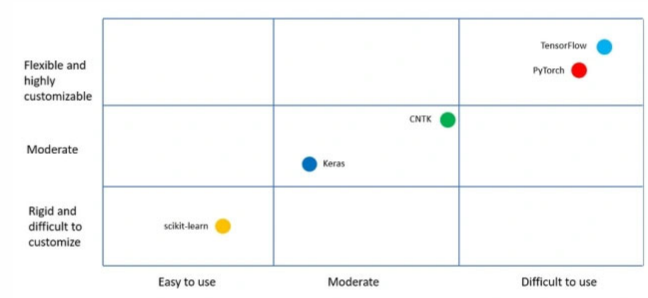
COMP9444: Neural Networks and Deep Learning

Week 2c. PyTorch

Raymond Louie
School of Computer Science and Engineering

Feb 26, 2024

Comparison 1



https://jamesmccaffrey.wordpress.com/2019/08/22/a-subjective-comparison-of-tensorflow-pytorch-keras-and-scikit-learn/

Comparison 2

	Keras	TensorFlow	PyTorch
Level of API	high-level API[1]	Both high & low level APIs	Lower-level API[2]
Speed	Slow	High	High
Architecture	Simple, more readable and concise	Not very easy to use	Complex[3]
Debugging	No need to debug	Difficult to debugging	Good debugging capabilities
Dataset Compatibility	Slow & Small	Fast speed & large	Fast speed & large datasets
Popularity Rank	1	2	3
Uniqueness	Multiple back-end support	Object Detection Functionality	Flexibility & Short Training Duration
Created By	Not a library on its own	Created by Google	Created by Facebook[4]
Ease of use	User-friendly	Incomprehensive API	Integrated with Python language
Computational graphs used	Static graphs	Static graphs	Dynamic computation graphs[5]

 https://medium.com/analytics-vidhya/ml03-9de2f0dbd62d

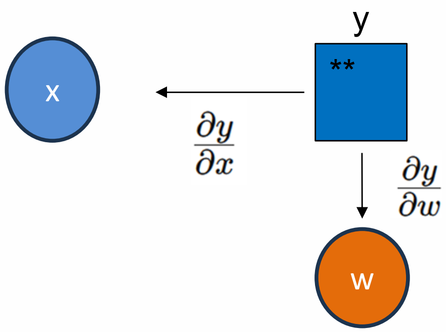
Comparison 3

Aspect	Keras	TensorFlow	PyTorch
Best For	Beginners, rapid prototyping	Production environments, enterprise apps	Research, dynamic models, academic use
Ease of Use	High (Simple and intuitive)	Moderate (Can be complex)	High (Pythonic and intuitive)
Flexibility	Low (Limited control)	High (Full control over architecture)	High (Dynamic graph for flexibility)
Performance	Moderate (High-level abstraction)	High (Optimized for large-scale models)	High (Efficient, especially for research)
Ecosystem	Strong (Through TensorFlow integration)	Very Strong (Production-ready ecosystem)	Growing (Improving deployment features)
Deployment	Moderate (Through TensorFlow)	Excellent (Highly scalable, many options)	Improving (TorchServe, but less mature)
Popularity	High (Part of TensorFlow)	Very High (Widely used in the industry)	Very High (Popular in research)

Keras vs. TensorFlow vs. PyTorch
Which ML Framework is Right for You?

 https://medium.com/@avijitsur_8327/keras-vs-tensorflow-vs-pytorch-which-ml-framework-is-right-for-you-c44299e79a8b

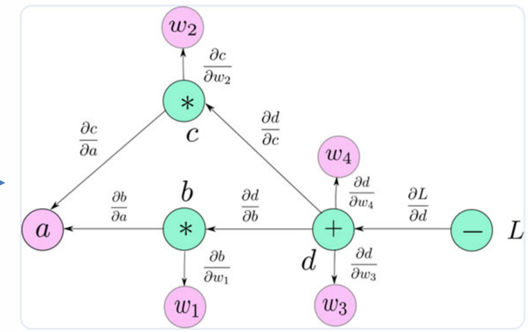
Computational graph and auto-differentiation (COMP9444_demo0_autoDiff)



y = w*x^2

Computational graph and auto-differentiation example

$$b = a \cdot w_1,\quad c = a \cdot w_2,\quad d = w_3 \cdot b + w_4 \cdot c,\quad L = -d.$$



https://www.digitalocean.com/community/tutorials/pytorch-101-understanding-graphs-and-automatic-differentiation

Computational Graphs

- PyTorch automatically builds a computational graph, enabling it to backpropagate derivatives.
- Every parameter includes `.data` and `.grad` components, for example:

A.data
A.grad

- If we need to stop the gradients from being backpropagated through a certain variable (or expression) A, we can exclude it from the computational graph by using:

A.detach()

Typical Structure of a PyTorch Program

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

```
# 0. Import packages
import torch
```


Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

```
# 1. Create model
class MyModel(torch.nn.Module):
    def __init__(self):   (a)
        super(MyModel, self).__init__() (b)                                    (c)
        self.weight = torch.nn.Parameter(torch.zeros((1), requires_grad=True))
    def forward(self, input): (d)
        output  = self.weight * input  (e)
        return(output)

model = MyModel()  (f)
```

a) Initializes MyModel when first created (constructor)
b) Inherits functionality from PyTorch's base model class
c) Define and initializes the parameter "weight"
d) Forward pass – Define function on how the model processes input
e) Output – define the actual processing
f) Create an instance of MyModel

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

```
# 2. Prepare data

input_train   = [[1]]           (a)
output_train  = [[2.343434]]

# Convert to tensors
input_train_tensor  = torch.tensor(input_train, dtype=torch.float32)  (b)
output_train_tensor = torch.tensor(output_train, dtype=torch.float32)

# Load training
batch_size=1                                                              (c)
train_dataset = torch.utils.data.TensorDataset(input_train_tensor,output_train_tensor)
train_loader  = torch.utils.data.DataLoader(train_dataset,batch_size=batch_size) (d)
```

a) Set input and output data (manually in this case)
b) Convert to tensors, required as input in later functions
c) Wraps input and output tensors to form a dataset – allows for easy iteration over (input, output) pairs during training
d) Create DataLoader object which allows loading of data batches efficiently, iteration over dataset, and others

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
↓
1. Create model
↓
2. Prepare data
↓
3. Choose optimizer
↓
4. Train model
↓
5. Test model

```
# 3. Choose optimizer

#  Neural network parameters
learning_rate = 1.9 # Learning rate

# Optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```


Choosing an Optimizer

Adadelta	Implements Adadelta algorithm.
Adafactor	Implements Adafactor algorithm.
Adagrad	Implements Adagrad algorithm.
Adam	Implements Adam algorithm.
AdamW	Implements AdamW algorithm.
SparseAdam	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm).
ASGD	Implements Averaged Stochastic Gradient Descent.
LBFGS	Implements L-BFGS algorithm.
NAdam	Implements NAdam algorithm.
RAdam	Implements RAdam algorithm.
RMSprop	Implements RMSprop algorithm.
Rprop	Implements the resilient backpropagation algorithm.
SGD	Implements stochastic gradient descent (optionally with momentum).

https://pytorch.org/docs/stable/optim.html

SGD optimizer

CLASS torch.optim.SGD(params, lr=0.001, momentum=0, dampening=0, weight_decay=0, nesterov=False, *, maximize=False, foreach=None, differentiable=False, fused=None) [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$
return θ_t

SGD stands for "Stochastic Gradient Descent"

optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=0.0001)

Adam optimizer

CLASS torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, *, foreach=None, maximize=False, capturable=False, differentiable=False, fused=None) [SOURCE]

Implements Adam algorithm.

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), *amsgrad*, *maximize*, ϵ (epsilon)
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** $\ldots$ **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
  $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_{t-1}}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon)$
return θ_t

Adam = Adaptive Moment Estimation (good for deep networks)

optimizer = torch.optim.Adam(net.parameters(),eps =0.000001, lr=0.01, betas=(0.5,0.999), weight_decay=0.0001)

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

```
# 4. Train model

epochs = 100

for epoch in range(1, epochs):

    for batch_id, (data,target) in enumerate(train_loader):          (a)
        optimizer.zero_grad() # zero the gradients         (b)
        output = model(data)    # apply network (Same as model.forward(data)) (c)
        loss = 0.5*torch.mean((output-target)*(output-target))  (d)

        print('Epoch%3d: zero_grad(): weight.data=%7.4f loss=%7.4f output=%7.4f target=%7.4f' \
                   % (epoch, model.weight.data, loss, output, target))

        loss.backward()        # compute gradients  (e)
        optimizer.step()       # update weights     (f)
        print('            step(): w.grad=%7.4f w.data=%7.4f' \
                   % (model.weight.grad, model.weight.data))
```

a) Iterate over epochs. Iterate over each training sample
b) Zero the gradients because PyTorch accumulates gradients
c) Forward pass
d) Calculate loss
e) Calculate the gradients
f) Update the weights

Example 1: COMP9444_demo1_simpleLinear

y=wx

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

```
Epoch  1: zero_grad(): weight.data= 0.0000 loss= 2.7458 output= 0.0000 target= 2.3434
          step(): w.grad=-2.3434 w.data= 4.4525
Epoch  2: zero_grad(): weight.data= 4.4525 loss= 2.2241 output= 4.4525 target= 2.3434
          step(): w.grad= 2.1091 w.data= 0.4453
Epoch  3: zero_grad(): weight.data= 0.4453 loss= 1.8015 output= 0.4453 target= 2.3434
          step(): w.grad=-1.8982 w.data= 4.0518
Epoch  4: zero_grad(): weight.data= 4.0518 loss= 1.4593 output= 4.0518 target= 2.3434
          step(): w.grad= 1.7084 w.data= 0.8059
Epoch  5: zero_grad(): weight.data= 0.8059 loss= 1.1820 output= 0.8059 target= 2.3434
          step(): w.grad=-1.5375 w.data= 3.7272
Epoch  6: zero_grad(): weight.data= 3.7272 loss= 0.9574 output= 3.7272 target= 2.3434
          step(): w.grad= 1.3838 w.data= 1.0980
Epoch  7: zero_grad(): weight.data= 1.0980 loss= 0.7755 output= 1.0980 target= 2.3434
          step(): w.grad=-1.2454 w.data= 3.4643

Epoch 92: zero_grad(): weight.data= 2.3436 loss= 0.0000 output= 2.3436 target= 2.3434
          step(): w.grad= 0.0002 w.data= 2.3433
Epoch 93: zero_grad(): weight.data= 2.3433 loss= 0.0000 output= 2.3433 target= 2.3434
          step(): w.grad=-0.0001 w.data= 2.3436
Epoch 94: zero_grad(): weight.data= 2.3436 loss= 0.0000 output= 2.3436 target= 2.3434
          step(): w.grad= 0.0001 w.data= 2.3433
Epoch 95: zero_grad(): weight.data= 2.3433 loss= 0.0000 output= 2.3433 target= 2.3434
          step(): w.grad=-0.0001 w.data= 2.3435
Epoch 96: zero_grad(): weight.data= 2.3435 loss= 0.0000 output= 2.3435 target= 2.3434
          step(): w.grad= 0.0001 w.data= 2.3433
Epoch 97: zero_grad(): weight.data= 2.3433 loss= 0.0000 output= 2.3433 target= 2.3434
          step(): w.grad=-0.0001 w.data= 2.3435
Epoch 98: zero_grad(): weight.data= 2.3435 loss= 0.0000 output= 2.3435 target= 2.3434
          step(): w.grad= 0.0001 w.data= 2.3434
Epoch 99: zero_grad(): weight.data= 2.3434 loss= 0.0000 output= 2.3434 target= 2.3434
          step(): w.grad=-0.0001 w.data= 2.3435
```


Example 2: COMP9444_demo2_LinearMultipleTraining

y=wx+b

0. Import packages
1. Create model → Add bias term ??
2. Prepare data → Multiple training samples
3. Choose optimizer → Learning rate
4. Train model
5. Test model

Example 2: COMP9444_demo2_LinearMultipleTraining

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

Learning rate=1.9

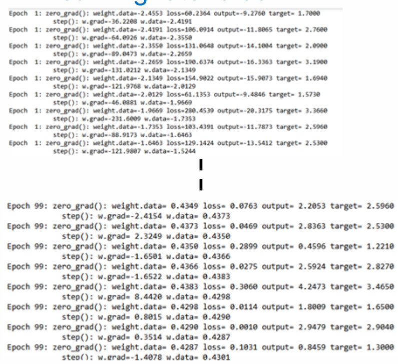
```
Epoch  1: zero_grad(): weight.data= 0.4529 loss= 0.3615 output= 0.8497 target= 1.7000
          step(): w.grad=-2.8059 w.data= 5.7841
Epoch  1: zero_grad(): weight.data= 5.7841 loss=279.9203 output=26.4210 target= 2.7600
          step(): w.grad=104.1082 w.data=-192.0214
Epoch  1: zero_grad(): weight.data=-192.0214 loss=607414.4375 output=-1100.1028 target= 2.0900
          step(): w.grad=-6062.0601 w.data=11325.8926
Epoch  1: zero_grad(): weight.data=11325.8926 loss=3045412352.0000 output=78046.9219 target= 3.1900
          step(): w.grad=523673.4688 w.data=-983653.6875
Epoch  1: zero_grad(): weight.data=-983653.6875 loss=24241367941120.0000 output=-6962953.0000 target= 1.6940
          step(): w.grad=-48253272.0000 w.data=90697560.0000
Epoch  1: zero_grad(): weight.data=90697560.0000 loss=764838498942320 64.0000 output=391110848.0000 target= 1.5730
          step(): w.grad=1630150144.0000 w.data=-3006587648.0000
Epoch  1: zero_grad(): weight.data=-3006587648.0000 loss=453952146389830991872.0000 output=-30131449856.0000 target= 3.3660
          step(): w.grad=-294655459328.0000 w.data=556838748160.0000
Epoch  1: zero_grad(): weight.data=556838748160.0000 loss=61211404278833700593991 68.0000 output=3498897047552.0000 target= 2.59
60
          step(): w.grad=21630183014400.0000 w.data=-40540506685440.0000
Epoch  1: zero_grad(): weight.data=-40540506685440.0000 loss=4939031490523594257887500 6976.0000 output=-314293862006784.0000 ta
rget= 2.5300

Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 3.3660
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 2.5960
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 2.5300
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 1.2210
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 2.8270
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 3.4650
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 1.6500
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 2.9040
          step(): w.grad=    nan w.data=    nan
Epoch 99: zero_grad(): weight.data=    nan loss=    nan output=    nan target= 1.3000
          step(): w.grad=    nan w.data=    nan
```

Example 2: COMP9444_demo2_LinearMultipleTraining

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

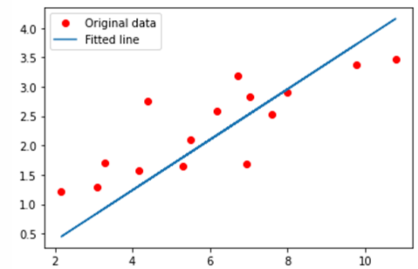
Learning rate=0.001



Example 2: COMP9444_demo2_LinearMultipleTraining

0. Import packages
1. Create model
2. Prepare data
3. Choose optimizer
4. Train model
5. Test model

Learning rate=0.001



Example 3: COMP9444_demo3_LinearExtension

0. Import packages
1. Create model → Run on GPU if available
2. Prepare data → Change batch size (batch)
3. Choose optimizer → Change optimizer to ADAM
4. Train model → Modify output to sum (because batch)
5. Test model

Example 4: COMP9444_demo4_LinearExtensionTorch

0. Import packages
1. Create model → Use PyTorch predefined linear model
2. Prepare data
3. Choose optimizer → Use PyTorch predefined loss function
4. Train model
5. Test model

Loss Functions

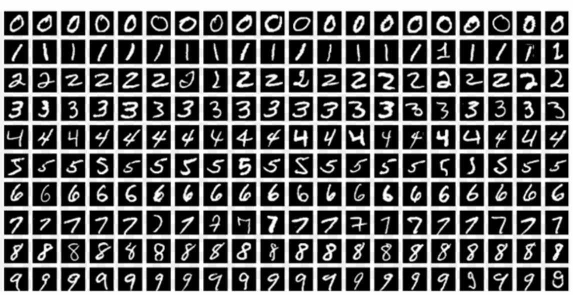
nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input xx and target yy.
nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input xx and target yy.
nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input logits and target.
nn.CTCLoss	The Connectionist Temporal Classification loss.
nn.NLLLoss	The negative log likelihood loss.
nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
nn.KLDivLoss	The Kullback-Leibler divergence loss.
nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
nn.BCEWithLogitsLoss	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
nn.MarginRankingLoss	Creates a criterion that measures the loss given inputs x1x1, x2x2, two 1D mini-batch or 0D *Tensors*, and a label 1D mini-batch or 0D *Tensor* yy (containing 1 or -1).
nn.HingeEmbeddingLoss	Measures the loss given an input tensor xx and a labels tensor yy (containing 1 or -1).
nn.MultiLabelMarginLoss	Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input xx (a 2D mini-batch *Tensor*) and output yy (which is a 2D *Tensor* of target class indices).
nn.HuberLoss	Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise.
nn.SmoothL1Loss	Creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise.
nn.SoftMarginLoss	Creates a criterion that optimizes a two-class classification logistic loss between input tensor xx and target tensor yy (containing 1 or -1).
nn.MultiLabelSoftMarginLoss	Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input xx and target yy of size (N,C)(N,C).
nn.CosineEmbeddingLoss	Creates a criterion that measures the loss given input tensors x1x1, x2x2 and a *Tensor* label yy with values 1 or -1.
nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input xx (a 2D mini-batch *Tensor*) and output yy (which is a 1D tensor of target class indices, 0≤y≤x.size(1)−10≤y≤x.size(1)−1):
nn.TripletMarginLoss	Creates a criterion that measures the triplet loss given an input tensors x1x1, x2x2, x3x3 and a margin with a value greater than 00.
nn.TripletMarginWithDistanceLoss	Creates a criterion that measures the triplet loss given input tensors aa, pp, and nn (representing anchor, positive, and negative examples, respectively), and a nonnegative, real-valued function ("distance function") used to compute the relationship between the anchor and positive example ("positive distance") and the anchor and negative example ("negative distance").

https://pytorch.org/docs/stable/nn.html#loss-functions

Example 5: COMP9444_demo5_LinearExtensionTest

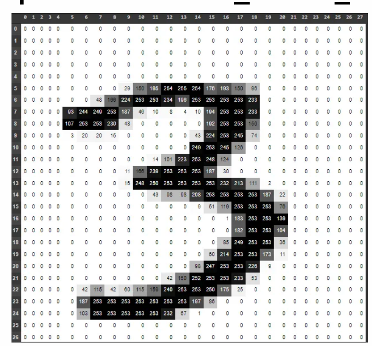
0. Import packages
1. Create model
2. Prepare data → Split data intro training and test
3. Choose optimizer
4. Train model → Train model on training set
5. Test model → Test model on test set

Example 6: COMP9444_demo6_NeuralNetwork (Dataset)



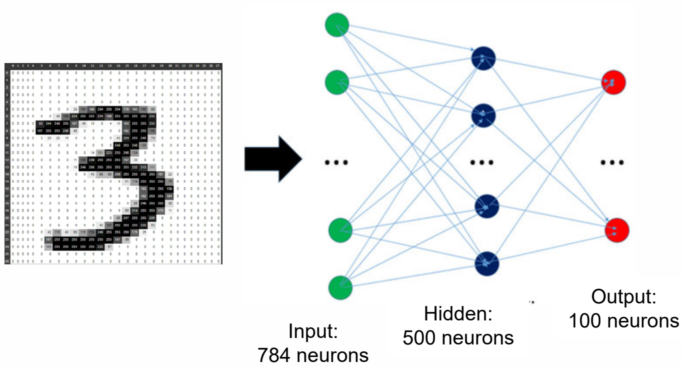
10 classes

Example 6: COMP9444_demo6_NeuralNetwork (Dataset)



- 28 x 28 = 784 inputs
- Number indicate darkness

Example 6: COMP9444_demo6_NeuralNetwork (Model)



Example 6: COMP9444_demo6_NeuralNetwork (Model)

0. Import packages
1. Create model → Create neural network model
2. Prepare data → Import new dataset
3. Choose optimizer
4. Train model → Train model on training set
5. Test model → Test model on test set

Sequential Components

Network layers:
- nn.Linear()
- nn.Conv2d() (Week 4)

Intermediate Operators:
- nn.Dropout()
- nn.BatchNorm() (Week 4)

Activation Functions:
- nn.Sigmoid()
- nn.Tanh()
- nn.ReLU() (Week 3)

More on the computational graph

- optimizer.zero grad() sets all .grad components to zero.
- loss.backward() updates the .grad component of all Parameters by backpropagating gradients through the computational graph.
- optimizer.step() updates the .data components.
- By default, loss.backward() discards the computational graph after computing the gradients.
- If needed, we can force it to keep the computational graph by calling it this way:

```
loss.backward(retain graph=True)
```

In summary…

-

Typical Structure of a PyTorch Program

```
# create neural network
net = MyNetwork().to(device) # CPU or GPU

# prepare to load the training and test data
train_loader = torch.utils.data.DataLoader(...)
test_loader = torch.utils.data.DataLoader(...)

# choose between SGD, Adam or other optimizer
optimizer = torch.optim.SGD(net.parameters,...)

for epoch in range(1, epochs):   # training loop
    train(args, net, device, train_loader, optimizer)
    # periodically evaluate network on test data
    if epoch %10 == 0:
        test( args, net, device, test_loader)
```


Defining a Network Structure

```
class  MyNetwork(torch.nn.Module):

    def __init__(self):
        super(MyNetwork, self).__init__()
        # define structure of the network here

    def forward(self, input):
        # apply network and return output
```


Declaring Data Explicitly

```
import torch.utils.data

# input and target values for the XOR task
input = torch.Tensor([[0,0],[0,1],[1,0],[1,1]])
target = torch.Tensor([[0],[1],[1],[0]])

xdata = torch.utils.data.TensorDataset(input,target)
train_loader = torch.utils.data.DataLoader(xdata,batch_size=4)
```

Loading Data from a .csv File

```
import pandas as pd

df = pd.read_csv("sonar.all-data.csv")
df = df.replace('R',0)
df = df.replace('M',1)
data = torch.tensor(df.values,dtype=torch.float32)
num_input = data.shape[1] - 1
input = data[:,0:num_input]
target = data[:,num_input:num_input+1]
dataset = torch.utils.data.TensorDataset(input,target)
```


Custom Datasets

```
from data import ImageFolder
    # load images from a specified directory
    dataset = ImageFolder(folder, transform)

import torchvision.datasets as dsets
    # download popular image datasets remotely
    mnistset = dsets.MNIST(...)
    cifarset = dsets.CIFAR10(...)
    celebset = dsets.CelebA(...)
```


Choosing an Optimizer

```
# SGD stands for "Stochastic Gradient Descent"
optimizer = torch.optim.SGD( net.parameters(),
    lr=0.01, momentum=0.9,
    weight_decay=0.0001)

# Adam = Adaptive Moment Estimation (good for deep networks)
optimizer = torch.optim.Adam(net.parameters(),eps=0.000001,
    lr=0.01, betas=(0.5,0.999),
    weight_decay=0.0001)
```


Training

```
def train(args, net, device, train_loader, optimizer):

    for batch_idx, (data,target) in enumerate(train_loader):

        optimizer.zero_grad()   # zero the gradients
        output = net(data)      # apply network
        loss = ...              # compute loss function
        loss.backward()         # compute gradients
        optimizer.step()        # update weights
```

Loss Functions

```
loss = torch.sum((output-target)*(output-target))
loss = F.nll_loss(output,target)                  # (Week 3)
loss = F.binary_cross_entropy(output,target)      # (Week 3)
loss = F.softmax(output,dim=1)                    # (Week 3)
loss = F.log_softmax(output,dim=1)                # (Week 3)
```


Testing

```
def test(args, net, device, test_loader):
    with torch.no_grad():   # suppress updating of gradients
        net.eval() # toggle batch norm, dropout
            for data, target in test_loader:
                output = model(data)
                test_loss = ...
                print(test_loss)
        net.train() # toggle batch norm, dropout back again
```