ML/DL for Everyone with PYTORCH

Lecture 6: Logistic Regression

Sung Kim <hunkim+ml@gmail.com> HKUST

Code: https://github.com/hunkim/PyTorchZeroToAll

Slides: http://bit.ly/PyTorchZeroAll

Call for Comments

Please feel free to add comments directly on these slides.

Other slides: http://bit.ly/PyTorchZeroAll

Picture from http://www.tssablog.org/archives/3280

ML/DL for Everyone with PYTORCH

Lecture 6:
Logistic Regression

Sung Kim <hunkim+ml@gmail.com> HKUST

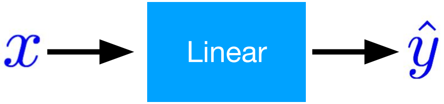
Code: https://github.com/hunkim/PyTorchZeroToAll

Slides: http://bit.ly/PyTorchZeroAll

Binary prediction (0 or 1) is very useful!

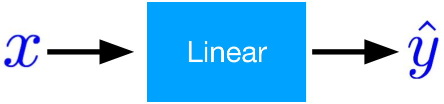
- Spent *N* hours for study, pass or fail?
- GPA and GRE scores for the HKUST PHD program, admit or not?
- Soccer game against Japan, win or lose?
- She/he looks good, propose or not?
- …

Linear model



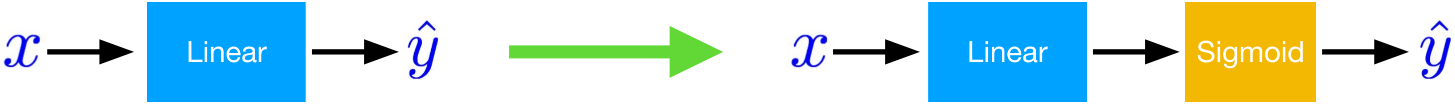
Hours (x)	Points
1	2
2	4
3	6
4	?

Logistic regression: pass/fail (0 or 1)



Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?

Logistic regression: pass/fail (0 or 1)

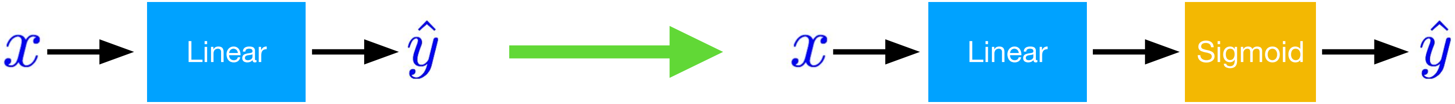


x → Linear → $\hat{y}$

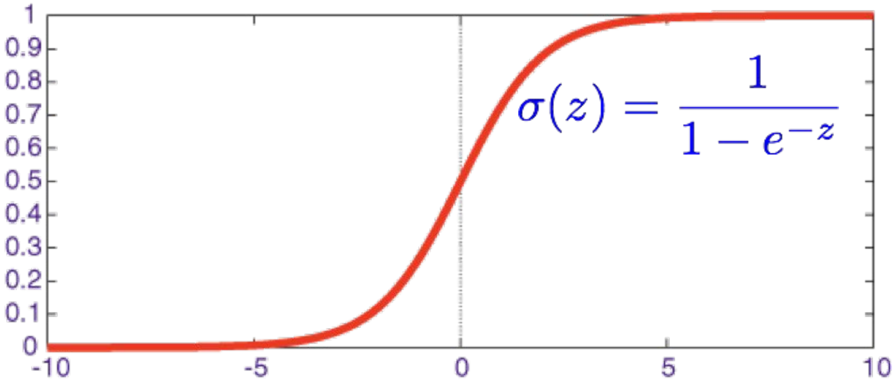
x → Linear → Sigmoid → $\hat{y}$

Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?

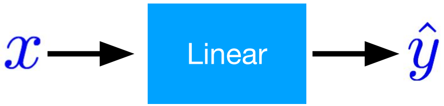
Meet sigmoid

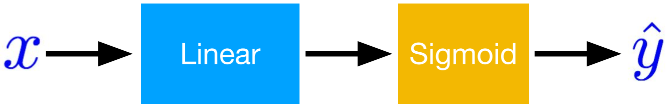


Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



Meet sigmoid





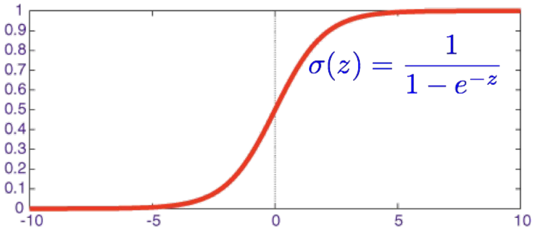
$$\hat{y} = x * w + b$$

→

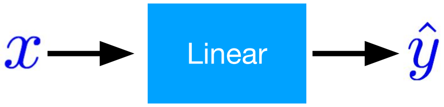
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

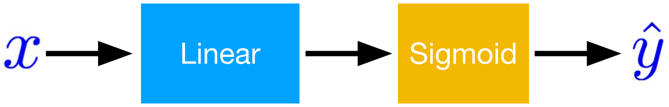
$$\hat{y} = \sigma(x * w + b)$$

Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



Meet Cross Entropy Loss





$$\hat{y} = x * w + b$$

→

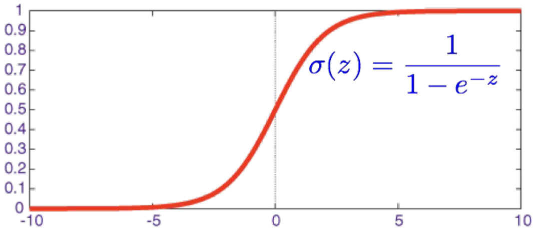
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

$$\hat{y} = \sigma(x * w + b)$$

$$loss = \frac{1}{N} \sum_{n=1}^{N} (\hat{y_n} - y_n)^2$$

$$loss = -\frac{1}{N} \sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

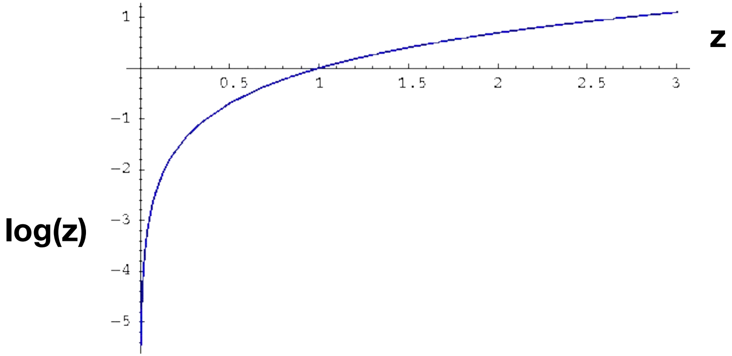
Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



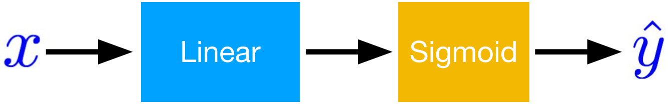
(Binary) Cross Entropy Loss

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

y	y_pred	loss
0	0.2	
0	0.8	
1	0.1	
1	0.9	

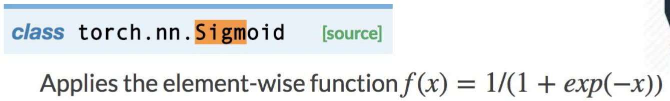


Logistic regression



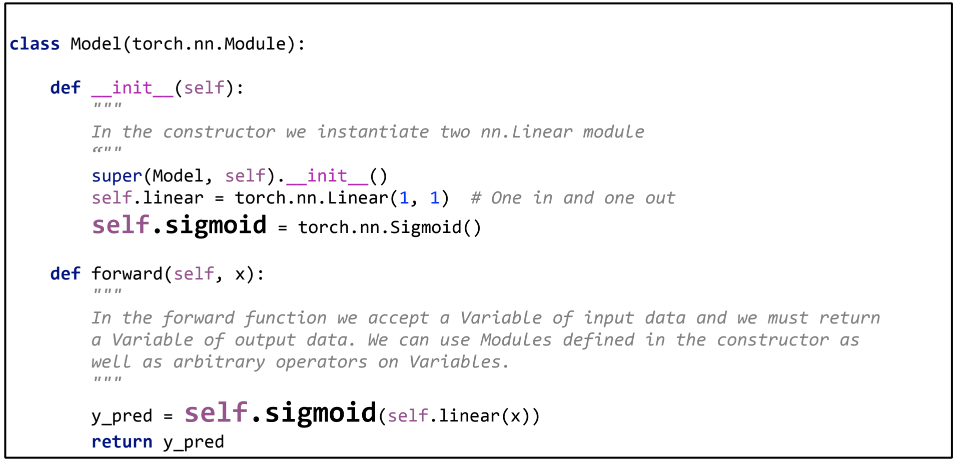
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

$$\hat{y} = \sigma(x * w + b)$$



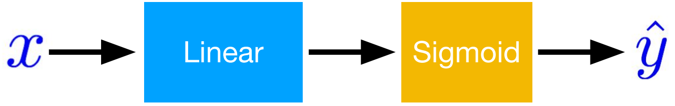
class torch.nn.Sigmoid [source]

Applies the element-wise function $f(x) = 1/(1 + exp(-x))$



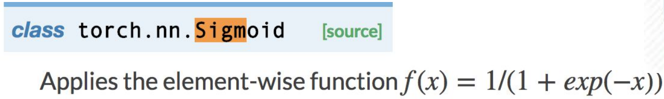
```python
class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred
```

Logistic regression



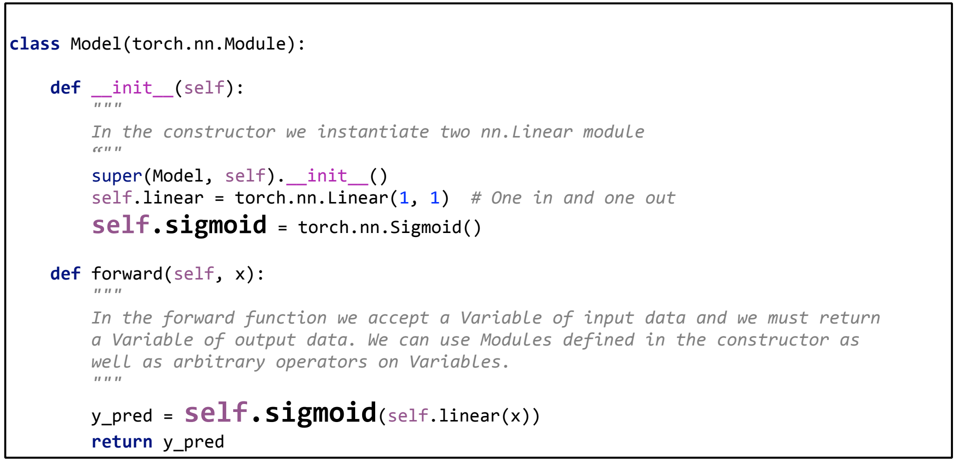
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

$$\hat{y} = \sigma(x * w + b)$$



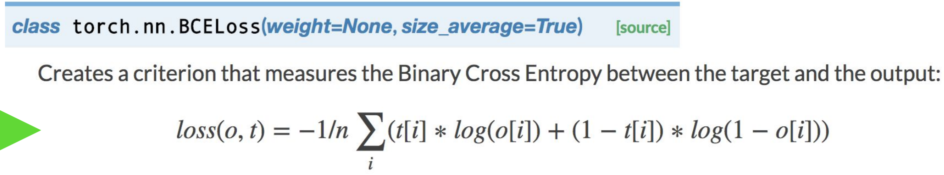
class torch.nn.Sigmoid [source]

Applies the element-wise function $f(x) = 1/(1 + exp(-x))$



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

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred
```

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)$$



class torch.nn.BCELoss(*weight=None*, *size_average=True*) [source]

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

$$loss(o, t) = -1/n \sum_i (t[i] * log(o[i]) + (1 - t[i]) * log(1 - o[i]))$$

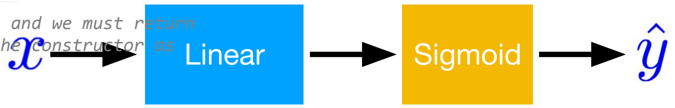
```
criterion = torch.nn.BCELoss(size_average=True)
```

Logistic regression

```python
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# After training
hour_var = Variable(torch.Tensor([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

Logistic regression

1 **Design your model using class**

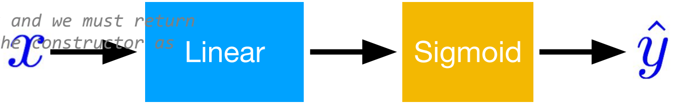


```python
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# After training
hour_var = Variable(torch.Tensor([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

Logistic regression

```
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# After training
hour_var = Variable(torch.Tensor([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

1 Design your model using class

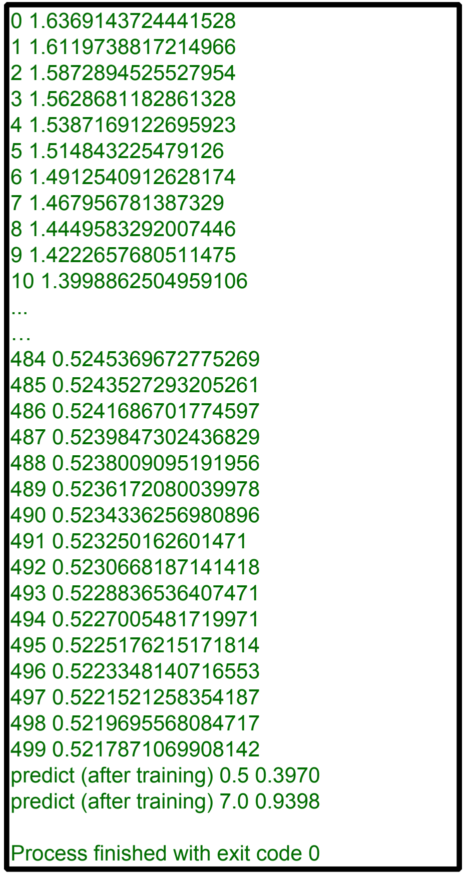


2 Construct loss and optimizer (select from PyTorch API)

3 Training cycle (forward, backward, update)

Logistic regression

```python
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.sigmoid(self.linear(x))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# After training
hour_var = Variable(torch.Tensor([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

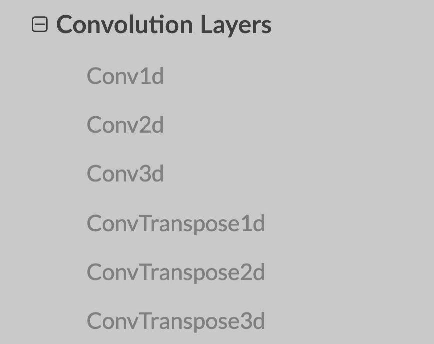


```
0 1.6369143724441528
1 1.6119738817214966
2 1.5872894525527954
3 1.5628681182861328
4 1.5387169122695923
5 1.514843225479126
6 1.4912540912628174
7 1.467956781387329
8 1.4449583292007446
9 1.4222657680511475
10 1.3998862504959106
...
...
484 0.5245369672775269
485 0.5243527293205261
486 0.5241686701774597
487 0.5239847302436829
488 0.5238009095191956
489 0.5236172080039978
490 0.5234336256980896
491 0.523250162601471
492 0.5230668187141418
493 0.5228836536407471
494 0.5227005481719971
495 0.5225176215171814
496 0.5223348140716553
497 0.5221521258354187
498 0.5219695568084717
499 0.5217871069908142
predict (after training) 0.5 0.3970
predict (after training) 7.0 0.9398

Process finished with exit code 0
```

Building fun models

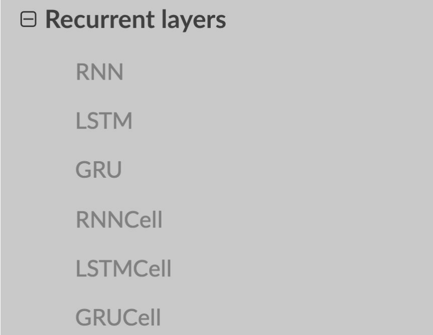
- Neural Net components
  - CNN
  - RNN
  - Activations
- Losses
- Optimizers

torch.nn



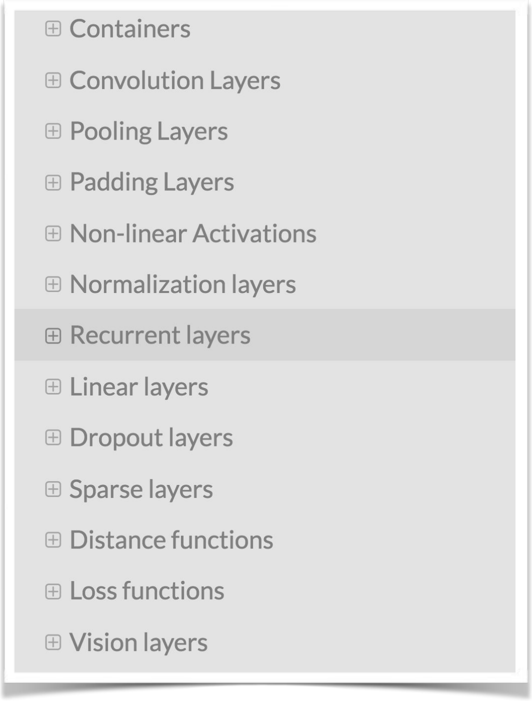
⊟ **Convolution Layers**

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d

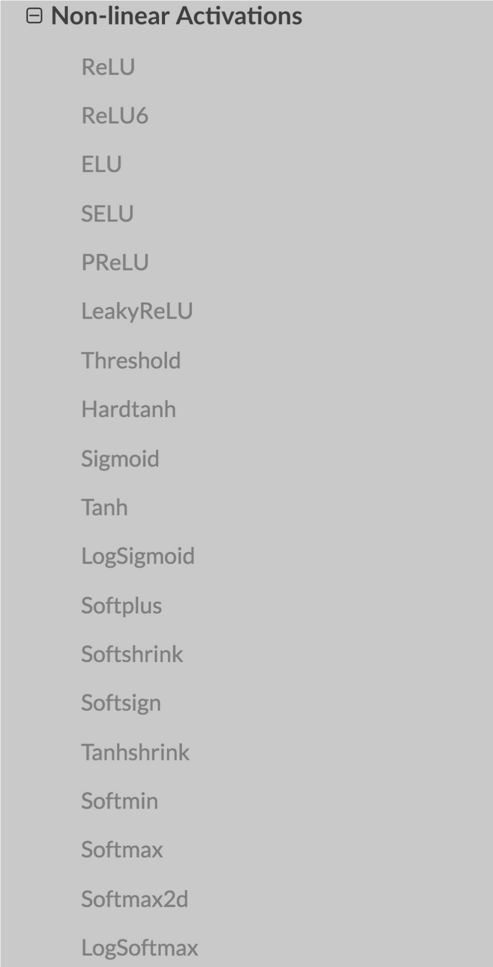


⊟ **Recurrent layers**

- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell



- ⊞ Containers
- ⊞ Convolution Layers
- ⊞ Pooling Layers
- ⊞ Padding Layers
- ⊞ Non-linear Activations
- ⊞ Normalization layers
- ⊞ Recurrent layers
- ⊞ Linear layers
- ⊞ Dropout layers
- ⊞ Sparse layers
- ⊞ Distance functions
- ⊞ Loss functions
- ⊞ Vision layers

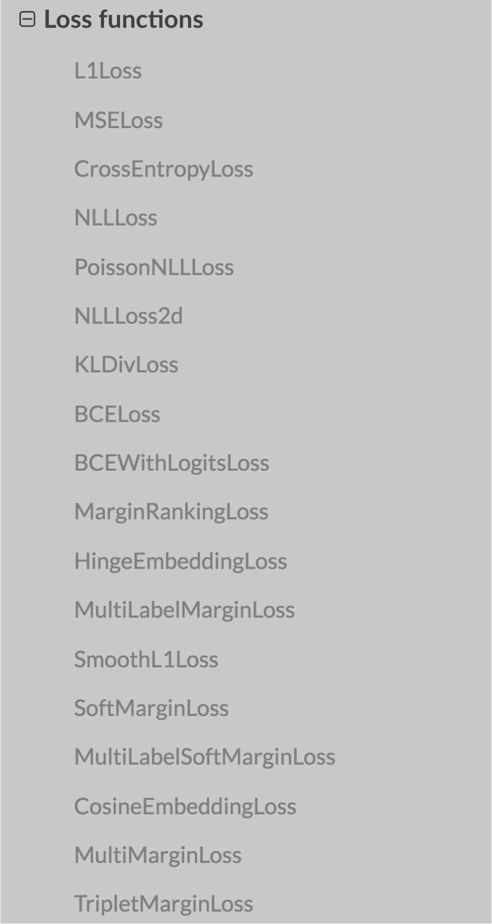


⊟ **Non-linear Activations**

- ReLU
- ReLU6
- ELU
- SELU
- PReLU
- LeakyReLU
- Threshold
- Hardtanh
- Sigmoid
- Tanh
- LogSigmoid
- Softplus
- Softshrink
- Softsign
- Tanhshrink
- Softmin
- Softmax
- Softmax2d
- LogSoftmax

http://pytorch.org/docs/master/nn.html

Loss functions



⊟ Loss functions

- L1Loss
- MSELoss
- CrossEntropyLoss
- NLLLoss
- PoissonNLLLoss
- NLLLoss2d
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss

http://pytorch.org/docs/master/nn.html

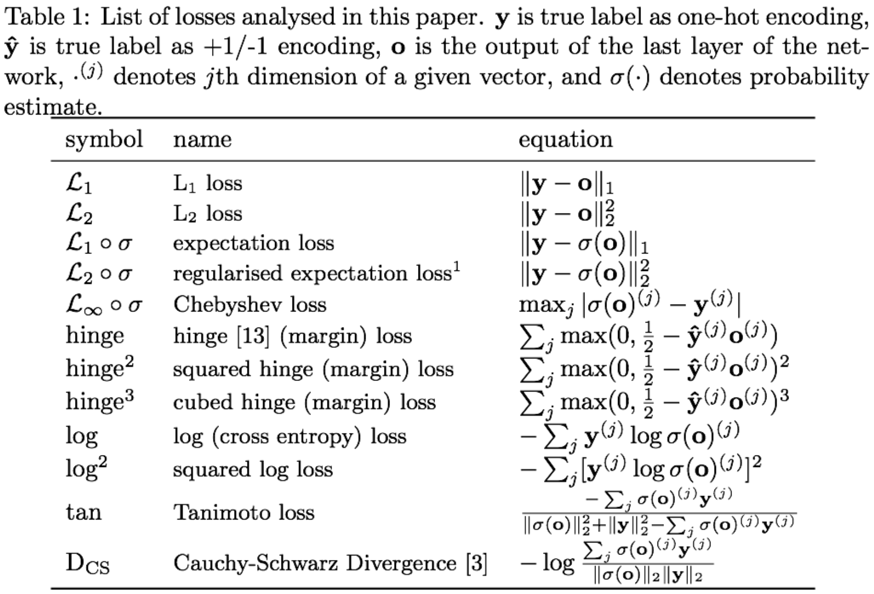


Table 1: List of losses analysed in this paper. $\mathbf{y}$ is true label as one-hot encoding, $\hat{\mathbf{y}}$ is true label as $+1/-1$ encoding, $\mathbf{o}$ is the output of the last layer of the network, $\cdot^{(j)}$ denotes jth dimension of a given vector, and $\sigma(\cdot)$ denotes probability estimate.

symbol	name	equation
$\mathcal{L}_1$	L_1 loss	$\lVert\mathbf{y}-\mathbf{o}\rVert_1$
$\mathcal{L}_2$	L_2 loss	$\lVert\mathbf{y}-\mathbf{o}\rVert_2^2$
$\mathcal{L}_1 \circ \sigma$	expectation loss	$\lVert\mathbf{y}-\sigma(\mathbf{o})\rVert_1$
$\mathcal{L}_2 \circ \sigma$	regularised expectation loss[1]	$\lVert\mathbf{y}-\sigma(\mathbf{o})\rVert_2^2$
$\mathcal{L}_\infty \circ \sigma$	Chebyshev loss	$\max_j \lvert\sigma(\mathbf{o})^{(j)}-\mathbf{y}^{(j)}\rvert$
hinge	hinge [13] (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})$
hinge^2	squared hinge (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^2$
hinge^3	cubed hinge (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^3$
log	log (cross entropy) loss	$-\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}$
log^2	squared log loss	$-\sum_j [\mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}]^2$
tan	Tanimoto loss	$\frac{-\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\lVert\sigma(\mathbf{o})\rVert_2^2+\lVert\mathbf{y}\rVert_2^2-\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}$
D_{CS}	Cauchy-Schwarz Divergence [3]	$-\log \frac{\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\lVert\sigma(\mathbf{o})\rVert_2\lVert\mathbf{y}\rVert_2}$

https://arxiv.org/pdf/1702.05659.pdf

torch.optim

- *class*torch.optim.Adadelta
- *class*torch.optim.Adagrad
- *class*torch.optim.Adam
- *class*torch.optim.Adamax
- *class*torch.optim.ASGD
- *class*torch.optim.RMSprop
- *class*torch.optim.Rprop
- *class*torch.optim.SGD

http://pytorch.org/docs/master/optim.html

Three simple steps

1. Design your model using class

2. Construct loss and optimizer
(select from PyTorch API)

3. Training cycle
(forward, backward, update)

Exercise 6-1

- Try different activation functions
  - Sigmoid to something else

- Try different optimizers

WHAT
NEXT

Lecture 7:
Wide and Deep