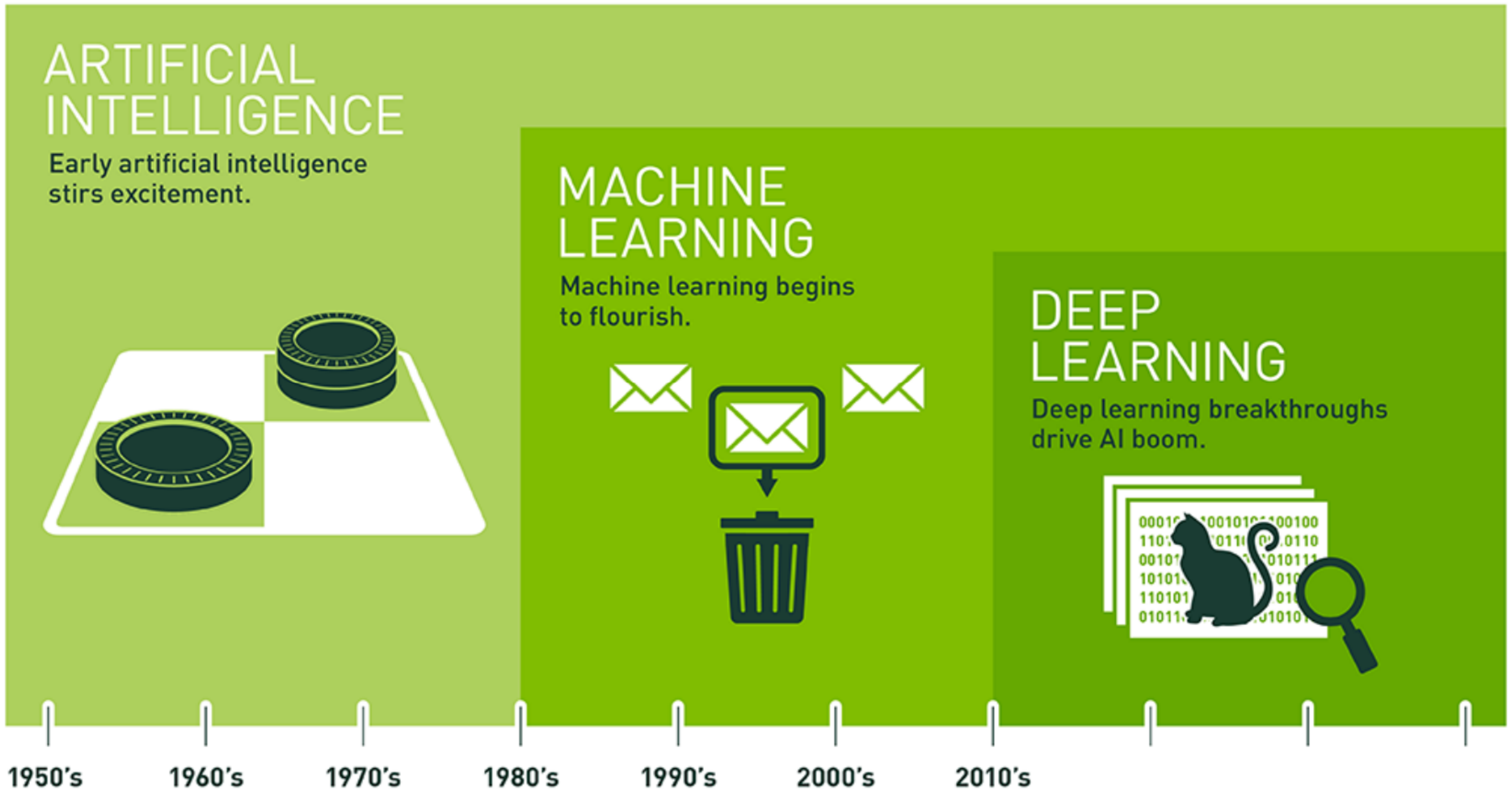
Introduction to deep learning

From an implementation view under Pytorch

Songpeng Zu @ 2020.11.17

AI, ML and DL


ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's

Artificial Intelligence
Artificial Intelligence is human intelligence exhibited by machines

Machine Learning
Field of study that gives computers the ability to learn without being explicitly programmed.

Deep Learning
Using deep neural networks to implement machine learning

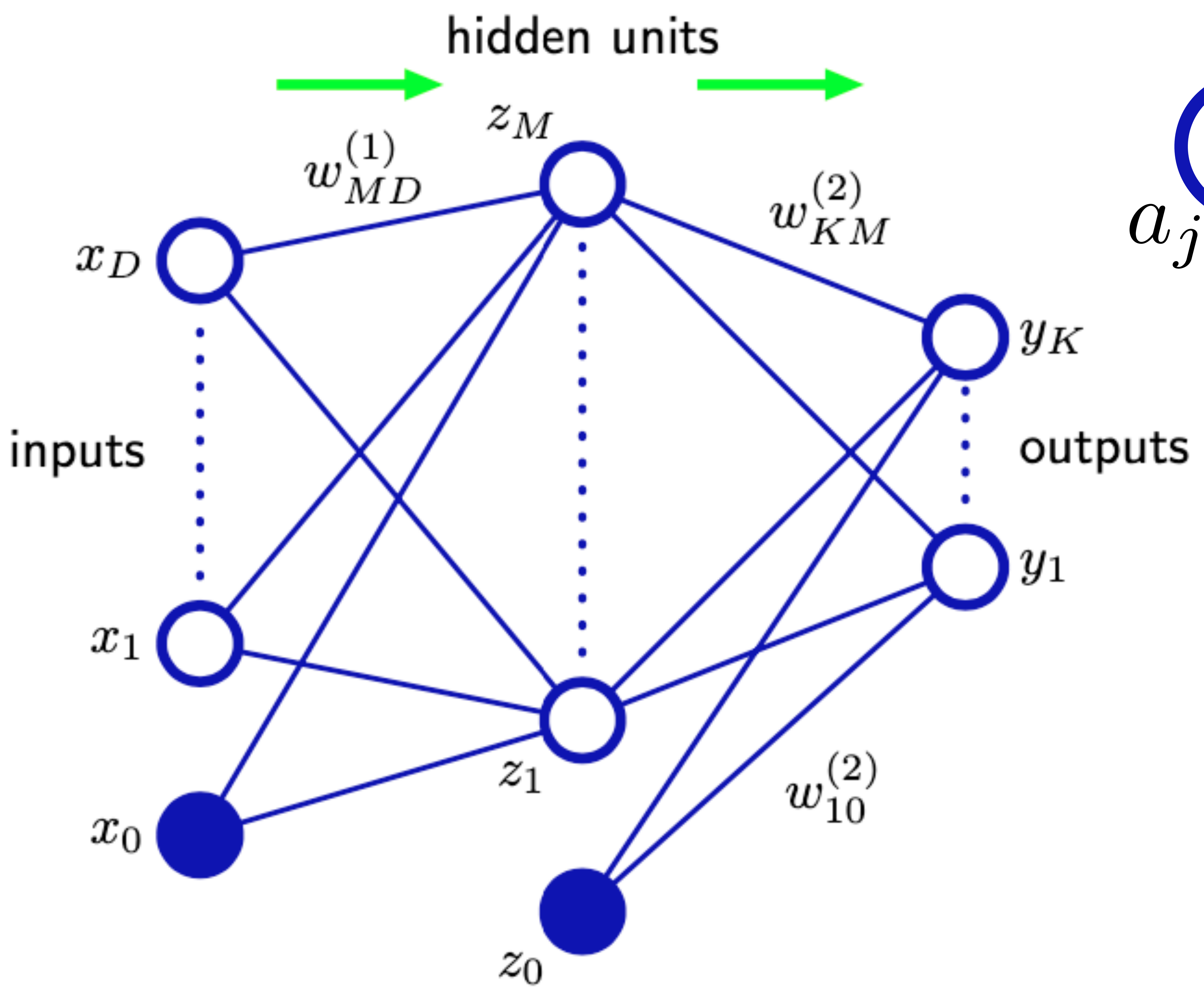
Deep Learning

Empirical Risk Minimization

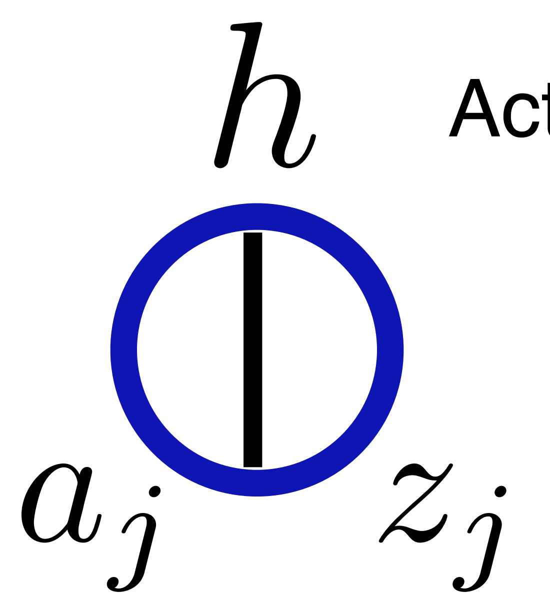
$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

In deep learning: the space of ***f*** *is the multiple-layer neural network*

A Typical Structure of Neural Network (NN)



Neural network or feed forward network or multilayer perceptron



Activation function

$$a_j = \sum_{i=1}^{D} w^{(1)}_{ji} x_i + w^{(1)}_{j0}$$

$$z_j = h(a_j)$$

$$a_k = \sum_{j=1}^{M} w^{(2)}_{kj} z_j + w^{(2)}_{k0}$$

$$y_k = \sigma(a_k)$$

Content

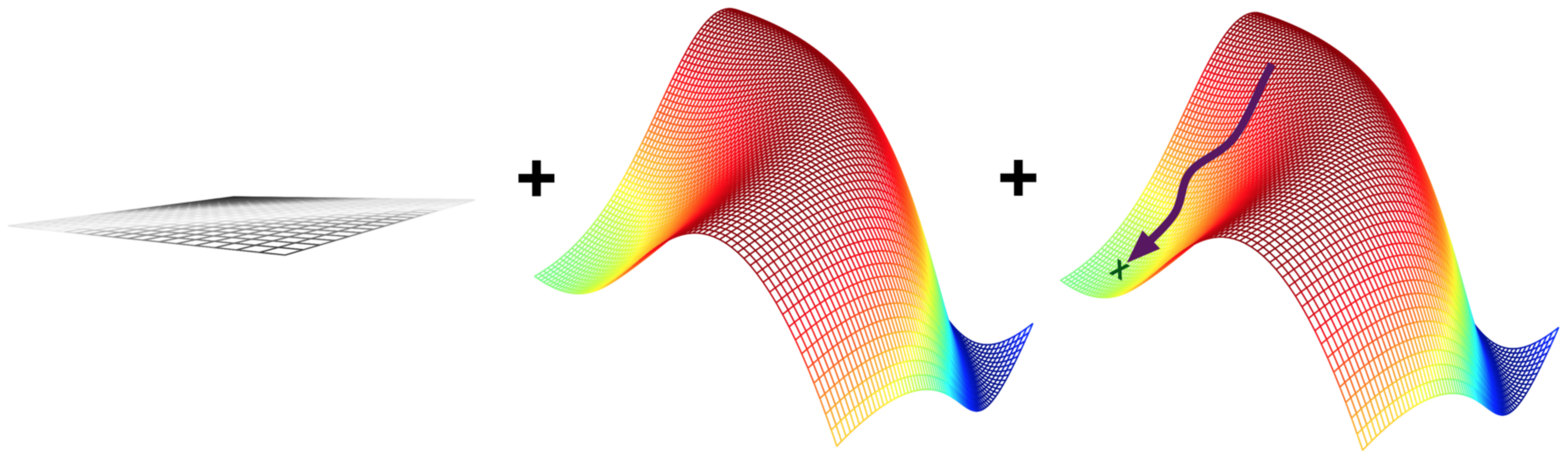
- Optimization in deep learning:
  - *Backpropagation (BP)*
  - *mini-batch gradient descent*
  - Momentum, adaptive learning rate
  - Tricks
- The implementation of deep learning
  - A deterministic, directed acyclic, computational graph
  - Elements: data organization; definition of neural network; and the optimization module
- Example: an implementation of VAE on MNIST

Optimization in deep learning

Learning and Optimization

Learning = Representation + Evaluation + Optimization

- Representation: Hypothesis space
- Evaluation: Objective/Loss function



Gradient-based optimization

First-order method

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- Approximation of $J(\cdot)$ at θ_t

$$J(\boldsymbol{\theta}_t + \boldsymbol{v}) \approx \hat{J}(\boldsymbol{\theta}_t + \boldsymbol{v}) = J(\boldsymbol{\theta}_t) + \nabla J(\boldsymbol{\theta}_t)^T \boldsymbol{v}$$

- Minimize the surrogate function

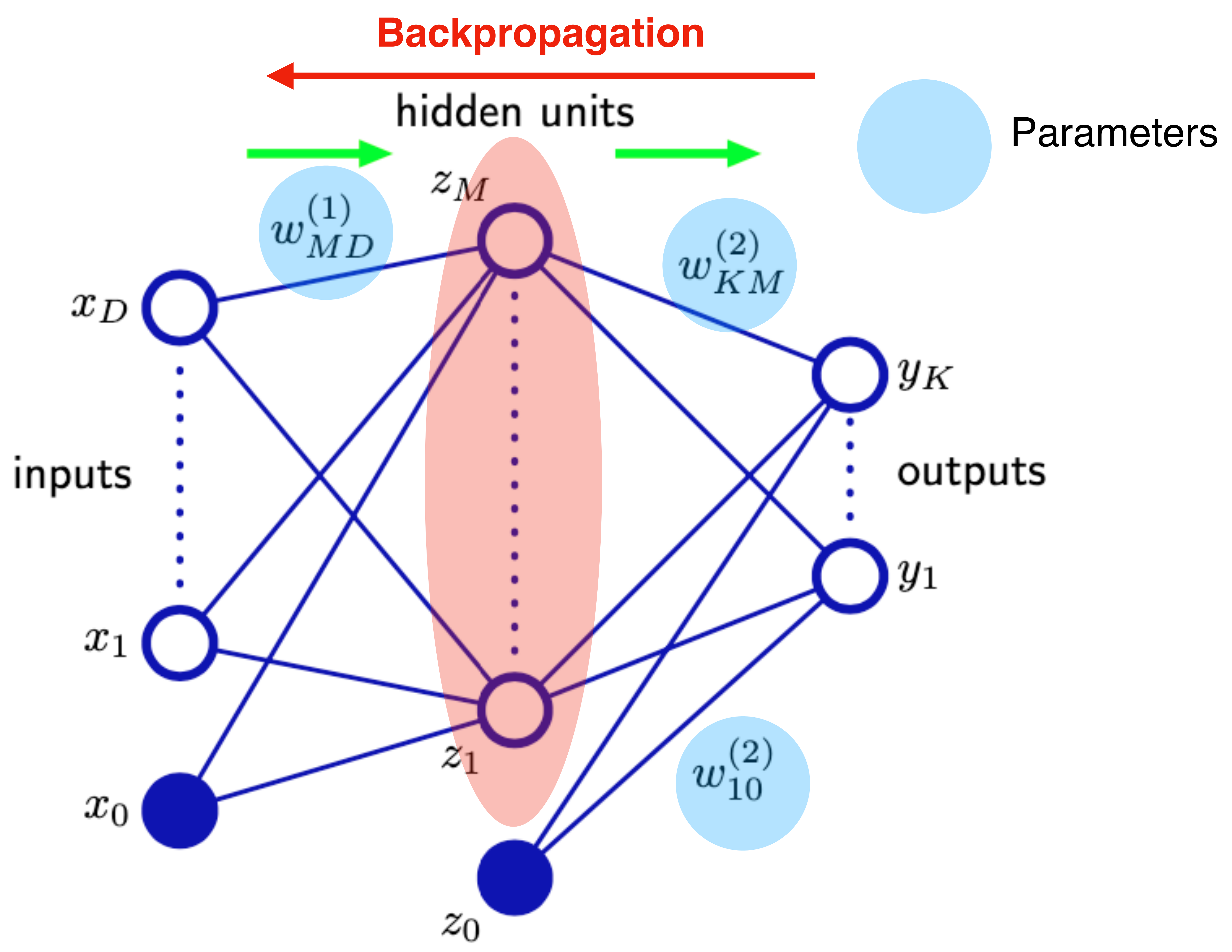
$$\boldsymbol{v}^* = \arg\min J(\boldsymbol{\theta}_t) + \nabla J(\boldsymbol{\theta}_t)^T \boldsymbol{v} + \frac{1}{2\alpha}\|\boldsymbol{v}\|_2^2 = -\alpha \nabla J(\boldsymbol{\theta}_t)$$

```
Initialization: $\boldsymbol{\theta}_0$
for $t = 0,1,\cdots$
    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J(\boldsymbol{\theta}_t)$
until stopping criterion is satisfied
```

Gradient descent

Backpropagation (BP)

Differentiation using Chain Rule



$$l = g(y_1, \cdots, y_k)$$

$$\nu = \left(\frac{\partial l}{\partial y_1}, \cdots, \frac{\partial l}{\partial y_K}\right)^t \quad J = \begin{bmatrix} \frac{\partial y_1}{\partial z_1}, & \dots, & \frac{\partial y_1}{\partial z_M} \\ \vdots, & \ddots, & \vdots \\ \frac{\partial y_K}{\partial z_1}, & \dots, & \frac{\partial y_K}{\partial z_M} \end{bmatrix}$$

$$\frac{\partial l}{\partial z} = J^t \cdot \nu$$

Mini-batch optimization

When we handle large scale of data set

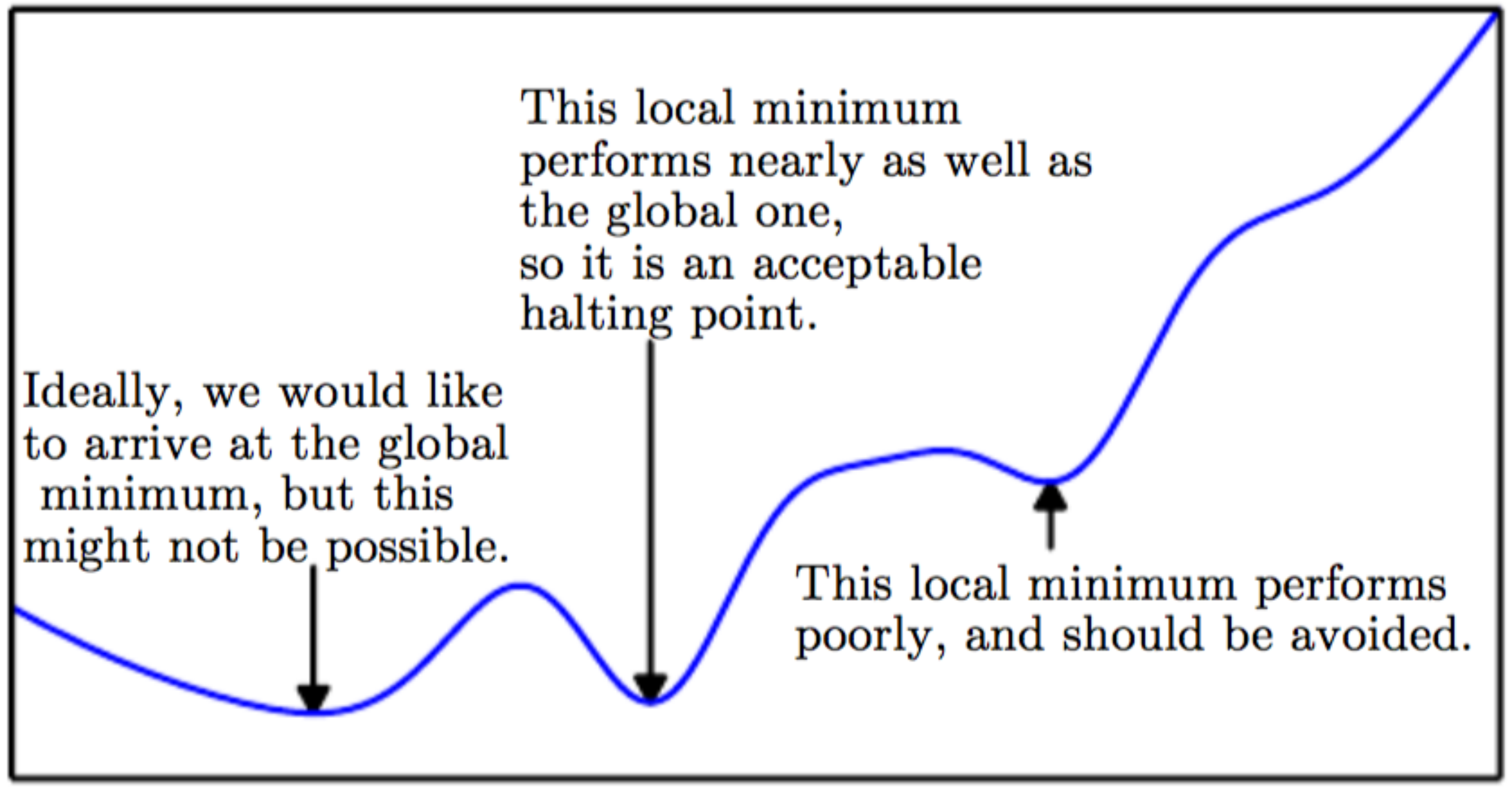
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\boldsymbol{x}, y; \boldsymbol{\theta})$$

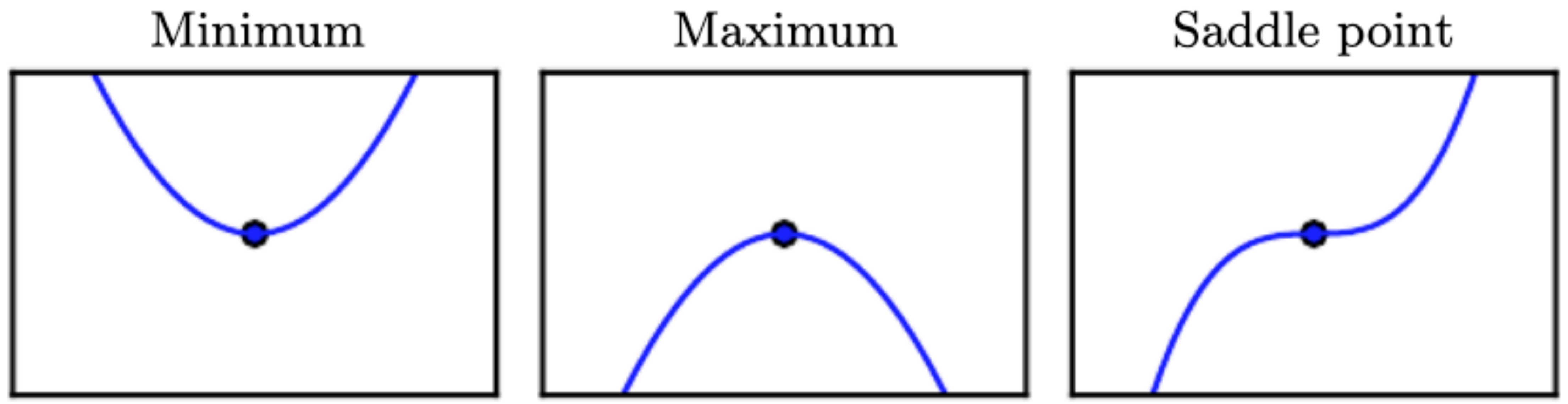
- Batch/deterministic method: process all the samples simultaneously
- Stochastic/online method: use only a single example at a time
  - Generalization error is often best.
  - Estimation is noisy, and need to carefully choose the learning rate.
- Mini-batch: use small part of data sampled from the entire data set
  - Standard error: less than linear returns (square root of n).
  - Small batches can offer a regularizing effect (perhaps due to the noise).
  - The noise is reduced (compared with stochastic method).
  - Hardware consideration:
    - Memory cost scales with the batch size.
    - Extremely small batches are usually underutilized by multicore architectures.
    - Trick: power of 2 batch sizes usually offer better runtime when using GPU.

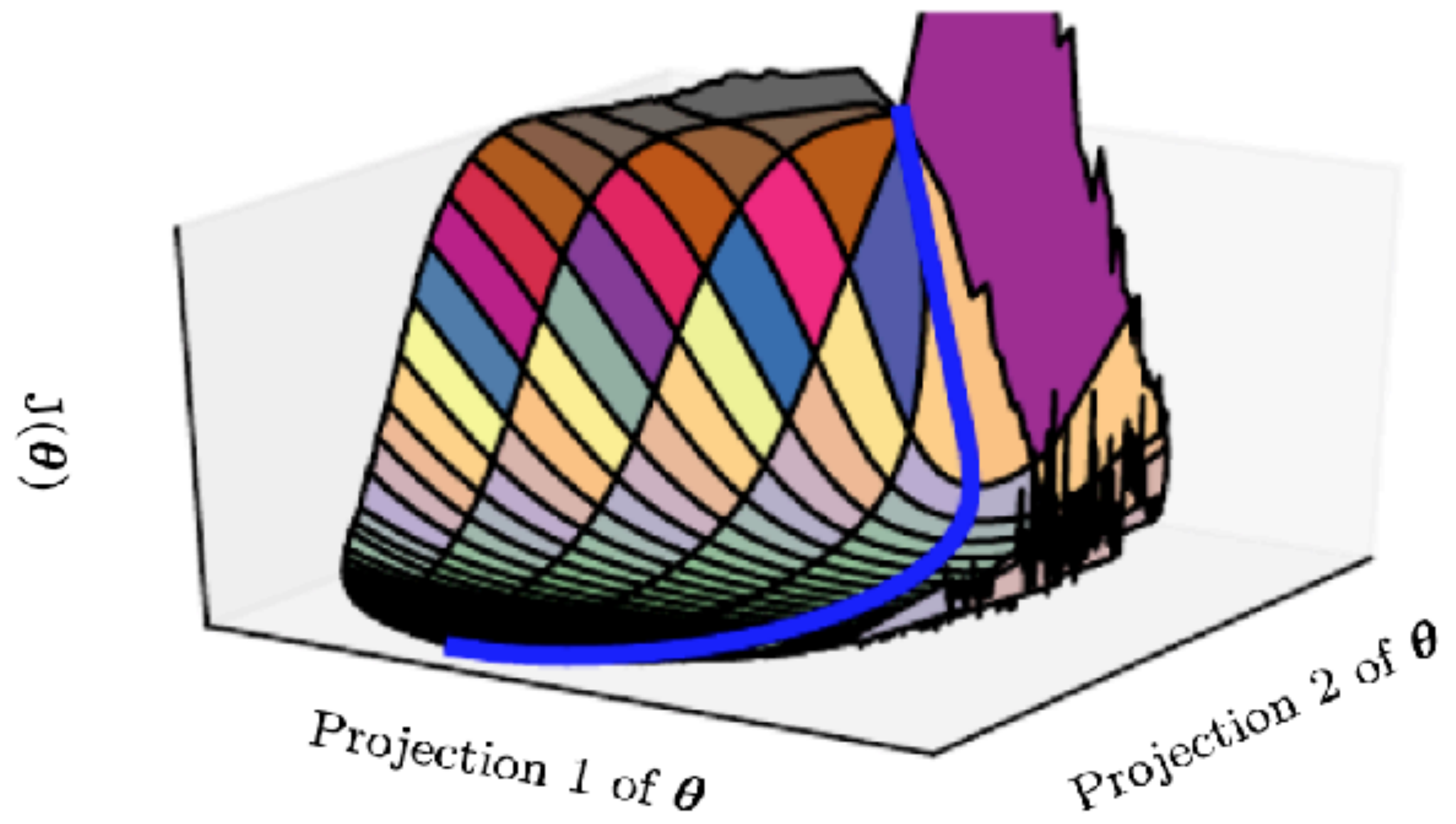
Challenges in NN optimization

- **Local minima** :
  - non-convex optimization
  - lots of local minima

- **Saddle points** in high-dim spaces
  - More common than local minima
  - Gradient-based methods seem to be able to escape empirically

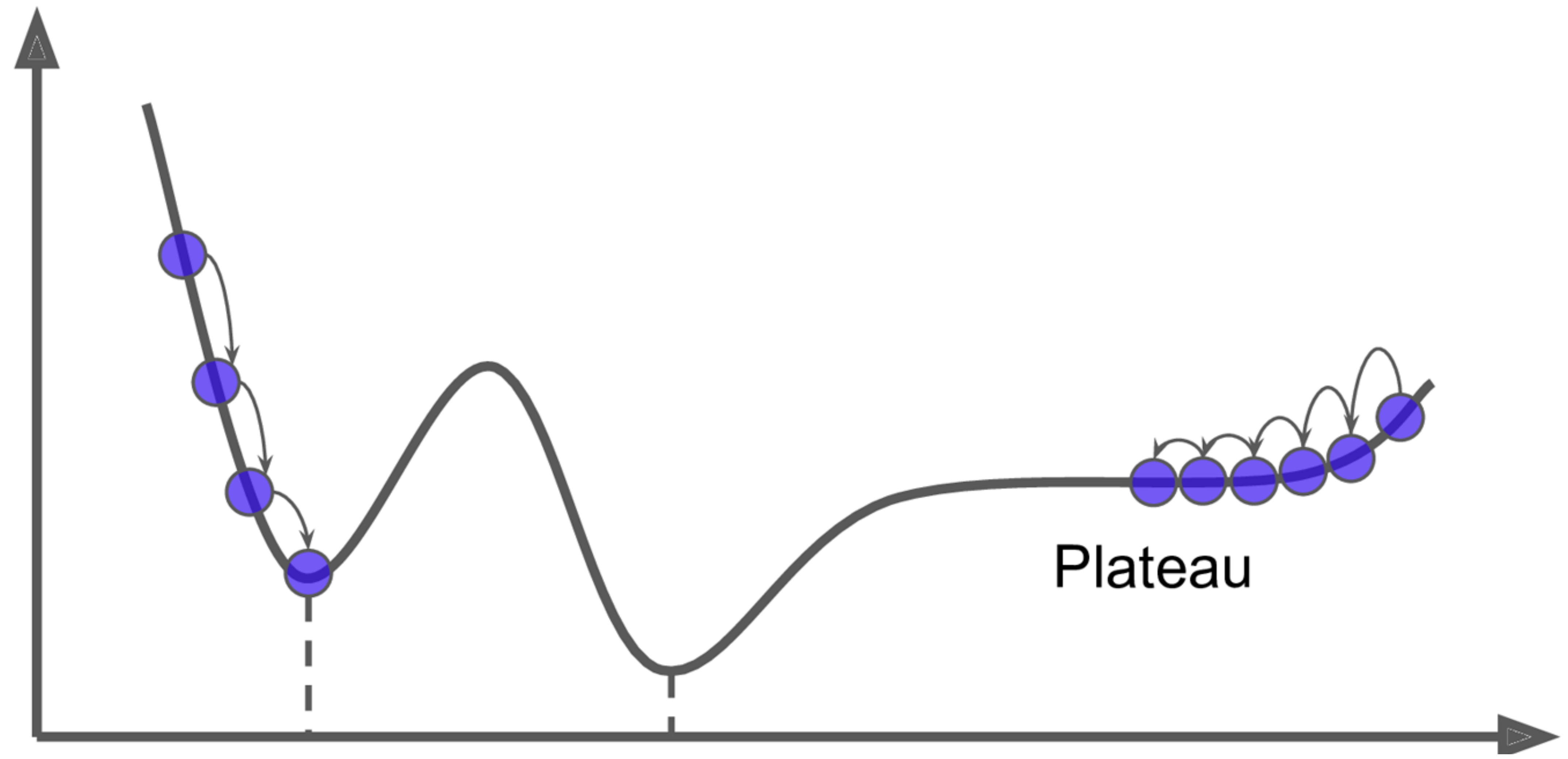


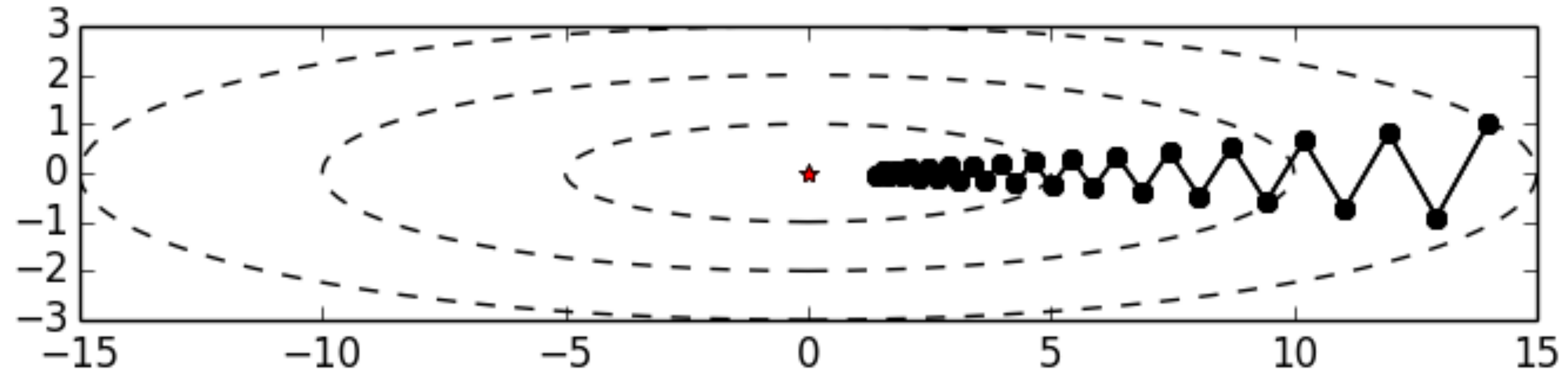




Challenges in NN optimization

Plateaus and ill-condition





Solutions: momentum and adaptive learning rate

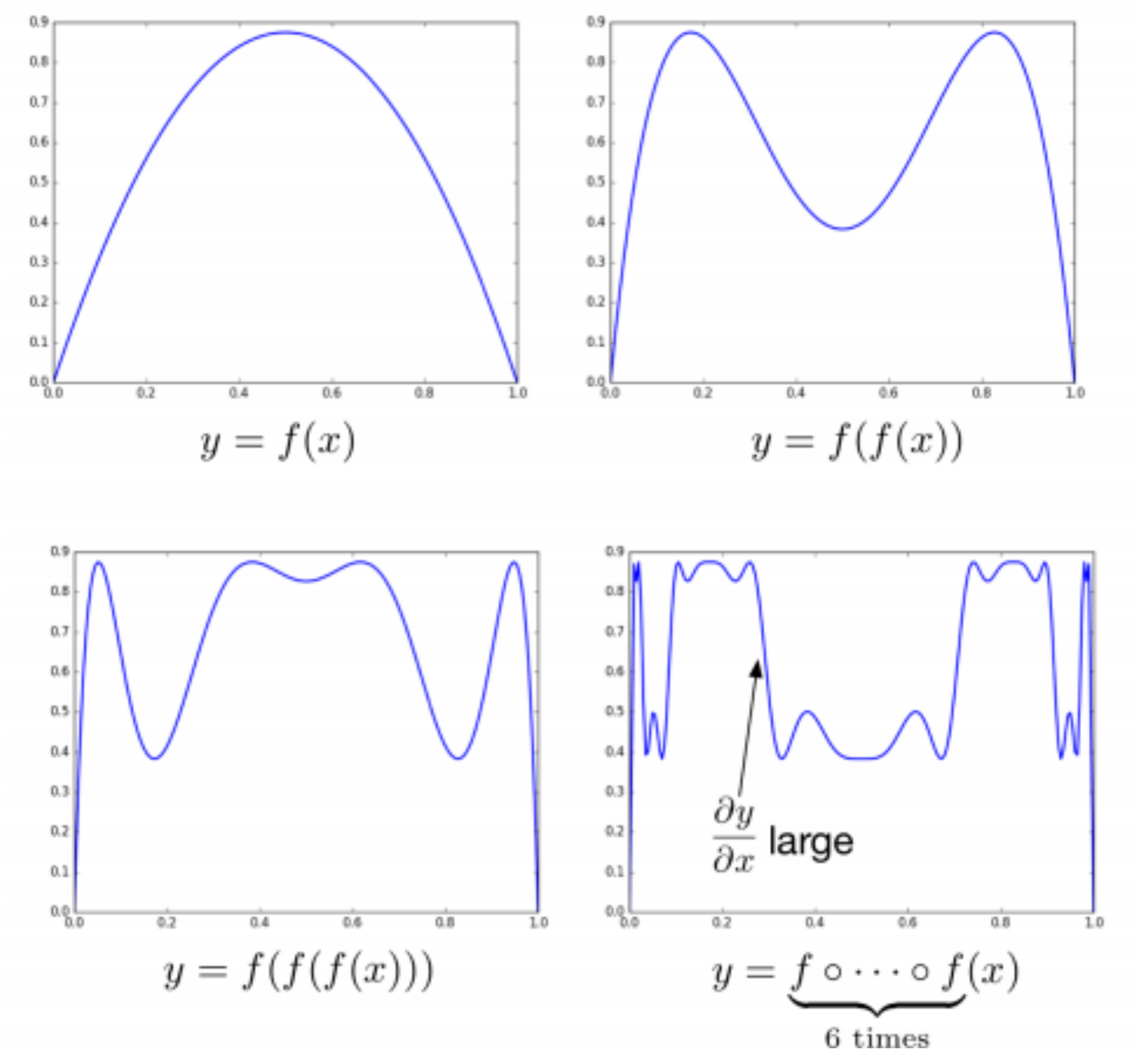
See details in “Deep Learning Book Chapter 8”

Challenges in NN optimization

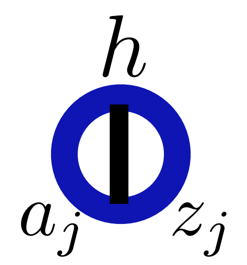
Vanishing and exploding gradients

- When a neural network is too deep

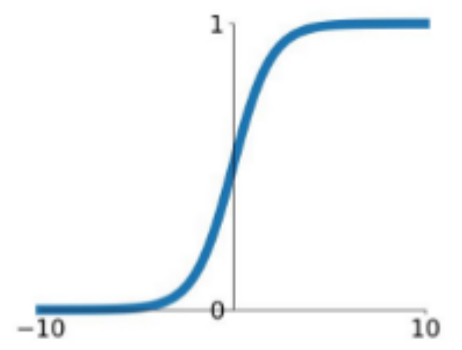
$$\boldsymbol{W}^t = \left(\boldsymbol{V}\mathrm{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\mathrm{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$$



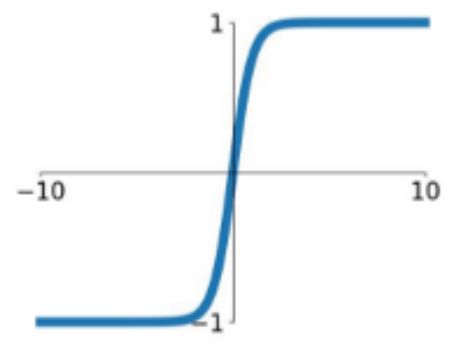
- When using sigmoid activation function



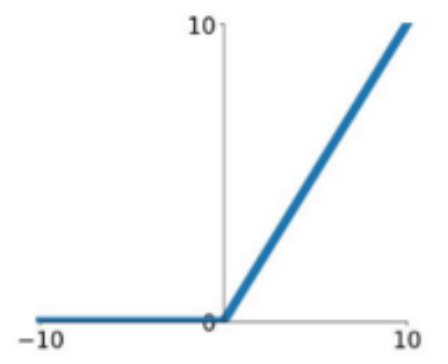
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



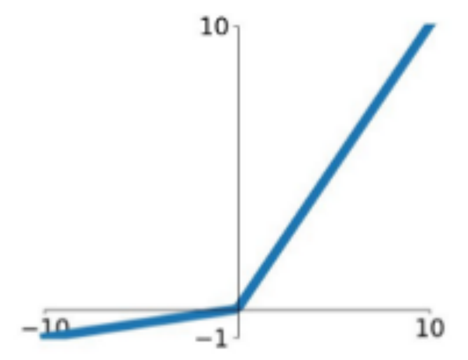
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

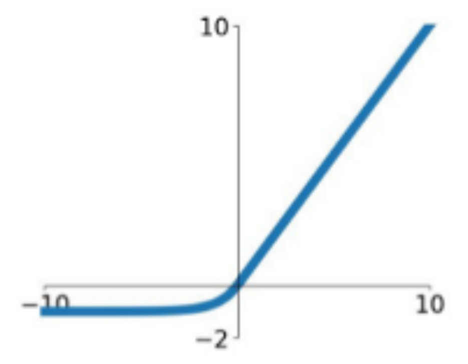


Leaky ReLU
$\max(0.1x, x)$

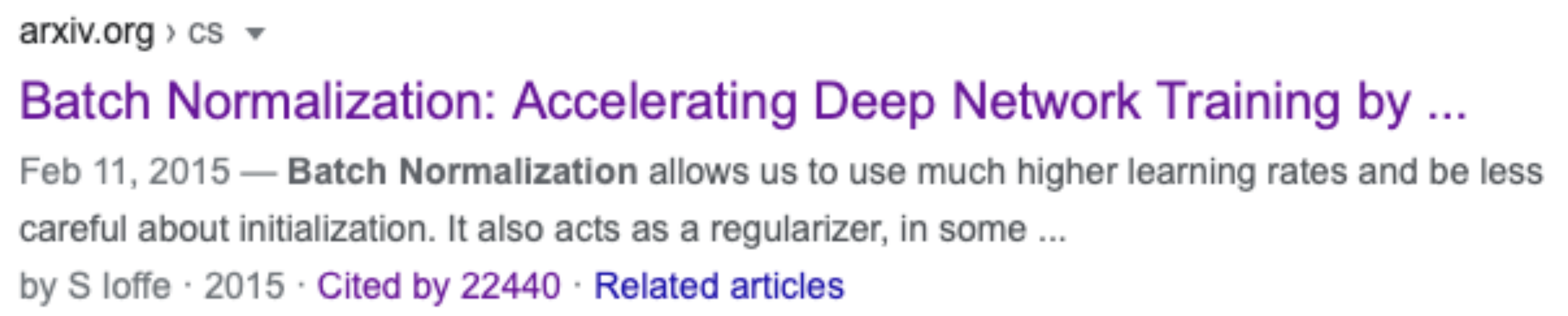


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Solutions:
Initialization; Gradient clipping; ResNets/LSTM; **Batch normalization**



arxiv.org › cs
Batch Normalization: Accelerating Deep Network Training by ...
Feb 11, 2015 — **Batch Normalization** allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some ...
by S Ioffe · 2015 · Cited by 22440 · Related articles

Optimization in deep learning

BP and beyond it

- BP: automation differentiation in neural network
- Mini-batch stochastic optimization
  - Mini-batch: dueling with big data
  - Stochastic: finding the “global” optimal points
- Gradient vanish or explosion
- Tricks
  - Batch normalization
  - Dropout
  - Regularization
  - Others: initializations, gradient clip, early stopping, …

Implementation of deep learning

Elements in deep learning

- *Data representation and organization*
- *Definition of the neural networks*
- *Optimization*
- Model store and reuse
- CPU - GPU
- Visualization
- C++ API…

Elements in deep learning

Data representation and organization

- *Data representation*
  - *Audio, language, images, parameters, …*
  - *transformations*
- *Data organization*
  - *Train, validation, test*
  - *Mini-batch optimization*
    - *Batch, mini-batch*
    - *Step, epoch*

Import torch

- *torch.tensor*
  - *Multi-dim matrix*
  - *Rich of functions for transformations, composition, change of shape…*
  - *Store the gradients*
- *torch.utils.data.[Dataset, DataLoader]*
  - *Dataset: __getitem__ and __len__*
  - *DataLoader: iterator*
    - *Shuffle dataset*
    - *Get Mini-batch dataset*

Elements in deep learning

Define neural networks

- *Neural networks*
  - *DNN, CNN, RNN, GNN*
  - *DIY network structure*
  - *Popular or latest structures in the community*

Import torch

- *torch.nn.Module*
  - *class YourNN(torch.nn.Module)*
  - *def forward(self, x)*
  - *Automatically store the parameters*
  - *Support composition*
  - *nn.Conv2d, nn.LSTM, nn.Embedding, …*

Elements in deep learning

Optimization

- *Automatic differentiation*
- *Optimizer*
  - *SGD*
  - *Adam*
  - *Adagrad*
  - *LBFGS*
  - *RMSprop, …*
- *Optimization tricks*
  - *Batch normalization*
  - *Dropout*
  - *Initialization …*

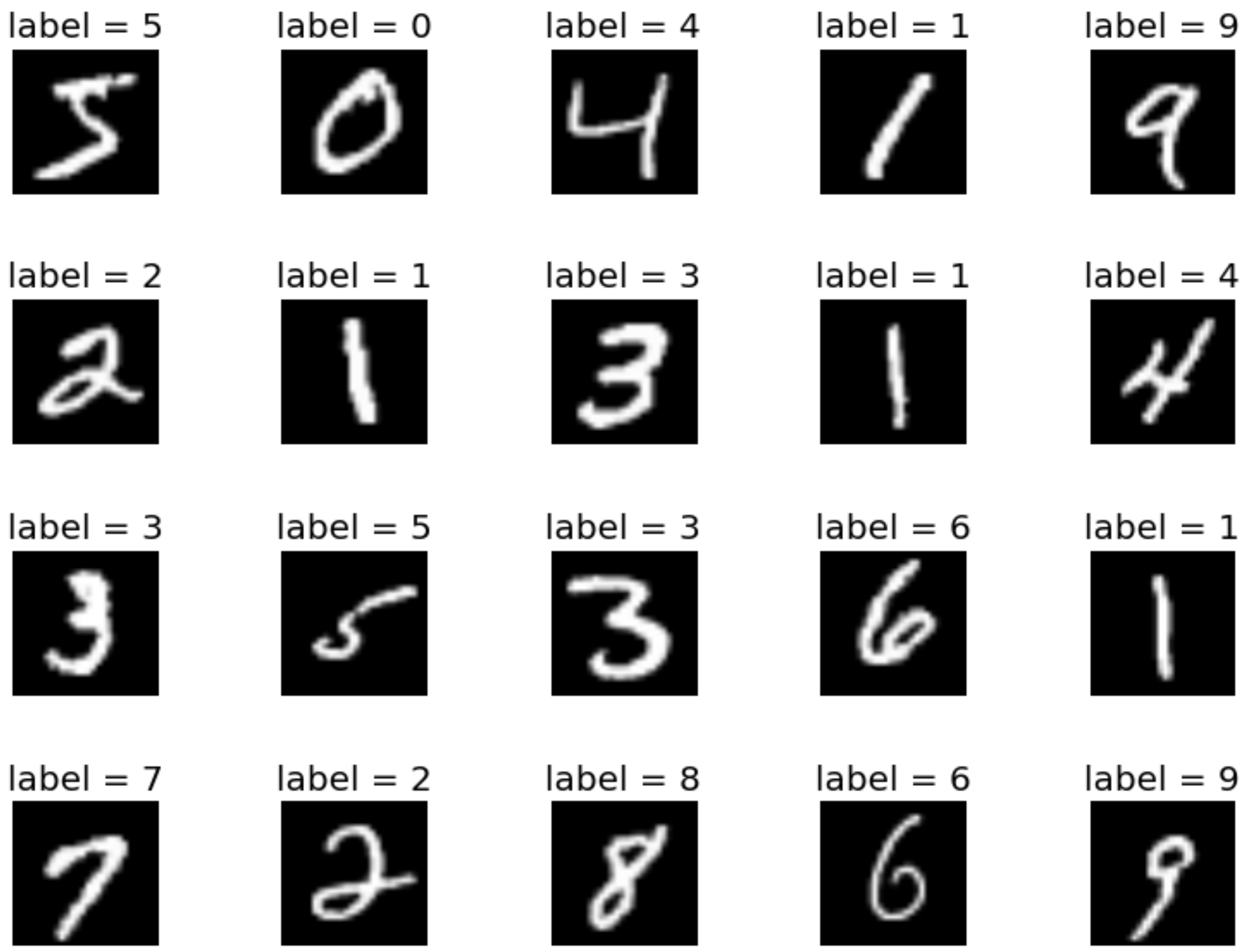
Import torch

- *torch.autograd, torch.Function, torch.Tensor*
  - *Autograd:*
    - *Backpropagation*
    - *Define-by-run*
  - *Tensor:*
    - *require_grad = True*
    - *.grad: accumulated the gradient*
    - *.grad_fn: refers to the torch.Function*
    - *.backward(): get the derivatives*
- *torch.optim*
  - *torch.optim.[optimizer_name](model.parameters(), other_arguments)*
- *torch.nn.[Normalization Layers]*
- *torch.nn.[Dropout Layers]*
- *torch.nn.init.[Methods]*

Example: VAE on MNIST

Example: VAE on MNIST

Non-linear low-dimensional representation learning



The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

https://github.com/beyondpie/intro_nn_with_torch/tree/main/homework

VAE on MNIST dataset

In this homework, we play with VAE model on the MNIST dataset. The materials are from https://github.com/pytorch/examples/blob/master/vae/main.py.

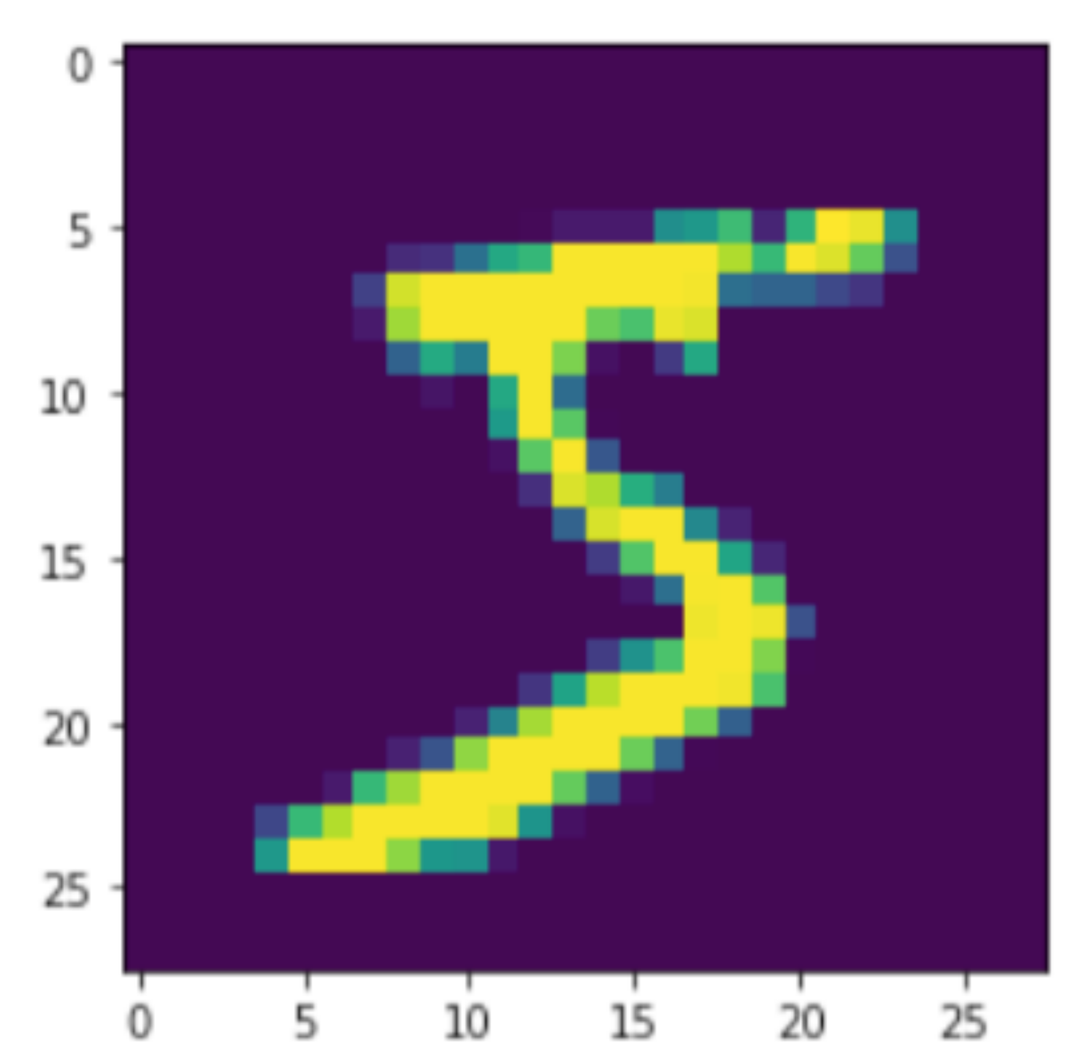
- We use python 3.0
- With more explanations

```python
# load the modules

## load pytorch module
import torch

## load torch data manager: Dataset, DataLoader
from torch.utils.data import Dataset, DataLoader

## load neural network module
from torch import nn
from torch.nn import functional as F

## load optimizer
from torch import optim

## load modules for MNIST dataset
from torchvision import datasets, transforms
from torchvision.utils import save_image

## load other python modules
### plt to view the image
import matplotlib.pyplot as plt
```

load MNIST dataset

In [18]:
```
## download MNIST trainign dataset
train_dataset: Dataset = datasets.MNIST("./data", train = True, download = True,
                                        transform = transforms.ToTensor())
## class Dataset provides:
## - function "__len__": to get the size of the dataset
## - function "__getitem__": to get the data point by the index

## show the data size
print(train_dataset)
## show one data point
## each data point has two element: one is the image, one is which number it is.

## print the tensor size: (1, 28, 28):
## the first dim: 1 is the number of channel (only one) here
## the second and third dim: the width and height for this channel.
print(train_dataset[0][0].shape)

## print the number
print(train_dataset[0][1])
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: ToTensor()
torch.Size([1, 28, 28])
5
```



In [24]:
```
## let's view this image
plt.imshow(train_dataset[0][0][0])
```

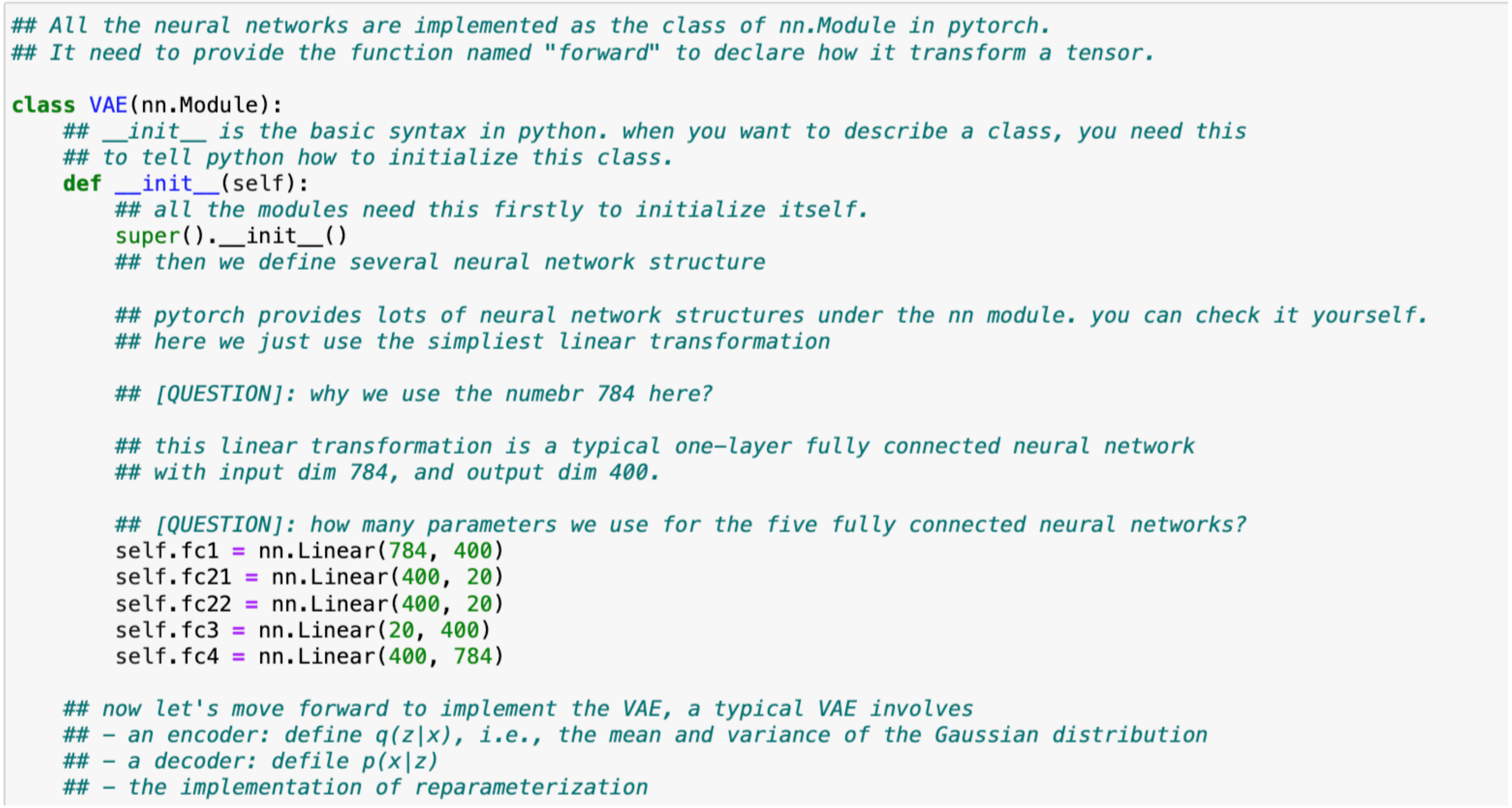
Out[24]:
```
<matplotlib.image.AxesImage at 0x7f8749393e50>
```

Use Dataloader to wrap up the dataset

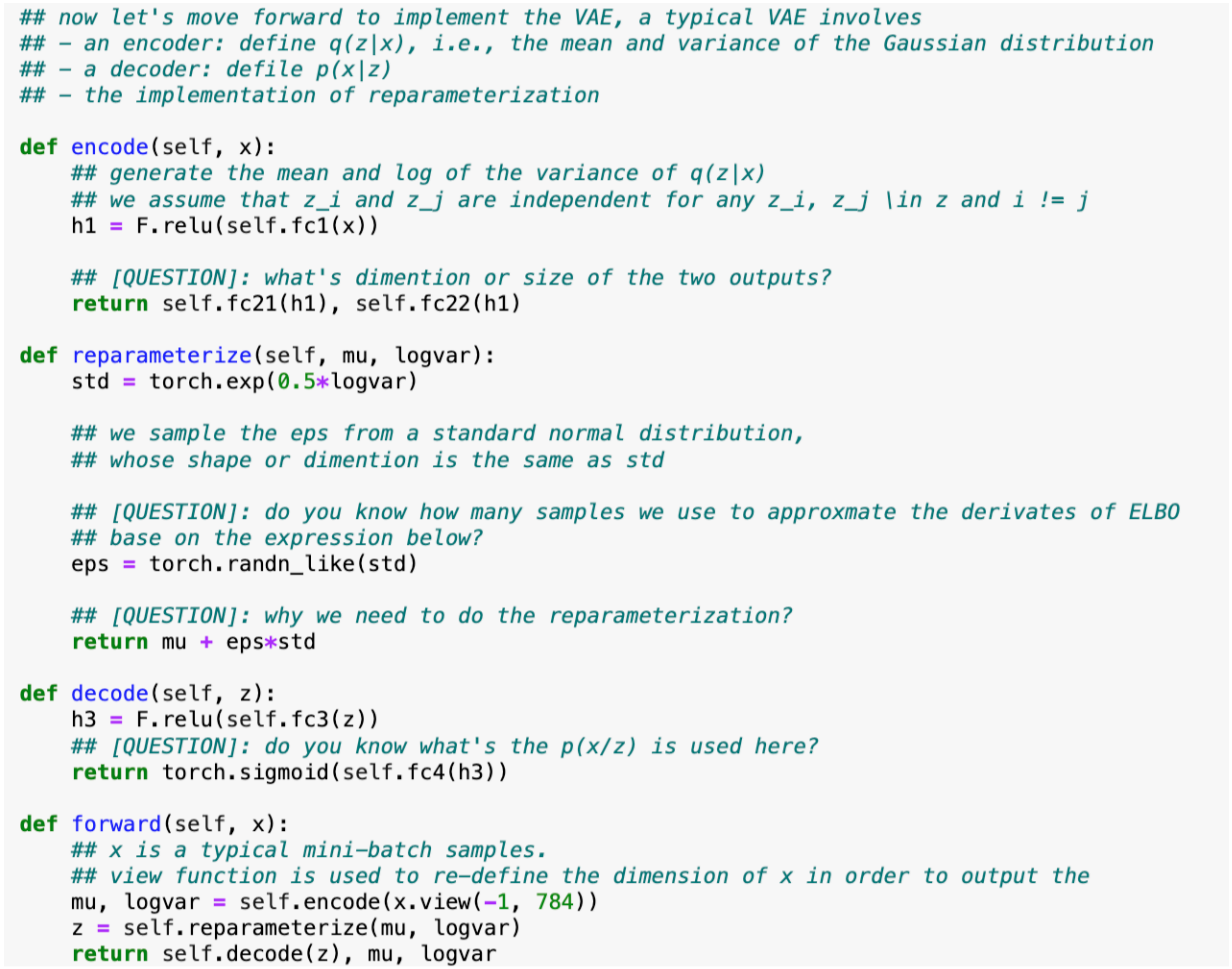
During the learning process, we will use stochasitic optmization, which means we will shuffle the sample, and put a small part of the dataset into the optimizer. You can do this by yourself, but pytorch actually provides the data manager named Dataloader to simplify this process, and it is usually used together with the Dataset class.

```
:  ## you can choose whatever batch_size you want.
   ## shuffle = True, means every epoch, we will shuffle the samplers
   ## by the sampling algorithm you or the default one.
   train_loader = DataLoader(train_dataset, batch_size = 32, shuffle = True)
   ## we don't need to shuffle the samples during test.
   test_loader = DataLoader(test_dataset, batch_size = 32, shuffle = False)
```

VAE



```
## All the neural networks are implemented as the class of nn.Module in pytorch.
## It need to provide the function named "forward" to declare how it transform a tensor.

class VAE(nn.Module):
    ## __init__ is the basic syntax in python. when you want to describe a class, you need this
    ## to tell python how to initialize this class.
    def __init__(self):
        ## all the modules need this firstly to initialize itself.
        super().__init__()
        ## then we define several neural network structure

        ## pytorch provides lots of neural network structures under the nn module. you can check it yourself.
        ## here we just use the simpliest linear transformation

        ## [QUESTION]: why we use the numebr 784 here?

        ## this linear transformation is a typical one-layer fully connected neural network
        ## with input dim 784, and output dim 400.

        ## [QUESTION]: how many parameters we use for the five fully connected neural networks?
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    ## now let's move forward to implement the VAE, a typical VAE involves
    ## - an encoder: define q(z|x), i.e., the mean and variance of the Gaussian distribution
    ## - a decoder: defile p(x|z)
    ## - the implementation of reparameterization
```



```
## now let's move forward to implement the VAE, a typical VAE involves
## - an encoder: define q(z|x), i.e., the mean and variance of the Gaussian distribution
## - a decoder: defile p(x|z)
## - the implementation of reparameterization

def encode(self, x):
    ## generate the mean and log of the variance of q(z|x)
    ## we assume that z_i and z_j are independent for any z_i, z_j \in z and i != j
    h1 = F.relu(self.fc1(x))

    ## [QUESTION]: what's dimention or size of the two outputs?
    return self.fc21(h1), self.fc22(h1)

def reparameterize(self, mu, logvar):
    std = torch.exp(0.5*logvar)

    ## we sample the eps from a standard normal distribution,
    ## whose shape or dimention is the same as std

    ## [QUESTION]: do you know how many samples we use to approxmate the derivates of ELBO
    ## base on the expression below?
    eps = torch.randn_like(std)

    ## [QUESTION]: why we need to do the reparameterization?
    return mu + eps*std

def decode(self, z):
    h3 = F.relu(self.fc3(z))
    ## [QUESTION]: do you know what's the p(x/z) is used here?
    return torch.sigmoid(self.fc4(h3))

def forward(self, x):
    ## x is a typical mini-batch samples.
    ## view function is used to re-define the dimension of x in order to output the
    mu, logvar = self.encode(x.view(-1, 784))
    z = self.reparameterize(mu, logvar)
    return self.decode(z), mu, logvar
```

Define the loss function

```
## Now let's define the loss function of VAE.
## There are two parts:
## - the loss of reconstructio, i.e., - p(x|z)
## - the KL divergence towards the prior, i.e., KL(q(z|x) ||  p(z))

## binary cross entropy loss
reconst_loss = nn.BCELoss(reduction = 'sum')

def loss_function(recon_x, x, mu, logvar):
    rcloss = reconst_loss(recon_x, x.view(-1, 784))
    ## analytic formulation of KL divergence of Gaussian distribution.
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return rcloss + kld
```

Select the optimizer

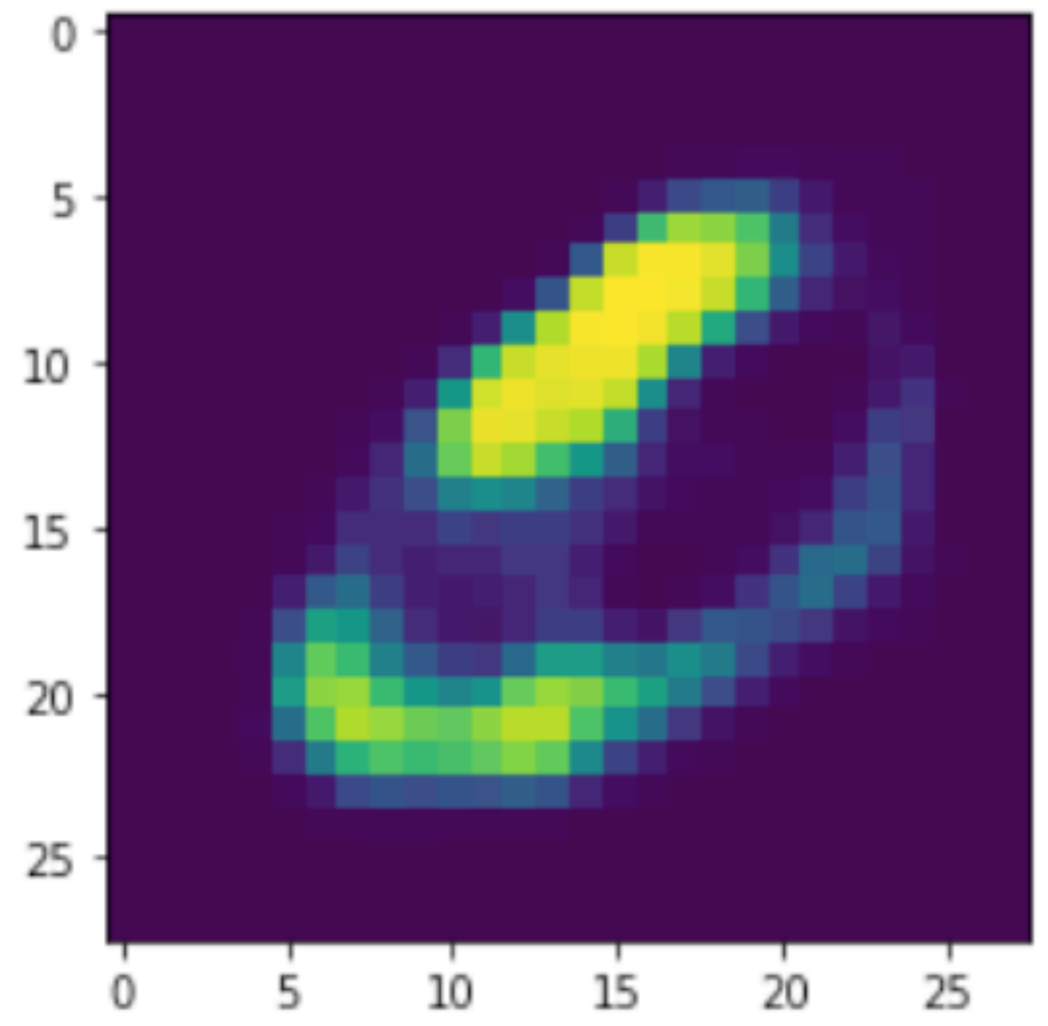
```
## lots of the optimizers you can select in torch.
## nowadays, people use Adam a lot.
## An interesting thing: the first author of Adam is the same first author of VAE.
optimizer = optim.Adam(model.parameters(), lr = 1e-3)
```

Let's start to train VAE!

```
## Epoch: once we run through all the samples in the training dataset, we finish one epoch.
def train(epoch):
    ## [IMPORTANT]: in pytorch, model will update the parameters when it's in training state.
    model.train()
    train_loss = 0
    ## we ignore the labels of MNIST, and only use the images.
    for batch_id, (x, _) in enumerate(train_loader):
        ## [IMPORTANT]: each differentiable tensor in pytorch will recognize the previous gradient.
        ## so we have to set them as zeros before we compute next time.
        optimizer.zero_grad()

        recon_batch, mu, logvar = model(x)
        loss = loss_function(recon_batch, x, mu, logvar)

        ## we do the back propagation on the tensor generated by the loss function.
        loss.backward()

        ## then update the parameters by the optmizer.
        optimizer.step()

        ## let's remember the loss
        train_loss += loss.item()

        ## print the loss
        if batch_id % 500 == 0:
            print(f"Train Epoch {epoch} [{batch_id}]: loss {loss.item() / len(x)}")
    print(f"Epoch: {epoch} average loss: {train_loss / len(train_loader.dataset)}")


for epoch in range(10):
    train(epoch)
```

Let's view the results

```
## no grad means the computations below will not be used for optimization.
%matplotlib inline
with torch.no_grad():
    for _ in range(10):
        sample = torch.randn(1, 20)
        rdm_image = model.decode(sample).view(28, 28)
        plt.imshow(rdm_image)

## [QUESTION*]: can you show the generated images given a specific number?
```



0
5
10
15
20
25
0
5
10
15
20
25