PyTorch

What is PYTORCH?

It's a Python based scientific computing package targeted at two use cases:

- A replacement for NumPy to use the power of GPUs
- A deep learning research platform that provides maximum flexibility and speed

PyTorch: A Python Based Framework

- A complete rewrite of Torch using Python
- With critical functions written in C/C++

PYTORCH + python™

(From GitHub)

PyTorch: A NumPy Replacement with GPU Acceleration

- Provides NumPy like capabilities
- `torch.Tensor` is similar to `numpy.ndarray`
- With support for both CPU* and GPU

Fig: Tensor operations

*NumPy only supports CPU

PyTorch: A Deep Learning Research Platform

- Allows you to solve problems using deep learning
- Caters to use cases in computer vision, text, speech, and so on

Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience

Fig: Packages in PyTorch

Fig: Object instance segmentation, a user case in computer vision

Salient Features

- Python first
- Use of Dynamic Computation Graph
- Intuitive programming model
- Easier debugging

Salient Features: Python First

- Deeply integrated with Python
- PyTorch computations run within the Python computation model
- It's imperative, just like Python
- PyTorch can be extended, just like you would extend Python

Salient Features: Use of Dynamic Computation Graph (1/2)

- PyTorch uses a Dynamic computation graph (versus Static)
- PyTorch: Computation graph gets created on the fly

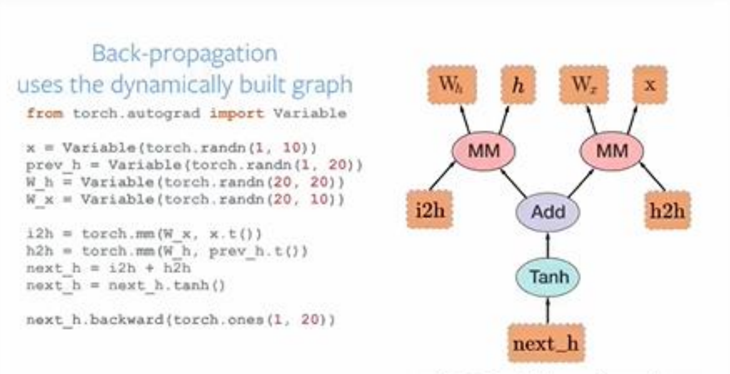


Fig: PyTorch dynamic graph

Salient Features: Use of Dynamic Computation Graph (2/2)

- PyTorch uses a Dynamic computation graph (versus Static)
- **Dynamic** computation graphs use the **imperative** style of programming
- **Static** computation graphs use the **declarative** style of programming

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
```

C, declarative

```
>>> print("Hello World")
Hello World
>>>
```

Python, imperative

Salient Features: Intuitive Programming Model

- Define and run
- Linear in thought
- Faster prototyping

Salient Features: Easier Debugging

- No separate virtual execution environment
- Uses host language’s runtime
- Debugging Pytorch code is just like debugging Python code
- Use the same debugging tools

PyTorch: Platforms Supported

Refresher: Scalar, Vector, Matrix, Tensor, Rank, and Dimension

Scalar	Vector	Matrix	Tensor
3, 1.2	[3, 4, 8]	[[1, 2, 3], [4, 5, 6]]	[[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]
Rank: 0 (always)	Rank: 1 (always)	Rank: 2 (always)	Rank: 3
Dim: ()	Dim: (3,)	Dim: (2, 3)	Dim: (2, 2, 3)

Tensors in PyTorch

- A multi-dimensional matrix containing elements of a single data type
- Part of the `torch` package
- Instantiated via `torch.Tensor` group of classes
- Can be stored on the CPU or GPU
- Operated on via functions available in the `torch` package, or via class methods
- Interoperable with NumPy `array`

```
>>> torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
1  2  3
4  5  6
[torch.FloatTensor of size 2x3]
```

Tensors in PyTorch

Data type	CPU tensor	GPU tensor
32-bit floating point	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

Fig: Tensor types

Refresher: Differentiation

If $y = \underset{\text{Model}}{f}(x) = 2\underset{\text{Model parameter}}{x}$

then $\frac{dy}{dx} = 2$

If $y = f(x1, x2, .., xn)$

then $[\frac{dy}{dx1}, \frac{dy}{dx2}, .., \frac{dy}{dxn}]$

Is the gradient of y w.r.t. $[x1, x2, .., xn]$

Function $f(x)$	Derivative $f'(x)$
a	0
x	1
$ax+b$	a
$\frac{1}{x}$	$-\frac{1}{x^2}$
x^a	ax^{a-1}
a^x	$a^x \ln a$
$\sqrt[x]{a}$	$-\frac{\sqrt[x]{a}\ln a}{x^2}$
$\log_a x$	$\frac{1}{x \ln a}$
x^x	$x^x(1+\ln x)$
$\Gamma(x)$	$\Gamma(x)\psi(x)$

Refresher: The Computation Graph

$c = a + b$

$d = b + 1$

$e = c * d$

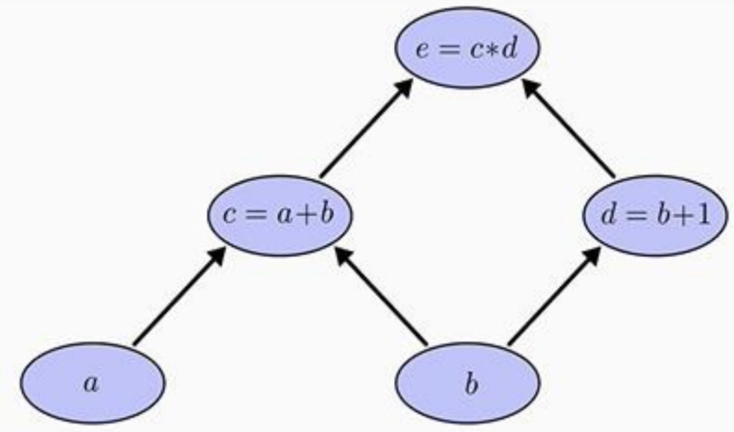


Fig: A computation graph

Refresher: The Computation Graph

Back-propagation
uses the dynamically built graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

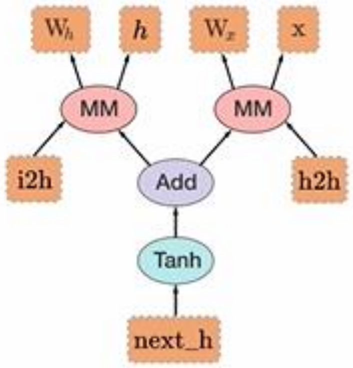
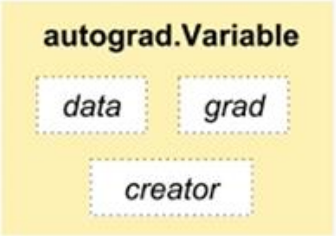


Fig: PyTorch dynamic graph

Variables in PyTorch

- Crucial data structure, needed for automatic differentiation
- Part of the `torch.autograd` package
- Instantiated via `torch.autograd.Variable` class
- A wrapper around a `tensor` object (the data)
- Holds the gradient w.r.t. it (the grad)
- Records reference to the function that created it (the creator)
- Holds the gradient of output w.r.t this tensor



Interoperability Between PyTorch Tensors and NumPy Arrays

- `torch.Tensor` similar to `numpy.ndarray`
- 200+ operations, similar to numpy
- Memory pointer shared between PyTorch tensor and NumPy array
- Conversions very fast: Zero memory copy

Numpy

```python
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch

```python
import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Handling Datasets in PyTorch

Concepts: Dataset, Epoch, Batch, Iteration

- Dataset: A collection of training examples
- Epoch: One pass of the entire dataset through your model during training
- Batch: A subset of training examples passed through your model at a time
- Iteration: An iteration is a single pass of a batch
- Here, a pass would involve a forward and a backward propagation

Accessing Custom Vision Datasets

- Create a custom `Dataset` class which inherits from `torch.utils.data.Dataset`
- Use `torch.utils.data.DataLoader` to iterate through the data
- Tutorial: http://pytorch.org/tutorials/beginner/data_loading_tutorial.html

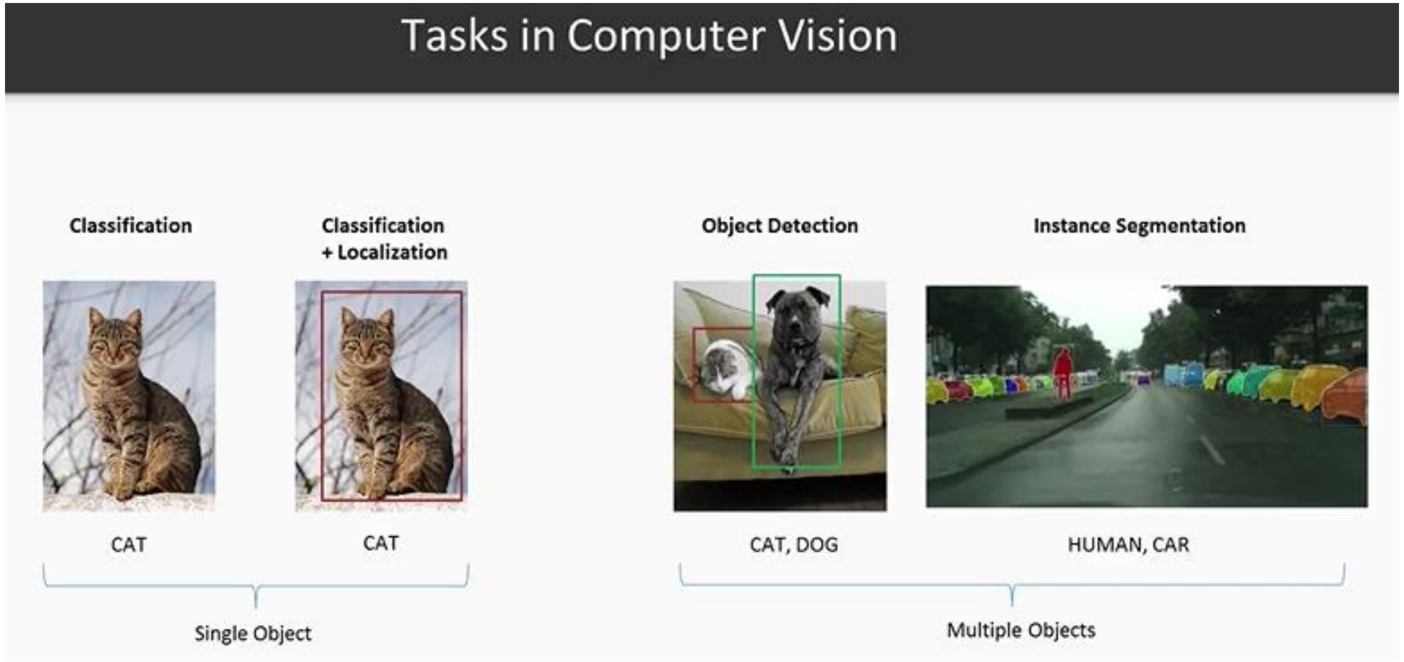
For a dataset of 1000 images

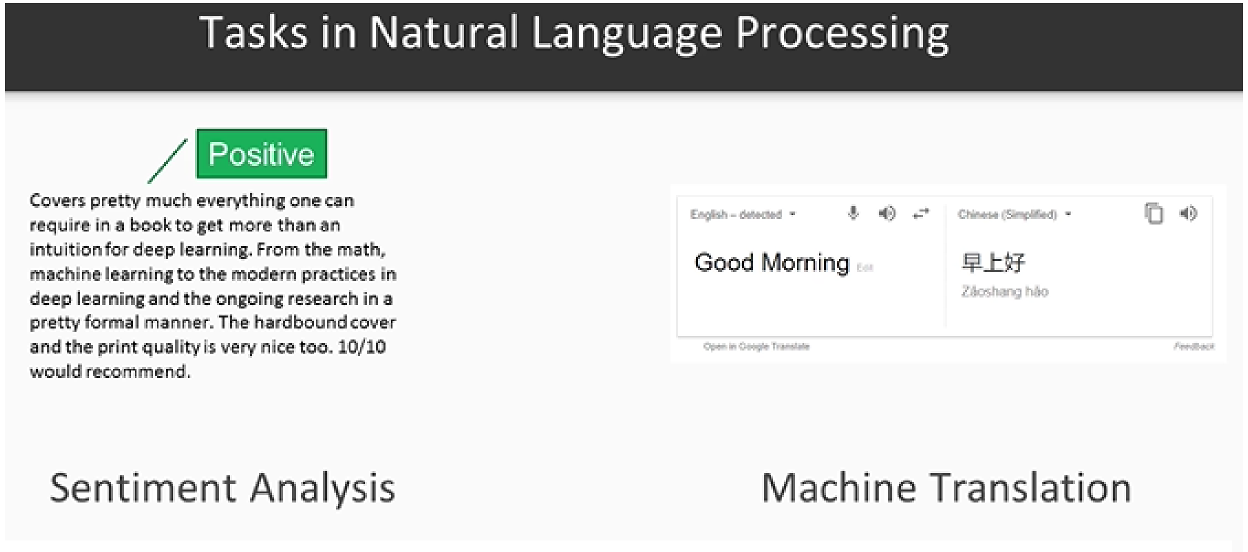
1 Epoch

With Batch Size of 50

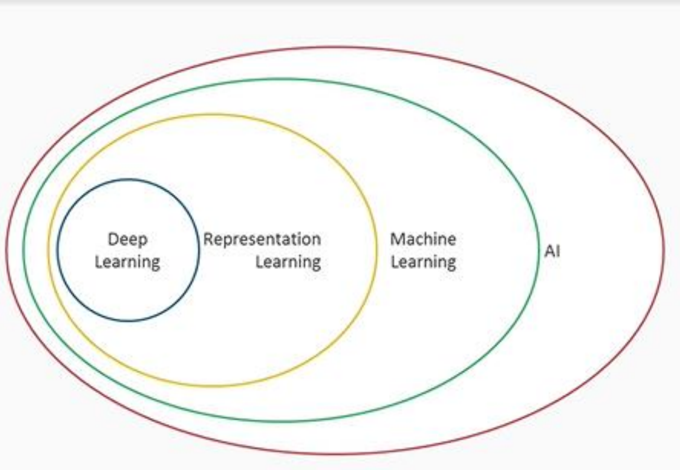
Leads to 20 Iterations

Deep Learning using PyTorch





What is Deep Learning?



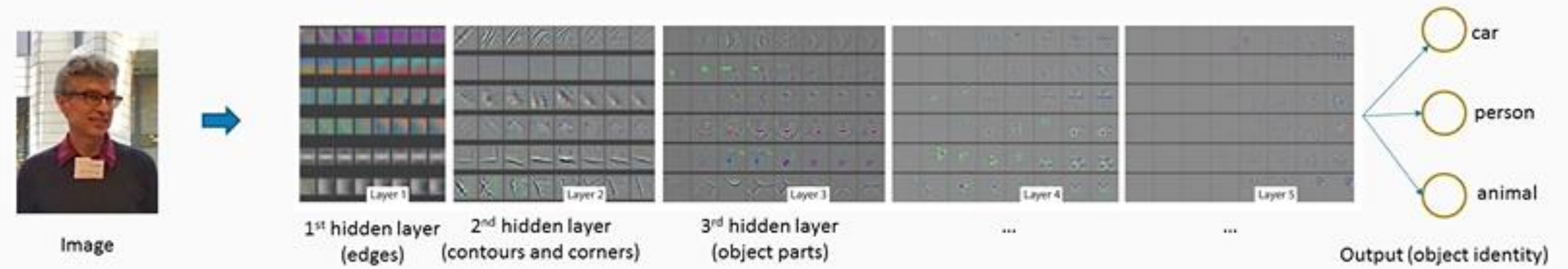


Fig: Hiererchy of information in a convolutional neural network

PyTorch Packages for Deep Learning

Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torch.legacy(.nn/.optim)	legacy code that has been ported over from torch for backward compatibility reasons

Train Your First Neural Network

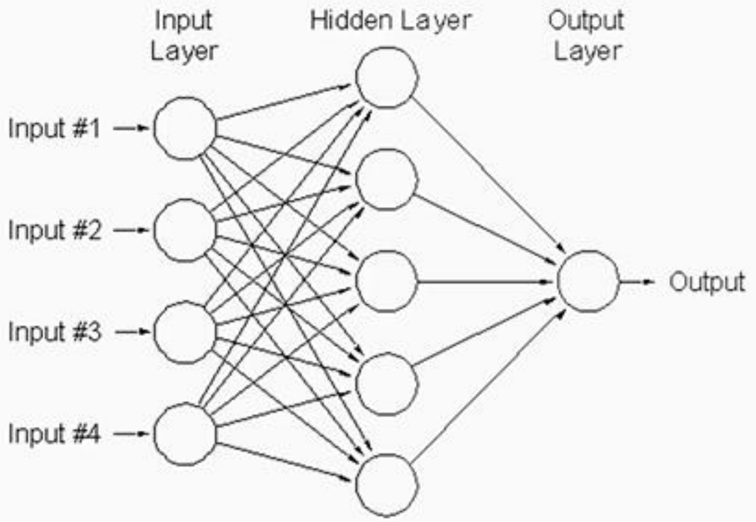
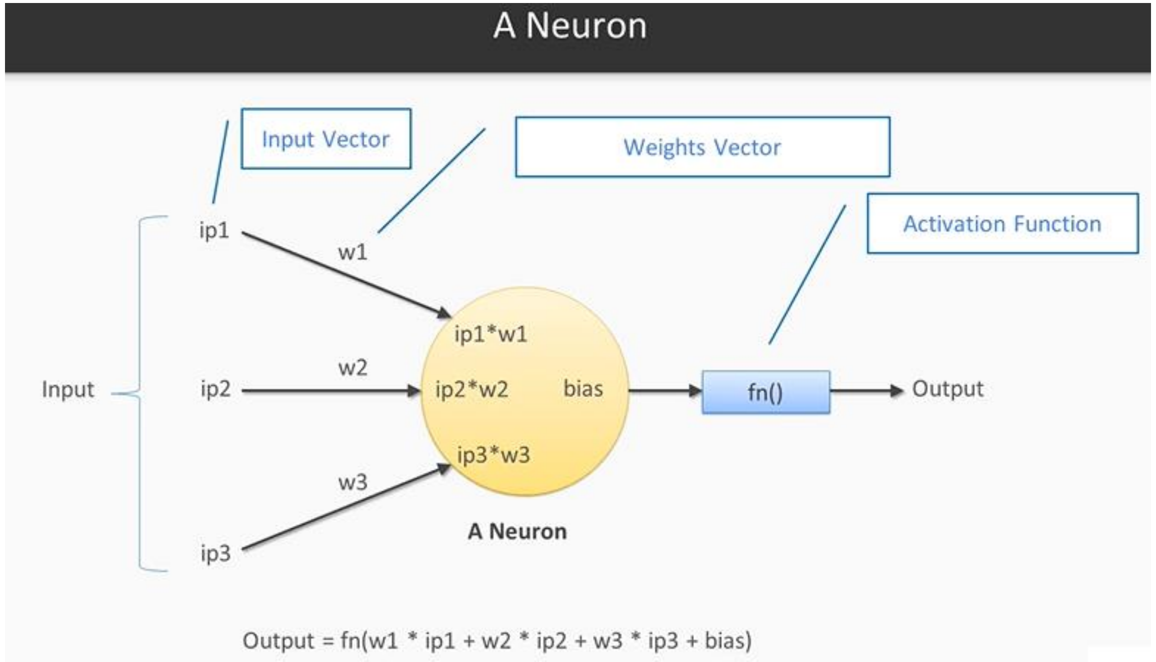
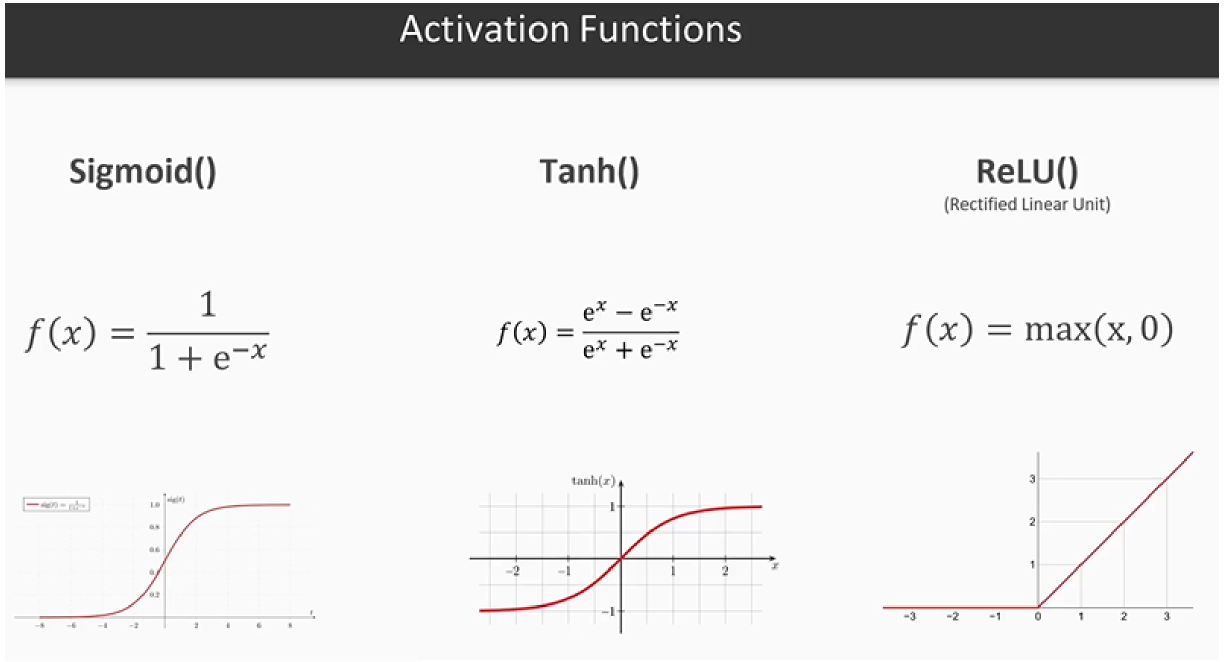


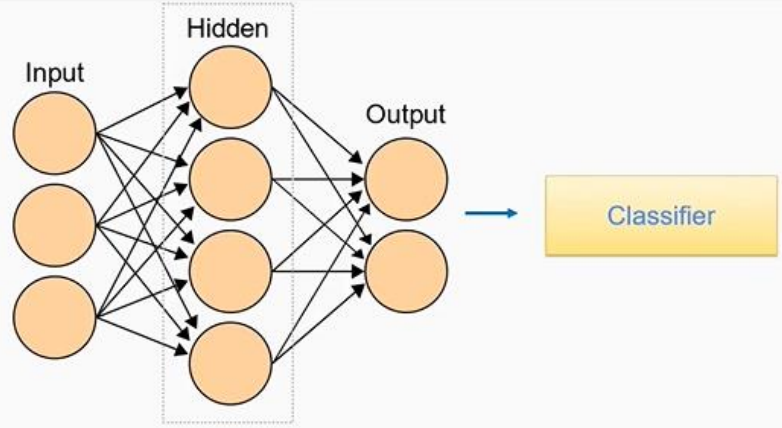
Fig: A simple neural network

Training a Neural Network with PyTorch



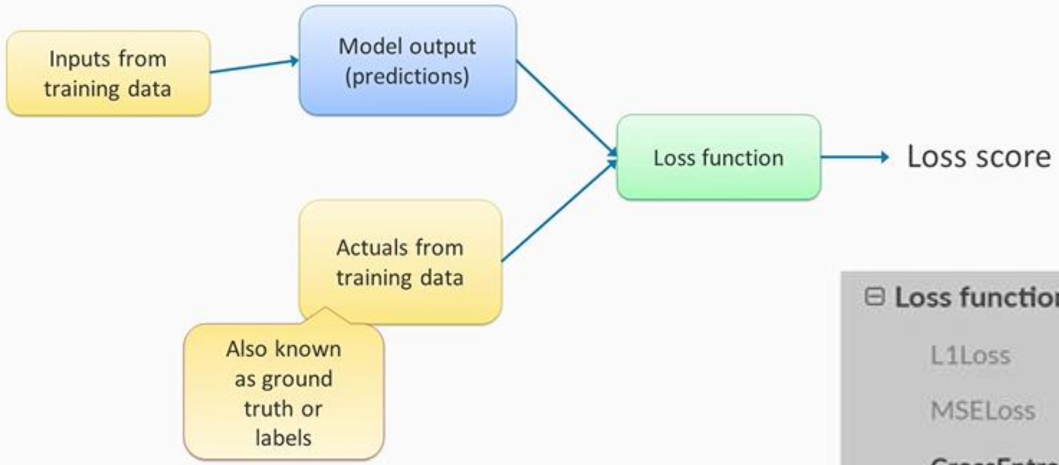


Artificial Neural Network



Hidden Layer: 3 incoming connections, 4 outgoing connections

Loss Function



⊟ **Loss functions**

- L1Loss
- MSELoss
- **CrossEntropyLoss**
- NLLLoss
- PoissonNLLLoss
- NLLLoss2d
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss

Cross Entropy Loss

For a training example:

Class:	0 (Apple)	1 (Mango)	2 (Banana)
Prediction:	0.02	0.88	0.1

Class: 1

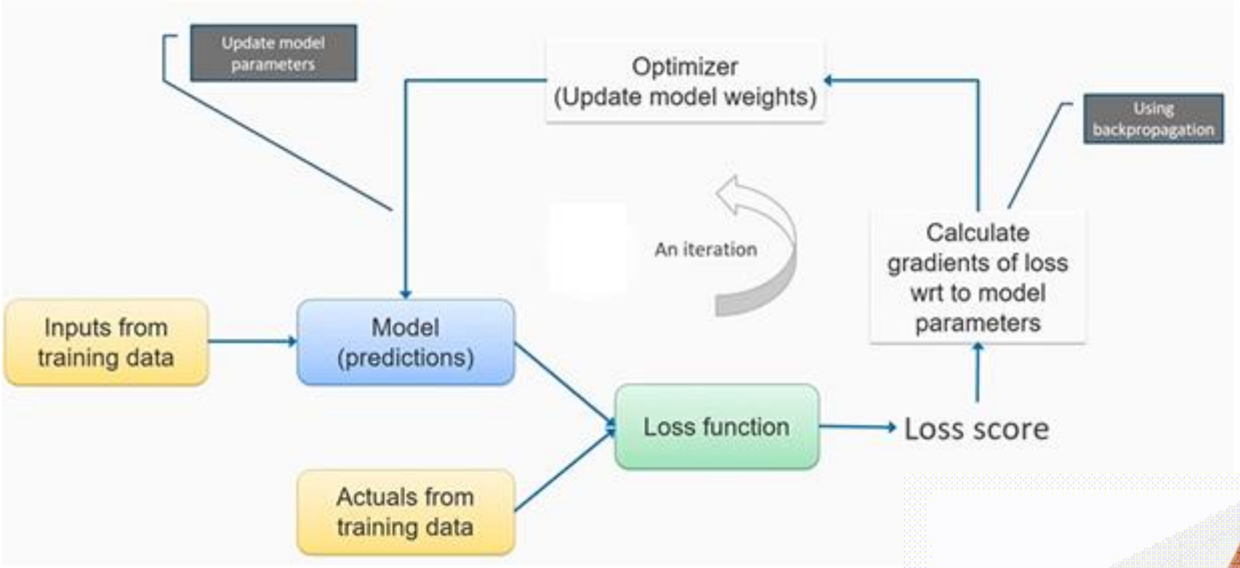
Model Predictions

Actual/Expected

Low loss score

Optimization Problem

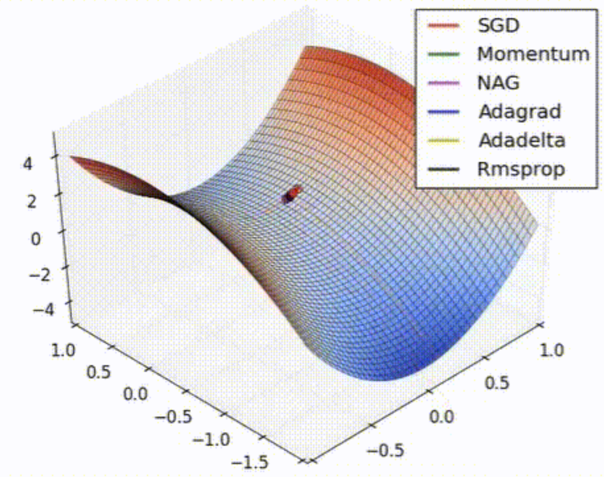
- An optimization problem is the problem of finding the best solution from all feasible solutions, given some criteria – Wikipedia
- Example: Which line to stand in at the supermarket

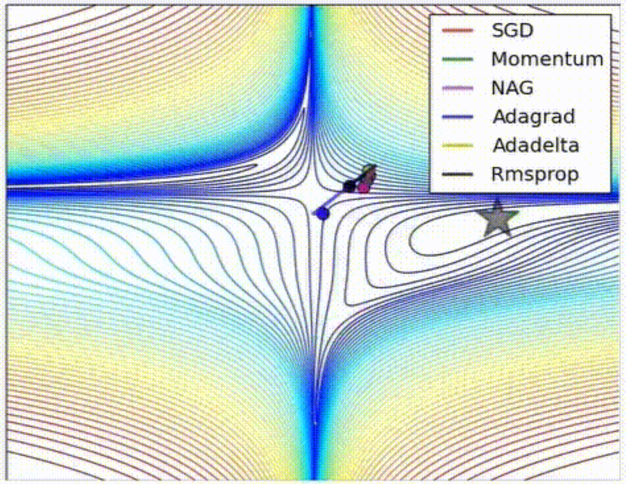


Available Optimizer

- SGD
- Adadelta
- Adagrad
- Adam
- RMSprop

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

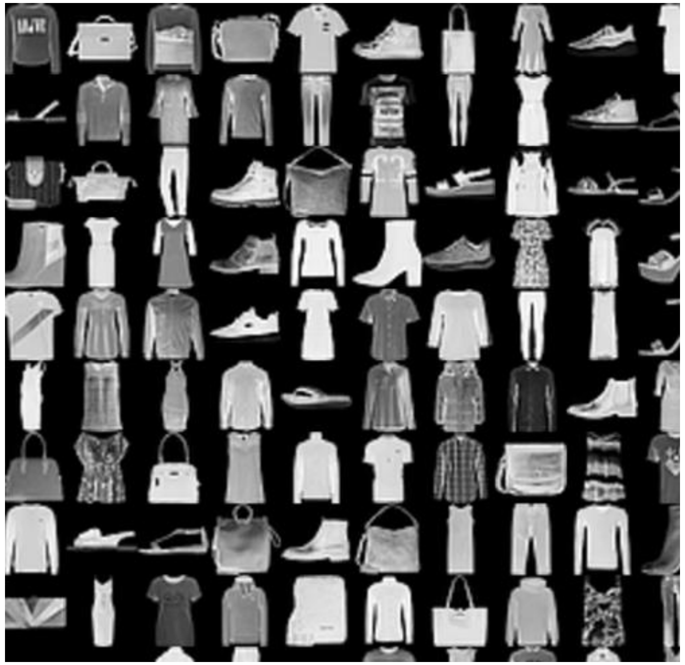




The Fashion MNIST Data Set

- Is made up of Zalando's images
- Contains 60,000 images
- Also has a test set of 10,000 images

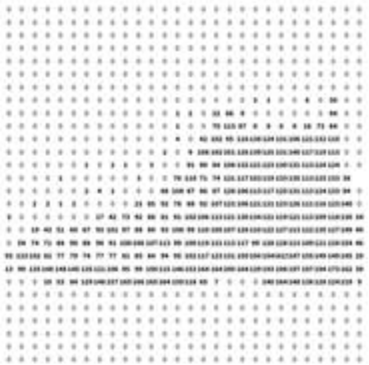
- Images 28 x 28 pixels
- Items fall into 10 categories

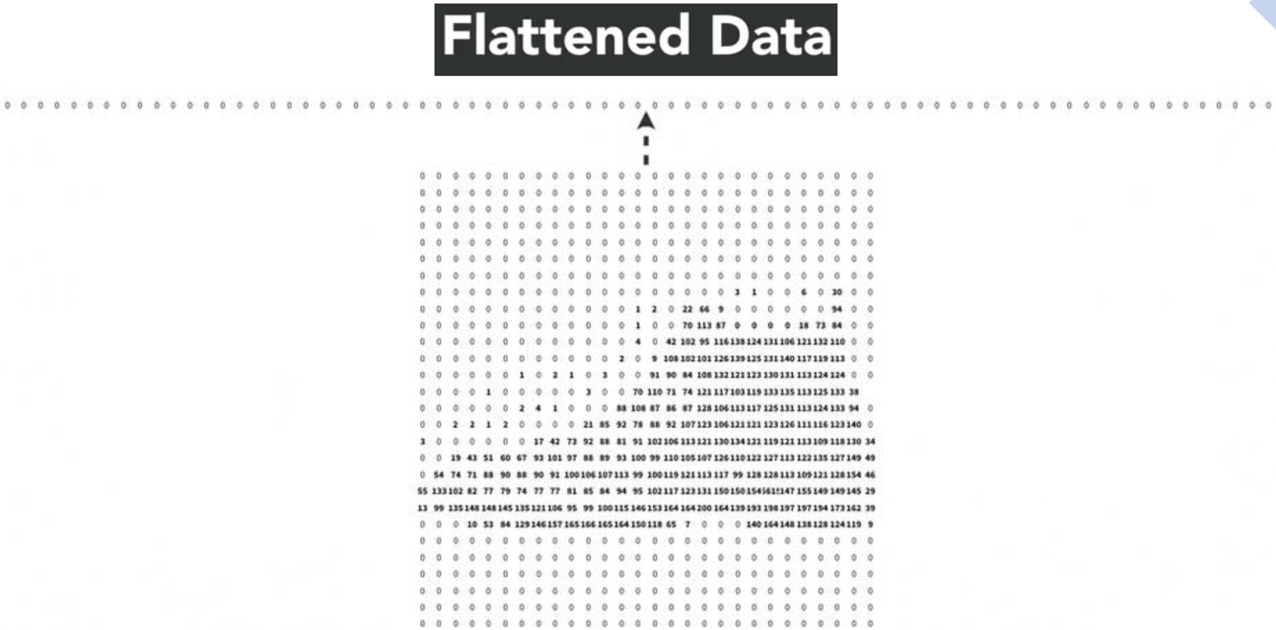


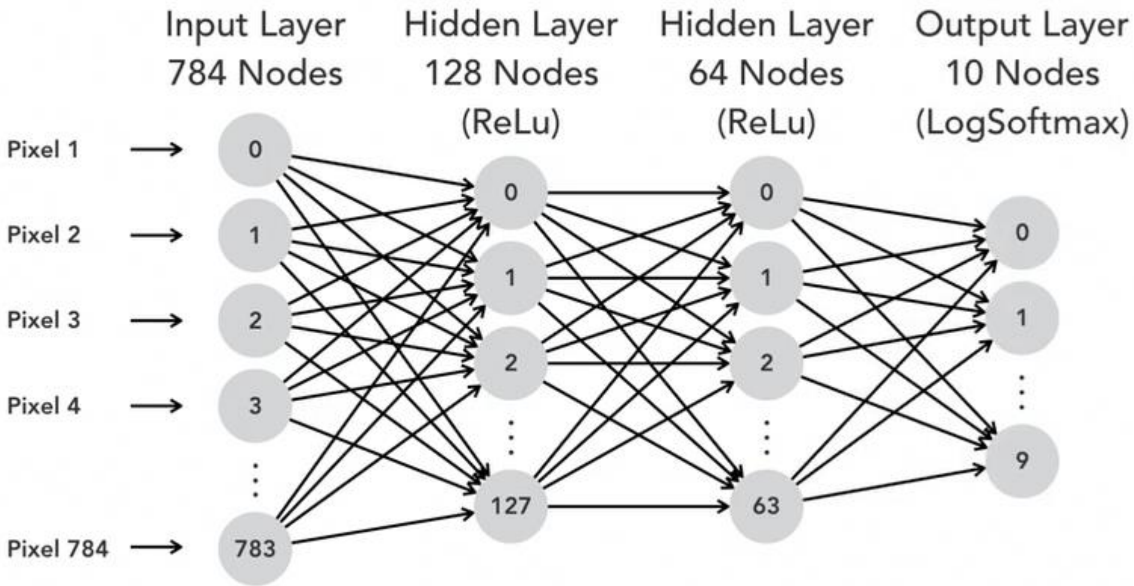
Labels

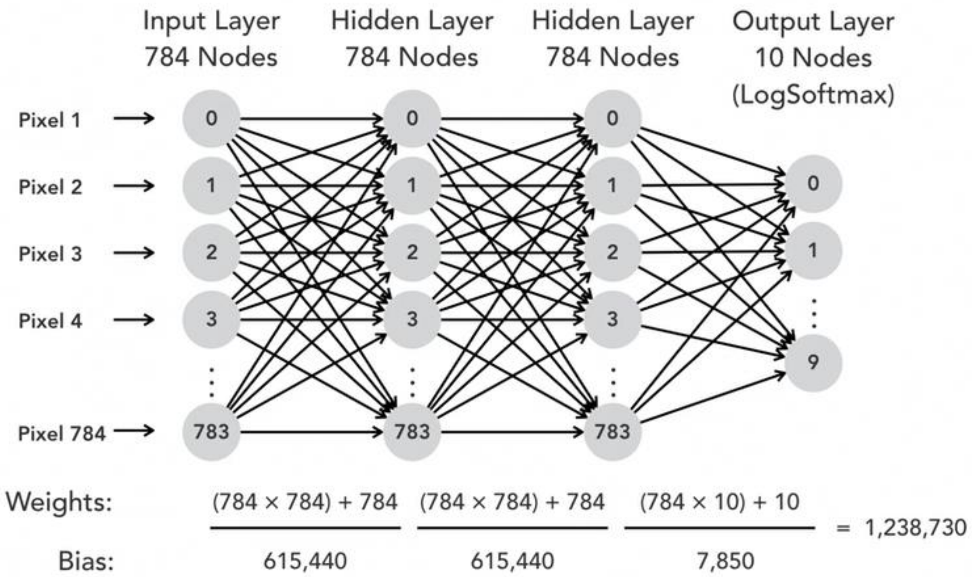
Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat

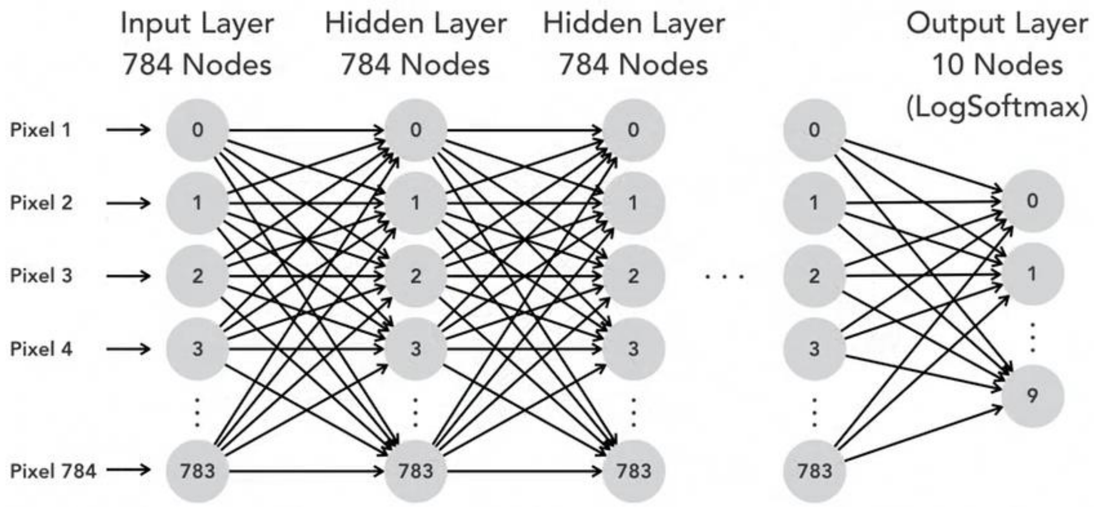
Label	Description
5	Sandal
6	Shirt
7	Sport shoe
8	Bag
9	Ankle boot



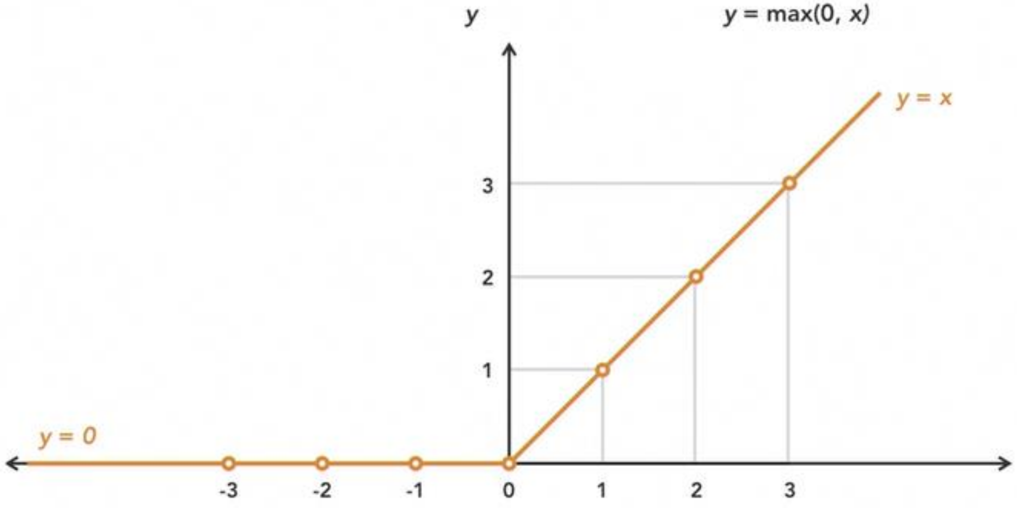

Flattened Data


Input Layer
784 Nodes
Hidden Layer
128 Nodes
(ReLu)
Hidden Layer
64 Nodes
(ReLu)
Output Layer
10 Nodes
(LogSoftmax)
Pixel 1
Pixel 2
Pixel 3
Pixel 4
Pixel 784
0
1
2
3
783
0
1
2
127
0
1
2
63
0
1
9


Input Layer
784 Nodes
Hidden Layer
784 Nodes
Hidden Layer
784 Nodes
Output Layer
10 Nodes
(LogSoftmax)
Pixel 1
Pixel 2
Pixel 3
Pixel 4
Pixel 784
0
1
2
3
783
0
1
9
Weights:
Bias:
(784 × 784) + 784
615,440
(784 × 784) + 784
615,440
(784 × 10) + 10
7,850
= 1,238,730



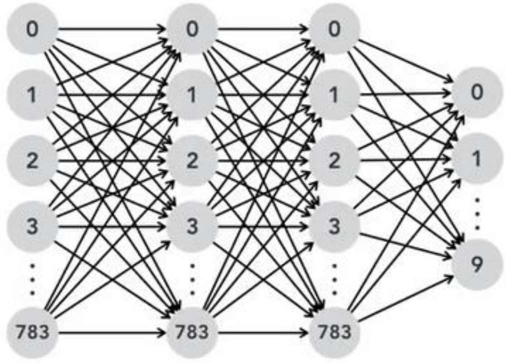
RELU



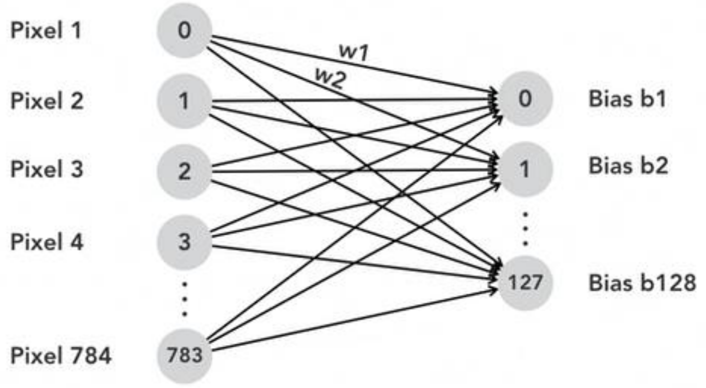
Plot of ankle boots, sports shoes & sandals

Classes Overview

```
class FMNIST(nn.Module):
 def __init__(self):
  super().__init__()
  self.fc1 = nn.Linear(784, 128)
  self.fc2 = nn.Linear(128,64)
  self.fc3 = nn.Linear(64,10)

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

  return x

model = FMNIST()
```



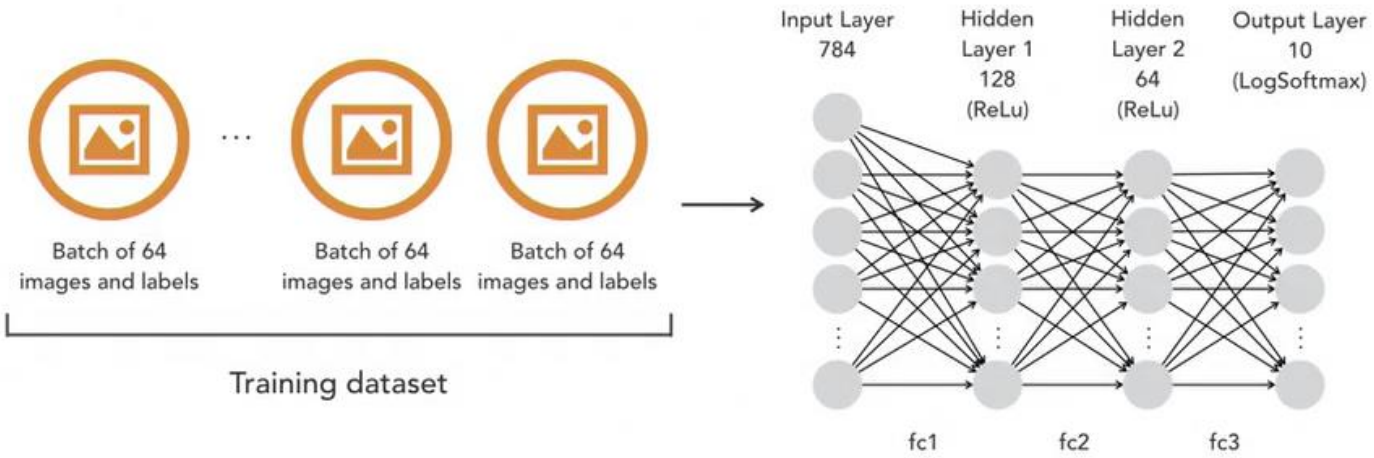
nn.Linear()



forward()

Defines the model structure, components, and order of the different layers

Training the Network

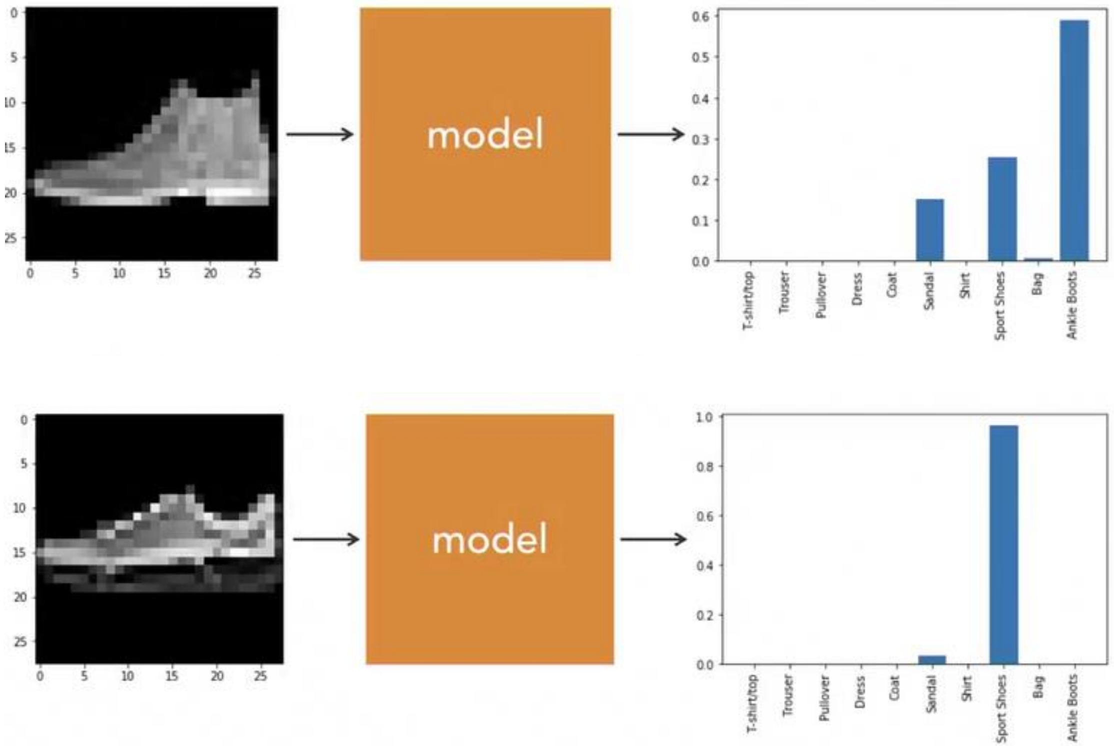


Training Steps

- Take a batch of images and targets
- Forward pass
- Calculate loss of network on batch
- Update weights of the neural network

```
from torch import optim

criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

num_epochs = 3

for i in range(num_epochs):
    cum_loss = 0

    for images, labels in trainloader:
        optimizer.zero_grad()
        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        cum_loss += loss.item()

    print(f"Training loss: {cum_loss/len(trainloader)}")
```

Loss



Loss Function for Problem Types

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	Binary crossentropy
Multiclass, single-label classification	softmax	Categorical crossentropy
Multiclass, multilabel classification	sigmoid	Binary crossentropy
Regression to arbitrary values	None	MSE (mean squared error)
Regression to values between 0 and 1	sigmoid	MSE or binary crossentropy

CrossEntropyLoss

"This criterion combines nn.LogSoftmax() and nn.NLLLoss() in one single class."

CROSSENTROPYLOSS

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `LogSoftmax` and `NLLLoss` in one single class.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

The losses are averaged across observations for each minibatch. If the `weight` argument is specified then this is a weighted average:

$$\text{loss} = \frac{\sum_{i=1}^{N} loss(i, class[i])}{\sum_{i=1}^{N} weight[class[i]]}$$

Softmax

CLASS `torch.nn.Softmax(dim=None)` [SOURCE]

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Returns

a Tensor of the same dimension and shape as the input with values in the range [0, 1]

Parameters

dim (*int*) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

• NOTE

This module doesn't work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use *LogSoftmax* instead (it's faster and has better numerical properties).

Examples:

```
>>> m = nn.Softmax(dim=1)
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

Softmax2d

CLASS `torch.nn.Softmax2d` [SOURCE]

Applies SoftMax over features to each spatial location.

When given an image of `Channels x Height x Width`, it will apply *Softmax* to each location $(Channels, h_i, w_j)$

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Softmax

"Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1."

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

LogSoftmax

CLASS `torch.nn.LogSoftmax(dim=None)` [SOURCE]

Applies the $\log(\text{Softmax}(x))$ function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as:

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Parameters

dim (*int*) – A dimension along which LogSoftmax will be computed.

Returns

a Tensor of the same dimension and shape as the input with values in the range [-inf, 0)

```
[docs]def log_softmax(input, dim=None, _stacklevel=3, dtype=None):
    # type: (Tensor, Optional[int], int, Optional[int]) -> Tensor
    r"""Applies a softmax followed by a logarithm.

    While mathematically equivalent to log(softmax(x)), doing these two
    operations separately is slower, and numerically unstable. This function
    uses an alternative formulation to compute the output and gradient correctly.

    See :class:`~torch.nn.LogSoftmax` for more details.

    Arguments:
        input (Tensor): input
        dim (int): A dimension along which log_softmax will be computed.
        dtype (:class:`torch.dtype`, optional): the desired data type of returned tensor.
          If specified, the input tensor is casted to :attr:`dtype` before the operation
          is performed. This is useful for preventing data type overflows. Default: None.
    """
    if dim is None:
        dim = _get_softmax_dim('log_softmax', input.dim(), _stacklevel)
    if dtype is None:
        ret = input.log_softmax(dim)
    else:
        ret = input.log_softmax(dim, dtype=dtype)
    return ret


softshrink = _add_docstr(torch._C._nn.softshrink, r"""
softshrink(input, lambd=0.5) -> Tensor

Applies the soft shrinkage function elementwise

See :class:`~torch.nn.Softshrink` for more details.
```

LogSoftmax

"Applies the log(Softmax(x)) function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as:"

$$LogSoftmax(x_i) = log(\frac{exp(x_i)}{\sum_j exp(x_j)})$$

NLLLoss (Negative Log Likelihood Loss)

```python
from torch import optim

criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

num_epochs = 3

for i in range(num_epochs):
    cum_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)

        optimizer.zero_grad()

        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        cum_loss += loss.item()

    print(f"Training loss: {cum_loss/len(trainloader)}")
```

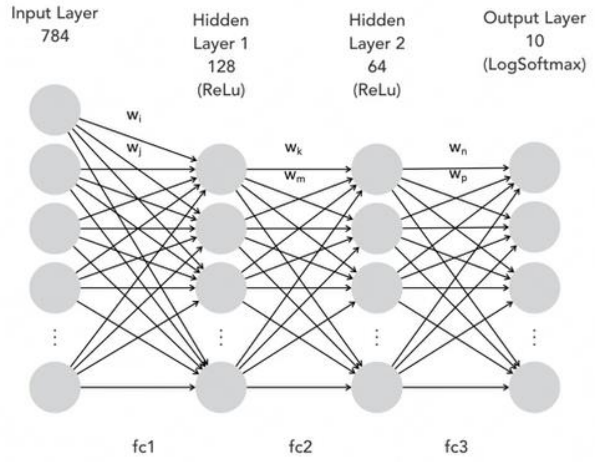
```python
class FMNIST(nn.Module):
  def __init__(self):
    super().__init__()
    self.fc1 = nn.Linear(784, 784)
    self.fc2 = nn.Linear(784,784)
    self.fc3 = nn.Linear(784,10)

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

    return x

model = FMNIST()
```

CrossEntropyLoss

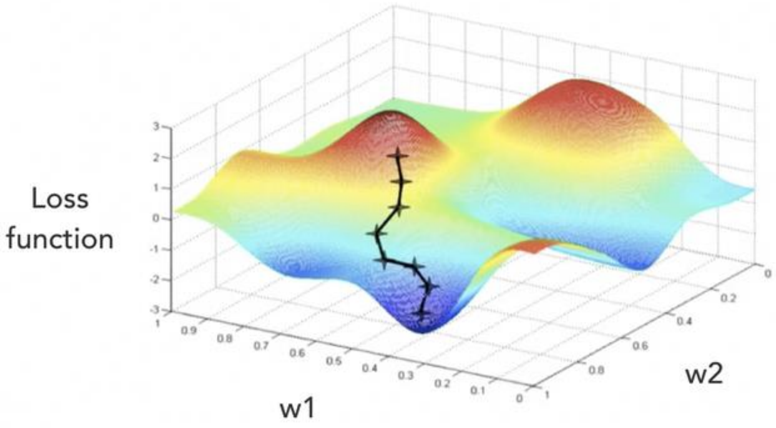
```python
from torch import optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

num_epochs = 3

for i in range(num_epochs):
    cum_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)

        optimizer.zero_grad()

        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        cum_loss += loss.item()

    print(f"Training loss: {cum_loss/len(trainloader)}")
```

```python
class FMNIST(nn.Module):
  def __init__(self):
    super().__init__()
    self.fc1 = nn.Linear(784, 784)
    self.fc2 = nn.Linear(784,784)
    self.fc3 = nn.Linear(784,10)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)

    return x

model = FMNIST()
```

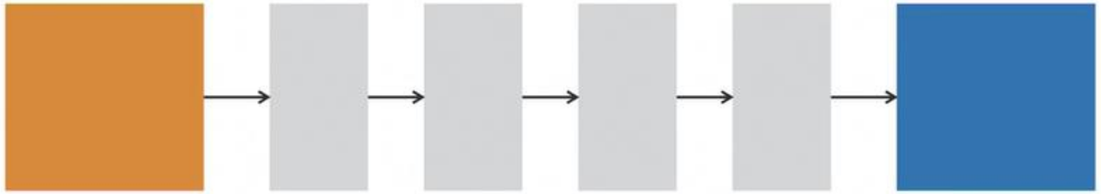
Our goal is to try and determine some way to adjust the weights so that the loss is minimized.



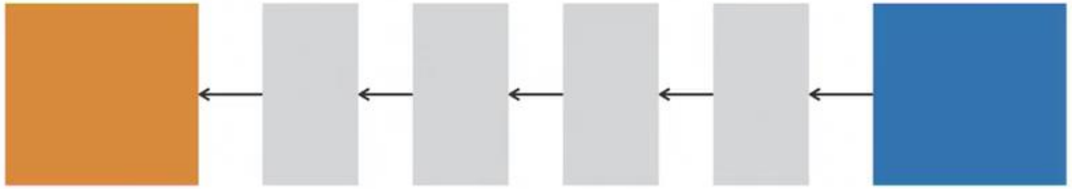
Autograd



Forward Pass



Backward Pass



Why Do You Need an Optimizer?

- The loss function determines the quantity that will be minimized during training
- The optimizer determines how the network will be updated based on the loss function; it implements a variant of stochastic gradient descent (SGD)

Training Steps

Take a batch of samples and targets

Forward pass

Calculate loss

Backward pass

Update weights of the parameter

- w = w – learning rate × [gradient of loss w.r.t. w]

Zero Out Gradients for Each Epoch

```
optimizer.zero_grad()
```

Debugging

h (help) [*command*]	print help about *command*
n (next)	execute current line of code, go to next line
c (continue)	continue executing the program until next breakpoint, exception, or end of the program
s (step into)	execute current line of code; if a function is called, follow execution inside the function
l (list)	print code around the current line
w (where)	show a trace of the function call that led to the current line
p (print)	print the value of a variable
q (quit)	leave the debugger
b (break) [*lineno* | *function* [, *condition*]]	set a breakpoint at a given line number or function, stop execution there if *condition* is fulfilled
cl (clear)	clear a breakpoint
! (execute)	execute a python command
<enter>	repeat last command

CPU vs. GPU

- CPU – general purpose computing
- GPU – intensive computational calculations

Moving from CPU to GPU

1. At the top of the notebook, specify the CUDA device.
2. Transfer neural network to the GPU.
3. Send all inputs and targets to the GPU.