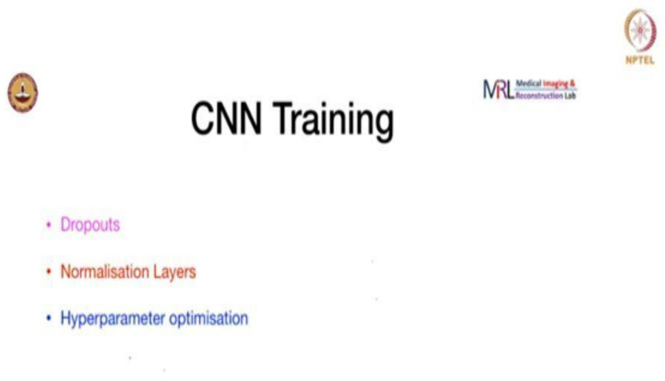
Medical Image Analysis
Professor. Ganapathy Krishnamurthy
Department of Engineering Design
Indian Institute of Technology, Madras
Lecture 46

CNN Training

(Refer Slide Time: 00:14)



CNN Training

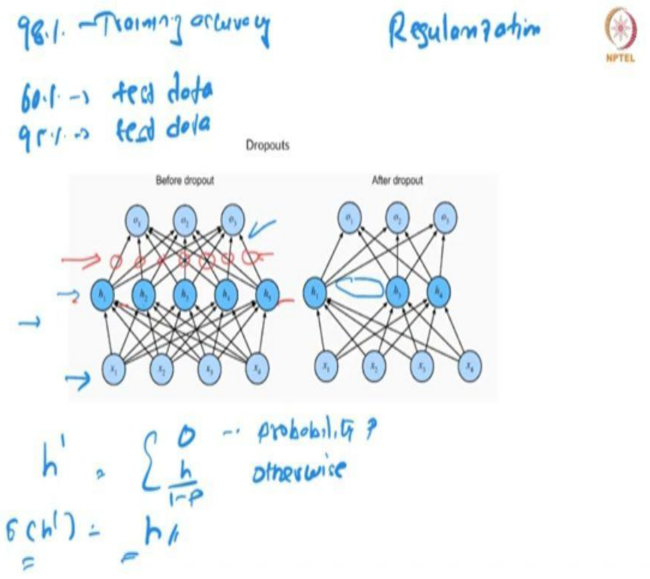
- Dropouts
- Normalisation Layers
- Hyperparameter optimisation

Hello and welcome back. So, this week we will continue with where we left off last time. So, where, we looked at convolutional neural networks and the various operations in it; primarily, the convolutional operation, using filters kernels as well as the max pooling operation. So, this week we will look at CNN training to some extent also to look at semantic segmentation.

So, as far as training is concerned, the algorithm for training is the same; in the sense learning is the same, the gradient descent. And the back propagation algorithm is used to estimate the gradient of the weights in the different layers, in the filter kernels in the different layers.

However, we are going to look at certain techniques which are adopted primarily for CNNs in many cases; one is called dropout; this is used for regularization. Some normalization layers, batch norm is one such technique, we will look at what that means. And finally, a small note on hyperparameter optimization to wrap up some aspects of CNN training.

(Refer Slide Time: 01:24)



$$h' = \begin{cases} 0 & \text{probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

$$E(h') = h$$

So, we will look at dropouts first. So Dropout is regularization technique, so regularization technique; so the idea is the following. So, we want to reduce the gap between your training and test error. So typically, after you finish training your fairly a large neural network, especially CNN; let us say for medical image analysis, the training error might be very small. So, for instance, if you are doing classification, for instance.

So, let us assume that you are trying to classify chest X-rays into two classes abnormal versus normal being any kind of abnormality; and you have 92 percent accuracy. Now, we want to look at sensitivity etcetera; but in general, you say they are all pretty good, and we have 92 percent accuracy; and this is your training, accuracy, training accuracy.

And if you take a dead data test data, new data comes in this data; maybe it is not is the, performance is not so good. Most of the time, it is not so good. But, but you would not want it too far away. Let us say you do not want your network to perform 60 percent accuracy on test data. So, you would rather have it perform at 95 percent or even 90 percent on test data; that is what you want.

So this, so which means, that we are trying to narrow the gap between the training performance and the test performance; and that for that purpose, these dropouts are used, and they act as regularizers because, if you think of regularization, what it does is to make sure that your parameters are do not memorize; so do not memorize the data.

So, what happens is because CNNs have a lot of parameters, especially in the last few layers, when especially for classification, and a lot of the layers are fully connected. There is a tendency tendency to overfit; overfitting is when your network memorizes the data. So, it will fit train data very well; but when the test data comes in, you have a very poor performance.

So, reduce that gap in performance, you do this regularization, and dropout is one form of regularization. So, what exactly happens in a dropout? So, in a standard dropout regularization, what we do is we zero out some fractions of the hidden units in every layer. And then we kind of compensate for the de-bias the layer by normalizing some fraction of the hidden units; that were not dropped out that were retained.

So, which means that we can write it in this form; you read in unit that you retain, h' is 0 with probability p; and if you and so $\frac{h}{1-p}$, otherwise. So, if you calculate the expectation of h' , it will turn out to be h. So, in expectation that you get the hidden units that you would like. So, the how we illustrated this in the picture here, where you have on the left hand side in this picture, the network is bottom to top.

The input layer is the x, the hidden units are h, and the output layer is o. And so in this layer, in this particular example, before dropouts, all the neurons are connected to all the other neurons in the sense between layers. But, if you any apply dropout, what you do is you zero out some of the neurons here; neurons h_2 and h_3 have been zeroed out here, that is why you do not see them there.

And then you train the network as it is. So in every epoch, you would drop out some of the neurons with a certain probability. And once and then you compensate for the de-bias that by scaling the, the existing the retain neurons or the hidden units with 1-p. So, how does this help? So, in the original paper which talked about dropouts, the explanation was offered was that, by training, it can happen that certain sets of neurons can get, can adopt co-adapt. Which means that they, they not only they just do not learn independently, but rather depend on what the what their input is.

So, if there are multiple hidden layers, let us say I had one more hidden layer somewhere there. The argument is that, this the neurons in this or hidden units in this layer will not learn the features independently; but, rather it will learn based on what its inputs are, which is basically coming from each h_1; so these two layers might co-adapt.

So, to make them independent, what you do is you cut out some of them; in this case, your cut out h_2 and h_5. I am sorry, I just said h_1 h_2 here the previously, cut out h_2 and h_5. So that way, the next layer does not will learn to impute; in a sense, it learns to, it learns to learn without the presence of these two hidden units; so it trains the neural network this way. You do not overfit, we have this regularization effect. And in and your training and test accuracy specifically improves.

So, the way to implement this is to for every hidden unit, in every layer, you would sample from the uniform distribution. Retain all the neurons or hidden units, which with they are associated probability greater than p. And while throwing away all the neurons are the hidden units, which have associated probability less than p.

Now, remember that this is done only during training of the network. So, whenever during training, every epoch you would go to every layer; and you would drop certain hidden units or neurons, as they call them with a certain probability; and you keep doing that till the training completes. Now, once the training is done, you would just keep the network as it is and then do your testing.

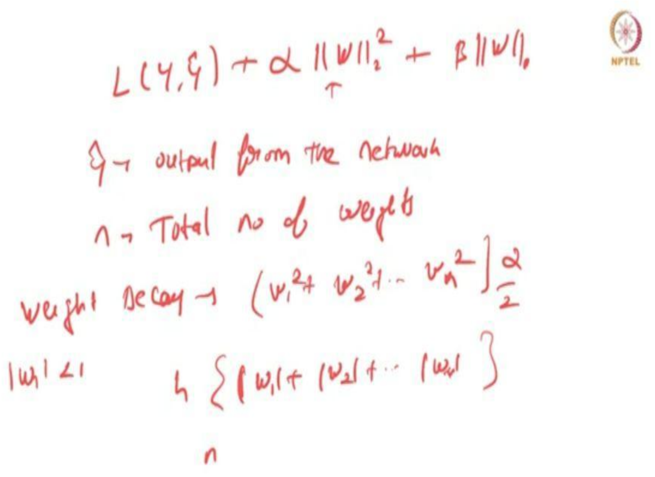
However, now some people have used it; some researcher used it to classify, not classify; sorry, misspoke, to estimate the uncertainty in the prediction. So, one way to do that it would be to drop out how multiple forward passes for prediction; and each forward pass you would drop out certain hidden units just to see, I know the variability in your output. So, another way of looking at it is that, since the neural networks you can think of them as a function, all you are trying to do is to figure out a smooth function.

So, that small changes in your input does not radically affect the output. So that is the that is another way of looking at this regularization effect offered by the dropouts algorithm. So, you can use this typically in the fully connected layers of a network, which happens in a lot of classification networks.

We looked at the ones that we had, we get it we are reviewed last week, last couple of layers are fully connected; and that is where a lot of the weights occur. So, a lot of the dropouts is actually performed only in those layers, mostly in a lot of the implementations; because nothing is stopping you from doing it in the earlier layers, in the convolutional layers also. So, when you do the dropouts in the fully connected layers, you can do several forward passes given a test data, just to quantify the uncertainty in the predictions.

That is one trick that people have also used; so that is dropouts. So, when I say regularization, that rapport is one technique. The other techniques for regularization, which is most standard, referred to as weight decay; so here, this is how you would do it.

(Refer Slide Time: 09:47)



$$L(y,\hat{y}) + \alpha \underset{\uparrow}{||w||_2^2} + \beta ||w||_1$$

$\hat{y} \rightarrow$ output from the network

$n \rightarrow$ Total no of weights

weight Decay $\rightarrow$ $(w_1^2 + w_2^2 + .. w_n^2]\frac{\alpha}{2}$

$|w_1| < 1$ $\quad \hookrightarrow \{|w_1| + |w_2| + .. |w_n|\}$

n

Let us let the $L(y,\hat{y})$ be the loss function; $\hat{y}$ is the output from the network. This estimate made by a network, y is the ground truth. Now, you would add to the loss function, several terms in a weighted appropriately; so there is this hyper parameter α, which would wait; we call this and we will also have this β. I will do it this way w_1 here. $L(y,\hat{y}) + \alpha||w||_2^2 + \beta||w||_1^2$

So, this one it is the this quite; this is the l2 norm supposedly. So, if we add n weights in your neural network, n is the total number of weights in your network; the standard weight decay that is what it is called in the neural network parlance, would be just $[w_1^2 + w_2^2 + ... + w_n^2]\frac{\alpha}{2}$ and so on and so forth, ambient square.

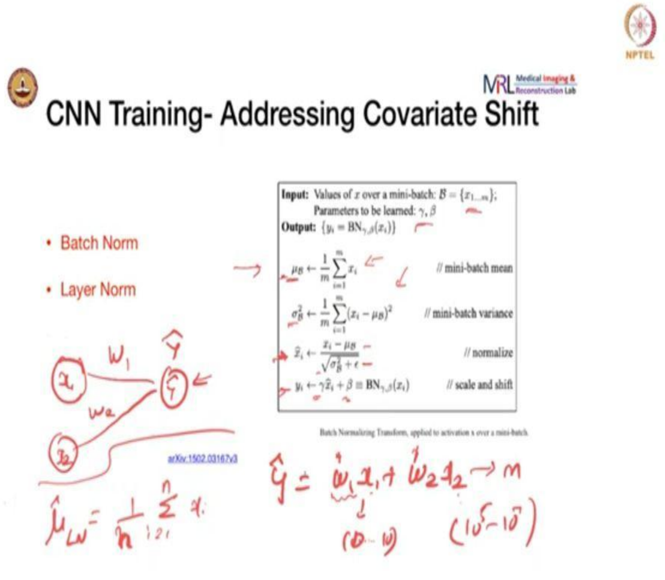
And of course, you might have to multiply this with the hyper parameter α. You have to choose α by hand, you have to do that validation study. The other version is you add the absolute values,

$w_1 + w_2 + \ldots\ldots\ldots + w_n$; n is the total number of weights. So, I am using this as n; n is total number of weights, you will turn that. So, these are the two standard regularizations. So, what this prevents so the if you look at the standard weight decay, where we add the square of the weights. This will prevent the weights from taking very large values.

So, some outsize value for any of the w's will be severely penalized. So, on an average all of these weights will be very small. So, in the sense that they will be driven to smaller and smaller values that is closer to 0; that is why it is called weight decay. All your values are w, because we the value of the weights of all less than 1. Let us say all the weights are your magnitude is less than 1; and w_n^2 becomes even smaller, or w_2^2 or w^2 becomes even smaller. So in that sense, you are driving your weights to smaller values; that is why it is called weight decay and it distributes the weights across a very large number of features, making the network more robust.

On the other hand, the l1 based there is basically the absolute the absolute values of the weights. So, this is one other regularization technique; here, this leads to a lot of the weights going to 0. Why while retaining very few of the weights with a very high weightage. And so this is sometimes not preferred when you want to re-estimate, when you when you think that your model relies only on very few features. Again, this is not necessarily only for neural networks, but any other machine learning model.

(Refer Slide Time: 12:53)



CNN Training- Addressing Covariate Shift

- Batch Norm
- Layer Norm

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

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$    // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$    // mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

Batch Normalizing Transform, applied to activation x over a mini-batch.

arXiv:1502.03167v3

$\hat{y} = w_1 x_1 + w_2 x_2 \rightarrow m$

$(0-1) \quad (10^5-10^7)$

$\hat{\mu}_{LN} = \frac{1}{n}\sum_{i=1}^{n} x_i$

So, in this video, we will look at batch normalization, which is like a very popular technique. I say it is there now in all deep neural networks, and supposedly very effective; because it helps to accelerate the convergence of deep neural networks during training. So, the paper that came up with this technique called rotate to address the so called covariant shift, we will; but, later works have shown that maybe the that is not the correct explanation. So, we will we will just talk about it briefly and also see what exactly the computations are for batch normalization. And of course, there is a related normalization technique called layer nom; so we will look at that also.

So, when we think of this neural networks, let us look at the inputs to a neural network. So, you can first consider even fully connected, feed forward neural networks multi layer perceptrons, for instance, where inputs are a bunch of features. So typically, what we would do is, would be to make sure that we pre-process the data, and that has a huge impact on the results. So, we would typically pre-process the data to have zero mean and unit variance across multiple observations. So, we will we will take our entire time data, and subtracted to subtract the mean from it; and divide by the standard deviation to make it zero mean and unit variance.

So, this is one of the pre-processing steps that are usually done. The idea behind doing that is that, if we consider your loss function as a function of the weights; so you have these features. And you do not you do not can not have these features; these these pre-processing techniques are helped with optimization; because it puts all the parameters on the same scale.

So, very simple example, usually, if you want to think about it; let say you have two inputs to my neural network. I have two inputs; call them x_1 and x_2. And there is a bunch of weights connecting the let us let us we can just say I have a very; in fact from a very linear regression point of w_1 and w_2. Or, the weights multiplying them and I have an output they provide a $\hat{y}$ as an output; which is basically $\hat{y}$ is w_1x_1 plus, let me rewrite this is gone out of scale.

So, I will rewrite output $\hat{y} = w_1x_1 + w_2x_2$. Now, why do we want the inputs x_1 and x_2 to have zero mean and unit standard deviation? That is the standardization so that they all have a similar scale, that is what we want to do is that it helps in better optimization. Because, if x_1 occupies range of 0 to 10, and x_2 occupies the range of 10^5 to 10^7, which would mean that, we would we would have to adjust w_1, w_2 differently. Because, w_1 should have be, the w_1 has to be so high in order to bring this sum of products to comparable to w_2x_2; and in that case w_2x_2 has been very low.

So, which means that when we, when we take steps in the gradient descent direction, the step size would have been very different for w_1 and w_2, which makes the optimization much slower. In order to prevent that you would typically scale x_1 and x_2, a very rails of scale, ways of scaling it, one is to make sure they have zero mean and unit standard deviation.

Of course, this is across the training data set. So, there are many many training examples you have; let say M-training examples. So, then you would average across the M-trainee examples, and then find the standard deviation across the M-training example M-training examples, and do the appropriate normalization. So, this helps in the helps optimizer, the optimization process.

So, but if you think of a neural network, so this is just I am I am, I have done this for a very simple linear regression kind of model. If you consider a deep neural network, there are multiple layers. And during the initial stages of the training as the day as the data comes at the input layer, and then you have the multiplication with the weight matrix non-linearity forms of a certain layer, and then and so on and so forth.

So, initially when the weights are not trained, then when you even though in the input layer, you make them; let say zero mean and unit standard deviation as it goes into the various layers, that standardization is lost. And it would be nice if we can arrest this drift in the distribution, if you can, if you can call it; because as we go, the input has been now we will start off with the value values zero mean; and unit standard deviation have now gotten strange because of multiplication, as well as application of nonlinearities through the layers.

So, the paper suggested this technique to maybe push to put it back into the same standardization that we send the inputs. The way to do that is the following is given in this flowchart right here, where you have the x is your input x is a vector; and m, the mini-batch there are examples of that vector. And we actually learn the parameters in order to standardize them. So, how do we learn them?

The way to that is to calculate the mini-batch mean; that is what it does, mean of the x the input vectors. Of course, when I say mini-batch, mean this is the input to that particular layer; so that is the thing. The x is the input to any of the layers that you are considering. You need not necessarily be the input layer, but some layer in between also; so you calculate the mean of the inputs. You calculate the standard deviation of the inputs, and then you do your usual standardization, the $\frac{x-\bar{x}}{\sigma}$. But, what you do is you rescale them using a learned γ and β.

So, the scale and shift parameters, γ and β, you you will learn; and because these are free parameters that are learned by the network. So, for every new, so I you would learn this as a layer. So for every neuron layer, you will learn some parameter like this; that is one implementation, correct.

So, just to reiterate, the idea is you would calculate the mean and standard deviation of the input to every layer over the mini-batch, for every neuron or every hidden unit. And you would scale your inputs like we did for for the actual input of the neural network. In this case, you are making it, you are normalizing it, so that its zero mean and unit standard deviation. And you will scale it by using a learned γ and β.

So this process, you can this is actually like a layer that you can drop in wherever you want to inside a deep neural network; and train because you will be learning γ and β as parameters also. So, batch normalization has shown to be very useful in accelerating the training for very deep networks trained much faster, because that has been proven.

Now, the way batch norm is supplied during training is different for different types of networks. So for instance, for fully connected layers, you would in the original paper of the batch normalization is applied after the linear combination of the activations from the previous layer. And following which you apply a nonlinear activation; because people are experimented with it half way applying the batch normalization after the activation. So, that is that is also there; so there is some debate about which is more effective, so you can do either way.

The another thing to consider is that, when you do the batch norm in a fully connected network, recall that we do this mean of the inputs to the each layer, across the mini-batch; so every hidden unit or every hidden unit or neuron, in a layer is treated separately. You do not average across the hidden unit simply now.

When you apply this the convolutional neural network, once again we can either apply it after the convolution and before the nonlinear activation. This we apply this operation is done on a per channel basis across all locations. So, which means that the key which means similar, but if you want to do something similar in a fully connected network; it would mean that we are averaging across the hidden units in a layer, which we do not do actually. But, for convolutional neural networks, when you do this normalization, it is normalization per channel.

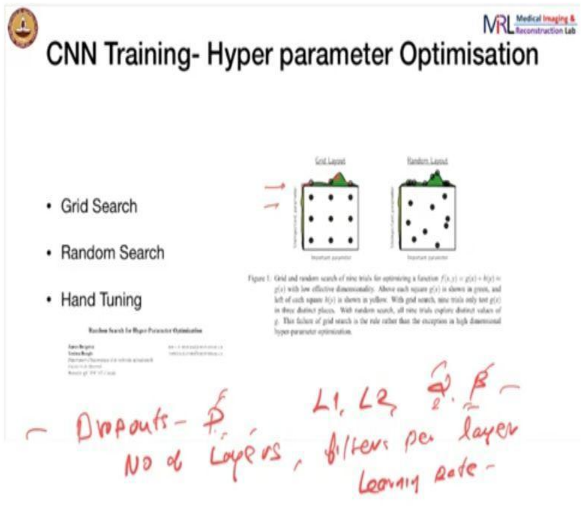
When you do it for a fully connected feed forward ml the neural network, the normalization is per hidden unit or per neuron. So, just to reiterate for the convolutional neural networks, we would carry out the batch normalization per channel. And now we would consider the mini-batch per channel. Consider mini batch per channel to calculate the mean and standard deviation, and, and then normalize the inputs to every layer. And each channel will have its own scale and shift parameters, and they are scalars. So, like but for a fully connected, we saw that every hidden unit will have its own scale and shift parameters that are estimated during back propagation.

Now that we know this, we can now look at layer norm; layer norm is very similar. And except that we just do it for one observation at a time, especially for CNN, it is very well defined; because every channel has its feature map and we have we can average specially. So, the formulas remain the same, except that the mu that you estimate here; the mu that you estimate here is over the dimensionality of your data.

So, I will quickly write this down here, your μ layer norm; I will call this layer norm is one over. In this case, I will use n, where n is the number of; the number n number of dimensions. So, every x is n dimensional vector; so you will just average over your vector dimensions. Similarly, you have a formula corresponding formula for the standard deviation variance. So, the layer normalization does not depend on mini-batch size; and we can do it for both train and test, and it actually just some deterministic transformation.

So, and basically, it is also helps in optimization; there are more details given the publication just search for layer norm, and you will be able to find it. During prediction, that is when you are testing with new data, then we might not have the luxury of running through every mini-batch statistics that we calculated, in order to make a prediction. But, rather the statistics is estimated, that is the statistics for the 'μ' B and 'σ' B that we see here; that is one that we need. Before we make the predictions, we it is usually gathered over the entire data set when the predictions are made.

(Refer Slide Time: 26:00)



CNN Training- Hyper parameter Optimisation

- Grid Search
- Random Search
- Hand Tuning

Random Search for Hyper-Parameter Optimization

[illegible]

Grid Layout
Random Layout
Important parameter
Important parameter

- Dropouts - P
$L1, L2$ α. β -
No of Layers, filters per layer
Learning Rate -

So, we are now going to look at this hyper parameter optimization which is again, a very key concept to understand when you are trying to train deep neural networks; or for that matter, any kind of machine learning based algorithm. So, as we saw in the previous few videos, there are several hyper parameters involved in training CNNs. So for instance, we looked at dropout.

So, when you say dropouts, for instance, say dropouts, there is this parameter P that you have to figure out. What is 0.25, 0.5, 0.1 etcetera? We also looked at batch normalization. So, batch norm, once again we have to decide, you have to decide how many such batch norm layers do you need, do you need that in every layer. We need that only in the earlier part of the network or in later part of the network, so on and so forth; so that is a decision you have to make.

There is we looked at this L_1, L_2 regularizers which are pre multiplied by this α and β; these are again scalars that we have to estimate. What is the correct value for these scalars? Do we need both L_1 and L_2? That is one other problem; that is one other hyper parameter. So, these are all hyper parameters. In addition to whether you know number of layers in a neural network, how many filters per layer? And we did not go through gradient descent algorithm in detail. But, a decent algorithm has a parameter called the learning rate which is again hyper parameter.

So, how do we figure out the correct values of these things? Because these are not learned during the training of neural network; so, I learned this is something you have to fix. So, the way to look at it is to put them in a grid. So, there is only a range of values that they can take; you would put them in a grid, multi-dimensional grid and perform the grid search. That would be the best way to do.

So, grid search is where you systematically step through the values of each of these parameters; and you would estimate a curve. What is this curve is shown by this green bound here? This curve is some metric that you would expect; for instance, the loss function itself. For a certain value of of these α , β and P, what is the best loss function you get? Or what is the series of loss functions you get for each combination of this alphabet?

And you would take the combination that would be the; that would give you the least or the, or is in this case the least loss function. So, loss function is one metric, you can do other metrics also. So, grid search is that you would systematically step through each one of these parameters. So, but the problem with this grid grid layout is actually shown in this picture is that, maybe your optimum was right here.

But, when you do a systematic research, you might actually miss out on it; and so you will not be able to sample this error surface as a properly. So, because why you recall the error surface? Because if you think of this hyper parameter, let say even take three of them the dropout with a probability P, α and β. These are the three parameters that you are trying to optimize the hyper parameters that you want to estimate. Then, you can with on a small portion of your training data, estimate the loss function as a function of these three parameters. And that is what I call the error surface.

So, if you do not sample these properly, which you which happens when you do this systematic grid search, by sampling these parameters at regular intervals, and running your training at all combinations of such parameters; then it is a good chance that you might miss out on the optimum.

Other hand, if you do a random search of your grid, like you do have these parameters ranges of these parameters; but then you pick them randomly. You might the good possibility that you would be able to come near the optimum value. So, typically you do it recur in a kind of a nested fashion, where you would do a very of course search, random search; and, you a random search, random basically random sampling.

And then, maybe when you are near an optimum value, you would then grid and then do a small systematic search. So, all of these are possible, of course, the least recommended is the hand tuning, wherein you just randomly try a few parameters; and you are happy with the results. And mostly those turn turn out to be either overfitting and underfitting; so that is a bad practice. So, random search systematically on all the hyper parameters would give the much better results.