11785 IDL Recitation 2
Network Optimization

Spring 2022

Rucha Khopkar, Urvil Kenia

Objective

Tricks for optimizing deep learning models which should be helpful for homework part 2’s, project, etc.

Topics

- Data Manipulation
- Model Tuning
- Ensembles

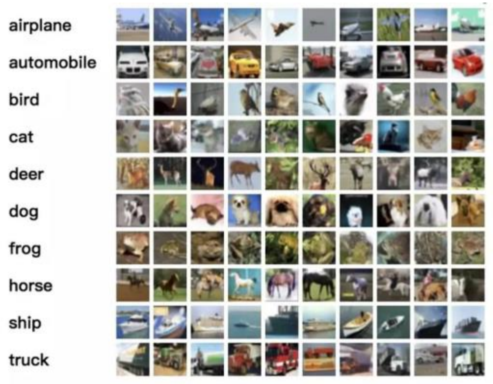
Data Manipulation

- Necessity of data
- Data Augmentation
- Data Normalization

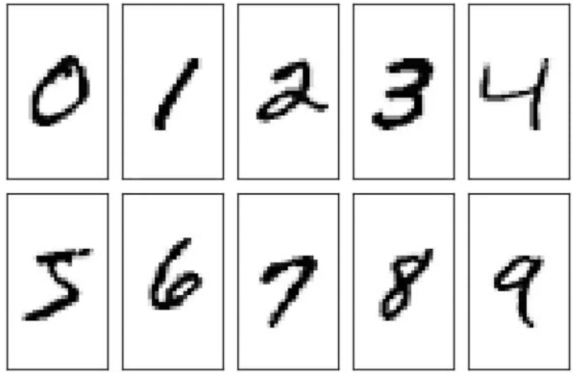
Necessity of ‘quality’ data

It’s obvious how important data is for deep learning. But we need clean large datasets for obtaining high accuracies especially for tasks such as machine translation, sentiment analysis, etc.

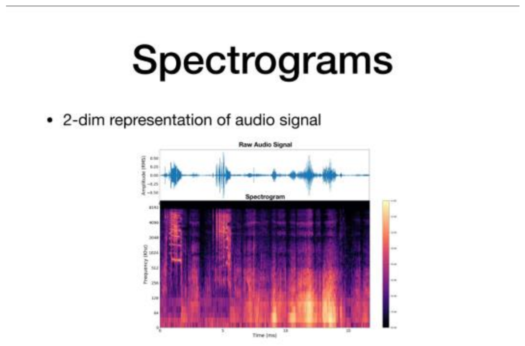
The amount of data required also depends on the dimensionality of the data. The higher the dimensionality, the higher is the amount of data needed. Data collected can also often be noisy, unannotated, or downright unusable. There are various pre-processing techniques you can use to obtain desired datasets.



CIFAR-10



MNIST

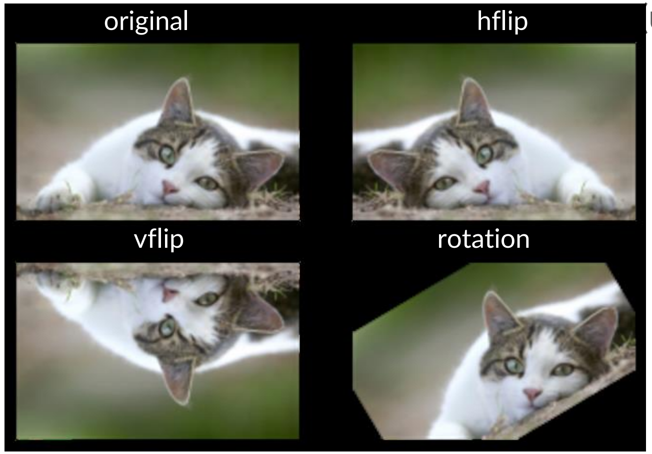


Spectrograms

- 2-dim representation of audio signal

Data Augmentation

- Sometimes, it is not possible to obtain the amount of data required. In such cases, we can use data augmentation techniques to generate more data and it also makes the model more generalizable.
- You can use image augmentation techniques such as flipping, rotating, etc. and also frequency
(useful for all HWs).



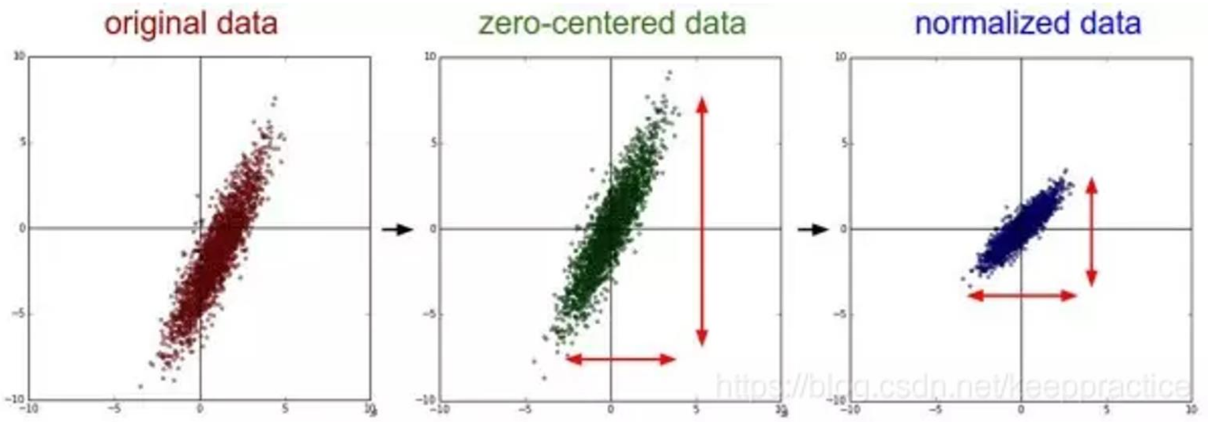
Ref: https://pytorch.org/vision/stable/transforms.html
https://pytorch.org/audio/master/tutorials/audio_feature_augmentation_tutorial.html

Data Normalization

Datasets obtained may have different scales for different features. In such cases, we scale the data which leads to faster convergence. It is also helpful when data is collected under varying conditions such as lighting in an image, gain in speech data, etc.

Types of normalization techniques:

- Z-score normalization (standardization)
- Min-max scaling
- Standard scaling

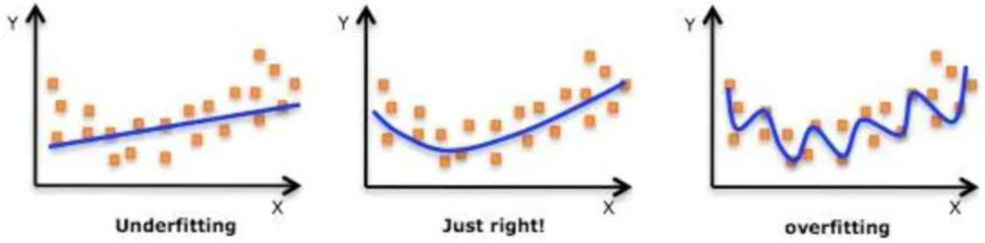


Model Tuning

- Choosing a Model
- Learning Rate
- Weight Initialization
- Optimizer

Choosing a Model

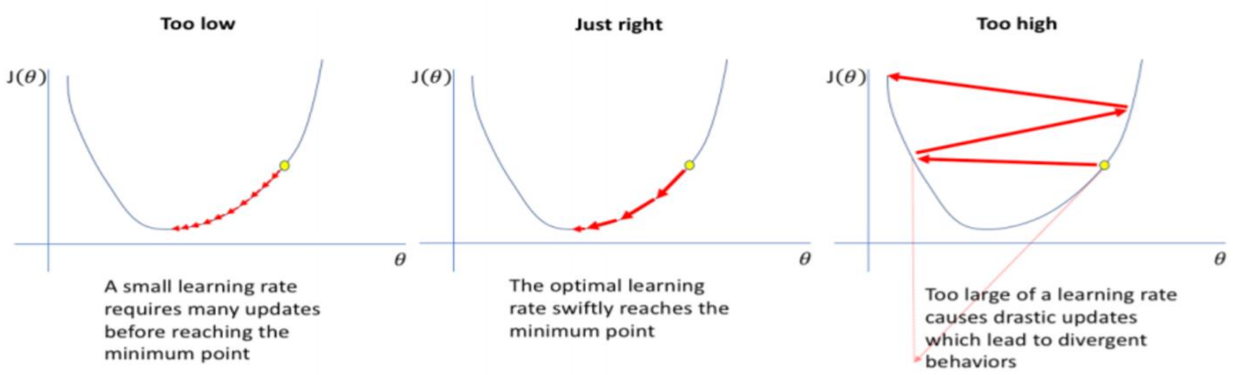
- There are tons of models available for most deep learning tasks. HW2P2 alone can be done using any one of the 15+ models such as Resnet, Alexnet, Mobilenet, etc.
- The best way to choose models is to read papers, DL blogs, Piazza (only for this course) and understand the intricacies and relative performance for your task.
- Deeper and wider models tend to generally perform better (more parameters, duh!) but they can cause overfitting. Most of the remaining slides are dedicated to solving the overfitting problem.



If I’ve learned anything from my studies of Deep Learning, it’s that one can solve most problems by just adding more layers

Learning Rate

- LR is one of the most important hyperparameters while training models. It dictates the weight updates and hence affects performance the most.
- You can experiment with different learning rates (also depends on the optimizer you select) as well as use a LR scheduler.
- The LR scheduler effectively changes the LR across epochs based on a function or a fixed schedule. Some students also change the LR manually (but that is not easy).


Too low
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
Just right
J(θ)
θ
The optimal learning rate swiftly reaches the minimum point
Too high
J(θ)
θ
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Ref for LR schedulers: https://pytorch.org/docs/stable/optim.html

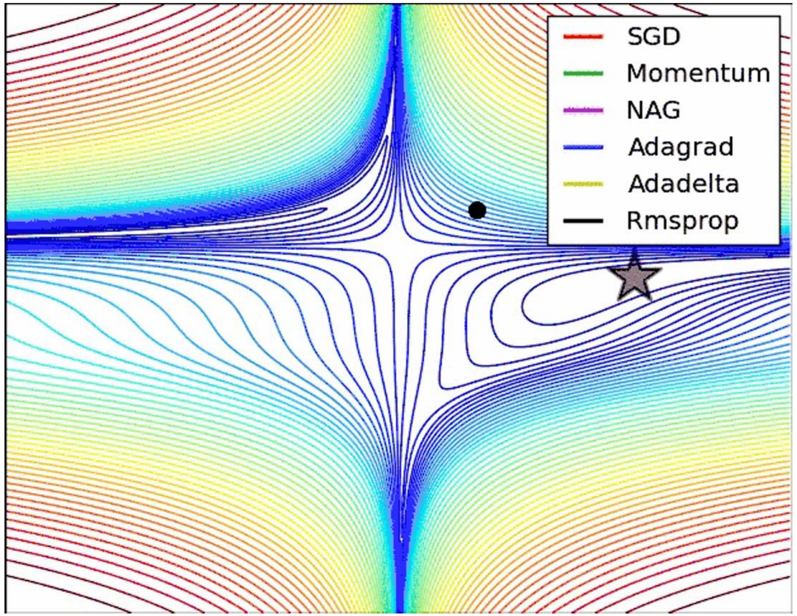
Weight Initialization

- Another important trick to improve your performance is using weight initialization for your hidden layers. This results in faster convergence as well as finding better minima.
- There are a number of initializations you can try, ex. Xavier, Kaiming, Uniform, Gaussian, etc. Depending on the type of activation function (ReLU, Sigmoid, etc.) and hidden layer (linear, CNN, RNN, etc.) the initialization strategy would vary.
- Initializations also sometimes help resolve the issue of vanishing or exploding gradients.
- Note: PyTorch uses some initialization techniques for all of its layers, read them before applying your own initialization strategies.

Ref: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
https://arxiv.org/pdf/1502.01852.pdf
https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899

Optimizer

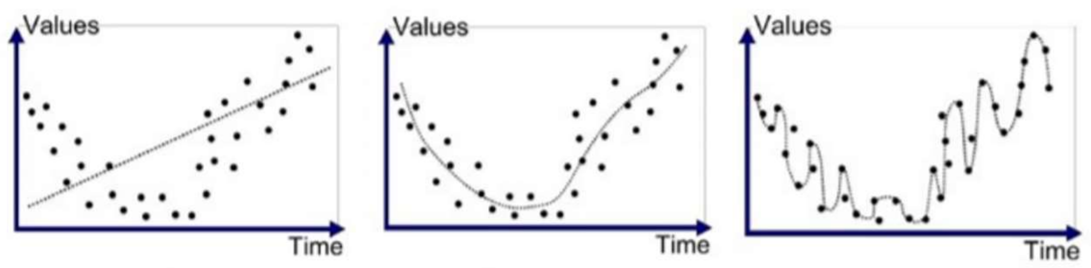
- Stochastic Gradient Descent (SGD) is one of the first and most heavily used optimizers. But now, there are many other optimizers such as Nesterov Accelerated Gradient (NAG), Adam, AdamW, Rmsprop, etc.
- Generally, Adam converges faster in most cases whereas SGD might converge slower but can find a better minima.
- The parameters for SGD need to be finely tuned to achieve higher performance.
- There is a suggested approach of initially using Adam and then switching to SGD: https://arxiv.org/abs/1712.07628.



Models Tuning

4. Regularization

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

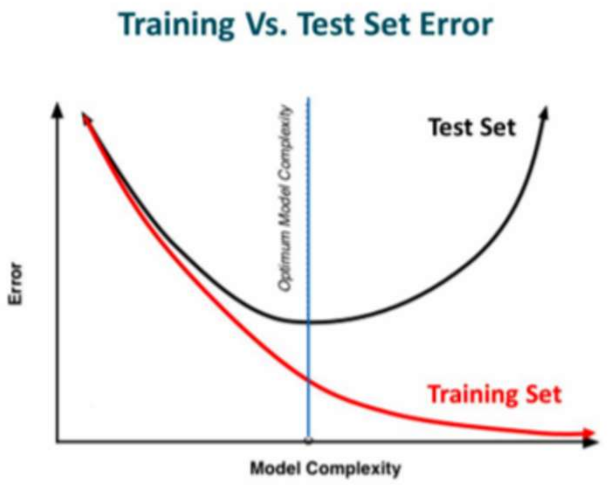


Underfitted | Good Fit/Robust | Overfitted

$y = a + w_0 x$

$y = a + w_0 x + w_1 x^2 + w_2 x^3$

$y = a + w_0 x + w_1 x^2 + w_2 x^3 + w_3 x^4 + \cdots + w_{10} x^{11}$



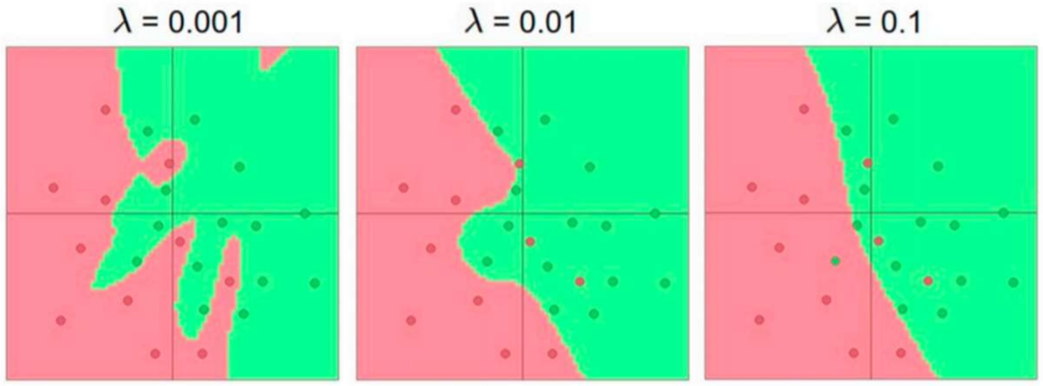
Models Tuning

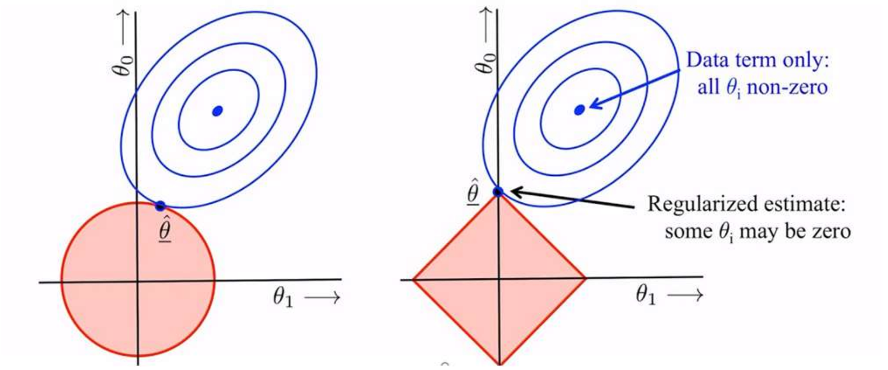
- **L1/L2 Regularization**

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)

L1 Norm $Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$

L2 Norm $Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$





$\theta_0 \longrightarrow$
$\hat{\theta}$
$\theta_1 \longrightarrow$
Data term only: all θ_i non-zero
Regularized estimate: some θ_i may be zero

Models Tuning

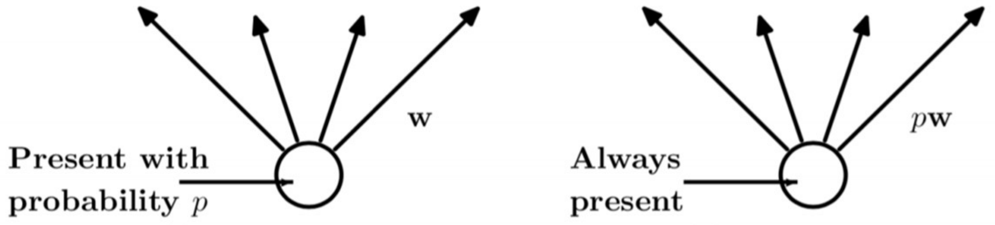
- **Dropout**

For each training batch, you turn off some neurons with a probability.

Motivation: With unlimited computation, the best way to "regularize" a fixed-sized model 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 with a probability
- **Test Time:** No parameters masked at test time but need to multiply with the dropout probability to approximate the expected output
- **Hyper-parameter:** dropout rate, usually from 0.1 to 0.5



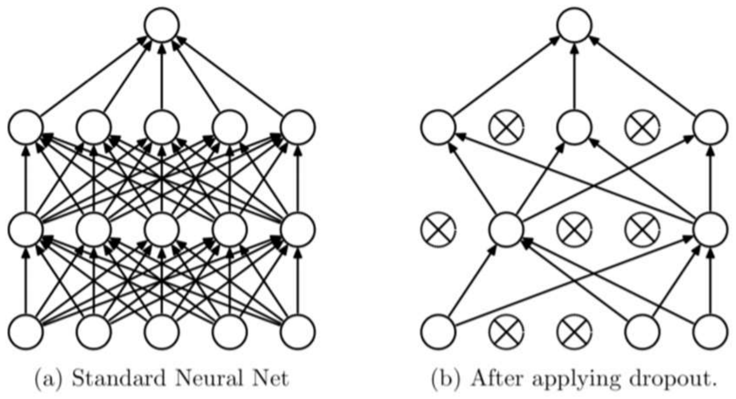
(a) At training time (b) At test time

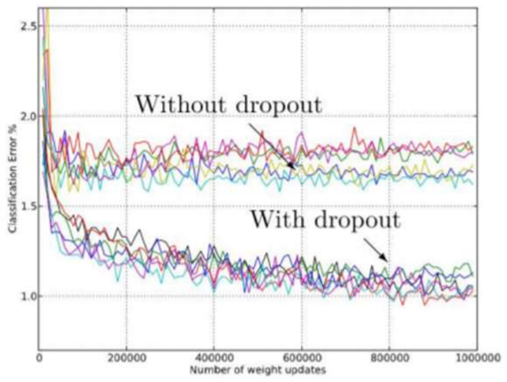
Models Tuning

- **Dropout**

Two problems resolved:

- **Overfitting:** disable some outputs so that the layers cannot overfit the data. Or we can see dropout is like generating many models, have different "overfitting" effect but some of them are opposite. We average them and can prevent overfitting in general
- **Generalization:** let model abandon some specific features with probability and decrease co-adaptation between the neurons





Models Tuning

- **Batch-Norm**

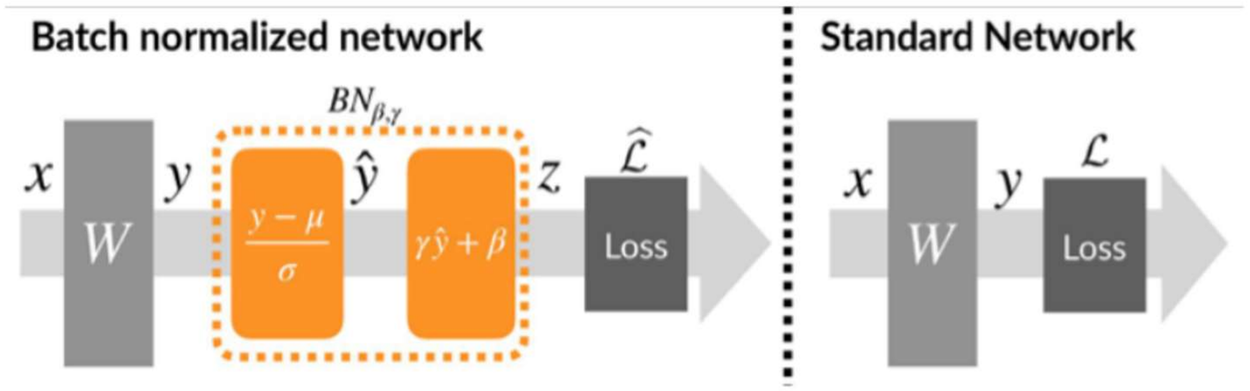
Wildly successful and simple technique for accelerating training and learning better neural network representations

Motivation: The general motivation of BatchNorm is the non-stationary 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 ICS (internal covariate shift), significantly slows learning.

ICS: In neural networks, the output of the first layer feeds into the second layer, the output of the second layer feeds into the third, and so on. When the parameters of a layer change, so does the distribution of inputs to subsequent layers.

These shifts in input distributions can be problematic for neural networks, especially deep neural networks that could have a large number of layers.

Models Tuning

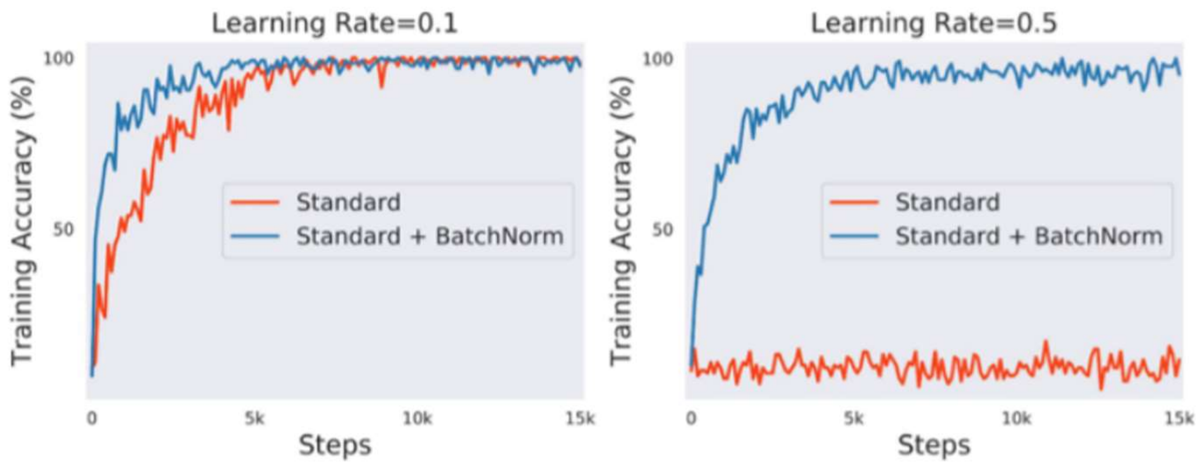


1.

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

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

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



2.

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

Models Tuning

- **Batch-Norm**

Benefits:

a) **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, BP through a layer is unaffected by the scale of its parameters

$$BN(Wu) = BN\big((aW)u\big) \quad \frac{\partial BN((aW)u)}{\partial u} = \frac{\partial BN(Wu)}{\partial u}$$

a) **Faster Convergence.**

b) **BN regularizes the models:** 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.

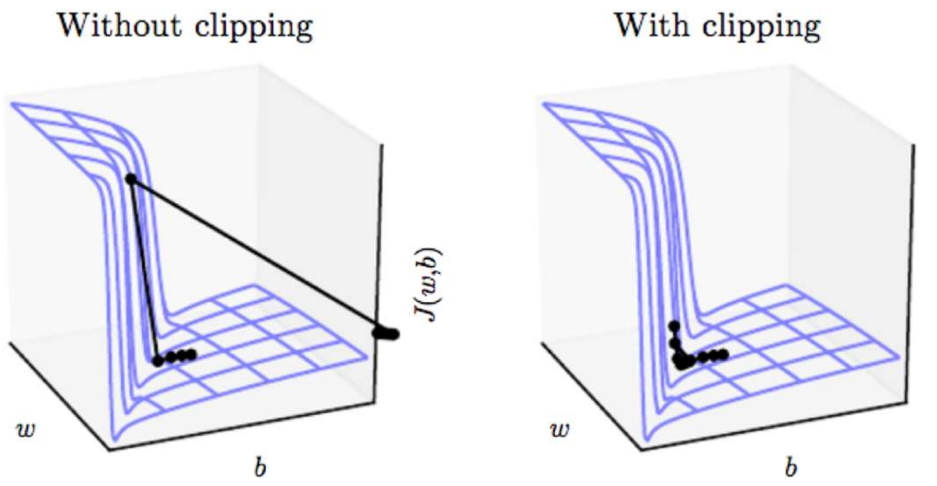
Models Tuning

- **Early Stop**

Literally, just stop the training when you see the validation score decreases, where overfitting begins

- **Gradient Clipping**

Once the gradient is over the threshold, clip and keep them to the threshold value. It is important for RNN.



Efficient Training

- **Mixed Precision Training (Pytorch > 1.6.0)**
  - combine FP32 and FP16 during training while achieving same accuracy as FP32 training
- **Why?**
  - faster training (2-3x)
  - less memory usage
  - larger batch size, larger model, larger input
- **How? …and loss of information?**
  - FP32 master copy of weights

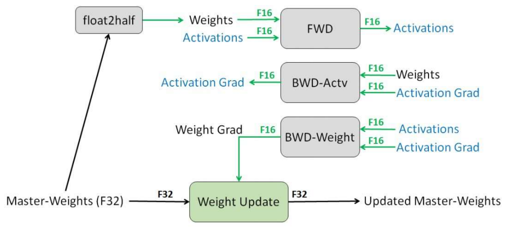
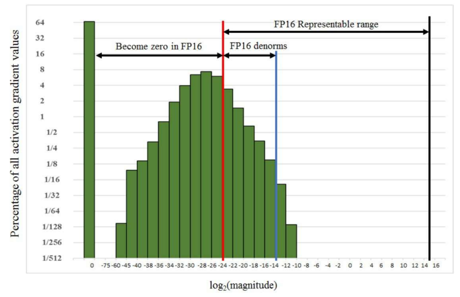
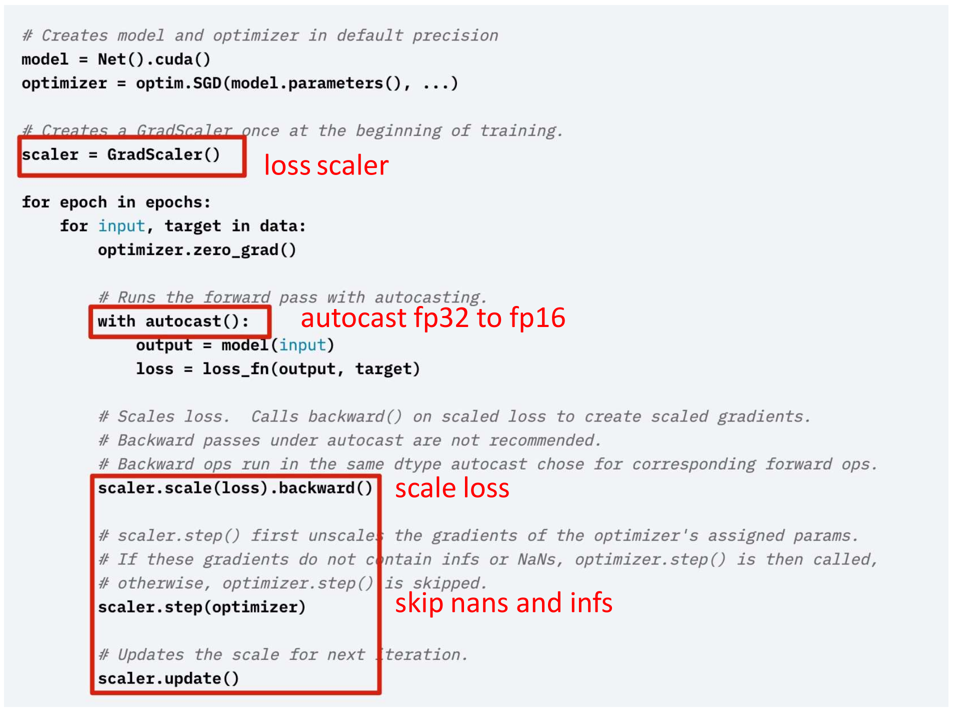


Figure 1: Mixed precision training iteration for a layer.

  - loss scaling



- Speak in Pytorch….



```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

- Ref
  - [1] Mixed Precision Paper
  - [2] Pytorch Mixed Precision Tutorial

Efficient Training

- **Want to train the model even faster**? with more than 1 gpu :)

- **DataParallel**

```
model = nn.DataParallel(model)
```

  - single-process multi-thread parallelism
  - split batch data on each card
  - replicate forward pass on each card
  - but…GPU memory is imbalanced across cards

- **DistributedDataParallel**
  - multi-process parallelism
  - broadcast happens at DDP construction rather than each forward

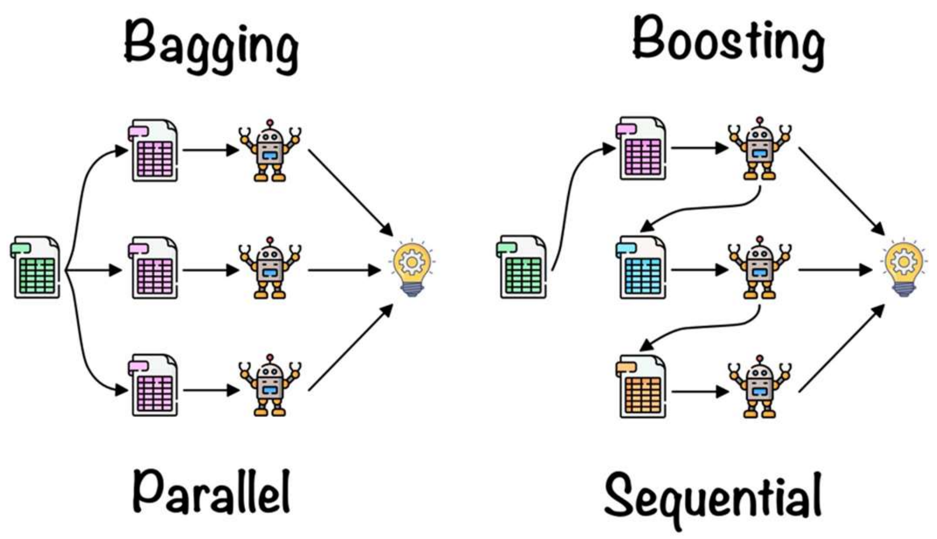
```
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    # initialize the process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()
```

```
def demo_basic(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
    setup(rank, world_size)

    # create model and move it to GPU with id rank
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

- Ref
  - [1] Pytorch Parallel Tutorial

Ensemble

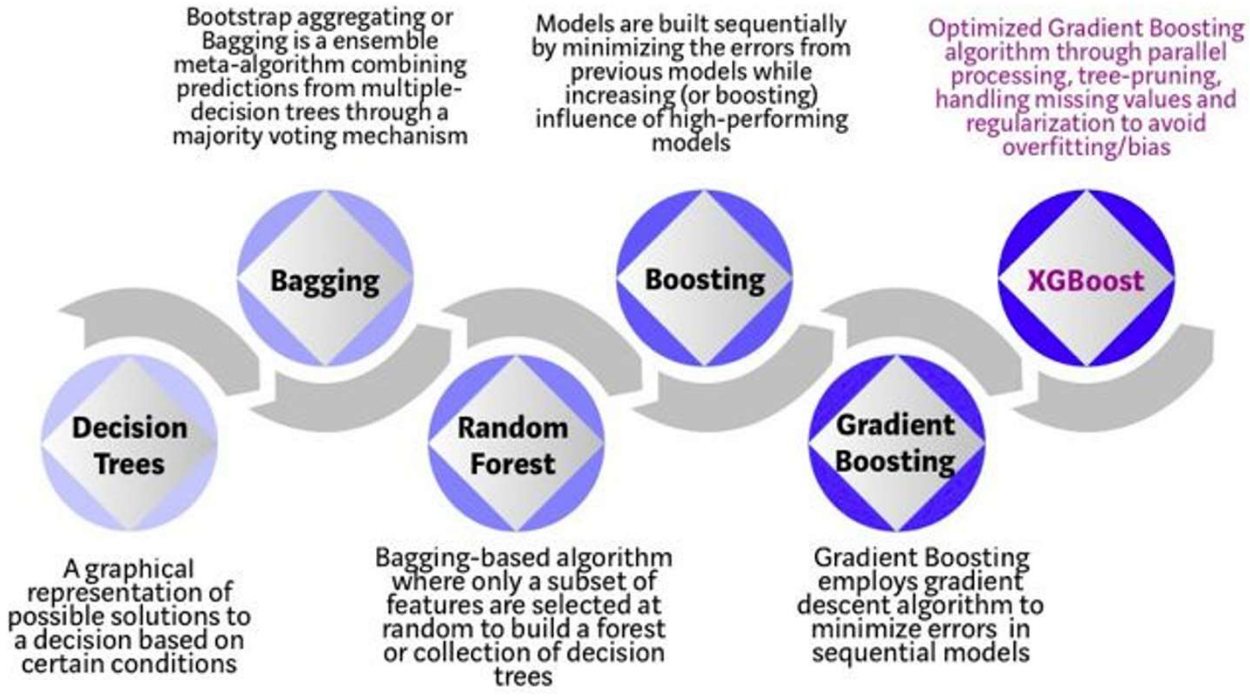
Ensemble learning is **a machine learning paradigm where multiple learners are trained to solve the same problem**. In contrast to ordinary machine learning approaches which try to learn one hypothesis from training data, ensemble methods try to construct a set of hypotheses and combine them to use.



Bagging: considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process. **Decrease variance.**

Boosting: considers homogeneous weak learners, learns them sequentially in a very adaptative way (a base model depends on the previous ones) and combines them following a deterministic strategy. **Decrease bias.**

Ensembles

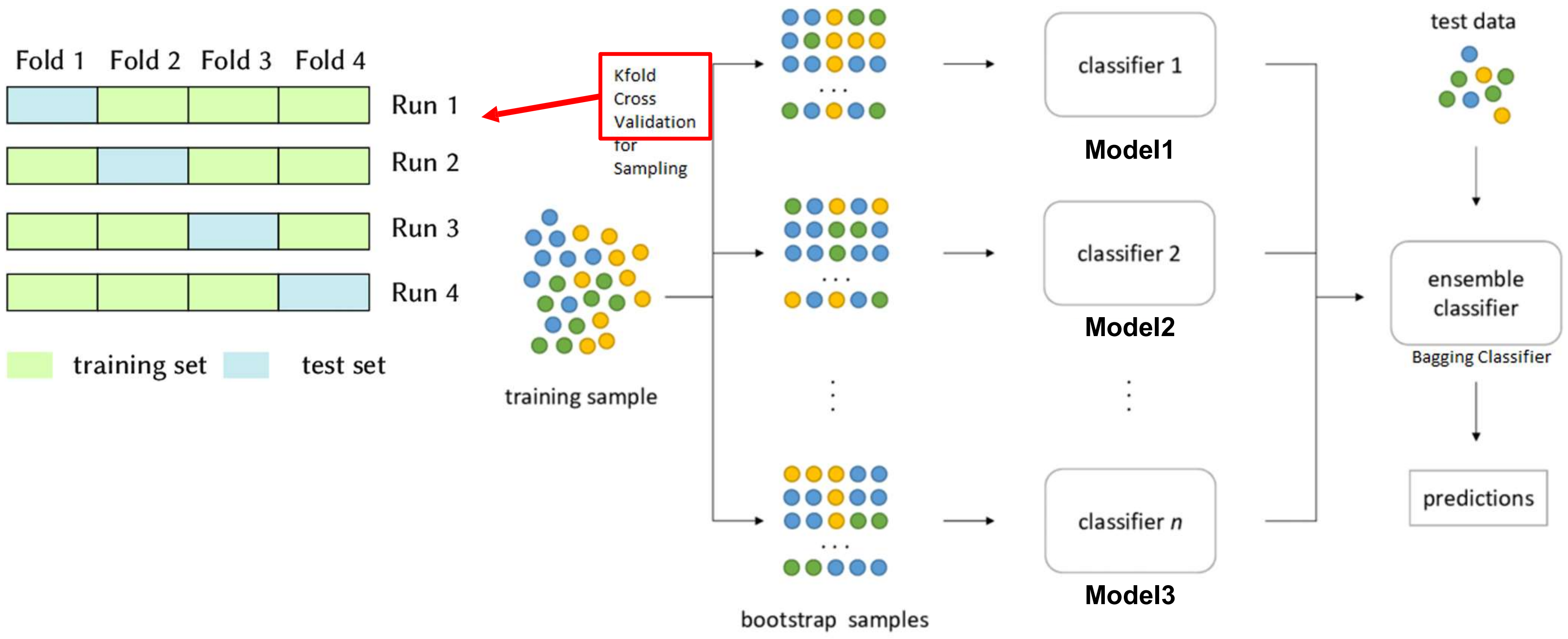

Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism
Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models
Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias
Bagging
Boosting
XGBoost
Decision Trees
Random Forest
Gradient Boosting
A graphical representation of possible solutions to a decision based on certain conditions
Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees
Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

Develop of ensembles

Refer to paper. Author

Ensembles

- **Bagging: Voting-based algorithm. Train different models in isolation and use average voting or weighted voting method to get result.**

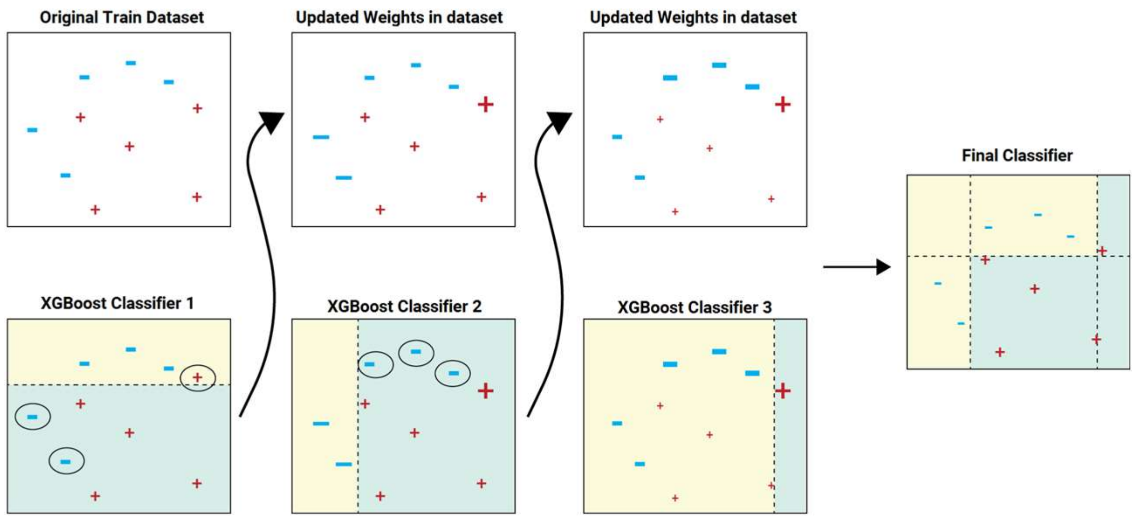


Bagging Classifier Process Flow

Ensembles

- **XGBoost: One of most powerful weapons in Kaggle. Many deep learner use it reached top in many competitions**

  XGBoost as a boosting method is a category of ensemble methods. Rather than training all of the models in isolation of one another, boosting trains models in succession, with **each new model being trained to correct the errors made by the previous ones.** Models are added sequentially until no further improvements can be made, that's why it is called additive model.



Refer to paper. Author Tianqi Chen joined CMU in 2020!

Ensembles

- **XGBoost: One of most powerful weapons in Kaggle. Many deep learner use it reached top in many competitions**

Pseudocode:

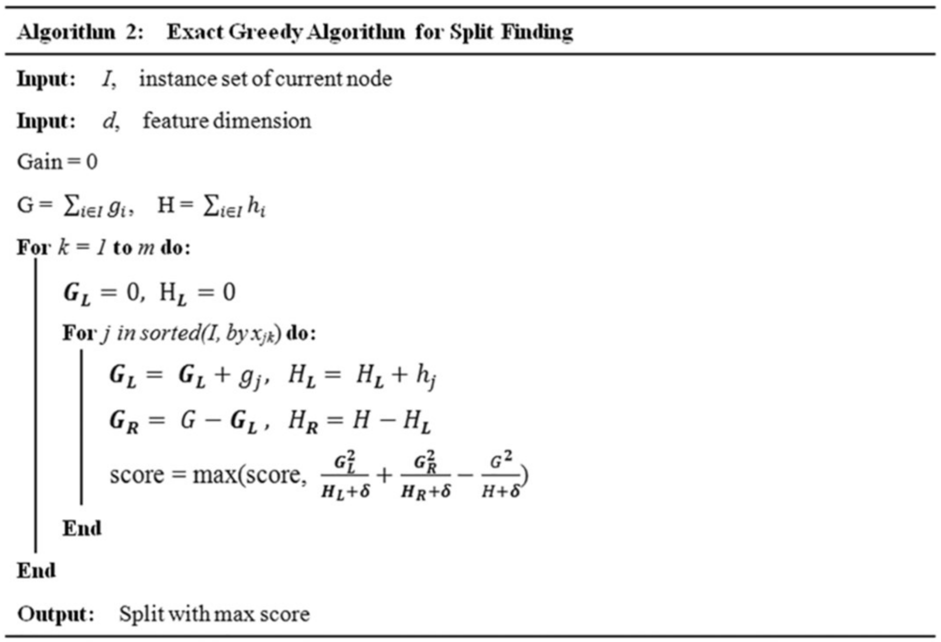
① Initialize $f_0(x)$;

② For m = 1 to M:

a. Compute the 1st derivative and 2nd derivative of loss function over each sample;

b. Recursively use "Greedy Algorithm for Split Finding" to generate the base learner;

c. Add the base learner to models.



```
Algorithm 2:  Exact Greedy Algorithm for Split Finding
Input:  $I$,  instance set of current node
Input:  $d$,  feature dimension
Gain = 0
$G = \sum_{i\in I} g_i$,  $H = \sum_{i\in I} h_i$
For $k = 1$ to $m$ do:
    $G_L = 0$,  $H_L = 0$
    For $j$ in sorted($I$, by $x_{jk}$) do:
        $G_L = G_L + g_j$,  $H_L = H_L + h_j$
        $G_R = G - G_L$,  $H_R = H - H_L$
        score = max(score, $\frac{G_L^2}{H_L+\delta} + \frac{G_R^2}{H_R+\delta} - \frac{G^2}{H+\delta}$)
    End
End
Output:  Split with max score
```

Refer to paper. Author Tianqi Chen joined CMU in 2020!

More Hints for your Homework

- **Always keep this in mind: "Practice make perfect!"** Besides the theory behind different algorithms and different tricks, Deep Learning is kind like an experimental topic. If you wonder what tricks can achieve best results or which parameters suit the model well, just give it try!
- **Remember to shuffle the data set:** if not, your performance will be pretty bad.
- **Note: Do not shuffle the test set.**
- **Choose learning rate wisely:** too large LR will not converge while too small can hardly get rid of local optima
- **Choose 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. In general, different batch size wouldn't cause too much difference. Larger batch size tends to have better performance but will occupy more memory in GPU. (if you have **cuda out of memory** error, try smaller bs)
- **Try self-ensemble:** average the parameters of your model at different training epochs.
- Tricky things come when using BatchNorm and Dropout together. (Paper)
- Don't forget **optimize.zero_grad()**
- Use LR_scheduler in the iterations loop.
- Try to use **torch.cuda.empty_cache()** and **del** to release all unoccupied cached memory

More and More Hints for your Homework

- DataLoader has bad default settings, so remember to tune **num_workers > 0** and default to **pin_memory = True** (colab will gradually restrict the num_workers from 8 to 4 to 2 and then only 1) :-(
- Try to use **torch.backends.cudnn.benchmark = True** to autotune cudnn kernel choice (this code can be put at the beginning)
- Max out the batch size for each GPU to amortize compute.
- Do not forget **bias = False** in weight layers before BatchNorms, it is a noop that bloats model.
- Try to use **for p in model.parameters(): p.grad = None** instead of model.zero_grad()
- Try every tricks or models that you can find in the papers or post!
- **Always Remember to save your model state after each iterations!!!!!!!!!!!!!!!!**

  (Use model.state_dict())

Before DDL, get an extremely good result
Forget to save model and Colab collapses
I will reach the top in leaderboard! I am unbeatable!!!!
Let me die......

Thanks to Jacob Li (TA in 20Fall) for some hints!

Other course students: Wow, you are taking 11785! You must be good at Deep Learning!
Me

Good Luck!

There are so many tricks or architectures waiting for you to explore and use them in your Kaggle Competitions. Just try your best and reach to the top!