

11-785/685/485

Fall 2024

Lab 2

# Network Optimization

Ablations, Hyperparameter Tuning, Normalizations

Romerik Lokossou | Alex Gichamba | Purusottam Samal

# Data Manipulation

- Data Augmentation
  - Data Normalization
-

# 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, scaling, changing perspective etc. and also Time masking and frequency masking for speech data used in all the



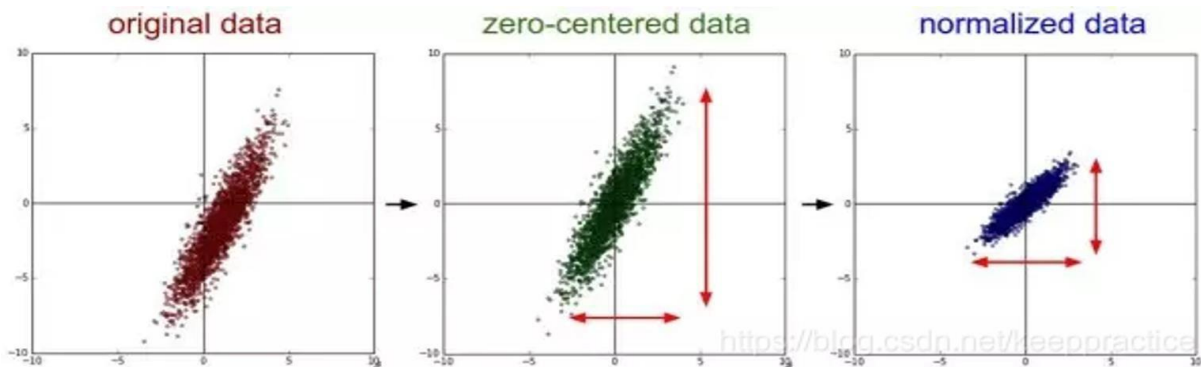
Ref: <https://pytorch.org/vision/stable/transforms.html>  
[https://pytorch.org/audio/master/tutorials/audio\\_feature\\_augmentation\\_tutorial.html](https://pytorch.org/audio/master/tutorials/audio_feature_augmentation_tutorial.html)

# 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
- Cepstral Mean Normalization



# Cepstral Mean Normalization

- Cepstral mean normalization (CMN) is a computationally efficient normalization technique for robust speech recognition.
- CMN minimizes distortion by noise contamination for robust feature extraction.
- CMN has been used in different applications as this technique has proven to provide better speech recognitions results in different environments.
- CMN has the capabilities to reduce differences between test and training data produced by channel distortions.
- CMN has also been found to be able to reduce differences in feature representation between speakers can also partly reduce the influence of background noise.

# Ablations

- How do I think about Model Architecture for HW1P2?

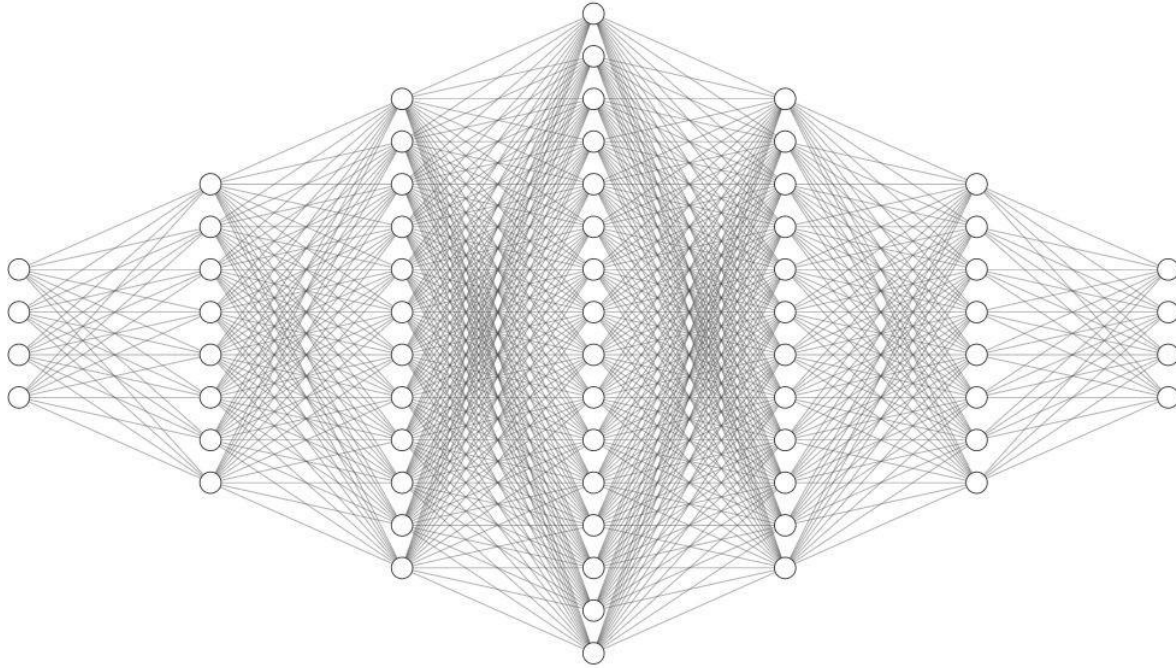


---

# Model Architecture

- As you might know already, one of your two biggest tasks in HW1P2 is to find the right model architecture for your MLP to be able to predict the phoneme of a given frame with context.
- Experimenting with finding the right model architecture is not easy - and takes a lot of time in a deep learning task.
- One of the ways to find useful architectures for your tasks is to read papers, blogs, Piazza (for this course) and understand the intricacies and measure relative performance for your task.
- Typically, Deeper and wider models tend to generally perform better (Why?)

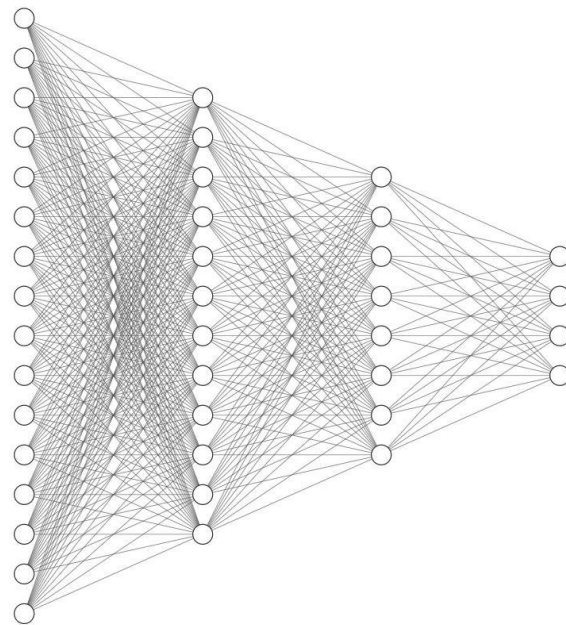
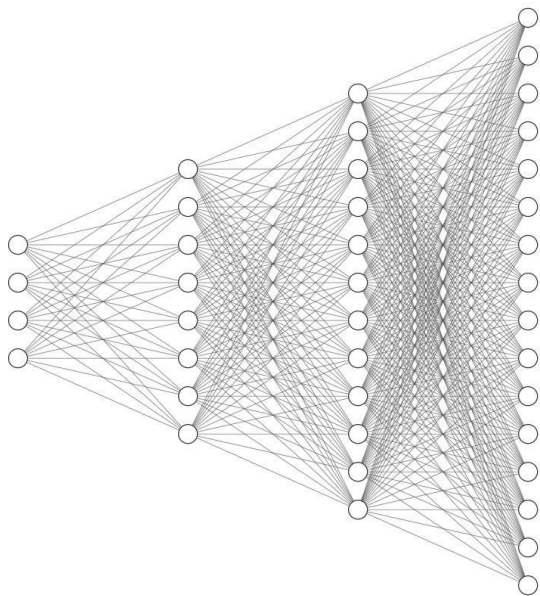
# Model Architecture - Diamond



Diamond Architecture

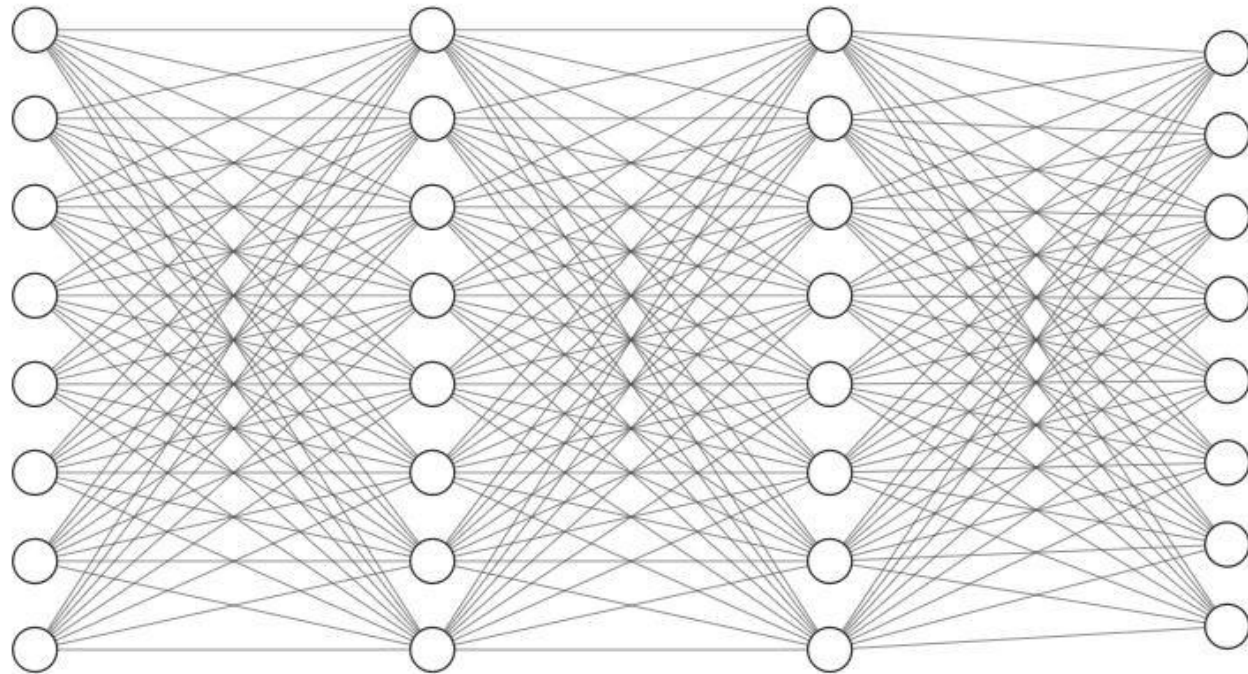


# Model Architecture - Pyramids



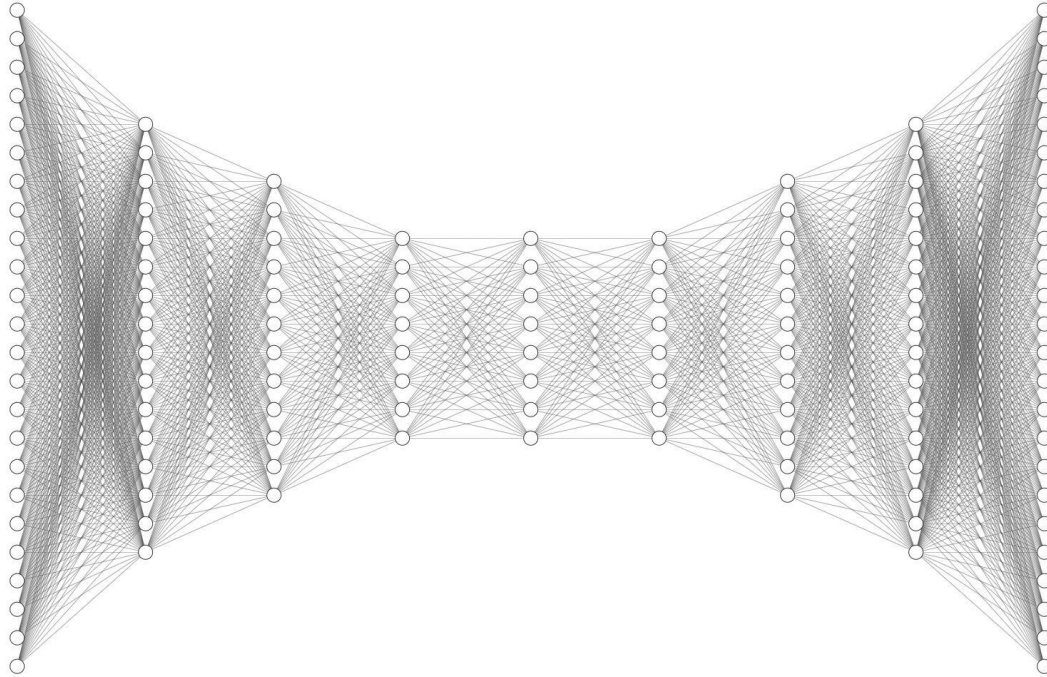
Pyramid/ Inverse-Pyramid

# Model Architecture - Cylinder



**Cylinder**

# Model Architecture - Hourglass



Or Any weird Architecture 🤪

# Model Architecture - Summary

While working with MLPs in HW1P2, you will have to experiment with the following possibilities in your architecture

- Network shape (as discussed in previous slides)
- Deeper Layers
- Wider Layers
- Activation Functions (ReLU, Sigmoid, GELU, Softplus, Tanh)
- Batchnorm (more on this in an upcoming slide)
- Dropout (more on this in an upcoming slide)

Remember, you have a parameter limit so you have to construct your architecture around a limited budget of parameters. This will enable you to actively experiment with hyperparameters and model structure than to just stick to models that are 10+ layers deep and over 8000 neurons wide at each layer and let it do its job.

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



# Ablations

- Schedulers

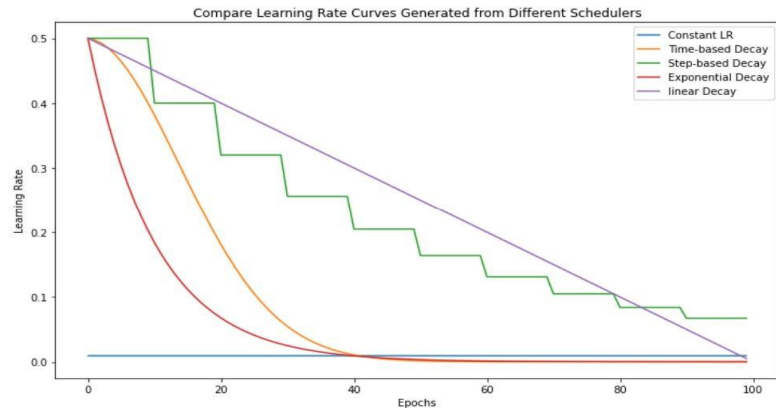
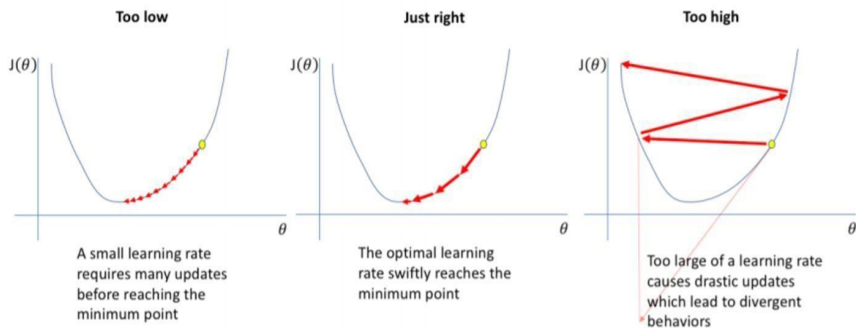
---

# What are Schedulers used for?

- The learning rate controls how big of a step for an optimizer to reach the minima of the loss function.
- A learning rate scheduler adjusts the learning rate according to a pre-defined schedule during the training process.
- PyTorch supports:
  - StepLR
  - MultiStepLR
  - ConstantLR
  - LinearLR
  - ExponentialLR
  - .....

# Learning Rate

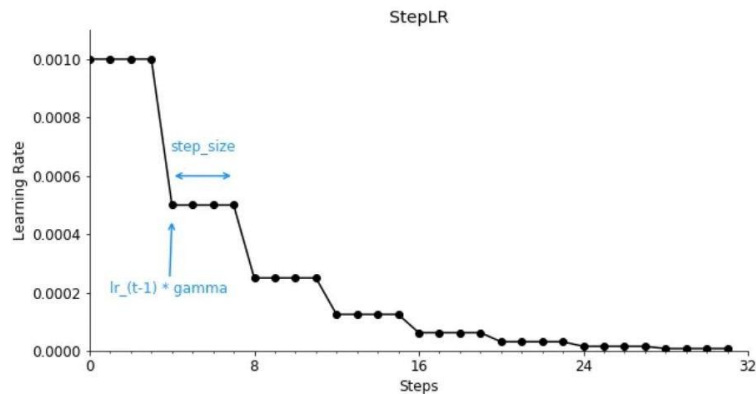
- LR is one of the most important hyperparameters while training models. It controls the weight update step taken by your model during gradient descent, which directly influences convergence of your model for a deep learning task.
- You can experiment with different learning rates (also depends on the optimizer you select) as well as use a LR scheduler.
- The LR scheduler effectively changes the LR across epochs based on a function or fixed schedule. You can also change the LR manually at certain epochs (but that may not be efficient in terms of resources spent monitoring training).



# Step LR

- The [StepLR](#) reduces the learning rate by a multiplicative factor after every predefined number of training steps.

```
scheduler = StepLR(optimizer, step_size = 4, gamma = 0.5)
```



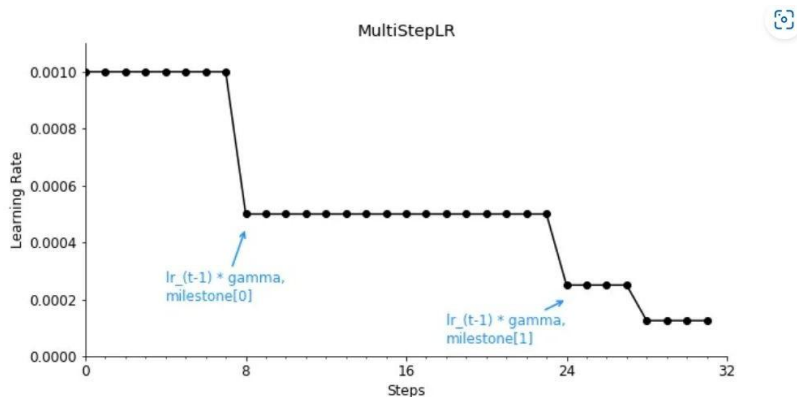
PyTorch Learning Rate Scheduler `StepLR` (Image by the author)



# Multistep LR

- The [MultiStepLR](#) — similarly to the [StepLR](#) — also reduces the learning rate by a multiplicative factor but after each pre-defined milestone.

```
scheduler = MultiStepLR(optimizer, milestones=[8, 24, 28], gamma = 0.5)
```

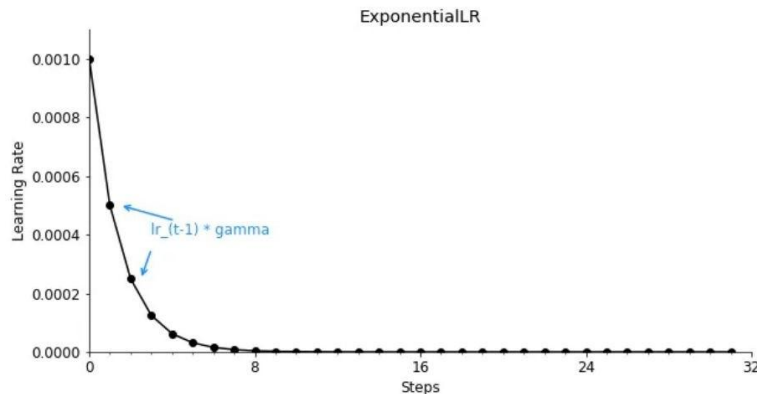


PyTorch Learning Rate Scheduler MultiStepLR (Image by the author)

# Exponential LR

- The [ExponentialLR](#) reduces learning rate by a multiplicative factor at every training step.

```
scheduler = ExponentialLR(optimizer, gamma = 0.5)
```

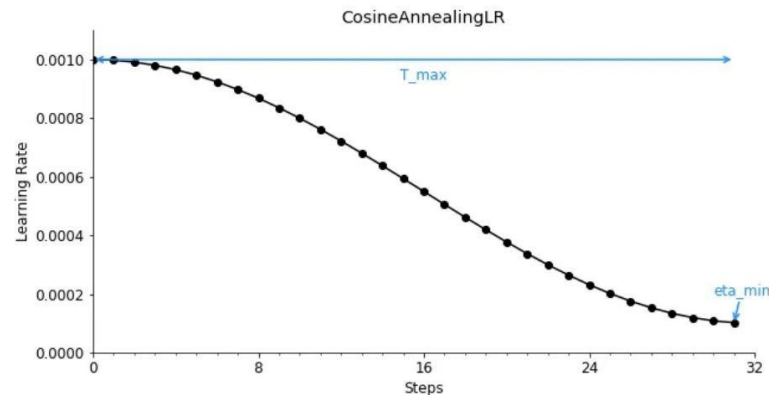
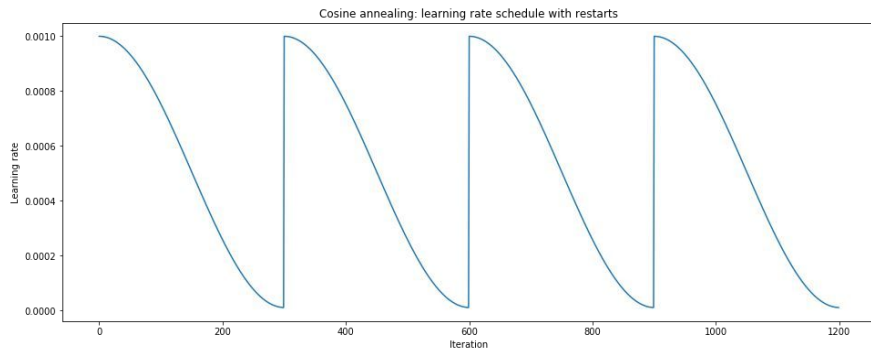


PyTorch Learning Rate Scheduler ExponentialLR (Image by the author)

# CosineAnnealingLR

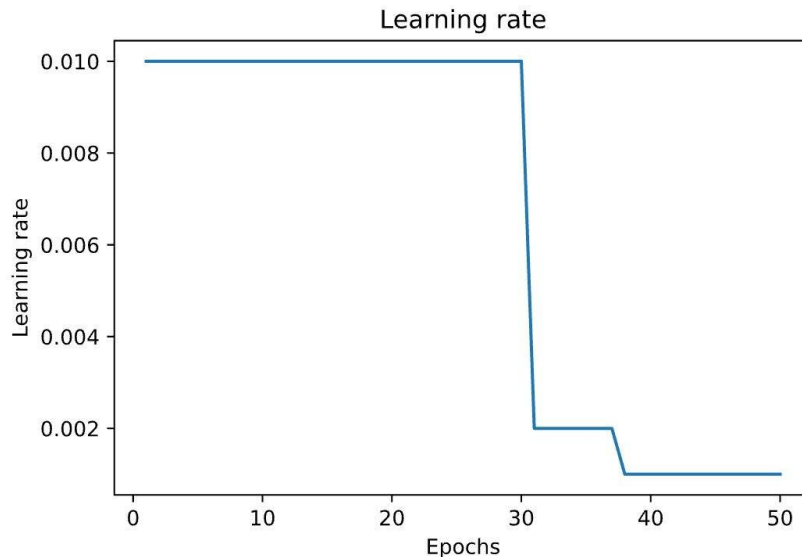
- The [CosineAnnealingLR](#) reduces learning rate by a cosine function.
- While you could technically schedule the learning rate adjustments to follow multiple periods, the idea is to decay the learning rate over half a period for the maximum number of iterations.

```
scheduler = CosineAnnealingLR(optimizer, T_max = 32, eta_min = 1e-4)
```



# ReduceLRonPlateau

Reduce learning rate when a metric has stopped improving. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.



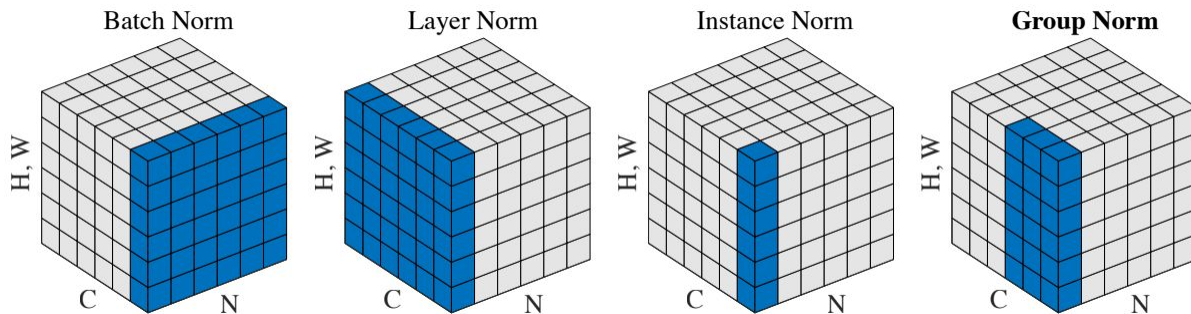
# Ablations

- Normalization Methods

---

# Normalizations

- Batch Norm ([paper](#))
- Layer Norm ([paper](#))
- Weight Norm ([paper](#))
- Instance Norm ([paper](#))
- Group Norm ([paper](#))
- Batch-Instance Norm ([paper](#))
- Switchable Norm ([paper](#))



# BatchNorm

- Normalizing batch of inputs to eliminate internal covariate shift
- `torch.nn.BatchNorm1d(num_features)` - Applies Batch Normalization over a 2D or 3D input
- For 4D inputs (N,C,H,W) - `torch.nn.BatchNorm2d(num_features)`

```
class Network(torch.nn.Module):
    def __init__(self, context):
        super(Network, self).__init__()
        in_size = (1+2*context)*15
        layers = [

            nn.Linear(in_size, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(),
            nn.Dropout(0.1),

            nn.Linear(1024, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(),
            nn.Dropout(0.1),

            nn.Linear(1024, 40)
        ]
        self.layers = nn.Sequential(*layers)

    def forward(self, A0):
        x = self.layers(A0)
        return x
```

# Batch Normalization

## 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

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

- 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.



# Layer Norm

## Problems with Batch Norm

- Poor performance if the batch size is small, possible for high dimensional inputs
- Running mean and variance might not be the best thing to calculate for sequential algorithms like RNNs

Layer normalization calculates mean and variance for each item within the batch

Pytorch syntax - `torch.nn.LayerNorm(normalized_shape)`

# Some Other Methods

- Weight Norm
  - Normalizes weights of each layer
  - `torch.nn.utils.weight_norm(nn.Linear(20, 40), name='weight')`
- Group Norm
  - Applies normalization over a mini-batch of inputs but splitted in groups of size `num_channels/num_groups`
  - `torch.nn.GroupNorm(num_groups, num_channels)`
- Instance Norm
  - Calculates normalization parameters across individual channels/features for each input
  - `torch.nn.InstanceNorm1d(num_features)`
  - `torch.nn.InstanceNorm2d(num_features)`

# Hyperparameter Optimization

---

# What are Hyperparameters?

- Explicitly specified parameters that control the training process

Parameter	Hyperparameter
Parameters are internal to the model - model weights, biases,	Hyperparameters are the explicitly specified parameters that control the training process.
Parameters are essential for making predictions.	Hyperparameters are essential for optimizing the model.
These are learned & set by the model by itself.	These are set manually by a machine learning engineer/practitioner.

# Why do we need to tune hyperparameters?

1. Improve Model Accuracy.
2. Faster convergence for Model.
3. Work with resource constraints (training time, infrastructure, cost requirements etc.).
4. Because I need to reach the HIGH cutoff. 🙄

# Examples of Hyperparameters

- Batch Size
- Learning Rate
- Scheduler Parameters
- Dropout Probability
- Context
- Size of layers/number of layers
- Optimizer
- Weight Initialization

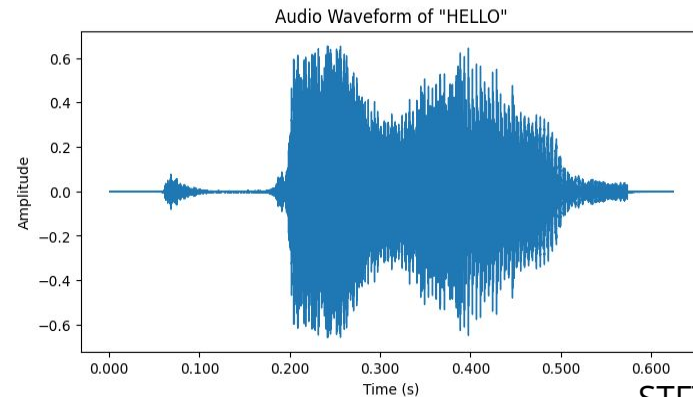
For Convolutional Neural Networks  
(HW2P2)

- Kernel size
- Stride

# Context ( but first ...task preliminaries)

Text: "HELLO"

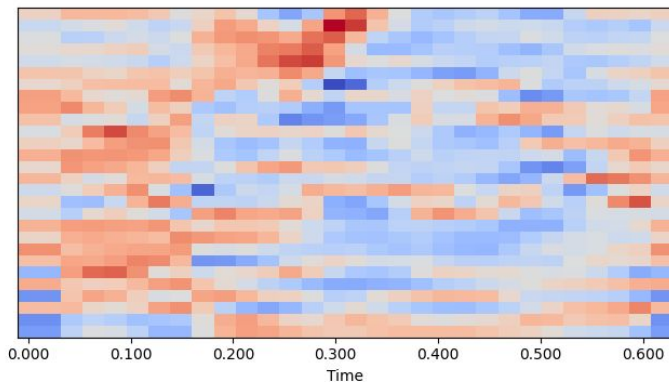
Phonemes: "HH" "AH" "L" "OW"



STFT + Mel



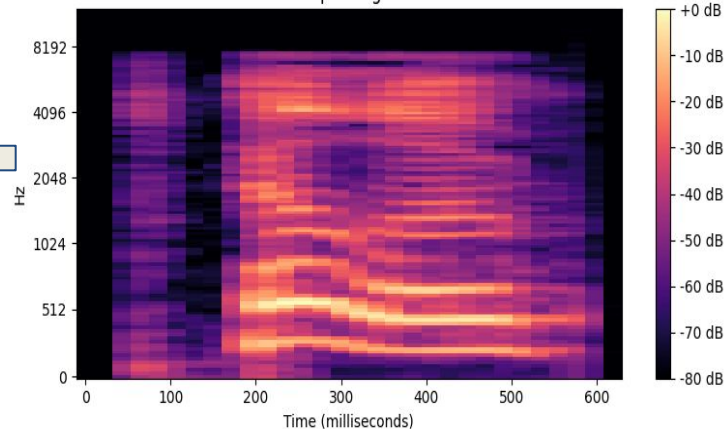
Normalized MFCC



Log + DCT +  
Norm

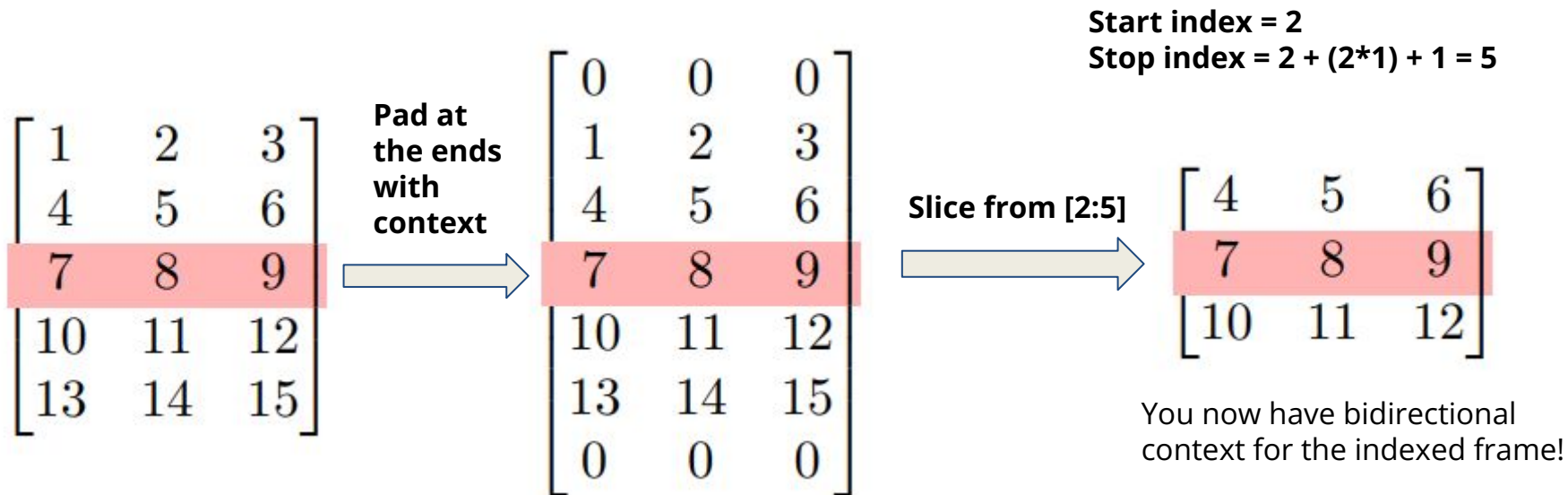


Melspectrogram



# How do we actually add context?

Assume a T x F MFCC as below. Let's say we want the frame at **index 2** with **context of 1**





# Context: a hyperparameter

The case for **longer context**:

(1) Frames are short (20 to 40ms) and may not capture the whole sound

(2) You shall know a ~~word~~ phoneme by the company it keeps. It is easier to predict what a phoneme is if you know what its neighbours are.

The case for **shorter context**: MLPs are not well suited for sequences. The performance will degrade if the context is too high.

Another reason why context that's too long will not be useful here?

**Solution:** Experiment with different context sizes

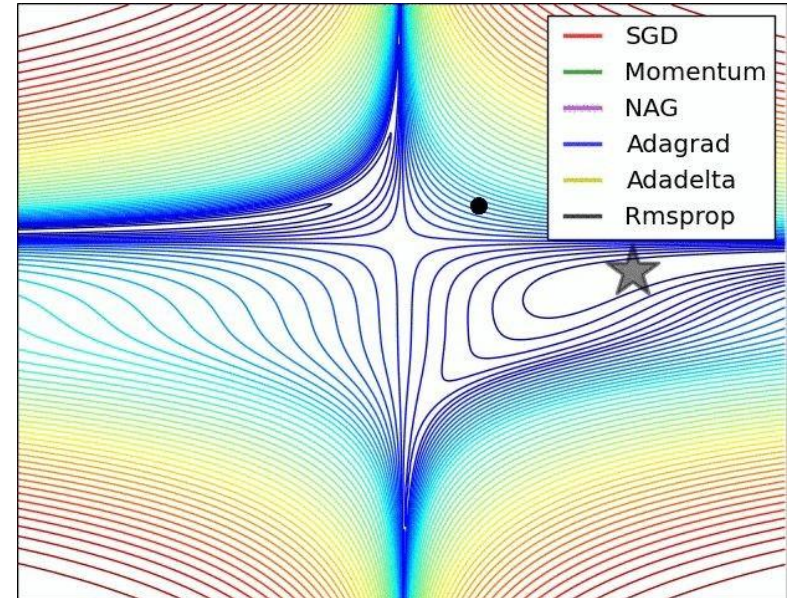
# Optimizer

- Gradient descent is the preferred way to optimize neural networks and many other machine learning algorithms but is often used as a black box.
- How exactly do you train your model in practice? How do you change the parameters of your model, how much, and when?
- Enter *optimizers*. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. The loss function is the guide to the terrain, telling optimizer when it's moving in the right or wrong direction.
- More formally, Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and hyperparameters such as learning rate to minimize loss during training.
- Examples of Optimizer Methods - Stochastic Gradient Descent (SGD), Adam (Adaptive Moment Estimation), AdamW, Sharpness Aware Minimization (SAM), Adagrad, RMSProp, etc.

**Resource to learn about Optimizers:** <https://ruder.io/optimizing-gradient-descent/>

# 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/generalize better.
- More theory on Optimizer Methods will be taught in a lecture by the Professor in the near future :)
- There is a suggested approach of initially using Adam and then switching to SGD:  
<https://arxiv.org/abs/1712.07628>.

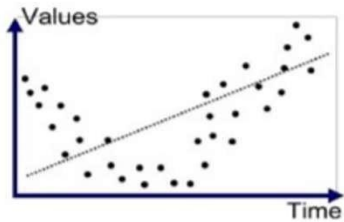


# Overfitting

When the model trains for too long on sample data or when the model is too complex, it can start to learn the “noise,” or irrelevant information, within the dataset.

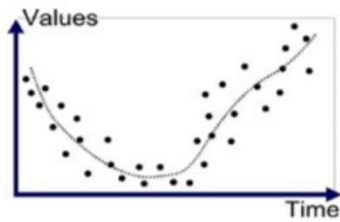
When the model memorizes the noise and fits too closely to the training set, the model becomes “overfitted,” and it is unable to generalize well to new data.

To combat overfitting, we use techniques to minimize the loss while also minimizing the weights. These techniques are often categorized under the name **regularization**.



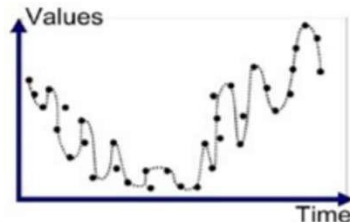
Underfitted

$$y = a + w_0x$$



Good Fit/Robust

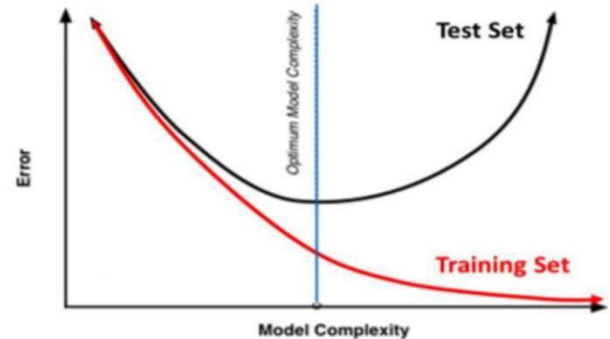
$$y = a + w_0x + w_1x^2 + w_2x^3$$



Overfitted

$$y = a + w_0x + w_1x^2 + w_2x^3 + w_3x^4 + \dots + w_{10}x^{11}$$

Training Vs. Test Set Error



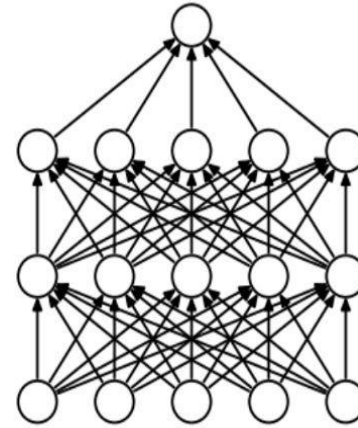
# Dropout

The term “dropout” refers to dropping out the neurons (input and hidden layer) in a neural network.

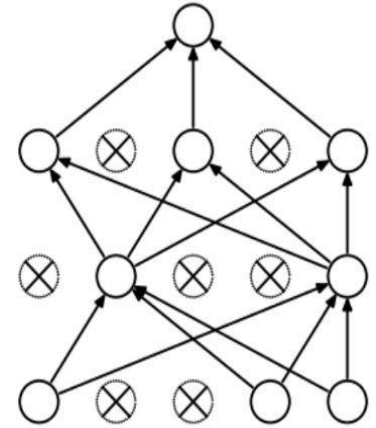
All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network.

Different nodes are dropped by a dropout probability of  $p$  during each forward pass.

This leads to complex co-adaptations, which in turn reduces the overfitting problem because of which the model generalises better on the unseen dataset.



(a) Standard Neural Net



(b) After applying dropout.

# Weight Initialization

- Another facet to improve your performance is using weight initialization for your hidden layers. This can result 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/lorot10a/lorot10a.pdf>  
<https://arxiv.org/pdf/1502.01852.pdf>  
<https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899>

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \quad W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

Xavier

kaiming

# Other Techniques

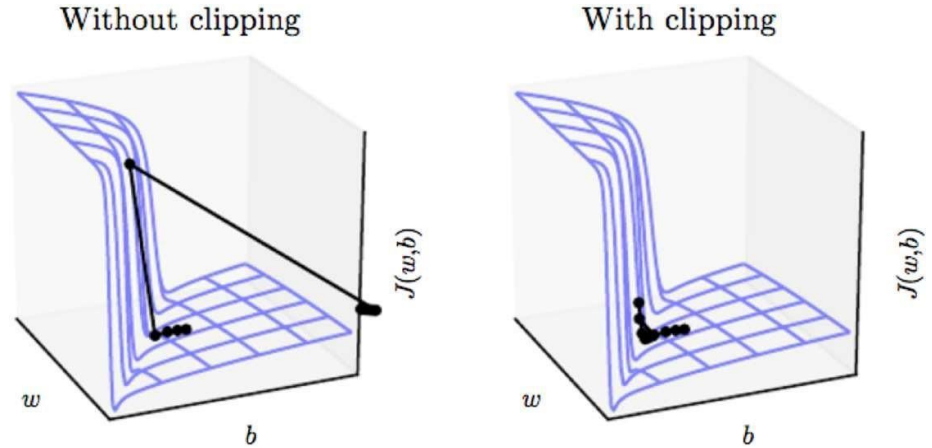
- **Early Stopping**

Literally, just stop the training when you see the validation score decreasing, where overfitting is likely to begin.



- **Gradient Clipping**

Once the gradient is over the threshold, clip and keep them to the threshold value. This helps avoid exploding gradients, especially useful in recurrent networks as you will see later in the course.



# Tips for HW P2s



# Mixed Precision Training

- **Mixed Precision Training (Pytorch > 1.6.0)**
  - combine FP32 and FP16 during training while achieving same accuracy as FP32 training
- **Why use Mixed Precision?**
  - faster training (2-3x)
  - less memory usage
  - larger batch size, larger model, larger input
- **How? What about loss of information?**
  - We keep a **master copy** of weights in FP32.
  - This is converted into FP16 during part of each training iteration (one forward pass, back-propagation and weight update).
  - At the end of the iteration, the weight gradients are converted back to FP32 and used to update the master weights during the optimizer step.

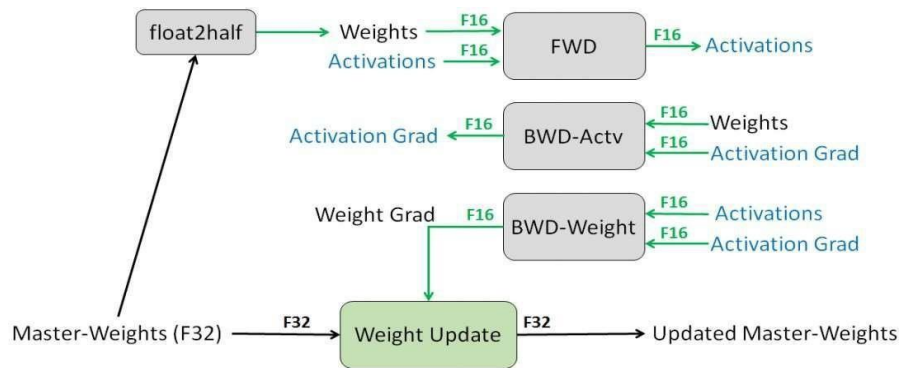
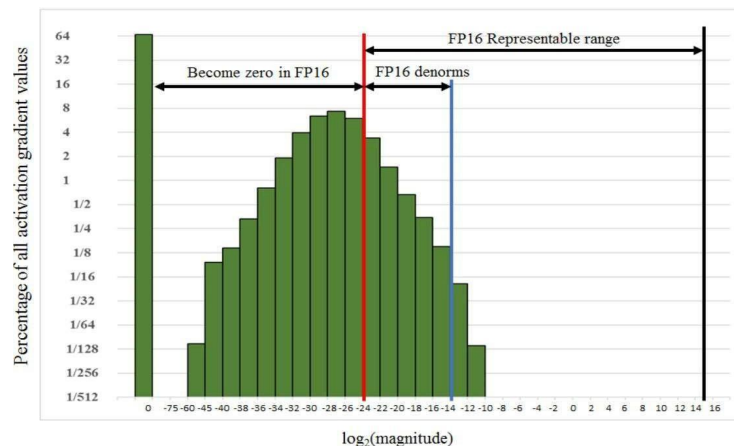


Figure 1: Mixed precision training iteration for a layer.



# Mixed Precision Training

```
# 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()
```

- **Mixed Precision Training works only for some GPUs, including Tesla T4, V100 and A100 GPUs. It does not work with a P100 GPU.**

# Things to keep in mind for HWs

- **DO NOT** spend time exploring all possible architectural and hyperparameter variations yourself. This is where you make use of the course collaboration policy to take help of your study group to divide and conquer your HW ablation studies.
- Spend some time discussing possible variations together and divide it among each other, and log your experiments in your wandb team. Divide and Conquer is the way to succeed in HWs in this course.
- Spending time on a portion/sample of the dataset initially might help you make choices quicker on what works and doesn't work. This can be very useful when exploring wildly different strategies at the beginning, and then filtering out variants that work and training them on the full dataset to eliminate further nuanced variations in the subset of architectures and hyperparameters.

# Things to keep in mind for HWs

- A huge portion of your time might go into just monitoring your training on Colab/AWS and being anxious about your architecture's performance.
- We recommend starting early for this very reason so that you don't have to worry about wasting too much time on architecture search to see what works well and doesn't in the last minute, forcing you to keep your eyes glued to training and wasting time doing nothing else.
- You will be provided with the medium cutoff architecture soon, look out for a post on Piazza regarding it in the coming week, and use it as a first step to ensure you get 70 points.

# Getting the most of Study Groups

- **Why study groups?**
  - Get more from the course
  - Move towards caviar hyperparameter tuning
  - Get to High Cutoffs!!
- **How to get the most out of them?**
  - Contact each other
  - Contact your mentors
  - Join the slack channels
  - Start creating ablation sheets



Trying each and every single way to get to the high cutoff alone




Working with my study group andw all of us getting As

# How to use Wandb effectively

# Tips for Weights and Biases

- You might already be familiar with weights and biases at this point, given we have provided you code to work with in the starter notebook for HW1P2. It is simple code to get you started, but you can do so much more with weights and biases such as hyperparameter sweeps. Please refer to Recitation 0, to learn more about Wandb. We highly recommend using it, at least for logging metrics at the minimum.
- If you haven't already - Use weights and biases as given in the starter notebook to log your experiments.
- Create a team in weights and biases and ask your study group members to join it, so that you can share ablation results with each other as you are conducting experiments as a group. This reduces anxiety in your architecture/hyperparameter search by a LOT and we highly recommend doing this.

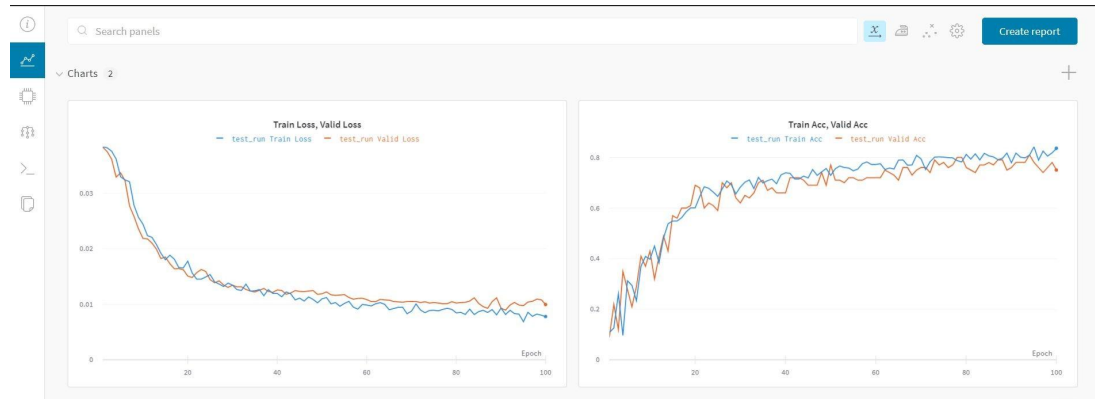
## Profile

 vishhvak

## Teams

idl-s22

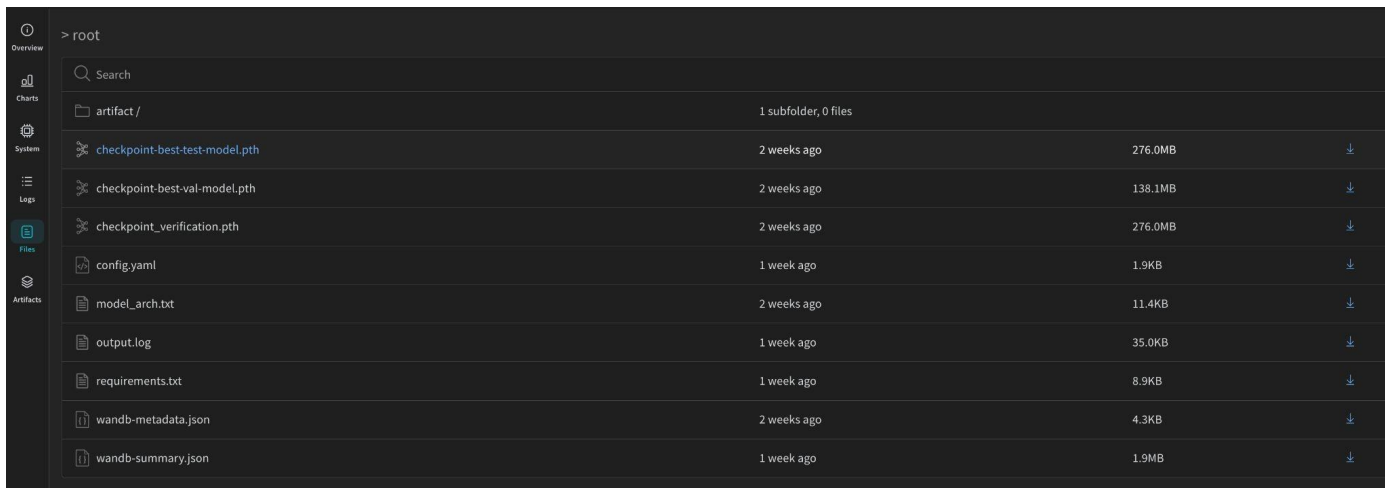
 Create new team



# Version Control and Artifacts

- Save Model Checkpoints, configs, architecture
- Access it from anywhere
- Resume logging from previous runs

```
if test_ver_acc >= best_ver_acc:
    #path = os.path.join(root, model_directory, 'checkpoint' + '.pth')
    print("Saved verification model")
    torch.save({'model_state_dict':model.state_dict(),
               'optimizer_state_dict':optimizer.state_dict(),
               'scheduler_state_dict':scheduler.state_dict(),
               'val_acc': test_ver_acc,
               'epoch': epoch}, './checkpoint_verification_finetune.pth')
    best_ver_acc = test_ver_acc
    wandb.save('checkpoint_verification_finetune.pth')
```



> root			
Search			
artifact/	1 subfolder, 0 files		
checkpoint-best-test-model.pth	2 weeks ago	276.0MB	↓
checkpoint-best-val-model.pth	2 weeks ago	138.1MB	↓
checkpoint_verification.pth	2 weeks ago	276.0MB	↓
config.yaml	1 week ago	1.9KB	↓
model_arch.txt	2 weeks ago	11.4KB	↓
output.log	1 week ago	35.0KB	↓
requirements.txt	1 week ago	8.9KB	↓
wandb-metadata.json	2 weeks ago	4.3KB	↓
wandb-summary.json	1 week ago	1.9MB	↓



# Wandb for sweeps

```
[ ] # Initialize the sweep and set the method (grid, random or bayes"ian")
```

```
sweep_config = {  
    'method': 'random'  
}
```

```
[ ] # What is the objective of the sweep (minimize loss, maximize accuracy)
```

```
metric = {  
    'name': 'loss',  
    'goal': 'minimize'  
}  
sweep_config['metric'] = metric
```

```
▶ # Hyperparameters to work with
```

```
parameters_dict = {  
    'optimizer': {  
        'values': ['sgd', 'adam']  
    },  
    'learning_rate': {  
        'distribution': 'uniform',  
        'min': 2e-4,  
        'max': 1e-1  
    },  
    'batch_size': {  
        'distribution': 'q_log_uniform_values',  
        'q': 4,  
        'min': 16,  
        'max': 128  
    },  
    'epochs': {  
        'value': 5  
    }  
}  
sweep_config['parameters'] = parameters_dict
```

Covered in Recitation 0P- part2

[Colab notebook](#)

[Video](#)

Insights from sweep:

<https://wandb.ai/11785-sg/CIFAR-Sweep/sweeps/5h4vbyqk>



# HW1P2

## Tips

# Hints for HW1P2:

- Always keep this in mind: **“Practice maketh a man 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 training data set. Note: **Do not shuffle the test or validation set.**
- **Choose learning rate wisely:** too large LR will result in the model taking huge steps in weight updates and diverging/not converging 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 BatchNorm 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.

# Hints for HW1P2:

- Explore ensembling - some examples are (more on this will be explained in a future lecture and recitation)
  - Training multiple models and taking a majority vote during classification
  - self-ensemble: average the parameters of your model at different training epochs.
  - Remember, you are however limited to 24 million parameters in total for your final overall network.
- Don't forget `optimize.zero_grad()` while training, and `torch.inference_mode()` while evaluating.
- Experiment with LR schedulers, they are very crucial in improving model performance - CosineAnnealing, ExponentialLR, StepLR, MultiStepLR, ReduceLROnPlateau, etc are some examples to explore.
  - Learning rate must shrink with time for convergence!
- Try to use `torch.cuda.empty_cache()` and `del` to release all unoccupied cached memory.

# More Hints for HW1P2:

- 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, unfortunately 🙄)
- If you are getting CUDA related errors, switch your code to CPU and run it - more often than not, it will give you a better error output and point you towards what is wrong/buggy with your code.
- Setting `bias = False` in weight layers before BatchNorms might be useful, as each BatchNorm layer will re-center the data anyway, removing the bias and making it a useless trainable parameter.
- Try possible tricks or models that you can find in the papers or posts online!
- Always Remember to checkpoint your model after each epoch in your code!



**Before DDL, get an extremely good result**

**I will reach the top in leaderboard!  
I am unbeatable!!!!**

**Forget to save model and Colab collapses**



**Let me die.....**

**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!**