CoE202
Fundamentals of Artificial intelligence
<Big Data Analysis and Machine Learning>

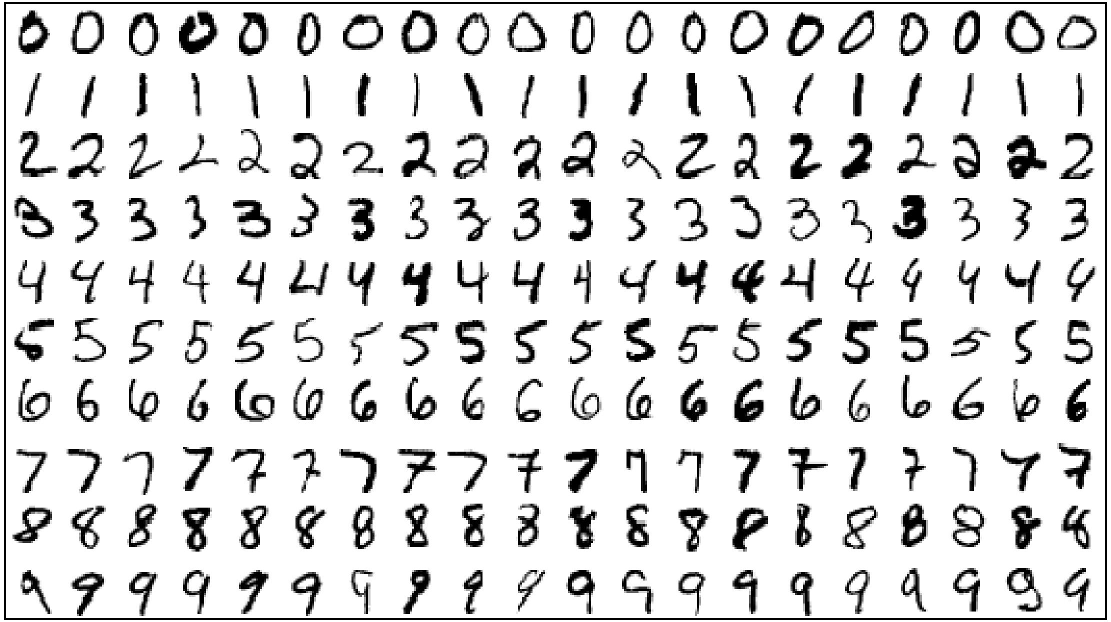
Deep Learning with libraries

Prof. Young-Gyu Yoon
School of EE, KAIST

Contents

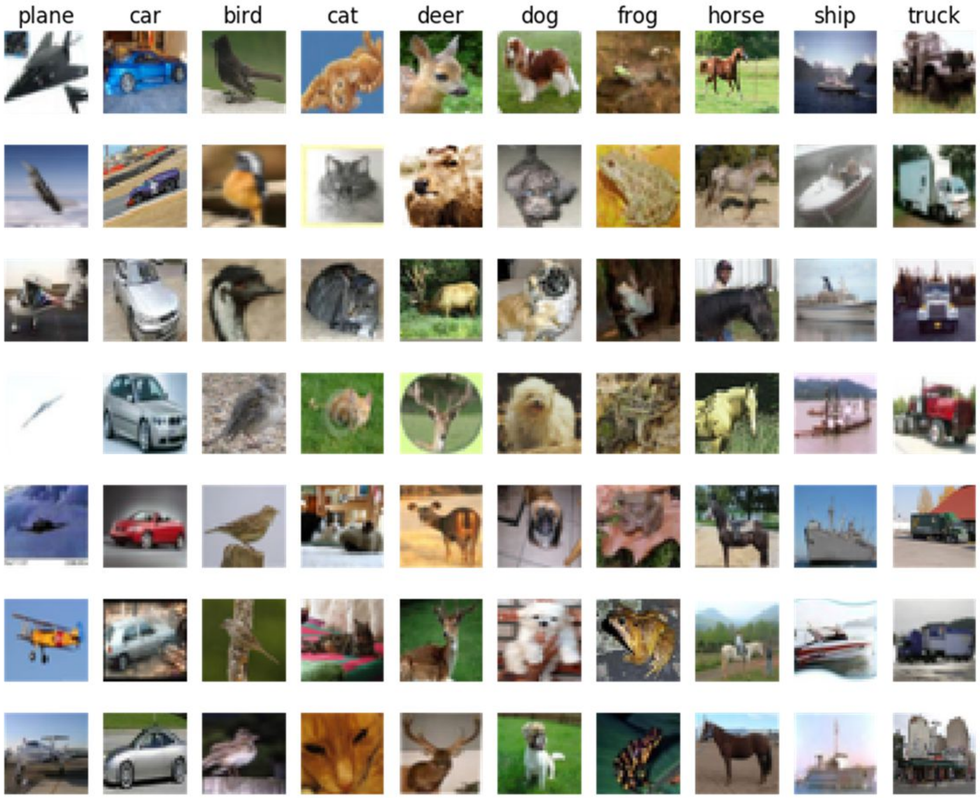
- Recap
  - Backpropagation
- Deep learning with libraries
  - Libraries
  - What we (human) do and what library does
  - Task formulation
  - NN design flow

With deep learning



- MNIST digit recognition
  - 70,000 images
  - Image size: 28 x 28 pixels
  - Number of classes: 10

With deep learning



- CIFAR-10 image classification
  - 60,000 images
  - Image size: 32 x 32 pixels
  - Number of classes: 10

Deep learning libraries

TensorFlow

- Developed by Google
- Good for professional use

PyTorch

- Developed by Facebook
- Easy to use

theano

- Developed by University of Montreal
- Good for professional use

Keras

- Developed by a Google engineer
- Interface for Tensorflow, R, Theano, …

Caffe

- Developed by University of California, Berkeley
- Caffe2 was merged into PyTorch

Why use library?

- **For example, with PyTorch**
  - Complicated “forward function” can be easily implemented by **calling built-in functions**
  - Lots of **loss functions** are implemented. We can just call them without having to worry about implementation
  - Lots of **optimization methods** are included (more than just SGD!)
  - **GPU acceleration** is supported
  - Supports **Autograd**
    - We just need to define the “forward propagation”
    - We don’t have to worry about complicated gradient calculation anymore. **Backpropagation is automatically done** for us ☺

Why use library?

- **For example, with PyTorch**
  - We can **download lots of datasets** through torchvision that are suited for studying deep learning
  - **Popular network architectures** are included
  - Useful (and common) **image transformation** functions are included
  - Come with data loading & shuffling functions
  - … and a lot more

Why use library?

- **Simply put, Pytorch can do (almost) everything for us**
  - It allows us to implement neural networks without even understanding
    - How to do backpropagation and calculate gradient
    - How GD (and other optimization method) works
    - How to calculate a loss function
    - …and the list goes on

Why use library?

- Does it really mean we don't have to know anything?
- Then, why are we "wasting" our time learning all this?

- The library
  - allows us to focus on the concept and innovation (rather than details inside)
  - does NOT debug itself: we still have to understand how things work (in general) to be able to fix, debug and improve things

PyTorch vs. Numpy

```python
class mlp_classifier(torch.nn.Module):
    # initialization
    def __init__(self):
        super(mlp_classifier, self).__init__()
        self.layer1 = torch.nn.Linear(2, 10)
        self.layer2 = torch.nn.Linear(10, 10)
        self.layer3 = torch.nn.Linear(10, 10)
        self.layer4 = torch.nn.Linear(10, 1)
        self.relu = torch.nn.ReLU()
```

Define network

```python
    # forward path
    def forward(self, x):
        # Used relu to add non-linearity
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.layer3(x)
        x = self.relu(x)
        x = self.layer4(x)
        return x
```

Define forward propagation

With PyTorch

vs.

Initialize parameters

```python
class nonlinear_classifier():

    # set initial weights
    def __init__(self, W1_init, W2_init):
        super(nonlinear_classifier, self).__init__()
        self.W1 = W1_init
        self.W2 = W2_init
```

Define forward propagation

```python
    # forward pass
    def forward(self, x):

        # xk : input, zk: Wxk+b, yk = h(zk) (h is the activation function) @ kth layer
        self.x1 = x  # input of the 1st layer = input of the network
        self.y1, self.z1 = self.forward_single_layer(self.W1, self.x1)
        self.x2 = self.y1 # input of the 2nd layer is the output of the first layer
        self.y2, self.z2 = self.forward_single_layer(self.W2, self.x2)
        y = self.y2 # output of the network = output of the 2nd layer

        return y # return network output
```

Define backpropagation

```python
    def backward(self, label):

        y = self.y2

        # calculate loss and accuracy
        loss = cross_entropy(y, label)
        loss_avg = np.mean(loss)
        prediction_threshold = (y>0.5)
        accuracy = np.mean(prediction_threshold==label)
        n_sample = label.shape[1]

        # back-propagate
        dLdy = (y-label)/(y*(1-y))
        dLdy, dLdw2 = self.backward_single_layer(self.W2, self.x2, self.y2, dLdy)
        dLdy, dLdw1 = self.backward_single_layer(self.W1, self.x1, self.y1, dLdy)

        return loss_avg, accuracy, dLdw1, dLdw2
```

Define single layer forward propagation

```python
    def forward_single_layer(self, W, x):

        # x : input, z: Wx+b, y = h(z) (h is the activation function)

        x_pad = np.concatenate( (x, np.ones((1, x.shape[1]))), axis=0) # x_pad = [x; 1]
        z = np.matmul(W,x_pad) # z = [W b] X x_pad
        y = 1/(1 + np.exp(-z)) # y = sigmoid(z)

        return y, z
```

Define single layer backpropagation

```python
    def backward_single_layer(self, W, x, y, dLdy):

        n_data = x.shape[1]

        dLdz = dLdy*y*(1-y) # backprop sigmoid
        x_pad = np.concatenate( (x, np.ones((1, x.shape[1]))), axis=0) # add ones (for bias term)
        dzdw = x_pad.T      # take transpose
        dzdx = W[:,:-1].T  # exclude bias part then transpose

        dLdw = np.matmul(dLdz,dzdw)/n_data # divide by the number of data points (average)
        dLdx = np.matmul(dzdx,dLdz)

        return dLdx, dLdw
```

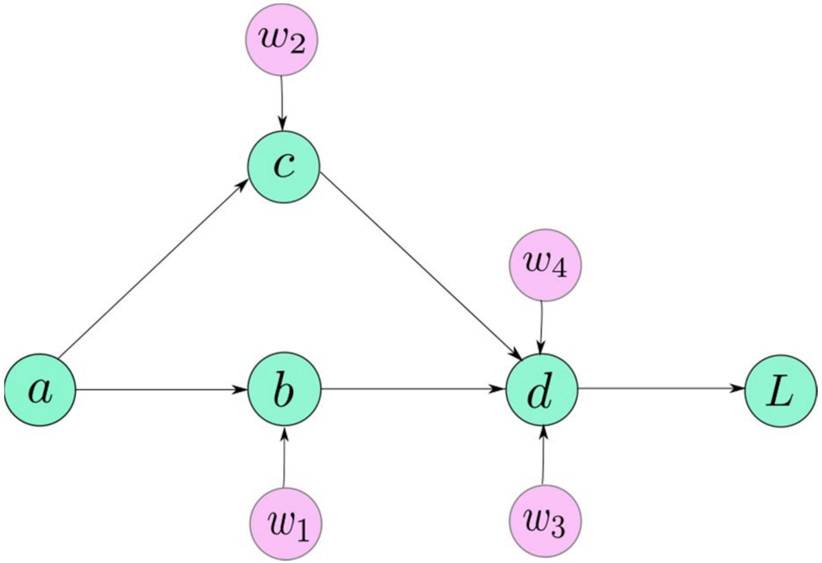
Parameter updata
(GD algorithm not included)

```python
    # update parameters
    def update(self, dW1, dW2):

        self.W1 = self.W1 + dW1
        self.W2 = self.W2 + dW2
```

Using just Numpy

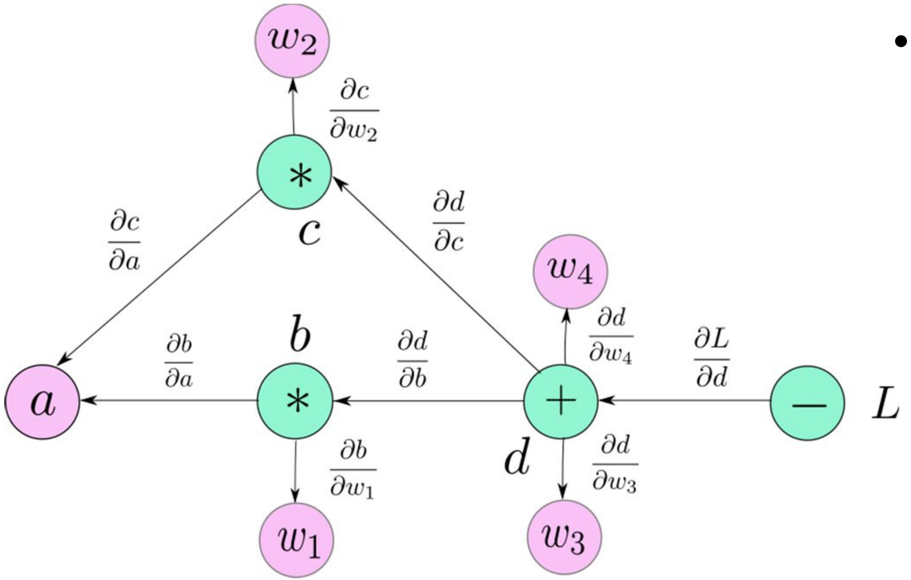
PyTorch



- **What users do:**
  - Define
    - the forward function (equivalent to a network or a directed graph**)**

From Ayoosh Kathuria's document

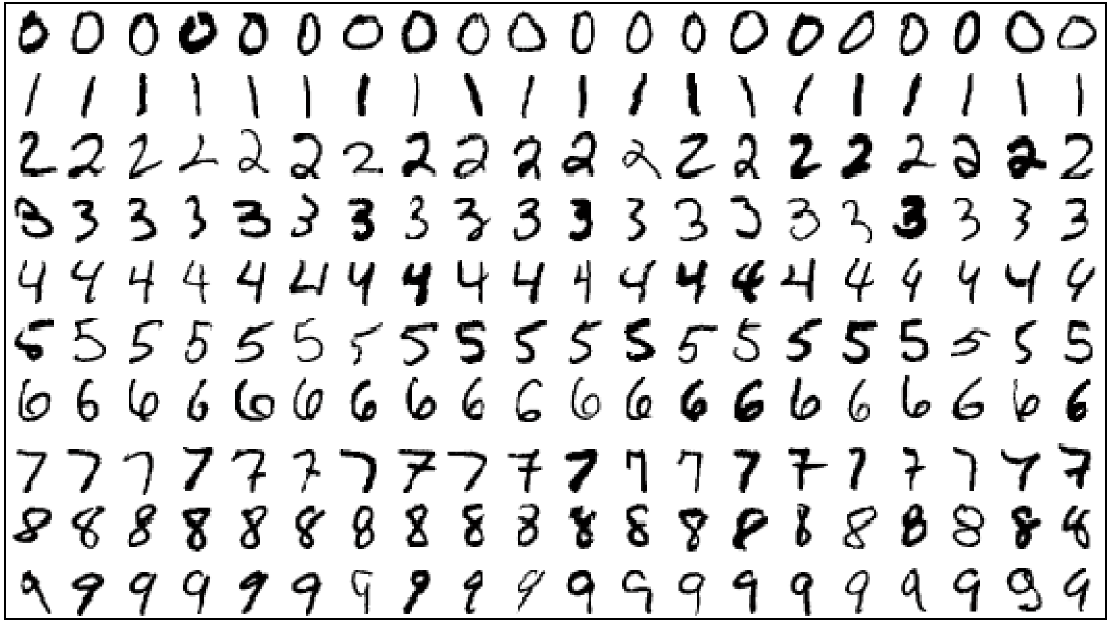
PyTorch



- **What PyTorch does:**
  - Calculate all partial derivatives and do backpropagation
  - …and update parameters using the algorithm and loss we choose
    - Of course we can build and use our own training algorithms and/or loss functions

From Ayoosh Kathuria's document

MINIST classification



X

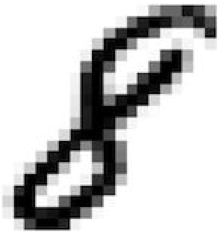
$\xrightarrow{f}$

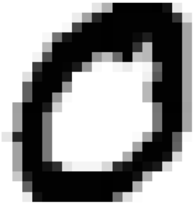
$0 \sim 9$

Y

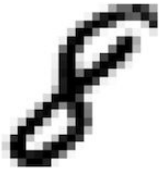
MINIST classification

- We want to build a function (neural network) that maps …

$f($  $) = 8$

$f($  $) = 0$

MINIST classification



Input: 28 x 28 digital image (or 28 x 28 array)

Output: 10x1 vector (or 10 x 1 array), one-hot encoded

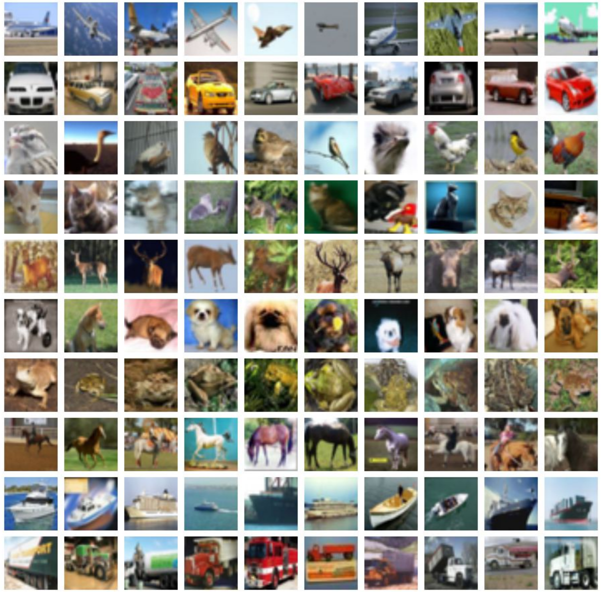
0	**1**	**2**	**3**		**9**
1	0	0	0		0
0	1	0	0		0
0	0	1	0		0
0	0	0	1		0
0	0	0	0	…	0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		1

Loss: cross entropy loss (as we did for linear classification)

Beware: If you are using Pytorch and torch.nn.CrossEntropyLoss to calculate the loss, you should NOT convert the ground truth (target) label to one-hot encoded vector. It simply takes a integer number as the ground truth label.

What about CIFAR-10 classification?

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck



Input: 32 x 32 digital image (or 32 x 32 x 3 array)

$\xrightarrow{f}$

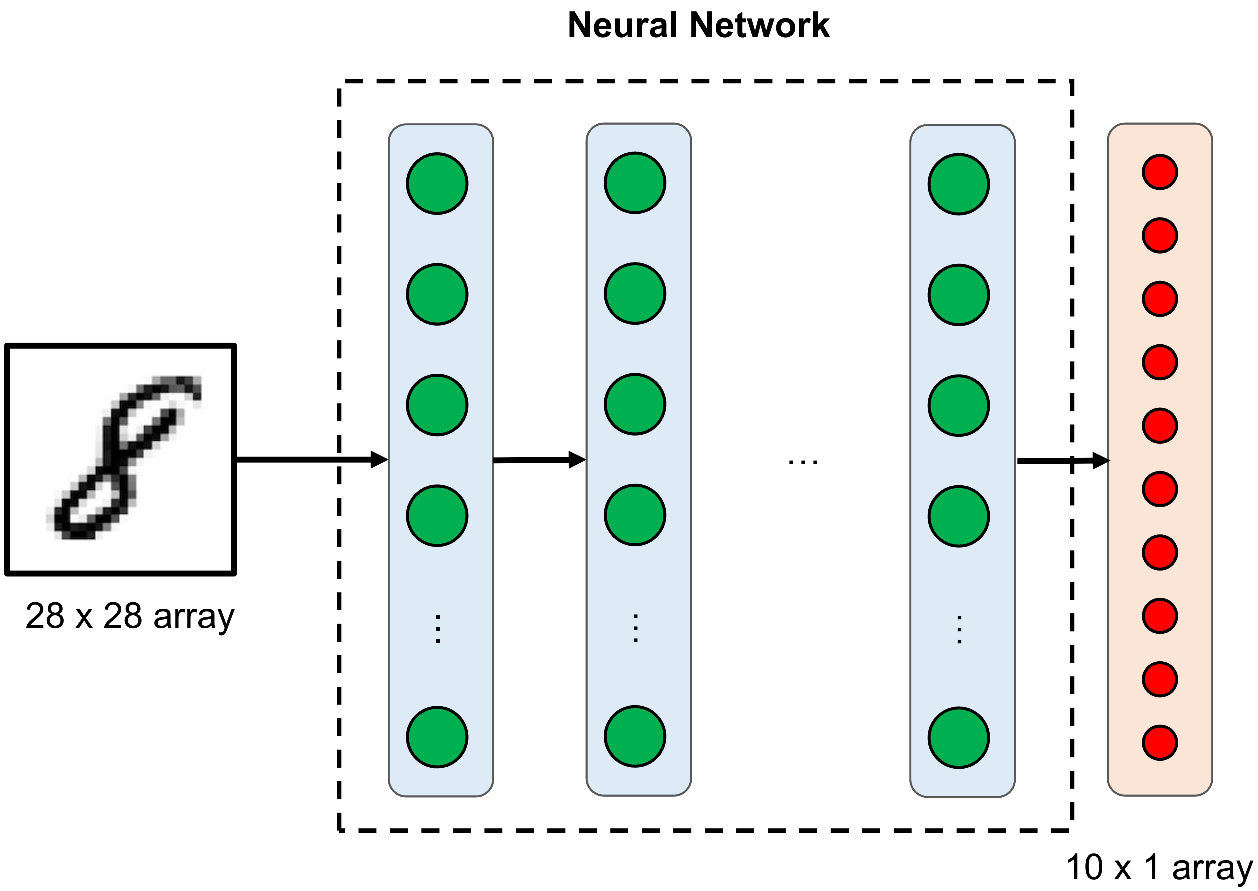
plane	car	bird	cat		truck
1	0	0	0		0
0	1	0	0		0
0	0	1	0		0
0	0	0	1	…	0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		0
0	0	0	0		1

Output: 10x1 vector (or 10 x 1 array), one-hot encoded

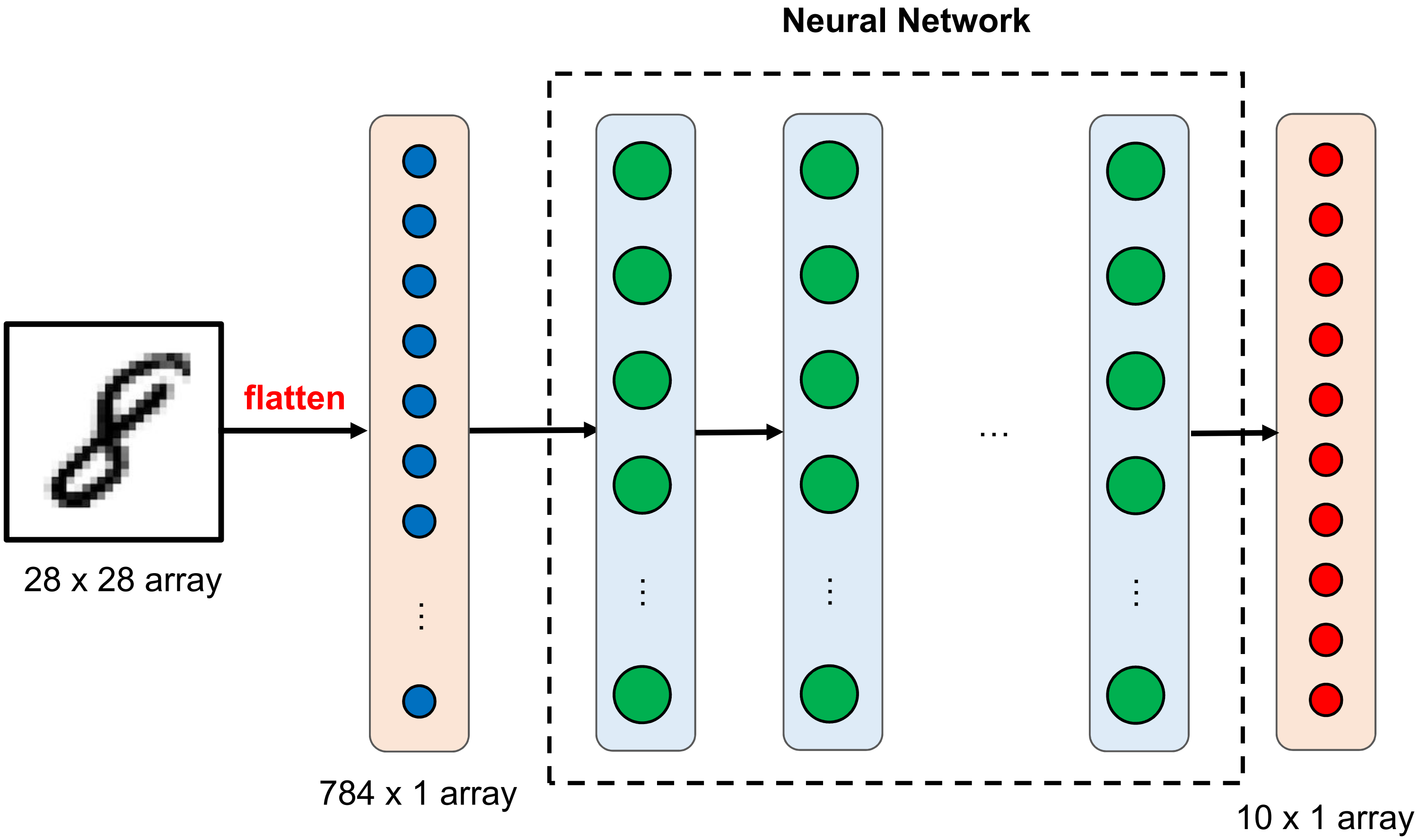
Loss: cross entropy loss (as we did for linear classification)

Almost everything is the same as MNIST classification …

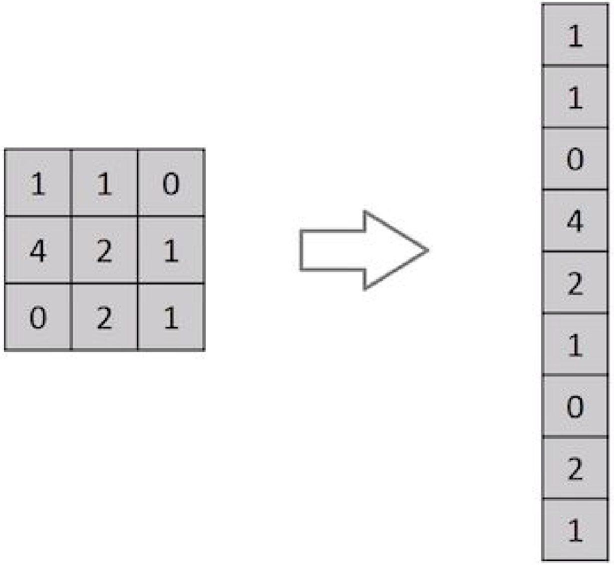
Formulation: MINIST classification



Formulation: MINIST classification



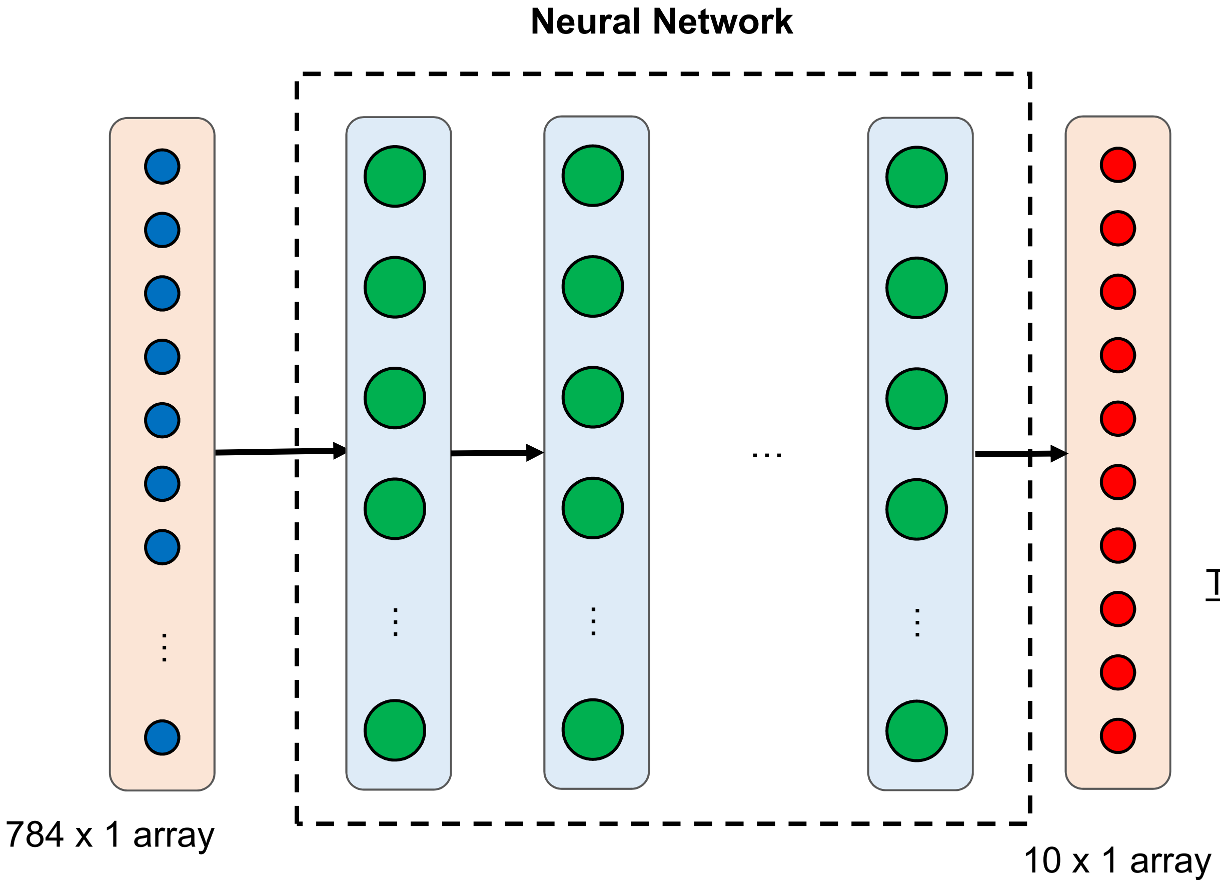
Flatten?



Flattening:
reshaping an array into a 1-dimensional array

3 x 3 → 9 x 1
28 x 28 → 784 x 1
15 x 15 x 3 → 675 x 1

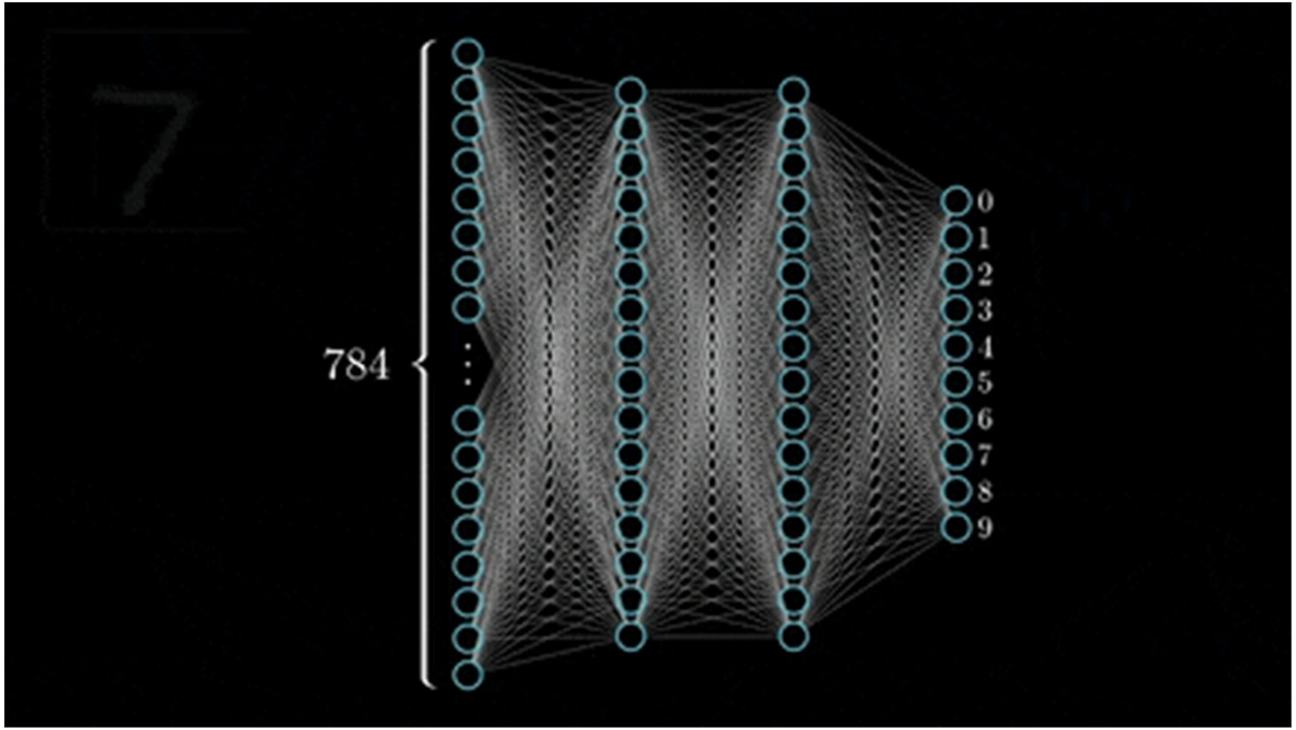
Formulation: MINIST classification



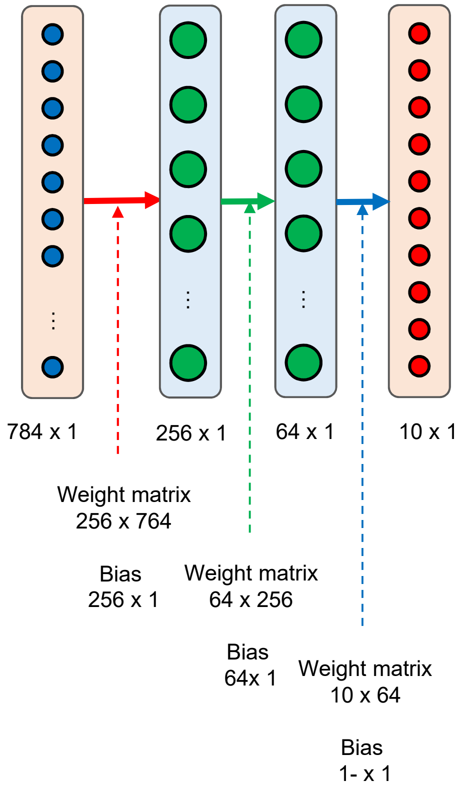
For MNIST classification,
We need to design a network
that takes 784x1 array as the input
and generates 10x1 array as the output
(**design constraint**)

<u>There are infinitely many possible options</u>

Formulation: MINIST classification



Neural Network Design

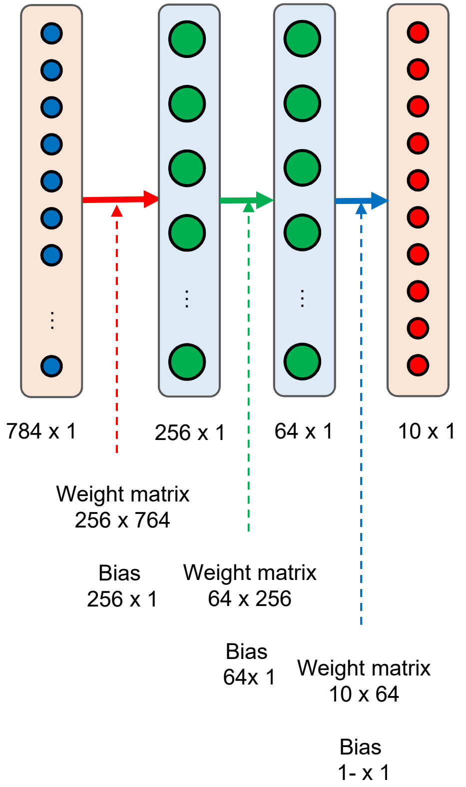


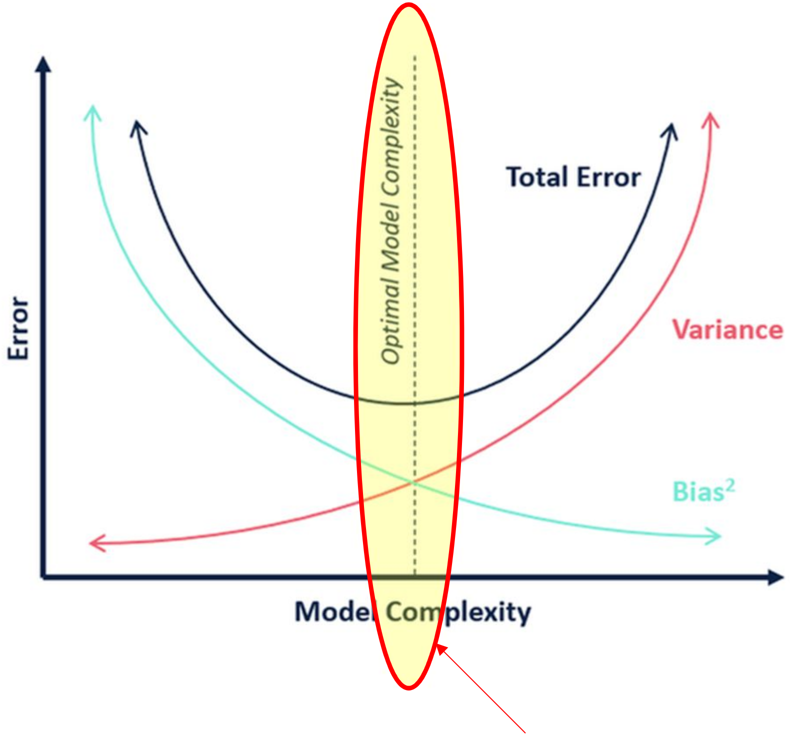
Example design

Input: 784 x 1
1st layer: 256 x 1 neurons
2nd layer: 64 x 1 neurons
3rd layer (output): 10 x 1 neurons

How do we know if this is a good design?

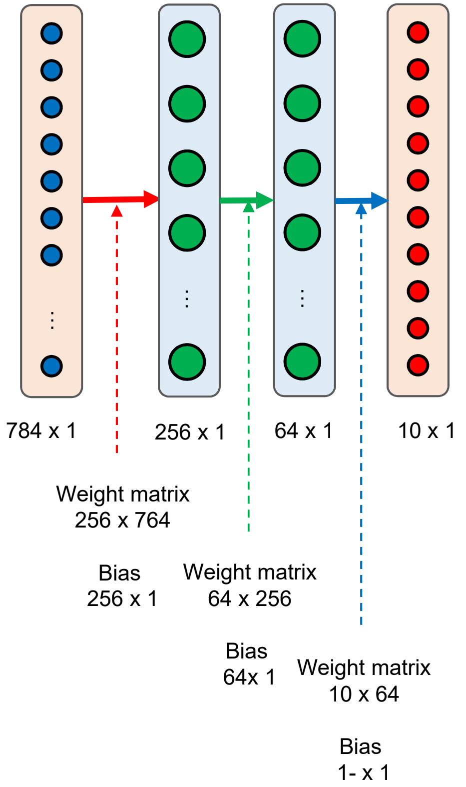
Neural Network Design: Bias-Variance Trade-off





We want our network to have just right complexity!

Neural Network Design: Bias-Variance Trade-off

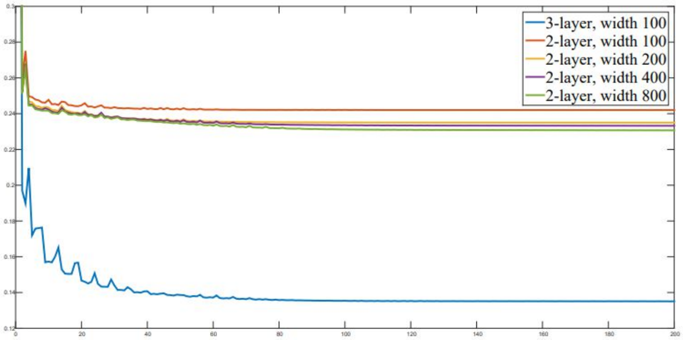


- This network has **3 layers**
- **Widths** are **256, 64**
- **Number of parameters**:
  - Weight matrix
    - 256 x 784 = 200,704
    - 64 x 256 = 16,384
    - 10 x 64 = 640
  - Bias
    - 256, 64, 10
  - Total:
    200,704+16,384+640+256+64+10 = **218,058**

Ok, this seems to tell us something about the complexity …

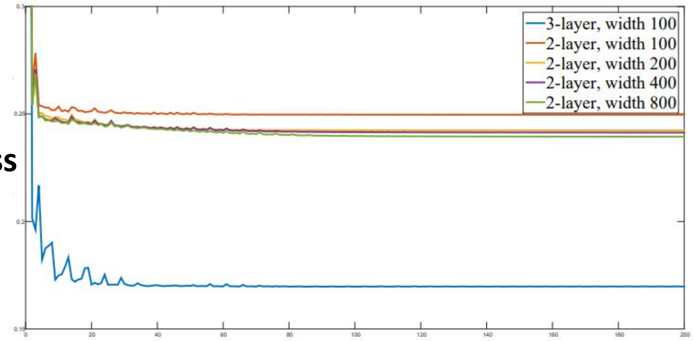
Neural Network Design: depth vs. width

Training loss



iterations

Validation loss

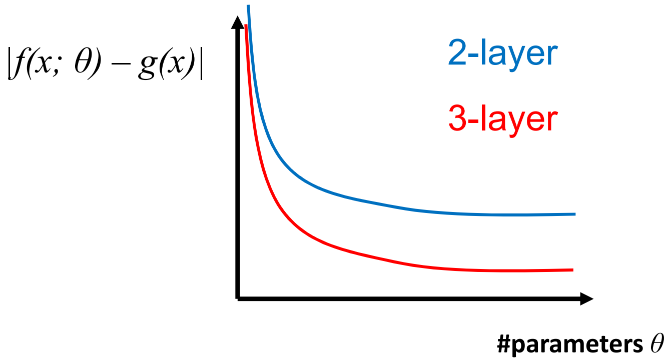


iterations

- **Increasing depth** (number of layers) usually results in "faster" capacity increase than increasing width
- A network with a **smaller number of parameters** tends to be less susceptible to overfitting

I. Safran and O. Shamir, Depth-Width Tradeoffs in Approximating Natural Functions with Neural Networks, ICML, 2017

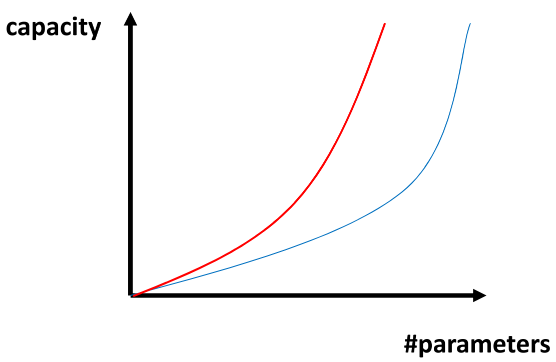
Neural Network Design: depth vs. width

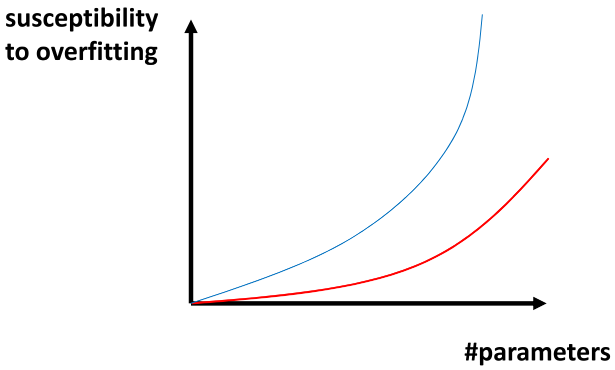


- We want to use different form of parameterized function $f(x;\,\theta)$ depending on our target function $g(x)$ (which we do not know)
- Some families of parameterized functions $f(x;\,\theta)$ can approximate $g(x)$ with a smaller error with smaller number of parameters than others

- …and this of course depends on $g(x)$ (task-dependent)

Neural Network Design: depth vs. width

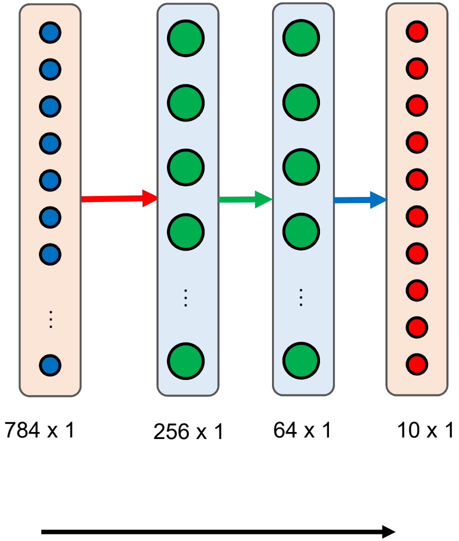




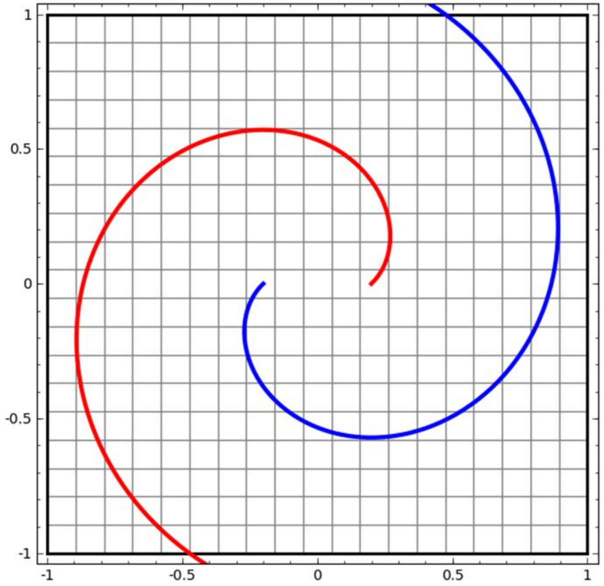
These are just conceptual plots!!!
Take-home message:
The capacity (capability of approximate complex functions) and susceptibility to overfitting of different parameterized functions (e.g., neural network) scale differently (and this is task-dependent)

We are essentially trying to approximate a function. If we can reduce the degree of freedom of our parameterized function $f(x;\theta)$ without compromising its capability to approximate the "target function," then it would be desired

Neural Network Design: avoid abrupt decrease



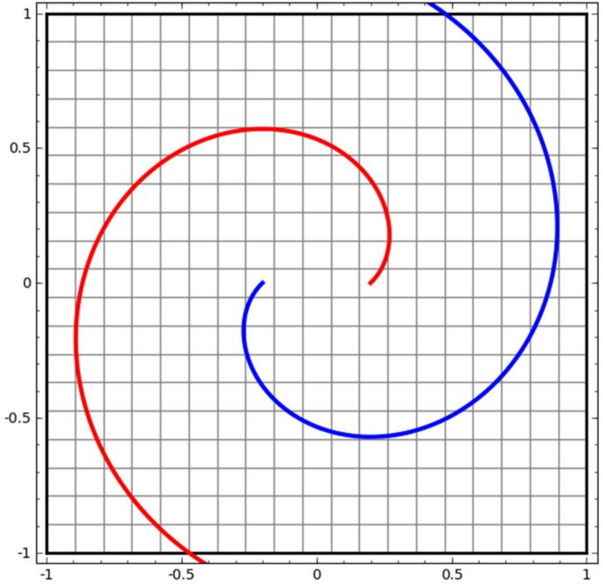
Size is gradually decreasing. Why?



Input data is being gradually “transformed”
via multiple rounds of matrix multiplication & activation,
so that it becomes **linearly separable** at the end

(why does it have to become linearly separable?)

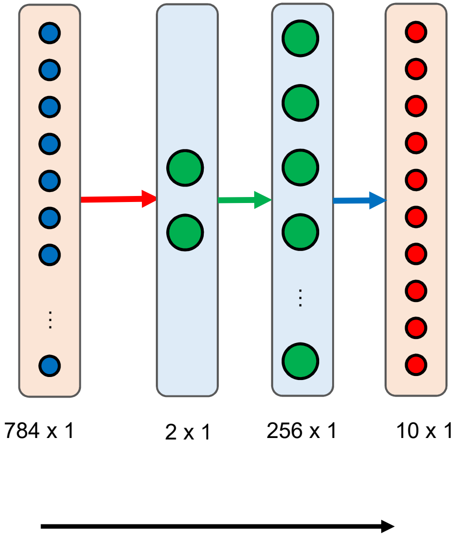
Neural Network Design: avoid abrupt decrease



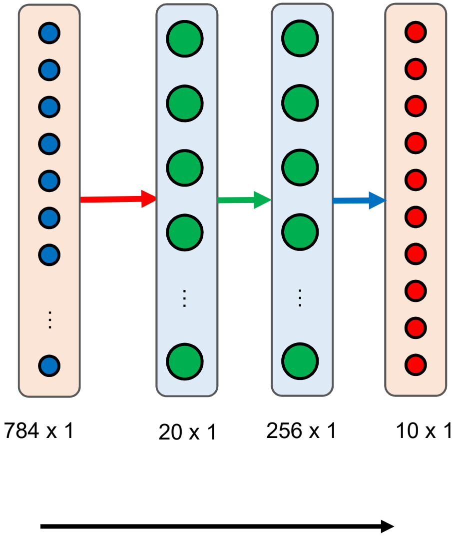
Input data is being gradually "transformed"
via multiple rounds of matrix multiplication & activation,
so that it becomes **linearly separable** at the end

- Neural network is not a magic
- Single layer can only do "*h(WX)*-much" of operation
- If we are asking a neural network to approximate a complicated function, we have to "give" enough depth to do that

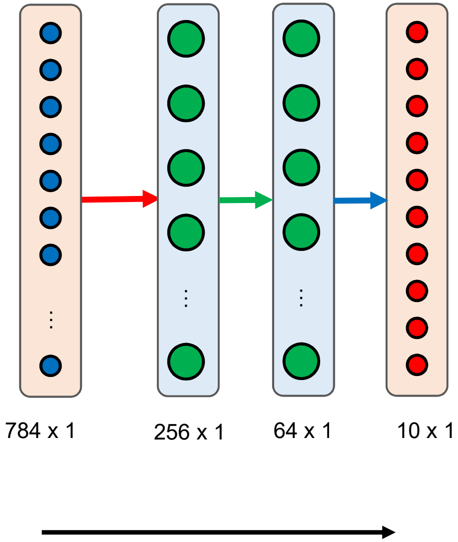
Neural Network Design: avoid abrupt decrease



What is wrong with this design?

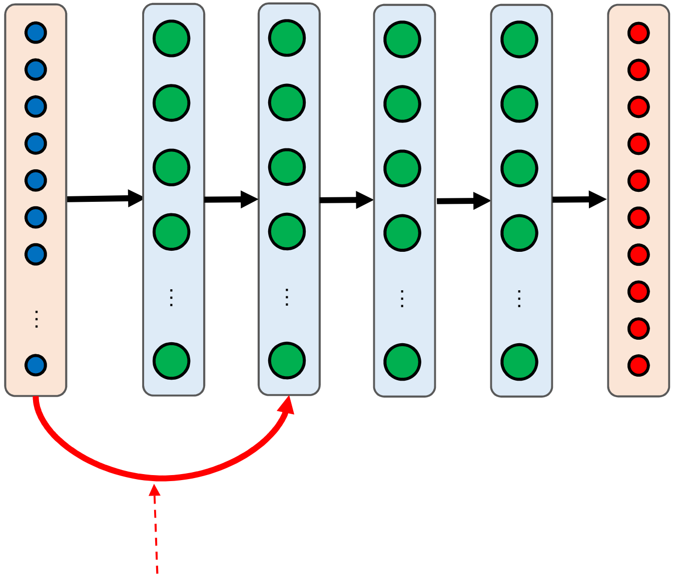


Is this OK?



Is this better?

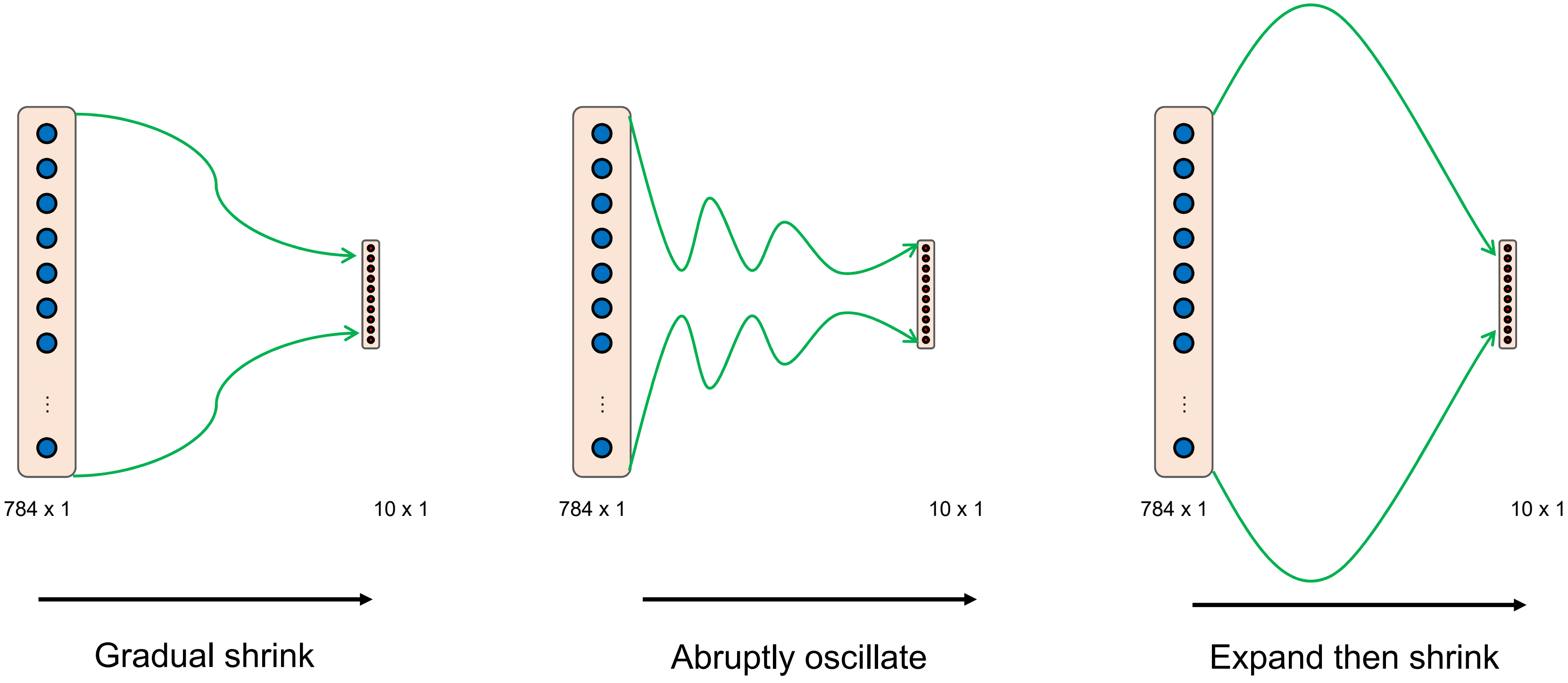
Neural Network Design: avoid abrupt decrease



- Mapping a high-dimensional vector to a low-dimensional vector always comes with information loss, and the question is, whether the information that is relevant for estimating the output has been lost or not.
- Reducing dimension, without loss of (relevant) information, requires that the network is capable of “extracting” important information

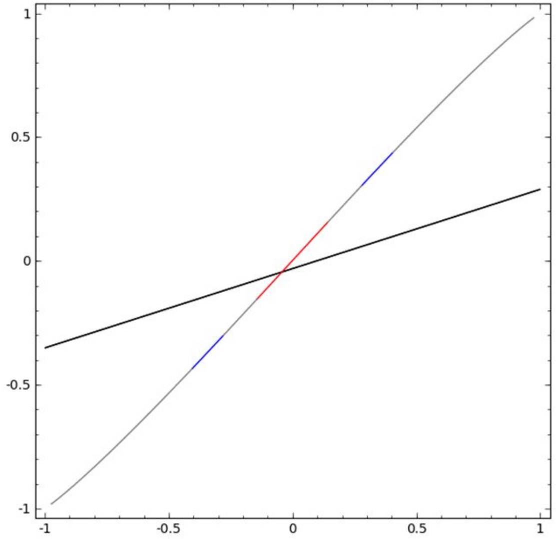
From here to there is just a two-layer neural network
which means it is unlike that this two-layer network is capable of
efficiently mapping the high-dimensional input to a low-dimensional form
(efficiently: with a very small amount of information loss)

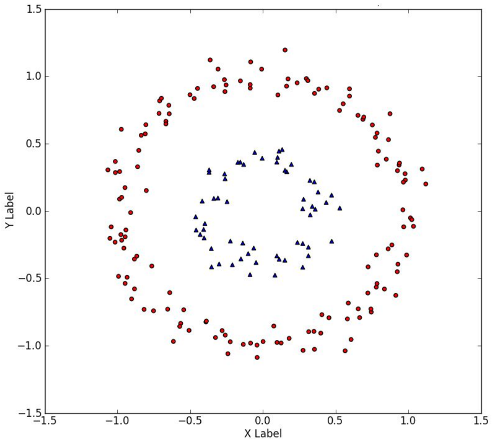
Neural Network Design: avoid abrupt decrease

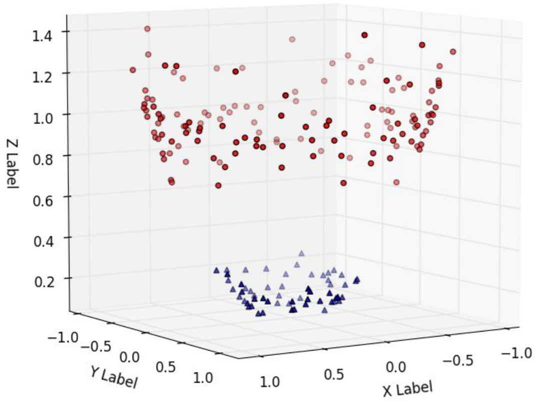


How do we ever want to increase width?

- If the output has lower dimension than the input, why do we ever want to increase width?
  - Short answer: representing low dimensional data in a high dimension can be useful to implement certain operations (or functions)



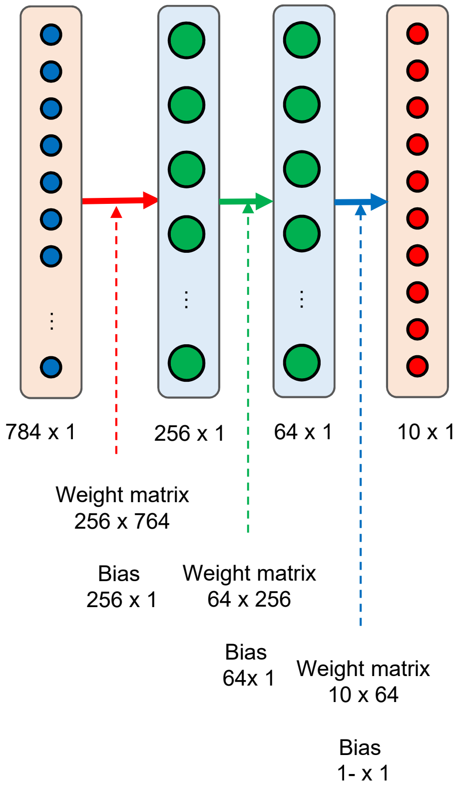




Neural Network Design: flow

- However, these design approaches still do not provide what actual depth and width (and other hyper-parameters) we should use
- **Possible Flow**
  - Build the first version then train (start with a small network)
  - Check training loss and validation loss
  - Increase the capacity until training loss sufficiently goes down
  - Stop increasing (or decrease) the capacity if the difference between training loss and validation loss becomes large
- The flow above is just one possible way (there are many other possible ways and many other things to consider)

Neural Network Design: PyTorch Implementation




```python
class mlp_classifier(nn.Module):
    def __init__(self):
        super(mlp_classifier, self).__init__()
        self.layer1 = nn.Linear(28*28, 256)
        self.layer2 = nn.Linear(256, 64)
        self.layer3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        x = self.relu(x)
        x = self.layer3(x)
        x = self.relu(x)
        return x
```

Summary

- Deep learning library makes it very easy to implement complicated neural networks
- …but we still have to understand how they work inside to fully utilize the power of NN
- We have to formulate the given tasks in a form that NN can handle
- NN design involves lots of heuristics (trial and error), but there still are a few things to remember

References

- Website
  - CS231n course website: https://cs231n.github.io/
- Pytorch tutorial
  - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- Tensorflow guide
  - https://www.tensorflow.org/guide/
- Matplotlib Guide
  - https://matplotlib.org/
- Neural Networks, Manifolds, and Topology
  - https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/