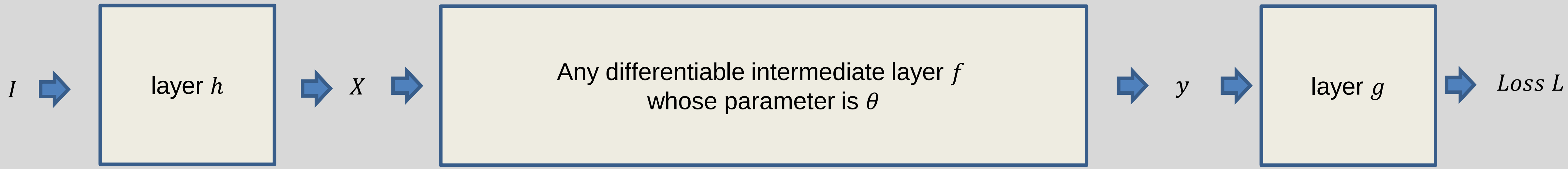
Computer Vision

Lecture 03: Review on PyTorch

Differentiable layers



We need to implement three things for an intermediate layer f :

forward rule:	$y = f(X;\theta)$	for	$g\big(f(X;\theta)\big) = \mathrm{L}$
backward rule:	$\frac{dy}{dX}$	for	$\frac{dL}{d\mathrm{X}} = \frac{dy}{d\mathrm{X}} \times \frac{dL}{d\mathrm{y}}$
parameter update rule:	$\frac{dy}{d\theta}$	for	$\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{d\mathrm{y}}$

Differentiable layers

I ⇨ layer h ⇨ X ⇨ Any differentiable intermediate layer f whose parameter is θ ⇨ y ⇨ layer g ⇨ $Loss\ \mathrm{L}$

We need to implement three things for an intermediate layer f :

forward rule: $y = f(X;\theta)$ for $g\big(f(X;\theta)\big) = \mathrm{L}$

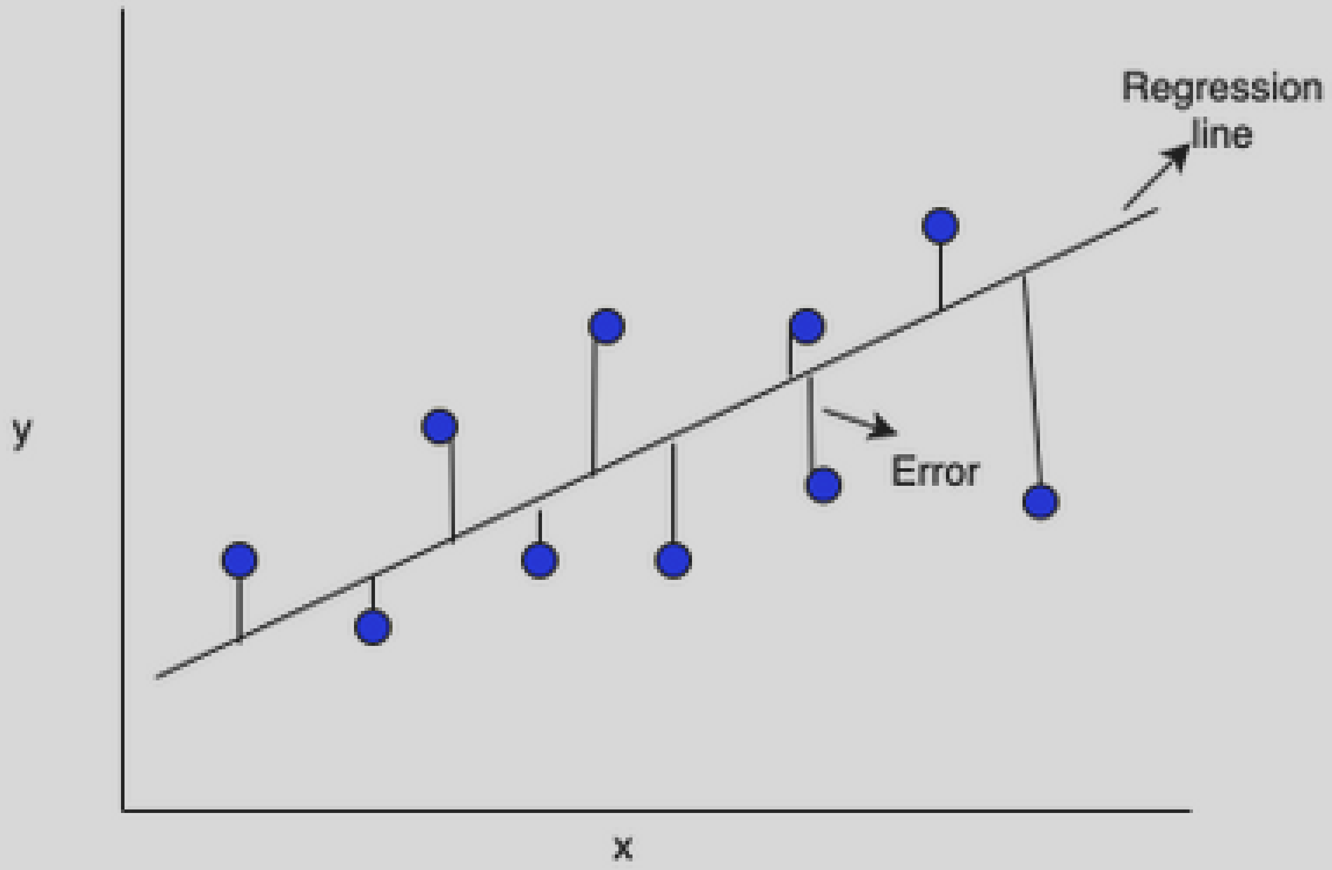
backward rule: $\frac{dy}{dX}$ for $\frac{dL}{dX} = \frac{dy}{dX} \times \frac{dL}{dy}$

parameter update rule: $\frac{dy}{d\theta}$ for $\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{dy}$

PyTorch can do these automatically.

loss.backward()
optimizer.step()

Linear regression



Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Learning rate and the variable we will accumulate our loss.

Loop until 100 iteration to change (W, b) solution using gradient descent.

Regression loss.

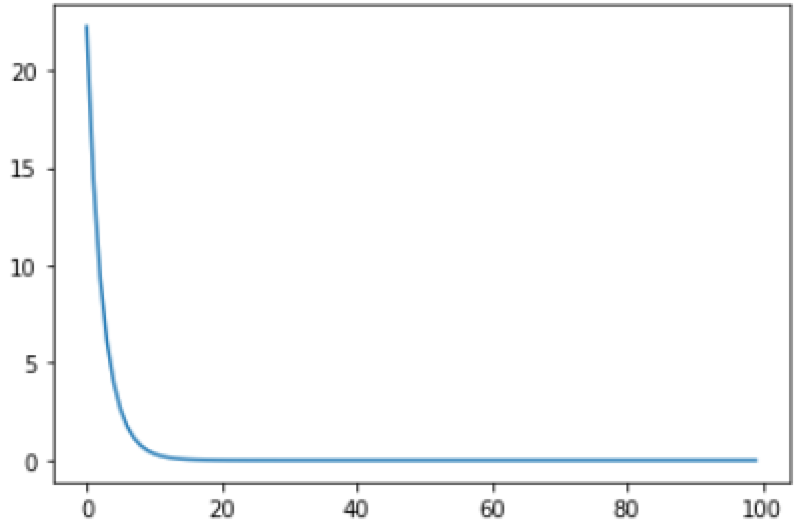
Gradient descent formula.

Implementing linear regression using PyTorch

```
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)

from matplotlib import pyplot as plt
plt.plot(losses)
```

Compare obtained (W, b) with their ground-truth.
It's same!

```
obtained w:  tensor([1.0000, 2.0000, 3.0000], requires_grad=True) true w: tensor([1., 2., 3.])
obtained b:  tensor([5.0000], requires_grad=True) true b: 5
[<matplotlib.lines.Line2D at 0x7f5db4a31f28>]
```



Implementing linear regression using PyTorch

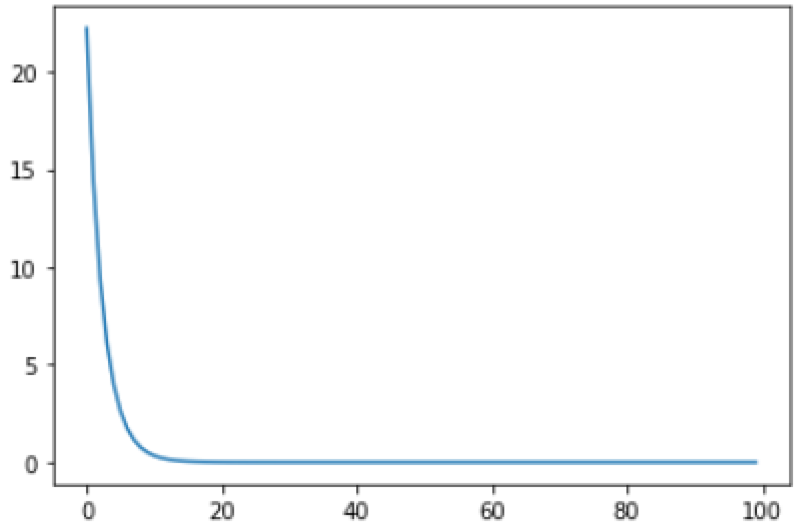
```python
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
```

Compare obtained (W, b) with their ground-truth.
It's same!

```python
from matplotlib import pyplot as plt
plt.plot(losses)
```

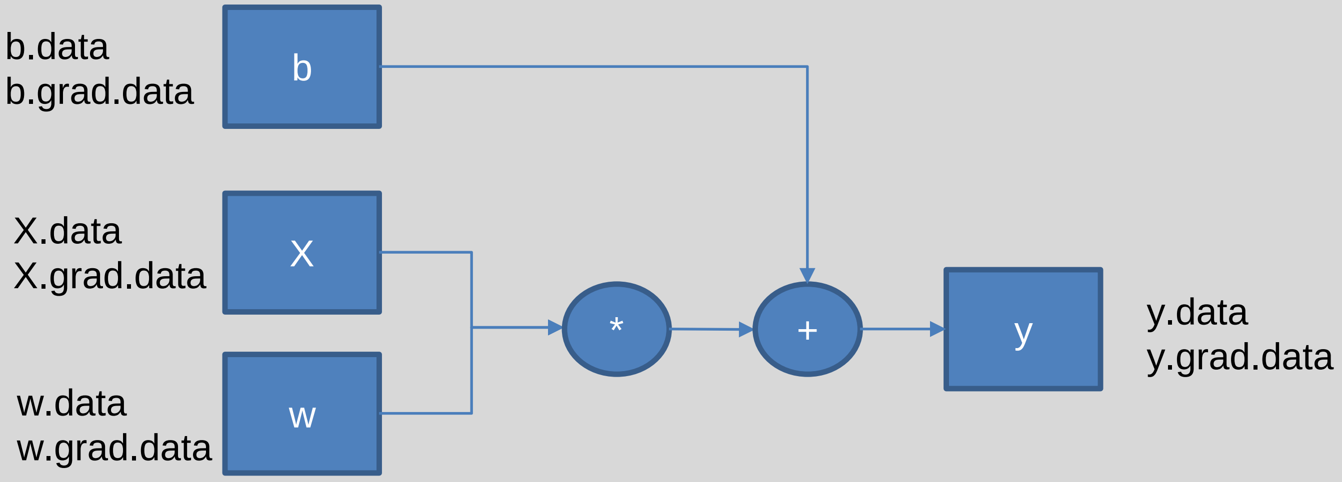
The loss plotted for each iteration. It is gradually reduced!

```
obtained w:  tensor([1.0000, 2.0000, 3.0000], requires_grad=True) true w: tensor([1., 2., 3.])
obtained b:  tensor([5.0000], requires_grad=True) true b: 5
[<matplotlib.lines.Line2D at 0x7f5db4a31f28>]
```



Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

w = torch.randn(3, requires_grad=True)
b = torch.randn(1, requires_grad=True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true

gamma = 0.01
losses = []

for i in range(500):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y-y_pred)**2)
  loss.backward()

  w.data -= gamma * w.grad.data
  b.data -= gamma * b.grad.data

  losses.append(loss.item())

print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
from matplotlib import pyplot as plt
plt.plot(losses)
```



After calling loss.backward(), .grad values are calculated.

PyTorch-style code

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

PyTorch-style code

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = net(X)

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

PyTorch-style code

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

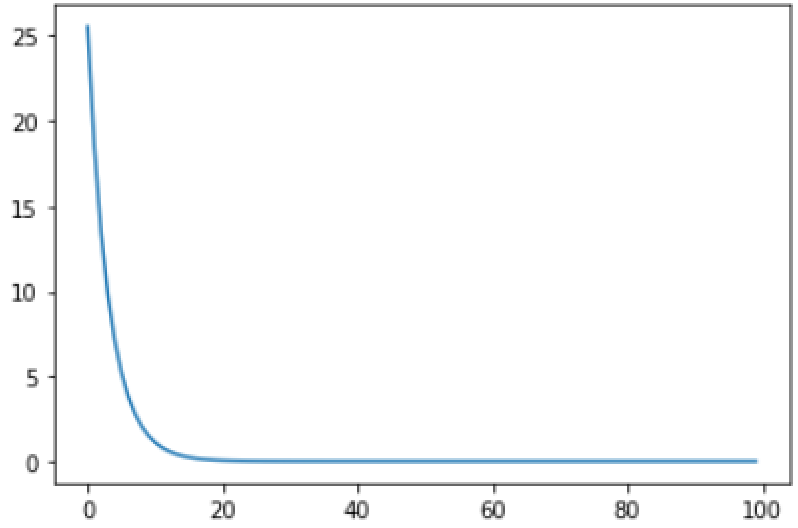
```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)
loss_fn = torch.nn.MSELoss()

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = loss_fn(y_pred.squeeze(1),y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

PyTorch-style code

```
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)

from matplotlib import pyplot as plt
plt.plot(losses)
```

```
print(list(net.parameters()))

from matplotlib import pyplot as plt
plt.plot(losses)
```

```
[Parameter containing:
tensor([[1.0000, 2.0000, 3.0000]], requires_grad=True), Parameter containing:
tensor([5.0000], requires_grad=True)]
[<matplotlib.lines.Line2D at 0x7f32a3f21710>]
```



PyTorch-style code

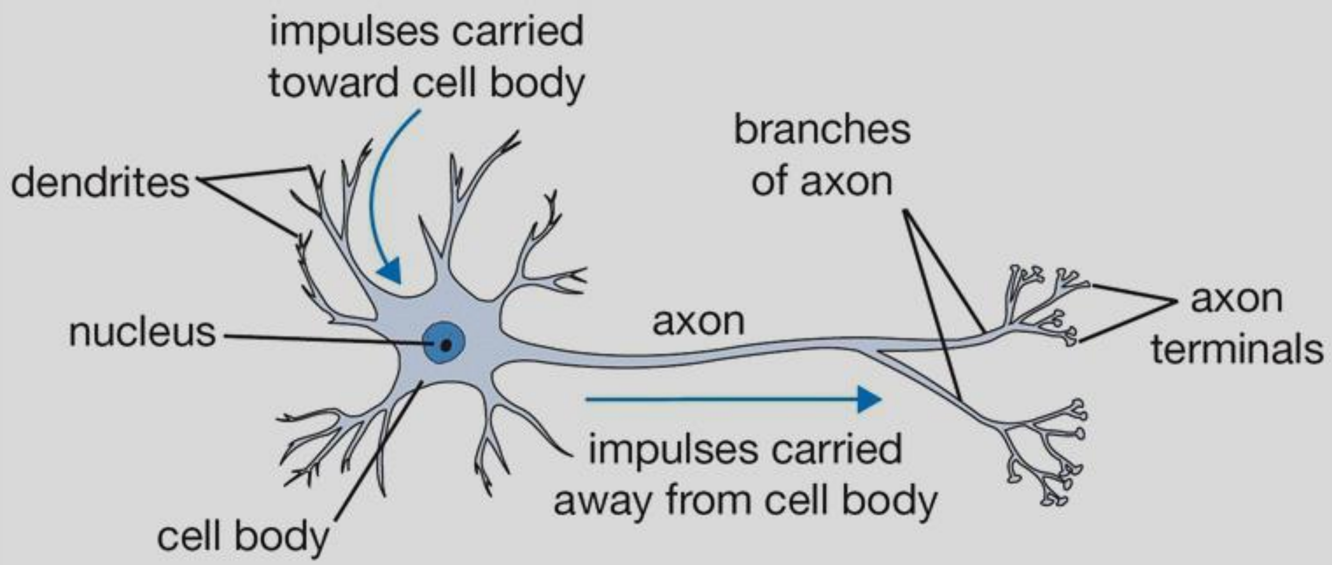
```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)
optimizer = torch.optim.SGD(net.parameters(), lr=gamma)
loss_fn = torch.nn.MSELoss()

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = loss_fn(y_pred.squeeze(1),y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())

print(list(net.parameters()))

from matplotlib import pyplot as plt
plt.plot(losses)
```

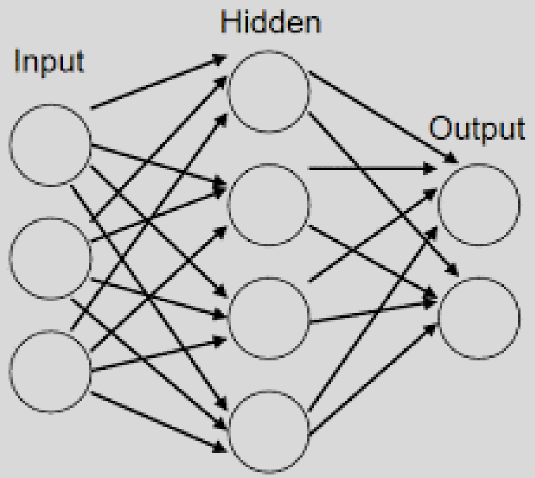
Data Preparation.

Network structure
Optimizer
Loss

Iterate for updating network parameters.
Initialize. gradients.
Forward pass.
Calculate loss.
Backward pass (Calc. gradients.).
Update network parameter.

Multi-layer perceptron





$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1 x + b_1)) + b_2) + b_3))$$

Multi-layer perceptron

$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1 x+b_1))+b_2)) +b_3))$$

σ : Sigmoid, Tanh, ReLu and so on…

These functions are also differentiable.

Multi-layer perceptron

Why we need σ ?

$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1 x + b_1)) + b_2)) + b_3))$$

$$\begin{aligned} y &= w_3(w_2(w_1 x + b_1) + b_2) + b_3 \\ &= (w_3 w_2 w_1)x + (w_3 w_2 b_1 + w_3 b_2 + b_3) \\ &= wx + b \end{aligned}$$

➡ Huge network converges to a simple linear regression task.

Implementing Multi-layer perceptron using PyTorch

```
import torch

num_data = 1000
num_epoch = 10000
x = torch.randn(num_data, 1)
y = (x**2) + 3

net = torch.nn.Sequential(
    torch.nn.Linear(1,6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 1),
)

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Make data.

Multi-layer perceptron structure.

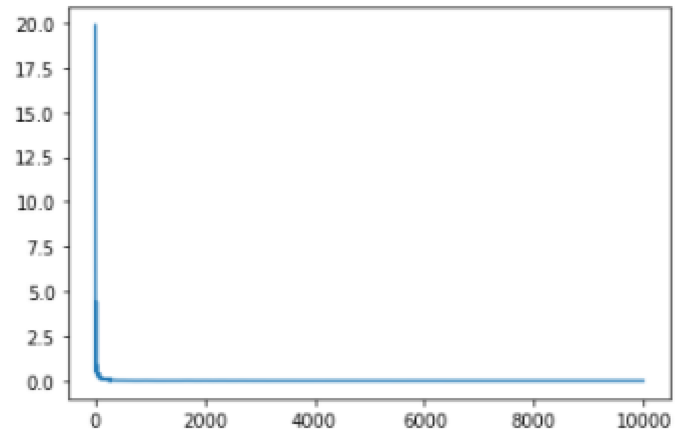
Define loss function and optimizer.

Optimize network through iterations.

Implementing Multi-layer perceptron using PyTorch

```python
from matplotlib import pyplot as plt
plt.plot(losses)
```

```
[<matplotlib.lines.Line2D at 0x7f32a25a5400>]
```



Implementing Multi-layer perceptron using PyTorch

```
x = torch.randn(5, 1)
y = (x**2) + 3
y_pred = net(x)

print(y)
print(y_pred)
```

```
tensor([[5.5024],
        [3.3881],
        [3.0404],
        [3.0684],
        [3.2888]])
tensor([[5.5617],
        [3.3964],
        [3.0547],
        [3.0892],
        [3.2588]], grad_fn=<AddmmBackward>)
```

CNNs

RGB image

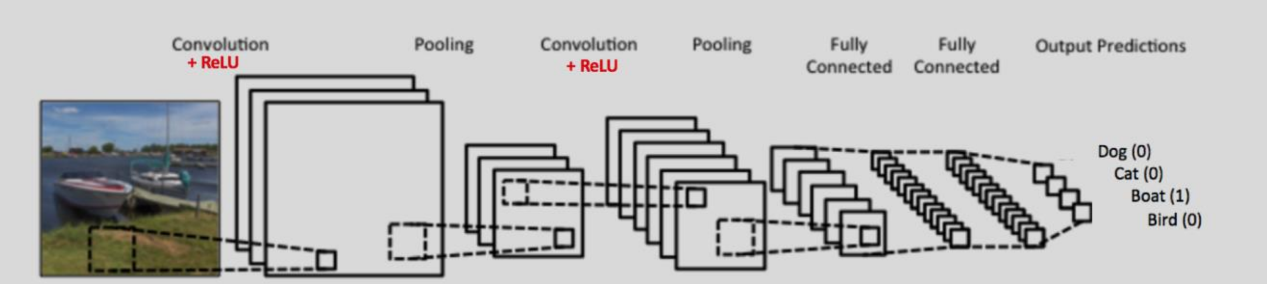
Any Differentiable Layers

cat

Semantic labels

CNNs

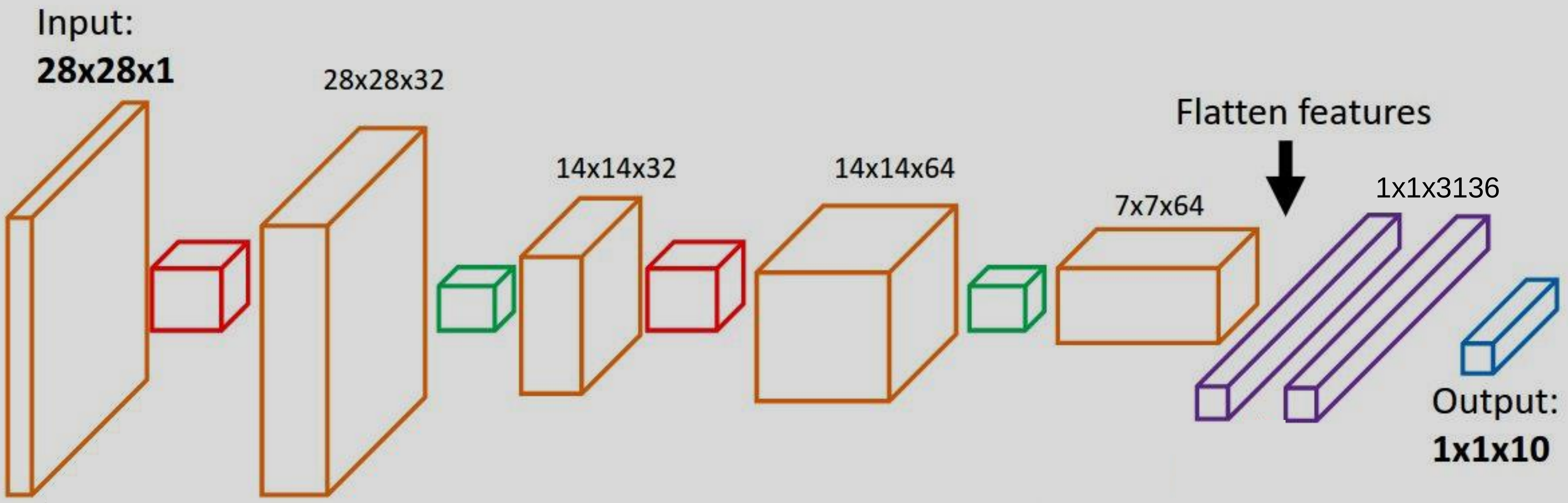
RGB image



cat

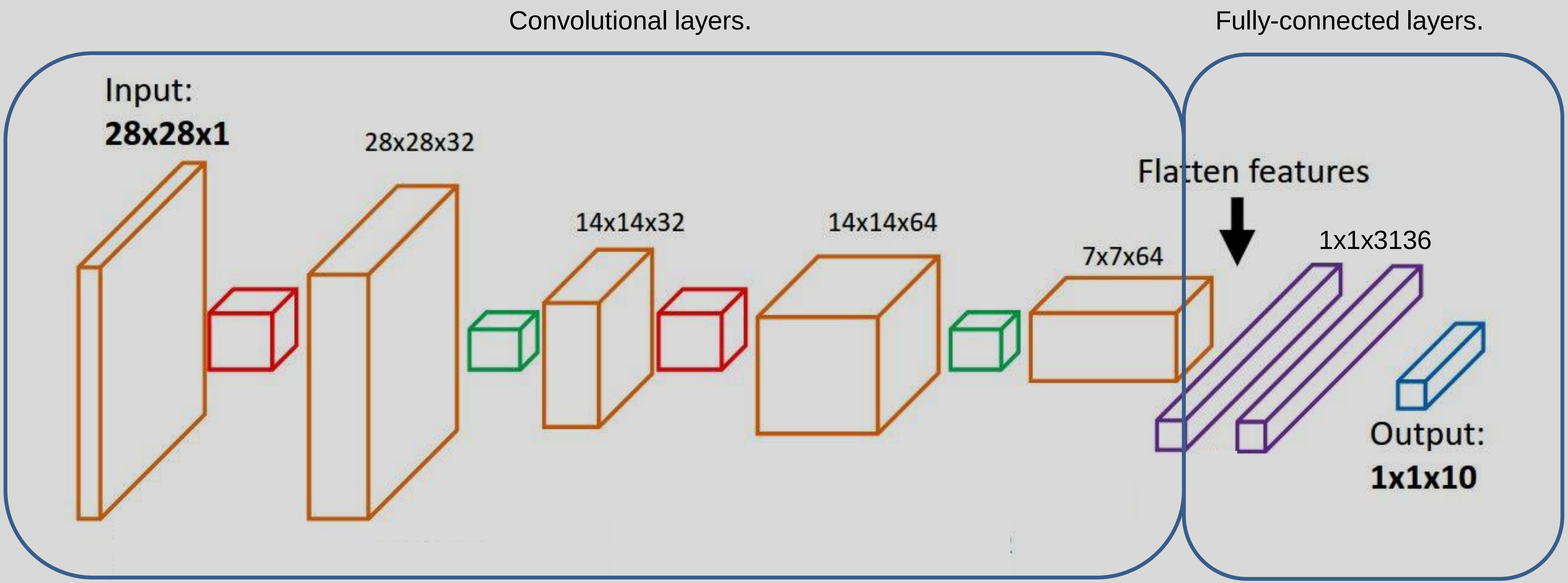
Semantic labels

Overall CNN architecture



Combination of differentiable layers → Differentiable architecture!

Overall CNN architecture



Combination of differentiable layers → Differentiable architecture!

CNN implementation in PyTorch

```
import torch

class MyCNN(torch.nn.Module):
  def __init__(self):
    super().__init__()



  def forward(self, x):
    return x
```

Called when your network is initialized.

Called when the forward pass is performed on input x.

CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out = out.view*(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

Convolutional layers are generated.

Fully connected layers are generated.

Forward pass is defined.

Backward is automatically performed when calling loss.backward()

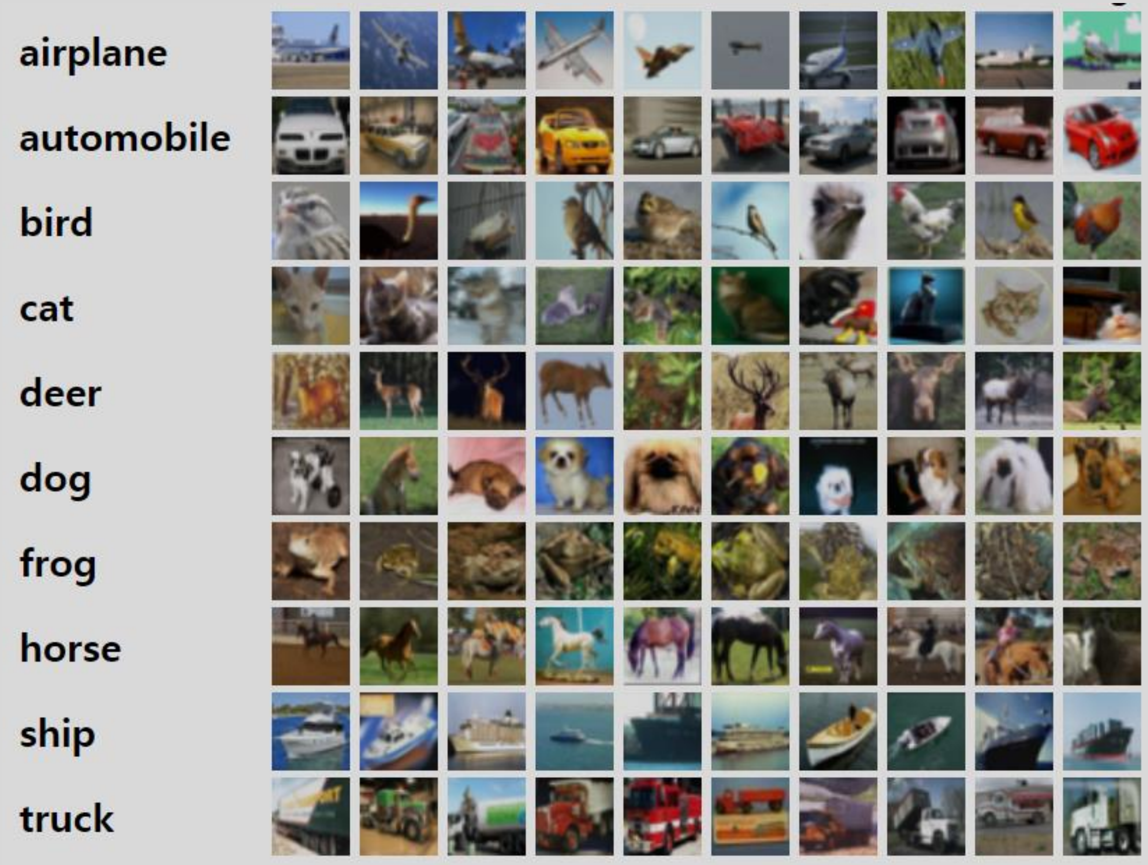
CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out = out.view*(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

```
import torch

net = MyCNN()

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Image Classification task

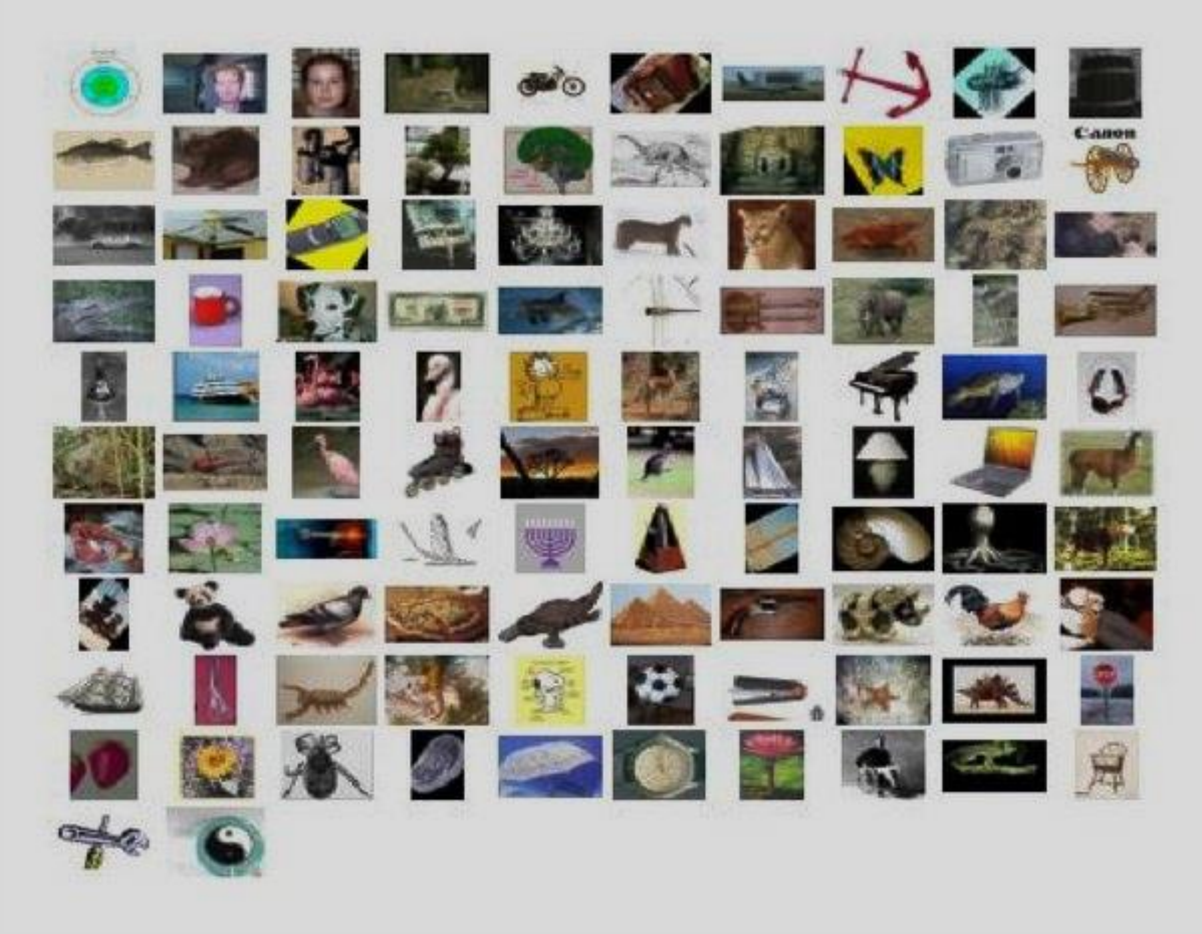
Cifar 10 dataset (10 classes)



MNIST dataset (10 classes)

Image Classification task

Caltech 101 dataset (101 classes)

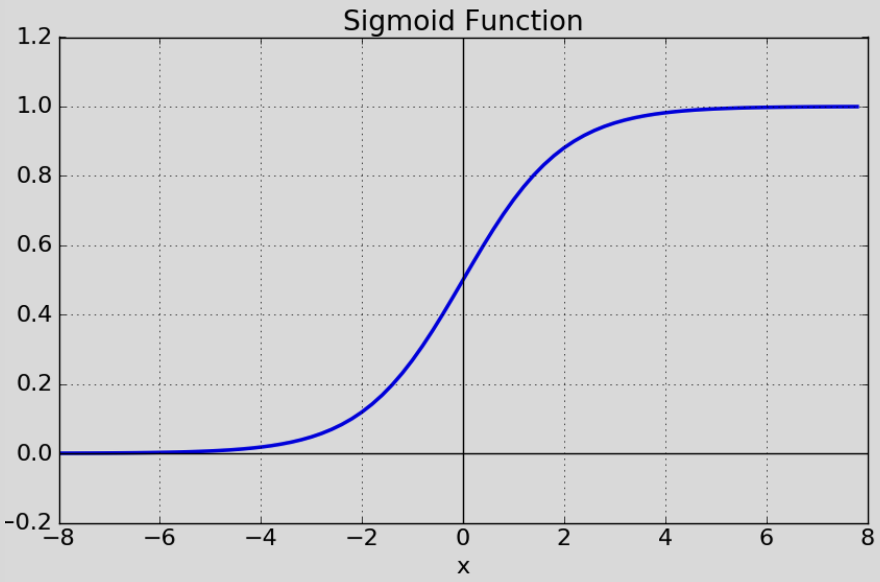


ImageNet dataset (1000 classes)

Logistic Regression

Called as the regression, but actually performs the **binary classification**!

$$z = \frac{1}{1 + e^{-\mathbf{w}x+b}}$$
$$= \sigma(-\mathbf{w}x + b)$$



Logistic Regression

$$Loss = \begin{cases} -\ln z_n, y_n = 1 \\ -\ln(1 - z_n), y_n = 0 \end{cases}$$

$$Loss = -\sum_n y_n \ln z_n + (1 - y_n) \ln(1 - z_n)$$

Logistic Regression

```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[:100]
y = iris.target[:100]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

losses = []

for epoch in range(100):
  optimizer.zero_grad()

  h = net(X)
  prob = nn.functional.sigmoid(h)

  loss = loss_fn(prob, y)
  loss.backward()

  optimizer.step()
  losses.append(loss.item())
```

Logistic Regression

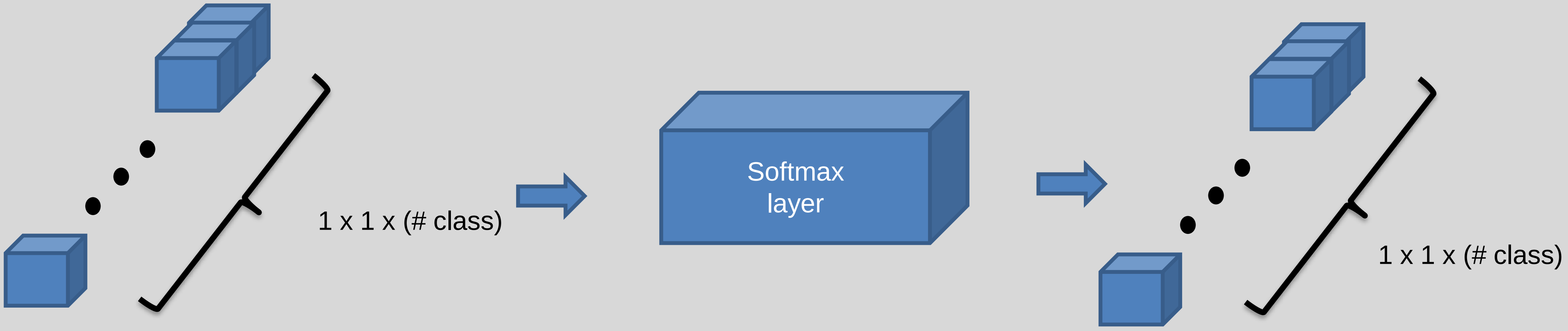
```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[:100]
y = iris.target[:100]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

losses = []

for epoch in range(100):
  optimizer.zero_grad()

  h = net(X)

  loss = loss_fn(h.view_as(y), y)
  loss.backward()

  optimizer.step()
  losses.append(loss.item())
```

Logistic Regression

```
import torch
import torch.nn as nn
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[[1, 51]]
y = iris.target[[1, 51]]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

net = nn.Linear(4, 1)

h=net(X)
prob = nn.functional.sigmoid(h)

loss_fn = nn.BCELoss()
loss_fn2 = nn.BCEWithLogitsLoss()

loss1 = loss_fn(prob, y)
loss2 = loss_fn2(h.view_as(y), y)

print(loss1, loss2)
```

```
tensor(1.2881, grad_fn=<BinaryCrossEntropyBackward>) tensor(1.2881, grad_fn=<BinaryCrossEntropyWithLogitsBackward>)
```

Softmax for multi-class classification



$$Softmax(y_i) = \frac{\exp(y_i)}{\sum_j \exp(y_j)}$$

The vector is L1-normalized. → It could mean probability for semantic classes.

Cross-entropy Loss

$$H(p,q) = -\sum_x p(x)\log q(x)$$

$$= \underbrace{-\sum_x p(x)\log p(x)}_{\text{Constant}} + \underbrace{\sum_x p(x)\log\frac{p(x)}{q(x)}}_{\text{KL divergence}}$$

Cross-entropy Loss

$$D_{KL}(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

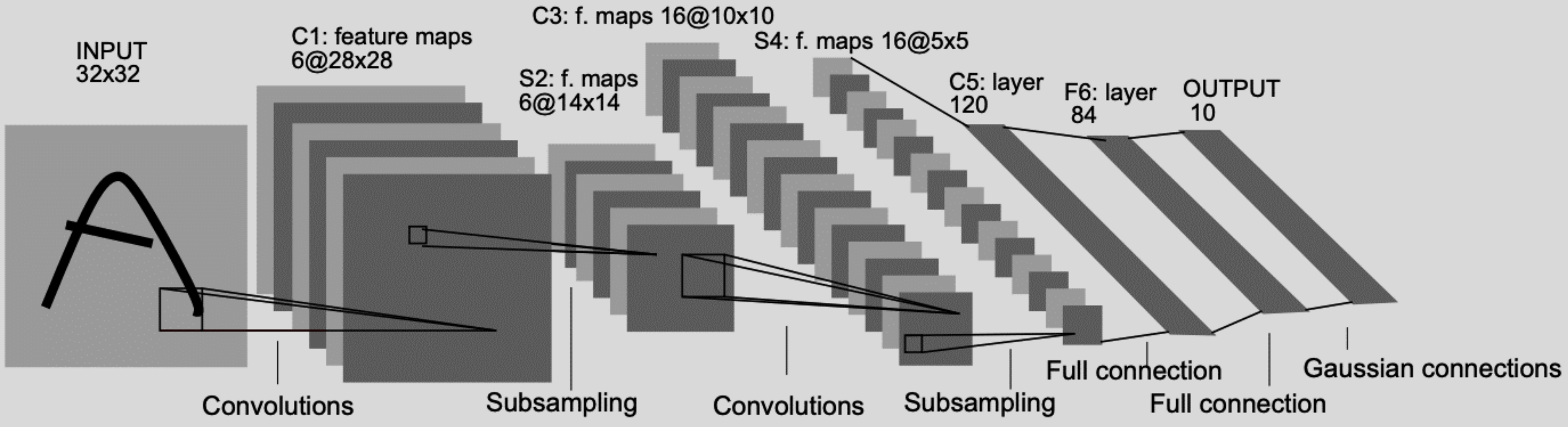
The value becomes 0 when p(x) = q(x). It is the minimum.

Cross-entropy Loss

```
import torch
import torch.nn as nn

loss = nn.CrossEntropyLoss()

input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)

output = loss(input, target)

print(input)
print(target)
print(output)
```

```
tensor([[-1.8600, -1.1599,  0.0525, -1.9408, -1.4070],
        [ 0.6205, -1.8228,  2.1522, -2.5549, -0.1288],
        [ 0.1473, -1.5891,  1.0388, -0.6910,  0.4188]], requires_grad=True)
tensor([2, 1, 0])
tensor(2.1822, grad_fn=<NllLossBackward>)
```

LeNet



[Yann Lecun 1998.]

Implementing LeNet

```
import numpy as np

import torch
import torch.nn as nn

from torch.utils.data import DataLoader
from torchvision import datasets, transforms

import matplotlib.pyplot as plt

LEARNING_RATE = 0.001
BATCH_SIZE = 32
N_EPOCHS = 100

IMG_SIZE = 32
N_CLASSES = 10
```

LeNet

```python
class LeNet5(nn.Module):

    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )

        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
        )


    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Data Loader

```
trans = transforms.Compose([transforms.Resize((32, 32)), transforms.ToTensor()])

train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=trans, download=True)
test_dataset = datasets.MNIST(root = 'mnist_data', train=False, transform=trans)

train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Frequently used Datasets are readily available!

https://pytorch.org/vision/stable/datasets.html

Data Loader

```
trans = transforms.Compose([transforms.Resize((32, 32)), transforms.ToTensor()])

train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=trans, download=True)
test_dataset = datasets.MNIST(root = 'mnist_data', train=False, transform=trans)

train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

You can also define your own dataset for data loader.

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

class CustomDataset(Dataset):

    def __init__(self):
        self.x_data = [[73, 80, 75], [93, 88, 93], [89, 91, 90], [96, 98, 100], [73, 66, 70]]
        self.y_data = [[152], [185], [180], [196], [142]]

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

    def __getitem__(self, idx):
        x = torch.FloatTensor(self.x_data[idx])
        y = torch.FloatTensor(self.y_data[idx])
        return x, y
```

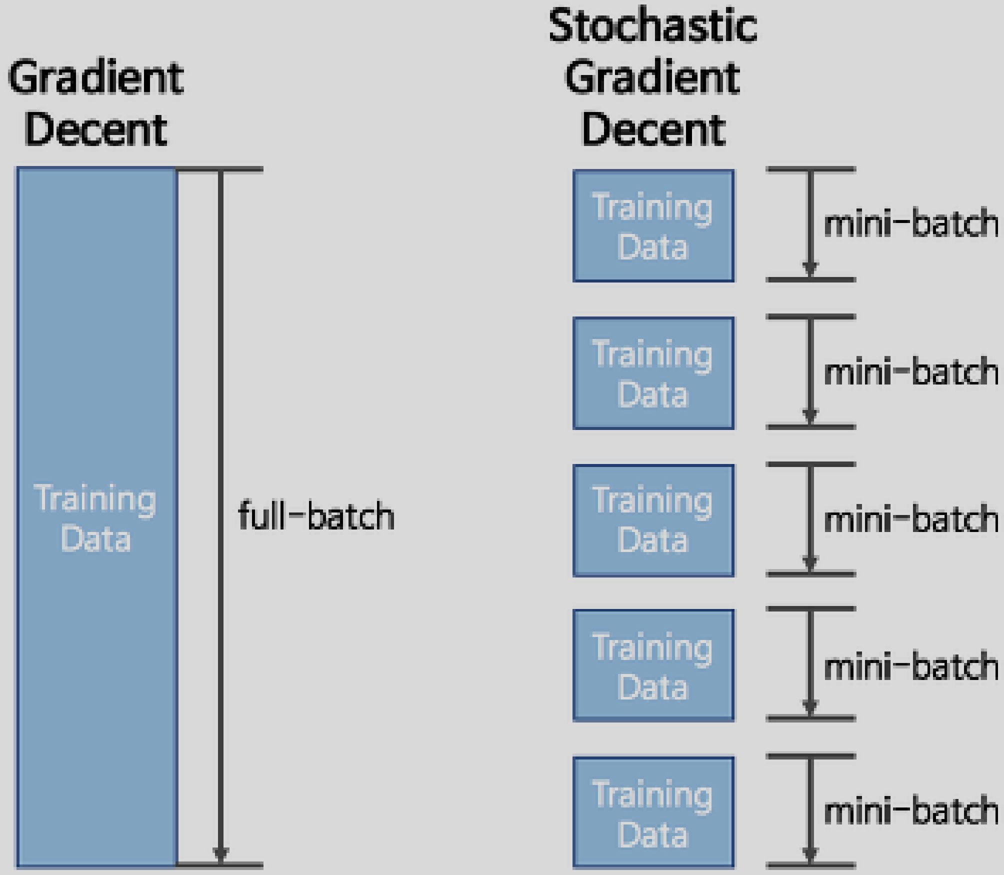
```
for X, y_true in train_loader:
```

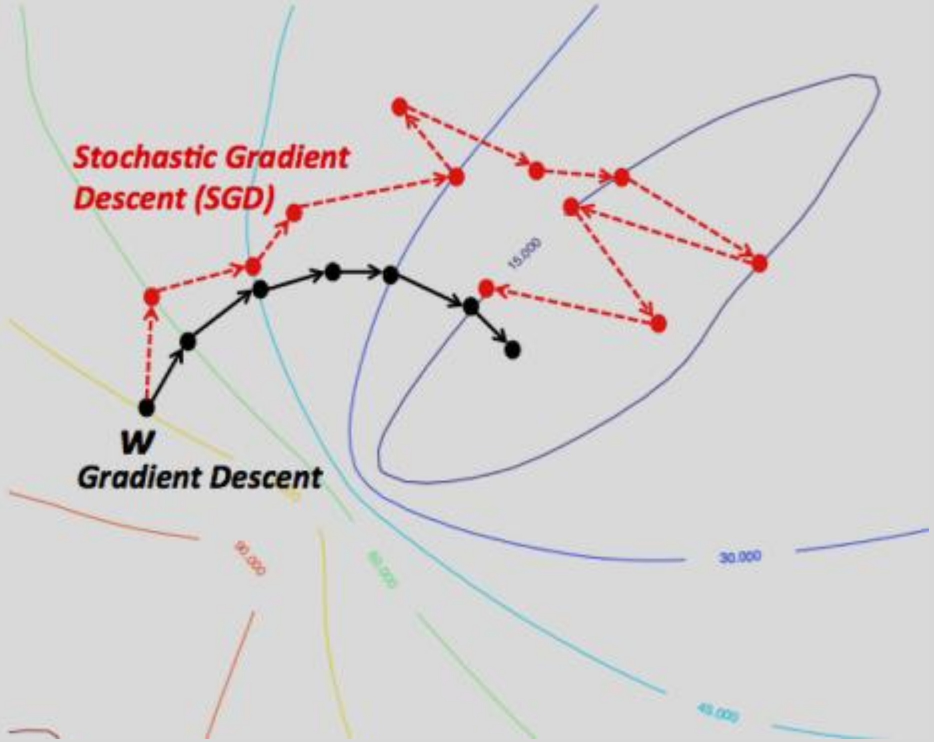
Stochastic Gradient Descent

- Gradient descent
  Calculate for all data (takes large amount of time).
  Go 1 optimal step.

- Stochastic gradient descent
  Calculate gradients for partial data (takes small amount of time).
  Go many non-globally-optimal steps, but converges.

Stochastic Gradient Descent





Implementing LeNet

```
def train(train_loader, model, criterion, optimizer):

    model.train()
    train_loss = 0
    correct = 0

    for X, y_true in train_loader:

        optimizer.zero_grad()

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        train_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

        loss.backward()
        optimizer.step()

    epoch_loss = train_loss / len(train_loader.dataset)
    acc = correct / len(train_loader.dataset)

    return model, optimizer, epoch_loss, acc
```

Training flag .train()

Same as before.

Implementing LeNet

```
def test(test_loader, model, criterion):

    model.eval()
    test_loss = 0
    correct = 0

    for X, y_true in test_loader:

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        test_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

    epoch_loss = test_loss / len(test_loader.dataset)
    acc = correct / len(test_loader.dataset)

    return model, epoch_loss, acc
```

Testing flag .eval()

Same as before, but without backward step.

Implementing LeNet

```
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, print_every=1):

    train_losses = []
    test_losses = []

    for epoch in range(epochs):

        model, optimizer, train_loss, train_acc = train(train_loader, model, criterion, optimizer)
        train_losses.append(train_loss)

        with torch.no_grad():
            model, test_loss, test_acc = test(test_loader, model, criterion)
            test_losses.append(test_loss)

        if epoch % print_every == (print_every - 1):

            print(f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Test loss: {test_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Test accuracy: {100 * test_acc:.2f}')

    return model, optimizer, (train_losses, test_losses)
```

No gradient calculation mode.

Implementing LeNet

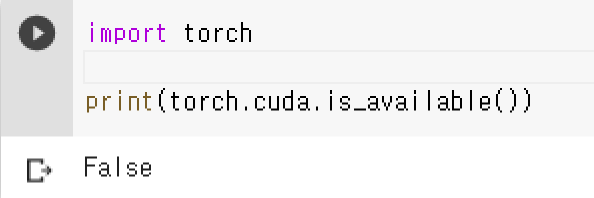
```
model = LeNet5(N_CLASSES)

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
criterion = nn.CrossEntropyLoss()

model, optimizer, _ = training_loop(model, criterion, optimizer, train_loader, test_loader, N_EPOCHS)
```

Training using CPU

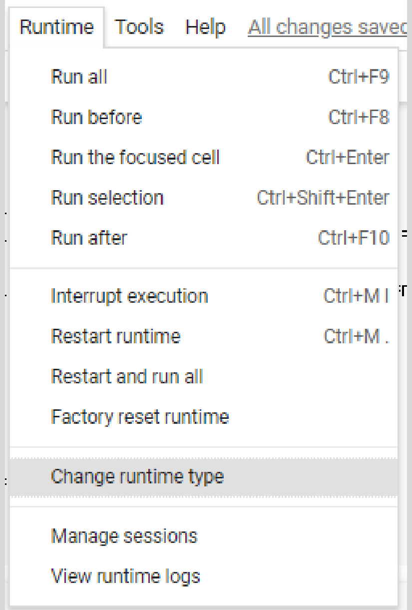
```
Epoch: 0 Train loss: 2.2817 Test loss: 2.2540 Train accuracy: 14.16 Test accuracy: 34.11
Epoch: 1 Train loss: 2.1948 Test loss: 2.0867 Train accuracy: 50.41 Test accuracy: 56.95
Epoch: 2 Train loss: 1.8551 Test loss: 1.5684 Train accuracy: 56.49 Test accuracy: 62.10
Epoch: 3 Train loss: 1.3052 Test loss: 1.0489 Train accuracy: 67.44 Test accuracy: 74.17
Epoch: 4 Train loss: 0.9017 Test loss: 0.7621 Train accuracy: 77.70 Test accuracy: 81.95
Epoch: 5 Train loss: 0.6930 Test loss: 0.6112 Train accuracy: 82.86 Test accuracy: 85.24
Epoch: 6 Train loss: 0.5783 Test loss: 0.5238 Train accuracy: 85.29 Test accuracy: 86.71
Epoch: 7 Train loss: 0.5090 Test loss: 0.4686 Train accuracy: 86.75 Test accuracy: 87.75
Epoch: 8 Train loss: 0.4633 Test loss: 0.4298 Train accuracy: 87.60 Test accuracy: 88.62
Epoch: 9 Train loss: 0.4306 Test loss: 0.4009 Train accuracy: 88.34 Test accuracy: 89.17
Epoch: 10 Train loss: 0.4056 Test loss: 0.3788 Train accuracy: 88.82 Test accuracy: 89.68
Epoch: 11 Train loss: 0.3855 Test loss: 0.3602 Train accuracy: 89.27 Test accuracy: 90.08
Epoch: 12 Train loss: 0.3686 Test loss: 0.3444 Train accuracy: 89.65 Test accuracy: 90.45
Epoch: 13 Train loss: 0.3540 Test loss: 0.3310 Train accuracy: 89.95 Test accuracy: 90.75
Epoch: 14 Train loss: 0.3407 Test loss: 0.3183 Train accuracy: 90.29 Test accuracy: 91.00
```

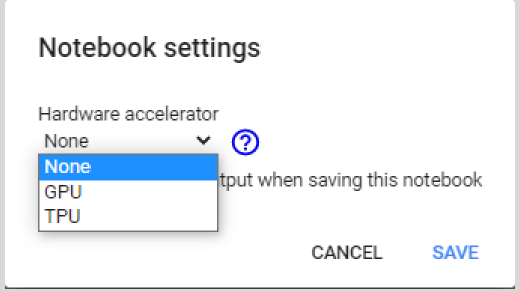
It takes about 15 minutes for training 15 epochs. – **slow!**

Using GPUs in Colab

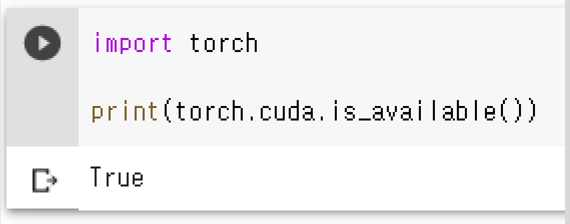


Colab does not support GPUs by default.





Select GPU.



Colab now supports GPUs!

Implementing LeNet

```
def train(train_loader, model, criterion, optimizer, device):

    model.train()
    train_loss = 0
    correct = 0

    for X, y_true in train_loader:
        optimizer.zero_grad()

        X = X.to(device)
        y_true = y_true.to(device)

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        train_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

        loss.backward()
        optimizer.step()

    epoch_loss = train_loss / len(train_loader.dataset)
    acc = correct / len(train_loader.dataset)

    return model, optimizer, epoch_loss, acc
```

Implementing LeNet

```
def test(test_loader, model, criterion, device):

    model.eval()
    test_loss = 0
    correct = 0

    for X, y_true in test_loader:

        X = X.to(device)
        y_true = y_true.to(device)

        y_hat = model(X)
        loss = criterion(y_hat, y_true)

        test_loss += loss.item()
        pred = y_hat.argmax(dim=1, keepdim=True)
        correct += pred.eq(y_true.view_as(pred)).sum().item()

    epoch_loss = test_loss / len(test_loader.dataset)
    acc = correct / len(test_loader.dataset)

    return model, epoch_loss, acc
```

Implementing LeNet

```
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, device, print_every=1):

    best_loss = 1e10
    train_losses = []
    test_losses = []

    for epoch in range(epochs):

        model, optimizer, train_loss, train_acc = train(train_loader, model, criterion, optimizer, device)
        train_losses.append(train_loss)

        with torch.no_grad():
            model, test_loss, test_acc = test(test_loader, model, criterion, device)
            test_losses.append(test_loss)

        if epoch % print_every == (print_every - 1):

            print(f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Test loss: {test_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Test accuracy: {100 * test_acc:.2f}')

    return model, optimizer, (train_losses, test_losses)
```

Implementing LeNet

```
DEVICE = ‘cuda’ if torch.cuda.is_available() else ‘cpu’

model = LeNet5(N_CLASSES).to(DEVICE)

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
criterion = nn.CrossEntropyLoss()

model, optimizer, _ = training_loop(model, criterion, optimizer, train_loader, test_loader, N_EPOCHS,
DEVICE)
```

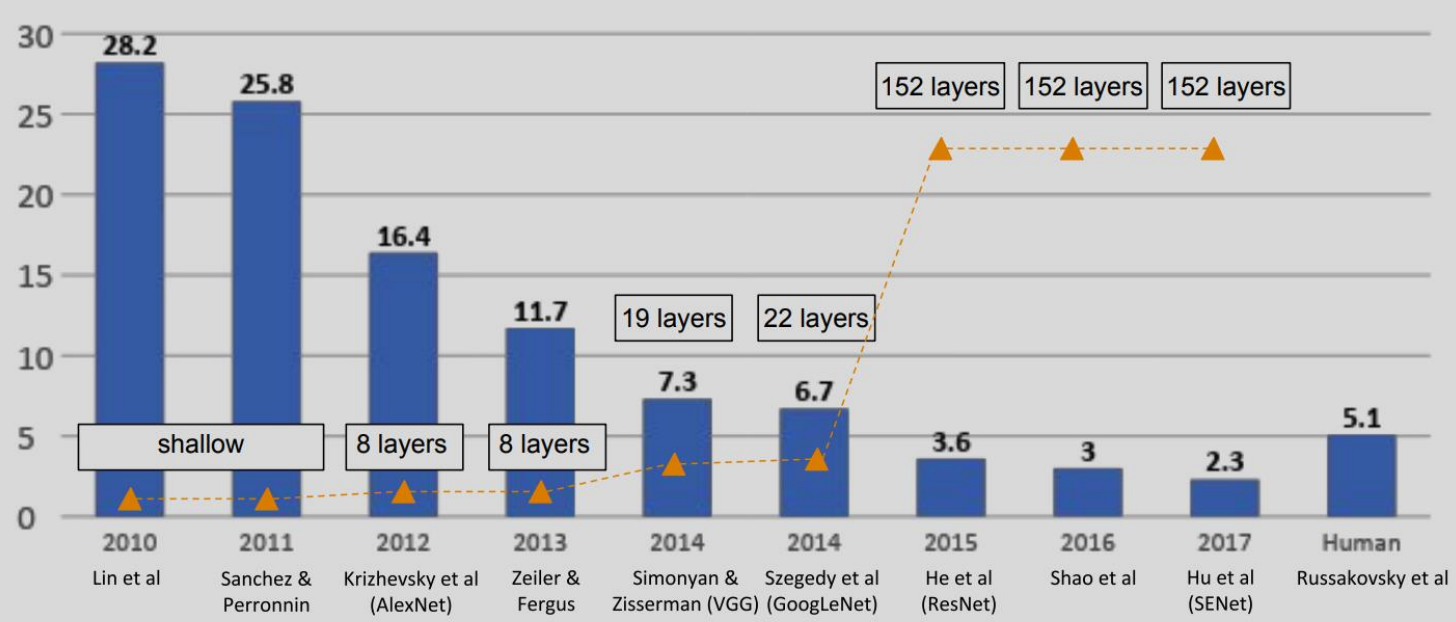
Training using GPU

```
Epoch: 0 Train loss: 2.2817 Test loss: 2.2540 Train accuracy: 14.16 Test accuracy: 34.11
Epoch: 1 Train loss: 2.1948 Test loss: 2.0867 Train accuracy: 50.41 Test accuracy: 56.95
Epoch: 2 Train loss: 1.8551 Test loss: 1.5684 Train accuracy: 56.49 Test accuracy: 62.10
Epoch: 3 Train loss: 1.3052 Test loss: 1.0489 Train accuracy: 67.44 Test accuracy: 74.17
Epoch: 4 Train loss: 0.9017 Test loss: 0.7621 Train accuracy: 77.70 Test accuracy: 81.95
Epoch: 5 Train loss: 0.6930 Test loss: 0.6112 Train accuracy: 82.86 Test accuracy: 85.24
Epoch: 6 Train loss: 0.5783 Test loss: 0.5238 Train accuracy: 85.29 Test accuracy: 86.71
Epoch: 7 Train loss: 0.5090 Test loss: 0.4686 Train accuracy: 86.75 Test accuracy: 87.75
Epoch: 8 Train loss: 0.4633 Test loss: 0.4298 Train accuracy: 87.60 Test accuracy: 88.62
Epoch: 9 Train loss: 0.4306 Test loss: 0.4009 Train accuracy: 88.34 Test accuracy: 89.17
Epoch: 10 Train loss: 0.4056 Test loss: 0.3788 Train accuracy: 88.82 Test accuracy: 89.68
Epoch: 11 Train loss: 0.3855 Test loss: 0.3602 Train accuracy: 89.27 Test accuracy: 90.08
Epoch: 12 Train loss: 0.3686 Test loss: 0.3444 Train accuracy: 89.65 Test accuracy: 90.45
Epoch: 13 Train loss: 0.3540 Test loss: 0.3310 Train accuracy: 89.95 Test accuracy: 90.75
Epoch: 14 Train loss: 0.3407 Test loss: 0.3183 Train accuracy: 90.29 Test accuracy: 91.00
```

Same results but it only takes 1-2 mins. For 15 epochs. – **much faster!**

Alex Net

1. Dropout and data augmentation are used.
2. ReLU is first used (Proves faster convergence).
3. Trained with Multi-GPUs.
4. First architecture that recorded as the SOTA in ImageNet challenge using deep learning.



Alex Net

```
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

VGG Net

1. VGGNet tried to investigate the relationship between the depth of the network and the overall accuracy.

2. Consisted with only 3x3 conv layer, max pooling and fully connected layers.

3. Experimented with 11-layer-model to 19-layer models.

VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_3 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )
```

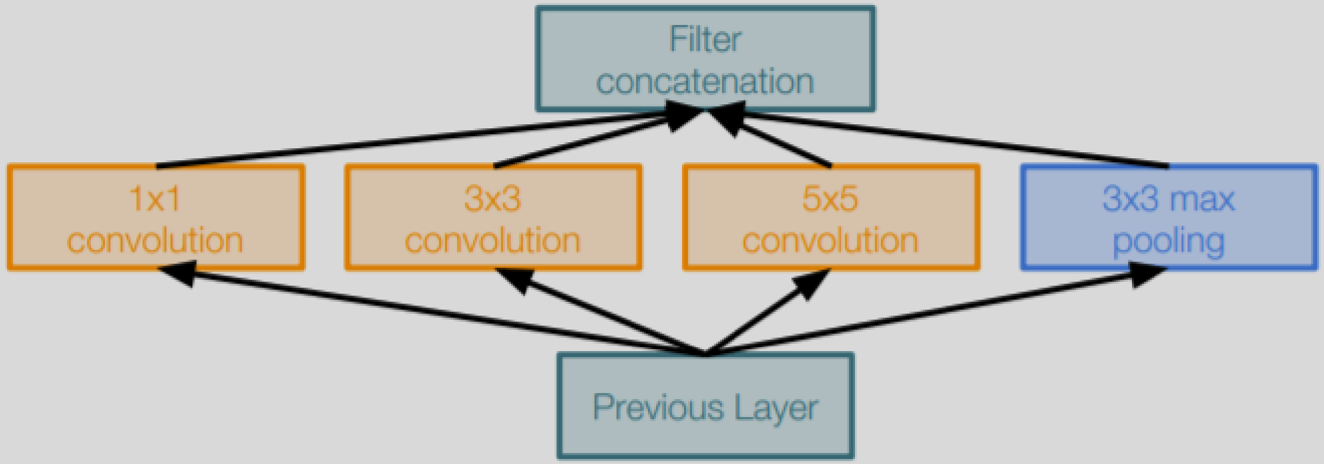
VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.block_4 = nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_5 = nn.Sequential(
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )
```

VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.65),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.65),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):

        x = self.block_1(x)
        x = self.block_2(x)
        x = self.block_3(x)
        x = self.block_4(x)
        x = self.block_5(x)
        x = x.view(x.size(0), -1)
        logits = self.classifier(x)
        probas = F.softmax(logits, dim=1)
        return probas
```

Google LeNet



Naïve "Inception" module:

Apply parallel filter operations on the input from previous layer:
Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
Pooling operation (3x3)
Concatenate all filter outputs together depth-wise

- 22 layers , 9 inception models
- Efficient "Inception" module
- No FC layers
-12x less parameters than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)

Google LeNet

```
class inception_module(nn.Module):

    def __init__(self,in_dim,out_dim_1,mid_dim_3,out_dim_3,mid_dim_5,out_dim_5,pool):
        super(inception_module,self).__init__()

        self.conv_1 = conv_1(in_dim,out_dim_1)
        self.conv_1_3 = conv_1_3(in_dim,mid_dim_3,out_dim_3)
        self.conv_1_5 = conv_1_5(in_dim,mid_dim_5,out_dim_5)
        self.max_3_1 = max_3_1(in_dim,pool)

    def forward(self,x):
        out_1 = self.conv_1(x)
        out_2 = self.conv_1_3(x)
        out_3 = self.conv_1_5(x)
        out_4 = self.max_3_1(x)
        output = torch.cat([out_1,out_2,out_3,out_4],1)

        return output
```

Google LeNet

```
class GoogLeNet(nn.Module):
    def __init__(self, base_dim, num_classes=2):
        super(GoogLeNet, self).__init__()
        self.layer_1 = nn.Sequential(
            nn.Conv2d(3,base_dim,7,2,3),
            nn.MaxPool2d(3,2,1),
            nn.Conv2d(base_dim,base_dim*3,3,1,1),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_2 = nn.Sequential(
            inception_module(base_dim*3,64,96,128,16,32,32),
            inception_module(base_dim*4,128,128,192,32,96,64),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_3 = nn.Sequential(
            inception_module(480,192,96,208,16,48,64),
            inception_module(512,160,112,224,24,64,64),
            inception_module(512,128,128,256,24,64,64),
            inception_module(512,112,144,288,32,64,64),
            inception_module(528,256,160,320,32,128,128),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_4 = nn.Sequential(
            inception_module(832,256,160,320,32,128,128),
            inception_module(832,384,192,384,48,128,128),
            nn.AvgPool2d(7,1),
        )
        self.layer_5 = nn.Dropout2d(0.4)
        self.fc_layer = nn.Linear(1024,1000)
```

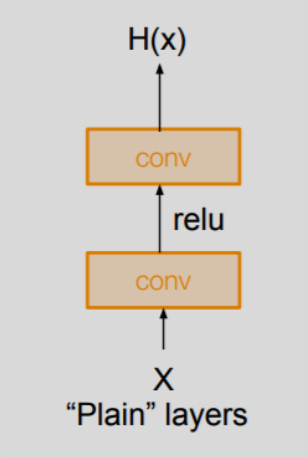
```
def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = out.view(batch_size,-1)
        out = self.fc_layer(out)
        return out
```

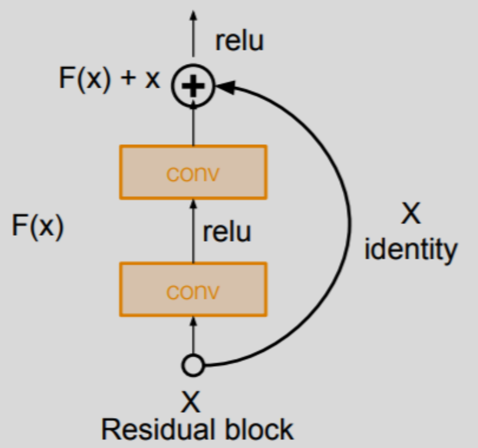
ResNet

20 layers vs 56 layers, training error and test error:

ResNet

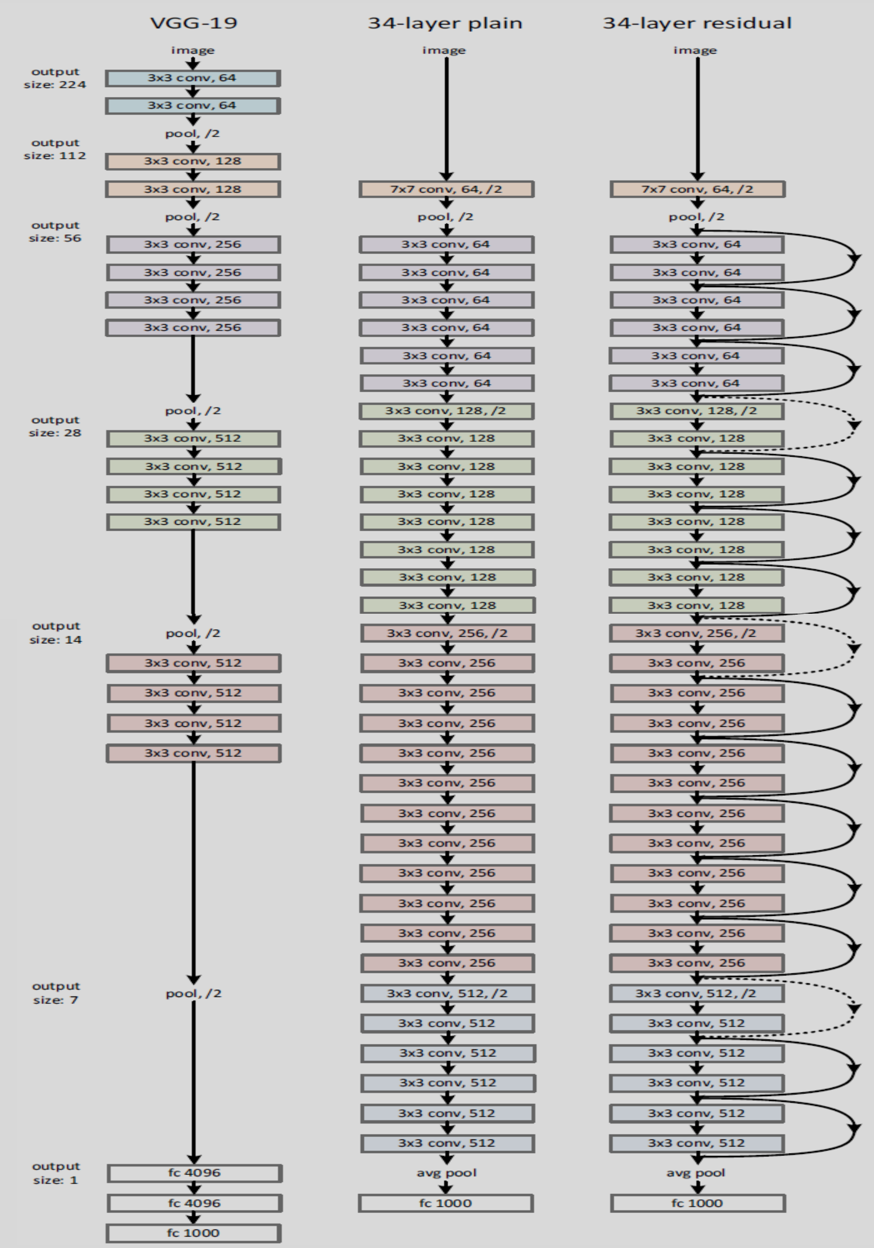
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping





Instead of learning H(x) directly, we ask what do we need to add/subtract in order to get H(x) ?
$H(x) = F(x) + x$

ResNet



Getting deeper without getting less accuracy.

ResNet

```
def conv_block_1(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=1),
        act_fn,
    )
    return model

def conv_block_1_stride_2(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=2),
        act_fn,
    )
    return model

def conv_block_1_n(in_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=1),
    )
    return model

def conv_block_1_stride_2_n(in_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=2),
    )
    return model

def conv_block_3(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=3, stride=1, padding=1),
        act_fn,
    )
    return model
```

ResNet

```
class BottleNeck(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )
        self.downsample = nn.Conv2d(in_dim,out_dim,1,1)

    def forward(self,x):
        downsample = self.downsample(x)
        out = self.layer(x)
        out = out + downsample

        return out


class BottleNeck_no_down(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck_no_down,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )

    def forward(self,x):
        out = self.layer(x)
        out = out + x

        return out
```

```
class BottleNeck_stride(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck_stride,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1_stride_2(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )
        self.downsample = nn.Conv2d(in_dim,out_dim,1,2)

    def forward(self,x):
        downsample = self.downsample(x)
        out = self.layer(x)
        out = out + downsample

        return out
```

ResNet

```
class ResNet(nn.Module):

    def __init__(self, base_dim, num_classes=2):
        super(ResNet, self).__init__()
        self.act_fn = nn.ReLU()
        self.layer_1 = nn.Sequential(
            nn.Conv2d(3,base_dim,7,2,3),
            nn.ReLU(),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_2 = nn.Sequential(
            BottleNeck(base_dim,base_dim,base_dim*4,self.act_fn),
            BottleNeck_no_down(base_dim*4,base_dim,base_dim*4,self.act_fn),
            BottleNeck_stride(base_dim*4,base_dim,base_dim*4,self.act_fn),
        )
        self.layer_3 = nn.Sequential(
            BottleNeck(base_dim*4,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_no_down(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_no_down(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_stride(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
        )
        self.layer_4 = nn.Sequential(
            BottleNeck(base_dim*8,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_stride(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
        )
        self.layer_5 = nn.Sequential(
            BottleNeck(base_dim*16,base_dim*8,base_dim*32,nn.ReLU()),
            BottleNeck_no_down(base_dim*32,base_dim*8,base_dim*32,self.act_fn),
            BottleNeck(base_dim*32,base_dim*8,base_dim*32,self.act_fn),
        )
        self.avgpool = nn.AvgPool2d(7,1)
        self.fc_layer = nn.Linear(base_dim*32,num_classes)
```

```
    def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = self.avgpool(out)
        out = out.view(batch_size,-1)
        out = self.fc_layer(out)

        return out
```

Thank you!