Summary of the experiments

Belkin Experiments

Neural Network

They use a fully connected network with one hidden layer. Datasets they use are **MNIST** and **CIFAR10** but they don't use the full datasets. For **MNIST**, `4000` training samples are randomly sampled among the `60,000` training samples. They don't mention whether they sub-sample the test set or not. Since the test dataset of **MNIST** has `10000` samples, it does not seem usual to train on `4000` samples and test on `10000` so I also sub-sampled the test set to be `1000` samples. It should not affect the overall results. The optimizer is vanilla `SGD` with initial learning rate of `1e-2` and momentum of `0.95`. The also use `MultiStepLR` learning rate scheduler with milestones of `500` and gamma of `0.9`. Each model is trained for `6000` epochs. The also use the `MSELoss` function.

Weight Reuse Before Interpolation Threshold?!

In the paper they claim:

> The ERM optimization problem in this setting is generally more difficult than that for RFF
> and ReLU feature models due to a lack of analytical solutions and non-convexity of the problem.
> Consequently, SGD is known to be sensitive to initialization. To mitigate this sensitivity, we
> use a "weight reuse" scheme with SGD in the under-parametrized regime (N < n), where the
> parameters obtained from training a smaller neural network are used as initialization for training larger networks. This procedure, detailed below, ensures decreasing training risk as the number of parameters increases. In the over-parametrized regime (N ≥ n), we use standard (random) initialization, as typically there is no difficulty in obtaining near-zero training risk.

But is this really true? These are the plots with both weight reuse and without it:

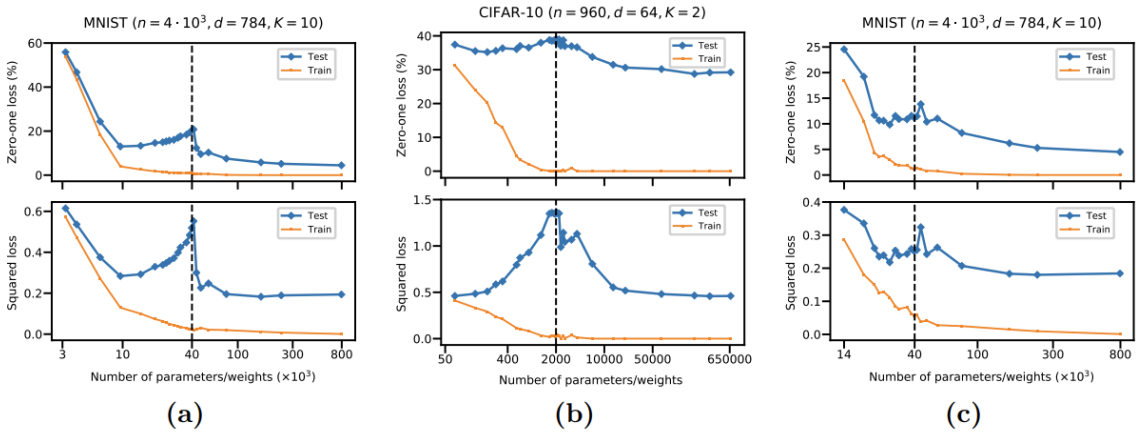
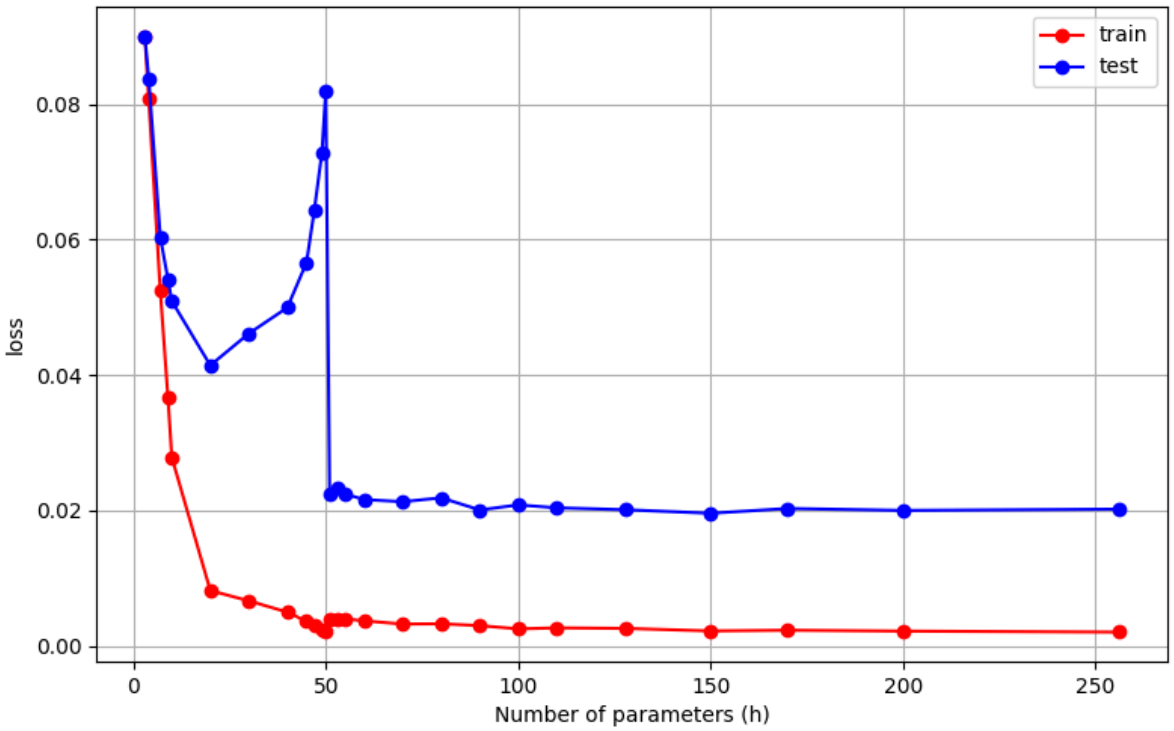
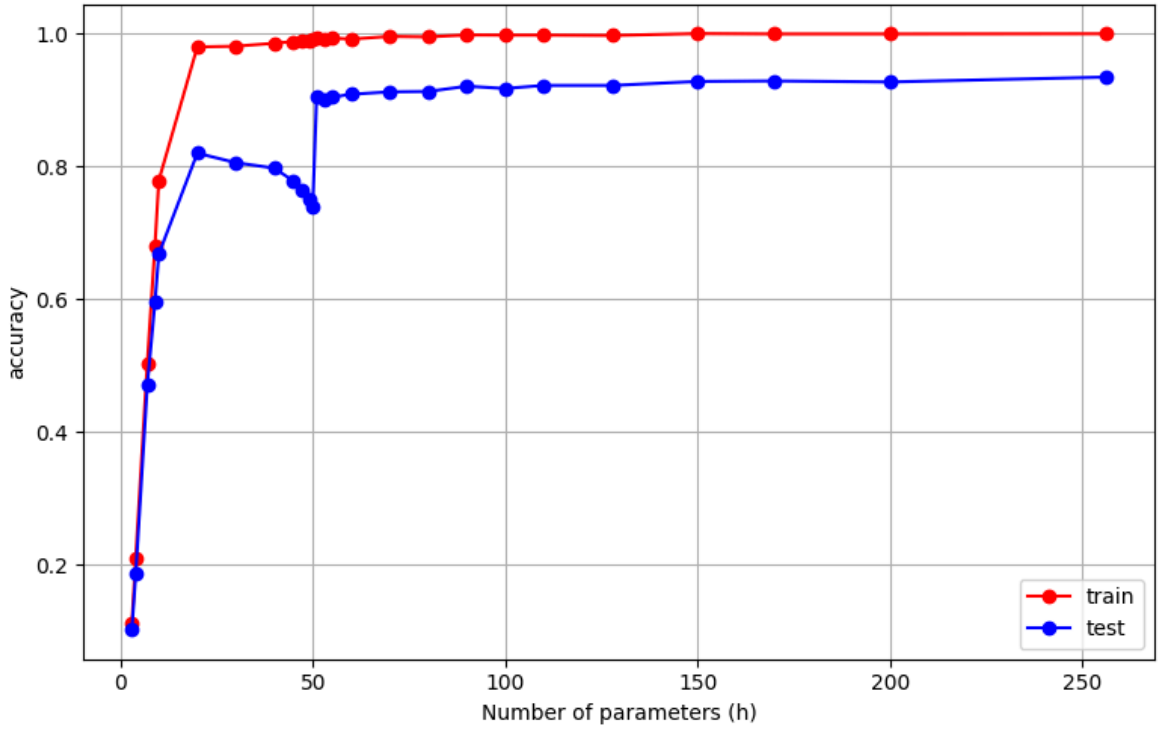


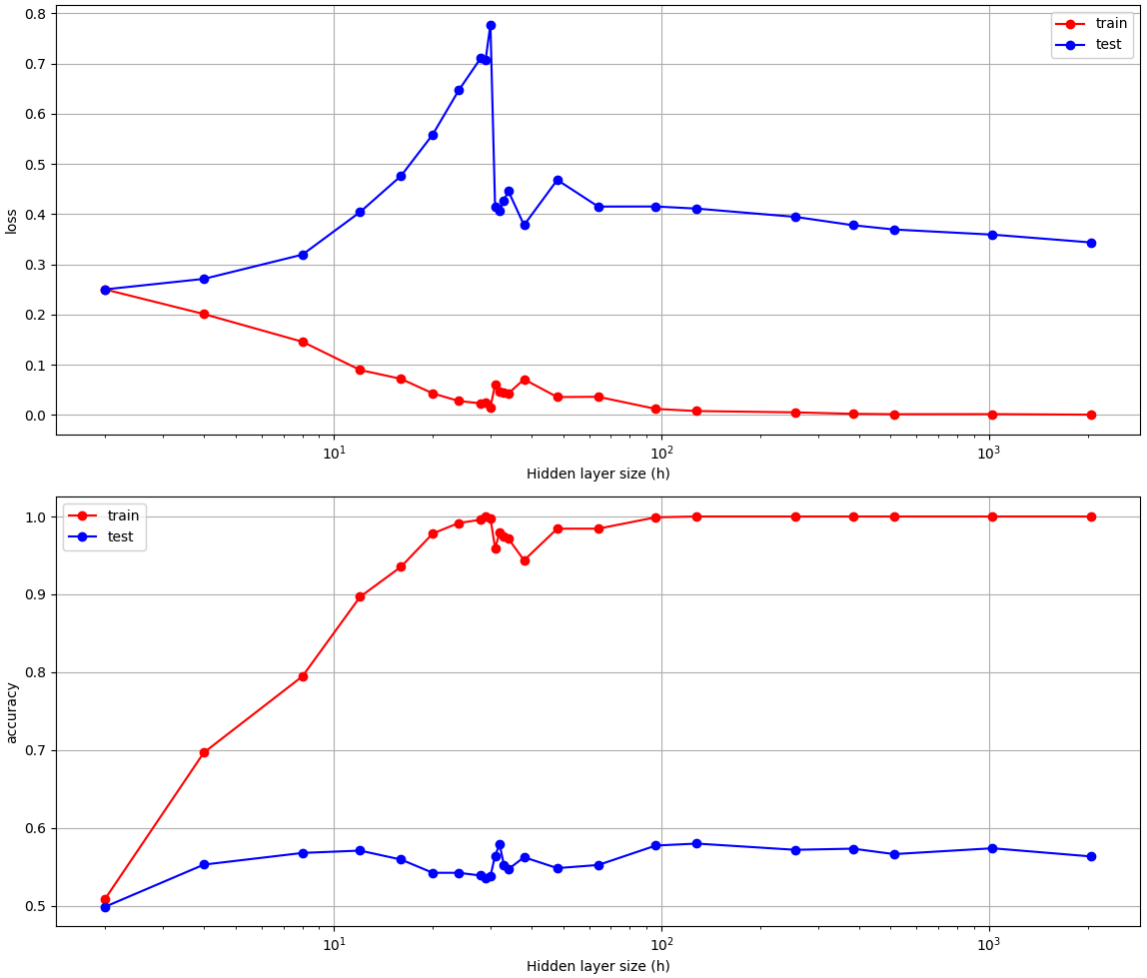
Figure 9: **Double descent risk curve for fully connected neural networks.** In each plot, we use a dataset with n subsamples of d dimension and K classes for training. We use networks with a single hidden layer. For network with H hidden units, its number of parameters is $(d+1) \cdot H + (H+1) \cdot K$. The interpolation threshold is observed at $n \cdot K$ and is marked by black dotted line in figures. (**a**) Weight reuse before interpolation threshold and random initialization after it on MNIST. (**b**) Same, on a subset of CIFAR-10 with 2 classes (cat, dog) and downsampled image features (8×8). (**c**) No weight reuse (random initialization for all ranges of parameters).

They claim without weight reuse (column **C** plots) the DD phenomena happens but the peaks are not as sharp as the weight reuse case. They claim that interpolation threshold should happen around `40,000` parameters (which is the number of training samples times number of classes as they claim this is the formula to get the number of parameters at interpolation threshold) for the **MNIST** experiment which considering the input dimension of `784 (28×28)` and output dimension of `10`, the hidden size of the FC network would be approximately `50` at the interpolation threshold. This is my experiment with Belkin setting for **MNIST** sub-sampled:

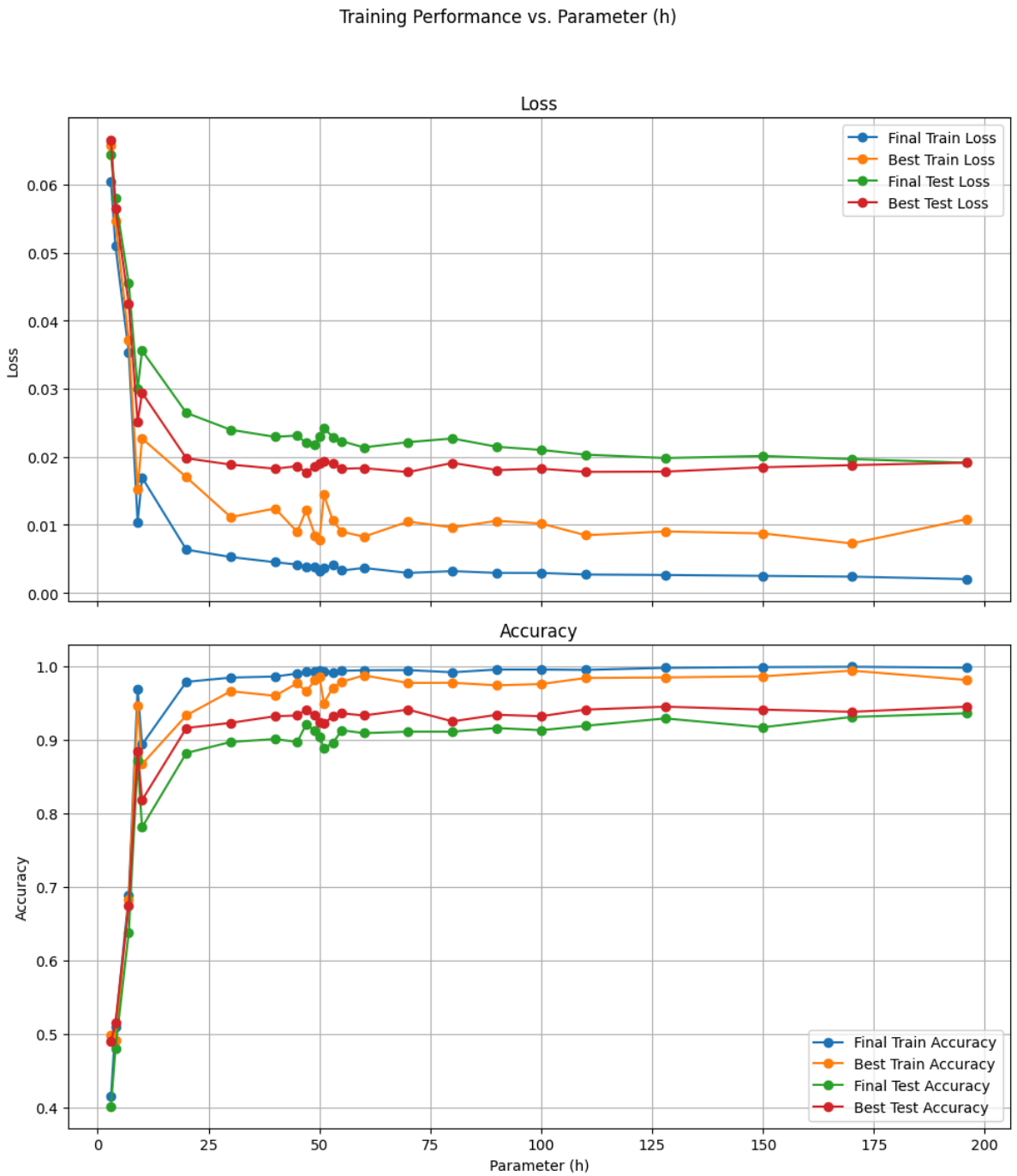




The difference with the paper plots might be due to paper using log scale on the x axis. But the results are the same. It seems the Double Descent is happening. Also these are the curves for the **CIFAR10** experiment (column **B** in the figure from the paper) where they use only `960` samples from only two classes:

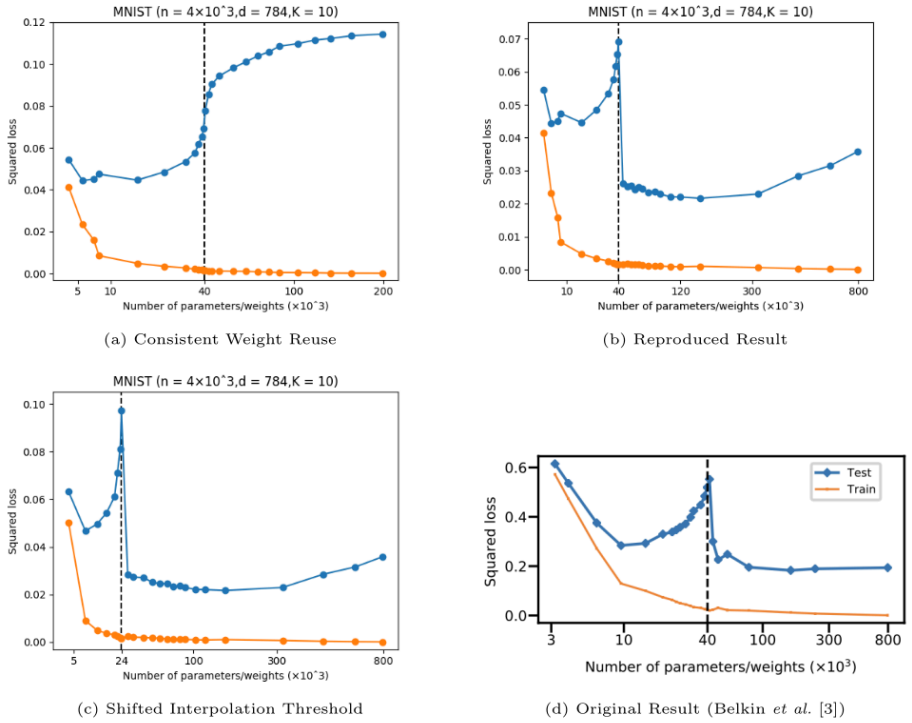


Also here it seems double descent is happening. But what if we don’t use weight reuse:



This is not similar to what they show in column **C** of their plots. The small oscillations in the final test loss does not resemble a double descent curve and is due to higher sampling (more models with parameters in that range). Also this pape r shows the same thing. In the following figure they show that if we continue reusing weights, the final test loss goes up —plot (a) in the figure below—. They also show if we use weight reuse until a lower threshold

(`24×10e3` while Belkin claims it should be at `40×10e3` as in plot (d)), we see the same patter which Belkin claims to be double descent —plot (c) in the figure below—. This means, with Belkin setting we can choose the interpolation threshold to be at any capacity!!!



(a) Consistent Weight Reuse

(b) Reproduced Result

(c) Shifted Interpolation Threshold

(d) Original Result (Belkin *et al.* [3])

Figure 3: Experiment Result for a fully connected one-layer neural network with weight-reuse initialization scheme trained on MNIST and a comparison to the result in Belkin *et al.* [3]. ($n = 4 \times 10^3, d = 784, K = 10$)

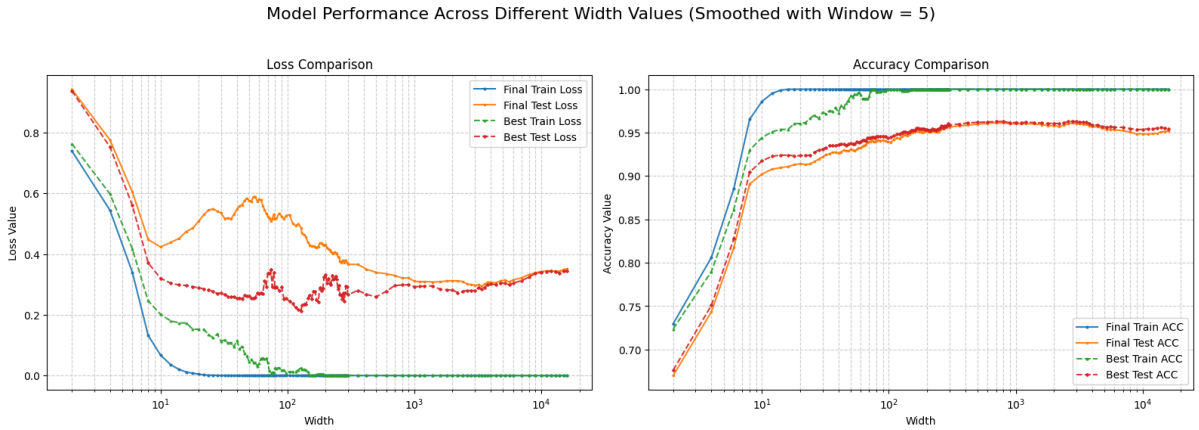
This clearly shows that the setting Belkin used for their experiments on neural networks is flawed. But, does this mean that double descent does not happen at all in this setting or this happens somewhere else in the parameter range?

Random Initialization, But Different Optimizer and Loss Function

I decided to check whether the double descent phenomenon happens in the task specified by Belkin setting or not (**MNIST** sub-sampled). Since the default setting did not show double descent, I decided to change the optimizer. I found it is easier to find the interpolation threshold (that is interpolating all training samples and thus fitting the noise) when I use `Adam` optimizer. So, the optimizer is changed to `Adam` the learning rate is `1e-4` and betas values are default `(0.9,`

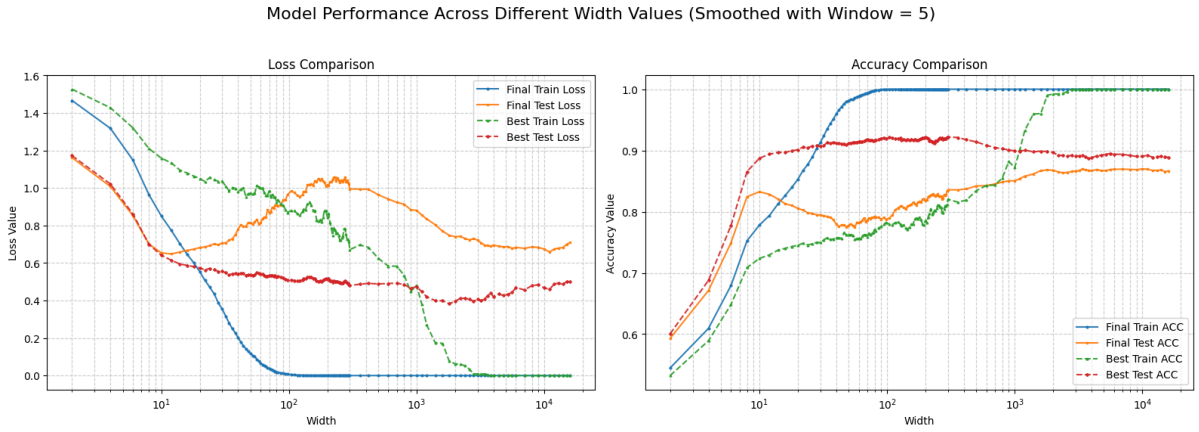
`0.999)` . Also instead of `6000` epochs, I trained each model for `2000` epochs. Furthermore, no learning rate scheduler was used and the dataset and network are the same. Finally, I changed the `MSELoss` function by `CrossEntropyLoss` . Changing the loss function effectively changes the whole task but for now my goal is to see whether double descent can happen with this dataset setting or not. As we can see by changing the optimizer and loss function, the double descent curves appear, interestingly, around the same parameter count claimed by Belkin (around width `50`):

Let’s look at the smoother versions:

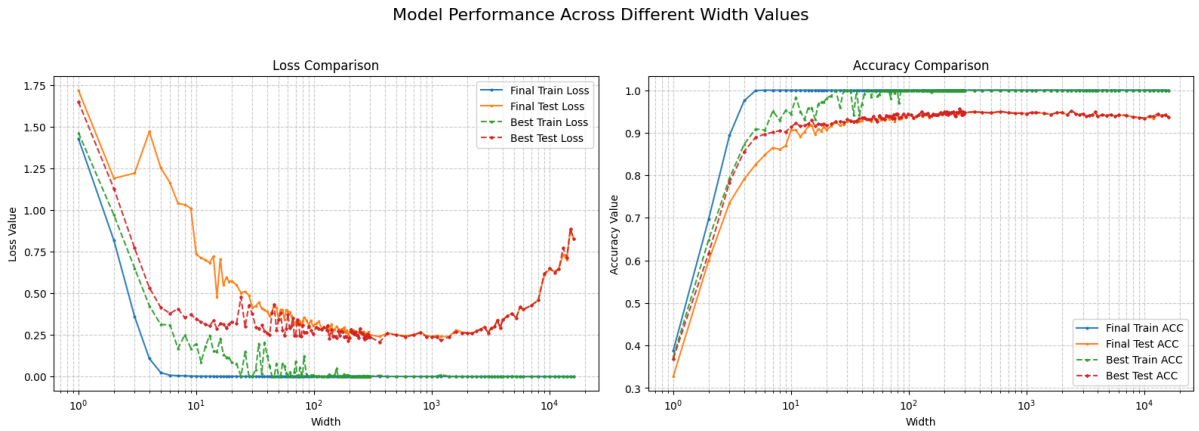


Let’s now see what happens with `20%` label noise. Adding noise shifts the interpolation threshold to a much higher model capacity:

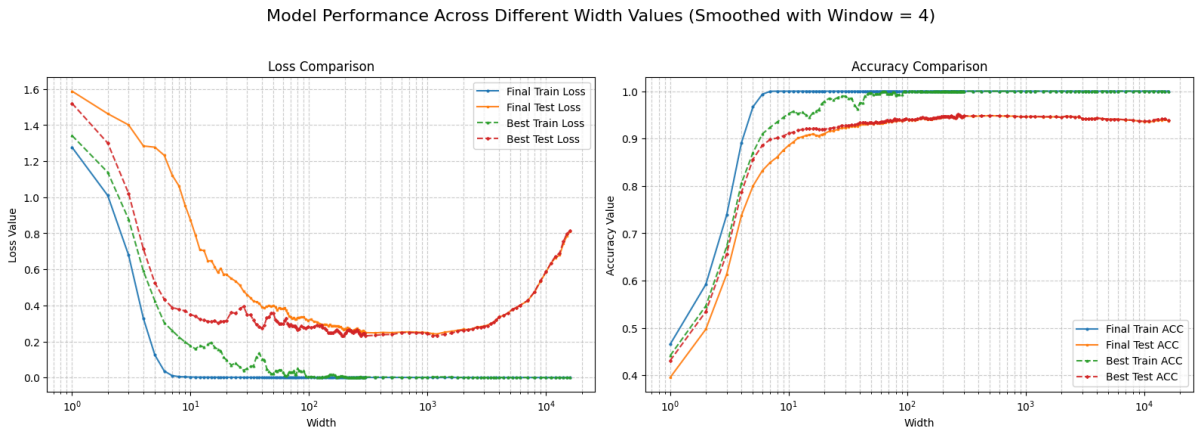
As we can see double descent is happening around `200-300` width for the hidden layer. Let’s smooth out the curves to make it more visible:



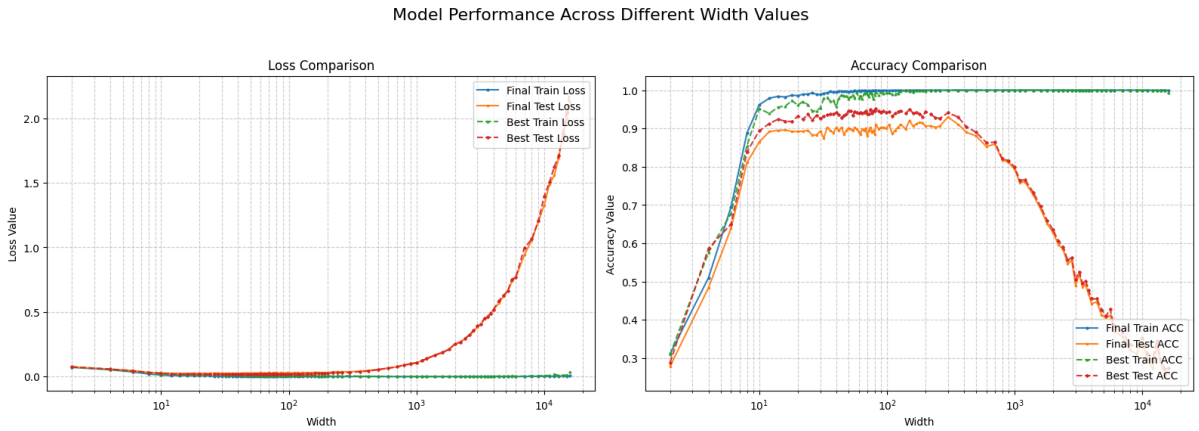
It is more clear now. So, overall, the double descent is happening in the dataset setting defined by Belkin, but their optimizer setting or the `MSELoss` function prevents the double descent curves to appear. Let’s see if we use `CrossEntropyLoss` with a similar `SGD` configuration to the Belkin setup, the double descent appears or not.



There is no double descent (the first peak around width `4` cannot be the interpolation threshold since the training loss is not zero yet at this capacity). Let's see the smoothed version:



Here it is more clear that no double descent is happening. So, the problem does not seem to be the loss function. Let's now try `Adam` with `MSELoss` so we have checked all combinations to find the exact factor that determines whether the double descent happens or not in the task defined by Belkin for the neural nets.



In this setting also double descent does not happen. It seems choosing `MSELoss` function completely eliminates the double descent phenomenon. Also one more thing worth mentioning is that when we use `MSELoss` for this classification task, as opposed to `CrossEntropyLoss` , the models which are over-parameterized, do not cross a good solution, and this is the reason we see the curves associated with best models, are getting worse along with final models.

However in the case of using `SGD` with `CrossEntropyLoss` where also double descent did not happen, I have no specific hypothesis for why this is happening. Maybe the learning rate scheduler is the problem or maybe the high initial learning rate and low gamma of the learning rate scheduler which at the last step would result in a learning rate of `1e-3` which is still high.

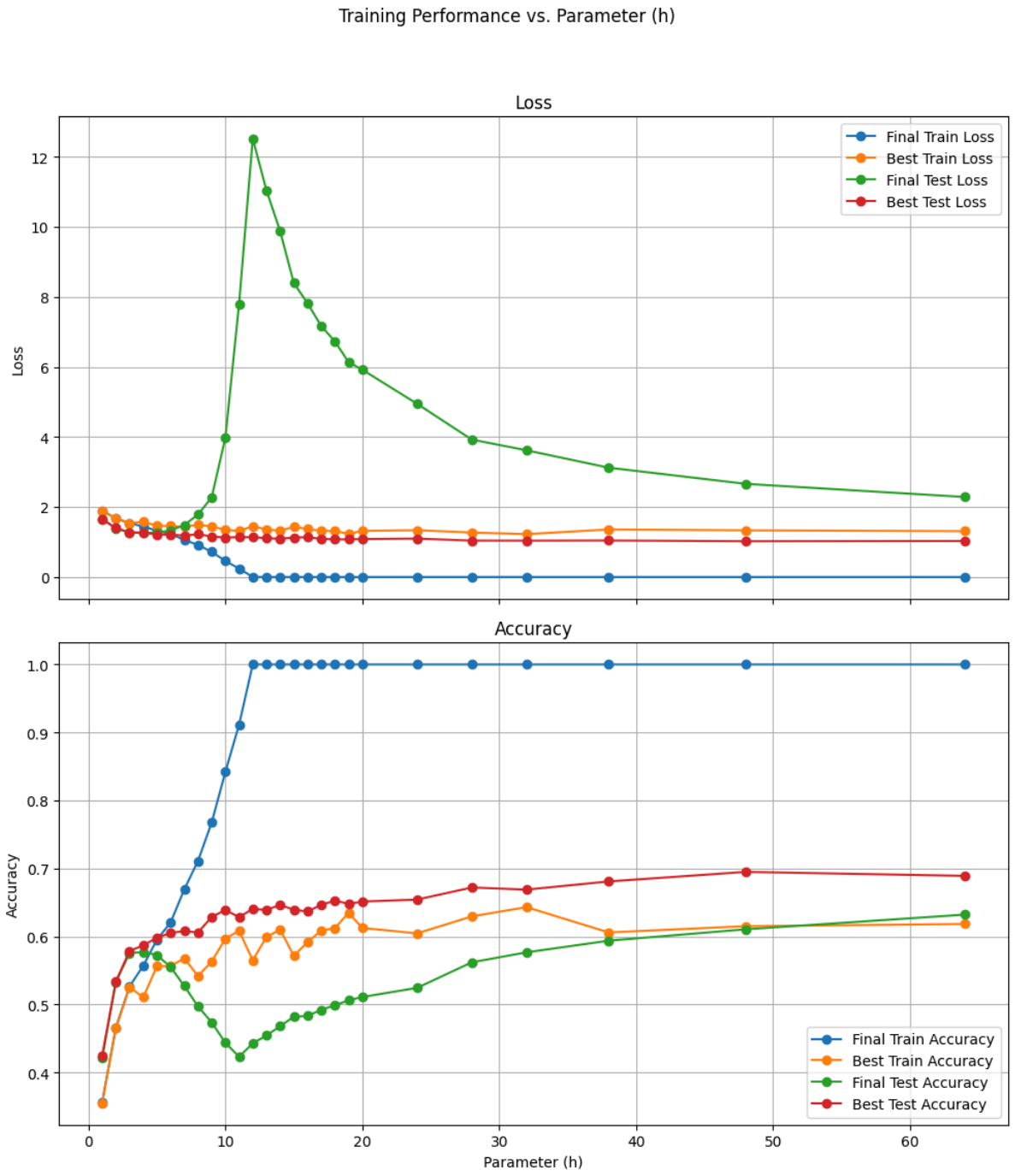
Nakkiran Experiments

Various Settings

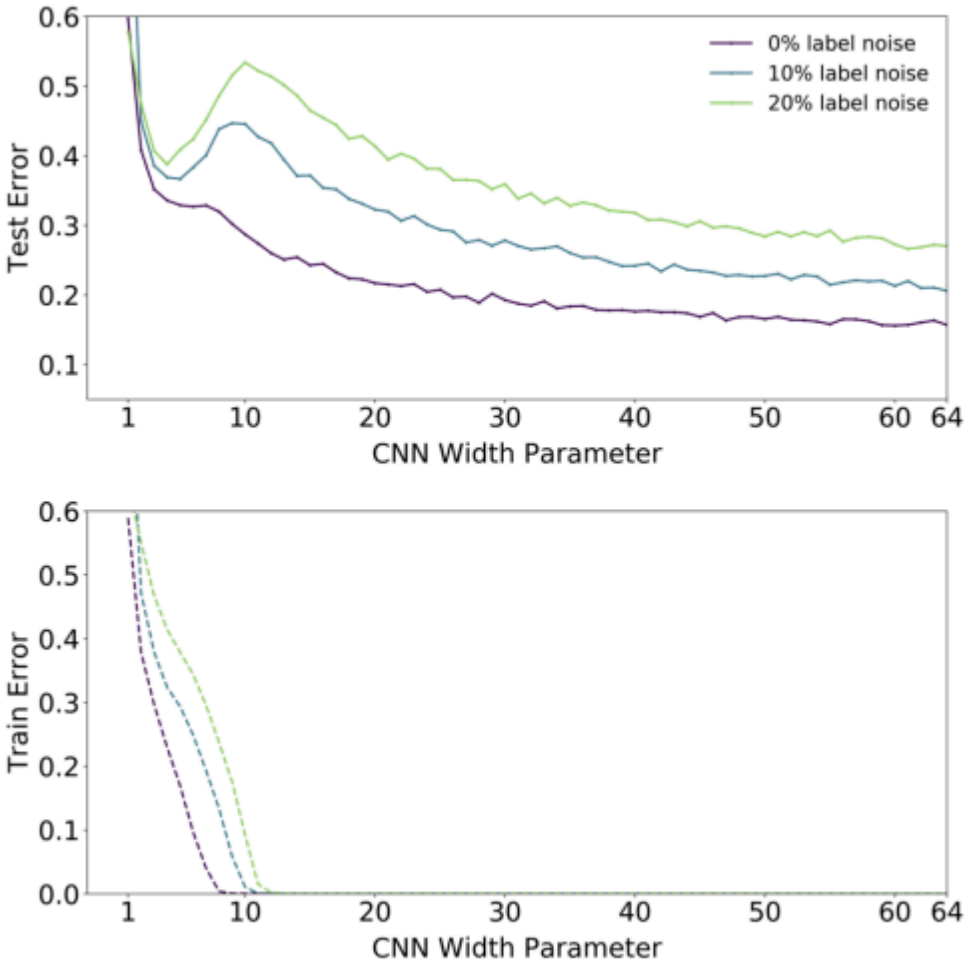
Here I show different settings I tried and the results also putting their respective experiments from the paper.

Model-Wise CNN5 CIFAR10

Five layer CNN as specified in the paper, `20%` label noise, no augmentation, optimized with SGD for `500000` gradient steps (approximately `1279` epochs with the batch size `128`), initial learning rate of `1e-2` and Inverse Square Root learning rate scheduler as described in the paper:



The same experiment form the paper (green curves):



(a) Without data augmentation.

Epoch Wise Double Descent

I could not reproduce the epoch wise double descent. The experiment I aimed to reproduce was the red curve in the first figure which is the same as the green plot in the column (a) of the second figure:

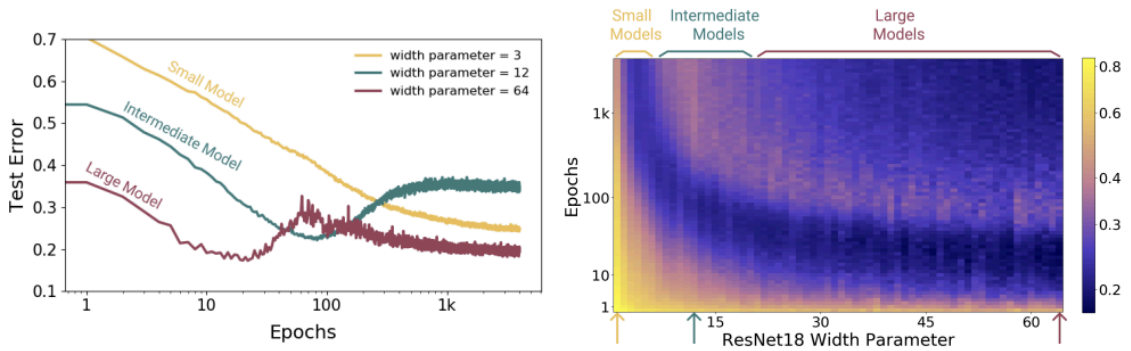
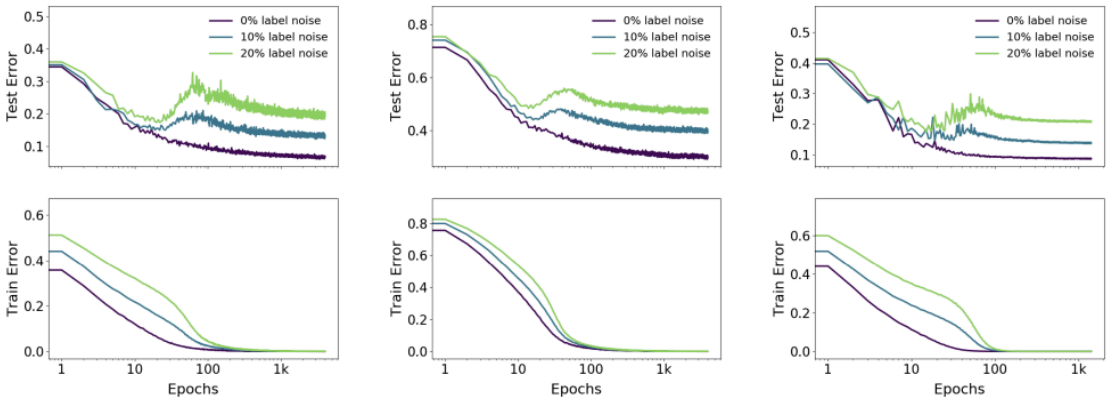


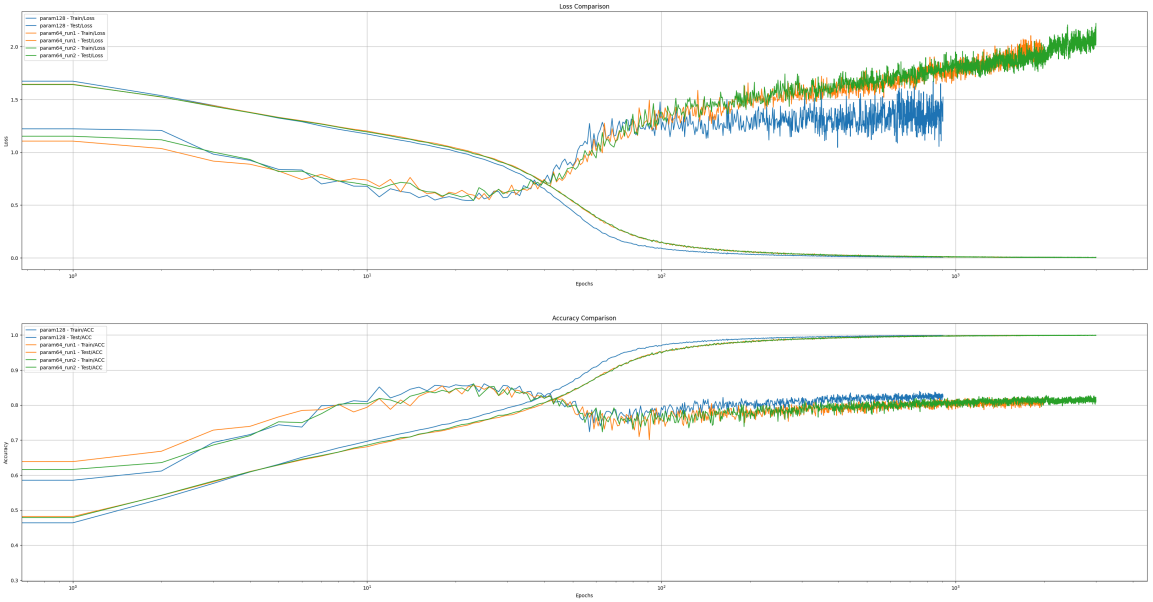
Figure 9: **Left:** Training dynamics for models in three regimes. Models are ResNet18s on CIFAR10 with 20% label noise, trained using Adam with learning rate 0.0001, and data augmentation. **Right:** Test error over (Model size $\times$ Epochs). Three slices of this plot are shown on the left.



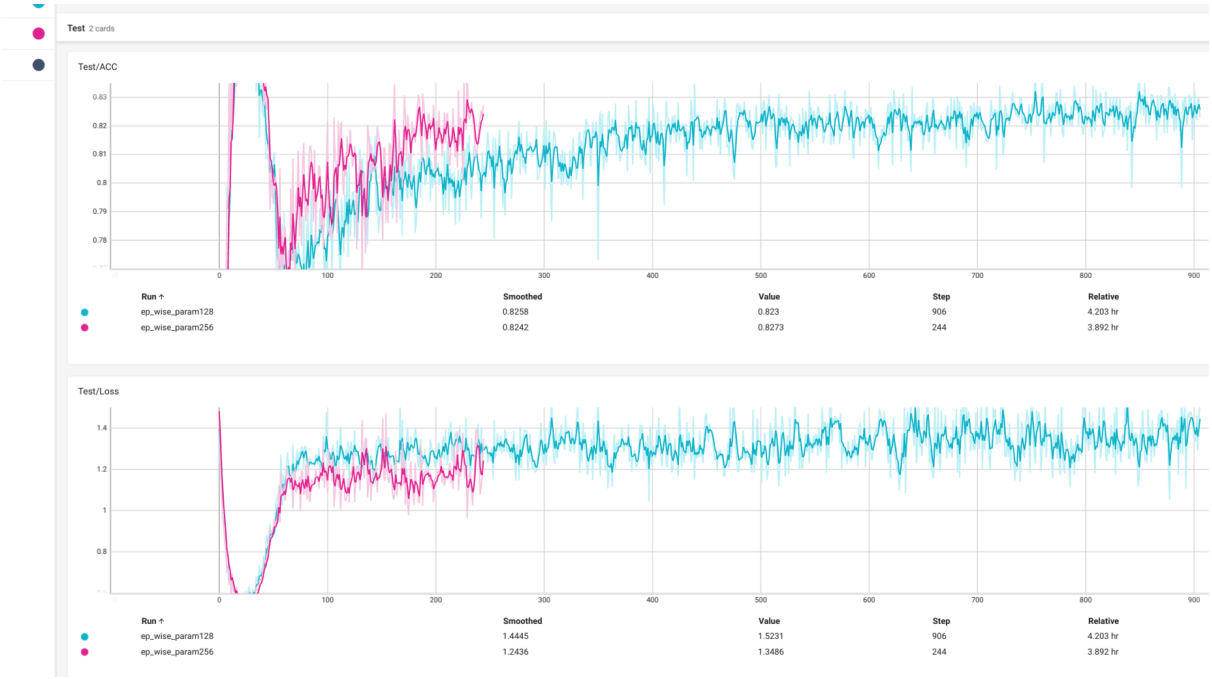
(a) ResNet18 on CIFAR10. (b) ResNet18 on CIFAR100. (c) 5-layer CNN on CIFAR 10.

Figure 10: **Epoch-wise double descent** for ResNet18 and CNN (width=128). ResNets trained using Adam with learning rate 0.0001, and CNNs trained with SGD with inverse-squareroot learning rate.

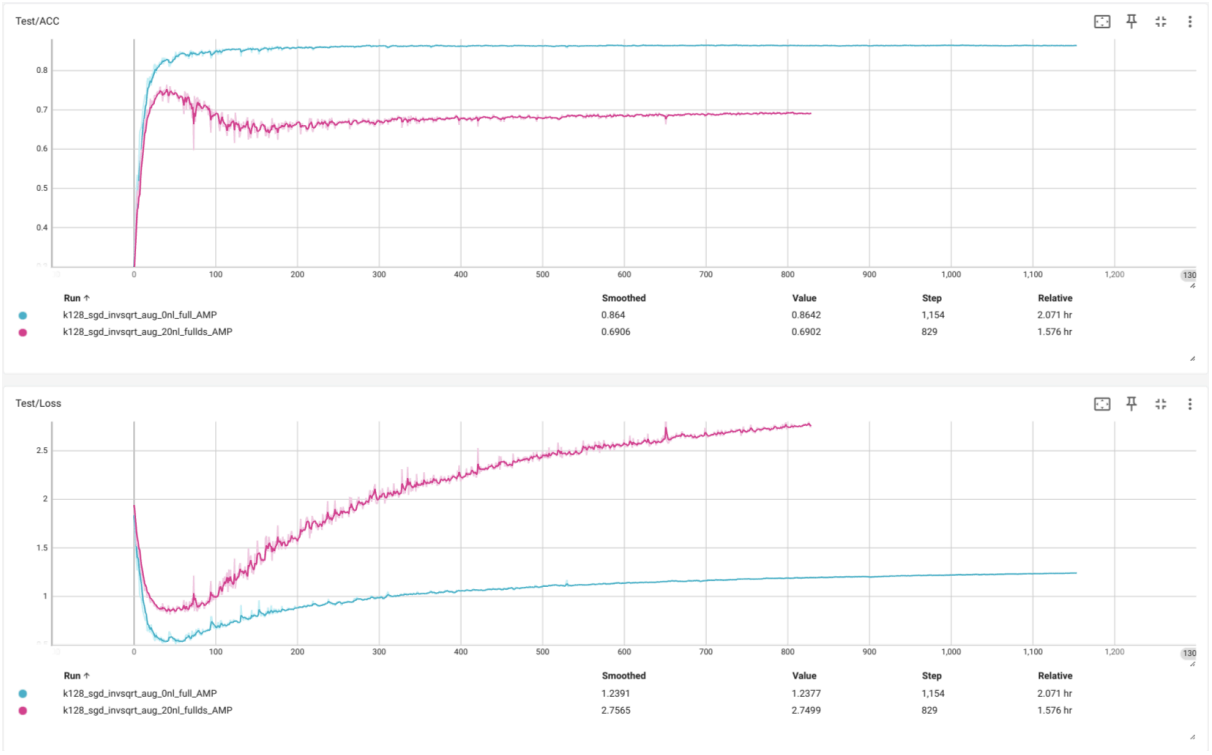
where they train **ResNet18** with `64` base number of kernels for almost `4k` epochs on **CIFAR10** with `20%` label noise and data augmentation enabled. Optimizer is `Adam` with learning rate `1e-4` . The green and orange curves in the following plot is the same experiment labeled as “Large Model” in the above plot:



It clearly does not resemble the pattern in the paper. So I thought maybe it is due to low capacity of the model and doubled (the blue curve) the number of kernels to `128` as the base number of kernels, to see if the problem was low capacity, however the second descent did not happen. I also tried the base kernel size of `256` but as it seemed the second descent is not going to happen, I canceled the training (purple curve, cyan is the same as blue in the above plot):



I also tried to reproduce the column (c) experiment of the figure from the paper above which is **CNN5** on **CIFAR10**, both with (`20%` noise purple curve) and without noise (cyan curve), I got the following curves:

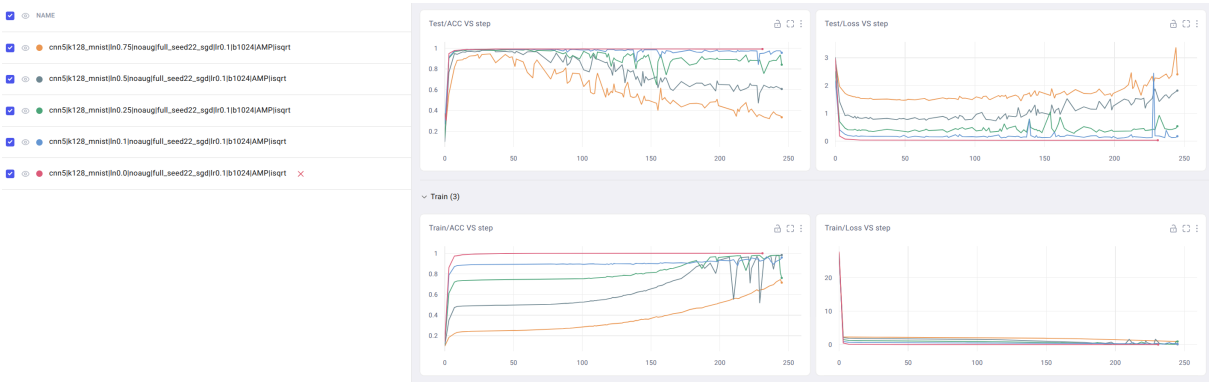


There is clearly no second descent in the test loss.

There is however a rather unusual pattern in these experiments where after the models enter the overfitting phase, meaning the test loss starts to increase, the accuracy of the model does not decrease, it either plateaus, or starts to improve again! This is more clear in the green and orange curves of the **ResNet18** model or in the purple curve of the **CNN5** trained on noisy samples. Also in the noiseless experiment with **CNN5** (cyan curve in the above figure), the test accuracy plateaus, but the test loss is increasing. We discussed this multiple times but did not came to any conclusion about what is happening. The only idea was that the model is losing confidence on easy samples and is starting to fit hard samples.

High Noise, Good Solutions

Here I tested different noise amounts on the training dataset to see it's effect on the learning curves of **CNN5** model on **MNIST** dataset (full). Despite high levels of noise (even `50` or `75` percent corrupted labels), the models cross a very good solution (more than `90%` accuracy on the test set even with `75%` noisy samples) in initial epochs which shows that for easy datasets like **MNIST** the models can result in good solutions if early stopping is applied:



The orange curve is for `75%` noise and dark grey is `50%` noise and so on.

Synthetic Dataset and Experiments on Weight Reuse and Depth

In the following experiments I used a synthetic dataset, which consists of a mixture of multi-modal Gaussians (each class might have multiple clusters), to have more control on the dataset and the exact amount of the noise in training samples (as we discussed, even datasets like **MNIST** and **CIFAR** have some amount of noise in their samples) to get a firm grasp of the role of noise in the double descent phenomenon. Input dimension (number of features of each Gaussian) is set to be `512` . Output dimension is `30` (number of classes). `100000` samples are drawn from the distribution and split to train-test set with `(70-30)%` ratio. The models are optimized using `Adam` with learning rate `1e-4` and default betas of `(0.9, 0.999)` and no learning rate schedule. All of the models are trained with `CrossEntropyLoss` function. For all of the experiments a fully connected network with one hidden layer is used except for the last part where we aim at analyzing the effect of the depth in Double Descent.

Random Initialization

No Noisy Samples

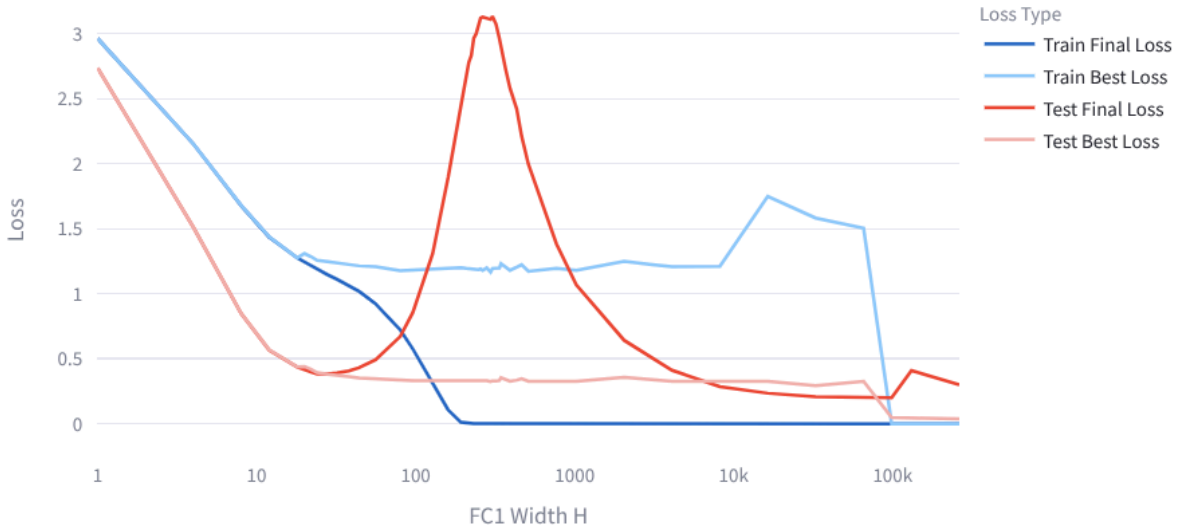
If there is no noise in the training samples, the double descent phenomenon does not appear:

`TODO put the figures here`

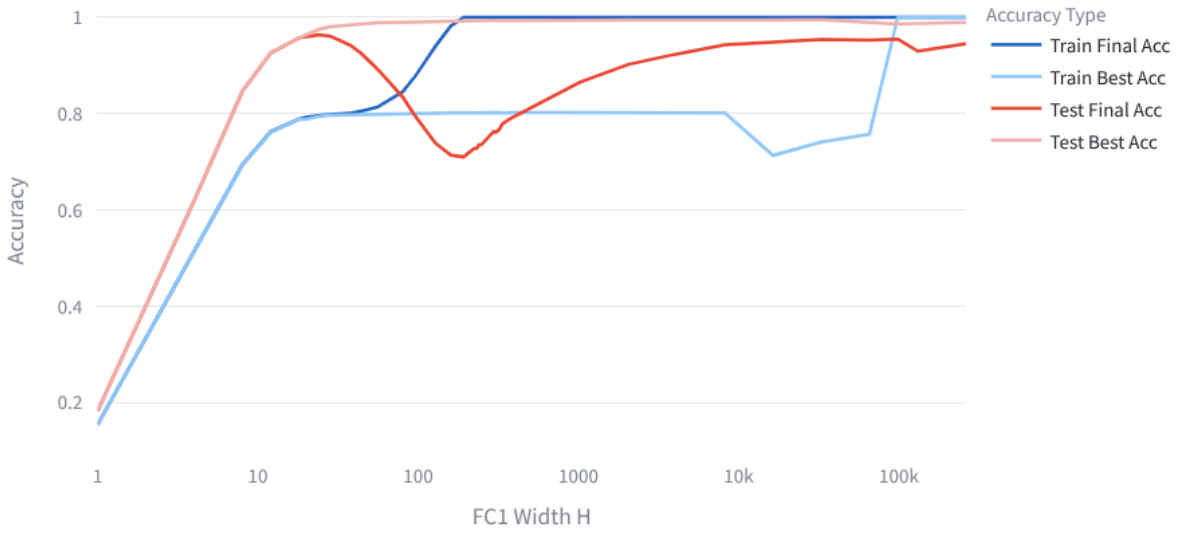
20% Noise

I applied `20%` noise to the training set and this is the resulting curves:

Loss vs. FC1 Width H



Accuracy vs. FC1 Width H



The x-axis is log scale. Interpolation threshold happens around `300` width. Until around `10k` width, the curves are smooth and as expected.

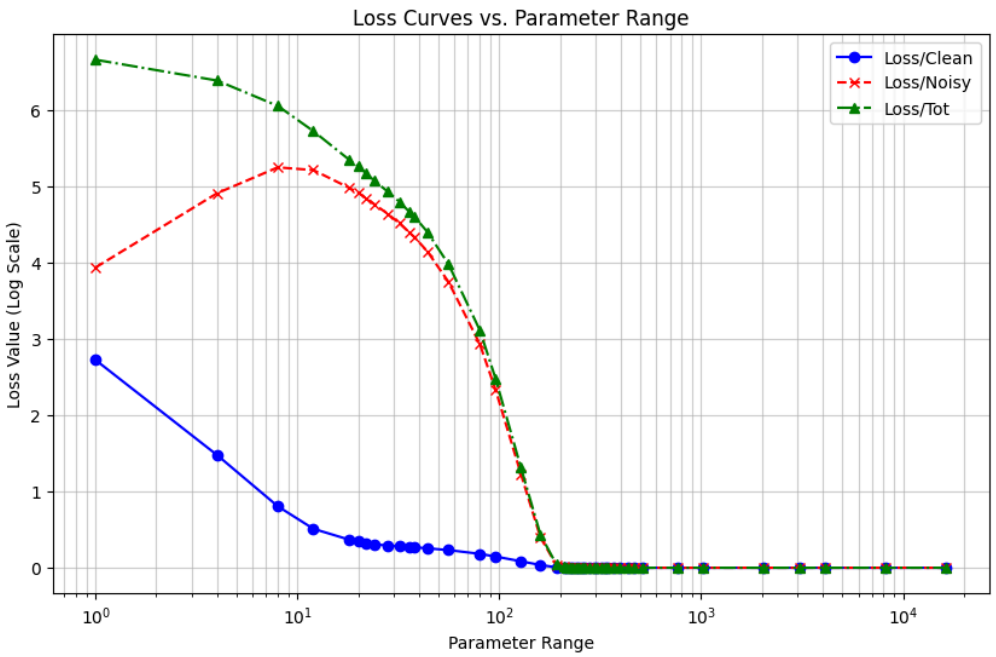
Observations:

- Up to the point of `65k` width, all the models cross the best solution during their training, and this best solution seems to be so close to the best

solution the previous smaller models achieved. This is evident by looking at the pink curves which depict the performance of the best model for each capacity during the training of that model. In other words, pink curves are the results if we used early stopping (in case of no epoch wise double descent which does not happen in this setting). It seems that most of the models despite having different capacities, and thus different loss landscapes, learn similar classifiers in the initial phases of training.

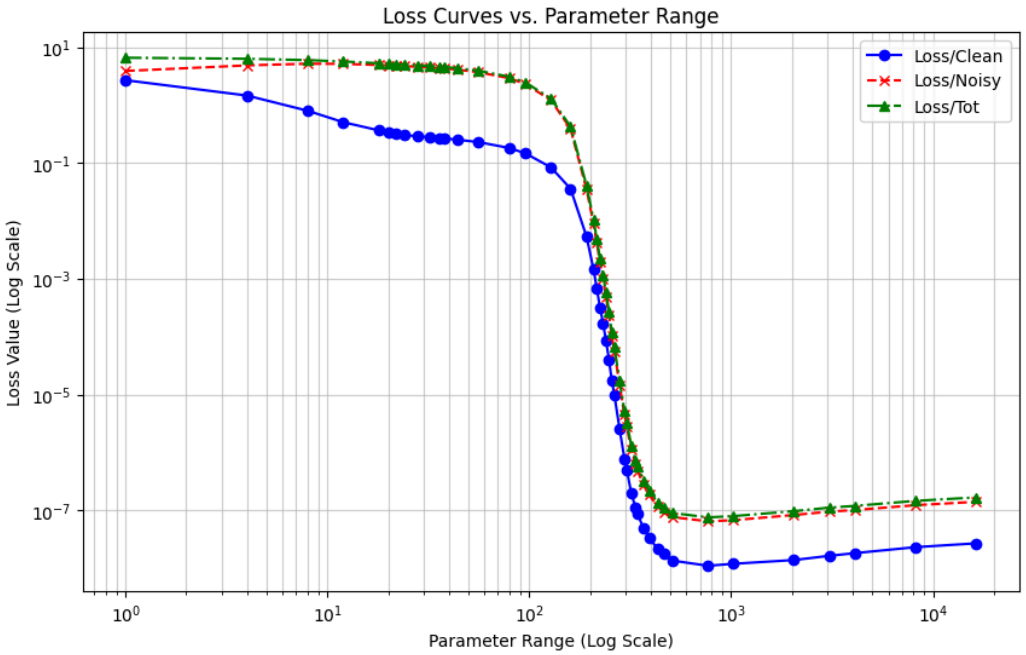
- One more thing to point here, the best solutions up to the `65k` point, are not the interpolating solutions, meaning, the best performance on the test set is not achieved by the models that get zero training error. However, after `65k`, best models become the interpolating solutions, meaning the models that perform the best on the test set, have zero training error. One more interesting thing here is that, despite a sudden decrease in the test loss achieved by the best model, the accuracy decreases. But one thing I have to mention is that I did not repeat this experiment with another seed to see if this pattern (after `65k` width) is persistent.

Also for this experiment, I decoupled the amount of loss for noisy and clean samples. The following figure shows the corresponding curves:



The x-axis is log scale but the y-axis is not, despite the plot saying it is log scale!

In the initial phase and first descent, the model fits the clean samples and the loss for the noisy samples increases. Then, as we go from the first descent towards the interpolation threshold (around `300` width), the model starts fitting noisy samples along with lowering its loss on clean samples as well. To see what happens after interpolation threshold, let's make the y-axis log scale:



as we can see, the loss of the model on the noisy samples is always higher and after interpolation threshold it is the loss on the noisy samples that contributes the most to the overall loss. However, the ratio of the loss between noisy and clean samples seems to remain constant after interpolation threshold.

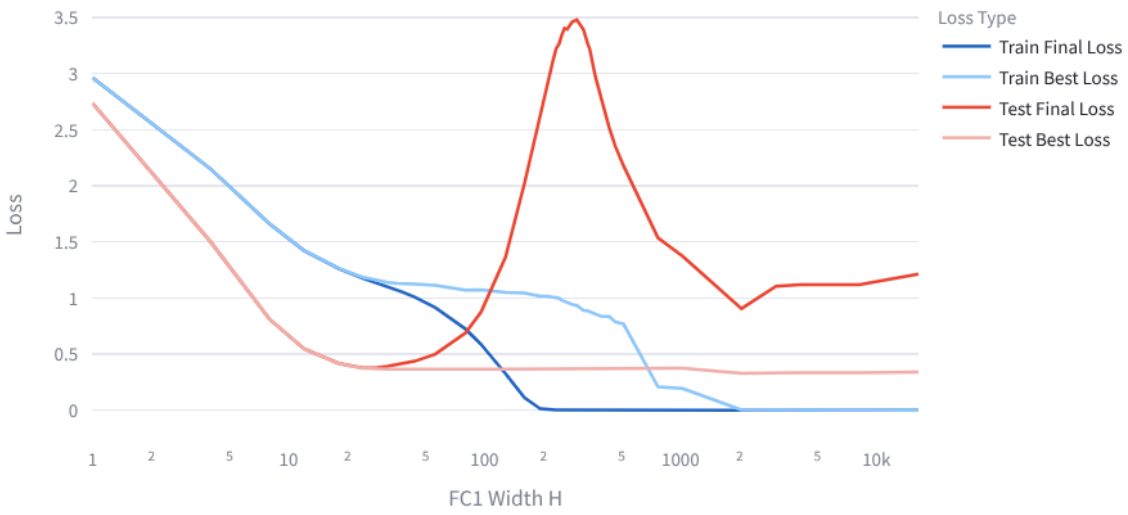
Initialization From the Previous Best Model

In the following experiments, we train the models sequentially. At each capacity, we take the best model from the previous capacity, and load its weights to the current model, and initialize the rest of parameters randomly. This is somehow like starting from a checkpoint, with some randomness which comes from the newly added parameters to increase the model capacity. For example, if we have a model `A`, with width `8` and a model `B` with width `10`, we first train `A` and during training save the best performing parameters, and then when we want to start training `B`, we first load `8` of its parameters with the weights saved from `A`, and then randomly initialize the remaining `2` parameters.

No Freezing

Here we keep the setting as it is. The loaded weights can be updated. No restriction on the gradient updates. Here is the resulting curves:

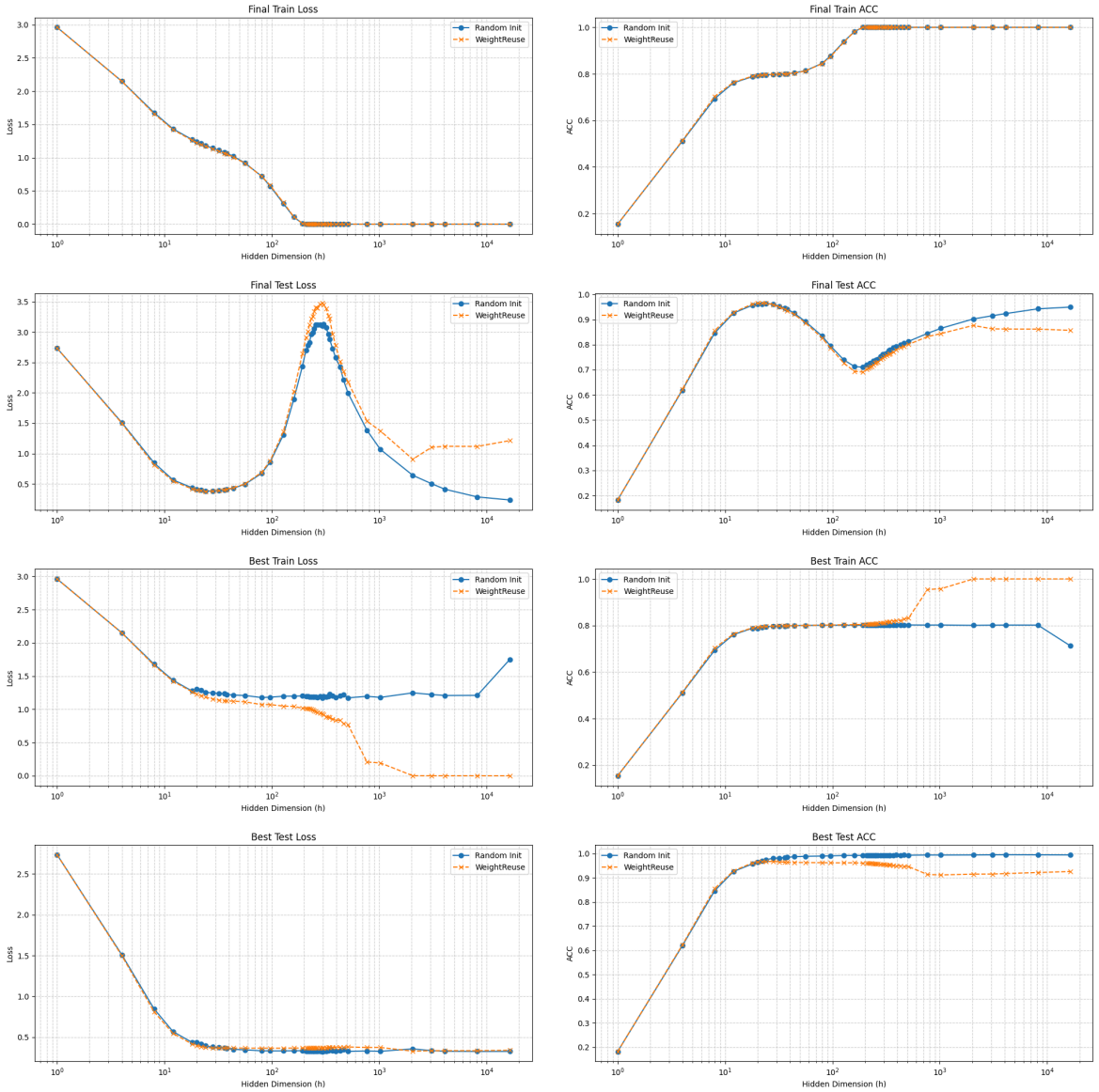
Loss vs. FC1 Width H



Accuracy vs. FC1 Width H

The interpolation threshold is happening at the same capacity as the case of random initialization. The changes are that, first, we are getting higher losses at interpolation threshold, and also the phenomenon that the best models become interpolating models (with near zero training loss) happens much much earlier in this case. Let's put both cases (random initialization and weight reuse) on the

same plots, to see the differences more clearly:

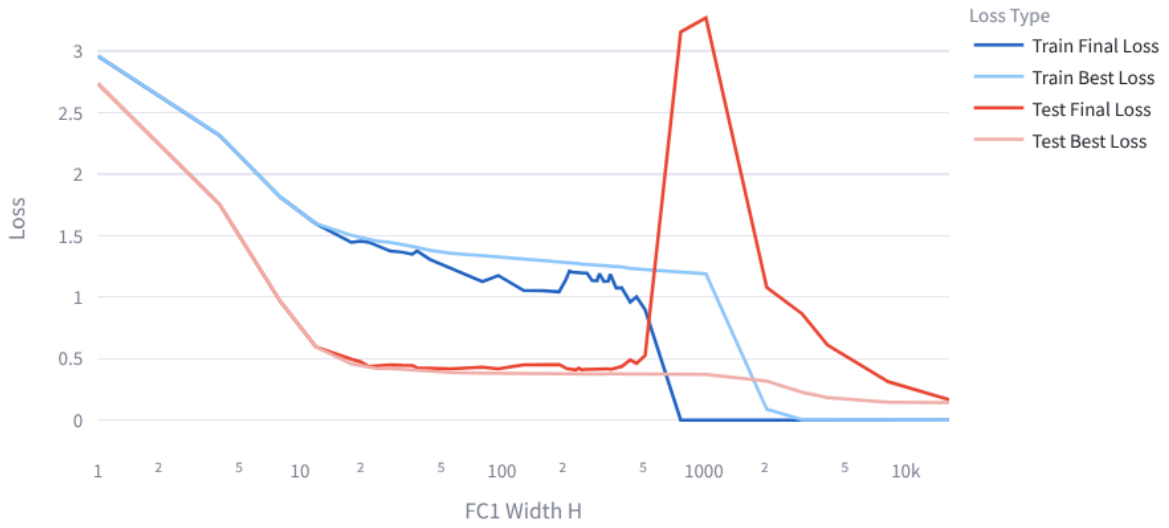


As we can see, the difference is, after some capacity (around `10e3`) the models with random initialization continue to improve their generalization, but the models which used previous best model weights, start to perform a bit worse on the test set, and the best solutions found during optimization of these models (models with capacity higher that `10e3` and with weight reuse), become interpolating solutions (zero loss on training samples) as opposed to randomly initialized models where the best models keep performing the same (the models which we normally get with early stopping).

Freezing the Reused Weights

In the following experiments, I follow the same path from the previous experiments where I reused the weights from the previous best model, but now, we also freeze these weights so the gradient updated do not affect them. This means that the optimizer can only update the randomly initialized weights. One thing to keep in mind here is that the gradient of loss is calculated with respect to all parameters, however, the update is only applied to the randomly initialized parameters or free parameters. This means, the updates are not towards the desired direction of the optimizer since all of the calculated gradients will not be used for updates. This is because we are freezing the single parameters in a single layer not like freezing layers that is common in fine tuning. Since PyTorch does not have a mechanism for freezing a portion of single layer, I manually block the updates using gradient hooks. So as opposed to fine tuning where the gradients are not calculated for some part of the network, here, the gradients are calculated but not applied. This might make the whole setting flawed, but let's just see what happens. The following plot shows what happens when we do this:

Loss vs. FC1 Width H



Accuracy vs. FC1 Width H

Let’s first specify the widths used for the models here so we have a more clear picture of what is happening:

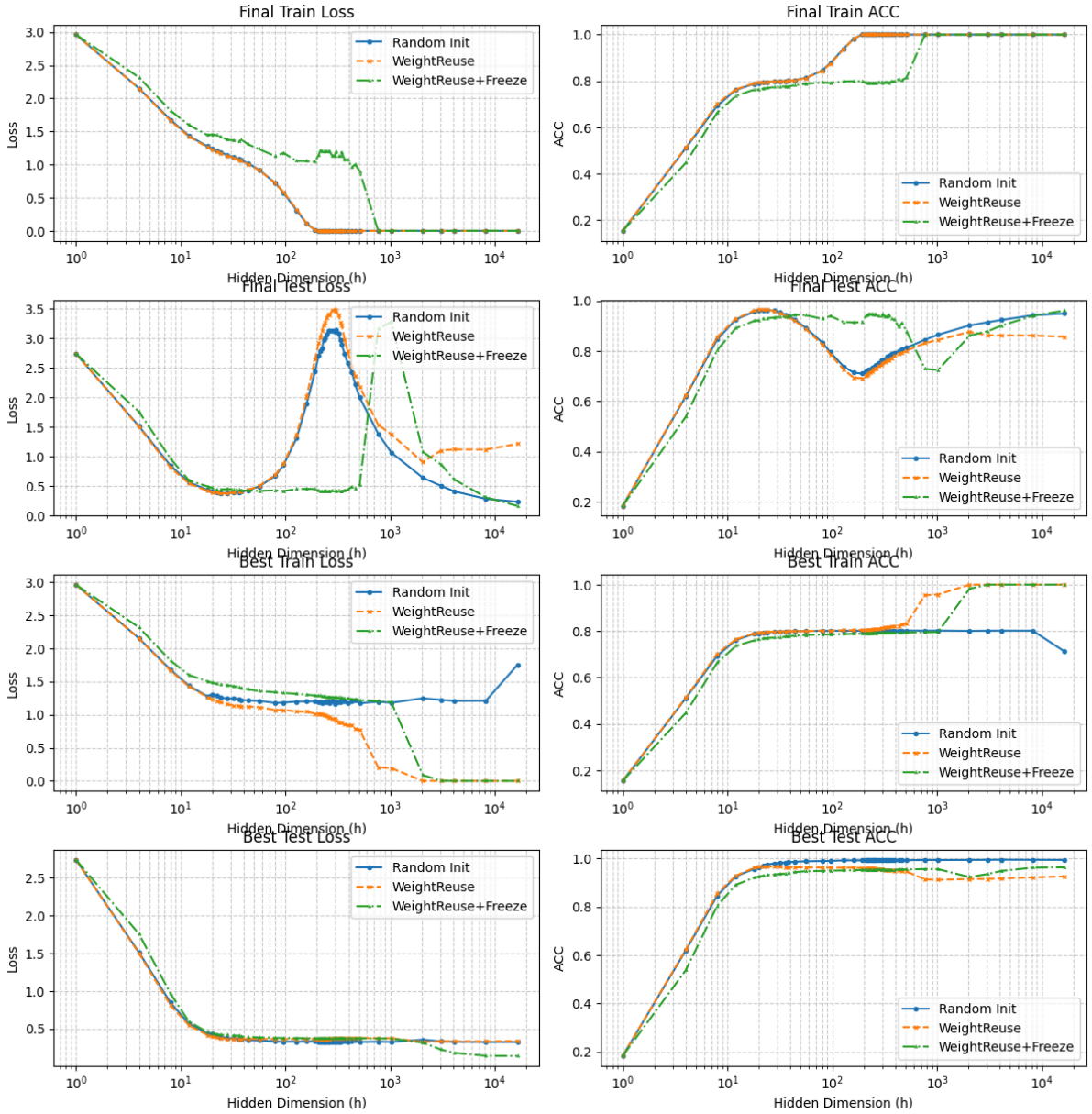
```
[1, 4, 8, 12, 18, 20, 22, 24, 28, 32, 36, 38, 44, 56, 80, 96, 128, 160, 192, 208,
216, 224, 232, 240, 248, 256, 264, 280, 296, 304, 320, 336, 344, 368, 392,
```

```
432, 464, 512, 768, 1024, 2048, 3072, 4096, 8192, 16384]
```

Observations:

- As we can see, the interpolation threshold has shifted to the right by a large amount however the first descent happens at the same capacity as before. The reason seems to be the gap between the capacity of sequential models, or in other words, the number of free parameters a model has. Starting from the first descent (around `20` width), as we can see, initially, where the number of free parameters compared to the frozen ones is low, the best and final models are approximately the same. This means, the model can not go out of the current local minimum when the amount of free parameters it has is low compared to the frozen ones. So it either uses those free parameters to improve the current frozen solution, or completely neutralize free parameters so that they don't affect the frozen solution.
- As soon as the optimizer gets enough free parameters to get out of the current local minimum (which defines the frozen function), it starts to use those free parameters to overfit the noise in the training samples. This happens exactly at the jump from `512` width to `768` . As we saw previously, from the first descent to the interpolation threshold, the model needed approximately `250-280` width increase. Here, we are giving the optimizer approximately the same amount of free capacity, and the sudden jump in the test loss clearly shows that the model directly jumps from the first descent performance, to the interpolation threshold performance.
- So, it seems that, in this setting, the optimizer treats the frozen function as a constant, and no matter the capacity of the frozen model, as soon as the optimizer is given enough capacity to fit the noisy and clean samples together, it goes for it!

Also let's see the three scenarios on the same plot:



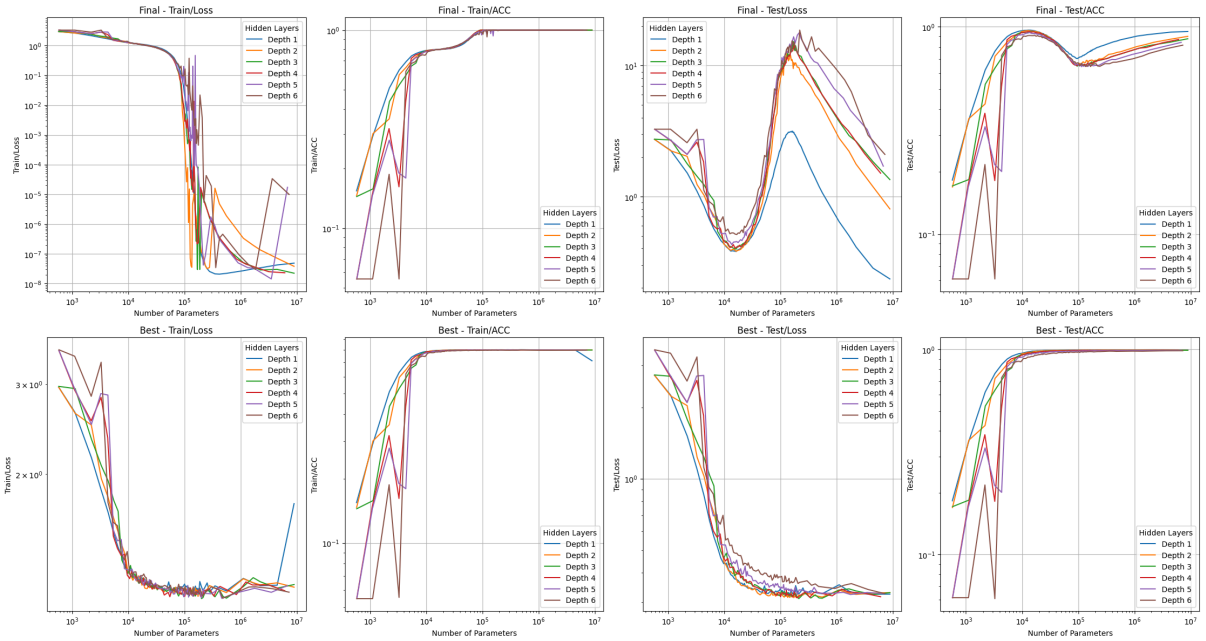
Another Direction! Checking the Effect of Depth on EMC

What happens if we keep the number of parameters the same, but make networks deeper? Does this affect Effective Model Capacity? It seems that as we increase the depth of the models, the ability of the model to learn more complex patterns increases. This means, we expect if we consider a constant

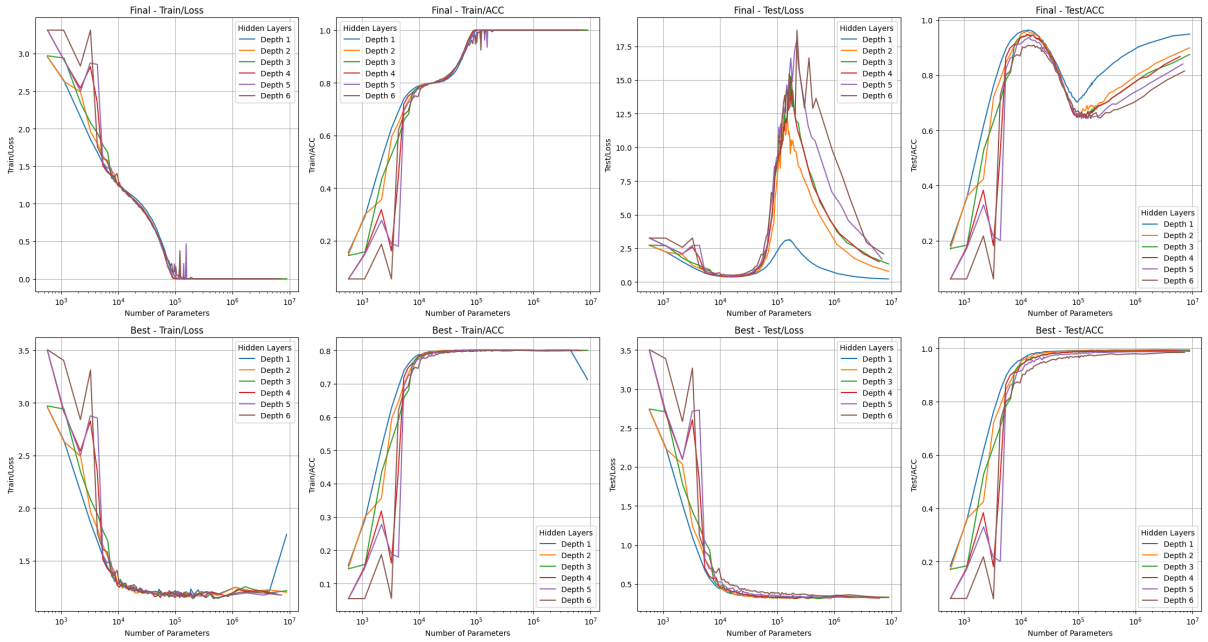
number of parameters let's say P , and then form two networks architectures A and B with P parameters, where network A is wider and network B is deeper, then network B should have higher EMC meaning it should be able to interpolate training data sooner. In the context of double descent phenomenon, this is equivalent to shifting interpolation threshold to the left. Let's see if this happens in practice.

Synthetic Dataset, Fully Connected Network

We start from the setting we used for training models on the synthetic dataset, namely, a fully connected network with one hidden layer of varying width. We put this network as our base, then build deeper networks such that their number of parameters are the same with this base network, with the possibility of a small residual of 30 parameters. We also put a bias in the architecture search so that we choose network configurations where the difference between layers' widths is as small as possible. Here we test 6 different widths. From 1 (which we have already seen) to 6 . The results are shown in the following figures:



In the above plots, x and y axes are both log scale. The following figure is the same as above, but with y axis no longer being log scale:



Observations:

- First, the high oscillation of the values in the lower capacities of deeper networks is due the high input dimension (`512`) and the problem of finding network configurations that can have a good balanced architecture in these lower capacities. For example, for these capacities, we might have a network of depth `3` with hidden layer widths of `(2,2,2)` where the overall dimensions from input to output would be `(512, 2, 2, 2, 30)` which is highly imbalanced.
- It seems that the capacity where the first descent is happening does not change with the depth. This seems counter-intuitive, since deeper networks clearly have the same EMC as the wider ones until this point.
- After the first descent, we can see that the interpolation threshold seems to be around the same capacity for all of the models. This is also counter-intuitive. Deeper networks get higher test loss at the interpolation threshold which shows that the deeper the network is, the higher the tendency to fit the noise and forget the simpler patterns it has learnt.

- All these plots seems counter-intuitive to me, if these patterns are consistent, then for fully connected networks, what is the point of using deeper architectures when wider networks are more robust to fitting noise? And also wider networks can get the same or better generalization?!