11-485/685/785
Fall 2023
Recitation-2

Network Optimization

Data Manipulation, Model Architecture, Hyperparameter Optimization

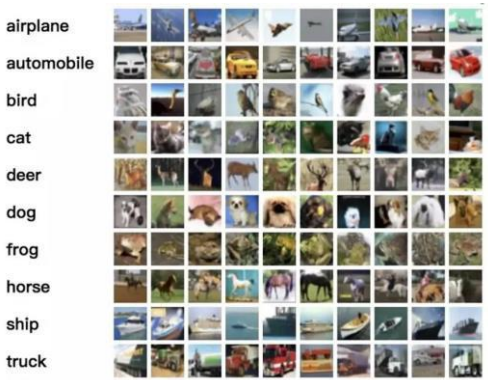
Josh Li | Harshit Mehrotra

Data Manipulation

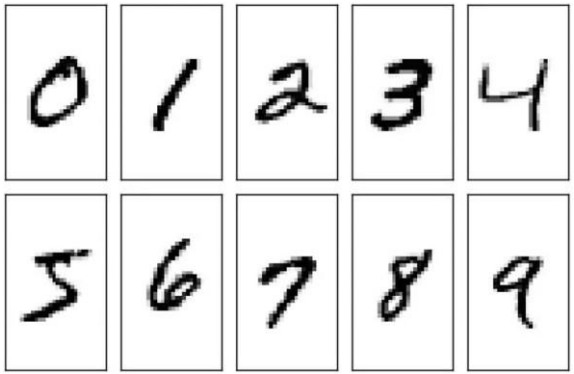
- Data Quality
- Data Augmentation
- Data Normalization

Data Quality

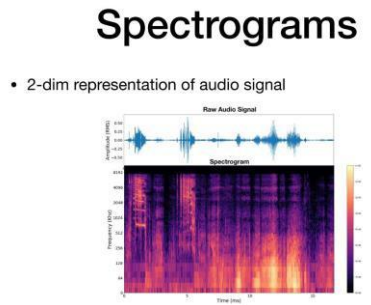
- It's obvious how important data is for deep learning. We need clean large datasets for obtaining high accuracies especially for tasks such as machine translation, sentiment analysis, etc.
- The amount of data required also depends on the dimensionality of the data. The higher the dimensionality, the 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

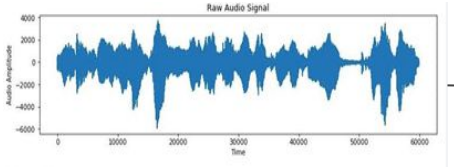


MNIST



Data Cleaning (H1P2) @127

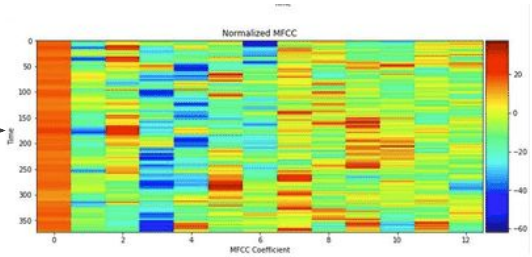
Speech Waveforms



LibriSpeech ASR corpus

Preprocessing

Mel Frequency Cepstral Coefficient
- MFCCs



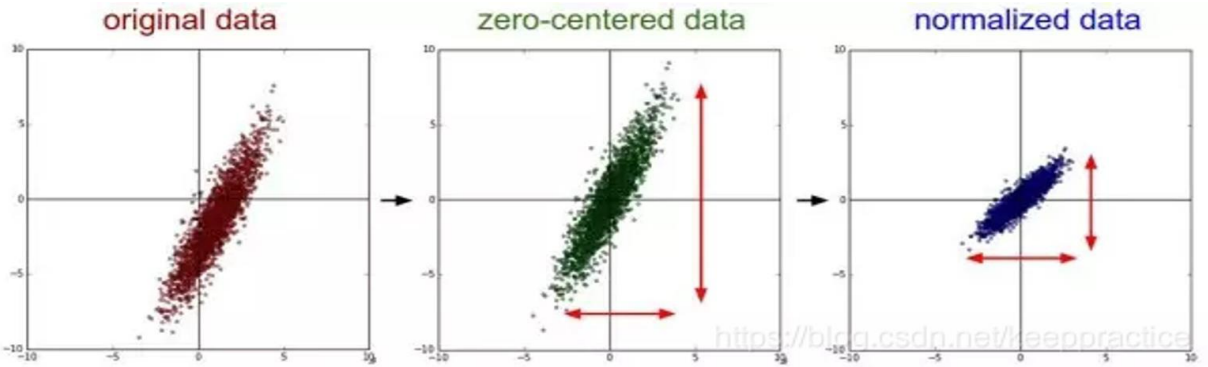
Input data for H1P2

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 (Important)

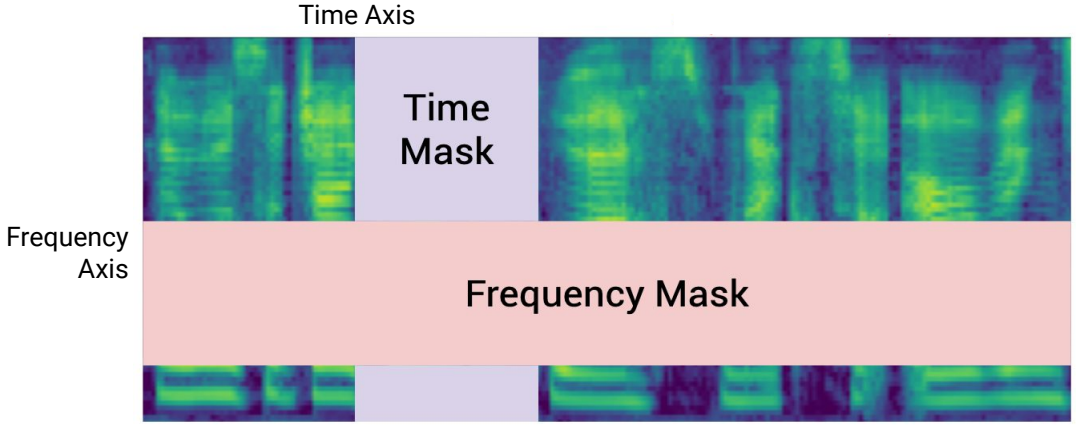
- Cepstral mean normalization (CMN) is a computationally efficient normalization technique for robust speech recognition and is a technique used to normalize the cepstral coefficients of a speech signal.
- CMN minimizes distortion by noise contamination for robust feature extraction.
- CMN has been used 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.

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 HW's.

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

Time and Frequency Masking (H1P2)



Src - https://spectra.mathpix.com/article/2021.09.00002/asr-data-augmentation

MFCC Inputs

TIMEMASKING

CLASS torchaudio.transforms.TimeMasking(
time_mask_param: int,
iid_masks: bool = False,
p: float = 1.0
) [SOURCE]

Apply masking to a spectrogram in the time domain.

Devices CPU, CUDA Properties Autograd, TorchScript

Proposed in *SpecAugment* [Park *et al.*, 2019].

FREQUENCYMASKING

CLASS torchaudio.transforms.FrequencyMasking(
freq_mask_param: int,
iid_masks: bool = False
) [SOURCE]

Apply masking to a spectrogram in the frequency domain.

Devices CPU, CUDA Properties Autograd, TorchScript

Proposed in *SpecAugment* [Park *et al.*, 2019].

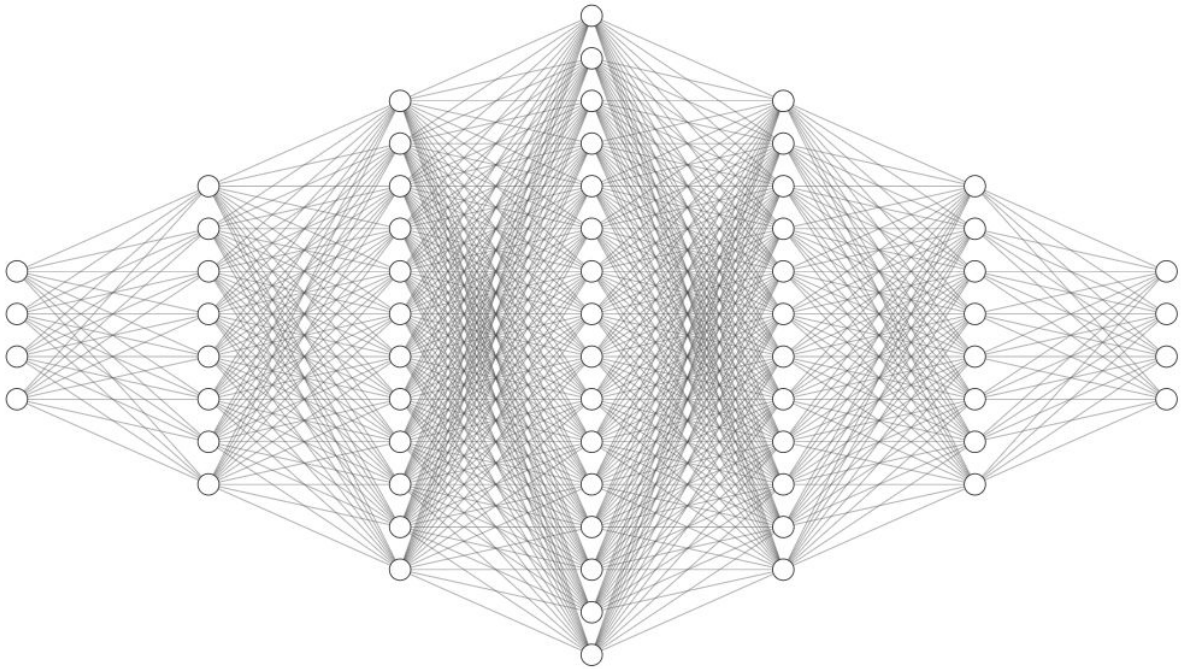
Model Architecture

- How do I think about Model Architecture for HW1P2? 🤔

Model Architecture

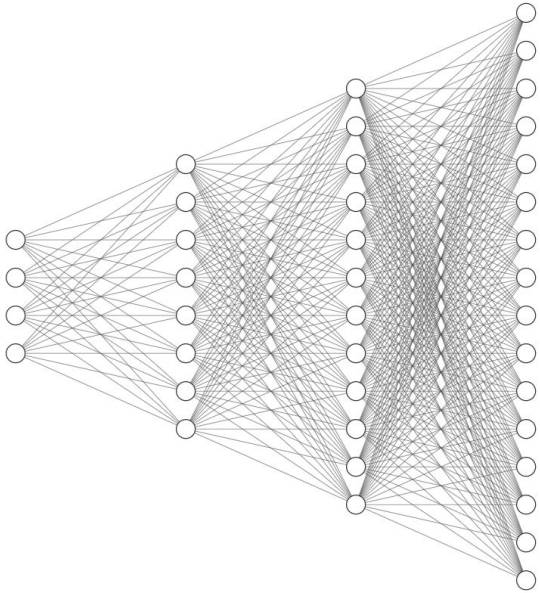
- As you might know already, one of your two biggest tasks in HW1P2 is to find the right model architecture for your MLP to be able to predict the phoneme of a given frame with context.
- Experimenting with finding the right model architecture is not easy - and takes a lot of time in a deep learning task.
- One of the ways to find useful architectures for your tasks is to 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?)

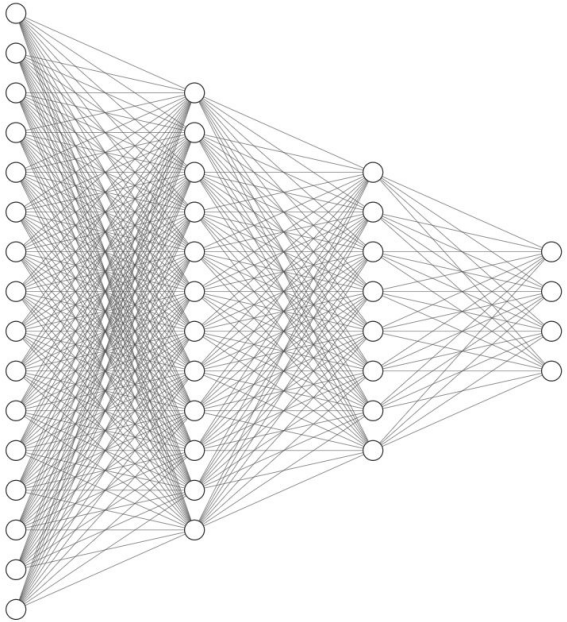
Model Architecture – Diamond



Diamond Architecture

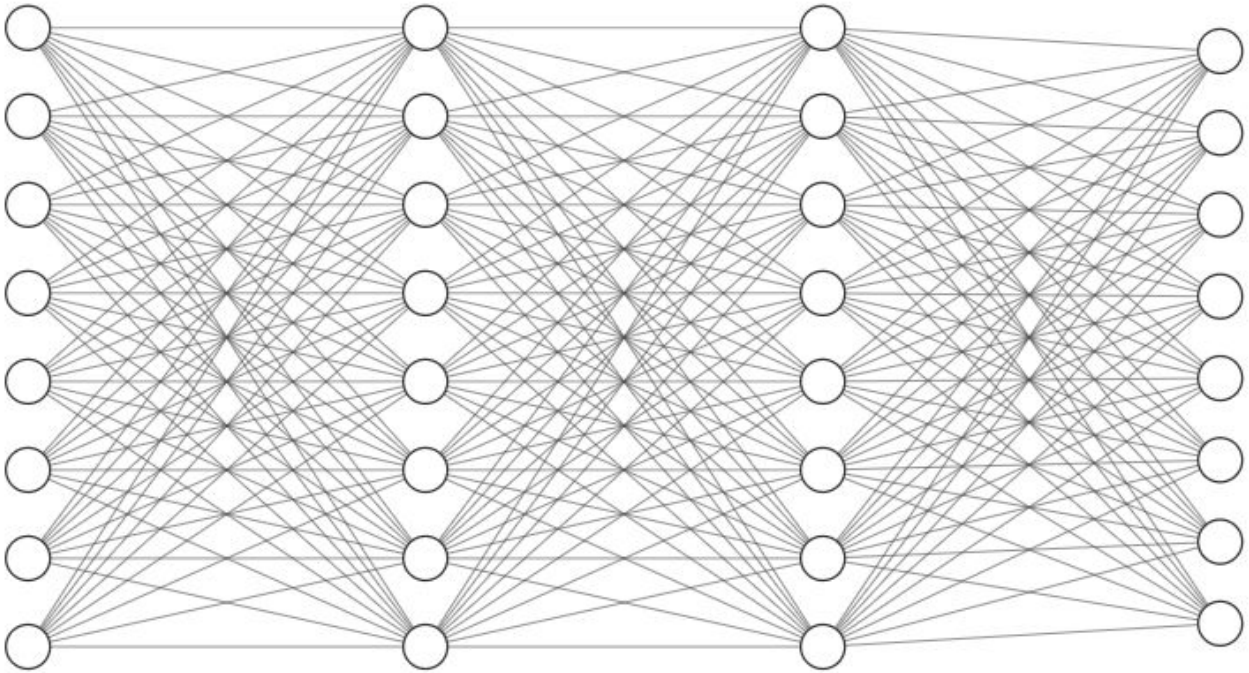
Model Architecture – Pyramids





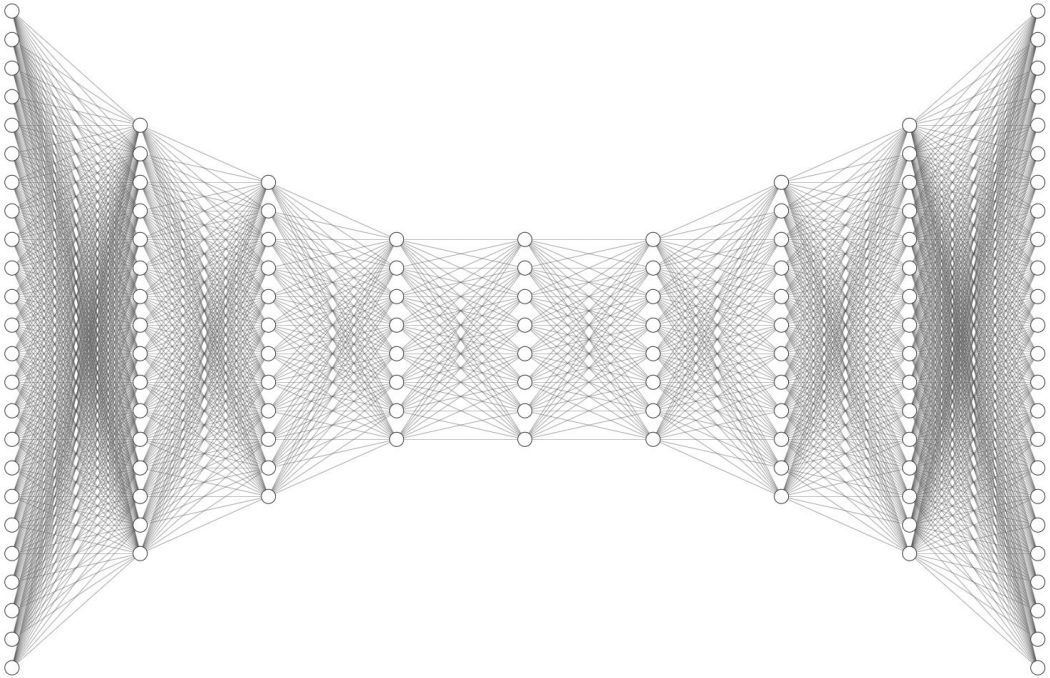
Pyramid/ Inverse-Pyramid

Model Architecture – Cylinder



Cylinder

Model Architecture



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

Hyperparameter Optimization

- What are Hyperparameters?

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. 🙃

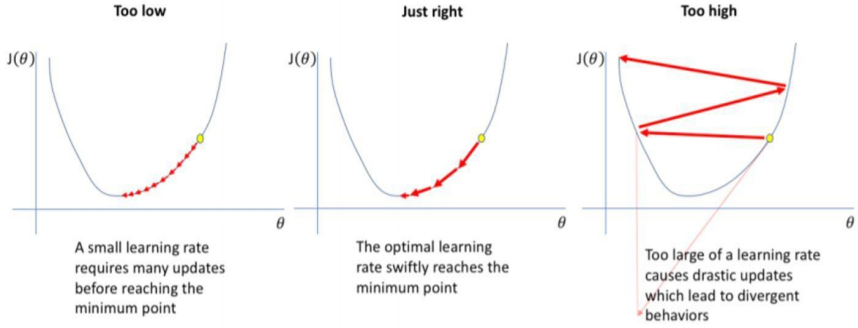
What are Hyperparameters?

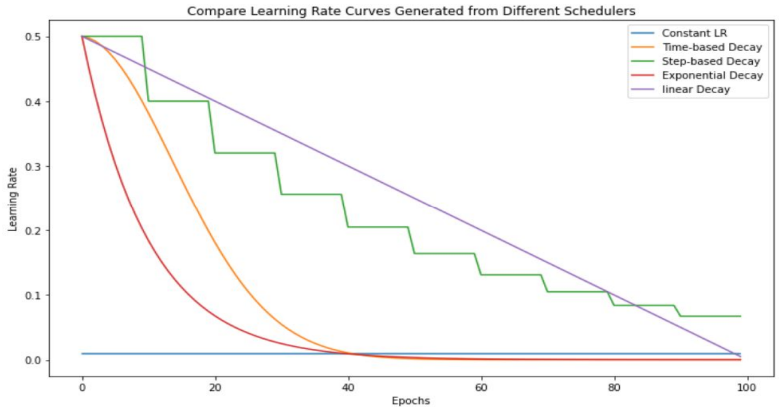
- Explicitly specified parameters that control the training process

Parameter	Hyperparameter
• Parameters are internal to the model. • These are learned & set by the model by itself.	• Hyperparameters are the explicitly specified parameters that control the training process. • These are set by the practitioner
• Model Weights	• Batch Size, Learning Rate, Scheduler Parameters, Dropout Probability • Context, Size of layers/number of layers • Optimizer, Weight Initialization

Learning Rate and Scheduling

- 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).

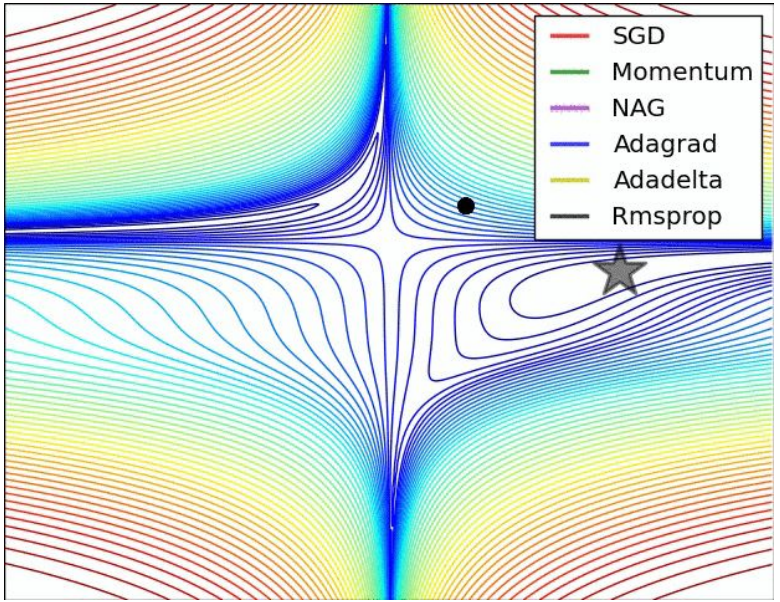




Ref for LR schedulers: https://pytorch.org/docs/stable/optim.html

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.



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, by 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 the 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/

Advantages and Disadvantages

- **Stochastic Gradient Descent (SGD):** This is a simple and widely-used optimizer that updates the model's parameters in the direction of the negative gradient of the loss function.
- **Adam:** Adaptive Moment Estimation (Adam) is an optimization algorithm that is based on gradient descent. It uses the moving averages of the parameters to provide a running estimate of the second raw moments of the gradients; the square of the gradient.
- **Adagrad:** Adagrad adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.
- **Adadelta:** Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.
- **RMSprop:** RMSprop is an extension of Adagrad that also addresses its monotonically decreasing learning rate. It uses a moving average of squared gradient to scale the learning rate.
- **Momentum:** Momentum is an optimization algorithm that helps accelerate gradient descent by updating the parameters in the direction of the accumulated past gradients.

Each optimizer has its own strengths and weaknesses and the choice of optimizer depends on the problem, dataset and the network architecture. It's worth trying multiple optimizers and comparing their performance on the specific problem.

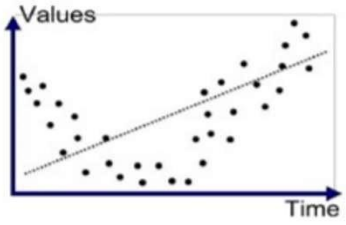
Resource to learn about Optimizers: https://ruder.io/optimizing-gradient-descent/

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.

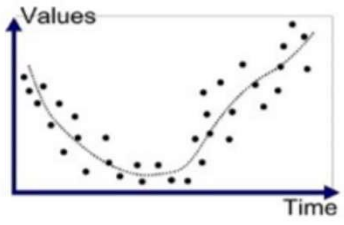
When the model memorizes the noise and fits too closely to the training set, the model becomes “overfitted,” and it is unable to generalize well to new data.

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



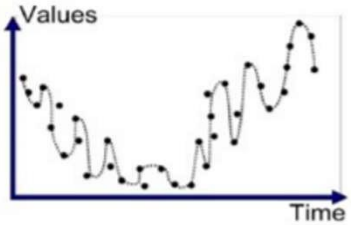
Underfitted

$y = a + w_0 x$



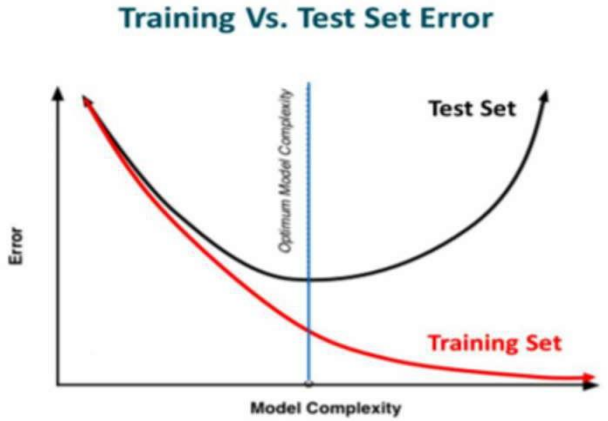
Good Fit/Robust

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



Overfitted

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

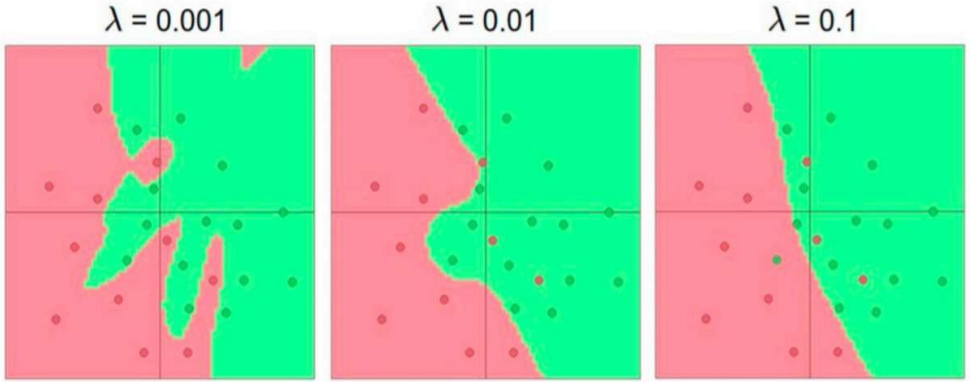


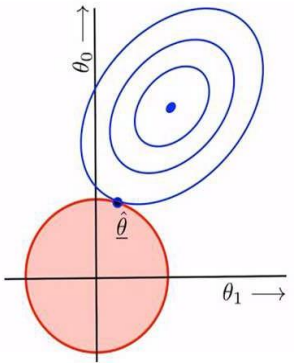
L1/L2 Regularization

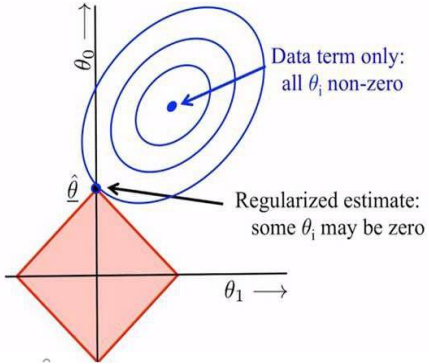
It involves adding an extra element to the loss function, which punishes our model for being too complex or, in simple words, for using too high values in the weight matrix (usually, lambda is 1e-4 or 1e-5). Lambda is also known as weight decay, as you will find in Pytorch's optimizers.

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

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







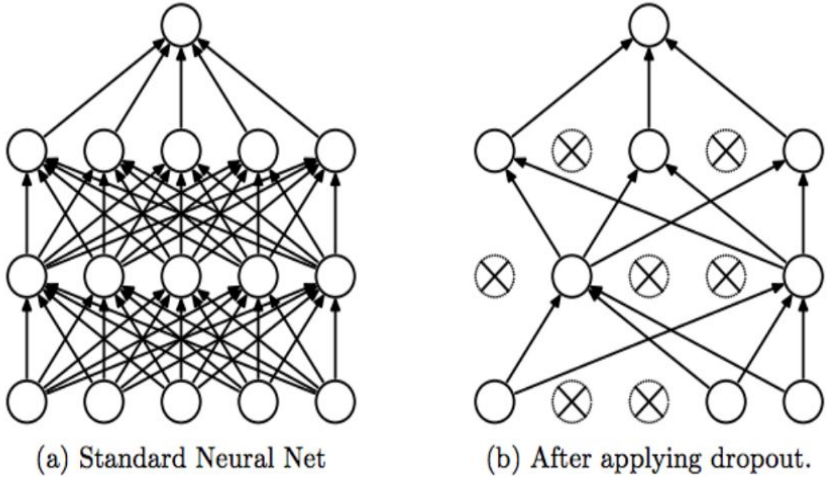
Dropout (Regularizing your model)

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/glorot10a/glorot10a.pdf
https://arxiv.org/pdf/1502.01852.pdf
https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

Xavier

$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

kaiming

Batch Normalization

- **Wildly successful and simple technique for accelerating training and learning better neural network representations**
- The key motivation behind BatchNorm is *internal covariate shift*. It is defined as the change in the distribution of network activations due to the change in network parameters during training.
- At every epoch of training, weights are updated and a different minibatch is being processed, which means that the inputs to a neuron is slightly different every time.
- As these changes get passed on to the next neuron, it creates a situation where the input distribution of every neuron is different at every epoch. This co-adaptation problem, significantly slows learning.
- Hence, these shifts in input distributions can be problematic for neural networks, especially deep neural networks that could have a large number of layers.

Performing Batch Normalization

- Batch normalisation normalises a layer input by subtracting the mini-batch mean and dividing it by the mini-batch standard deviation.
- However, if each layer is normalised, the weight changes made by the previous layer and noise between data is partially lost, as some non-linear relationships are lost during normalisation. This can lead to suboptimal weights being passed on.
- To fix this, batch normalisation adds two learnable parameters, gamma γ and beta β, which can scale and shift the normalised value.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

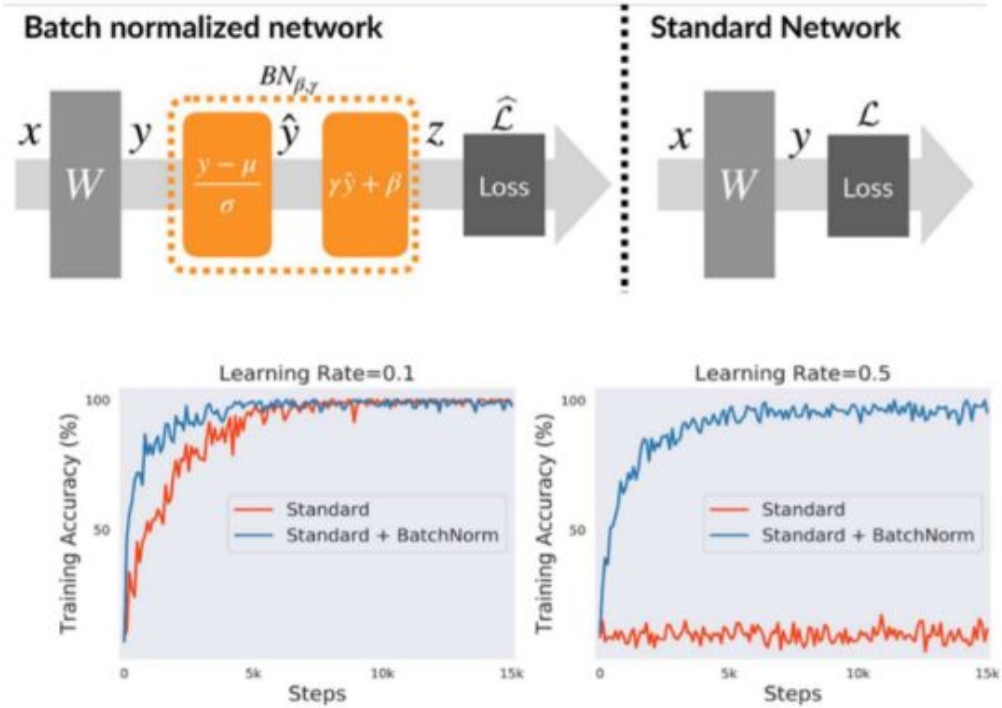
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

- During training, the mean and standard deviation are calculated using samples in the mini-batch. However, in testing, it does not make sense to calculate new values.

- Hence, batch normalisation uses a running mean and running variance that is calculated during training.



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

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

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.

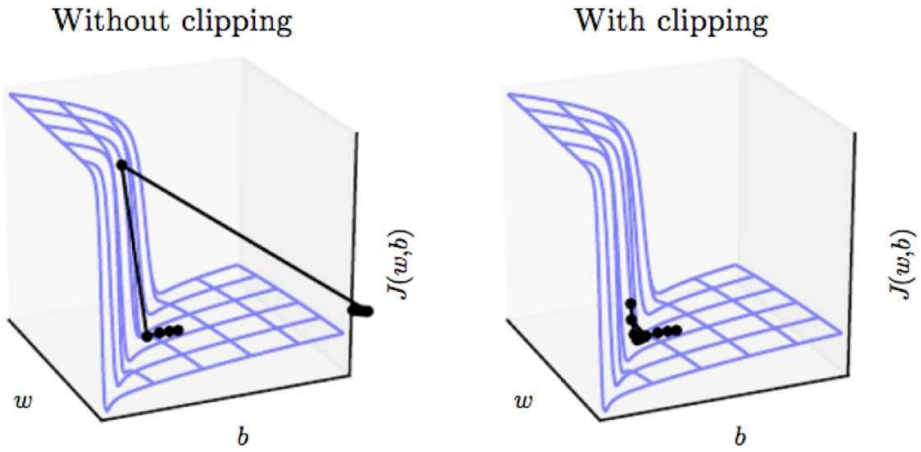
Other Techniques

- **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.



Efficient Training and Logging

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.

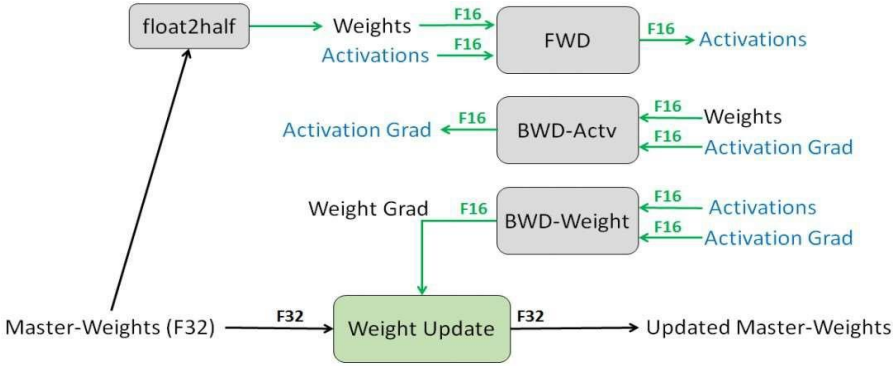
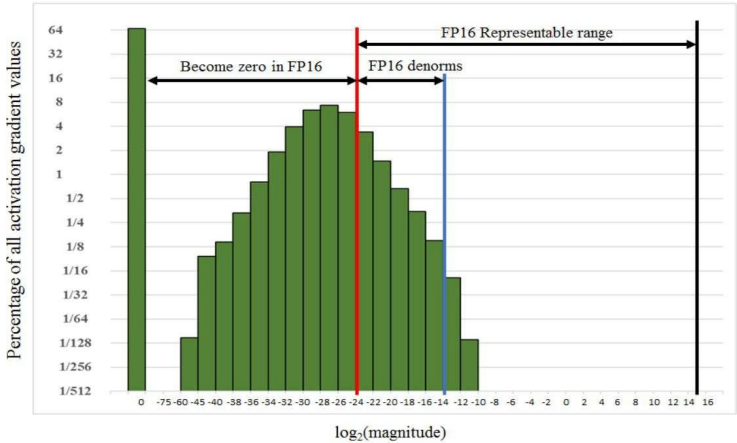


Figure 1: Mixed precision training iteration for a layer.



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.**

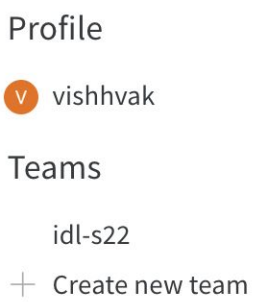
https://pytorch.org/docs/stable/notes/amp_examples.html

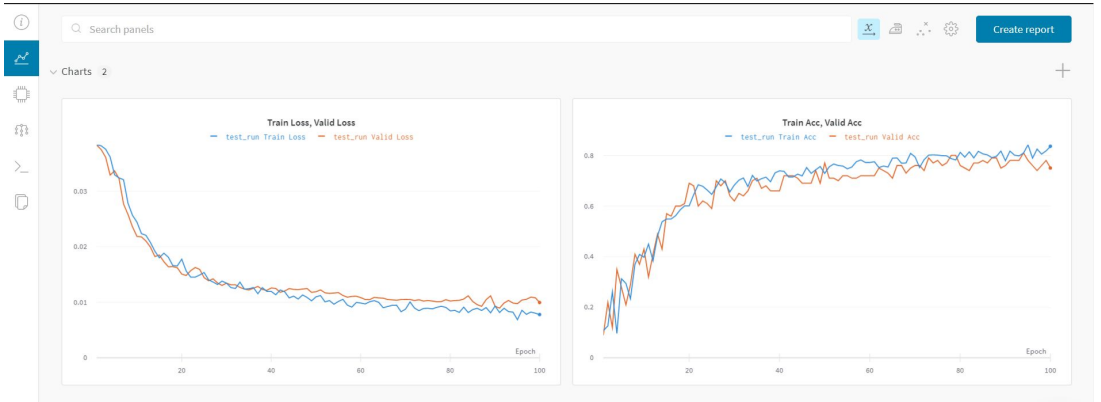
Tips for Weights and Biases

- 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 0H, 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.





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.

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 20 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(), del and gc.collect() 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!

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!

Thank you! :) Have fun!