- For non-linearity (and better forward/backward propagation)
- It is supposed to be differentiable



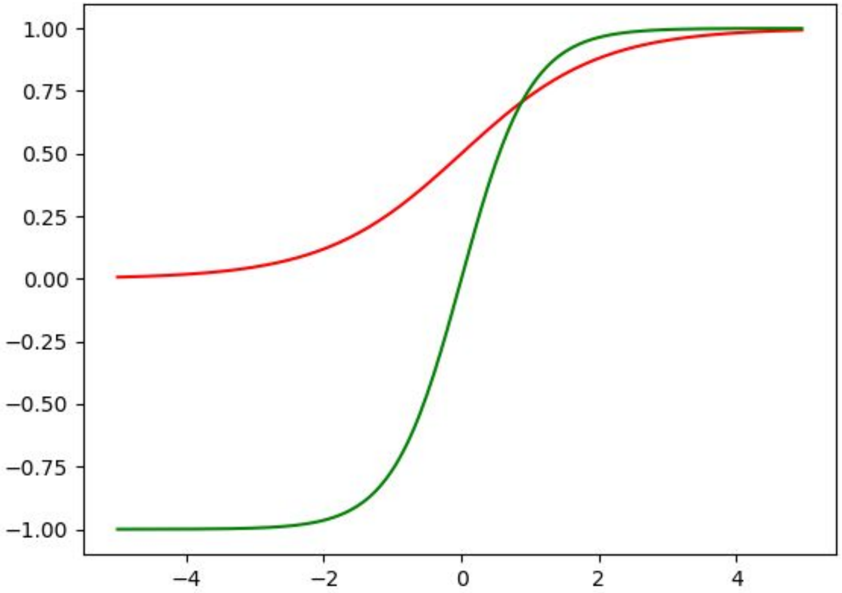
Logistic sigmoid / tanh

- Traditional choice ([0, 1] or [-1, 1])
- What's the benefit of tanh?



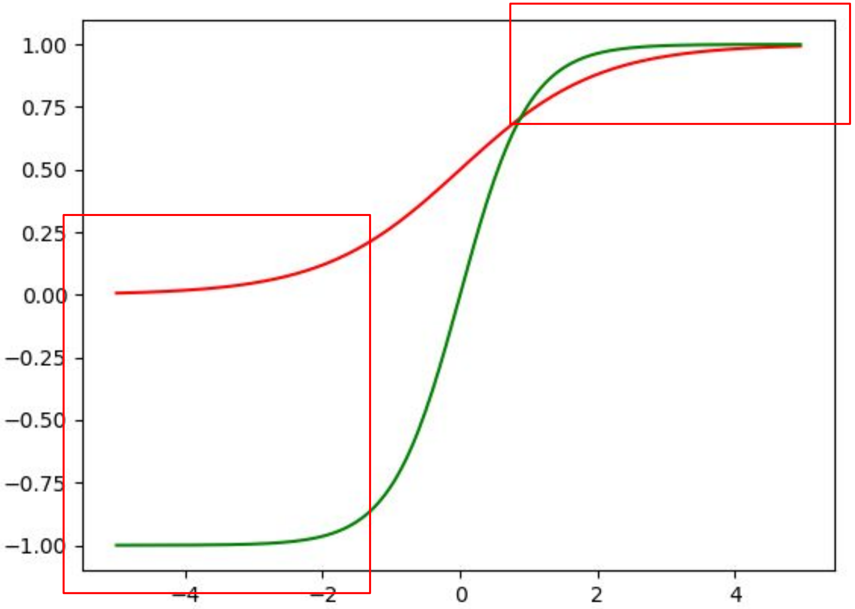
What's the issue with Logistic sigmoid / tanh?

- Hint: Look at the output for small/large values of z



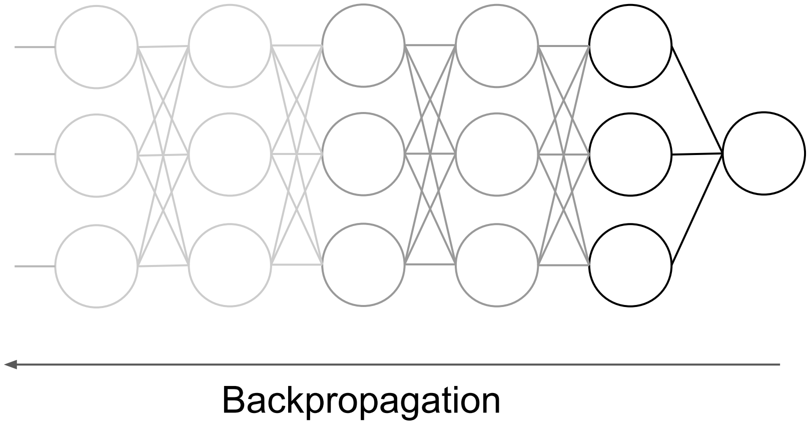
What’s the issue with Logistic sigmoid / tanh?

- Hint: Look at small values of z
- Gradients can become very small at large/small input → **Vanishing gradient problem**



Vanishing Gradient Problem

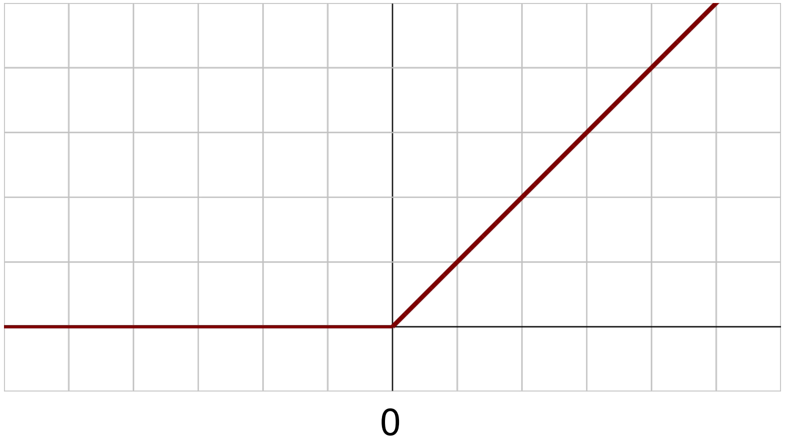
- **Errors** are not propagated in the backpropagation step due to too small gradient values → Shallow layers are not appropriately updated
    - cf. Exploding Gradient Problem (due to too large gradient values)



Rectified Linear Unit (ReLU)

- For better gradient propagation

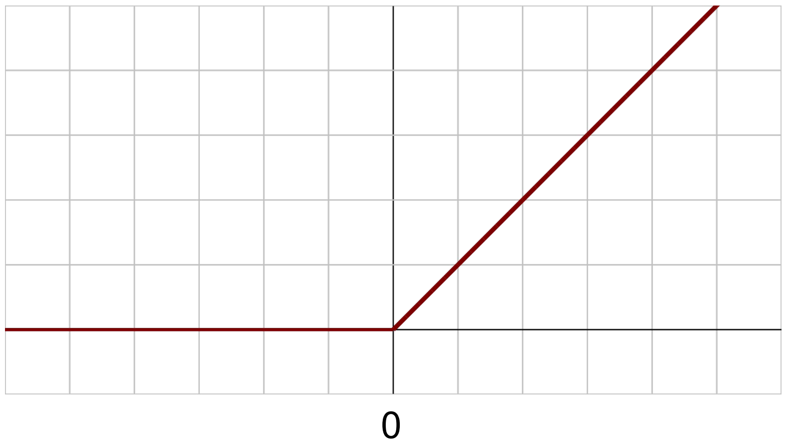
$$f(x) = \max(0, x)$$



Rectified Linear Unit (ReLU)

- For better gradient propagation

$$f(x) = \max(0, x)$$



The function is not differentiable at x = 0
but we don't care about it (?!)

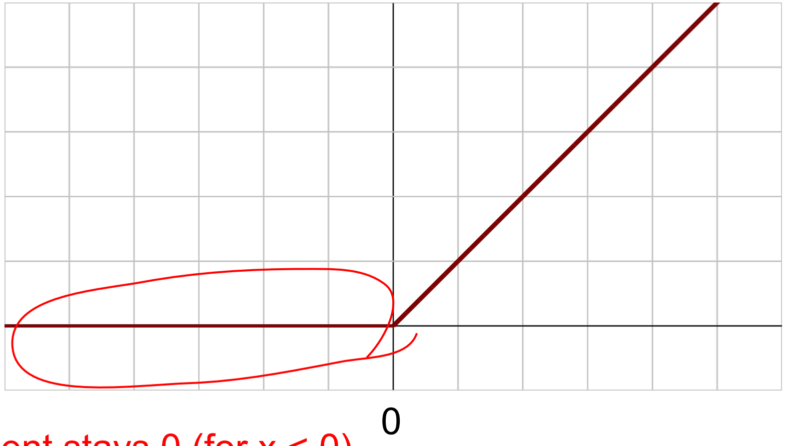
$$f'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

cf. subdifferentiable

Rectified Linear Unit (ReLU)

- For better gradient propagation

$$f(x) = \max(0, x)$$



The gradient stays 0 (for x < 0)

The function is not differentiable at x = 0
but we don't care about it (?!)

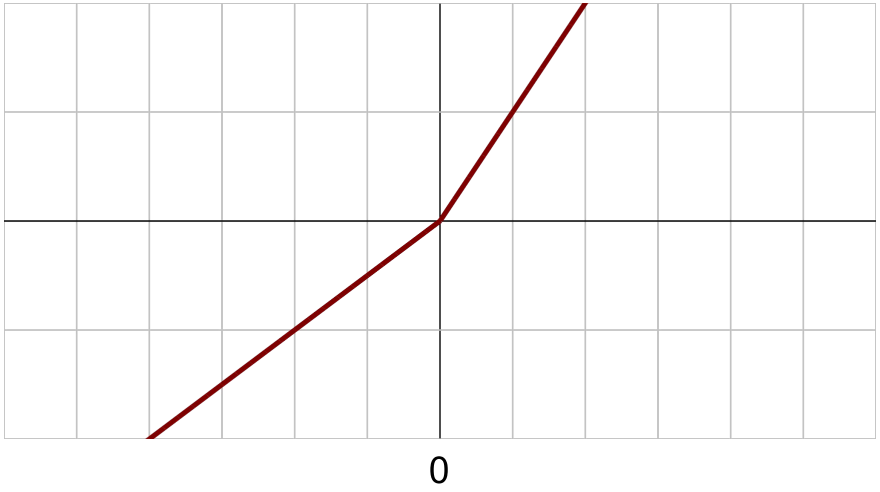
$$f'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

cf. subdifferentiable

Leaky ReLU

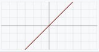
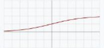
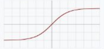
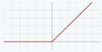
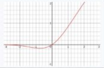
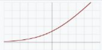
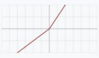
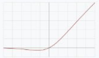
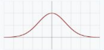
- A ReLU variant

$$f(x) = \max(0.01x,\ x)$$



$$f'(x) = \begin{cases} 0.01 & x \le 0 \\ 1 & x > 0 \end{cases}$$

A list of activation functions (from Wikipedia)

Name	Plot	Function, $f(x)$	Derivative of f, $f'(x)$	Range	Order of continuity
Identity		x	1	$(-\infty,\infty)$	C^∞
Binary step		$\begin{cases}0 & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$	$\begin{cases}0 & \text{if } x\neq 0\\ \text{undefined} & \text{if } x=0\end{cases}$	$\{0,1\}$	C^{-1}
Logistic, sigmoid, or soft step		$\sigma(x)=\frac{1}{1+e^{-x}}$ [1]	$f(x)(1-f(x))$	$(0,1)$	C^∞
Hyperbolic tangent (tanh)		$\tanh(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$1-f(x)^2$	$(-1,1)$	C^∞
Rectified linear unit (ReLU)[9]		$\begin{cases}0 & \text{if } x\leq 0\\ x & \text{if } x>0\end{cases}$ $=\max\{0,x\}=x\mathbf{1}_{x>0}$	$\begin{cases}0 & \text{if } x<0\\ 1 & \text{if } x>0\\ \text{undefined} & \text{if } x=0\end{cases}$	$[0,\infty)$	C^0
Gaussian Error Linear Unit (GELU)[4]		$\frac{1}{2}x\left(1+\text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $=x\Phi(x)$	$\Phi(x)+x\phi(x)$	$(-0.17\ldots,\infty)$	C^∞
Softplus[10]		$\ln(1+e^x)$	$\frac{1}{1+e^{-x}}$	$(0,\infty)$	C^∞
Exponential linear unit (ELU)[11]		$\begin{cases}\alpha(e^x-1) & \text{if } x\leq 0\\ x & \text{if } x>0\end{cases}$ with parameter α	$\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x>0\\ 1 & \text{if } x=0 \text{ and } \alpha=1\end{cases}$	$(-\alpha,\infty)$	$\begin{cases}C^1 & \text{if } \alpha=1\\ C^0 & \text{otherwise}\end{cases}$
Scaled exponential linear unit (SELU)[12]		$\lambda\begin{cases}\alpha(e^x-1) & \text{if } x<0\\ x & \text{if } x\geq 0\end{cases}$ with parameters $\lambda=1.0507$ and $\alpha=1.67326$	$\lambda\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$	$(-\lambda\alpha,\infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)[13]		$\begin{cases}0.01x & \text{if } x<0\\ x & \text{if } x\geq 0\end{cases}$	$\begin{cases}0.01 & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0
Parameteric rectified linear unit (PReLU)[14]		$\begin{cases}\alpha x & \text{if } x<0\\ x & \text{if } x\geq 0\end{cases}$ with parameter α	$\begin{cases}\alpha & \text{if } x<0\\ 1 & \text{if } x\geq 0\end{cases}$	$(-\infty,\infty)$[2]	C^0
Sigmoid linear unit (SiLU,[4] Sigmoid shrinkage,[15] SiL,[16] or Swish-1[17])		$\frac{x}{1+e^{-x}}$	$\frac{1+e^{-x}+xe^{-x}}{(1+e^{-x})^2}$	$[-0.278\ldots,\infty)$	C^∞
Mish [18]		$x\tanh(\ln(1+e^x))$	$\frac{(e^x(4e^{2x}+e^{3x}+4(1+x)+e^x(6+4x)))}{(2+2e^x+e^{2x})^2}$	$[-0.308\ldots,\infty)$	C^∞
Gaussian		e^{-x^2}	$-2xe^{-x^2}$	$(0,1]$	C^∞
Growing Cosine Unit (GCU)[7]		$x\cos(x)$	$\cos(x)-x\sin(x)$	$(-\infty,\infty)$	C^∞

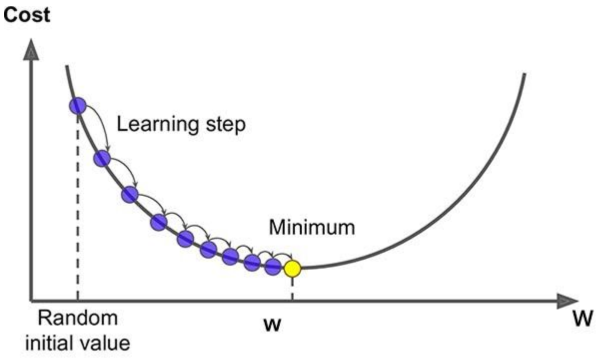
Summary: Activation functions

- Traditionally, Logistic sigmoid or tanh were used, which may cause the **vanishing gradient problem**
- The de-facto standard choice: **ReLU** (or ReLU variants)

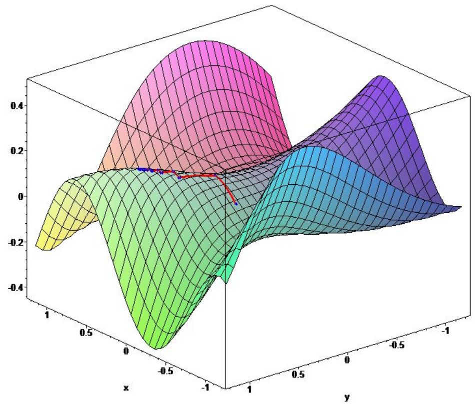
Optimizers

Stochastic Gradient Descent

- Optimization method based on gradient that is calculated by **(randomly chosen) samples**



1d parameter space (Convex)

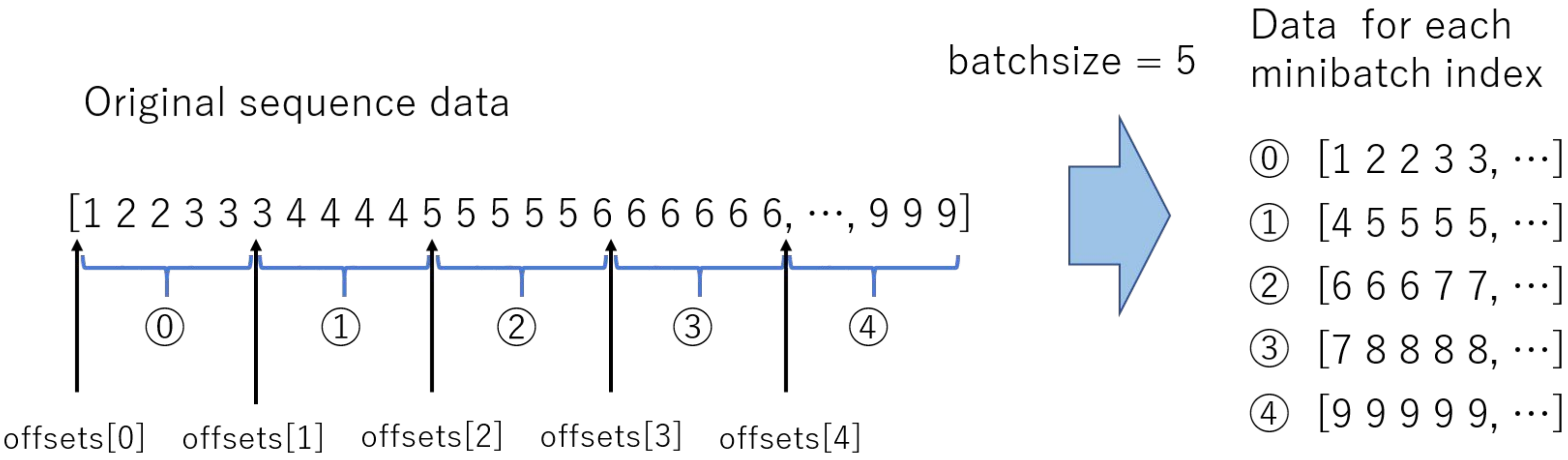


2d parameter space (Non-convex)

$$w^t \leftarrow w^{t-1} - \eta \nabla w$$

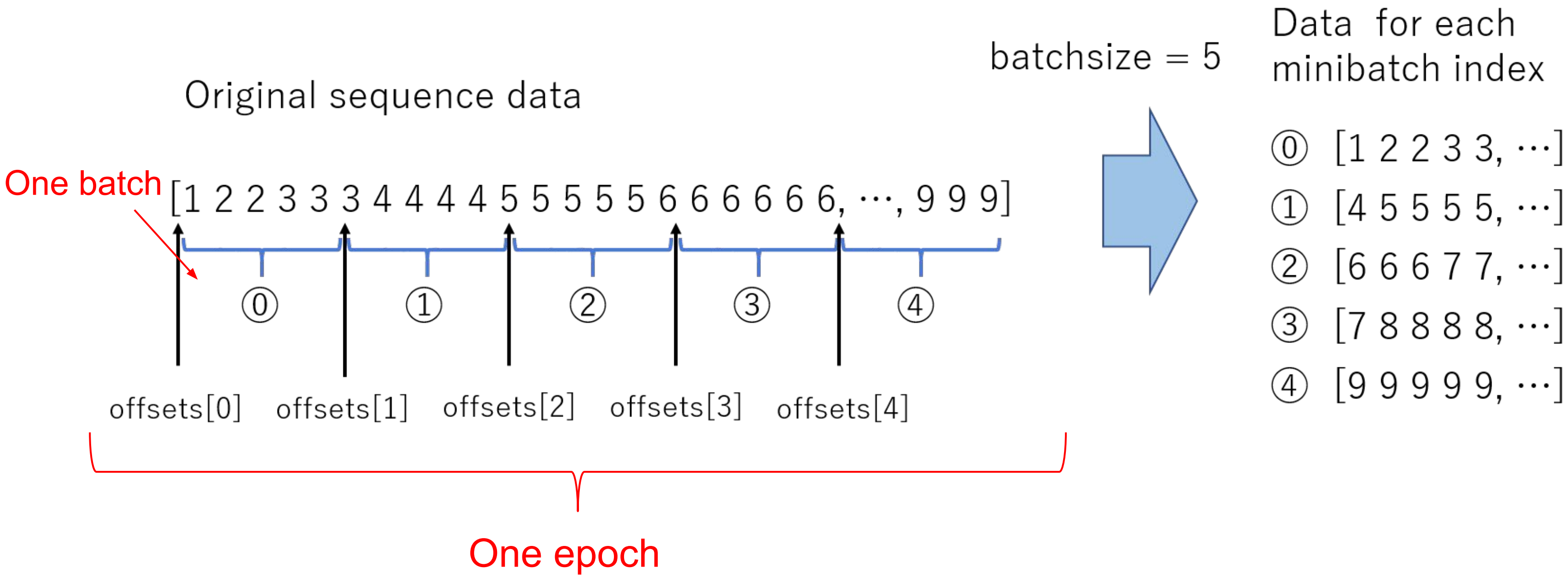
Optimization method = Gradient (direction) + Learning rate (Step size)

Mini-batch Sampling



http://corochann.com/training-rnn-with-simple-sequence-dataset-1304.html

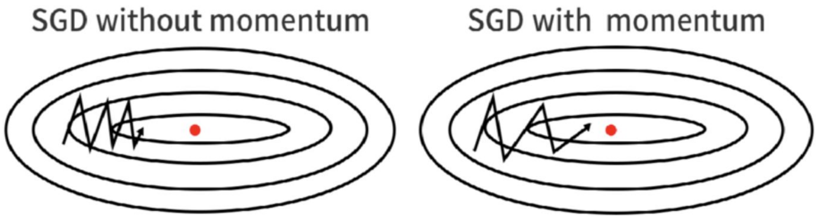
Mini-batch Sampling



http://corochann.com/training-rnn-with-simple-sequence-dataset-1304.html

SGD + Momentum

- Adding the notion of **velocity (*v*)** to SGD
  - Intuition: “Momentum” should help the optimization step less sensitive to the current sample (especially when the batch size is small)

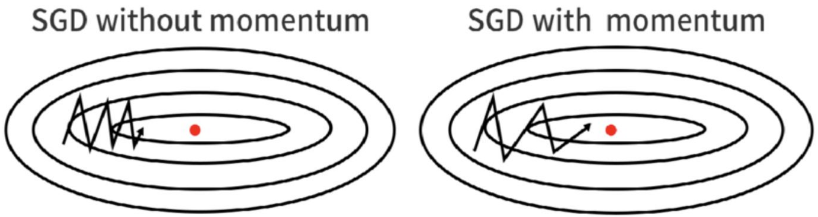


$$v^t \leftarrow \alpha v^{t-1} - \varepsilon \nabla w$$

$$w^t \leftarrow w^{t-1} + v^t$$

SGD + Momentum

- Adding the notion of **velocity (*v*)** to SGD
  - Intuition: “Momentum” should help the optimization step less sensitive to the current sample (especially when the batch size is small)



$$v^t \leftarrow \alpha v^{t-1} - \varepsilon \nabla w$$

$$w^t \leftarrow w^{t-1} + v^t$$

Algorithms with Adaptive Learning Rates

- AdaGrad
- RMSProp
- ...
- **Adam**

Adam (Adaptive moments) (Kingma and Ba 2014)

- SGD + first-order moment
- **+ Second-order moment**

$$m^t \leftarrow \beta_1 m^{t-1} + \left(1-\beta_1\right) \nabla w^t$$

$$v_t \leftarrow \beta_2 v^{t-1} + \left(1-\beta_2\right) (\nabla w^t)^2$$

- **+ Bias correction**

$$\hat{m}^t \leftarrow \frac{m^t}{1-(\beta_1)^t}$$

$$\hat{v}^t \leftarrow \frac{v^t}{1-(\beta_2)^t}$$

initial learning rate

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\hat{v}^t + \varepsilon}} \hat{m}^t$$

$$\beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}$$

Suggested hyper-parameters

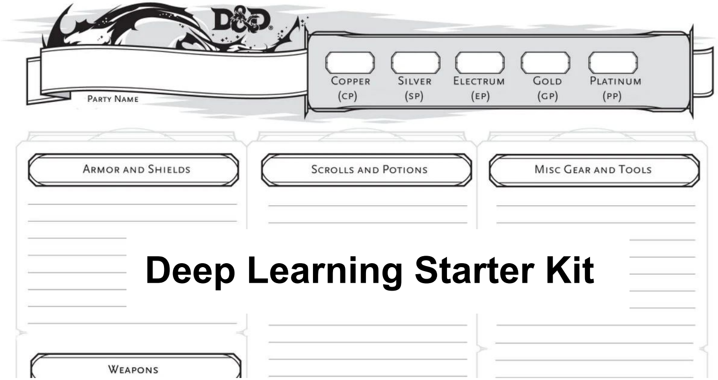
Summary: Optimizers

- Grand Design: **SGD + mini-batch sampling**
- **Learning rate tuning is key** → Algorithms with adaptive learning rates
- The de-facto standard choice: **Adam**

Further reading: [An overview of gradient descent optimization algorithms]
[[2007.01547] Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers]

Inventory

- Layers
  - Linear Layer
- Activation functions
  - Logistic sigmoid, tanh
  - ReLU, Leaky ReLU
- Optimizers
  - SGD w/wo Momentum
  - Adam



Part II: PyTorch Basics

4pm?

PyTorch Basics

- Quick overview of PyTorch
- PyTorch sample code for training a machine learning model
  - (1) Dataset class
  - (2) Model class
  - (3) Training iteration

The aim of this part is **NOT** reproducing (excellent) tutorials
BUT to help you get a feel for PyTorch

… and share some sample code

PyTorch

Why PyTorch?

- De-facto standard library for CV/NLP research
- Why not TensorFlow?
  - No significant difference b/w PyTorch and TensorFlow anymore
  - Tensorflow has TensorFlow eager (for define-by-run)
  - PyTorch has TorchScript (for define-and-run)
- More research code assets written with PyTorch than Tensorflow

What is PyTorch?

- PyTorch != scikit-learn
  - PyTorch is NOT a machine learning/DL library

- PyTorch =~ NumPy
  - PyTorch is a fundamental library (designed for neural networks)

NumPy

- Everything is `ndarray`
- `ndarray` has data type

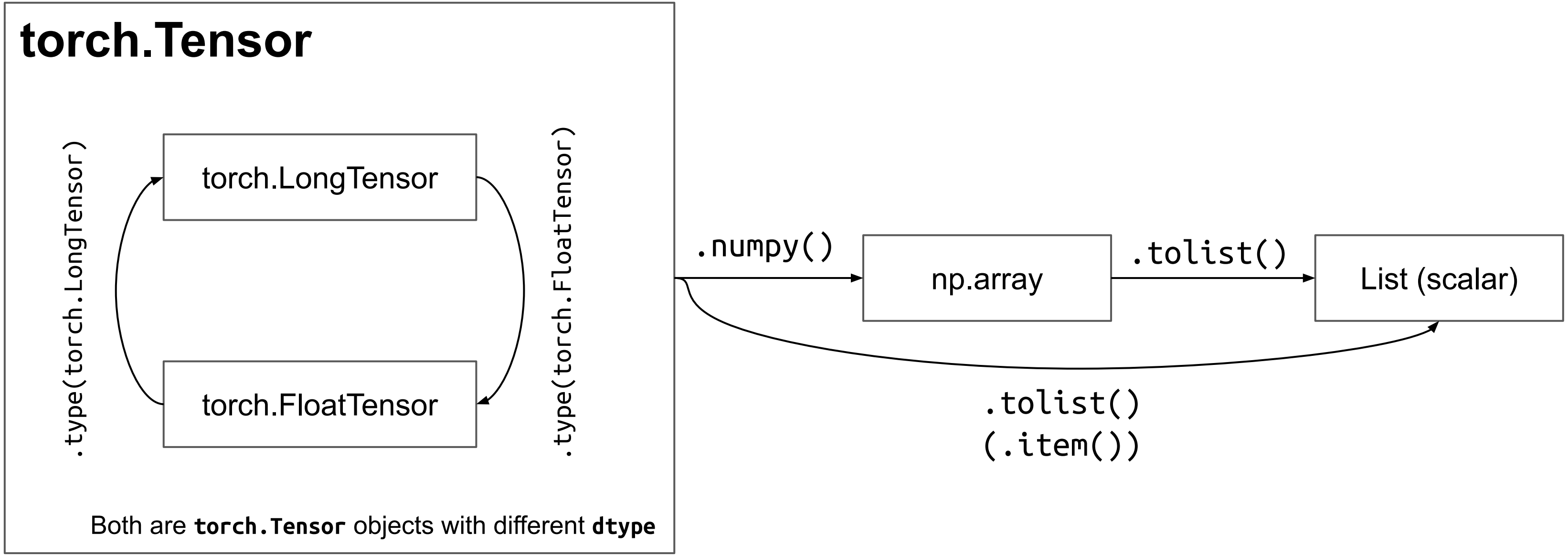
```
>>> import numpy as np
>>> a = np.array([1, 2, 3])

>>> a
array([1, 2, 3])

>>> a.tolist()
[1, 2, 3]

>>> a.dtype
dtype('int64')
```

PyTorch

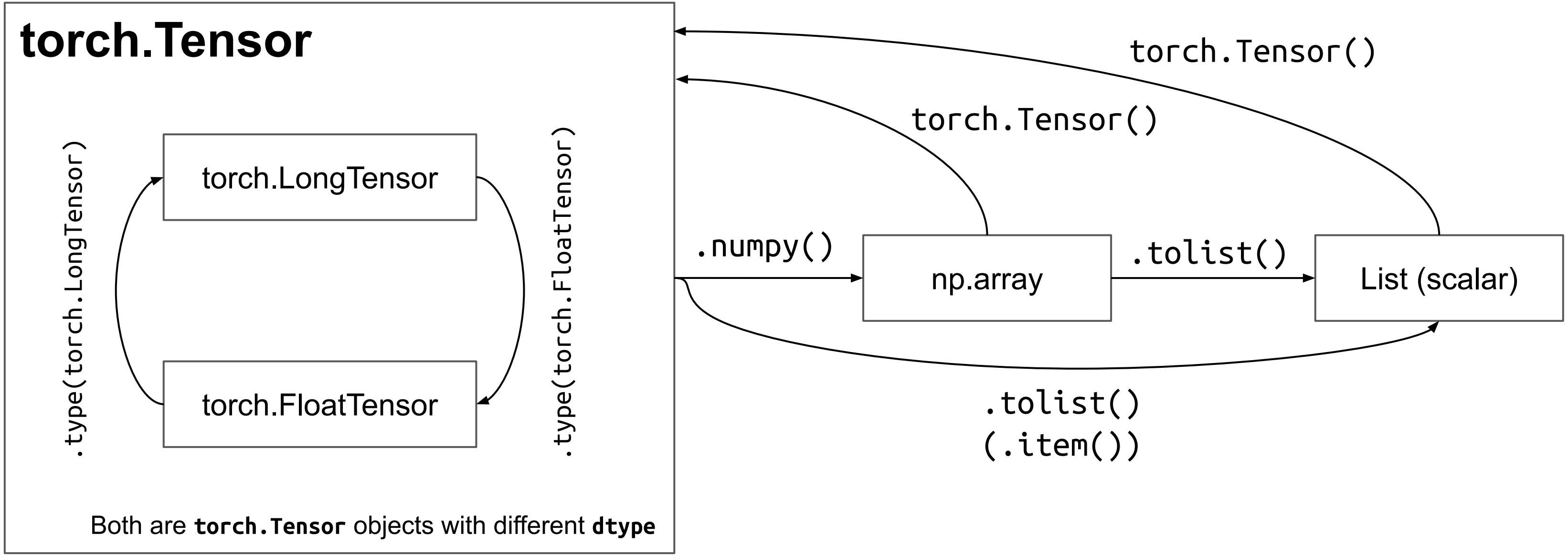
- Everything is `torch.Tensor`
- `Tensor` has data type

```
>>> import torch
>>> t = torch.Tensor([1, 2, 3])
>>> t
tensor([1., 2., 3.])

>>> t.tolist()
[1.0, 2.0, 3.0]


>>> t.dtype
torch.float32
```

PyTorch ⇔ NumPy ⇔ built-in objects

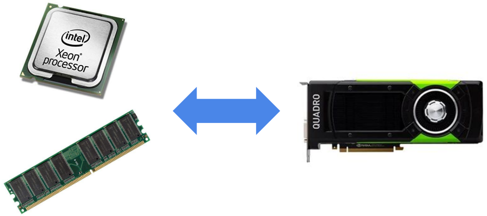


PyTorch ⇔ NumPy ⇔ built-in objects



Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`
Boolean	`torch.bool`	`torch.BoolTensor`	`torch.cuda.BoolTensor`

CPU ⇔ GPU



- Use `.to()` method to transfer object to/from GPU
- PyTorch methods are **device-agnostic**
  - Note: Objects that are used for calculation need to be in the same “world”

```
>>> import torch
>>> device = torch.device('cuda')
>>> t = torch.Tensor([1, 2, 3])
>>> t
tensor([1., 2., 3.])

>>> t = t.to(device)  # Not destructive method
>>> t * 3
tensor([3., 6., 9.], device='cuda:0')


>>> t.detach().cpu() * 2
tensor([2., 4., 6.])
```

Tips: GPU/CPU compatible code

Template

```
import torch
import torch.nn as nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Switching GPU IDs using the environment variable `CUDA_VISIBLE_DEVICES`

```
$ CUDA_VISIBLE_DEVICES=0 python train.py   # Force to use GPU0 (if exists)
$ CUDA_VISIBLE_DEVICES="" python train.py  # Force to use CPU
```

Basic Operations

- Arithmetic operations
  - `+, -`
- Matrix operations
  - `.dot()`
  - `.matmul()`
- Concatenation
  - `torch.cat()`

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py

Cheatsheet

Math operations

Numpy	PyTorch	Notes

`x+y`	`x+y`
	`y.add_(x)`
	`torch.add(x,y)`
		addition

`np.dot(x,y)` `np.matmul(x,y)`	`torch.mm(x,y)` `x.mm(y)`	matrix multiplication
`x*y`	`x*y`	element-wise multiplication
`np.max(x)`	`torch.max(x)`	
`np.argmax(x)`	`torch.argmax(x)`	
`x**2`	`x**2`	Element-wise powers

Array creation

Numpy	PyTorch	Notes
`np.empty((2, 2))`	`torch.empty(5, 3)`	empty array
`np.random.rand(3,2)`	`torch.rand(5, 3)`	random
`np.zeros((5,3))`	`torch.zeros(5, 3)`	zeros
`np.array([5.3, 3])`	`torch.tensor([5.3, 3])`	from list
`np.random.randn(*a.shape)`	`torch.randn_like(a)`	
`np.arange(16)`	`torch.range(0,15)`	array starting from 0 ending at 15 (inclusive)

Array manipulations

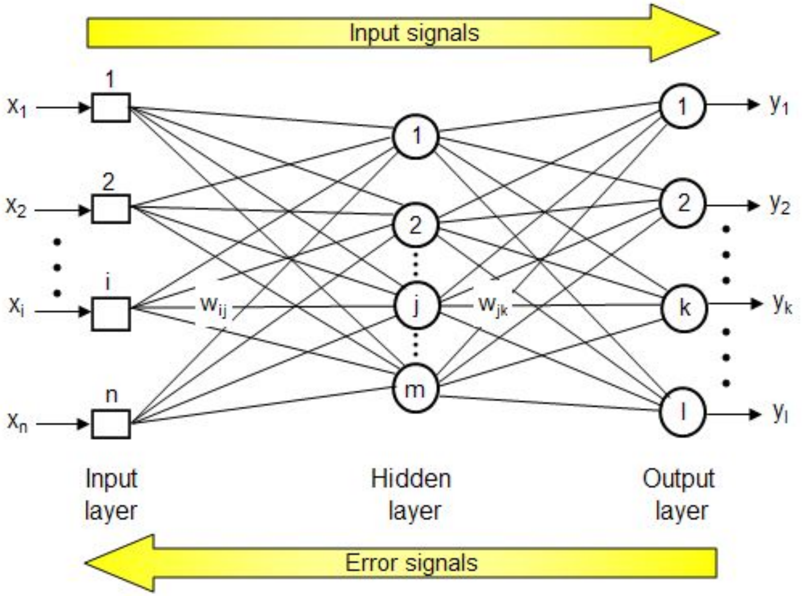
Numpy	PyTorch	Notes
`x.T` `np.transpose(x)`	`torch.transpose(x, 0, 1)` `torch.transpose(x, 1, 0)`	transpose
`a = a.reshape(-1, 2)`	`a = a.view(-1,2)`	reshape array to have two columns and however as many rows
`np.concatenate([a, b])`	`torch.cat([a,b])`	concatenate list of arrays/tensors

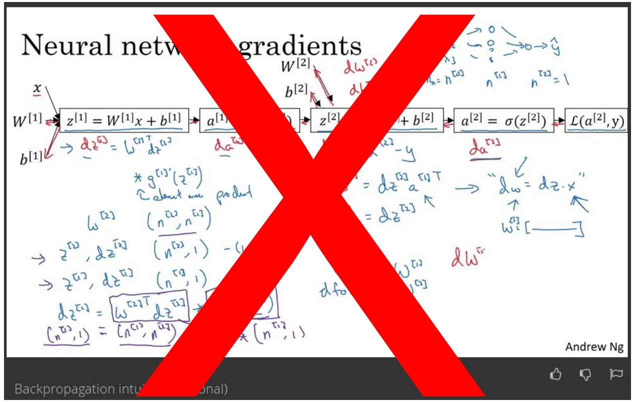
https://github.com/myazdani/numpy-pytorch-cheatsheet

Training Flow with PyTorch

Forward and Backward propagation

- Backward propagation = Gradient calculation

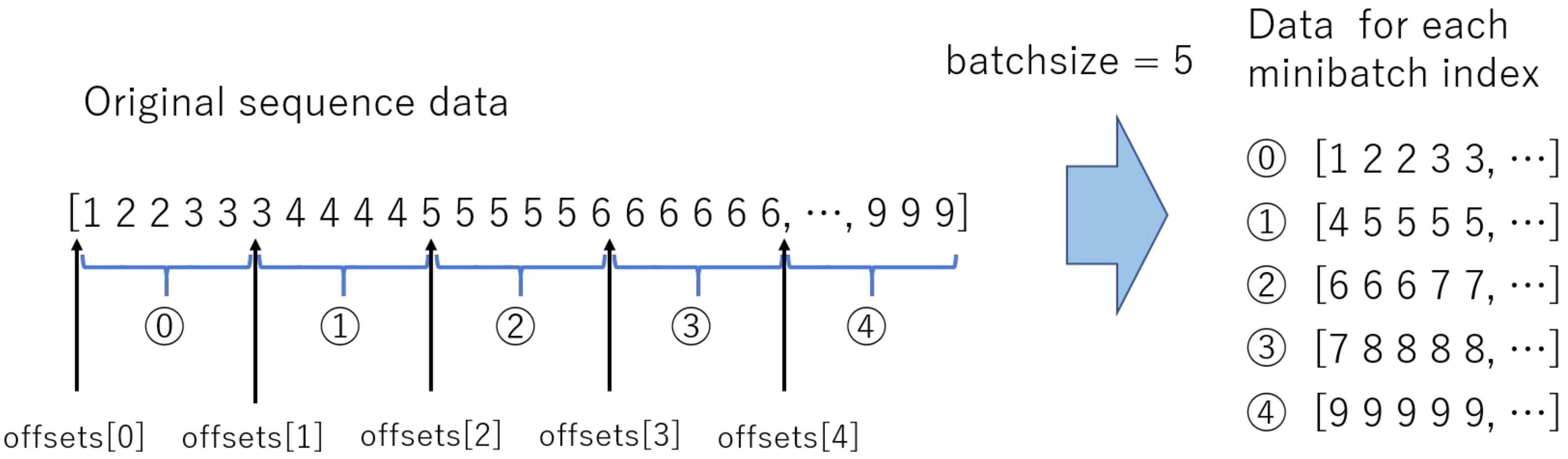




Only need to define **forward propagation** function
PyTorch **AutoGrad** takes care of backward propagation :)

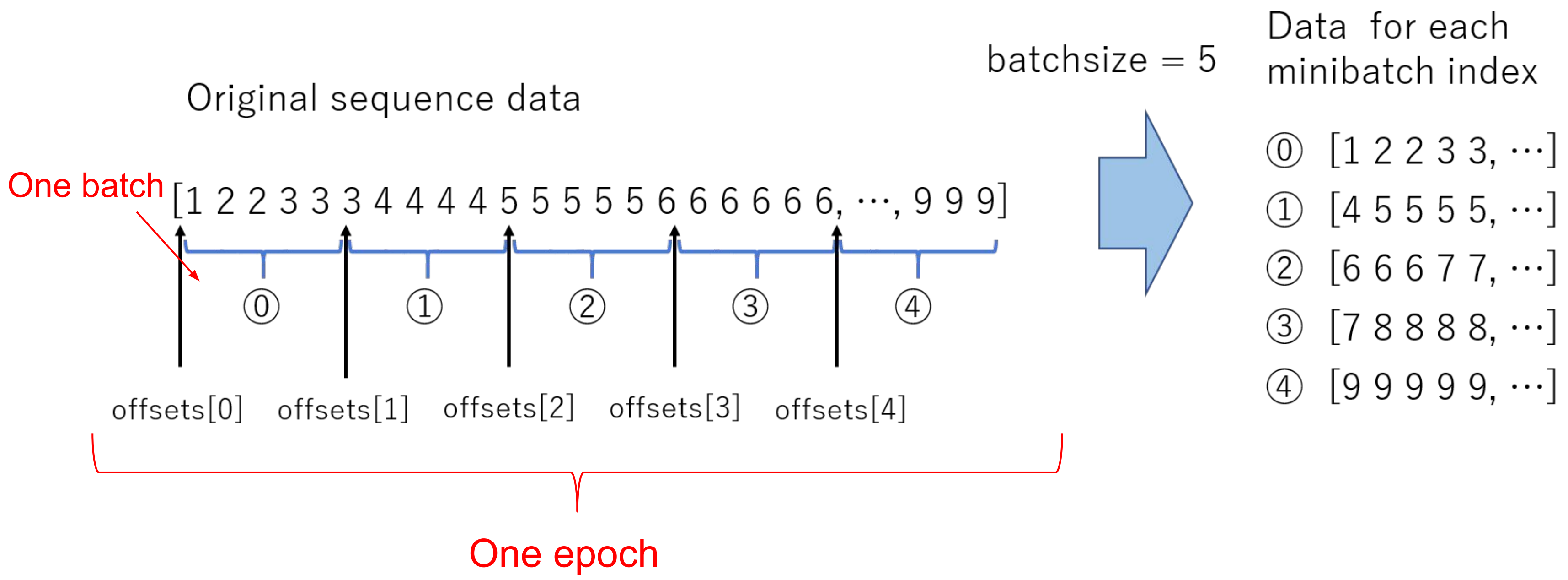
Recap

Mini-batch Sampling



http://corochann.com/training-rnn-with-simple-sequence-dataset-1304.html

Mini-batch Sampling



http://corochann.com/training-rnn-with-simple-sequence-dataset-1304.html

Pseudo Code for Training

- Training script has double loops: One for **epochs** and the other for **batches**

```
Input: parameters, data, NumEpoch, BatchSize

For i = 1 to NumEpoch
    batches ← CreateMiniBatch(data, BatchSize)
    For j = 1 to len(Batches)
        batch ← batches[j]
        update_parameters(parameters, batch)
```

Writing Training Script with PyTorch

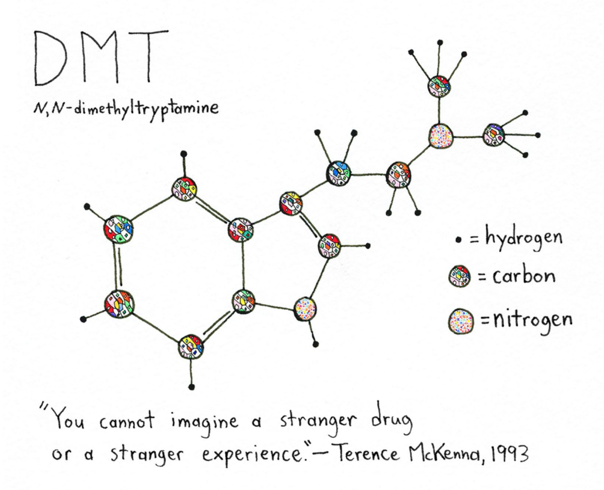
Training Script: 3 Essential Blocks (D-M-T)

- **D**ata Preparation
- **M**odel Preparation
- **T**raining Iteration

Training Script: 3 Essential Blocks (D-M-T)

- **D**ata Preparation
- **M**odel Preparation
- **T**raining Iteration

Not this DMT



Training Script (1): Data Preparation

- Step 1. Create a Dataset object
- Step 2. Create a DataLoader object

```
dataset = TensorDataset(torch.Tensor(X),
                        torch.LongTensor(y))
dataloader = DataLoader(dataset)

for epoch_idx in range(num_epochs):
    for batch_idx, batch in enumerate(dataloader):
        # Do something
```

Training Script (1): Data Preparation

- Step 1. Create a Dataset object
- Step 2. Create a DataLoader object

```
dataset = TensorDataset(torch.Tensor(X),
                        torch.LongTensor(y))
```

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None)
```

Training Script (2): Model Preparation

- Step 1. Create a model object
- Step 2. Create an optimizer object. Set the parameters of the model
- Step 3. Create a loss function

```
model = MyModel()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.)
criterion = nn.CrossEntropyLoss()
```

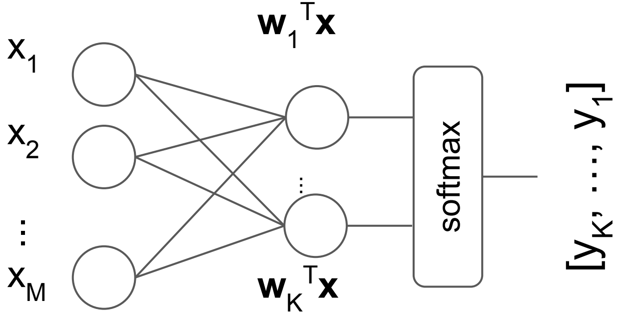
Training Script (3): Training iteration

```
for epoch in range(10):
    training_loss = 0.0
    valid_loss = 0.0
    model.train()
    for batch_idx, batch in enumerate(dataloader):
        optimizer.zero_grad()                       # Initialize gradient information
        X, y = batch                                # Get feature/label from batch
        pred = model(X)                             # Get output from the model
        loss = criterion(pred, y)                   # Evaluate loss values
        loss.backward()                             # Backpropagate
        optimizer.step()                            # Update parameters

        training_loss += loss.data.item() * batch_size
    training_loss /= len(train_iterator)
```

Model Preparation:
Design a Custom Model

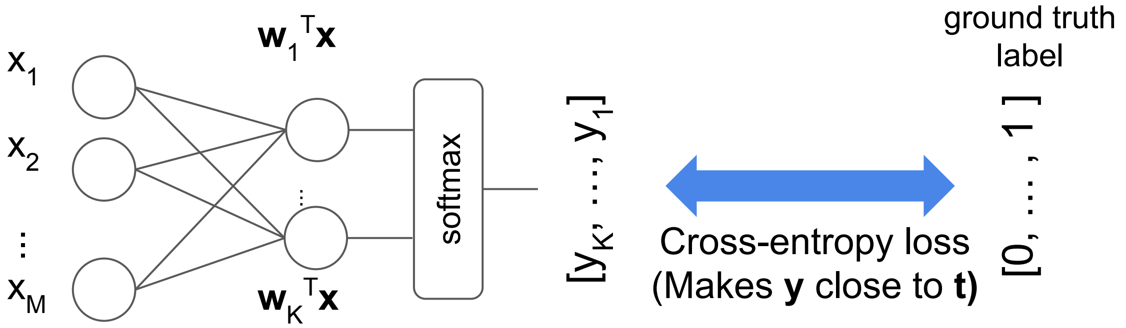
Logistic Regression as an NN model

- A single-layer NN
- (If multi-class) the final layer is a softmax function that converts output values into probabilities



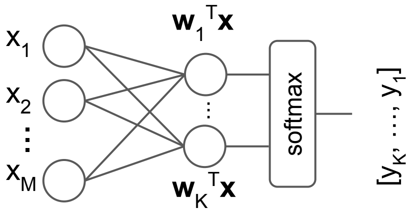
Logistic Regression as an NN model

- A single-layer NN
- (If multi-class) the final layer is a softmax function that converts output values into probabilities
- Use **cross-entropy loss** for training



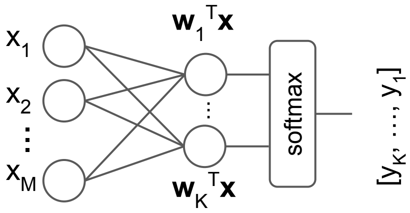
$$-\sum_{i=1}^{n}[t_i \log(\underbrace{\sigma(z)}_{y_i}) + (1-t_i)\log(1-\underbrace{\sigma(z)}_{y_i})]$$

Logistic Regression




```
class LogisticRegression(nn.Module):
    def __init__(self,
                 num_input: int,   # M
                 num_output: int): # K
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(num_input, num_output)

    def forward(self, X):
        out = self.linear(X)
        return F.softmax(out)
```

Logistic Regression




```
class LogisticRegression(nn.Module):
    def __init__(self,
                 num_input: int,   # M
                 num_output: int): # K
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(num_input, num_output)

    def forward(self, X):
        out = self.linear(X)
        return out F.softmax(out)
```

Ref. CrossEntropyLoss

CrossEntropyLoss

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

Recap

<u>D</u>-M-T: Data Preparation

- `dataset = TensorDataset(X, y)`
- `dataloader = DataLoader(dataset)`

D-M-T: Model Preparation

- `model = MyModel()`
- `optimizer = Optimizer(model.parameters())`
- `criterion = LossFunction()`

D-M-T: Training Iteration

- `# Training`
- `model.train()`
- `for each batch`
  - `optimizer.zero_grad()`
  - `pred = model(X)`
  - `loss = criterion(pred, batch.label)`
  - `loss.backward()`
  - `optimizer.step()`

Summary

Takeaway: D-M-T

- PyTorch is yet another NumPy
- Remember **D**ataset, **M**odel, **T**raining iteration
- `TensorDataset` wraps the conventional formats of X, y

Hands-on Time!

https://colab.research.google.com/github/suhara/cis6930-fall2021/blob/main/notebooks/cis6930_week1_deep_learning_basics.ipynb

Any Questions?

Appendix: Tips

Getting Started

PYTORCH

Deep Learning with PyTorch: A 60 Minute Blitz

Writing Custom Datasets, DataLoaders and Transforms

PyTorch

TensorBoard

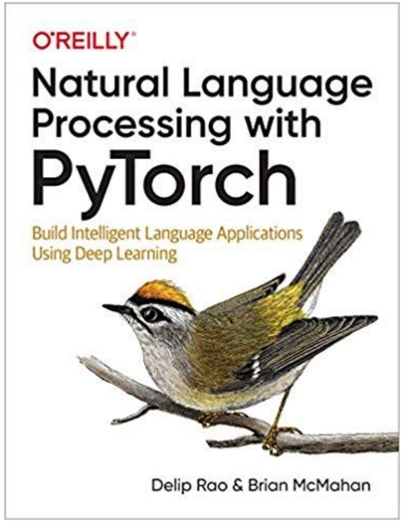
Visualizing Models, Data, and Training with TensorBoard

https://pytorch.org/tutorials/

PyTorch

nlp | Latest | New (3) | Unread | Top | New Topic

Topic	Replies	Views	Activity
About the nlp category	0	605	Dec '17
KeyError: 'GRU'	2	109	12h
String matching with LSTM not converging • new	0	9	12h
LSTM training loss does not decrease	9	69	15h
Loss is same in each batch and gradients are also same • new	1	19	17h
ValueError: Requested tokenizer basic_english, valid choices are a callable that takes a single string as input, • new	2	20	21h
Training with and without pretrained embeddings	12	115	3d



Define MyDataset

- Only 2 methods to implement: `__len__()` and `__getitem__()`

```
class MyDataset(Dataset):
    def __len__(self):
        """Returns # of instances."""
        pass

    def __getitem__(self, idx):
        """Returns single instane."""
        pass
```

Example

train.csv

label	data
1	0.72
0	0.56
1	0.67
...	...

```
class MyDataset(Dataset):
    def __init__(self):
        self.df = pd.read_csv("train.csv")

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        s = self.df.iloc[idx]
        return {"label": torch.LongTensor(s["label"]),
                "data": torch.FloatTensor(s["data"])}
```

Returned value does not have to be `Dict[str, torch.Tensor]`
For example, it can be a hierarchical dictionary `Dict[str, Dict[str, torch.Tensor]]`

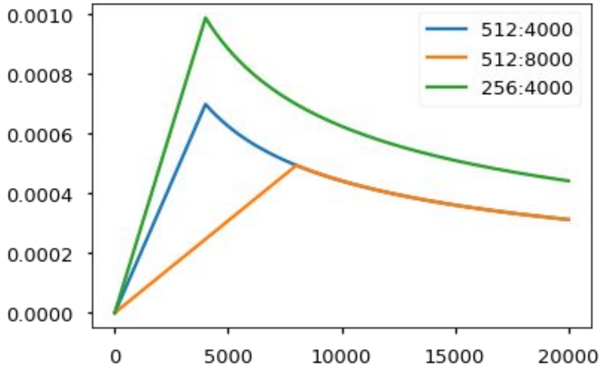
Appendix: Customized Optimizer for Transformer

Optimizer

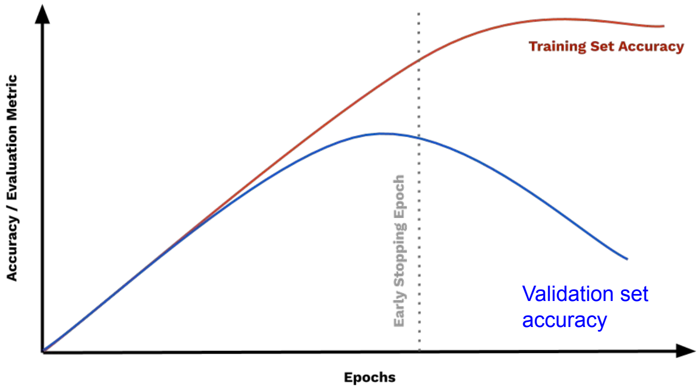
We used the Adam optimizer (cite) with $\beta_1=0.9$, $\beta_2=0.98$ and $\epsilon=10^{-9}$. We varied the learning rate over the course of training, according to the formula: This corresponds to increasing the learning rate linearly for the first $warmup_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup_steps=4000$.

> *Note: This part is very important. Need to train with this setup of the model.*

```
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
        "Implement `lrate` above"
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))

def get_std_opt(model):
    return NoamOpt(model.src_embed[0].d_model, 2, 4000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```



Tips: Early Stopping




```
best_val_loss = None
earlystop_counter = 0

for epoch_idx in range(num_epochs):
    # Training
    model.train()
    for batch_idx, batch in enumerate(train_dataloader):
        ...
    # Validation
    model.eval()
    for batch_idx, batch in enumerate(valid_dataloader):
        ...

    # Early stopping checkpoint
        if best_val_loss is None or \
           running_val_loss < best_val_loss:
        best_val_loss = running_val_loss
        earlystop_counter = 0
    else:
        earlystop_counter += 1

    if earlystop_counter >= patience:
        print("Early stopping.”)
        break
```

Tips: Transferring data to GPU

- “data” must be in the GPU world if you use GPU
  - One solution is to wrap DataLoader and call `.to(device)` method for each data
  - The following code can handle

```
def generate_batches(dataset,
                     batch_size,
                     shuffle=True,
                     drop_last=True,
                     device="cpu"):
    dataloader = DataLoader(dataset=dataset,
                            batch_size=batch_size,
                            shuffle=shuffle,
                            drop_last=drop_last)

    for data_dict in dataloader:
        if device == "cpu":
            yield data_dict
        else:
            out_data_dict = {}
            for name, tensor in data_dict.items():
                if type(tensor) == dict:
                    out_data_dict[name] = {}
                    for n, t in tensor.items():
                        out_data_dict[name][n] = data_dict[name][n].to(device)
                else:
                    out_data_dict[name] = data_dict[name].to(device)
            yield out_data_dict
```