# Recitation 3 Part 2

Shentong Mo

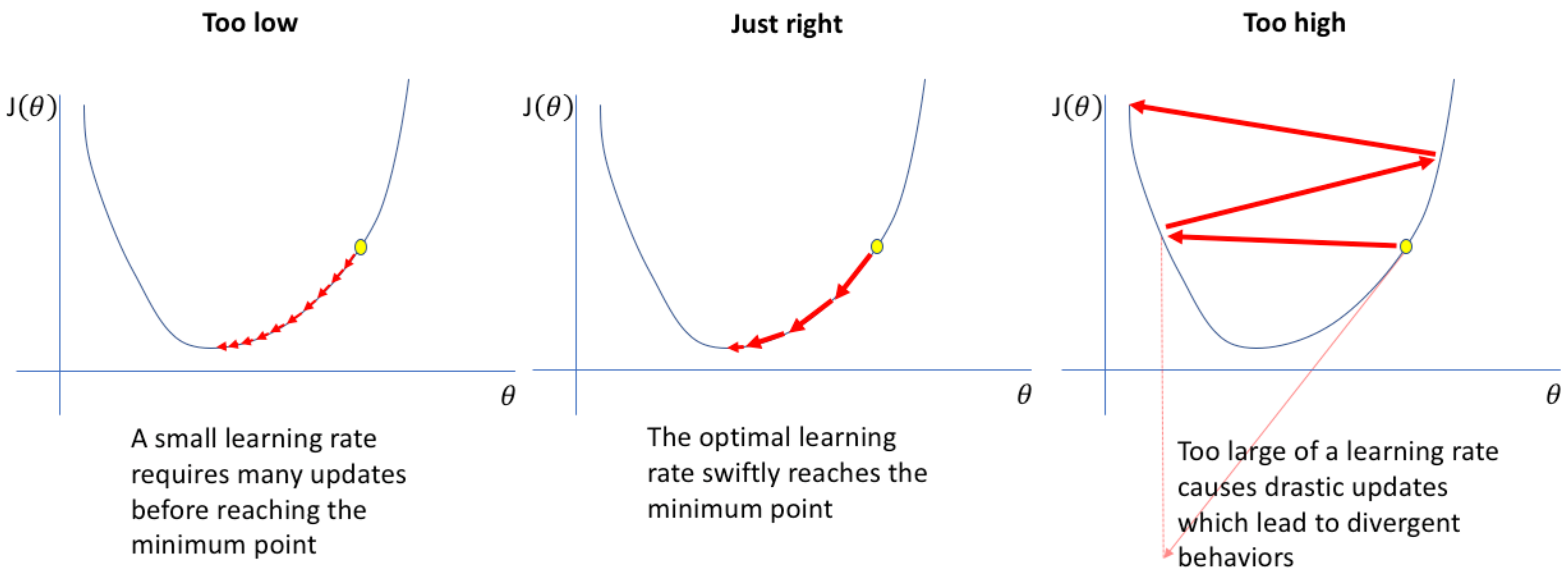
- **Learning Rate Scheduling**
- **Regularization**
- **Common Pitfalls**
- **Tips for HW1P2**

# Annealing Your Learning Rate

1. **Among all the hyper parameters existing in neural network, (personally speaking -.- ) learning rate is one of the most important hyper parameter that affects your training performance.**

2. **It is usually helpful to anneal the learning rate over time. (Increase of Decrease)**

3. **Tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can.**



Too low

A small learning rate requires many updates before reaching the minimum point

Just right

The optimal learning rate swiftly reaches the minimum point

Too high

Too large of a learning rate causes drastic updates which lead to divergent behaviors

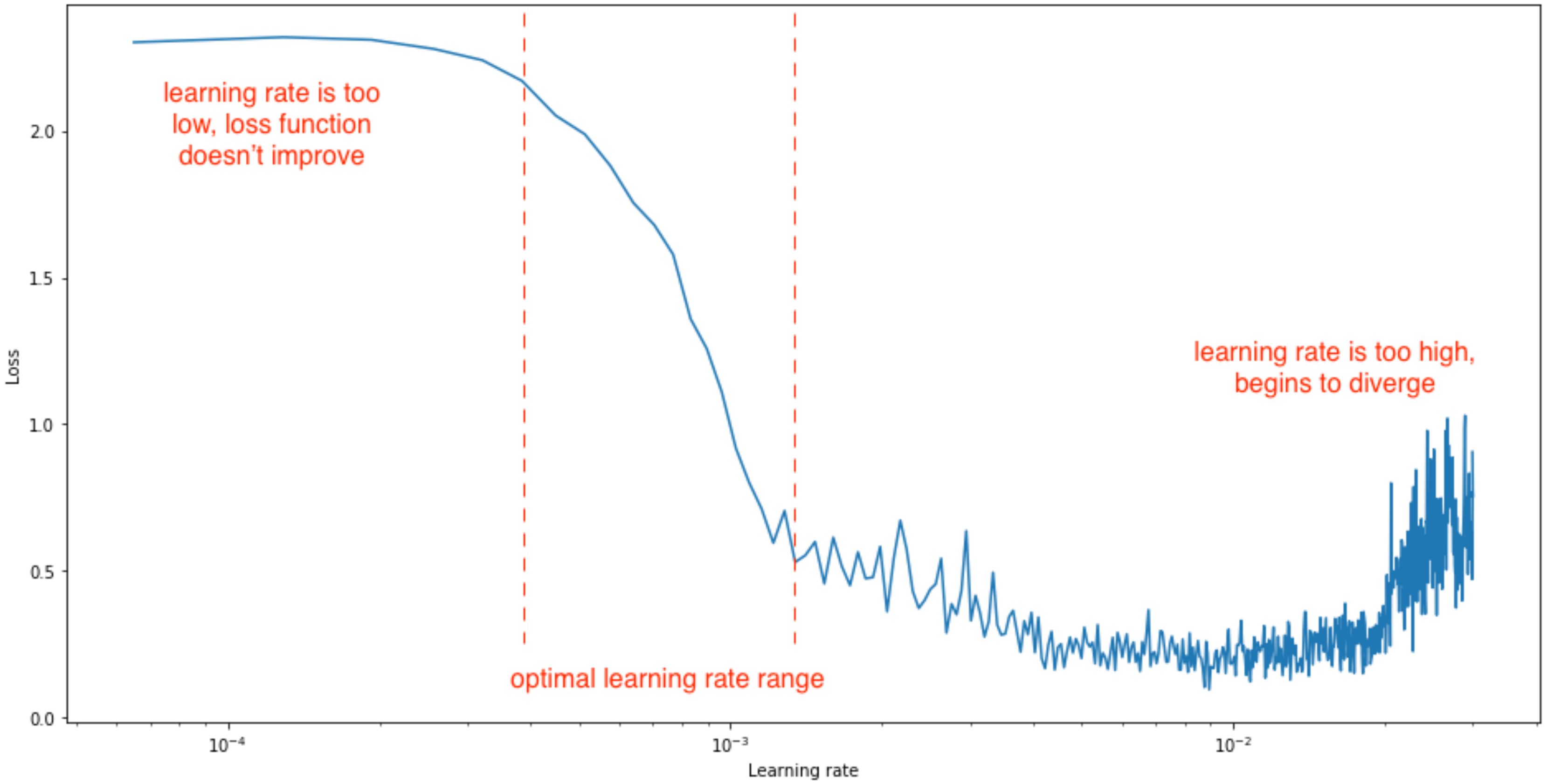
**source: https://www.jeremyjordan.me/nn-learning-rate/**

# Three common used methods

(to the best of my knowledge -.- )

- **Step Decay:** Reduce the learning rate by a factor every few epochs. Typical values might be reducing the learning rate by 0.5 every *X* epochs, or by 0.1 every *Y* epochs. *X, Y* depend heavily on the type of problem and the performance of your model.

- **(ReduceOnPlateau)** One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.



source: https://www.jeremyjordan.me/nn-learning-rate/

# Three common used methods

(to the best of my knowledge -.- )

- **Exponential Decay: Reduce the learning rate according to the following mathematical form**

$$\alpha = \alpha_0 e^{-kt}$$

**where *k* and *e* are hyper parameters and *t* is the epoch number.**

- **1/*t* Decay: Reduce the learning rate according to the following mathematical form**

$$\alpha = \alpha_0/(1 + kt)$$

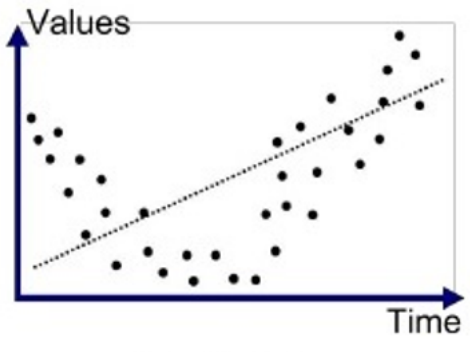
**where *t* is the epoch number.**

★ **Step decay is slightly preferable because the hyper-parameters it involves are more interpretable than the hyper-parameter *k***

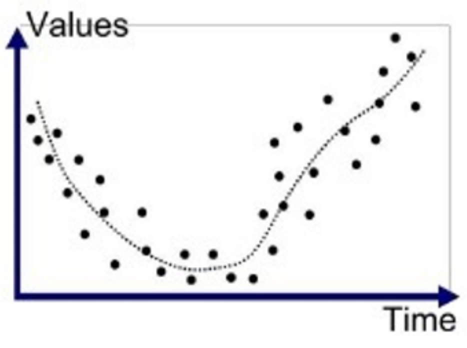
source: https://www.jeremyjordan.me/nn-learning-rate/

# Regularization

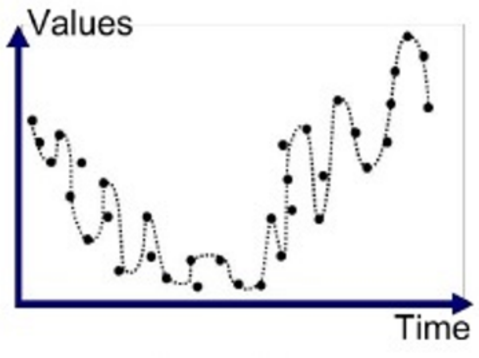
**Overfitting** is a modeling error that occurs when a function is too closely fit to a limited set of data points.



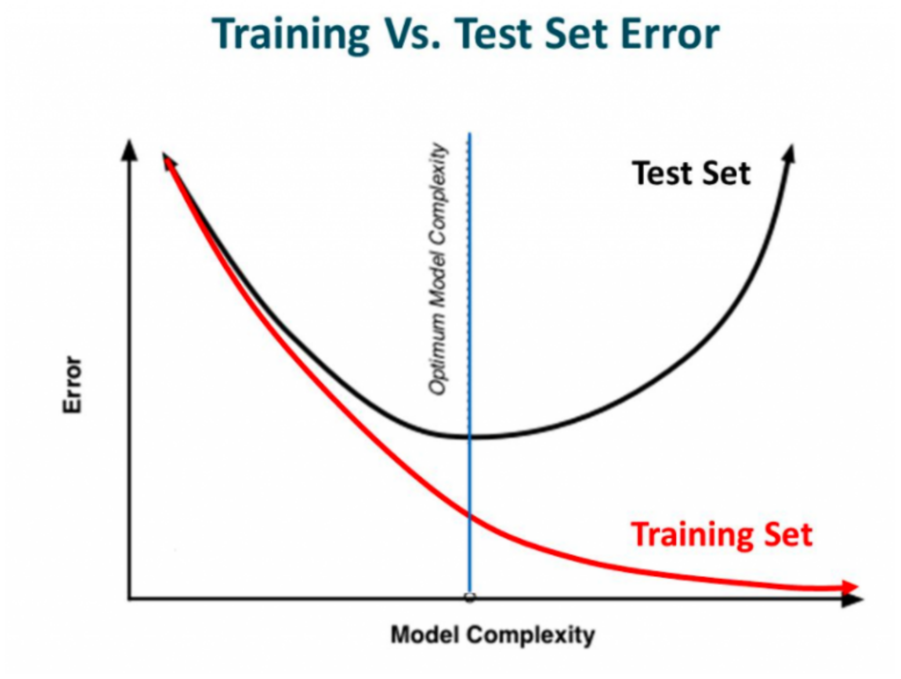
Underfitted



Good Fit/Robust

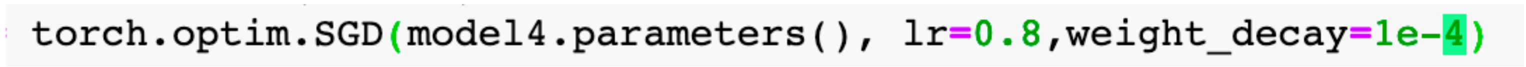


Overfitted



# Regularization in NN

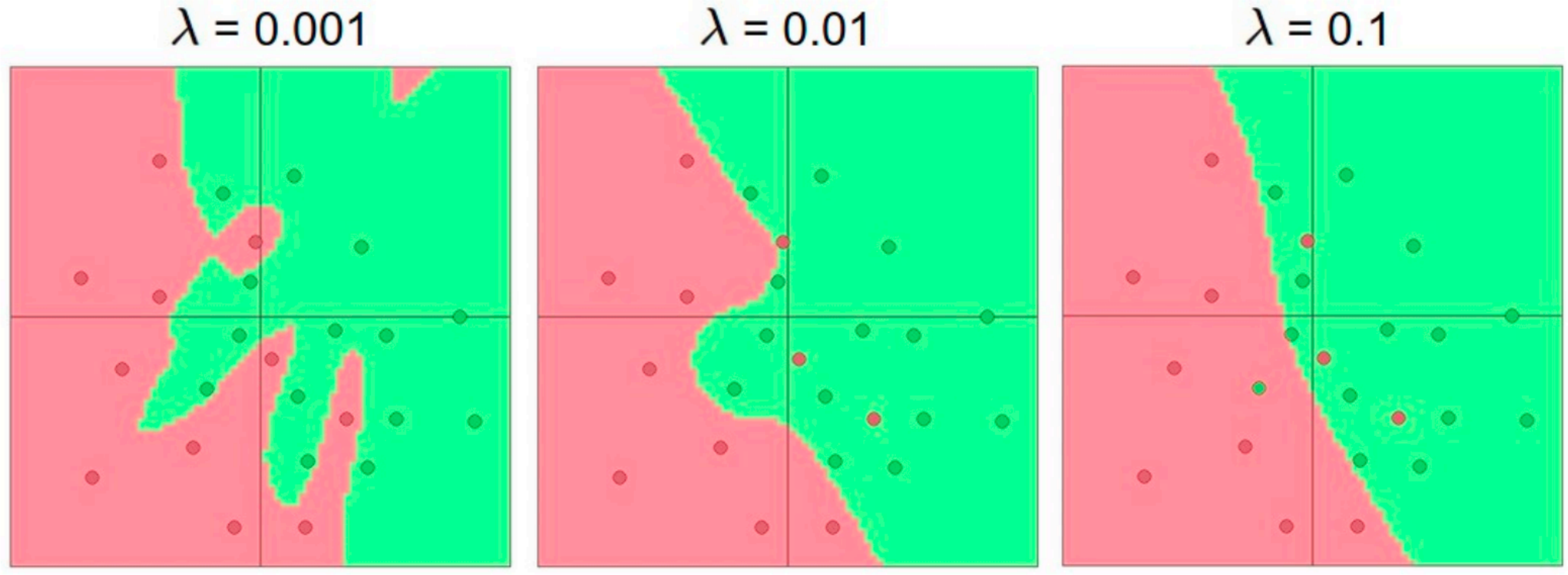
- **L1/L2 Regularization: It involves adding an extra element to the loss function, which punishes our model for being too complex or, in simple words, for using too high values in the weight matrix. (usually, lambda is 1e-4 or 1e-5)**



```
torch.optim.SGD(model4.parameters(), lr=0.8,weight_decay=1e-4)
```

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

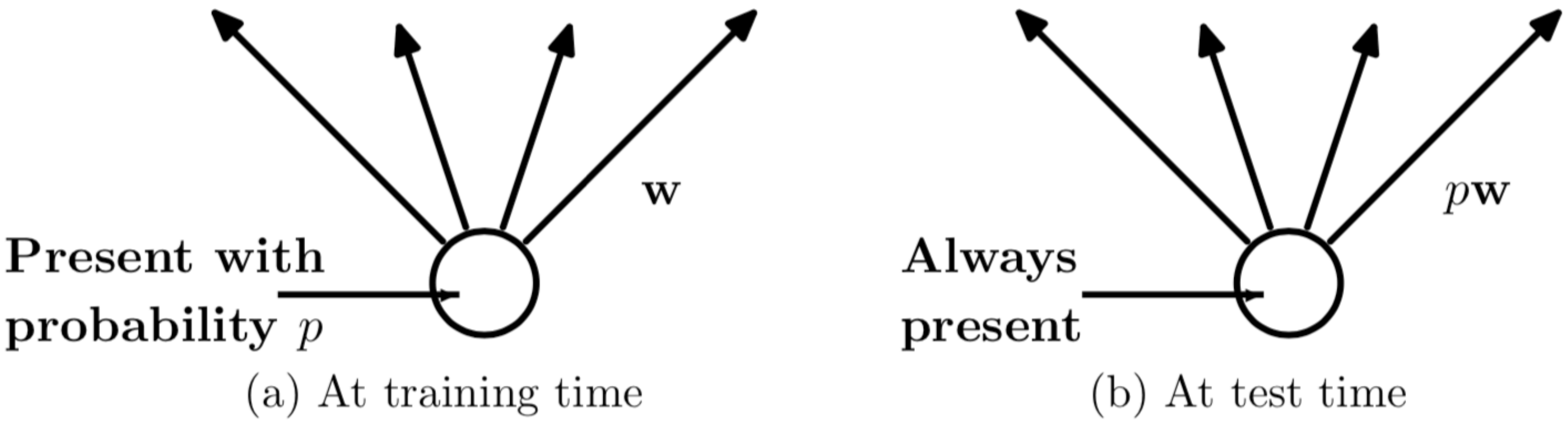


# Regularization in NN

- **Dropout: for each training batch, you turn off some neurons with a probability.**

- **Motivations:** With unlimited computation, the best way to "regularize" a fixed-sized model is to average the predictions of all possible settings of the parameters. Practically, it's computationally prohibitive. So dropout provides a method to use O(n) neural network to approximate O(2^n) different architectures with shared O(n^2) parameters.

- **Implementation:**
  - **Train Time:** Mask some neuron outputs as 0 with a probability.
  - **Test Time:** No parameters masked at test time but need to multiply with the dropout probability to approximate the expected output.
  - **Typical dropout rate:** [0.1, 0.5]. (Hyper-parameter)



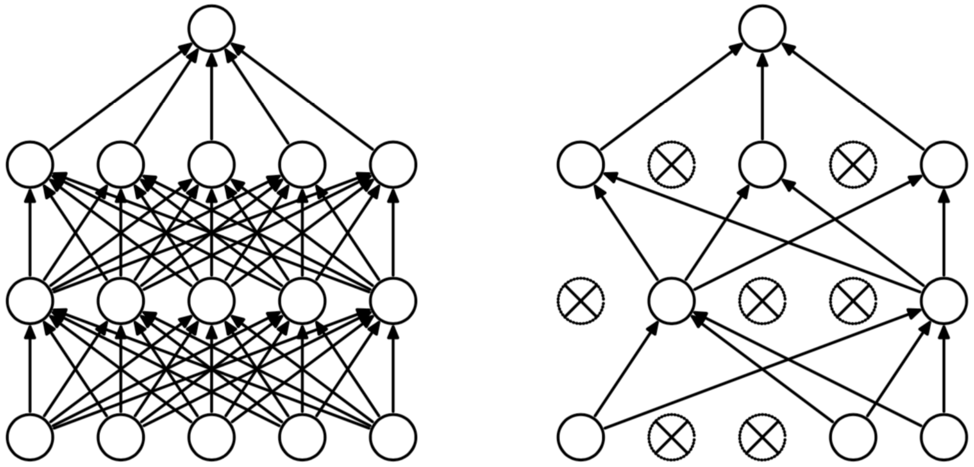
(a) At training time (b) At test time

**Paper: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (https://jmlr.org/papers/v15/srivastava14a.html)**

# Regularization in NN

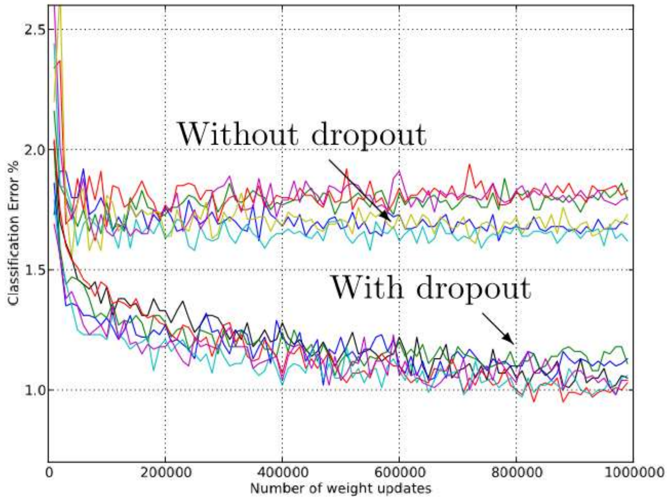
- **Dropout: for each training batch, you turn off some neurons with a probability.**

- **Results:**
  - **It addresses two problems:**
  - **1) overfitting:** disable some outputs so the later layers cannot overfit the data.
  - **2) generalization:** Model combination nearly always improves the performance of machine learning methods.



(a) Standard Neural Net

(b) After applying dropout.



**Paper: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (https://jmlr.org/papers/v15/srivastava14a.html)**

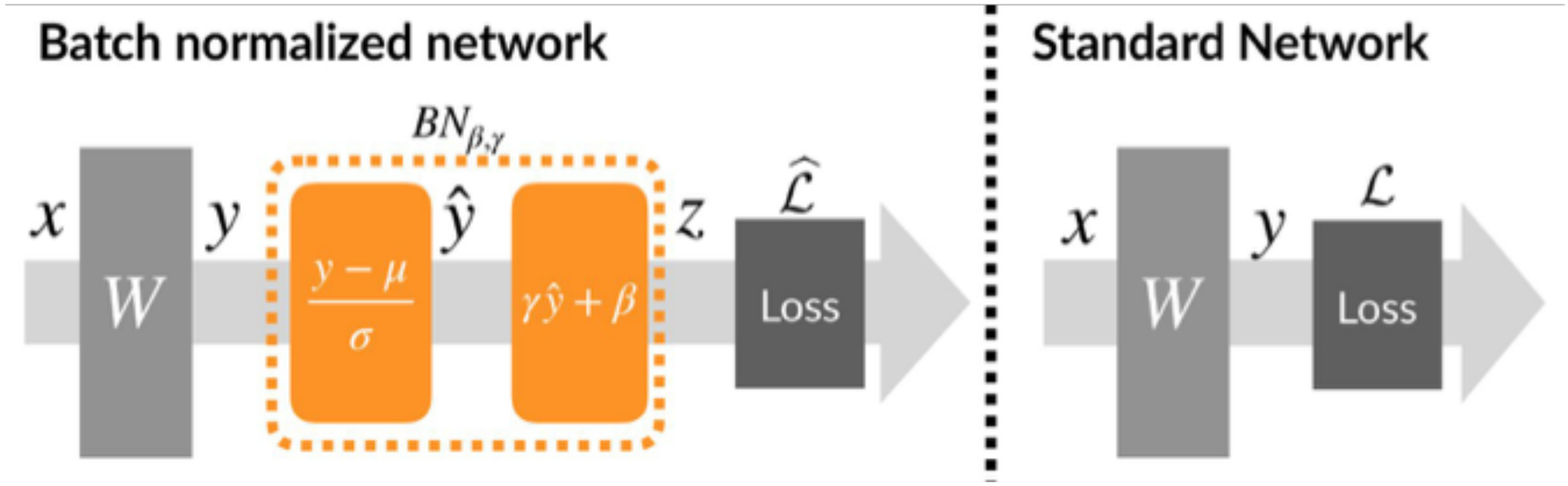
# Regularization in NN

- **Batch-Norm: wildly successful and simple technique for accelerating training and learning better neural network representations.**

- **Motivation:** The general motivation of BatchNorm is the non-stationarity of unit activity during training that requires downstream units to adapt to a non-stationary input distribution. This co-adaptation problem, which the paper authors refer to as internal covariate shift, significantly slows learning.

**Paper: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (https://arxiv.org/abs/1502.03167)**

# Regularization in NN

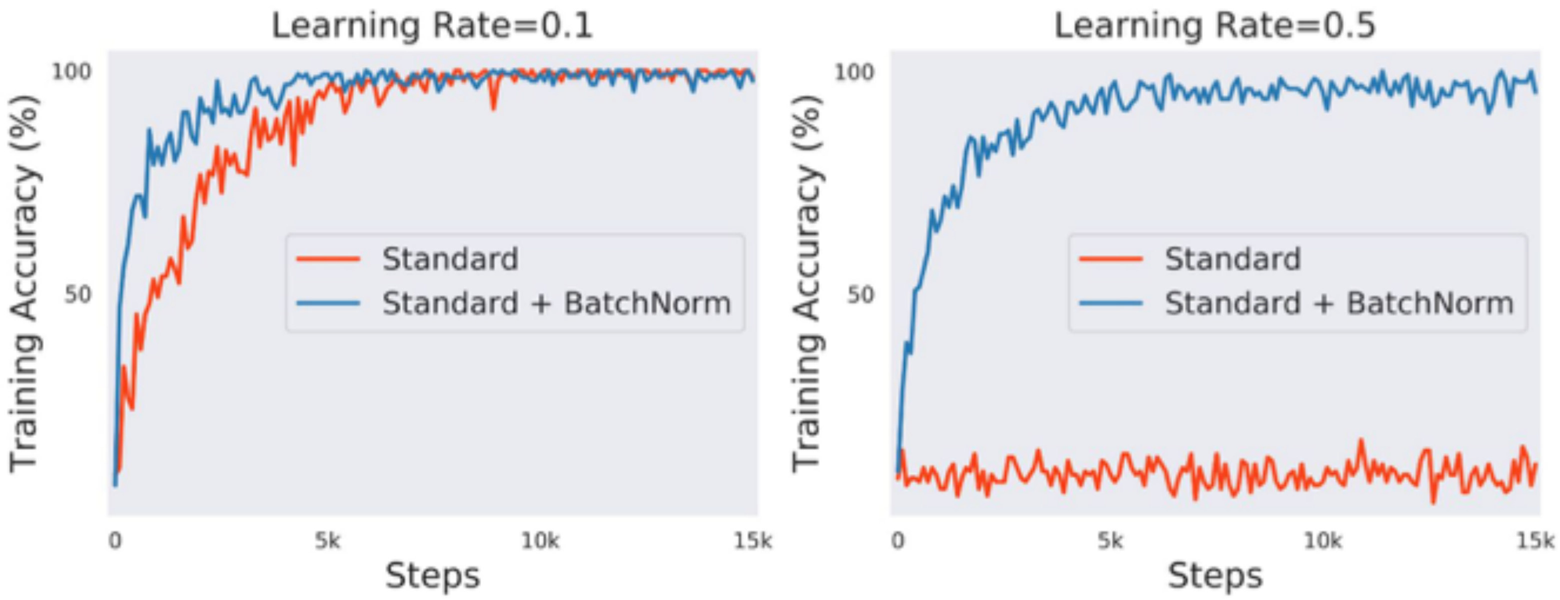


1.

$$u_{\mathcal{B}}^{(k)} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i^{(k)}$$

$$(\sigma_{\mathcal{B}}^2)^{(k)} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(\mathbf{x}_i^{(k)} - u_{\mathcal{B}}^{(k)}\right)^2$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$



2.

$$\mathbf{y}_i \leftarrow \gamma\hat{\mathbf{x}}_i + \beta$$

**Paper: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (https://arxiv.org/abs/1502.03167)**

# Regularization in NN

- **1) BN enables higher training rate:** Normally, large learning rates may increase the scale of layer parameters, which then amplify the gradient during back propagation and lead to the model explosion. However, with Batch Normalization, back propagation through a layer is unaffected by the scale of its parameters.

$$\mathrm{BN}(W\mathrm{u}) = \mathrm{BN}((aW)\mathrm{u}) \qquad \frac{\partial \mathrm{BN}((aW)\mathrm{u})}{\partial \mathrm{u}} = \frac{\partial \mathrm{BN}(W\mathrm{u})}{\partial \mathrm{u}}$$

- **2) Faster Convergence.**

- **3) BN regularizes the model:** a training example is seen in conjunction with other examples in the mini-batch, and the training network no longer producing deterministic values for a given training example. In our experiments, we found this effect to be advantageous to the generalization of the network.

**Paper: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (https://arxiv.org/abs/1502.03167)**

# Other Regularization in NN

- **Early Stop:** **literally, just stop the training when you see the validation score decreases. (overfit begins)**

- **Gradient Clipping:** **this is very important for RNN models.**
- **https://deepai.org/machine-learning-glossary-and-terms/gradient-clipping.**

- **Parameter Initializations:** **https://cs231n.github.io/neural-networks-2/#init**

# Common Pitfalls

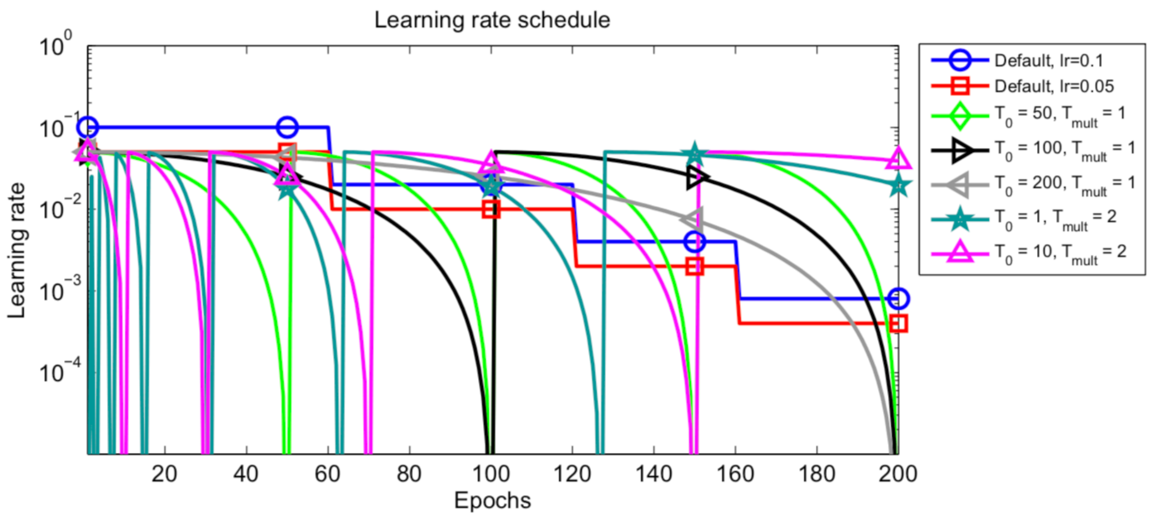
- **Zero initialization:** this would a very bad performance issue. -> each neuron would be identical.
- **Forget to shuffle the data set** -> Equally as bad as zero initialization because there is no randomness.
- **Choose of learning rate.**
  - Too Large: don't converge.
  - Too Small: don't get rid of the local optima.
- **Choose of batch size**
  - According to the property of SGD, smaller batch size leads to better convergence rate. But smaller batch size would deteriorate the performance of BN layer and running speed.
- Problem with momentum method. (prone to local minima)

# Common Pitfalls

- Order of BatchNorm and activation function. (Paper: https://arxiv.org/abs/1905.05928v1 )
  - Linear->BN->ReLU
  - Linear->ReLU->BN (reference paper)
- Tricky things come when using BatchNorm and Dropout together. (Paper: https://arxiv.org/abs/1801.05134)
  - Theoretically, they work against each other.
- Forget to normalize your input data. This would not be a problem if you are using BN.
- Put LR_scheduler in batch loop
- Forget optimizer.zero_grad()

# Tips for Improving the models

- Ensemble:
  - Different initializations, different architecture, different optimizers.
  - Different epochs.
- Dropout
  - What if we use dropout before the first linear layer?
- Deeper and wider
- Learning Rate Scheduler
  - Try cyclic learning rate scheduler. e.g. CosineAnnealingLR
  - Learning Rate warmup: solve the Radom initialization problem. Lead to stable learning process.


Learning rate schedule
Learning rate
$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
Epochs
Default, lr=0.1
Default, lr=0.05
$T_0 = 50, T_{mult} = 1$
$T_0 = 100, T_{mult} = 1$
$T_0 = 200, T_{mult} = 1$
$T_0 = 1, T_{mult} = 2$
$T_0 = 10, T_{mult} = 2$