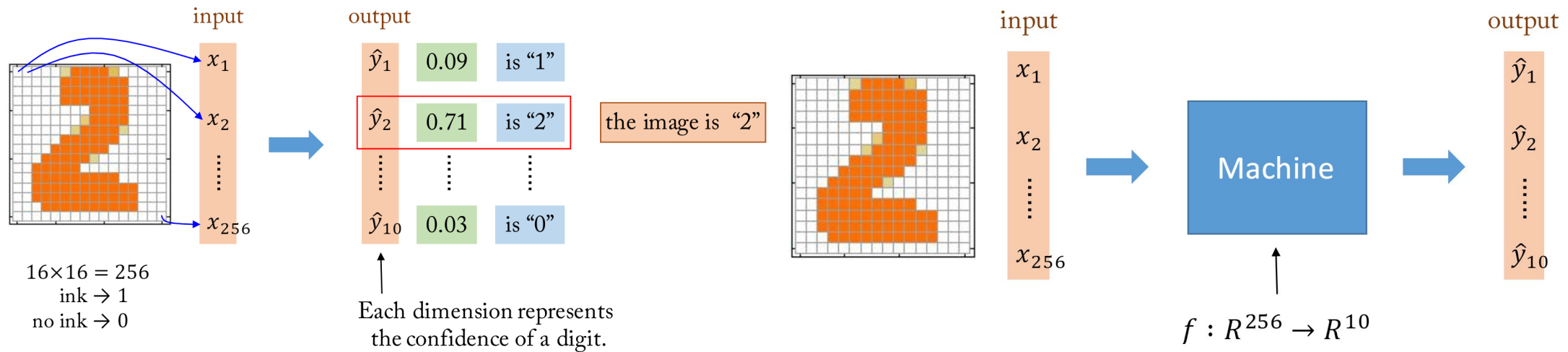


Classifying MNIST digits using MLP

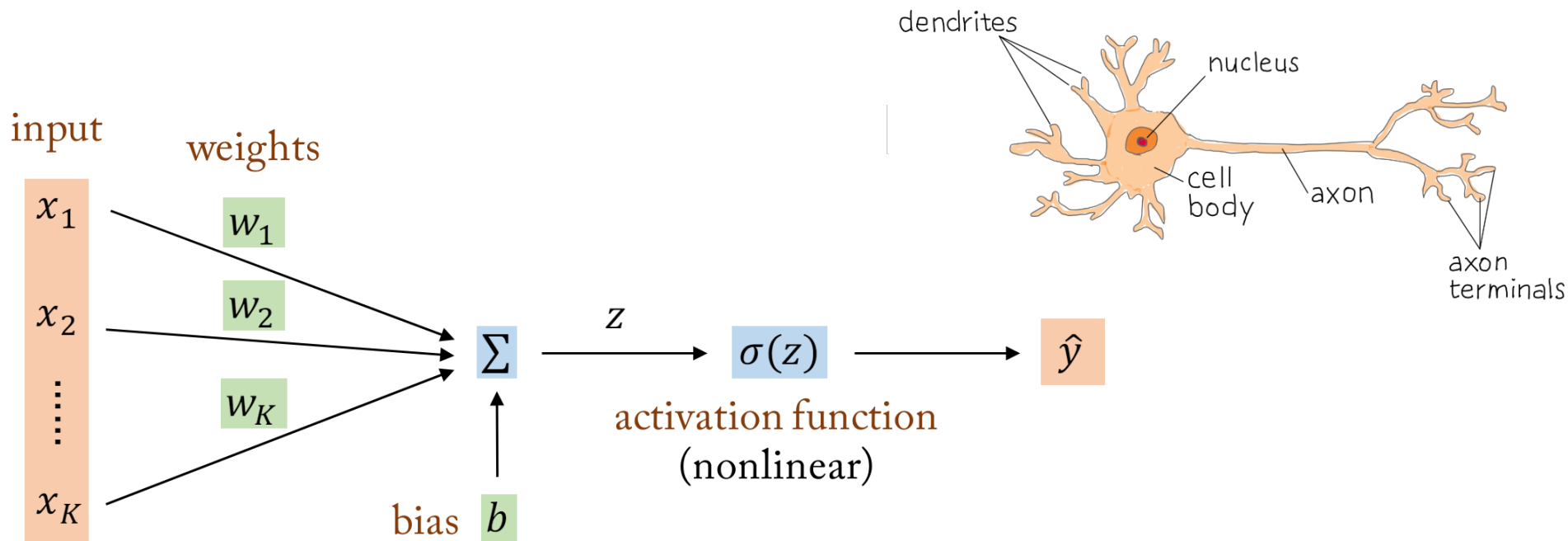
TA. Jaewoong Han

1. Deep Learning motivation



- In Deep Learning, the function f is represented by a Neural Network.

2. Perceptron



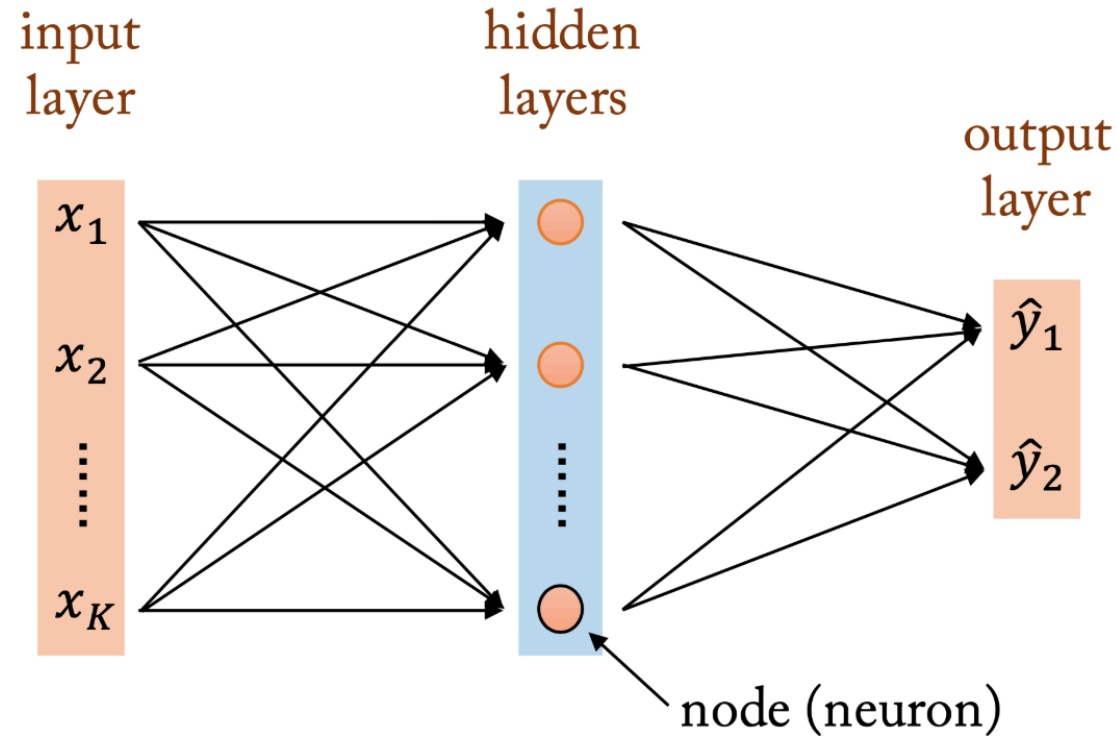
$$f : \mathbb{R}^K \rightarrow \mathbb{R}$$

$$\hat{y} = \sigma(z) = \sigma(w_1x_1 + w_2x_2 + \dots + w_Kx_K + b)$$

nonlinear

weighted sum (linear)

3. Multilayer Perceptron (MLP)



- MLP is a class of feedforward artificial neural network.
- MLP consists of an input layer, several hidden layers and an output layer.
- Except input nodes, each node is a neuron (perceptron) that uses a nonlinear activation function.

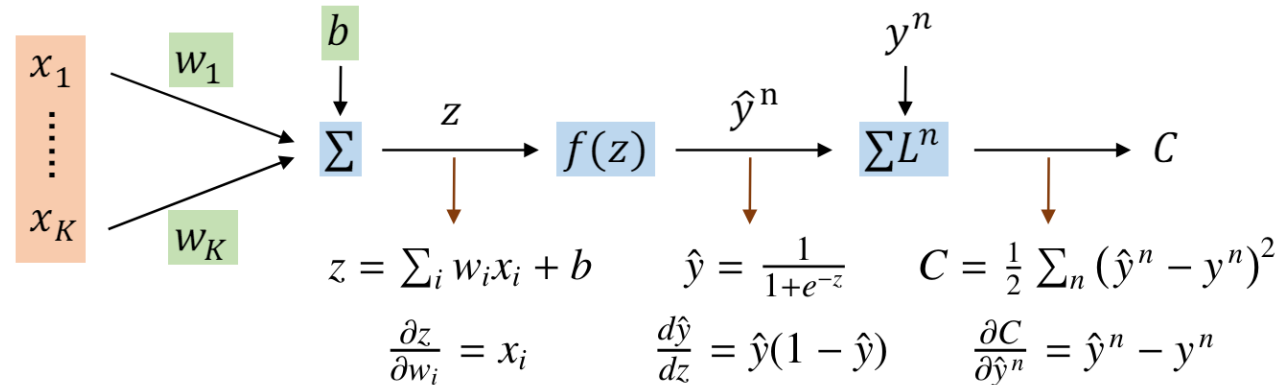
4. Backpropagation

- Backpropagation is the way of computing the gradient efficiently to update millions of network parameters
- Gradient Descent
 - Goal: find the optimal parameter w^* , which minimizes the total cost (MSE)

$$C = \frac{1}{2} \sum_{n=1}^N (\hat{y}^n - y^n)^2 \text{ where } \hat{y}^n = f(w^T x^n + b)$$

- Backpropagation: iteratively updating the model parameters w to decrease C by using

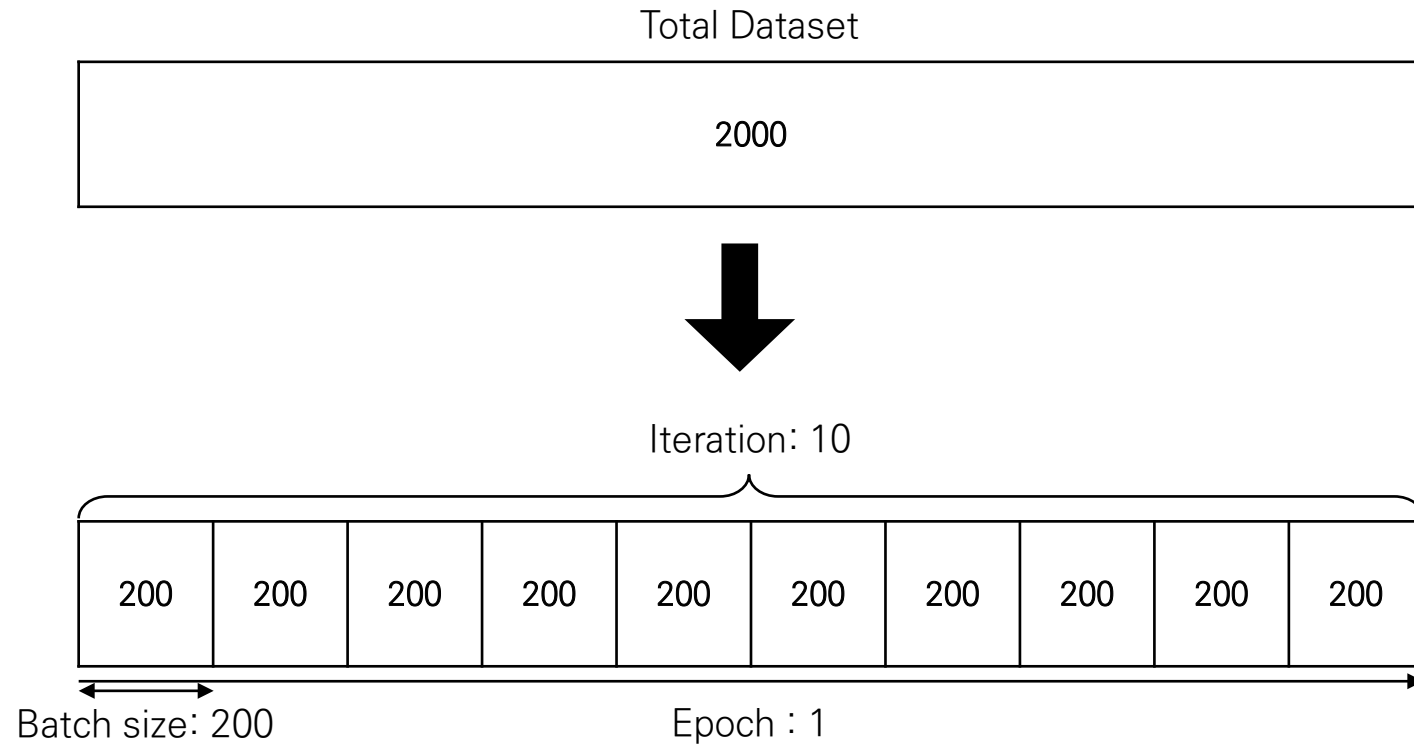
$$w_i(t+1) = w_i(t) - \eta \frac{\partial C}{\partial w_i}$$



$$\frac{\partial C}{\partial w_i} = \sum_{n=1}^N \frac{\partial z^n}{\partial w_i} \frac{d\hat{y}^n}{dz^n} \frac{\partial C}{\partial \hat{y}^n} = \sum_{n=1}^N x_i^n \hat{y}^n (1 - \hat{y}^n) (\hat{y}^n - y^n)$$

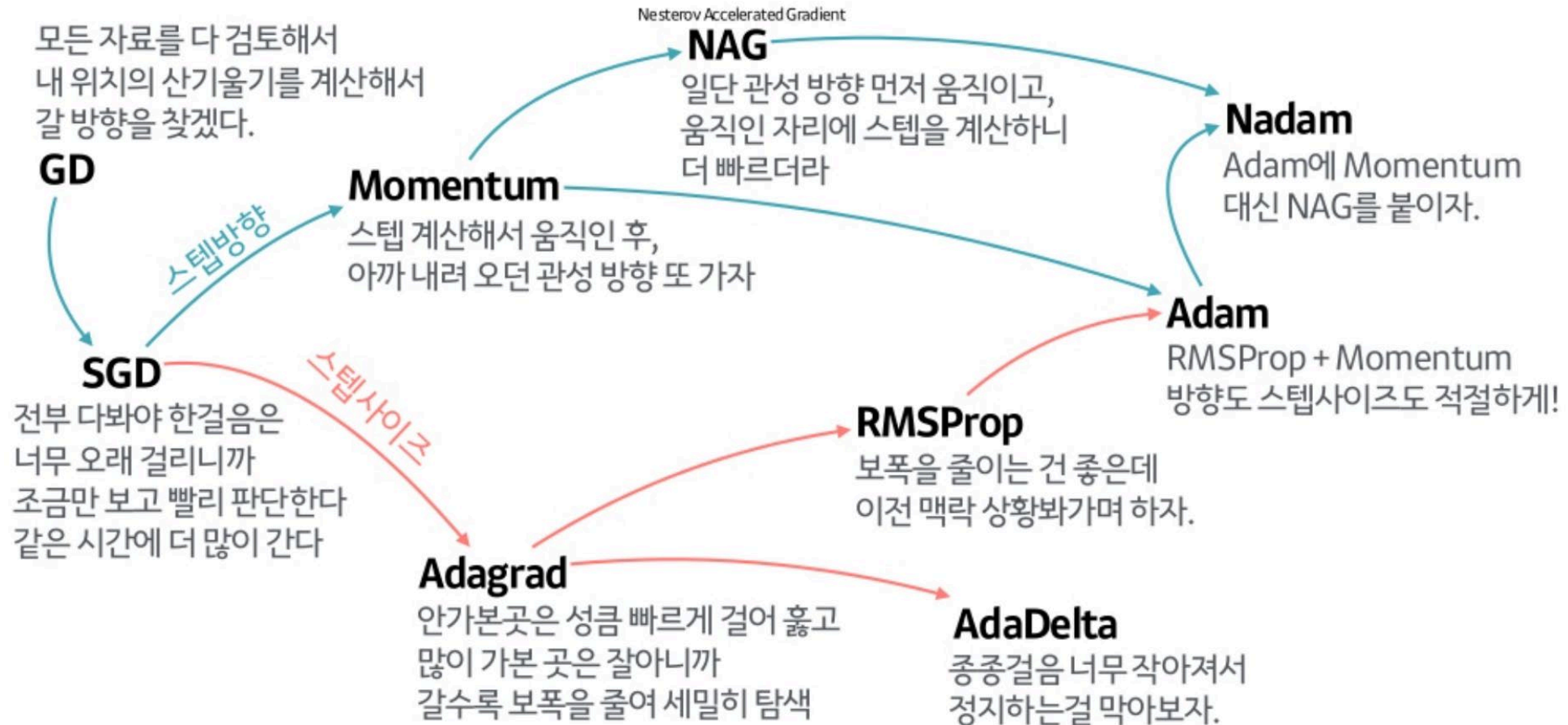
Chain rule

5. Batch, Minibatch, Epoch

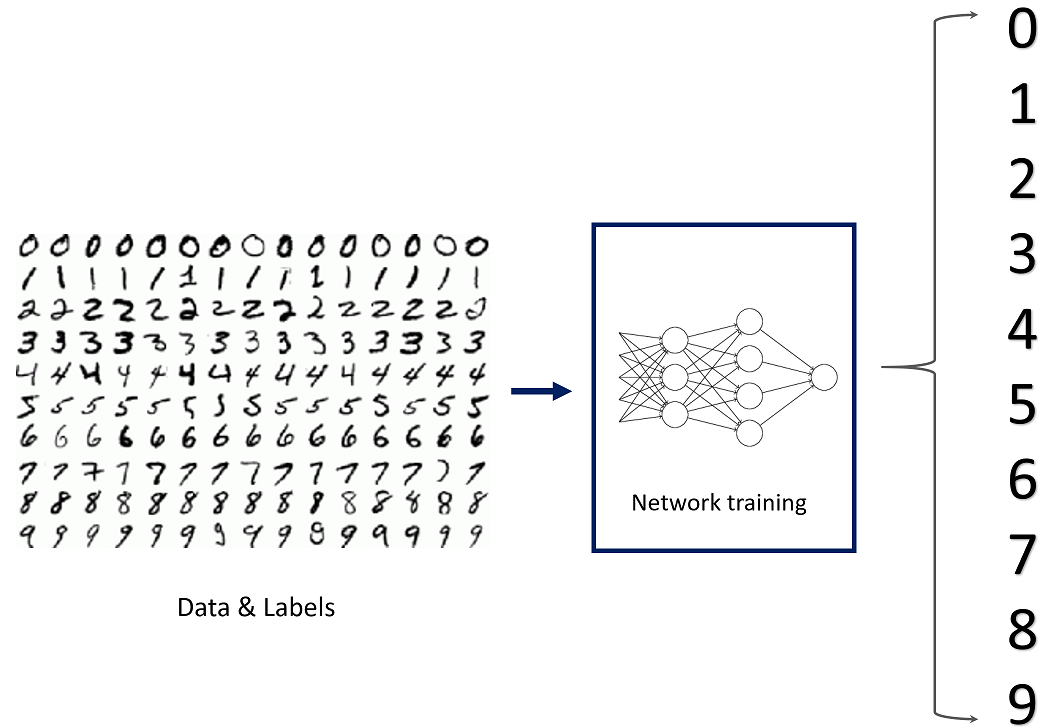


- Batch gradient Descent: Use the entire dataset per update
- Mini-batch Gradient Descent: Use a mini-batch of dataset per update
- Stochastic Gradient Descent: Use a single data sample per update

6. After Gradient Descent Algorithm ...



1. MNIST



- Modified National Institute of Standards and Technology database
 - 60,000 images for the training and 10,000 images for the test.
 - Size : 28x28 pixels
 - Only 1 channel (Gray scale)

2. Preparing Data

1. Load

- Use easy loading function in torchvision library
- Need to create custom loading function when trying to load your own dataset.

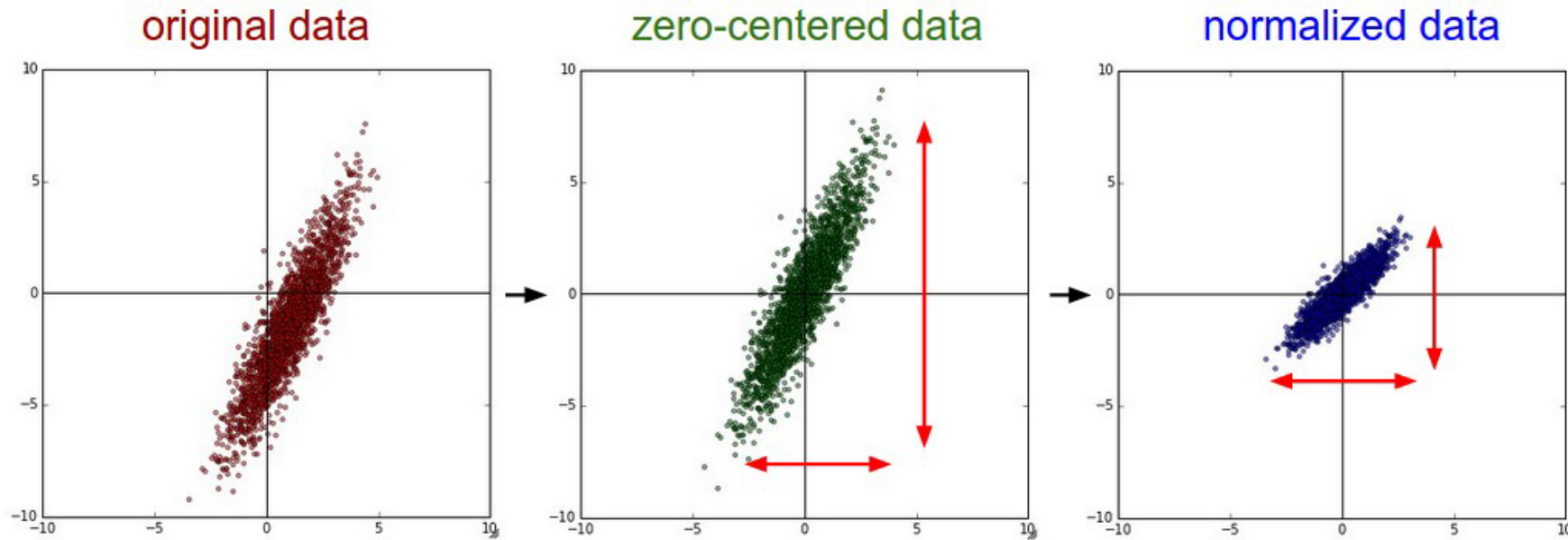
2. Normalize

- Standardize features by removing the mean and scaling to unit variance
- Divide by the maximum value

3. Convert to one-hot encoding/vector

- Need when training a neural network which classifies given data

3. Normalize

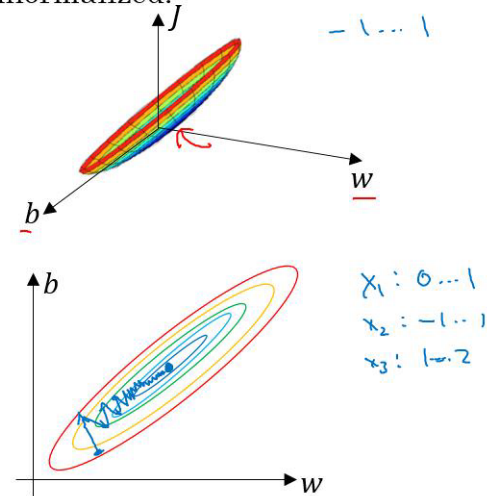


- Common data preprocessing pipeline.
 - Left: Original toy, 2-dimensional input data.
 - Middle: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin.
 - Right: Each dimension is additionally **scaled by its standard deviation**.
- The red lines indicate the extent of the data – they are of unequal length in the middle, but of equal length on the right.

3. Normalize

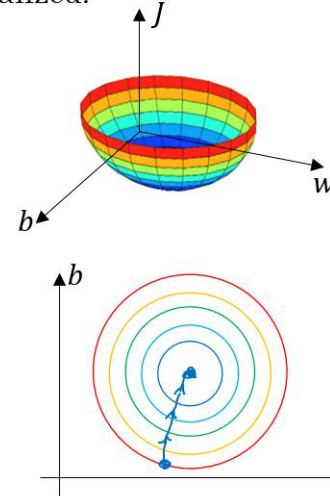
Why normalize inputs?

Unnormalized:
 $w_1, x_1: 1 \dots 1000 \leftarrow$
 $w_2, x_2: 0 \dots 1 \leftarrow$
 $-1 \dots 1$



Normalized:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$



$$\nabla J(w) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2} \right]$$

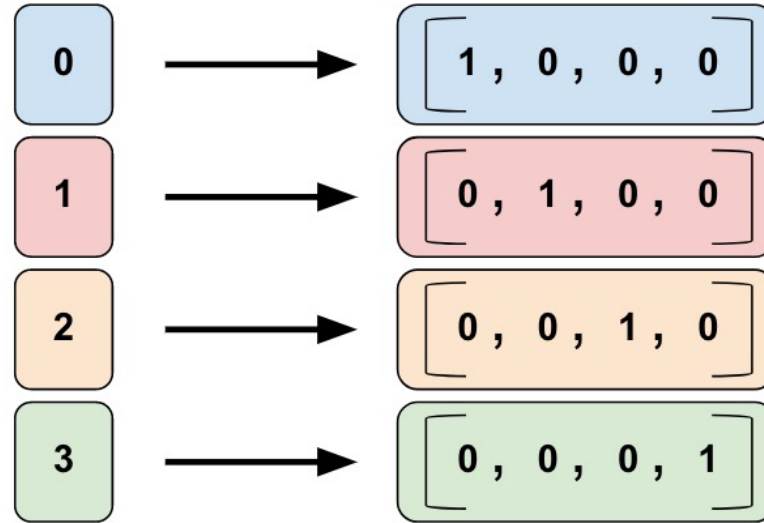
Andrew Ng

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Contents

- Introduction to Deep Learning
 - Deep Learning Motivation
 - Perceptron
 - Multi Layer Perceptron (MLP)
 - Backpropagation
 - Batch, Minibatch, Epoch
 - After gradient Descent Algorithm ...
- MLP with Deep Learning
 - MNIST
 - Preparing Data
 - Normalize
 - Convert to one-hot encoding/vector
 - Creating a Model
 - Dense Layer
 - Activation Function
 - Dropout Layer
 - Softmax
 - Training a network
 - Test a network

4. Convert to one-hot encoding/vector



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512) # (input vector, output)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784) # (batch size, 28, 28) -> (batch size, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)
```

5. Creating a model

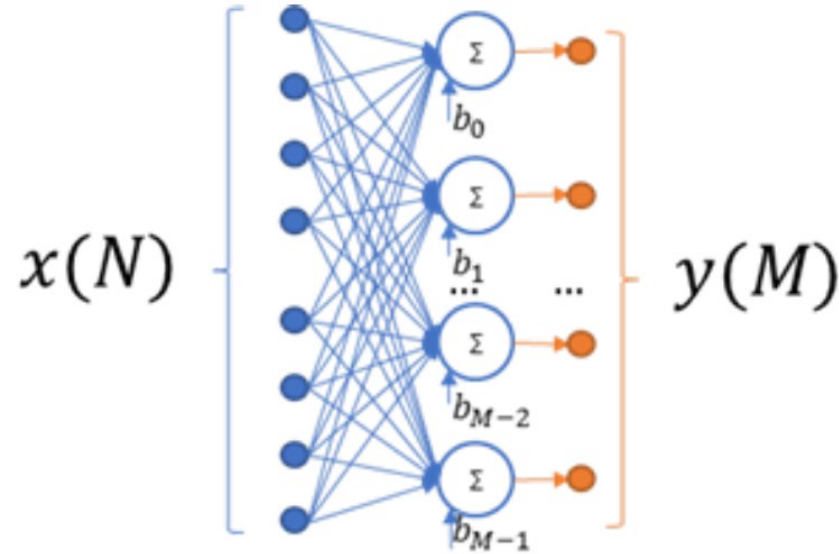
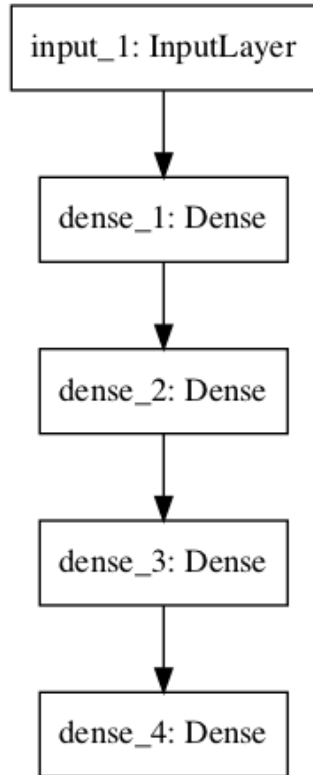
- Define custom PyTorch class (torch)
 - Easy way of creating a network.
 - Convenient, but hard to use when creating complex networks.
- Dense Layer
 - Fully connected layer.
 - Consists of weights and biases
- Activation Functions
 - Defines the output of the node
 - ReLU, tanh, sigmoid
- Dropout Layer
 - Prevents overfitting
 - Randomly removes the connections between weights
- Softmax
 - Converts the scores to the probabilities
 - Usually locates at the end of the network

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)

model = Net()
```


6. Dense Layer



```

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)

model = Net()
  
```

$$y_i = x_j W_{i,j} + b_i$$

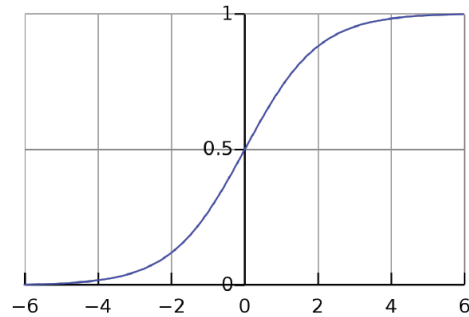
Where:

- x_j - j_{th} value in input tensor.
- y_i - i_{th} value in output tensor.
- W_{ij} - weight of j_{th} input element for i_{th} neuron.
- b_j - bias for i_{th} neuron.

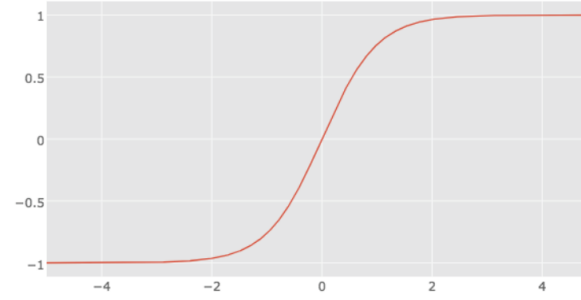
7. Activation Functions

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

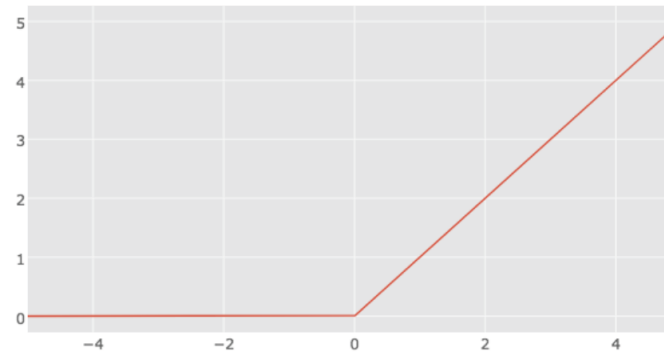
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



Sigmoid



Tanh



ReLU (Rectified Linear Unit)

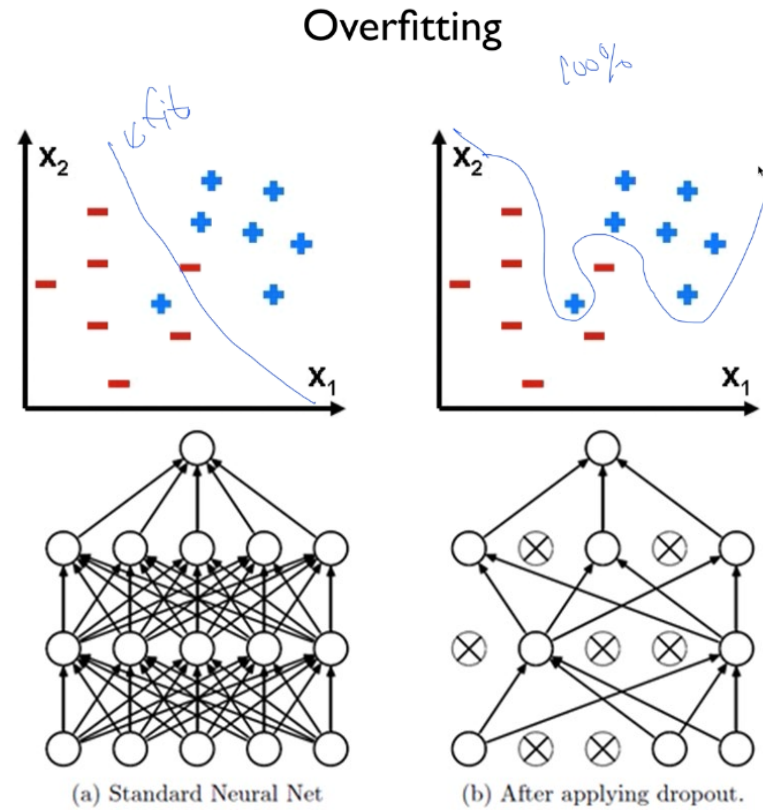
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)

model = Net()
```

- Defines the output strength of the signal to the following layer
- Select proper and better activation functions by trial and errors

8. Dropout Layer



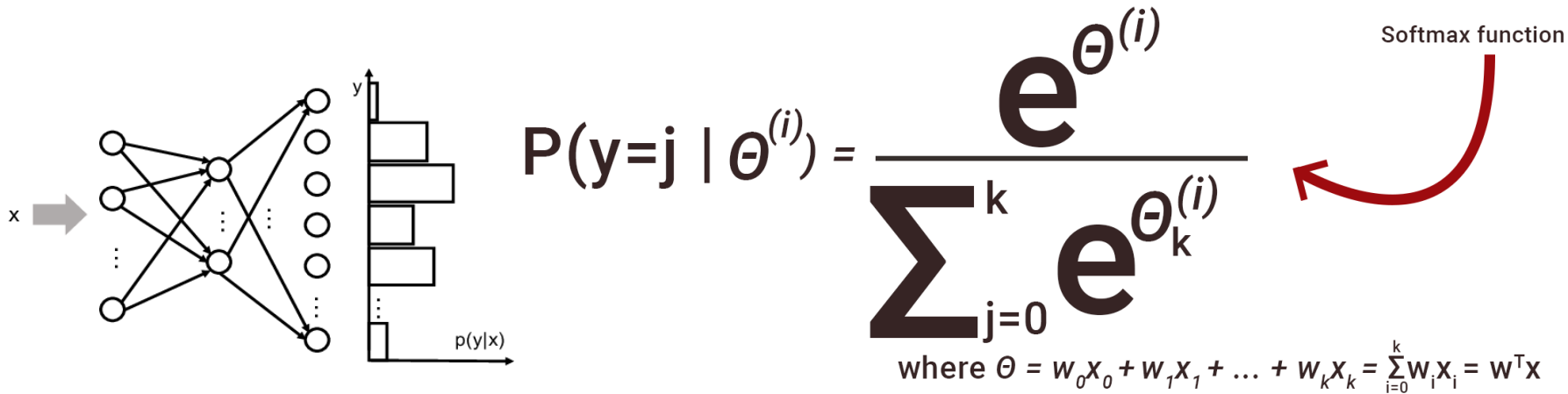
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)

model = Net()
```

- Prevents overfitting when the network has many hidden layers
- Skips / removes the connections between weights

9. Softmax



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)

model = Net()
```

- Softmax layer is used as the output layer
- Converts the scores to the probabilities
- We select the index of the largest probability as a classification result

10. Training a network

```
criterion = nn.CrossEntropyLoss()           Choose an appropriate loss function for current task (loss function ex: MSE, ...)  
optimizer = optim.RMSprop(model.parameters()) Sets the optimizer (ex: GD, ...)  
  
for epoch in range(epochs):  
    for i, (images, labels) in enumerate(train_loader):  
        outputs = model(images) # images.shape == [128, 1, 28, 28]  
        loss = criterion(outputs, labels)  
  
        optimizer.zero_grad() Clear the gradients of all optimized parameters  
        loss.backward()         Computes the gradient of the loss with respect to the model parameters  
        optimizer.step()        Updates the model parameters based on the computed gradients  
  
        if (i+1) % 100 == 0:  
            print(f'Epoch [{epoch+1}/{epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')
```

11. Test a network

```
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    # To store a few test images for visualization
    examples = []
    num_examples = 16

    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

Question
