Advanced Deep Learning

Lec 2: Multilayer Perceptrons

STAT4744/5744

Xin (Shayne) Xing • Virginia Tech

Outlines

1. Intro to Pytorch with a simple linear regression model
2. Limitation of linear model
3. Multilayer Perceptrons

PyTorch

- PyTorch is an open-source deep learning framework that's known for its flexibility and ease-of-use. This is enabled in part by its compatibility with the popular Python high-level programming language favored by machine learning developers and data scientists.

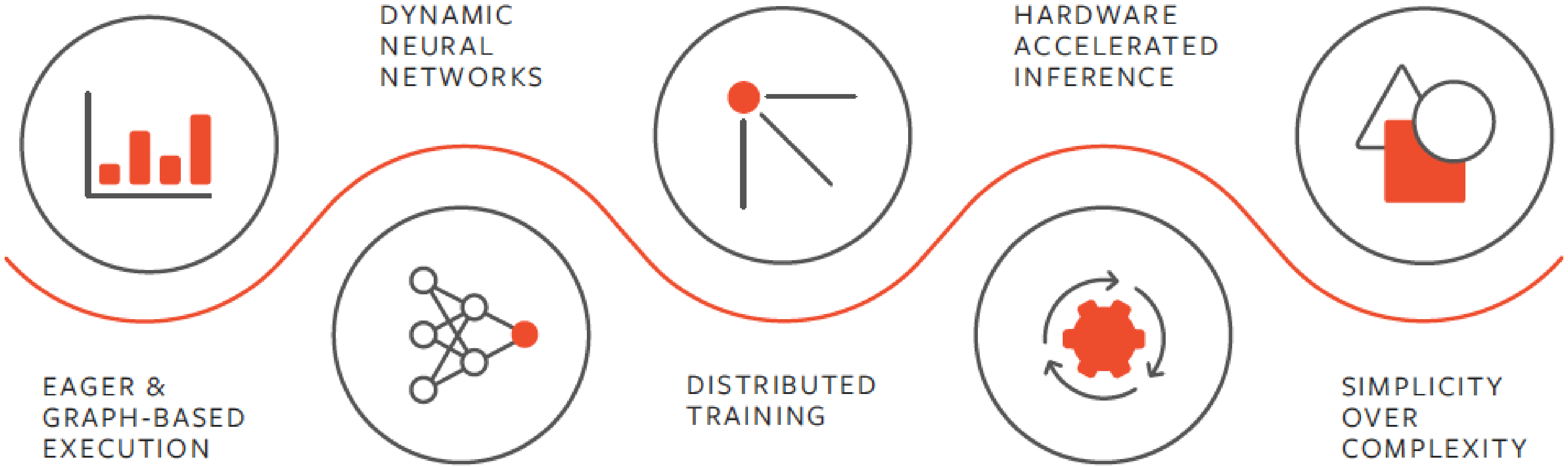


Image reference https://pytorch.org/features/

Pytorch

- A torch.Tensor is a multi-dimensional array containing elements of a single data type.

```
a = torch.tensor(1)
b = torch.tensor([1,1])
c = torch.zeros((3,3))
d = torch.randn(3,3,3)
```

Device and CUDA

- Use cuda.is_available() to find out if you have a GPU at your disposal and set your device accordingly.

```
import torch
import torch.optim as optim
import torch.nn as nn


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

```
'cuda'
```

- Specify the device for PyTorch tensor using .to() method.

```
a.to(device)
```

```
tensor(1, device='cuda:0')
```

Autograd (with simple linear regression example)

- Considering the following simple linear regression model:

$$y = a + bx + \epsilon$$

- We have N i.i.d. observation (y_i, x_i), the least square loss function is

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - a - bx_i)^2$$

A gradient is a partial derivative — why partial? Because one computes it with respect to (w.r.t.) a single parameter.

Gradient Decent

- Compute the gradients

$$\frac{\partial MSE}{\partial a} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial a} = \frac{1}{N} \sum_{i=1}^{N} 2(y_i - a - bx_i) \cdot (-1) = -2\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)$$

$$\frac{\partial MSE}{\partial b} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} 2(y_i - a - bx_i) \cdot (-x_i) = -2\frac{1}{N} \sum_{i=1}^{N} x_i(y_i - \hat{y}_i)$$

- Update the parameters

$$a = a - \eta \frac{\partial MSE}{\partial a}$$

$$b = b - \eta \frac{\partial MSE}{\partial b}$$

Creating Parameters

- Create regular tensors and send them to the device

```
a = torch.randn(1, dtype=torch.float).to(device)
b = torch.randn(1, dtype=torch.float).to(device)
# and THEN set them as requiring gradients...
a.requires_grad_()
b.requires_grad_()
print(a, b)
```

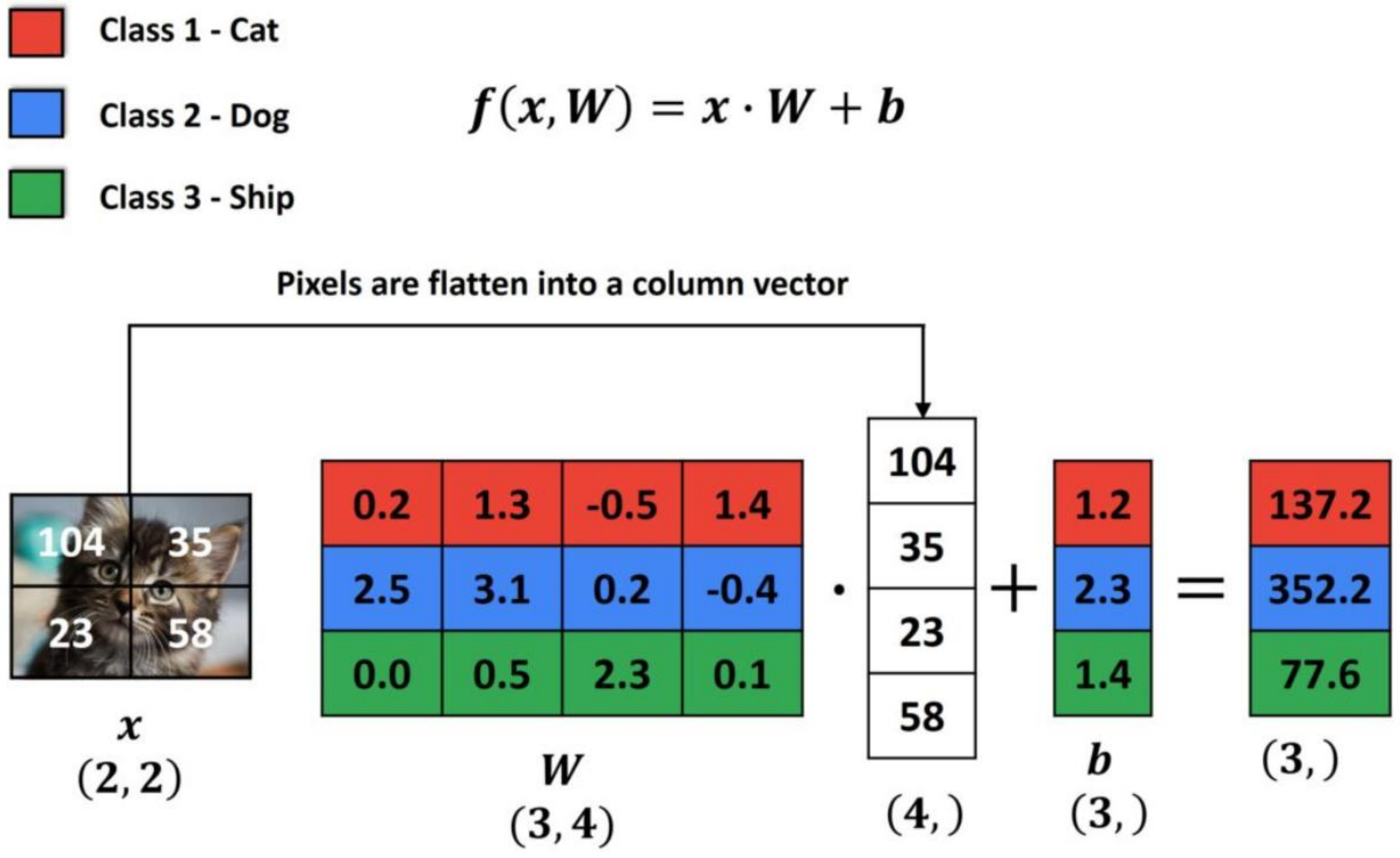
Autograd

- Autograd is PyTorch’s automatic differentiation package. Use backward() method for gradient calculation.

```
lr = 1e-1
n_epochs = 20
for epoch in range(n_epochs):
  yhat = a + b * x_train_tensor
  error = y_train_tensor - yhat
  loss = (error ** 2).mean()
  loss.backward()
  print(a.grad)
  print(b.grad)
  with torch.no_grad():
      a -= lr * a.grad
      b -= lr * b.grad

  a.grad.zero_()
  b.grad.zero_()
print(a, b)
```

Can you build a linear classifier?

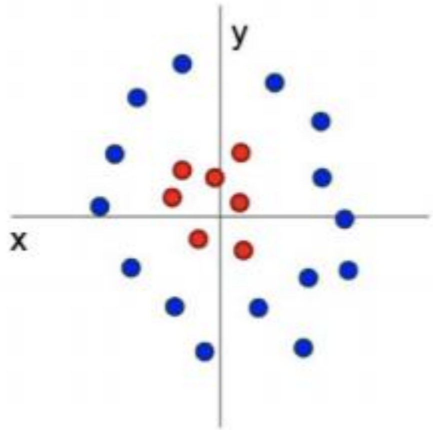


Limitation of Linear Models

Visual Viewpoint

Linear classifiers learn one template per class

Geometric Viewpoint



Linear classifiers can only draw linear decision boundaries

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

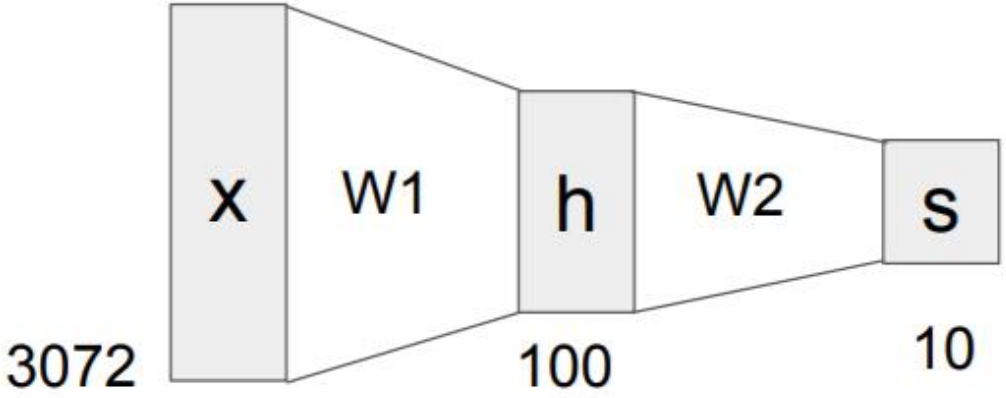
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

"Neural Network" is a very broad term; these are more accurately called "fully-connected networks" or sometimes "multi-layer perceptrons" (MLP)

Neural networks: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



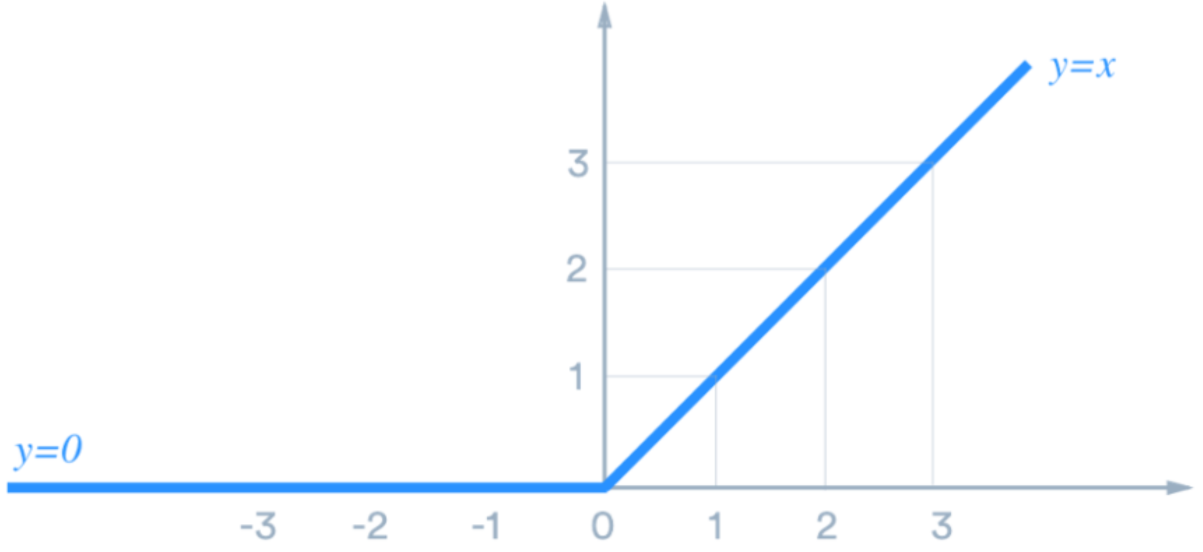
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Activation function

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \boxed{\max(0,} W_1 x)$

- The function is called the **Rectified Linear Unit (ReLU)** activation function.



Activation function

- The function is called the activation function.
- Q: What if we try to build a neural network without one?

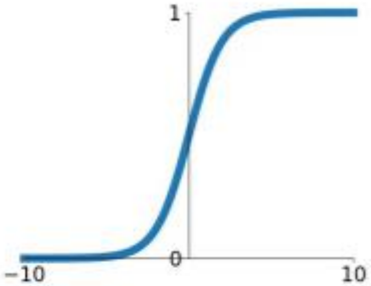
$$f = W_2 W_1 x \qquad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

- We end up with a linear classifier again!

Activation Functions

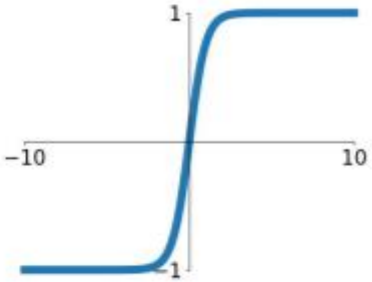
Sigmoid

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



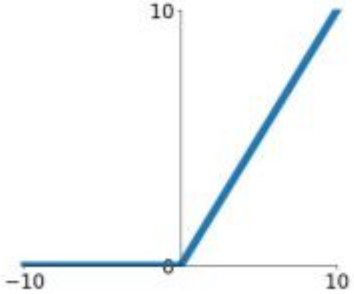
tanh

$\tanh(x)$



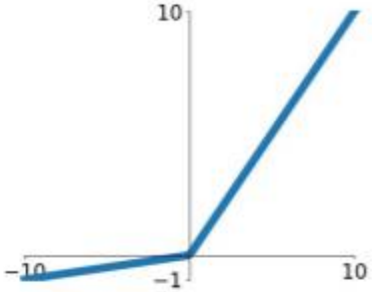
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

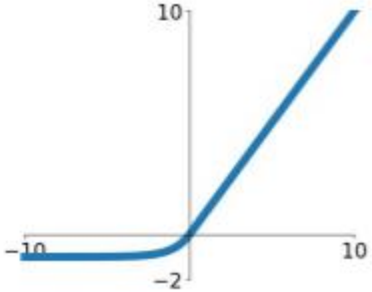


Maxout

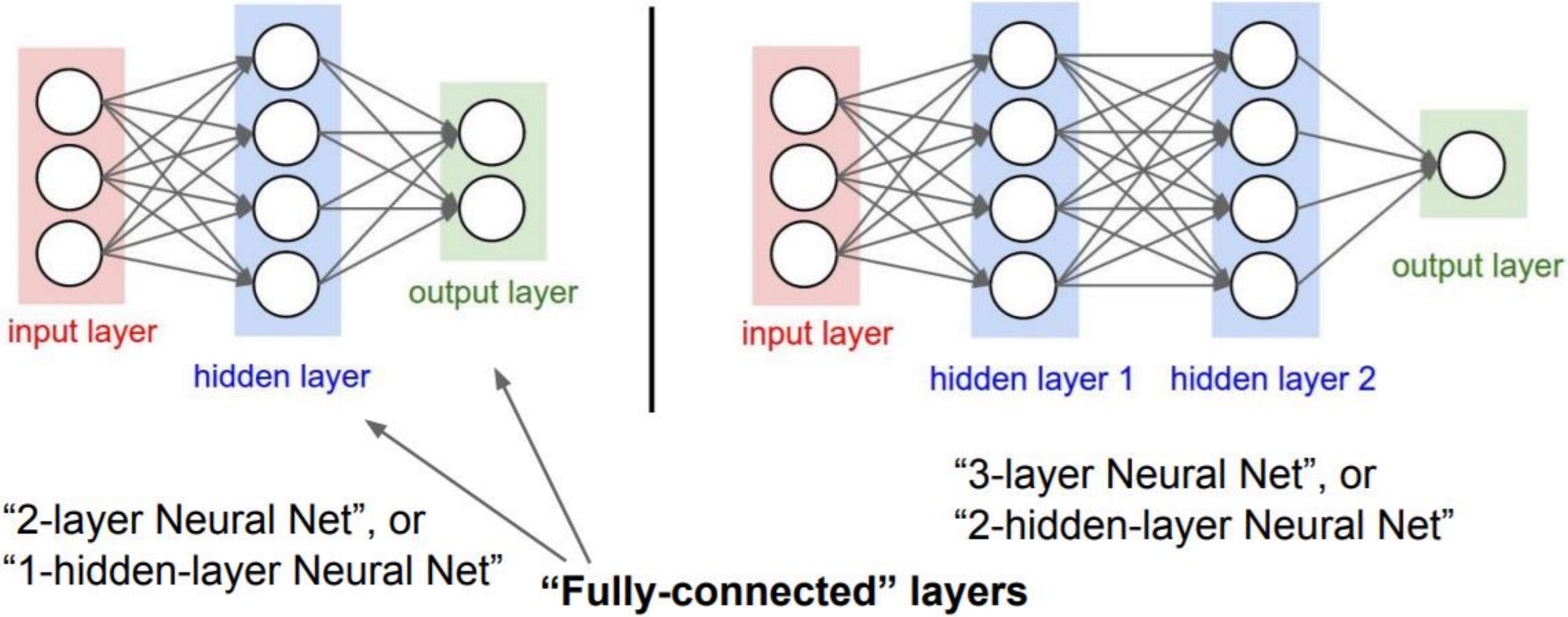
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

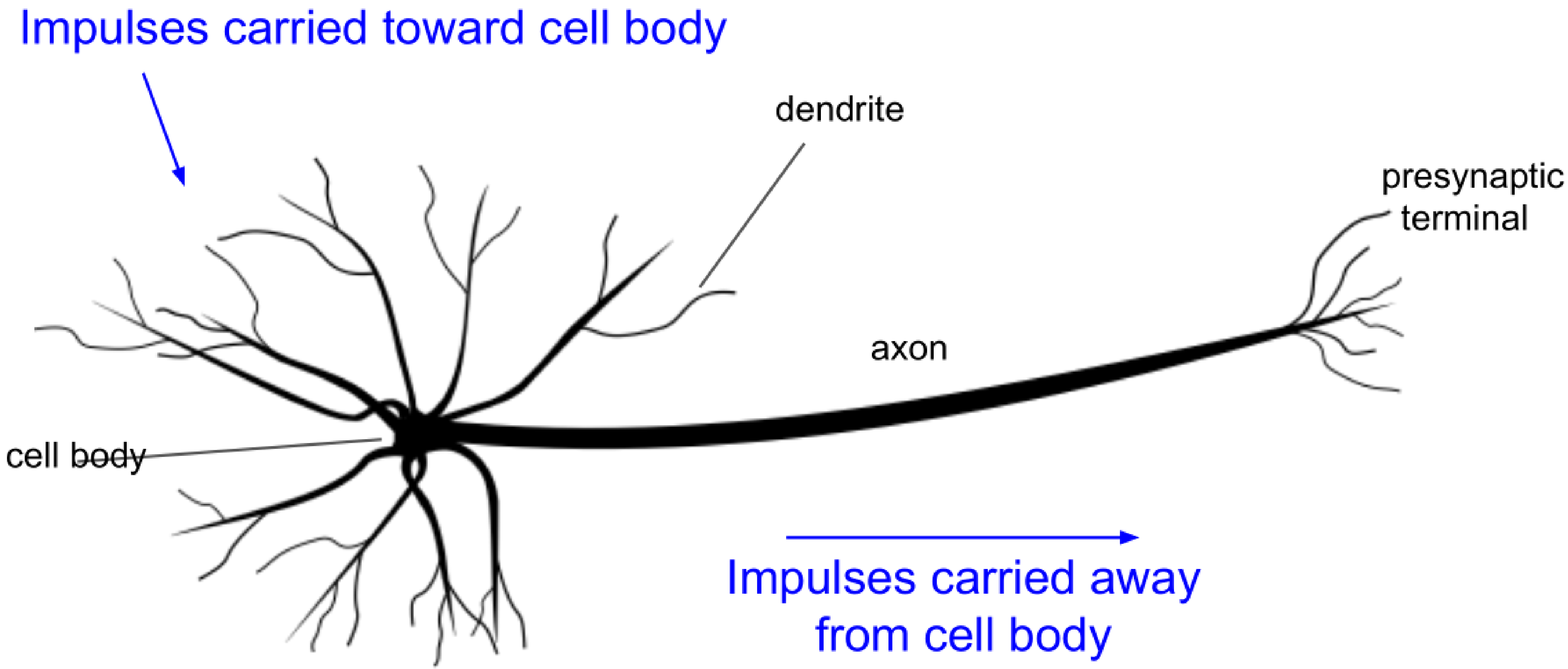
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Architectures

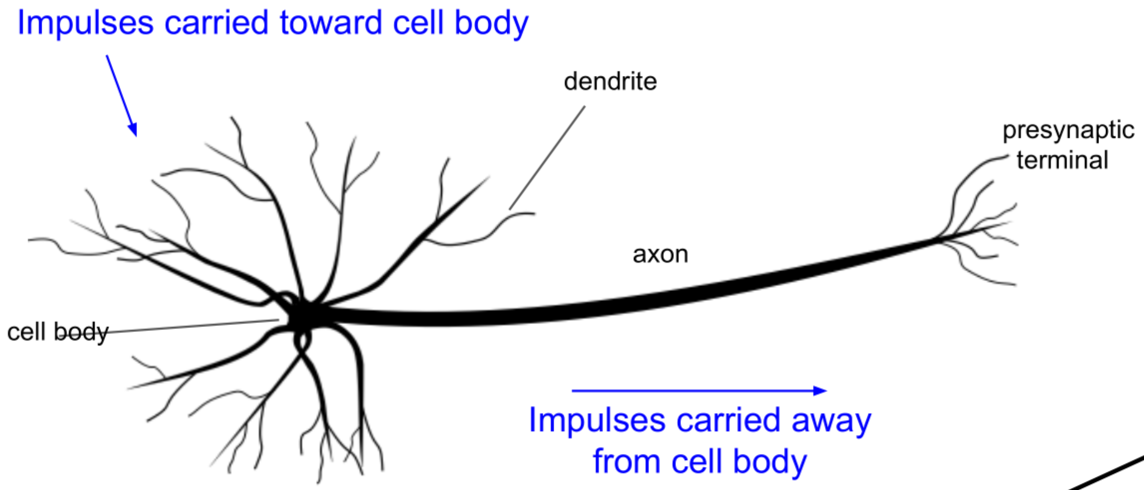


Biological Neuron

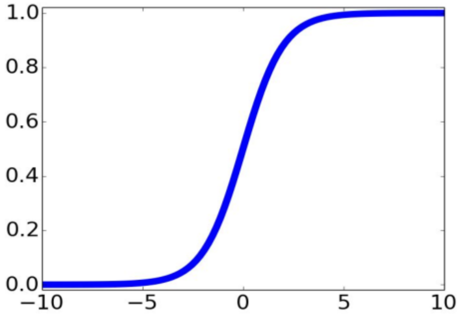


This image by Felipe Perucho
is licensed under CC-BY 3.0

Biological Neuron vs. Artificial Neuron

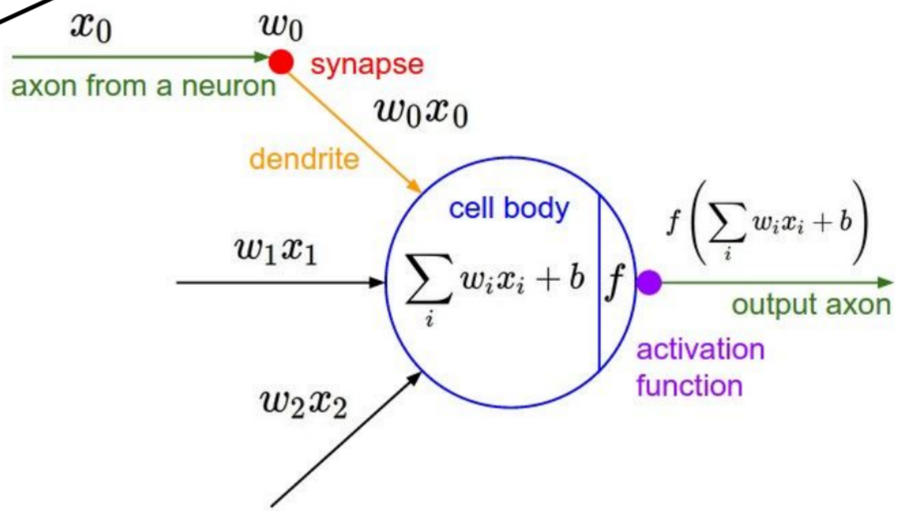


This image by Felipe Perucho is licensed under CC-BY 3.0



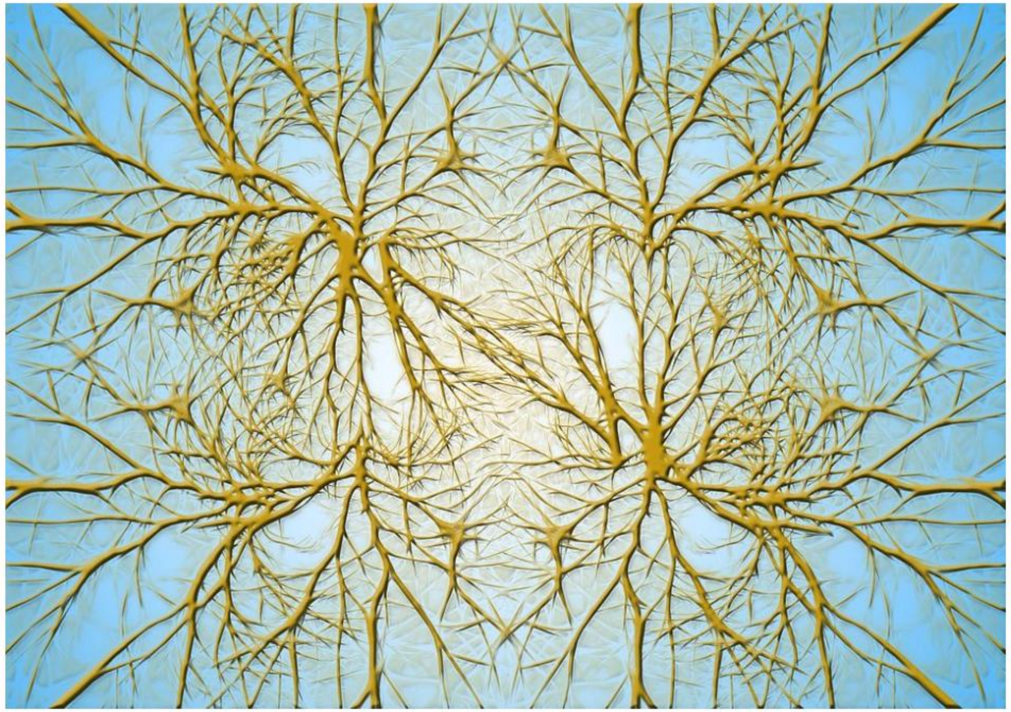
sigmoid activation function

$$\frac{1}{1+e^{-x}}$$



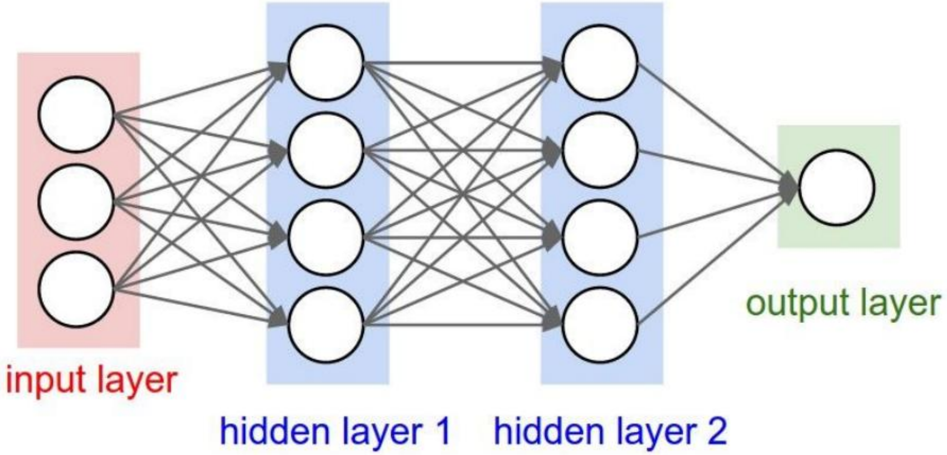
Biological Connectivity

Biological Neurons:
Complex connectivity patterns



This image is CC0 Public Domain

Neurons in a neural network:
Organized into regular layers for computational efficiency



Be very careful with your brain analogies!

Biological Neurons:

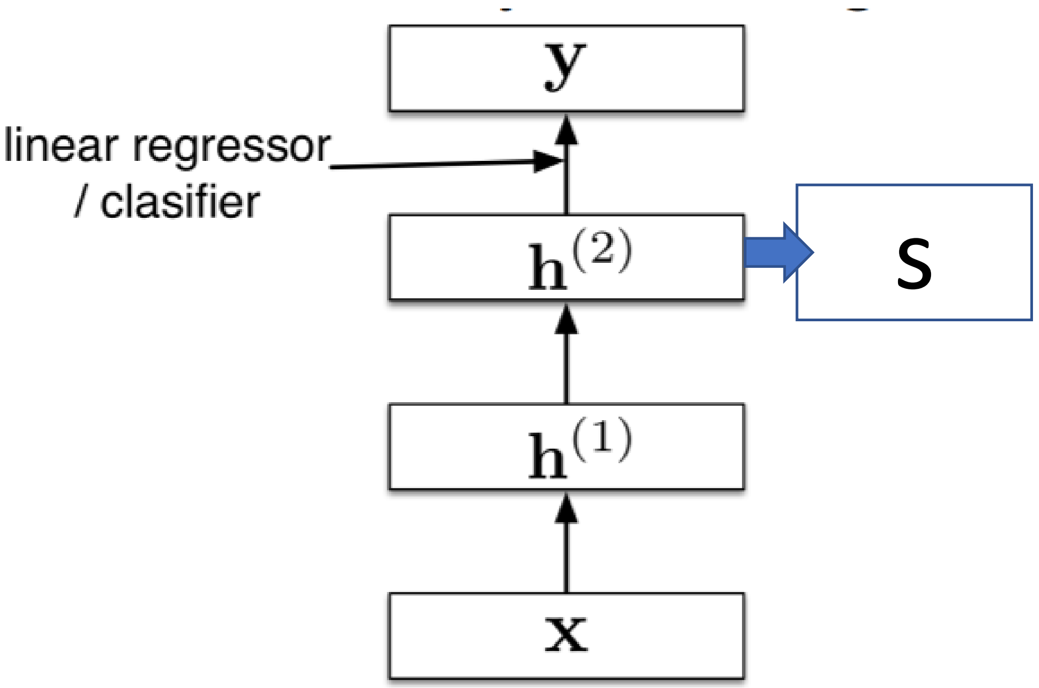
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

Mathematical definition

- Feed-forward neural network.

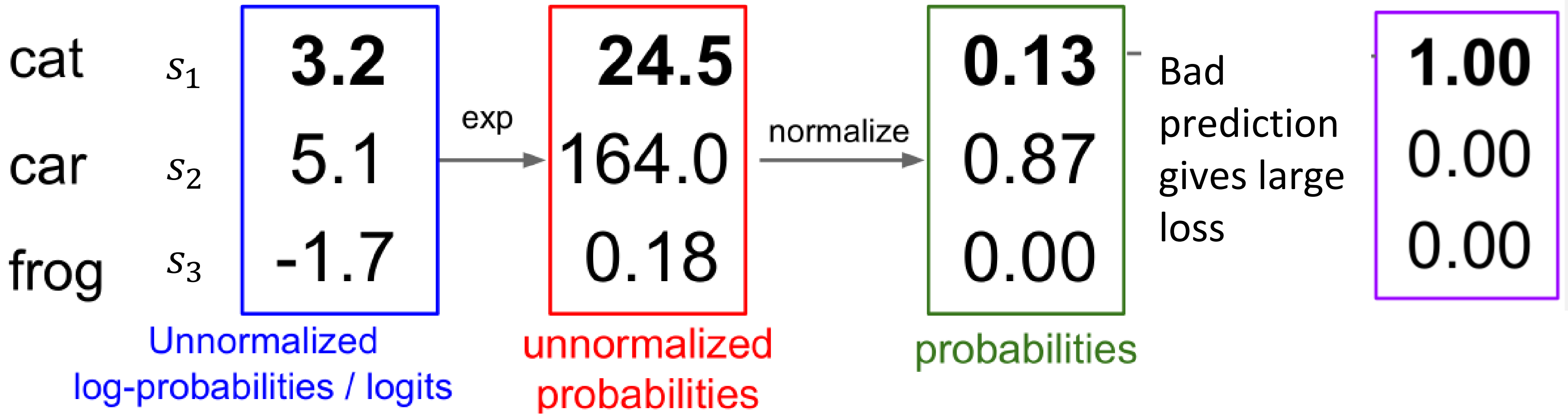
$$h^{(1)} = \phi^{(1)}(xW_1 + b_1)$$
$$s = h^{(2)} = \phi^{(2)}\left(h^{(1)}W_2 + b_2\right)$$

- How to build a linear classifier using the scores?



Softmax function

$$P(Y=k|X=x_i)=\frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function



Measure the goodness-of-fit

$$\boldsymbol{y}_i = (y_{i1}, y_{i2}, y_{i3})$$

	probabilities			
cat	**0.13**	p_{i1}	**1.00**	y_{i1}
car	0.87	p_{i2}	0.00	y_{i2}
frog	0.00	p_{i3}	0.00	y_{i3}

Likelihood

$$p_{i1}^{y_{i1}} p_{i2}^{y_{i2}} p_{i3}^{y_{i3}}$$

Negative Log-likelihood (Cross Entropy)

$$\mathcal{L}_i = -\log(P(Y = \boldsymbol{y}_i | X = \boldsymbol{x}_i))$$

$$-(y_{i1} \log p_{i1} + y_{i2} \log p_{i2} + y_{i3} \log p_{i3})$$

Loss Function

- A loss function tells how good our current classifier is

Given a dataset of examples

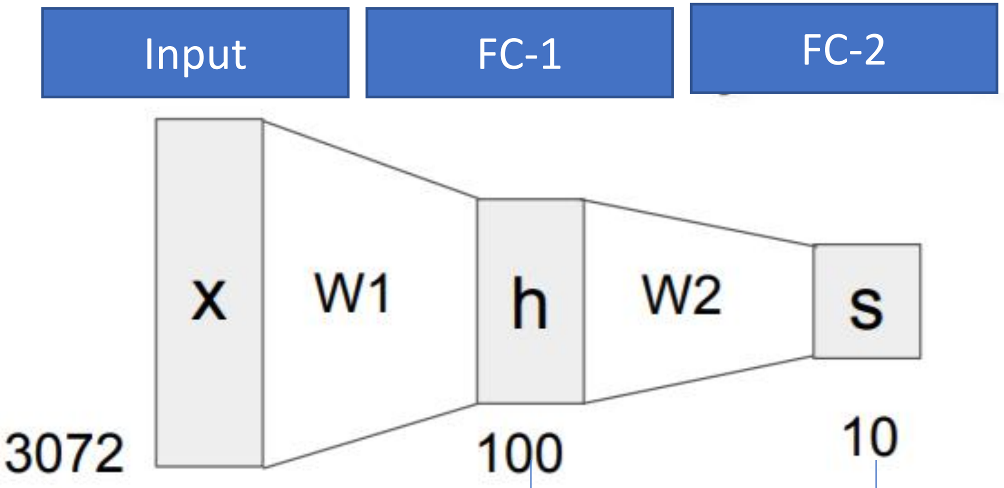
$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and
y_i is (integer) label

Loss over the dataset is a
average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i) \qquad L_i = -\log P(Y = y_i | X = x_i)$$

Number of parameters in fully connected NNs

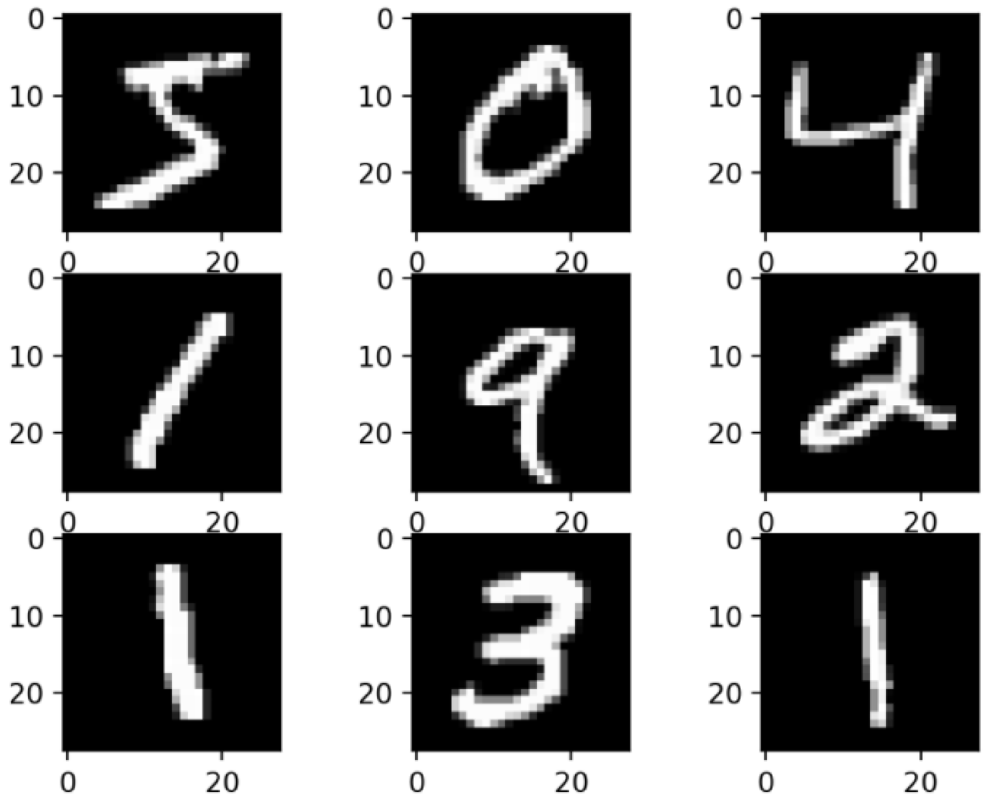


$$h = f_1(W_1 x + b_1)$$
$$W_1 \in \mathbb{R}^{3072\times 100}, b_1 \in \mathbb{R}^{100}$$
# of parameters:
3072x100+100

$$s = f_2(W_2 h + b_2)$$
$$W_2 \in \mathbb{R}^{100\times 10}, b_1 \in \mathbb{R}^{10}$$
# of parameters:
100x10+10

Example with MNIST data

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.



Create MPL using PyTorch NN module

- Module: Base class for all neural network modules and Your models should also subclass this class.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return F.log_softmax(x, dim=1)
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
net = Net().to(device)
```

Training with autograd

```
loss_criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.005)
```

```
for epoch in range(10):
    for data in trainset:
        X, y = data
        net.zero_grad()
        output = net(X.view(-1,784))
        loss = loss_criterion(output, y)
        loss.backward()
        optimizer.step()
    print(loss)
```

Evaluate test accuracy

```
correct = 0
total = 0

with torch.no_grad():
    for data in testset:
        X, y = data
        X = X.to(device)
        y = y.to(device)
        output = net(X.view(-1,784))

        for idx, i in enumerate(output):
          if torch.argmax(i) == y[idx]:
            correct += 1
          total += 1

print("Accuracy: ", round(correct/total, 2))
```

torch.no_grad() impacts the autograd engine and deactivate it. It will reduce memory usage and speed up