Spring 2024

CMPE 258-01 Deep Learning

Dr. Kaikai Liu, Ph.D. Associate Professor

Department of Computer Engineering

San Jose State University

Email: kaikai.liu@sjsu.edu

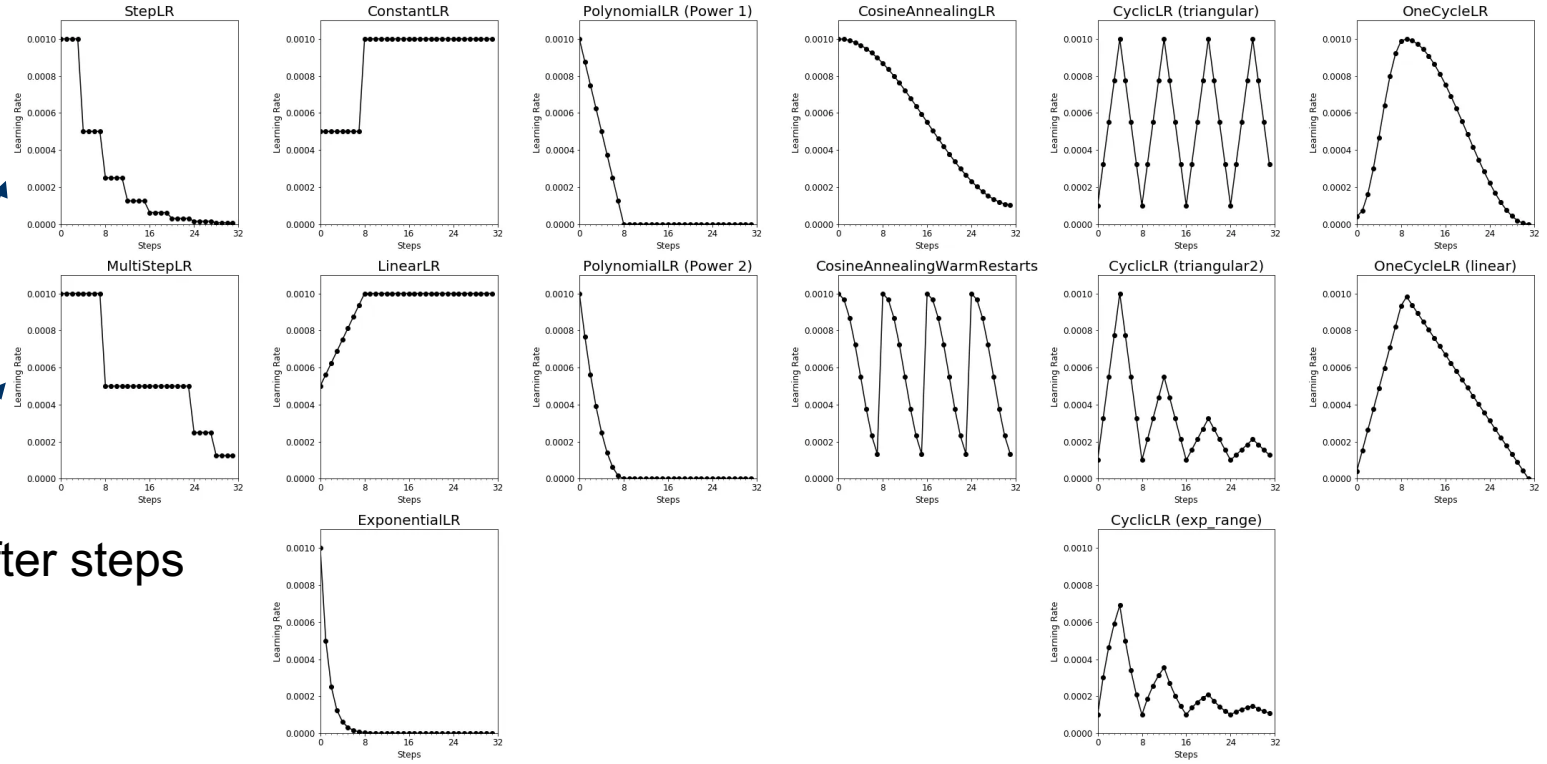
Website: https://www.sjsu.edu/cmpe/faculty/tenure-line/kaikai-liu.php

Homework1

- Design your own neural network and testing various training options for the provided sample code
  - https://github.com/lkk688/DeepDataMiningLearning/blob/main/DeepDataMiningLearning/symboldetection.ipynb
  - Goal: achieve better results (i.e., low BER)
  - Individual work
  - Due: 2/25 11:59PM

Torch Optim

- torch.optim.lr_scheduler provides several methods to adjust the learning rate based on the number of epochs.
  - https://pytorch.org/docs/stable/optim.html#module-torch.optim.lr_scheduler
  - Learning rate scheduling should be applied after optimizer's update

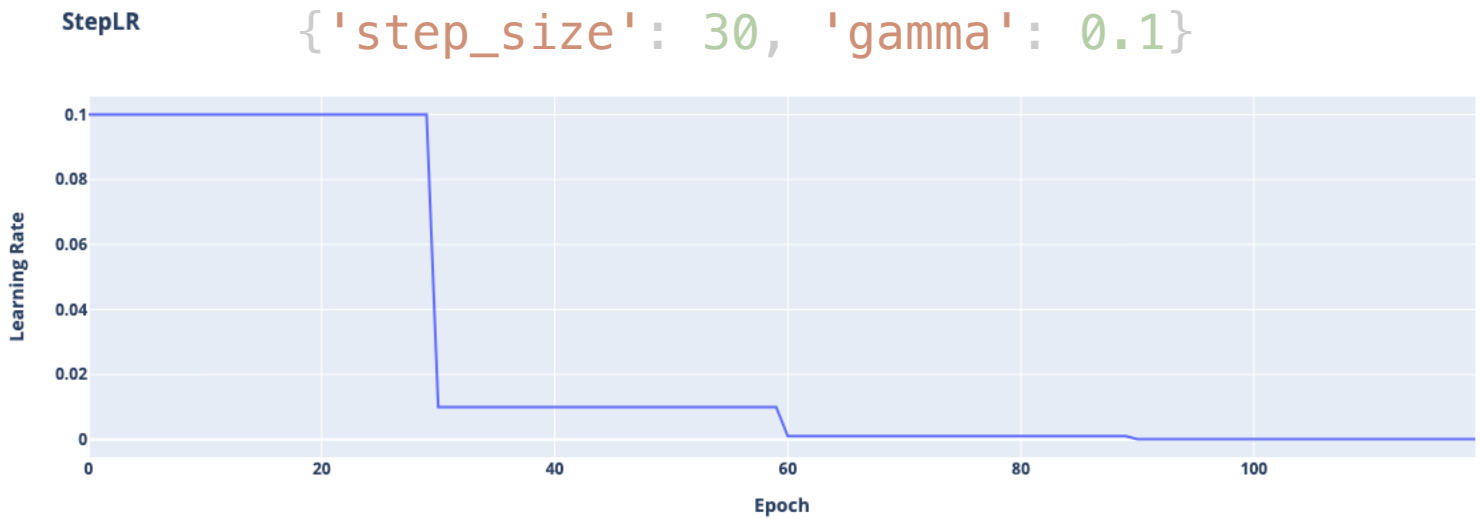


Decay gamma after steps

Torch Optim

- torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1).
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.StepLR.html

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **step_size** (*int*) – Period of learning rate decay.
- **gamma** (*float*) – Multiplicative factor of learning rate decay.
Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.



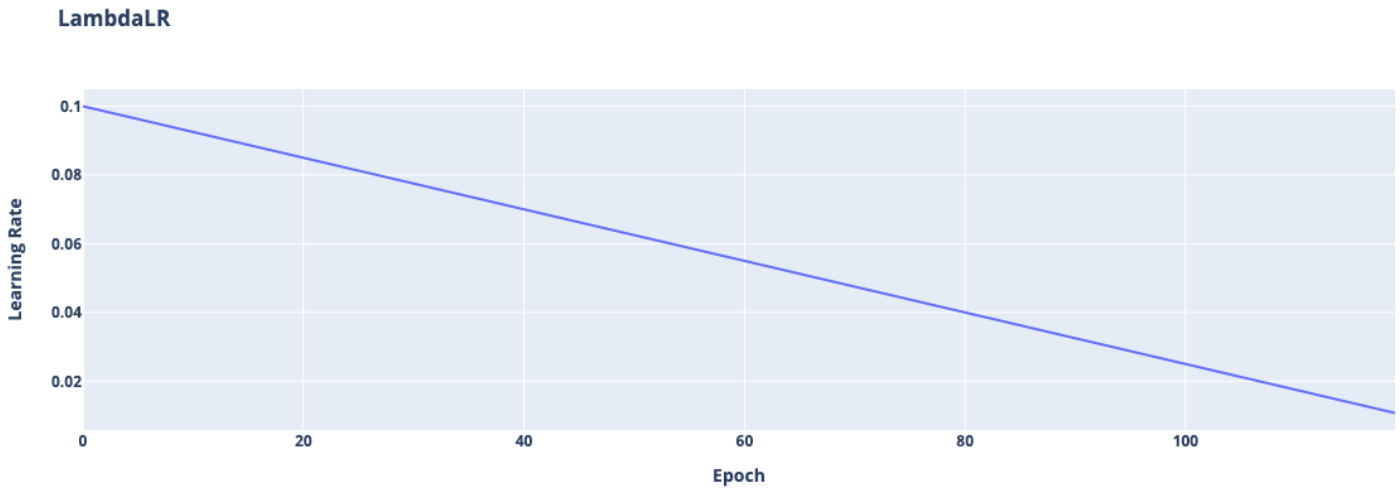
Torch Optim

- LambdaLR(optimizer, lr_lambda, last_epoch=-1)
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.LambdaLR.html#torch.optim.lr_scheduler.LambdaLR

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **lr_lambda** (*function or list*) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

```
final_lr = 0.01
initial_lr = 0.1
num_epochs = 120
```

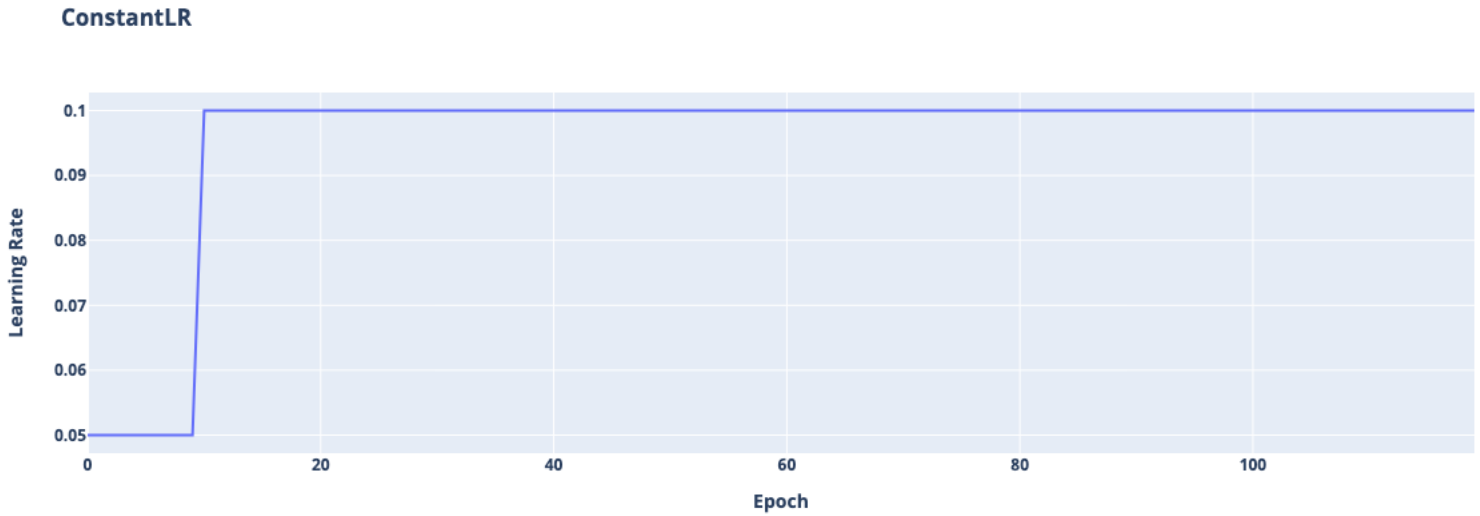
```
# Lambda function for learning rate decay
lambda_lr = lambda epoch: final_lr / initial_lr + (1 - epoch / num_epochs) * (1 - final_lr / initial_lr)
```



Torch Optim

- ConstantLR(self.opt, factor=0.5, total_iters=4)
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ConstantLR.html#torch.optim.lr_scheduler.ConstantLR
  - Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total_iters.

- **factor** (*float*) – The number we multiply learning rate until the milestone. Default: 1./3.
- **total_iters** (*int*) – The number of steps that the scheduler decays the learning rate. Default: 5.

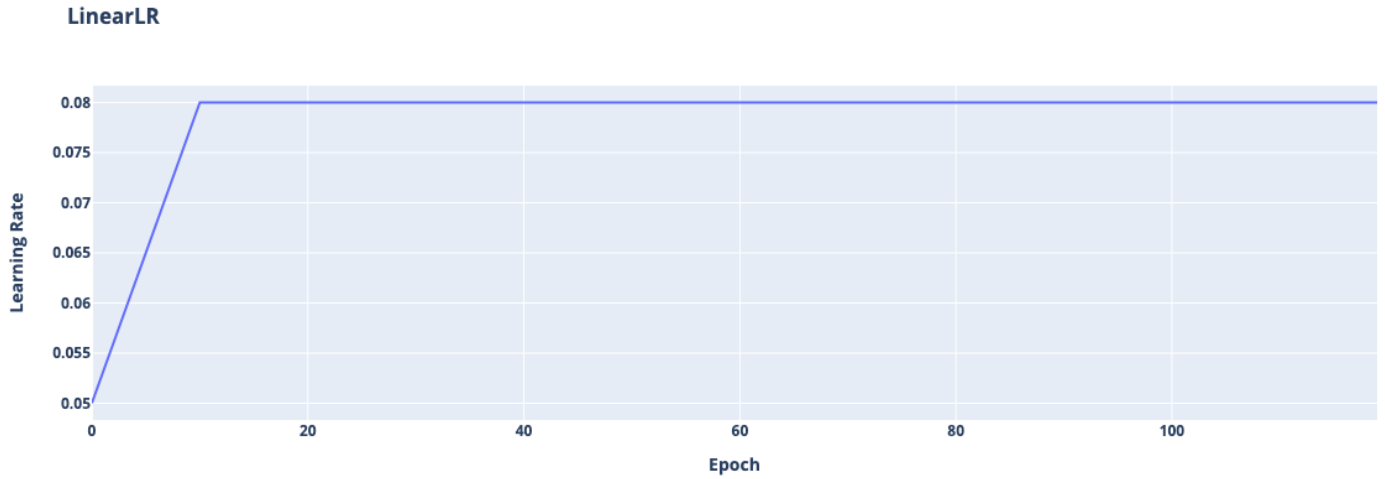


Torch Optim

- LinearLR
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.LinearLR.html#torch.optim.lr_scheduler.LinearLR

- **start_factor** (*float*) – The number we multiply learning rate in the first epoch. The multiplication factor changes towards end_factor in the following epochs. Default: 1./3.
- **end_factor** (*float*) – The number we multiply learning rate at the end of linear changing process. Default: 1.0.
- **total_iters** (*int*) – The number of iterations that multiplicative factor reaches to 1. Default: 5.

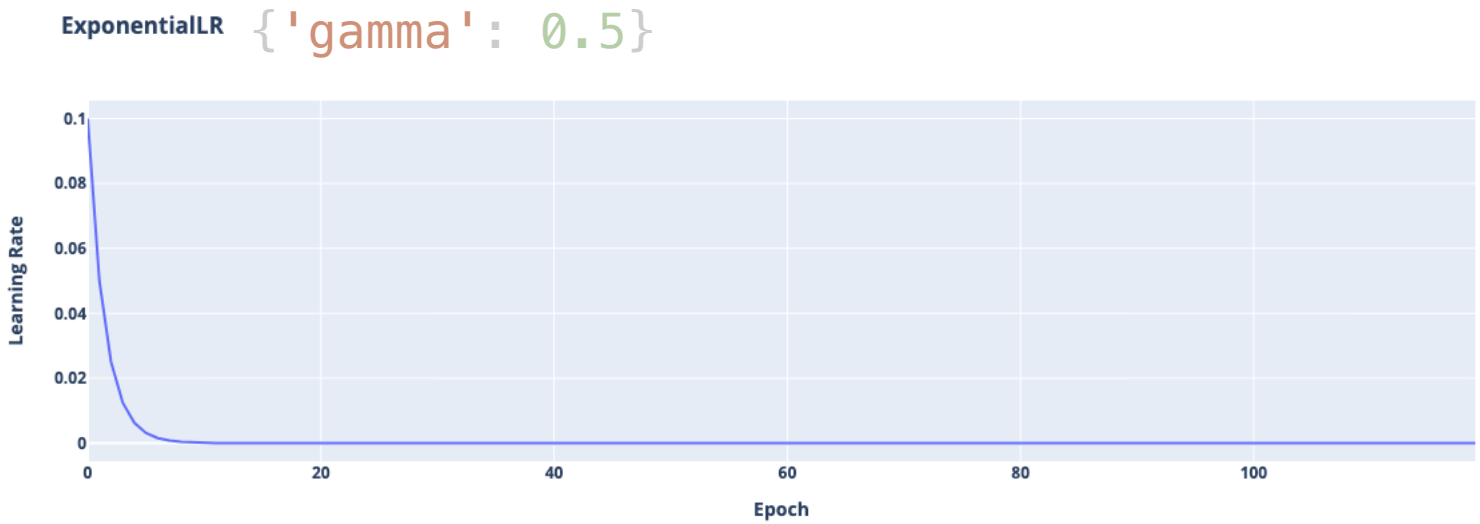
```
{'start_factor': 0.5, 'end_factor':
0.8, 'total_iters':10}
```



Torch Optim

- ExponentialLR
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ExponentialLR.html#torch.optim.lr_scheduler.ExponentialLR
  - Decays the learning rate of each parameter group by gamma every epoch. When last_epoch=-1, sets initial lr as lr.

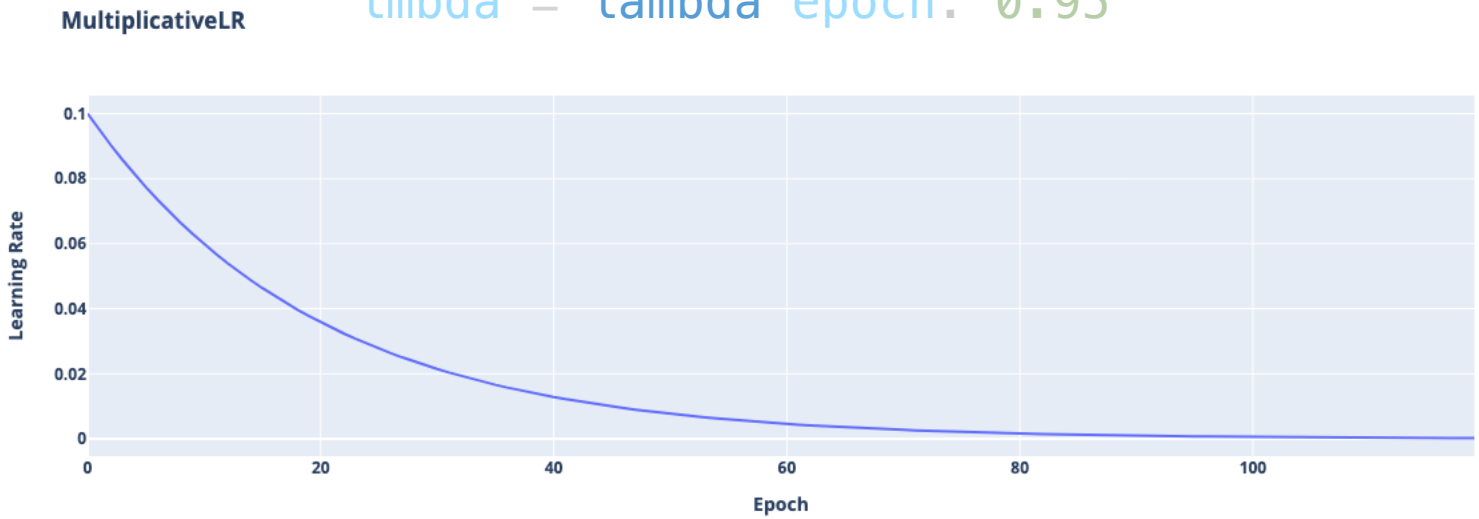
- **gamma** (*float*) – Multiplicative factor of learning rate decay.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.



Torch Optim

- MultiplicativeLR(optimizer, lr_lambda=lmbda)
  - https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.MultiplicativeLR.html#torch.optim.lr_scheduler.MultiplicativeLR
  - lr_lambda (function or list) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.

```
lmbda = lambda epoch: 0.95
```



Torch Loss Function

- torch.nn.L1Loss: The L1 loss function computes the mean absolute error between each value in the predicted and target tensor. It computes the sum of all the values returned from each absolute difference computation and takes the average of this sum value to obtain the Mean Absolute Error (MAE).

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

- torch.nn.SmoothL1Loss: The smooth L1 loss function combines the benefits of MSE loss and MAE loss through a heuristic value beta. It uses a squared term if the absolute error falls below one and an absolute term otherwise. It is less sensitive to outliers than the mean square error loss and, in some cases, prevents exploding gradients.

$$l_n = \begin{cases} 0.5(x_n - y_n)^2 / beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$

If do square for high values may result in exploding gradients

Torch Loss Function

- Mean Squared Error Loss(MSE): torch.nn.MSELoss
  - The Mean Square Error shares similarities with the Mean Absolute Error. It computes the square difference between values in the prediction tensor and that of the target tensor.
  - By doing so, relatively significant differences are penalized more, while relatively minor differences are penalized less. MSE is considered less robust at handling outliers and noise than MAE.

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

```
loss_fn = nn.L1Loss(size_average=None, reduce=None,
reduction='mean')

loss = nn.SmoothL1Loss()

loss = nn.MSELoss(size_average=None,
reduce=None, reduction='mean')
```

Torch CrossEntropy

- Cross-entropy (torch.nn.CrossEntropyLoss) as a loss function is used to learn the probability distribution of the data. While other loss functions like squared loss penalize wrong predictions, cross-entropy gives a more significant penalty when incorrect predictions are predicted with high confidence.
  - CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)
  - Computes the cross entropy loss between input logits and target, e.g., classification problem with C classes.
  - The **input** is expected to contain the unnormalized logits for each class (which do not need to be positive or sum to 1, in general). input has to be a Tensor of size (C) for unbatched input, (N,C) or (N,C, d1,d2, ...dk) for the K-dimensional case (e.g., computing cross entropy loss per-pixel for 2D images), N: number of batch.
  - The **target** that this criterion expects should contain either: Class indices in the range [0,C). if ignore_index is specified, this loss also accepts this class index (this index may not necessarily be in the class range). If containing class indices, the target shape should be: shape (), (N) or (N, d1,d2,...dk)

Torch CrossEntropy

- Computes the cross entropy loss between input logits and target
  - CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)
  - Input: unnormalized logits for each class (N,C, d1,d2, ...dk), N: number of batch.
  - The target: Class indices in the range [0,C), e.g., (N, d1,d2,...dk)
  - If provided, the optional argument weight should be a 1D Tensor assigning weight to each of the classes: useful when you have an unbalanced training set.
  - The unreduced (i.e. with reduction set to 'none') loss

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

  - If reduction is not 'none' (default 'mean'), then

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

Torch CrossEntropy

- Computes the cross entropy loss between input logits and target
  - CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)

```
loss = nn.CrossEntropyLoss()
N = 4 # batch size
C = 5 # number of classes
input = torch.randn(N, C, requires_grad=True)
target = torch.empty(N, dtype=torch.long).random_(C)
Target tensor([3, 0, 2, 0])
output = loss(input, target)
output.backward()
print(output) #tensor(1.6370, grad_fn=)
```

Torch Loss Function

- Negative Log-Likelihood Loss: torch.nn.NLLLoss
  - CLASS torch.nn.NLLLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')
  - The negative log likelihood loss. It is useful to train a classification problem with C classes.
  - The input given through a forward call is expected to contain log-probabilities of each class. input has to be a Tensor of size either (N, C) or (N, C, d1, d2,..dk). The latter is useful for higher dimension inputs, such as computing NLL loss per-pixel for 2D images.
  - The target that this loss expects should be a class index in the range [0, C-1]

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$ reduction set to 'none'

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & \text{if reduction} = \text{‘mean’;} \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{‘sum’.} \end{cases}$$

Torch Loss Function

- Negative Log-Likelihood Loss: torch.nn.NLLLoss
  - CLASS torch.nn.NLLLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')
  - Obtaining log-probabilities in a neural network is easily achieved by adding a LogSoftmax layer in the last layer of your network. You may use CrossEntropyLoss instead, if you prefer not to add an extra layer.
  - The negative expected value of the log probabilities is the information entropy of an event. Likelihoods are often transformed to the log scale, and can be interpreted as the degree to which an event supports a statistical model.
  - Cross-entropy also penalizes wrong but confident predictions and correct but less confident predictions. In contrast, negative log loss does not penalize according to the confidence of predictions.

Torch Loss Function

- Negative Log-Likelihood Loss: torch.nn.NLLLoss
  - CLASS torch.nn.NLLLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')

```
m = nn.LogSoftmax(dim=1)
loss = nn.NLLLoss()
# input is of size N x C = 3 x 5
N, C = 3, 5
input = torch.randn(N, C,
requires_grad=True)
y = m(input)
target = torch.tensor([1, 0, 4]
output = loss(y, target)
```

```
N, Num_C = 5, 4 #Num_C number of classes
loss = nn.NLLLoss()
# input is of size N x C x height x width
C, H, W = 3, 10, 10 #image size CHW
K = 3
data = torch.randn(N, C, H, W)
conv = nn.Conv2d(C, Num_C, (K, K))
m = nn.LogSoftmax(dim=1)
y = m(conv(data)) #N, Num_C, Hout, Wout
torch.Size([5, 4, 8, 8])
target = torch.empty(N, Hout, Wout,
dtype=torch.long).random_(0, C)
torch.Size([5, 8, 8])
output = loss(y, target)
```

Torch Loss Function

- Binary Cross-Entropy loss
  - torch.nn.BCELoss is a particular class of Cross-Entropy losses used for the unique problem of classifying data points into only two classes. Labels for this type of problem are usually binary, and our goal is to push the model to predict a number close to zero for a zero label and a number close to one for one label.
  - Creates a criterion that measures the Binary Cross Entropy between the target (targets *y* should be numbers between 0 and 1) and the input probabilities
  - Usually, when using BCE loss for binary classification problems, the neural network's output is a Sigmoid layer to ensure that the output is either a value close to zero or a value close to one.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$ reduction set to 'none'

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{‘mean’;} \\ \text{sum}(L), & \text{if reduction} = \text{‘sum’.} \end{cases}$$

Torch Loss Function

- Binary Cross-Entropy loss
  - CLASS torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')
  - Input: (∗), where ∗ means any number of dimensions.
  - Target: (∗), same shape as the input.

```
m = nn.Sigmoid()
loss = nn.BCELoss()
input = torch.randn(3, requires_grad=True)
target = torch.empty(3).random_(2)
y = m(input)
y.size()

output = loss(y, target)
print(output)
```

```
N, C = 5, 4
data = torch.randn(N, C)
m = nn.Sigmoid()
y = m(data) #N, C
y.size()

loss = nn.BCELoss()
target = torch.empty(N, C).random_(0, 2)

output = loss(y, target)
print(output)
```

Torch Loss Function

- Binary Cross-Entropy Loss with Logits
  - It adds a Sigmoid layer and the BCELoss in one single class. It provides numerical stability for log-sum-exp. It is more numerically stable than a plain Sigmoid followed by a BCELoss.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right], \quad \text{reduction set to 'none'}$$

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

It's possible to trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$\ell_c(x, y) = L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c})) \right],$$

where c is the class number ($c > 1$ for multi-label binary classification, $c = 1$ for single-label binary classification), n is the number of the sample in the batch and p_c is the weight of the positive answer for the class c.

$p_c > 1$ increases the recall, $p_c < 1$ increases the precision.

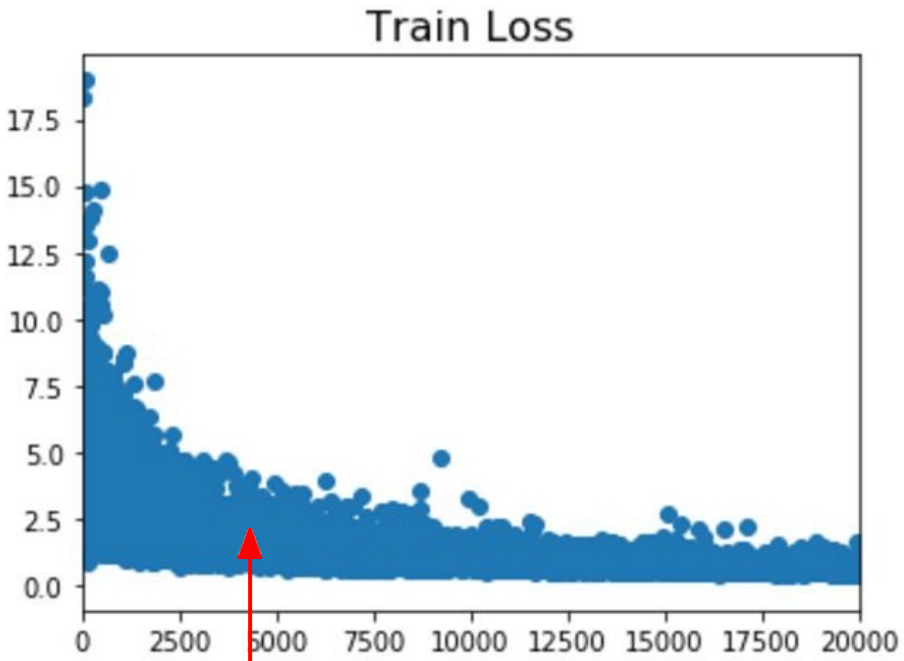
Torch Loss Function

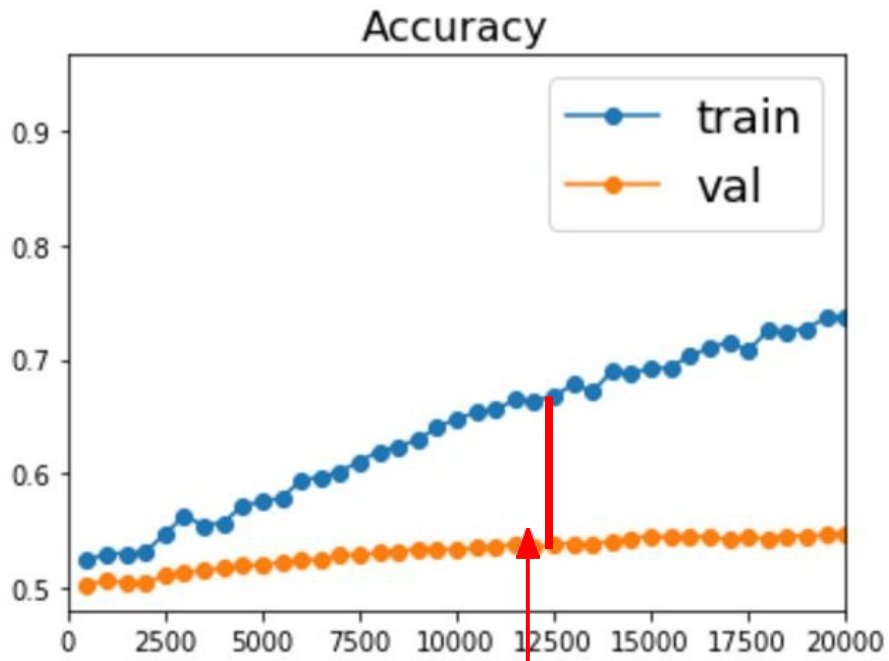
- Binary Cross-Entropy Loss with Logits
  - It adds a Sigmoid layer and the BCELoss in one single class. It provides numerical stability for log-sum-exp. It is more numerically stable than a plain Sigmoid followed by a BCELoss.

```
N, C = 4, 3 # 3 classes, batch size = 4
target = torch.ones([N, C], dtype=torch.float32)
output = torch.full([N, C], 1.5) # A prediction (logit)
pos_weight = torch.ones([C]) # All weights are equal to 1

criterion = torch.nn.BCEWithLogitsLoss(pos_weight=pos_weight)
loss = criterion(output, target) # -log(sigmoid(1.5))
print(loss) #tensor(0.2014)
```

Beyond Training Error



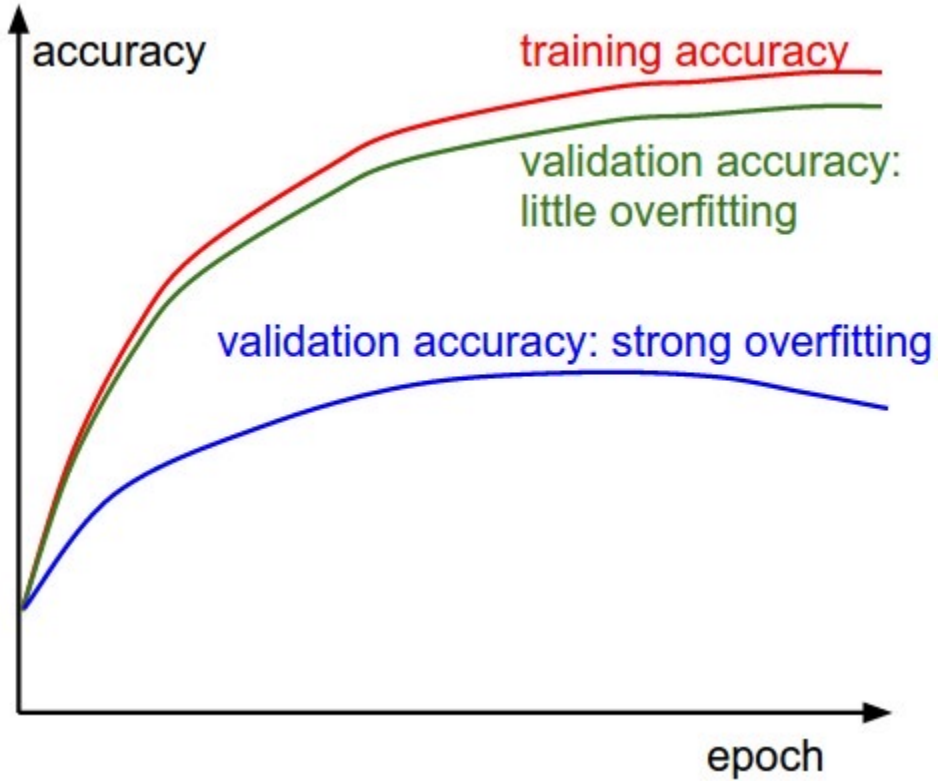


Better optimization algorithms help reduce training loss

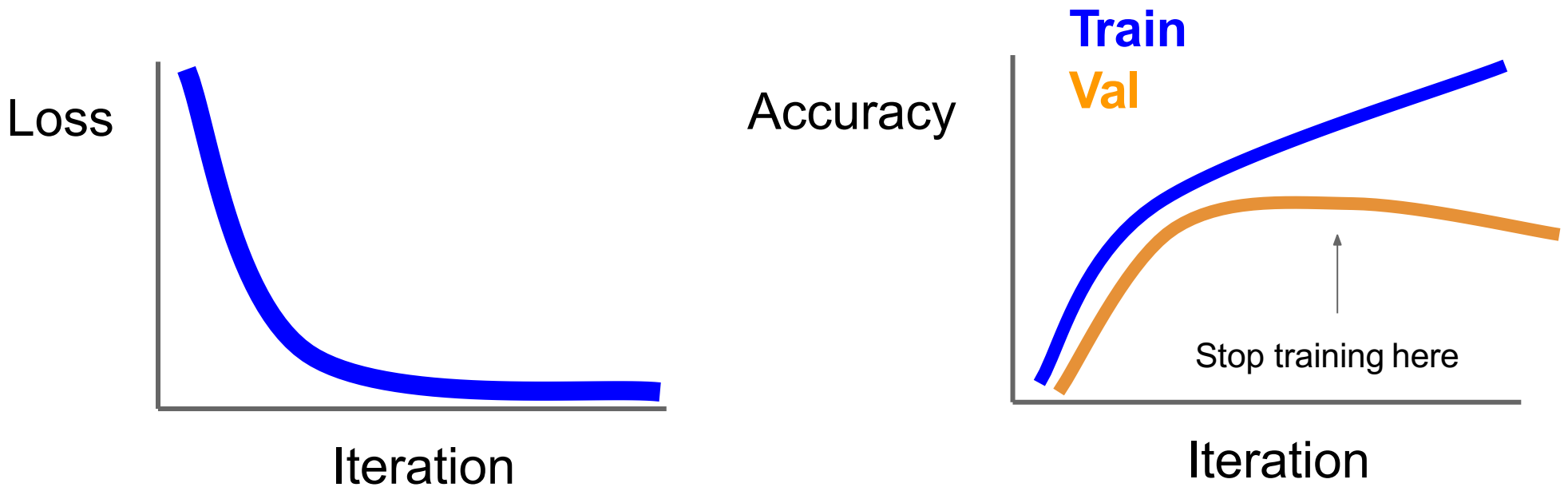
But we really care about error on new data - how to reduce the gap?

Overfitting

- The gap between the training and validation accuracy indicates the amount of overfitting.

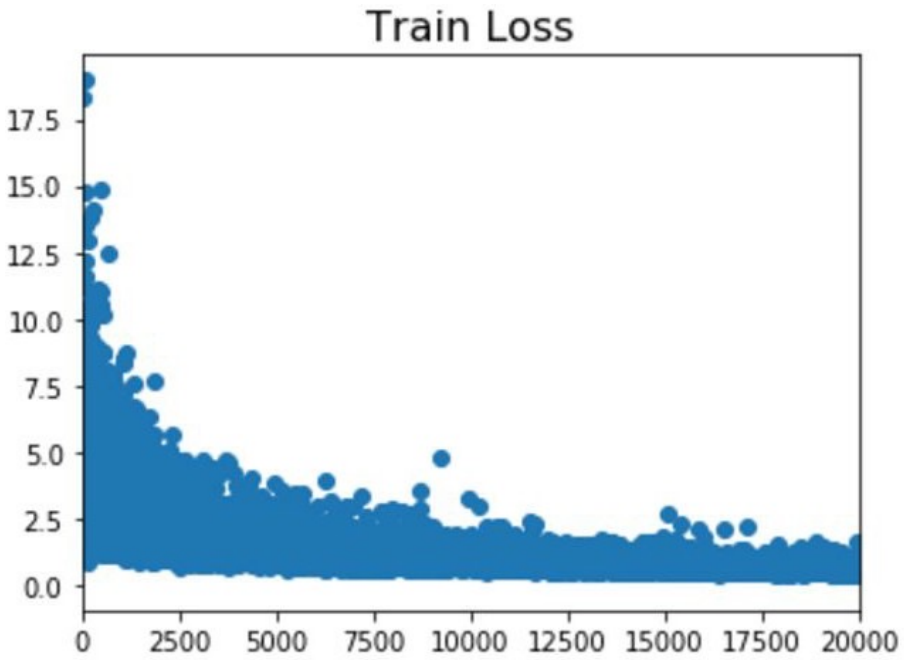


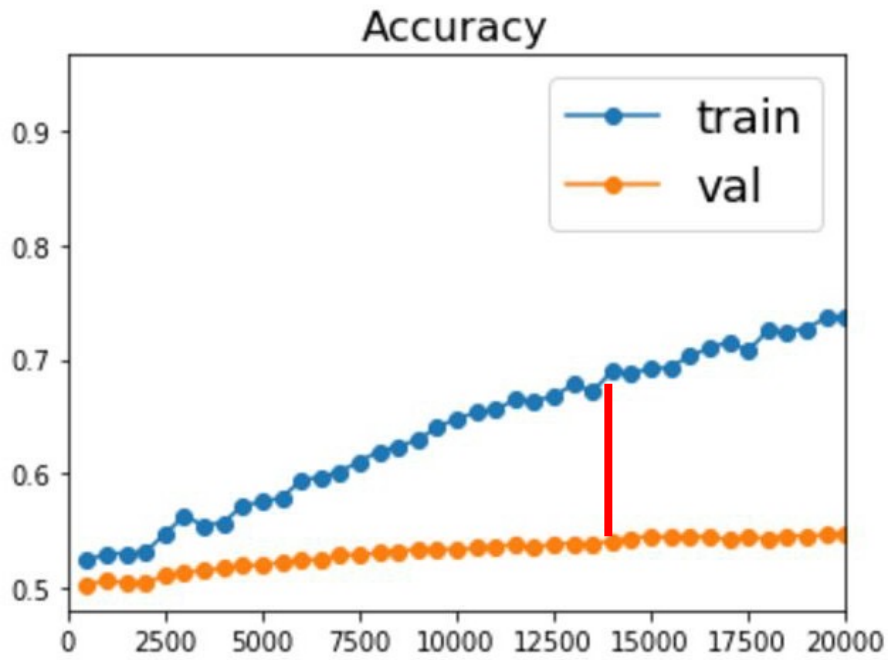
Early Stopping: Always do this



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val

How to improve single-model performance?





Regularization

Bias-Variance Tradeoff

- We have minimized the error (loss) with respect to **training data**
  - Low training error does not imply good expected performance: **over-fitting**
- We would like to reason about the **expected loss (Prediction Risk)** over:
  - Training Data: $\{(y_1, x_1), \ldots, (y_n, x_n)\}$
  - Test point: (y_*, x_*)
- We will decompose the expected loss into:

$$\mathbf{E}_{D,(y_*,x_*)}\left[(y_* - f(x_*|D))^2\right] = \text{Noise} + \text{Bias}^2 + \text{Variance}$$

Bias-Variance Tradeoff

- Expanding on the model estimation error:

$$\mathbf{E}_D\left[(h(x_*) - f(x_*|D))^2\right]$$

- Completing the squares with $\mathbf{E}\left[f(x_*|D)\right] = \bar{f}_*$

$$\mathbf{E}_D\left[(h(x_*) - f(x_*|D))^2\right]$$
$$= \underbrace{(h(x_*) - \mathbf{E}\left[f(x_*|D)\right])^2}_{\text{(Bias)}^2} + \underbrace{\mathbf{E}\left[(f(x_*|D) - \mathbf{E}\left[f(x_*|D)\right])^2\right]}_{\text{Variance}}$$

- Tradeoff between bias and variance:
  - **Simple Models:** High Bias, Low Variance
  - **Complex Models:** Low Bias, High Variance

- Choice of models balances bias and variance.
  - Over-fitting ➔ Variance is too High
  - Under-fitting ➔ Bias is too High

Bias Variance Plot

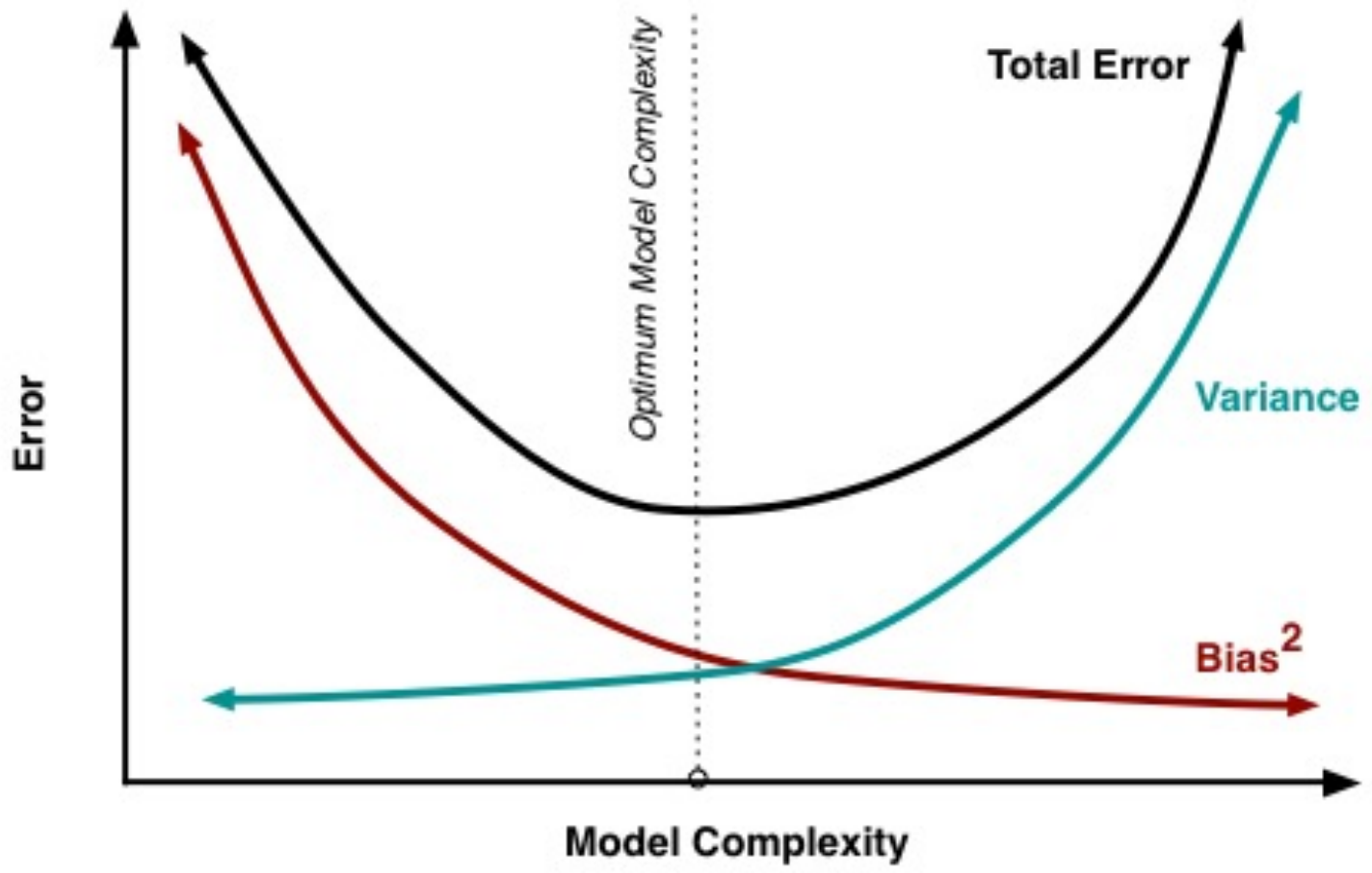


Image from http://scott.fortmann-roe.com/docs/BiasVariance.html

Regularization

- The bias error is an error from wrong assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs. This is called **underfitting**.
- The variance is an error from sensitivity to small fluctuations in the training set. High variance may result in modeling the random noise in the training data. This is called **overfitting**.
- The bias-variance tradeoff is a term to describe the fact that we can reduce the variance by increasing the bias. Good regularization techniques strive to simultaneously minimize the two sources of error. Hence, achieving better generalization.

Regularization

- The most common family of approaches used before the Deep Learning era in estimators such as linear and logistic regression, are parameters norm penalties. Here we add a parameter norm penalty Ω(θ) to the loss function

$$J'(\theta; X, y) = J(\theta; X, y) + a\Omega(\theta)$$

  - where θ denotes the trainable parameters, X the input, and y and target labels.
  - a is a hyperparameter that weights the contribution of the norm penalty, hence the effect of the regularization.
  - Two popular methods: L1 and L2 regularization

Regularization

- L2 regularization, also known as weight decay or ridge regression, adds a norm penalty

The loss function: $J'(w; X, y) = J(w; X, y) + \frac{a}{2}||w||_2^2$

compute the gradients: $\nabla_w J'(w; X, y) = \nabla_w J(w; X, y) + aw$

single training step and a learning rate λ: $w = (1 - \lambda a)w - \lambda \nabla_w J(w; X, y)$

  - The equation effectively shows us that each weight of the weight vector will be reduced by a constant factor on each training step.
  - Note here that we replaced θ with w. This was due to the fact that usually we regularize only the actual weights of the network and not the biases b.
  - As a result, we reduce the variance of our model, which makes it easier to generalize on unseen data.

Regularization

- L1 regularization chooses a norm penalty

$$\Omega(\theta) = ||w||_1 = \sum_i |w_i|.$$

- The L1 regularizer introduces sparsity in the weights by forcing more weights to be zero instead of reducing the average magnitude of all weights (as the L2 regularizer does). In other words, L1 suggests that some features should be discarded whatsoever from the training process.
  - The derivative of L_2 is 2 * *weight*.
    - forces the weights to be small but does not make them zero and does non sparse solution
  - The derivative of L_1 is *k* (a constant, whose value is independent of weight).
    - Can push weights to zero, removing some feature altogether. This works well for **feature selection** in case we have a huge number of features.

Regularization: reduce overfit

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use: **L2 regularization** L1 regularization
Elastic net (L1 + L2)

$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad \text{(Weight decay)}$$

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

Weight Decay

- L2 regularization, also known as weight decay or ridge regression
- CLASStorch.optim.SGD
  (*params*, *lr=0.001*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, ***, *maximize=False*, *foreach=None*, *differentiable=False*)
  - **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- Manually Add L1 and L2 in loss function

Label smoothing

- Noise injection is one of the most powerful regularization strategies. By adding randomness, we can reduce the variance of the models and lower the generalization error.
- **Label smoothing** is a way of adding noise at the output targets, aka labels.
  - Let's assume that we have a classification problem. In most of them, we use a form of cross-entropy loss and softmax to output the final probabilities.
  - The target vector has the form of [0, 1 , 0 , 0]. Because of the way softmax is formulated, it can never achieve an output of 1 or 0. As a result, the model will continue to be trained, pushing the output values as high and as low as possible. The model will never converge. That, of course, will cause overfitting.
  - To address that, label smoothing replaces the hard 0 and 1 targets by a small margin. where k is the number of classes.

1: $1 - \epsilon$, 0: $\frac{\epsilon}{k-1}$

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

Dropout

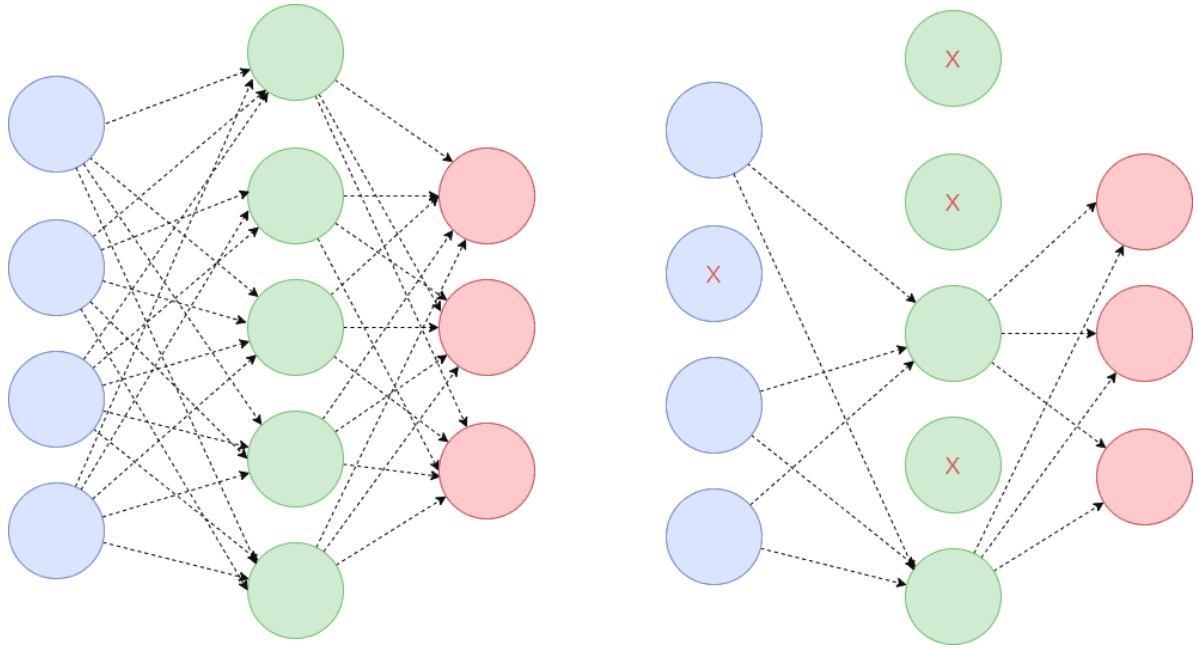
- Another strategy to regularize deep neural networks is dropout. Dropout falls into noise injection techniques and can be seen as noise injection into the hidden units of the network.
- In training, some number of layer outputs are randomly ignored (dropped out) with probability p.
- During test time, all units are present, but they have been scaled down by p.
  - This is happening because after dropout, the next layers will receive lower values. In the test phase though, we are keeping all units so the values will be a lot higher than expected. That's why we need to scale them down.
- By using dropout, the same layer will alter its connectivity and will search for alternative paths to convey the information in the next layer. As a result, each update to a layer during training is performed with a different "view" of the configured layer. Conceptually, it approximates training a large number of neural networks with different architectures in parallel.

Dropout

- "Dropping" values means temporarily removing them from the network for the current forward pass, along with all its incoming and outgoing connections. Dropout has the effect of making the training process noisy. The choice of the probability p depends on the architecture.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

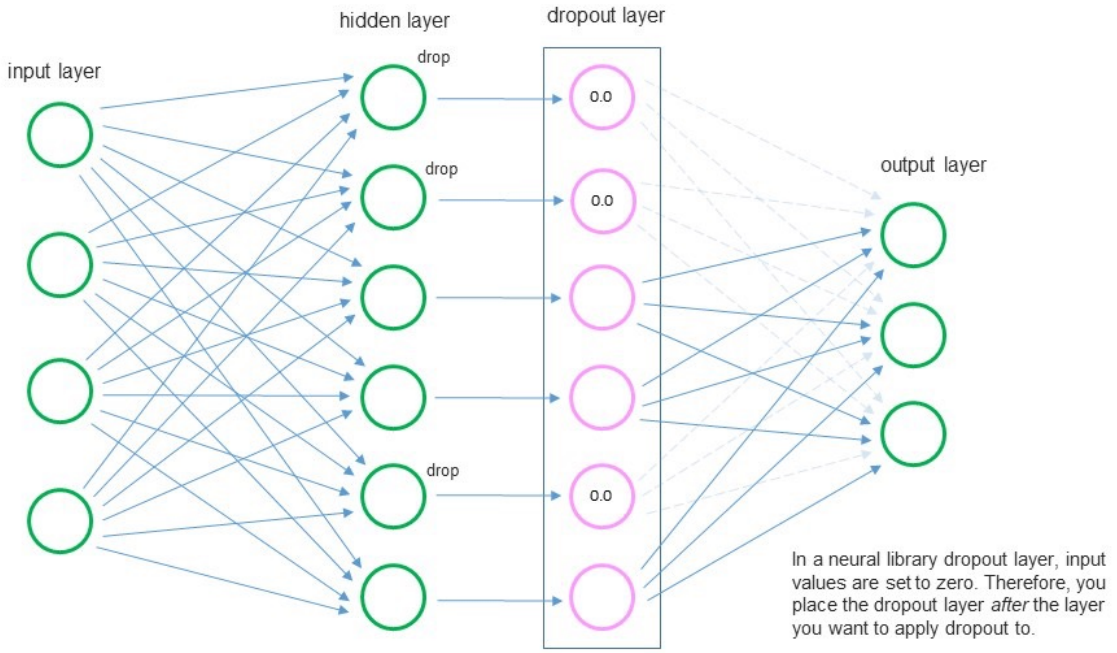


Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR2014

https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

Dropout

- Dropout
  - Dropout is a solution proposed to this problem by Nitish Srivastava, Geoffrey Hinton and few other students at the University of Toronto in 2012. Hinton is now an employee at Google, leading to the giant picking up the patent for the technology.
    - https://patents.google.com/patent/WO2014105866A1/en



OPINIONS

Google Activates 'Dropout' Patent, Neural Network Engineers To Be On Alert?

ANIRUDH VK - 3 WEEKS AGO

Thank You

Address:
ENG257, SJSU

Email Address:
Kaikai.liu@sjsu.edu