ECE 1508S2: Applied Deep Learning

Chapter 3: Advancing Our Toolbox

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2025

Tuning Hyperparameters

A problem that we intuitively discussed but *left open in Chapter 1* was

How can we tune the hyperparameters of a model?

In this section, we are going to *get the genie out of the bottle!*

- + *Shouldn't we set hyperparameters as much as we could?! Make NNs as deep and wide as our computer let?*
- – Not really! We don't need very deep and wide NNs always
- + *But, what if we are computationally strong?! Then, we are fine! Right?!*
- – No! In fact with large NNs we can do the so-called *overfitting!*

Let's see a very simple example!

Example: *Fitting Polynomial from Noisy Data*

We start by a *classical example which is not that of NN we expect*

We have a machine which gets real-valued x and returns

$$y = x^2 + 3x + 3$$

We however don't know this relation: the only thing that we know is that the inputs and labels are related via a polynomial

We invoke ML to learn this machine

Let's start with making the ML components, i.e.,

1. Dataset
2. Model
3. Loss

Example: *Polynomial Fitting - Dataset*

We start by collecting data: *we give input x_b to this machine and measure its output for a batch of inputs. Our measurements are however noisy, i.e.,*

$$v_b = {x_b}^2 + 3x_b + 3 + \varepsilon_b$$

where ε_b is noise with bounded magnitude, i.e., $|\varepsilon_b| \leqslant \alpha$ for some constant α

We make our dataset as

$$\mathbb{D} = \{(x_b, v_b) : i = 1, \ldots, B\}$$

Example: *Polynomial Fitting - Model*

We know that machine is polynomial: *we assume a polynomial model*

$$y = w_0 + w_1 x + w_2 x^2 + \dots + w_P x^P$$

for some integer order P

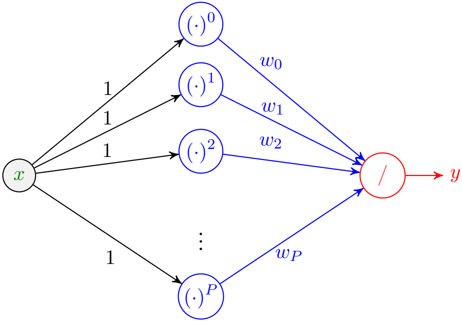
We can write it down as

$$\begin{aligned} y &= w_0 + w_1 x + w_2 x^2 + \dots + w_P x^P \\ &= \underbrace{\begin{bmatrix} w_0 & w_1 & w_2 & \dots & w_P \end{bmatrix}}_{\mathbf{w}^\mathsf{T}} \underbrace{\begin{bmatrix} x^0 \\ x^1 \\ x^2 \\ \dots \\ x^P \end{bmatrix}}_{\mathbf{h}} = \mathbf{w}^\mathsf{T} \mathbf{h} \end{aligned}$$

Example: *Polynomial Fitting - Model*

We can look at our model as *an NN with dummy neurons*

$\mathbf{h}$ is what we get from hidden layer and $\mathbf{w}$ includes weights of output layer



The key hyperparameter in this network is P

Example: *Polynomial Fitting - Loss*

We have a regression problem: *recall that*

in regression, the labels are real-valued

We use *squared error as the loss function, i.e.,*

$$\mathcal{L}\left(y, v\right) = \left(y - v\right)^2$$

for measurement v and NN's output y

Now, the components are ready

let's start training

Example: *Polynomial Fitting - Training*

For training, we follow what *we already learned in previous lectures*

1. *We split $\mathbb{D}$ into a training dataset and test dataset*
2. *We start use gradient descent to train over the training dataset*
3. *We test our trained model over the test dataset*

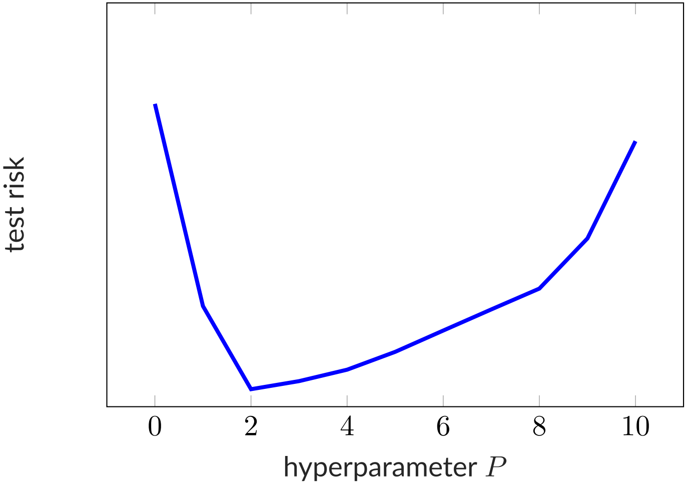
Before we go on with training, let's take a look back

> *Naive conclusion was that making the NN large* is *always good; if so*
>
> *when we increase P, we should always see lower test risk*
>
> *However, it's* not *the case!*

Let's see *how the test risk changes against hyperparameter P*

Example: *Polynomial Fitting*

Test risk against the hyperparameter P looks like the curve below!



Over and Underfitted Model

+ *What is happening here?*
- As we pass $P = 2$, we are overfitting!

Overfitting

Overfitting occurs when training fits the model, i.e., NN, to the training dataset, so that it does not generalize to new data-points

We may also pay attention to the term *generalize* in this definition

Generalization

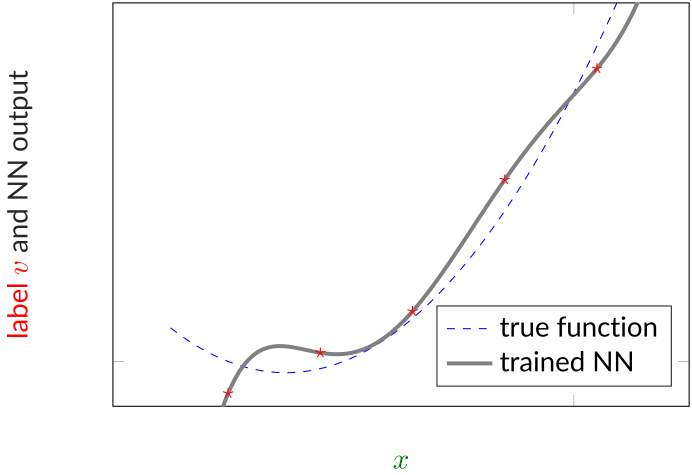
We say a model, i.e., NN, generalizes well if not only its training risk, but also its test risk is *small*

In simple words:

trained NN generalizes $\equiv$ *it does what we want on new data*

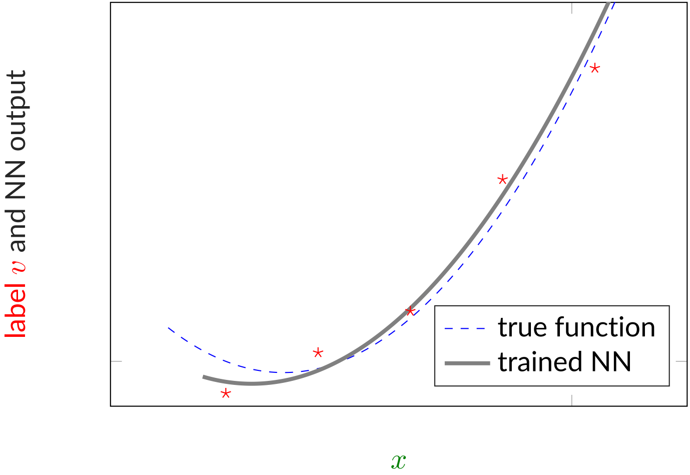
Overfitting: *Polynomial Fitting*

Let's take a look back on polynomial fitting example: *for large P the NN fits very well to the training data, but it deviates greatly from the true function*



Overfitting: *Polynomial Fitting*

We can see the importance of hyperparameter tuning: *if we set P to a right choice; then, our NN generalizes well, i.e., it closely track the true function*



Underfitting

The other side of the coin is *underfitting*: *it happens when our NN does not have enough parameters to train*

Underfitting

Underfitting occurs when the model, i.e., NN, neither fits to the training dataset, nor generalizes to new data

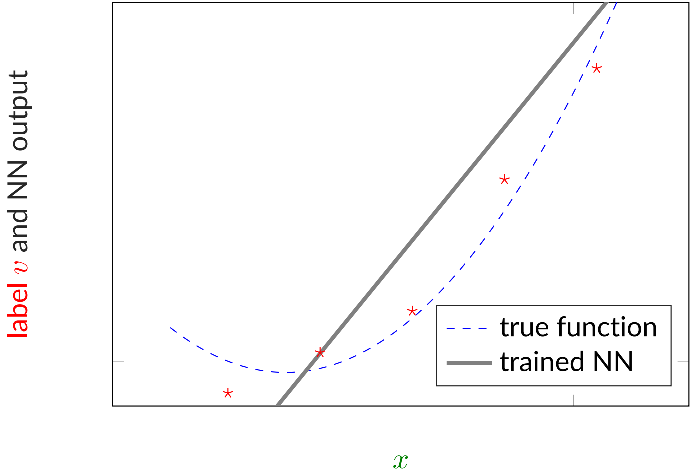
Se would also need to prevent underfitting; however,

with current NNs, underfitting can hardly occur

This is why *it's less discussed in the literature*

Underfitting: *Polynomial Fitting*

A linear model underfits in our example: *setting $P = 1$ will lead to a line that can never fit our training dataset*



Validation: *First Step Against Overfitting*

+ *What is connection to our main task, i.e., hyperparameter tuning?*
– Well! Before everything, we need to tune the hyperparameters right to avoid over or underfitting

---

Hyperparameter tuning is done by *validation*: *we change hyperparameters among possible choices and for each choice, we validate our model*

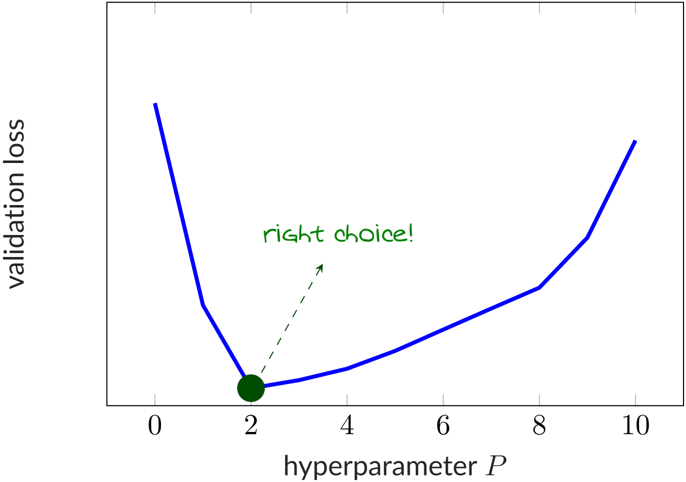
In validation we train the NN with the specified hyperparameters and then test it on a validation set separated from training and test sets

We set hyperparameters to the choice that

$$\textit{gives minimal validation risk} \equiv \textit{generalizes the best}$$

Validation: *Polynomial Fitting*

In fact what we did in our dummy example was validation



Why Overfitting Happens

+ *But can we really do hyperparameter tuning in a very deep NN?*
- Not really! We may tune some general hyperparameters like number of layers, but *cannot really do a complete validation*

In NNs, we invoke other approaches as well to combat overfitting

- *Regularization*
- *Dropout*
- *Data Augmentation*
- . . .

To understand these approaches, we should first answer the following question

When does overfitting happen in a NN?

Let's take a look!

Why Overfitting Happens: *Model Capacity*

We know the initial answer: *in our dummy example, it happened because*

we assumed large polynomial order

In other words

our model was too complex for our learning task

We can extend this to NNs: *overfitting happens when the model is too complex, i.e., it's suited for learning complicated functions*

When does overfitting happen in a NN? *It happens when*

1. *the model has a large capacity*

Though *model capacity* has a concrete definition, for our purpose

model capacity $\equiv$ *ability of model to learn different functions*

Why Overfitting Happens: *Dataset Size*

- + *But how can we find this out? It does not seem to be easy!*
- – That's right! This is why we look into *other reasons as well*

Let's get back to our dummy example: *this time we check it a bit differently*

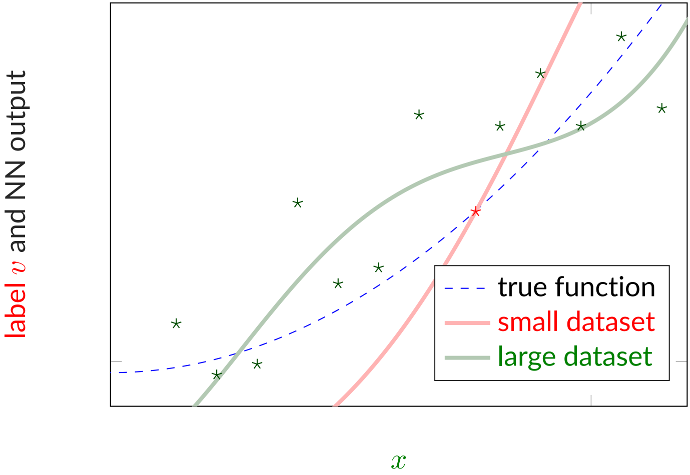
In our polynomial example, we consider an overfitted model with $P = 5$ and train it on two randomly generated datasets

1. *a dataset with 20 data-points*
2. *a dataset with only 4 data-points*

After training: *we compare trained models with the true function*

Why Overfitting Happens: *Dataset Size*

As we can see: *overfitting is reduced as we increase the number of data-points*



Why Overfitting Happens: *Dataset Size*

This is a general behavior: *if we have a large enough dataset the model cannot really overfits too much*

+ *How large it should be?*
- It depends on the NN

A general rule is that *the more learnable parameters the model has, the larger the training dataset should be*

So, we can add to our answers

When does overfitting happen in a NN? *It happens when*

1. *the model has a large capacity*
2. *our training dataset is small*

Why Overfitting Happens: *Co-Adaptation*

Another way to see overfitting is to look at *how model parameters change as optimizer iterates*. To see it, *let's get back to our dummy polynomial-fitting NN*

Consider the following setting: *we have a high-capacity NN with $P = 5$ and dataset with 8 noisy samples. We train this NN using full-batch SGD*

We now take a look at the trained NN at different iterations: *recall that the vector of model parameters is*

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_5 \end{bmatrix}$$

We start with vector of all zeros and keep on going

Why Overfitting Happens: *Co-Adaptation*

Recall that *the ground truth* $\mathbf{w}^\star$ for our polynomial machine

$$\mathbf{w}^\star = \begin{bmatrix} 3 \\ 3 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, let's look at few iterations

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_1)} = \begin{bmatrix} 2.61 \\ 2.36 \\ 0.71 \\ 0.01 \\ 0.02 \\ 0.01 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix}$$

$\mathbf{w}^{(t_2)}$ *looks good! But, what if we keep on training*

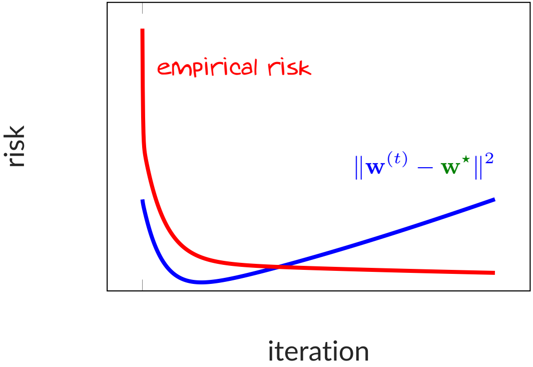
$$\mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix} \rightsquigarrow \cdots \rightsquigarrow \mathbf{w}^{(t_n)} = \begin{bmatrix} 2.36 \\ 4.43 \\ 3.13 \\ -2.1 \\ 1.98 \\ -1.2 \end{bmatrix}$$

Why Overfitting Happens: *Co-Adaptation*

Let's formulate what we observed

> *Weights start to get close to what we want up to intermediate number of iterations t_2. But, by further training they start to deviate $\equiv$ overfit*

We can also see this behavior in the figure below



Why Overfitting Happens: *Co-Adaptation*

This behavior is co-adaptation of the parameters

In initial iterations, NN fits to the true model: *since data comes from a quadratic function, the first iterations of SGD*

update majorly w_0, w_1 and w_2

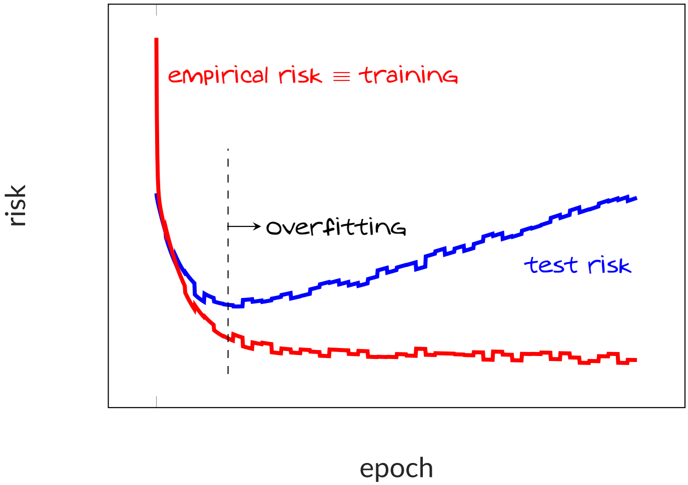
After NN has gone close to ground truth, it starts to overfit: *due to noise, quadratic model can't perfectly fit; thus,*

w_3, w_4 *and* w_5 *try to co-adapt, i.e., compensate the gap caused by noise*

In simple words: *in first iterations NN learns true function; however, at some point it starts to learn noise!*

Co-adaptation is the most observable implication of overfitting with NNs

Co-Adaptation: *Typical Learning Curve*



Why Overfitting Happens: *Final List*

So, let's complete the answer list

When does overfitting happen in NNs? *It happens when*

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

Attention: *sources are mutually related*

If we have a very large model capacity, i.e., very deep with too many neurons; then, training it by a small dataset leads to overfitting, especially if we keep on training for too many epochs!

+ *Now, can we do anything to avoid overfitting?*

– Yes! Depending on what we see as source, we use different tricks

Classical Solutions to Overfitting

Overfitting happens when

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

The key tricks to address overfitting in each of these cases are

Done
1. *We can tune the hyperparameters to restrict the NN's capacity*
   ↳ *For instance, we can validate our FNN with 2, 3 and 4 hidden layers and choose the model with minimal validation risk*

Wait
2. *We can increase our dataset by the so-called data augmentation*
   ↳ *For instance, we can add rotated and shifted versions of images inside the dataset with the same label: a rotated image of a dog is still a dog!*

Next
3. *We can regularize our empirical risk to penalize co-adapted solutions*
   ↳ *For instance, we can drop out randomly some neurons in each mini-batch*

Training by Penalized Risk: *Regularization*

Regularization aims to resolve overfitting by treating co-adaptation

Let's recall co-adaptation in our dummy polynomial fitting NN: *we set $P = 5$ and train our NN via the noisy samples inside training dataset; clearly,*

as training proceeds, empirical risk drops

In our particular example with

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_n)} = \begin{bmatrix} 2.36 \\ 4.43 \\ 3.13 \\ -2.1 \\ 1.98 \\ -1.2 \end{bmatrix}$$

This means that $\hat{R}(\mathbf{w}^{(t_n)}) \leqslant \hat{R}(\mathbf{w}^{(t_2)}) \leqslant \hat{R}(\mathbf{w}^{(0)})$

Training by Penalized Risk: *Regularization*

Regularization follows this idea: *can we modify empirical risk, such that it stops dropping after t_2?*

Let's continue with our example: *assume risk's value at each $\mathbf{w}$ is*

$$\hat{R}(\mathbf{w}^{(t_3)}) = 0.001 \qquad \hat{R}(\mathbf{w}^{(t_2)}) = 0.01 \qquad \hat{R}(\mathbf{w}^{(0)}) = 100$$

We may note that as training progresses, vector $\mathbf{w}^{(t)}$ becomes larger. So, what if we add a penalty to risk that is proportional to $\|\mathbf{w}^{(t)}\|^2$: this way when risk becomes too small, this penalty becomes large and thus the sum increases. Let's look at this sum at *different iterations*

$$\begin{aligned}\tilde{R}(\mathbf{w}^{(0)}) &= \hat{R}(\mathbf{w}^{(0)}) + \|\mathbf{w}^{(0)}\|^2 = 100\\ \tilde{R}(\mathbf{w}^{(t_2)}) &= \hat{R}(\mathbf{w}^{(t_2)}) + \|\mathbf{w}^{(t_2)}\|^2 = 19.04\\ \tilde{R}(\mathbf{w}^{(t_n)}) &= \hat{R}(\mathbf{w}^{(t_n)}) + \|\mathbf{w}^{(t_n)}\|^2 = 44.761\end{aligned}$$

Training by Penalized Risk: *Regularization*

Penalized risk shows a different behavior

$$\tilde{R}(\mathbf{w}^{(0)}) = 100 \qquad \tilde{R}(\mathbf{w}^{(t_2)}) = 19.04 \qquad \tilde{R}(\mathbf{w}^{(t_n)}) = 44.761$$

From above values, we can say: *if we apply SGD to minimize penalized risk we may get from $\mathbf{w}^{(0)}$ to $\mathbf{w}^{(t_2)}$; however, we will not get from $\mathbf{w}^{(t_2)}$ to $\mathbf{w}^{(t_n)}$*

This idea is called regularization which can prevent NNs from overfitting

Regularization

In training with regularization, we minimize a penalized (regularized) form of the empirical risk, i.e.,

$$\min_{\mathbf{w}} \hat{R}\left(\mathbf{w}\right) + \Pi\left(\mathbf{w}\right) \qquad \text{(Regularized Training)}$$

$\Pi\left(\mathbf{w}\right)$ *is a penalty that describes the behavior of* $\mathbf{w}$ *in the case of overfitting*

Classical Regularization Approaches

There are various regularization penalties: *some important ones are*

- ℓ_2 or *Tikhonov regularization in which we add a term proportional to* $\|\mathbf{w}\|^2$

$$\Pi\left(\mathbf{w}\right) = \lambda \|\mathbf{w}\|^2$$

↳ *This way we avoid very large weights*

↳ *This prevents perfect fit to training dataset reducing chance of overfitting*

- ℓ_1 or *Lasso regularization in which we add a term proportional to* $\|\mathbf{w}\|_1$

$$\Pi\left(\mathbf{w}\right) = \lambda \|\mathbf{w}\|_1 = \lambda \sum_{i=1}^{D} |w_i|$$

↳ *This way enforce* $\mathbf{w}$ *to be sparse, i.e., to have to many zeros*

↳ *This way we reduce the capacity of NN and thus prevent overfitting*

Regularizing by *Dropout*

A less conventional regularization approach *is dropout* that was proposed by *Hinton et al.* first in *their 2012 paper* and then in *their 2014 paper*:[1] *the idea is at the same time easy and effective*

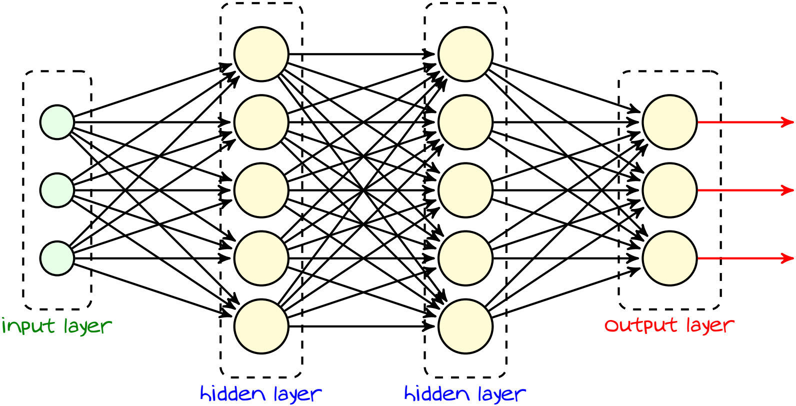
> *for each training iteration, we deactivate some nodes of NN at random*

or in other words we drop them out

[1] Click to check out the papers!

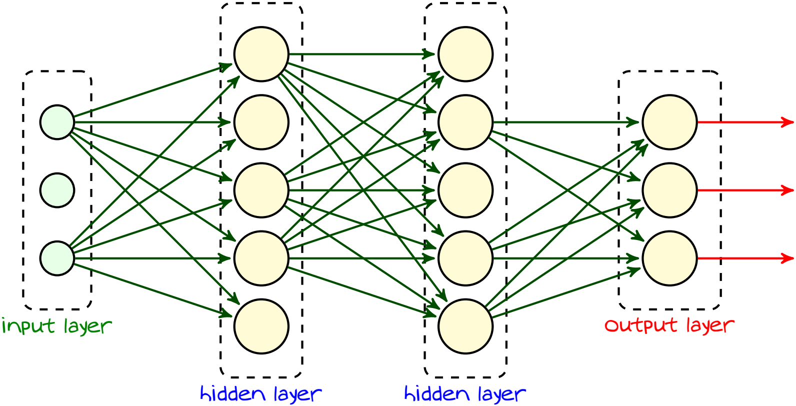
Dropout: *Schematic*

Let's say this is the dense NN



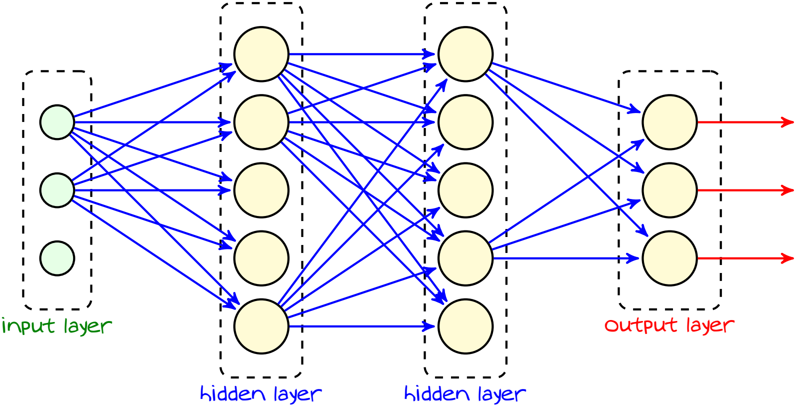
Dropout: *Schematic*

For first forward-backward we select few nodes in each layer



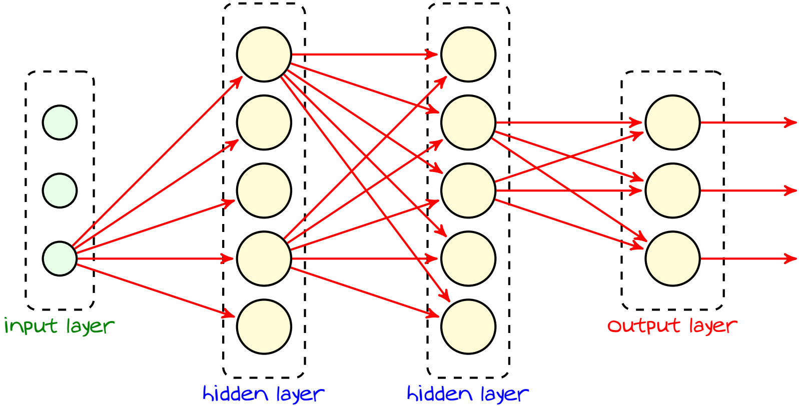
Dropout: *Schematic*

For second forward-backward we select new nodes in each layer at random



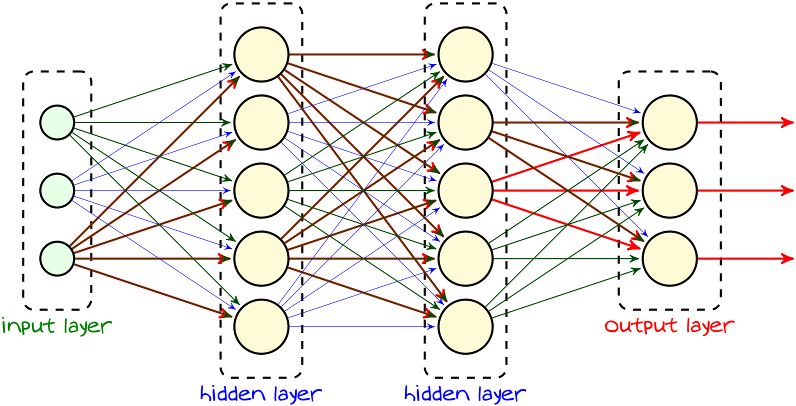
Dropout: *Schematic*

For next forward-backward we select again few nodes in each layer at random



Dropout: *Schematic*

At the end, we average the gradients determined over these reduced NNs



Dropout: *Intuition*

+ *But, why does dropout work?*
– We can explain it *heuristically*

Recall that *observing overfitting means that NN is larger than required*. With dropout, *in each iteration we train a smaller version of NN*

- *We randomly switch among these smaller versions*
- *Many of these smaller versions do not overfit*

We can look at the training loop with dropout as an averaged training of these smaller NNs; hence, training loop gets less chances to overfit

+ *Why do we do it randomly? Why not sticking to one smaller version?*
– Not all smaller NNs are good, and we cannot check all of them: *it's exponentially hard to check all smaller NNs*

Dropout: *Training Loop*

How does training change with dropout? *Training with dropout is exactly as before. Say we use mini-batches: for each mini-batch*

- *we compute the gradient by forward and backpropagation*
- *we give the gradient to the optimizer to apply the next iteration*

The only thing that is different now is that we set the output of some nodes to zero in the forward pass $\equiv$ *only forward propagation changes*

Let's make it concrete: *when we pass forward, we generate random masks for each layer* $\ell = 0, \ldots, L$. *Mask of layer* ℓ *is a vector whose length is layer's width and entries are 0 or 1*, i.e., $\mathbf{s}_\ell \in \{0,1\}^{\mathcal{W}_\ell}$. *Entries of* $\mathbf{s}_\ell$ *are generated randomly*

$$\textit{each entry of } \mathbf{s}_\ell = \begin{cases} 1 & \text{with probability } p_\ell \\ 0 & \text{with probability } 1 - p_\ell \equiv \textit{dropout probability} \end{cases}$$

Dropout: *Forward Propagation*

Let's show generation of random mask $\mathbf{s}_\ell$ by following notation

$$\mathbf{s}_\ell = \mathtt{mask}\left(\mathcal{W}_\ell | p_\ell\right)$$

We are going to do forward propagation for each data-point as

```
DropoutForwardProp():
Initiate with $\mathbf{y}_0 = \mathbf{x}$
for $\ell = 0, \ldots, L$ do
    Generate $\mathbf{s}_\ell = \mathtt{mask}\left(\mathcal{W}_\ell | p_\ell\right)$          # random mask
    Set $\mathbf{y}_\ell = \mathbf{y}_\ell \odot \mathbf{s}_\ell$                              # dropout nodes
    Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$    # forward affine
    Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$                      # forward activation
end for
for $\ell = 1, \ldots, L+1$ do
    Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
end for
```

Dropout: *Backpropagation*

The backpropagation goes exactly as before: *of course those outputs that were dropped out participate with value zero in gradient computation*

One final piece of trick

After training is over, we scale weights of each layer with its retain probability p_ℓ: say T is the last iteration of training loop; then, we finally do

$$\mathbf{W}_\ell^{(T)} \leftarrow p_\ell \mathbf{W}_\ell^{(T)}$$

+ *Why do we do that?*
- Well! It's practically understood; however, we can justify it as follows: *each weight could be what has been computed with probability p_ℓ and zero with probability $1 - p_\ell$. We hence compute the average*

Dropout: *Implementation*

Dropout is implemented in almost all deep learning libraries

```
>> import torch
>> torch.nn.Dropout()
```

Typical choices of retain probability p_ℓ are

- *for input layer*, i.e., *layer 0*, $p_\ell = 0.8$
- *for hidden layers* $p_\ell = 0.5$

It's generally suggested to drop out more at hidden layers

With dropout forward pass changes in training and evaluation

- *In training we use random mask* $\rightsquigarrow$ `model.train()`
- *In evaluation we don't use random mask* $\rightsquigarrow$ `model.eval()`