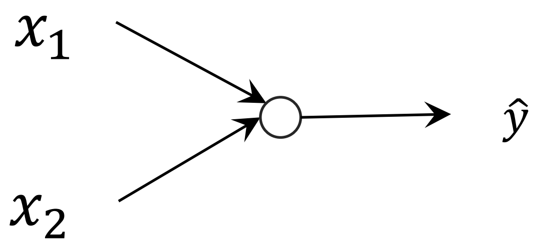
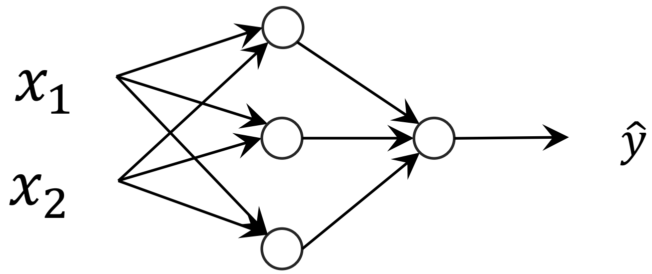
Deep Neural Networks

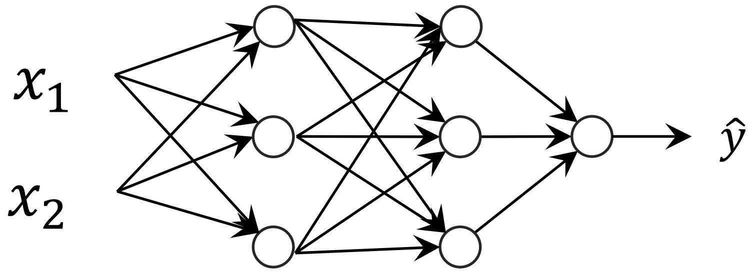
What is a deep neural network?



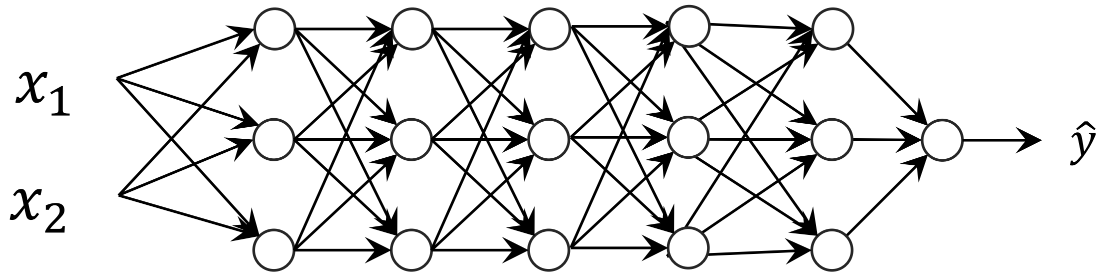
Logistic regression



1 hidden layer

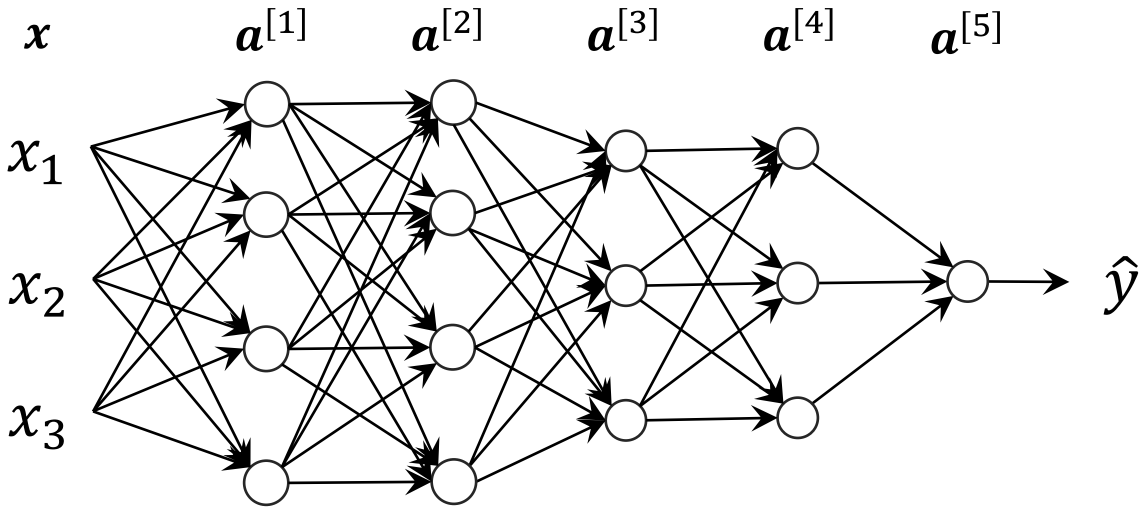


2 hidden layers



5 hidden layers

Deep neural network



f: activation function (e.g., ReLU)

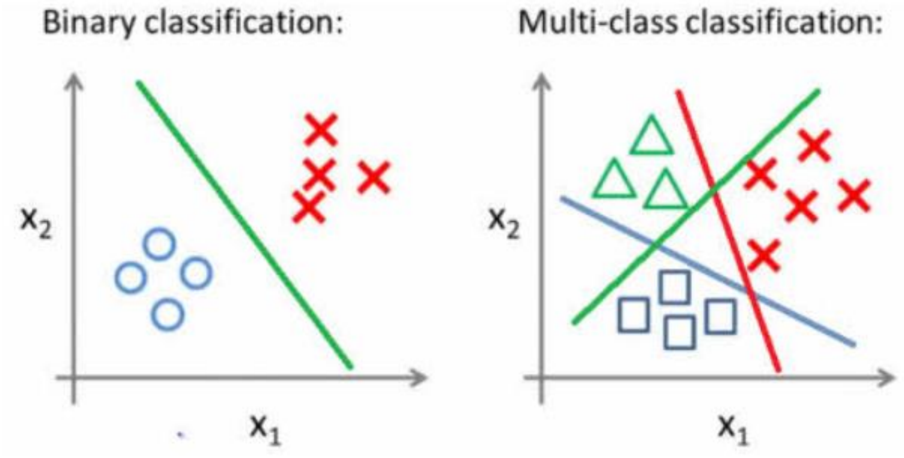
$$\boldsymbol{a}^{[1]} = f\left(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}\right) \qquad \boldsymbol{a}^{[i]} = f\left(\boldsymbol{W}^{[i]}\boldsymbol{a}^{[i-1]} + \boldsymbol{b}^{[i]}\right) \qquad \hat{y} = \boldsymbol{a}^{[5]} = \sigma\left(\boldsymbol{W}^{[5]}\boldsymbol{a}^{[4]} + \boldsymbol{b}^{[5]}\right)$$

For $i = 2,3,4$

Output types

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

Output types



Amey band (2020)

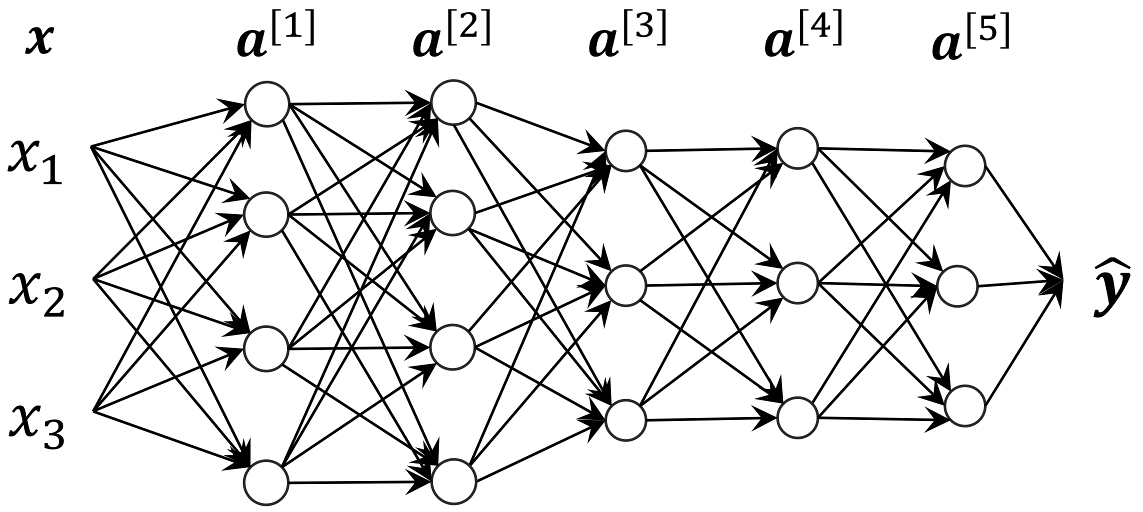
	Binary classification	(n-ary) Classification
Output classes	0/1	1/2/3/../n
Output activation function	$\sigma(x) = \frac{1}{1+e^{-x}}$	$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{i'=1}^{n} e^{x_{i'}}}$
Loss function	$-\hat{y}\log y - (1-\hat{y})\log(1-y)$	$-\sum_{i=1}^{n} y_i \log \hat{y}_i$
	Binary cross entropy	Cross entropy

Softmax outputs a categorical (multinoulli) distribution

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{i'=1}^{n} e^{x_{i'}}} \qquad \mathbb{R}^n \to (0,1)^n$$

- $softmax(\boldsymbol{x})_i \geq 0$
- $\sum_{i=1}^{n} softmax(\boldsymbol{x})_i = 1$

Deep neural network



f: activation function (e.g., ReLU)

$$\boldsymbol{a}^{[1]} = f\left(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}\right)$$

$$\boldsymbol{a}^{[i]} = f\left(\boldsymbol{W}^{[i]}\boldsymbol{a}^{[i-1]} + \boldsymbol{b}^{[i]}\right)$$

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{[5]} = softmax\left(\boldsymbol{W}^{[5]}\boldsymbol{a}^{[4]} + \boldsymbol{b}^{[5]}\right)$$

Parameters vs Hyper parameters

- (Model) parameters
  - A model parameter is a configuration variable that is internal to the model and whose value can be estimated from data
  - Examples) $\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}, \dots, \boldsymbol{b}^{[1]}, \boldsymbol{b}^{[2]}, \dots,$
- Hyper parameters
  - A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data
  - Examples
    - Learning rate
    - Number of layers
    - Number of hidden units for each layer

Stochastic Gradient Descent
Mini-Batch Gradient Descent

Stochastic gradient descent

- A stochastic approximation of gradient descent optimization

Approximate $\nabla J(\boldsymbol{w})$ by $\nabla J_i(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} L(y_i, \hat{y})$

Stochastic approximation

X	P(X)
1	0.2
2	0.5
10	0.2
20	0.1

Expectation

$E[X] = 1 * 0.2 + 2 * 0.5 + 10 * 0.2 + 20 * 0.1 = 5.2$

Stochastic approximation of the expectation (Sample mean)

$Step$ 1) Draw k sample values according to P

e.g.) 10, 2, 2, 1

Step 2) Take the average of the values

e.g.) 15/4 = 3.75

Let Y be the sample mean

E[X] = E[Y]

Y is an *unbiased* estimator of E[X] regardless of k

Stochastic gradient descent (SGD)

$$J(\theta) = \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

$$\nabla J(\theta) = \sum_{i=1}^{m} \nabla L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

- A stochastic approximation of gradient descent optimization

Approximate $\nabla J(\boldsymbol{w})$ by $\nabla J_i(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} L\left(\hat{y}^{(i)}, y^{(i)}\right)$

Gradient descent

1. Initialize parameters $\boldsymbol{w}$
2. For each epoch:
3. $\quad d\boldsymbol{w} = 0$
4. $\quad$ For $i = 1$ to m:
5. $\quad\quad \mathrm{d}\boldsymbol{w} \coloneqq \mathrm{d}\boldsymbol{w} + \nabla J_i(\boldsymbol{w})$
6. $\quad \nabla J(\boldsymbol{w}) = \frac{d\boldsymbol{w}}{m}$
7. $\quad \boldsymbol{w} \coloneqq \boldsymbol{w} - \eta \nabla J(\boldsymbol{w})$

Batch size: m

Easy to parallelize

Inefficient

Stable

Stochastic gradient descent

1. Initialize parameters $\boldsymbol{w}$
2. For each epoch:
3. $\quad$ Randomly shuffle training examples
4. $\quad$ For $i = 1$ to m:
5. $\quad\quad \boldsymbol{w} \coloneqq \boldsymbol{w} - \eta \nabla J_i(\boldsymbol{w})$

Batch size: 1

Hard to parallelize

Fast update

Unstable

Stochastic gradient descent

(Batchsize) = 1

Mini-batch gradient descent

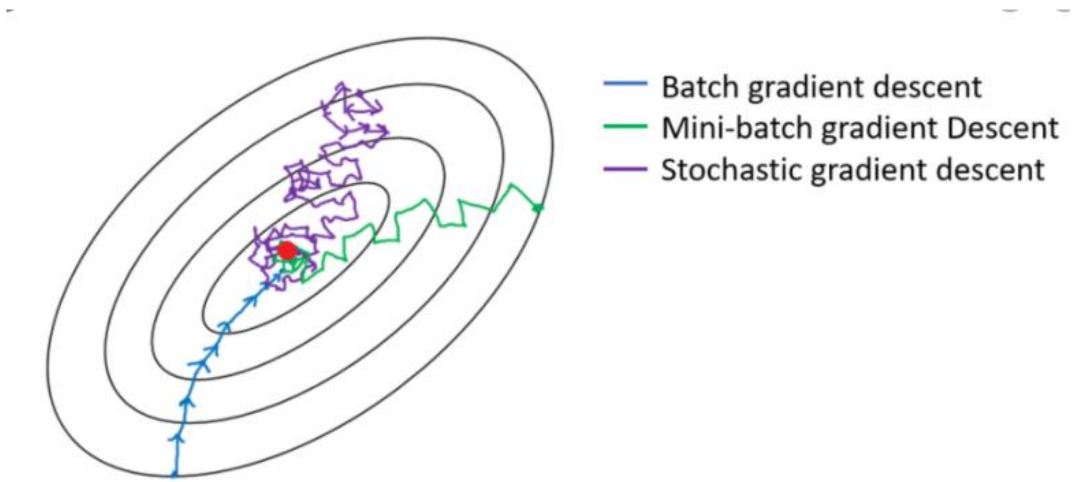
1<(Batchsize) <m

Gradient descent

(Batchsize) = m

- Mini-batch gradient

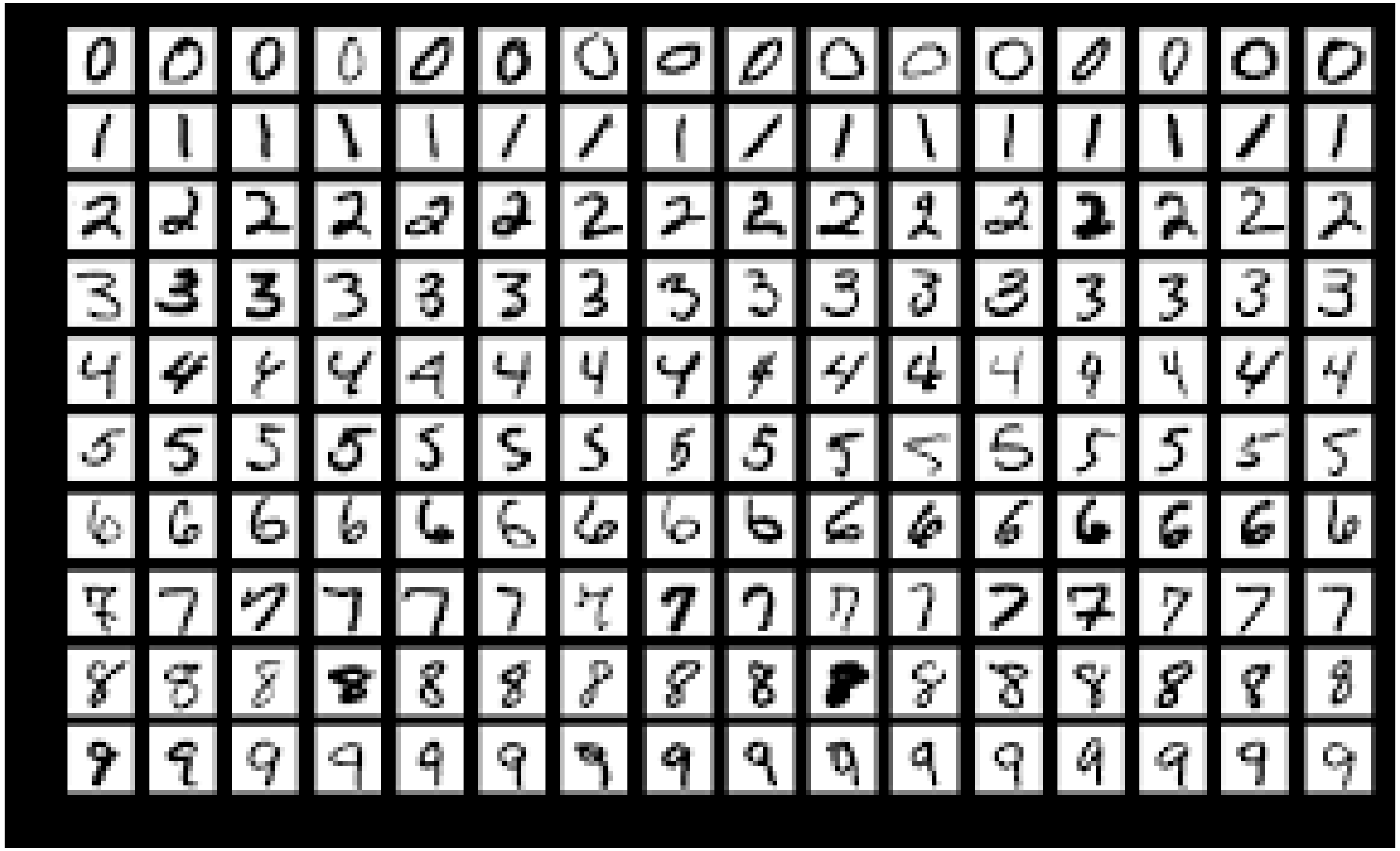
1. Initialize parameters $\boldsymbol{w}$
2. For each epoch:
3. Randomly shuffle training examples
4. For $b = 1$ to $\frac{m}{batchsize}$:
5. $\boldsymbol{w} := \boldsymbol{w} - \eta \nabla J^b(\boldsymbol{w})$



$$J^b(\boldsymbol{w}) = \frac{1}{batchsize} \sum_{(x_i, y_i) \in b_{th}\ batch} L(y_i, \hat{y})$$

Implementing a DNN with PyTorch

MNIST



Data loading

- Import torchvision
- From torchvision import datasets

```
batch_size = 12

train_data = datasets.MNIST('D:\datasets', train=True, download=True, transform=transforms.ToTensor())
test_data = datasets.MNIST('D:\datasets', train=False, download=True,transform=transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size)
```

Model

```python
class MLP(nn.Module):
    def __init__(self):
        super().__init__()

        self.in_dim = 28*28 # MNIST
        self.out_dim = 10

        self.fc1 = nn.Linear(self.in_dim, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, self.out_dim)

        self.relu = nn.ReLU()
        self.log_softmax = nn.LogSoftmax()

    def forward(self, x):
        a1 = self.relu(self.fc1(x.view(-1, self.in_dim)))
        a2 = self.relu(self.fc2(a1))
        a3 = self.relu(self.fc3(a2))
        a4 = self.relu(self.fc4(a3))
        logit = self.fc5(a4)
        return logit
```

Train

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

```python
for epoch in range(10):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if (i+1) % 2000 == 0:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.209
[1,  4000] loss: 0.739
[2,  2000] loss: 0.316
[2,  4000] loss: 0.230
[3,  2000] loss: 0.154
[3,  4000] loss: 0.144
[4,  2000] loss: 0.112
[4,  4000] loss: 0.101
[5,  2000] loss: 0.075
[5,  4000] loss: 0.082
[6,  2000] loss: 0.061
[6,  4000] loss: 0.063
[7,  2000] loss: 0.046
[7,  4000] loss: 0.054
[8,  2000] loss: 0.033
[8,  4000] loss: 0.041
[9,  2000] loss: 0.029
[9,  4000] loss: 0.033
[10,  2000] loss: 0.021
[10,  4000] loss: 0.025
Finished Training
```

Test

```python
import matplotlib.pyplot as plt
import numpy as np
```

```python
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```python
dataiter = iter(test_loader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images, nrow = batch_size))
print('GroundTruth')
print("  "+' '.join('%3s' % label.item() for label in labels))

outputs = model(images)
_, predicted = torch.max(outputs, 1)
print("Prediction")
print("  "+' '.join('%3s' % label.item() for label in predicted))
```


```
GroundTruth
    7   2   1   0   4   1   4   9   5   9   0   6
Prediction
    7   2   1   0   4   1   4   9   6   9   0   6
```

Test

```python
n_predict = 0
n_correct = 0

for data in test_loader:
    inputs, labels = data
    outputs = model(inputs)
    _, predicted = torch.max(outputs, 1)

    n_predict += len(predicted)
    n_correct += (labels == predicted).sum()


print(f"{n_correct}/{n_predict}")
print(f"Accuracy: {n_correct/n_predict:.3f}")
```

```
9761/10000
Accuracy: 0.976
```

nn.CrossEntropyLoss

- Softmax를 포함하고 있으므로 모델에서는 logit을 return

```
def forward(self, x):
    a1 = self.relu(self.fc1(x.view(-1, self.in_dim)))
    a2 = self.relu(self.fc2(a1))
    a3 = self.relu(self.fc3(a2))
    a4 = self.relu(self.fc4(a3))
    logit = self.fc5(a4)
    return logit
```

CROSSENTROPYLOSS

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=- 100, reduce=None, reduction='mean', label_smoothing=0.0)` [SOURCE]

This criterion computes the cross entropy loss between input and target.

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 (C) for unbatched input, $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The last being useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C)$ where C is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{`mean';} \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{`sum'.} \end{cases}$$

Note that this case is equivalent to the combination of `LogSoftmax` and `NLLLoss`.

Sequential

- Modules will be added to it in the order they are passed in the constructor

```
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
```

```
# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

A DNN with hyper parameters

MLP with hyper parameters

```
class MLP_h(nn.Module):
    def __init__(self,hidden_units = [512,256,128]):
        super().__init__()

        self.in_dim = 28*28 # MNIST
        self.out_dim = 10

        self.l_layers = []
        self.l_layers.append(nn.Linear(self.in_dim, hidden_units[0]))
        for i in range(len(hidden_units)-1):
            self.l_layers.append(nn.Linear(hidden_units[i], hidden_units[i+1]))
        self.l_layers.append(nn.Linear(hidden_units[-1], self.out_dim))

        self.relu = nn.ReLU()
        self.log_softmax = nn.LogSoftmax()

    def forward(self, x):
        a = x.view(-1, self.in_dim)
        for l in range(len(self.l_layers)):
            z = self.l_layers[l](a)
            if l == len(self.l_layers) -1:
                logit = z
            else:
                a = self.relu(z)

        return logit
```

```
model = MLP_h([2,3])
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most re
<ipython-input-21-e9ee64f89c5c> in <module>
      1 model = MLP_h([2,3])
      2 criterion = nn.CrossEntropyLoss()
----> 3 optimizer = optim.SGD(model.parameters(), lr = 0.01

~#anaconda3#lib#site-packages#torch#optim#sgd.py in __i
ov)
     67         if nesterov and (momentum <= 0 or dampening
     68             raise ValueError("Nesterov momentum re
---> 69         super(SGD, self).__init__(params, defaults
     70
     71     def __setstate__(self, state):

~#anaconda3#lib#site-packages#torch#optim#optimizer.py
     48         param_groups = list(params)
     49         if len(param_groups) == 0:
---> 50             raise ValueError("optimizer got an e
     51         if not isinstance(param_groups[0], dict):
     52             param_groups = [{'params': param_groups}

ValueError: optimizer got an empty parameter list
```

ModuleList

- Holds submodules in a list
- Can be indexed like a regular Python list
- Modules it contains are properly registered, and will be visible by all Module methods

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

Programming Assignment 3: DNN

- Dataset: MNIST
- Requirement
  - Plot accuracy varying the number of layers (2,3,4,5 layers)
  - Nothing more, but, using ModuleList may save your time
  - Hint:
- Due: 2022/10/23 PM 11:59
- Submission
  - Report (pdf or docx)
    - A figure: accuracy vs # layers
    - Source code
  - Source file (.py or .ipynb)

Evaluation measures

Evaluation measures :Classification

- Accuracy

$$(Accuracy) = \frac{CorrectPredictions}{TotalPredictions}$$

Evaluation measures :Binary classification

- 4 numbers
  - True positives
    - # positive observations the model correctly predicted as positive
  - True negatives
  - False positives
  - False negatives

		Predicted class	
		True	False
Actual Class	True	TP	FN
	False	FP	TN

Evaluation measures
:Binary classification

- Precision
  - $precision = \frac{TP}{TP+FP}$
- Recall
  - $recall = \frac{TP}{TP+FN}$
- F1-score
  - The harmonic mean of precision and recall
  - $(F1 - score) = \frac{2 * precision * recall}{precision + recall}$

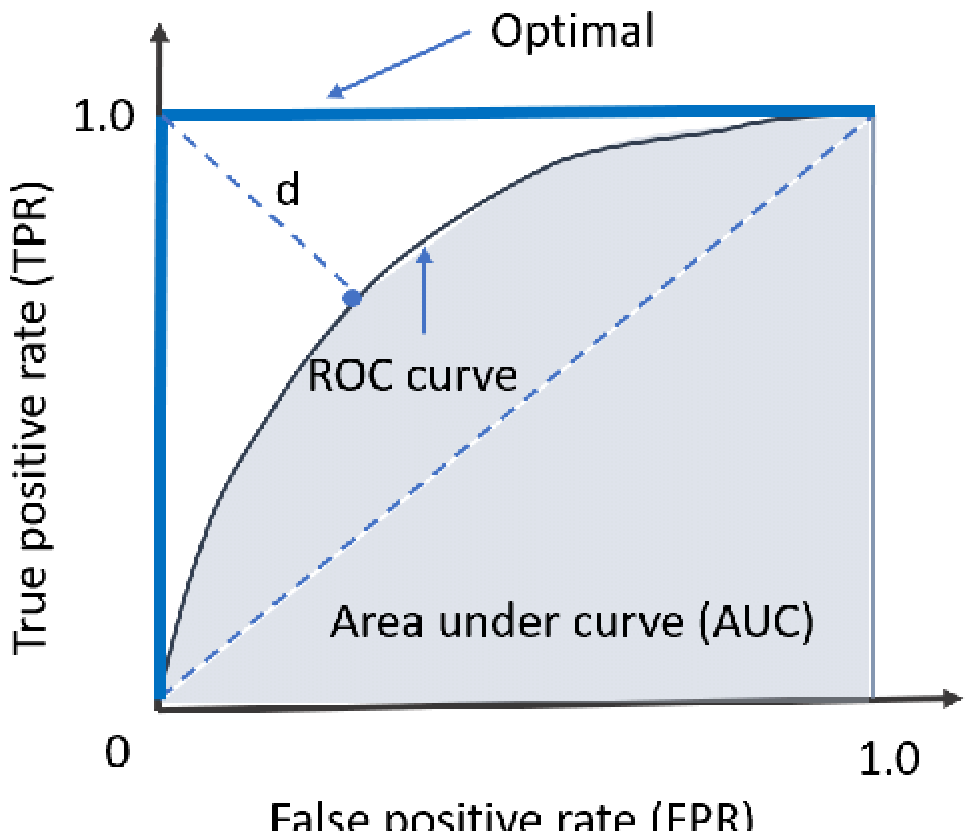
Evaluation measures :ROC curve and AUC

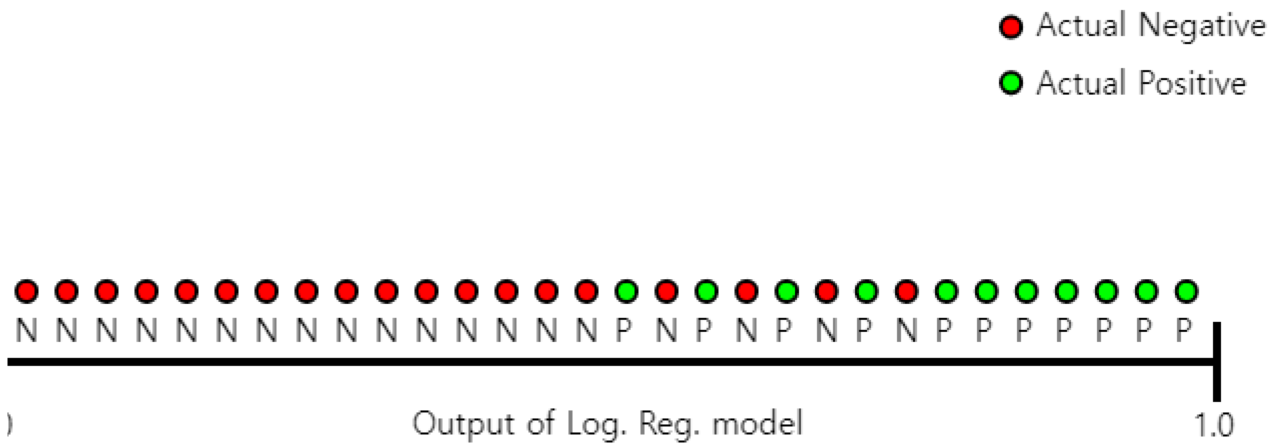
True positive rate

$$TPR = \frac{TP}{TP + FN}$$

False positive rate

$$FPR = \frac{FP}{FP + TN}$$





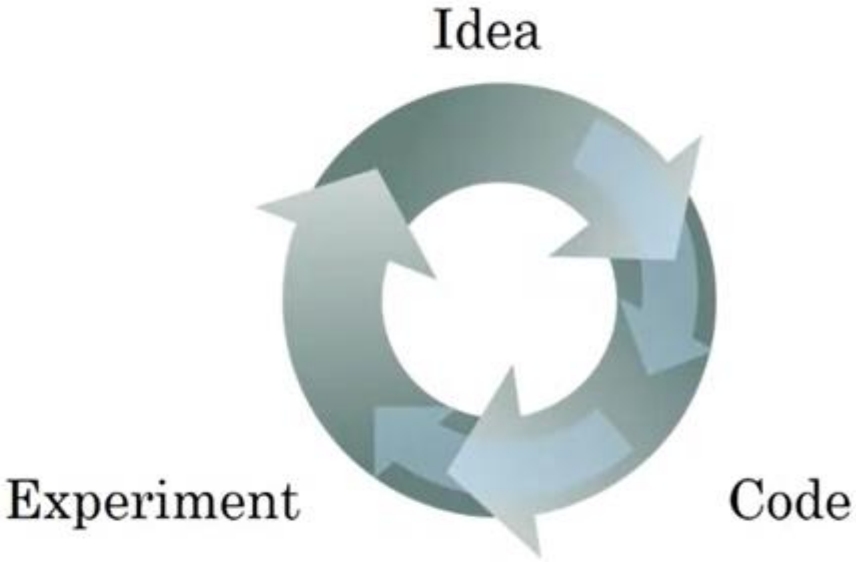
https://www.researchgate.net/figure/Receiver-operating-characteristic-ROC-curve-and-threshold-determinator-65-The-area_fig3_343846811

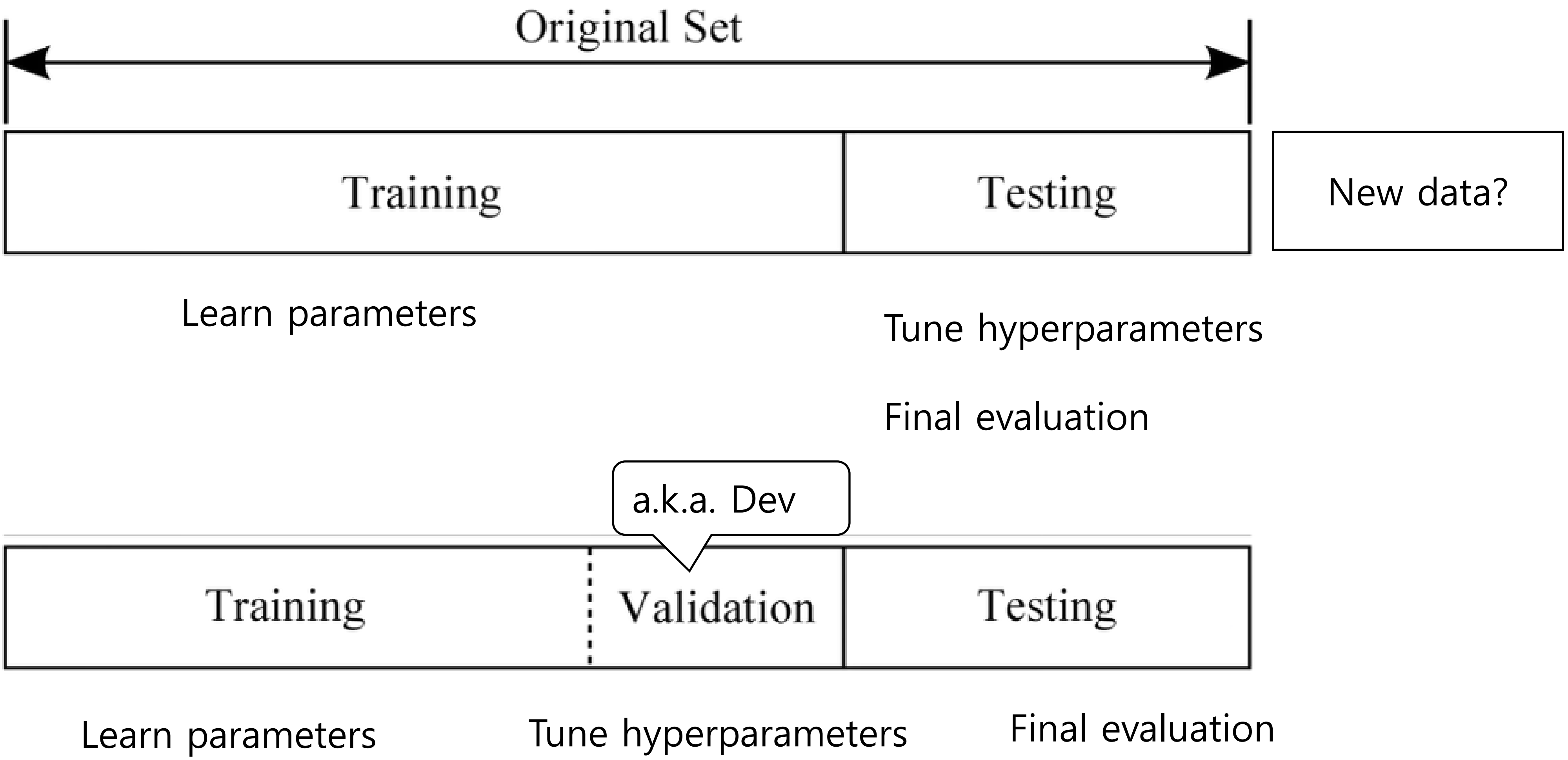
https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

Train, Validation, Test datasets

Applied ML is a highly iterative process

- Hyper parameters
  - # layers
  - # hidden units
  - Learning rates
  - Activation functions
  - …




Original Set
Training
Testing
New data?
Learn parameters
Tune hyperparameters
Final evaluation
a.k.a. Dev
Training
Validation
Testing
Learn parameters
Tune hyperparameters
Final evaluation

Mismatched train/test distribution

Training set:
Cat pictures from webpages

Dev/test sets:
Cat pictures from users using your app

Test

Dev

Test